Kompatibel zu Java 8

Joachim Goll
Cornelia Heinisch

Java als erste Programmiersprache

Grundkurs für Hochschulen

8. Auflage

Springer Vieweg

Java als erste Programmiersprache

Joachim Goll • Cornelia Heinisch

Java als erste Programmiersprache

Grundkurs für Hochschulen

8., überarbeitete Auflage

Springer Vieweg

Joachim Goll
STZ Softwaretechnik
Esslingen, Deutschland

Cornelia Heinisch
FOM Hochschule
Stuttgart, Deutschland

ISBN 978-3-658-12117-4 ISBN 978-3-658-12118-1 (eBook)
DOI 10.1007/978-3-658-12118-1

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 2000, 2001, 2002, 2005, 2007, 2011, 2014, 2016
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 GmbH ist Teil der Fachverlagsgruppe Springer Science+Business Media (www.springer.com)

Vorwort

Die Sprache Java ist durch ihre Betriebssystemunabhängigkeit ideal für die Realisierung verteilter Systeme, die aus verschiedenartigsten Rechnern vom Handy bis zum Großrechner aufgebaut sein können. Java wird heute bereits im Informatik-Unterricht an den Gymnasien unterrichtet und ist fester Bestandteil des Studiums von Ingenieuren und Betriebswirten geworden.

Java stellt im Grunde genommen eine einfache Sprache dar. Darüber hinaus werden jedoch in umfangreichen Klassenbibliotheken wertvolle und weitreichende Hilfsmittel zur Verfügung gestellt, die den Bau verteilter Systeme mit Parallelität, Oberflächen, Kommunikationsprotokollen und Datenbanken in erheblichem Maße unterstützen.

Dieses Buch konzentriert sich auf die Sprache und nicht auf den Bau verteilter Systeme und die dafür vorhandenen nützlichen Bibliotheken. Es wendet sich an Studierende, welche das Interesse haben, die Grundlagen von Java fundiert zu erfahren. Es erlaubt, Java ohne Vorkenntnisse anderer Programmiersprachen zu erlernen. Daher lautet der Titel "Java als erste Programmiersprache". Ein Neuling sollte aber unbedingt die Übungsaufgaben am Ende eines Kapitels bearbeiten. Wer hingegen das Buch nur lesen möchte, sollte jedoch bereits über die Kenntnisse einer anderen Programmiersprache verfügen.

Dieses Buch hat das ehrgeizige Ziel, dem Neuling die Sprachkonzepte von Java und die Grundkonzepte der objektorientierten Programmierung so präzise wie möglich und dennoch in leicht verständlicher Weise vorzustellen. Dabei werden Fortgeschrittenen-Themen als solche explizit ausgewiesen, damit ein Anfänger sich zunächst auf die Grundlagen von Java konzentrieren kann. Ausgehend von dieser Basis können darauf aufbauend die fortgeschrittenen Themen erschlossen werden.

Alle Kapitel enthalten Übungsaufgaben, die zum selbstständigen Programmieren herausfordern. Selbst zu modellieren und zu programmieren ist das Ziel des Flughafen-Projekts, welches begleitend zu den Kapiteln 10 bis einschließlich 18 durchgeführt werden kann und zu einem System führt, das die Fluglotsen bei Start und Landung der Flugzeuge unterstützt.

Auf viele Details von Java kann wegen des Umfangs des Buchs leider nicht eingegangen werden. Die zentralen Themen der Sprache sind jedoch im Buch erfasst. Auf die Java Language Specification unter der Internet-Adresse

`http://docs.oracle.com/javase/specs/`

wird verwiesen.

Es soll hier nicht unerwähnt bleiben, dass die Java-Klassenbibliothek über 4000 Klassen enthält. Hierzu treten noch zahlreiche kommerzielle bzw. offene Bibliotheken beispielsweise für einen komfortablen Zugriff auf Datenbanken oder das Einlesen und Erzeugen von `xml`-Dateien.

Das vorliegende Buch ist kompatibel zu Java 8. Besonderheiten von Java 8 wie Lambda-Audrücke oder die neue Stream API sprengen jedoch den Rahmen eines Einsteiger-Buches und können hier nicht behandelt werden.

Unser besonderer Dank bei dieser Auflage gilt Herrn Markus Schuler für zahlreiche Diskussionen und Herrn Steffen Wahl sowie Herrn Antonio Pomarico für die professionelle Durchführung des Konfigurationsmanagements.

Esslingen, im Dezember 2015 J. Goll / C. Heinisch

Wegweiser durch das Buch

"Lernkästchen", auf die grafisch durch eine kleine Glühlampe aufmerksam gemacht wird, weisen auf für das Verständnis wichtige Aspekte eines Kapitels hin. Die Zusammenfassung eines jeden Kapitels erlaubt dem Leser eine rasche Wiederholung des Stoffes.

"Warnkästchen" hingegen weisen den Leser zum einen auf gerne begangene Fehler hin, zum anderen warnen sie den Leser vor einer Fehlinterpretation des Textes an "gefährlichen" Stellen des Buchs.

Gerade ein Anfänger in einer Programmiersprache macht gerne den Fehler, sich beim Lesen an nicht ganz so wesentlichen Einzelheiten festzubeißen. Damit ein Leser leichter erkennen kann, welche Information grundlegend für das weitere Vorankommen ist und welche Information nur ein Detailwissen darstellt – und deshalb auch noch zu einem späteren Zeitpunkt vertieft werden kann – gibt dieses Buch eine einfache Hilfestellung. Es weist Kapitel oder Kapitelteile, die beim ersten Lesen des Buchs übersprungen werden können, mit dem "Überspringe und komm zurück"-Symbol aus[1].

Generell ist es empfehlenswert, ein oder mehrere Kapitel zu überfliegen, um sich einen Überblick zu verschaffen und dann erst mit der Feinarbeit zu beginnen und gründlicher zu lesen. Dennoch gilt: Eine Vorgehensweise, die sich für den einen Leser als optimal erweist, muss noch lange nicht für alle Leser das Allheilmittel darstellen. Wenn Sie zu den Lesern gehören, die es gewohnt sind, von Anfang an möglichst gründlich zu lesen, um möglichst viel sofort zu verstehen, so sollten Sie zumindest darauf achten, dass Sie in den Kapiteln mit dem "Überspringe und komm zurück"-Symbol beim ersten Durchgang nicht allzu lange verweilen.

Bei all den guten Ratschlägen gilt: Programmieren hat man zu allen Zeiten durch Programmierversuche erlernt. "Do it yourself" heißt der Schlüssel zum Erfolg. So wie ein Kleinkind beim Erlernen der Muttersprache einfach zu sprechen versucht, so sollten auch Sie möglichst früh versuchen, in der Programmiersprache zu sprechen – das heißt, eigene Programme zu schreiben. Gestalten Sie den Lernvorgang abwechslungsreich – lesen Sie einen Teil und versuchen Sie, das Erlernte beim Programmieren gleich umzusetzen. Die Beispielprogramme des Buches können zum Ausprobieren auf der Internetseite des Verlags unter www.springer.com bezogen werden. Diese Internetseite enthält auch die Bilder der einzelnen Kapitel, die Übungsaufgaben und Lösungen sowie das Flughafenprojekt. Aber bitte versuchen Sie zuerst, die angebotenen Aufgaben alleine zu lösen.

[1] Je höher der Schwierigkeitsgrad der Texte, umso mehr dieser Symbole werden vergeben.

Die nachfolgende Tabelle soll es dem Leser erleichtern, einzuordnen, welche Kapitel zu den Grundlagen (Symbol ☑) zählen und auf jeden Fall verstanden werden sollten, sowie, welche Kapitel zuerst übersprungen werden können und dann bei Bedarf gelesen werden sollten (Symbol ◌). Hier die bereits erwähnte Tabelle:

1	Grundlagen der Programmierung	☑
2	Objektorientierte Konzepte	☑
3	Einführung in die Programmiersprache Java	☑
4	Einfache Beispielprogramme	☑
5	Lexikalische Konventionen	☑
6	Datentypen und Variablen	☑
7	Ausdrücke und Operatoren	☑
8	Kontrollstrukturen	☑
9	Blöcke und Methoden	☑
10	Klassen und Objekte	☑
11	Vererbung und Polymorphie	☑
12	Pakete	☑
13	Ausnahmebehandlung	☑
14	Schnittstellen	☑
15	Geschachtelte Klassen	◌
16	Threads	☑
17	Generizität	◌ ◌
18	Collections	◌
19	Ein-/ Ausgabe und Streams	◌

Anhang

A	Der ASCII-Zeichensatz
B	Gültigkeitsbereiche von Namen
C	Die Klasse `System`
D	`jar`-Dateien

Schreibweise

In diesem Buch sind der Quellcode und die Ein-/Ausgabe von ganzen Beispielprogrammen sowie einzelne Anweisungen und Ein-/Ausgaben in der Schriftart `Courier New` geschrieben. Dasselbe gilt für Programmteile wie Variablennamen, Methodennamen etc., die im normalen Text erwähnt werden. Ziffern werden in Einzelfällen ebenfalls in der Schriftart `Courier New` geschrieben, um sie vom Text deutlich abzugrenzen. Wichtige Begriffe im normalen Text sind fett gedruckt, um sie hervorzuheben.

Ihre Verbesserungsvorschläge und kritischen Hinweise, die wir gerne annehmen, erreichen uns via E-Mail:

Cornelia.Heinisch@it-designers.de

Inhaltsverzeichnis

1 Grundlagen der Programmierung ... 1
1.1 Das erste Java-Programm ... 2
1.2 Programme und ihre Abläufe ... 4
1.3 Entwurf mit Nassi-Shneiderman-Diagrammen ... 11
1.4 Zeichen ... 17
1.5 Variablen ... 19
1.6 Datentypen ... 21
1.7 Zusammenfassung ... 27
1.8 Übungen ... 29
2 Objektorientierte Konzepte ... 31
2.1 Modellierung mit Klassen und Objekten ... 32
2.2 Kapselung, Abstraktion und Information Hiding ... 43
2.3 Individuelle und geteilte Daten von Objekten in Java ... 45
2.4 Abstraktion und Brechung der Komplexität ... 46
2.5 Erstes Programmbeispiel mit Objekten ... 50
2.6 Flughafen-Projekt ... 53
2.7 Zusammenfassung ... 66
2.8 Übungen ... 68
3 Einführung in die Programmiersprache Java ... 69
3.1 Sprachkonzepte von Java ... 70
3.2 Eigenschaften von Java ... 72
3.3 Die Java-Plattform ... 73
3.4 Programmerzeugung und -ausführung ... 77
3.5 Das Java Development Kit ... 82
3.6 Begriffe im Java-Umfeld ... 86
3.7 Zusammenfassung ... 87
3.8 Übungen ... 88
4 Einfache Beispielprogramme ... 91
4.1 Lokale Variablen, Ausdrücke und Schleifen ... 92
4.2 Erzeugen von Objekten ... 97
4.3 Initialisierung von Objekten mit Konstruktoren ... 99
4.4 Schreiben von Instanzmethoden ... 102
4.5 Zusammengesetzte Objekte ... 107

4.6 Selbst definierte Untertypen durch Vererbung 111
4.7 Ein- und Ausgabe am Bildschirm 114
4.8 Zusammenfassung 120
4.9 Übungen 121
5 Lexikalische Konventionen 129
5.1 Zeichenvorrat von Java 130
5.2 Der Unicode 132
5.3 Lexikalische Einheiten 133
5.4 Zusammenfassung 151
5.5 Übungen 152
6 Datentypen und Variablen 155
6.1 Klassifikation der Datentypen von Java 156
6.2 Einfache Datentypen 158
6.3 Der Typ einer Klasse 160
6.4 Variablen 166
6.5 Array-Typ 184
6.6 Aufzählungstyp 196
6.7 Zeichenketten 205
6.8 Wandlung von Datentypen 218
6.9 Annotationen 229
6.10 Zusammenfassung 234
6.11 Übungen 236
7 Ausdrücke und Operatoren 245
7.1 Operatoren und Operanden 246
7.2 Ausdrücke und Anweisungen 248
7.3 Nebeneffekte 250
7.4 Auswertungsreihenfolge 251
7.5 L-Werte und R-Werte 254
7.6 Zusammenstellung der Operatoren 255
7.7 Konvertierung von Datentypen 276
7.8 Zusammenfassung 285
7.9 Übungen 287
8 Kontrollstrukturen 291
8.1 Blöcke – Kontrollstrukturen für die Sequenz 292
8.2 Selektion 293

8.3 Iteration ... 299
8.4 Sprunganweisungen ... 306
8.5 Zusammenfassung ... 309
8.6 Übungen ... 310
9 Blöcke und Methoden ... 317
9.1 Blöcke und ihre Besonderheiten ... 318
9.2 Methodendefinition und -aufruf ... 324
9.3 Polymorphie von Operationen ... 335
9.4 Überladen von Methoden ... 337
9.5 Parameterliste variabler Länge ... 340
9.6 Parameterübergabe beim Programmaufruf ... 342
9.7 Iteration und Rekursion ... 345
9.8 Zusammenfassung ... 351
9.9 Übungen ... 354
10 Klassen und Objekte ... 361
10.1 Information Hiding ... 362
10.2 Klassenvariablen und Klassenmethoden ... 365
10.3 Die `this`-Referenz ... 372
10.4 Initialisierung von Datenfeldern ... 379
10.5 Instanziierung von Klassen ... 392
10.6 Freigabe von Speicher ... 394
10.7 Die Klasse `Object` ... 398
10.8 Zusammenfassung ... 399
10.9 Übungen ... 403
11 Vererbung und Polymorphie ... 413
11.1 Das Konzept der Vererbung ... 414
11.2 Erweitern und Überschreiben ... 419
11.3 Besonderheiten bei der Vererbung ... 429
11.4 Polymorphie und das liskovsche Substitutionsprinzip ... 453
11.5 Verträge ... 466
11.6 Identifikation der Klasse eines Objektes ... 474
11.7 Operatoren für Referenztypen ... 478
11.8 Konsistenzhaltung von Quell- und Bytecode ... 479
11.9 Kovarianz und Kontravarianz beim Überschreiben ... 482
11.10 Zusammenfassung ... 492

11.11 Übungen ... 497
12 Pakete ... 507
12.1 "Programmierung im Großen" ... 508
12.2 Pakete zur Gruppierung ... 511
12.3 Erstellung von Paketen ... 511
12.4 Benutzung von Paketen ... 513
12.5 Paketnamen ... 517
12.6 Gültigkeitsbereich von Klassennamen ... 521
12.7 Zugriffsmodifikatoren ... 524
12.8 Zusammenfassung ... 532
12.9 Übungen ... 536
13 Ausnahmebehandlung ... 545
13.1 Das Konzept des Exception Handling ... 546
13.2 Implementierung von Exception Handlern in Java ... 549
13.3 Die Klasse `Throwable` ... 554
13.4 Die Exception-Hierarchie ... 557
13.5 Behandlung von Ausnahmen ... 560
13.6 Vorteile des Exception-Konzeptes ... 574
13.7 Assertions ... 574
13.8 Zusammenfassung ... 580
13.9 Übungen ... 581
14 Schnittstellen ... 587
14.1 Trennung von Spezifikation und Implementierung ... 589
14.2 Ein weiterführendes Beispiel ... 591
14.3 Aufbau einer Schnittstelle ... 595
14.4 Verwenden von Schnittstellen ... 598
14.5 Vergleich Schnittstelle und abstrakte Basisklasse ... 612
14.6 Die Schnittstelle `Cloneable` ... 615
14.7 Marker Interfaces ... 623
14.8 Zusammenfassung ... 624
14.9 Übungen ... 627
15 Geschachtelte Klassen ... 635
15.1 Elementklassen ... 638
15.2 Lokale Klassen ... 642
15.3 Statisch geschachtelte Klassen und Schnittstellen ... 653

15.4 Realisierung von geschachtelten Klassen 656
15.5 Zusammenfassung 662
15.6 Übungen 663
16 Threads 669
16.1 Prozesskonzept für Betriebssystem-Prozesse 670
16.2 Zugriffe auf Betriebsmittel und Synchronisation 673
16.3 Schwergewichtige und leichtgewichtige Prozesse 675
16.4 Zustände und Zustandsübergänge von Betriebssystem-Prozessen. 678
16.5 Zustände und Zustandsübergänge von Threads 679
16.6 Programmierung von Threads 682
16.7 Scheduling von Threads 690
16.8 Zugriff auf gemeinsame Ressourcen 691
16.9 Dämon-Threads 714
16.10 Zusammenfassung 715
16.11 Übungen 717
17 Generizität 723
17.1 Generische Klassen 726
17.2 Eigenständig generische Methoden 742
17.3 Bounded Typ-Parameter und Wildcards 746
17.4 Generische Schnittstellen 755
17.5 Die Klasse `Class<T>` 766
17.6 Generizität bei variablen Parameterlisten 770
17.7 Generizität und Polymorphie 772
17.8 Zusammenfassung 775
17.9 Übungen 779
18 Einführung in Collections 793
18.1 Das Konzept der Collection-API 794
18.2 Eigenschaften und Fähigkeiten der Collections 802
18.3 Zusammenfassung 806
18.4 Übungen 807
19 Ein-/Ausgabe und Streams 809
19.1 Klassifizierung von Streams 810
19.2 Das Stream-Konzept 814
19.3 Bytestream-Klassen 817
19.4 Characterstream-Klassen 832

19.5 Standardeingabe und Standardausgabe 840
19.6 Ein Beispiel für die Anwendung von Stream-Klassen 844
19.7 Ein- und Ausgabe der Datenfelder von Objekten 848
19.8 Zusammenfassung 856
19.9 Übungen 857
Begriffsverzeichnis 861
Abkürzungsverzeichnis 871
Anhang A Der ASCII-Zeichensatz 873
Anhang B Gültigkeitsbereiche von Namen 877
Anhang C Die Klasse `System` 883
Anhang D `jar`-Dateien 887
Literaturverzeichnis 889
Index 891

Kapitel 1

Grundlagen der Programmierung

1.1 Das erste Java-Programm
1.2 Programme und ihre Abläufe
1.3 Entwurf mit Nassi-Shneiderman-Diagrammen
1.4 Zeichen
1.5 Variablen
1.6 Datentypen
1.7 Zusammenfassung
1.8 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_1

1 Grundlagen der Programmierung

Bevor man mit einer Programmiersprache umzugehen lernt, muss man wissen, was ein Programm prinzipiell ist, und wie man Programme konstruiert. Damit wird sich das erste Kapitel befassen. Leser, die bereits eine höhere Programmiersprache erlernt haben, können prüfen, ob sie die hier präsentierten Grundlagen (noch) beherrschen. Sie können ggf. dieses Kapitel "überfliegen".

Ehe es "zur Sache geht", wird zunächst als spielerischer Einstieg in Kapitel 1.1 das Programm "Hello, world" vorgestellt. Kapitel 1.2 stellt Programme und ihre Abläufe vor. Kapitel 1.3 erläutert die Veranschaulichung des Ablaufs von Programmen mittels Nassi-Shneiderman-Diagrammen, welche auch unter der Bezeichnung Struktogramme bekannt sind. Kapitel 1.4 diskutiert Zeichen, die über einen Texteditor eingegeben werden, und dabei insbesondere ihre rechnerinterne Darstellung. Anschließend befasst sich Kapitel 1.5 mit Variablen sowie Kapitel 1.6 mit der Erläuterung einiger einfacher Datentypen und des Typs einer Klasse.

1.1 Das erste Java-Programm

Seit Kernighan und Ritchie, den Vätern der Programmiersprache C, ist es Usus geworden, als erstes Beispiel in einer neuen Programmiersprache mit dem Programm "Hello, world" zu beginnen. Das Programm "Hello, world" macht nichts anderes, als `Hello, world!` auf dem Bildschirm auszugeben. In Java sieht das Programm "Hello, world" folgendermaßen aus:

```
// Datei: HelloWorld.java

public class HelloWorld   // Klasse zur Ausgabe von "Hello, world!"
{
   public static void main (String[] args)   // Methode main() zur
   {                                         // Ausgabe der Zeichen-
     System.out.println ("Hello, world!");[2]  // kette "Hello, world!"
   }
}
```

Die Methode `println()` – "print line" ausgesprochen – wird über `System.out.println()` aufgerufen und schreibt die Zeichenfolge `Hello, world!` auf den Bildschirm. In der Programmiersprache Java stellt man eine Zeichenfolge durch eine sogenannte **Zeichenkette** (einen **String**) dar. Eine Zeichenkette beginnt mit einem Anführungszeichen " und endet mit einem Anführungszeichen. Die Anführungszeichen sind nur Begrenzer und werden nicht auf dem Bildschirm ausgegeben.

Bitte erstellen Sie dieses Programm mit einem Texteditor, der Ihnen vertraut ist, und speichern Sie es unter dem Dateinamen `HelloWorld.java` in einer Datei ab. Die-

[2] Der Strichpunkt begrenzt eine sogenannte "Anweisung" an den Computer. Auf Anweisungen wird noch eingegangen.

ses Programm besteht aus einer **Klasse** mit dem Namen `HelloWorld`[3]. Eine Klasse ist dadurch gekennzeichnet, dass sie das Schlüsselwort `class` trägt. Beachten Sie, dass alles, was hinter zwei Schrägstrichen in einer Zeile steht, zusammen mit diesen beiden Schrägstrichen einen sogenannten **Kommentar** darstellt. Ein Kommentar dient zur Dokumentation eines Programms und hat keinerlei Einfluss auf den Ablauf des Programms. Es soll ferner nicht unerwähnt bleiben, dass eine "ablauffähige" Klasse eine Methode `main()` enthält.

In **Java** kann man nur **objektorientiert** programmieren. Alle Programme in Java basieren von ihrem Aufbau her komplett auf Klassen.

Bitte achten Sie beim Eintippen des Programms im Texteditor und bei der Vergabe des Dateinamens auf die **Groß- und Kleinschreibung**, da in Java zwischen Groß- und Kleinbuchstaben unterschieden wird. In anderen Worten: Java ist **case sensitiv**.

Kompilieren Sie das Programm mit dem **`javac`-Compiler**[4] des Java Development Kits[5] durch die folgende Eingabe auf der Kommandozeile:

```
javac HelloWorld.java
```

Nach der Eingabe dieses Befehls drücken Sie bitte die `<RETURN>`-Taste. Der `javac`-Compiler übersetzt dann den Java-Quellcode der Datei `HelloWorld.java` in sogenannten Bytecode (siehe Kapitel 3.3.1) und legt diesen in der Datei `HelloWorld.class` ab. Durch die Eingabe von

```
java HelloWorld
```

und das anschließende Drücken der `<RETURN>`-Taste wird der **Bytecode-Interpreter** `java` gestartet, der den Bytecode interpretiert, d. h. in Maschinencode[6] übersetzt und zur Ausführung bringt. Java-Anwendungen können – wie hier gezeigt wird – von der Kommandozeile aus gestartet werden. Sie können aber auch aus Entwicklungsumgebungen wie z. B. aus Eclipse aufgerufen werden.

Bild 1-1 zeigt die Ein- und Ausgaben in einer Windows-Konsole:

Bild 1-1 Kompilieren und Starten über Kommandos in der Windows-Konsole

[3] Java ist objektorientiert. Man arbeitet mit sogenannten Klassen, die Methoden und Variablen für die Speicherung von Daten – sogenannte Datenfelder – aufweisen. Methoden "tun" etwas. Sie können z. B. eine Rechenaufgabe oder eine Ausgabe durchführen.

[4] Der Name `javac` wurde als Abkürzung für **Java C**ompiler gewählt.

[5] Die Installation des Java Development Kits wird in Kapitel 3.5 beschrieben.

[6] Hierbei ist Maschinencode ein spezieller Code, den der entsprechende Prozessor versteht.

Zu beachten ist, dass der Interpreter `java` den Klassennamen `HelloWorld` und nicht den Dateinamen `HelloWorld.class` verlangt!

So schnell kann es also gehen. Das erste Programm läuft schon. Sie hatten "ein Händchen" im Umgang mit Texteditor, Compiler und Interpreter. Da es hier nur darum geht, ein allererstes Programm zu starten, wird auf eine detaillierte Erläuterung des Programms verzichtet.

1.2 Programme und ihre Abläufe

Der Begriff **Programm** ist eng mit dem Begriff **Algorithmus** verbunden. Algorithmen sind Vorschriften für die **Lösung eines Problems**, welche die Handlungen und ihre Abfolge – kurz, die Handlungsweise – beschreiben. Im Alltag begegnet man Algorithmen in Form von Bastelanleitungen, Kochrezepten und Gebrauchsanweisungen. Abstrakt kann man sagen, dass die folgenden Bestandteile und Eigenschaften zu einem Algorithmus gehören:

- eine **Menge von Objekten**, die durch den Algorithmus bearbeitet werden,
- eine **Menge von Operationen**, die auf den Objekten ausgeführt werden,
- ein definierter **Anfangszustand**, in dem sich die Objekte zu Beginn befinden,
- und ein gewünschter **Endzustand**, in dem die Objekte nach der Lösung des Problems sein sollen.

Dies sei am Beispiel Kochrezept erläutert:

Objekte:	Zutaten, Geschirr, Herd, ...
Operationen:	waschen, anbraten, schälen, passieren, ...
Anfangszustand:	Zutaten im "Rohzustand", Teller leer, Herd kalt, ...
Endzustand:	fantastische Mahlzeit auf dem Teller, Herd aus.

Was noch zur Lösung eines Problems gebraucht wird, ist eine Anleitung, ein Rezept oder eine Folge von Anweisungen, und jemand, der diese Anleitung auch umsetzt. Mit anderen Worten, man benötigt einen **Algorithmus** – eine Rechenvorschrift – und einen ausführenden **Prozessor**.

Während aber bei einem Kochrezept viele Dinge gar nicht explizit zum Ausdruck gebracht werden müssen, sondern dem Koch aufgrund seiner Erfahrung implizit klar sind – z. B., dass er den Kuchen aus dem Backofen holen muss, bevor er schwarz ist –, muss einem Prozessor alles explizit und eindeutig durch ein Programm, das aus Anweisungen einer **Programmiersprache** besteht, gesagt werden.

Ein **Algorithmus in einer Programmiersprache** besteht aus **Anweisungen**, die von einem Prozessor ausgeführt werden können.

Bild 1-2 zeigt Anweisungen, die im Arbeitsspeicher des Rechners abgelegt sind und nacheinander durch den Prozessor des Rechners abgearbeitet werden:

Bild 1-2 Der Prozessor bearbeitet eine Anweisung eines Programms nach der anderen

1.2.1 Der Algorithmus von Euklid als Beispiel für Algorithmen

Als Beispiel wird der Algorithmus betrachtet, der von Euklid ca. 300 v. Chr. zur Bestimmung des größten **gemeinsamen Teilers** (**ggT**) zweier natürlicher Zahlen aufgestellt wurde. Der größte gemeinsame Teiler wird zum Kürzen von Brüchen benötigt. Den Einsatz des ggT zeigt die folgende Formel:

$$\frac{x_{ungekürzt}}{y_{ungekürzt}} = \frac{x_{ungekürzt}/ggT(x_{ungekürzt}, y_{ungekürzt})}{y_{ungekürzt}/ggT(x_{ungekürzt}, y_{ungekürzt})} = \frac{x_{gekürzt}}{y_{gekürzt}}$$

Hierbei ist $ggT(x_{ungekürzt}, y_{ungekürzt})$ der größte gemeinsame Teiler der beiden Zahlen $x_{ungekürzt}$ und $y_{ungekürzt}$.

Beispiel: $\frac{24}{9} = \frac{24/ggT(24,9)}{9/ggT(24,9)} = \frac{24/3}{9/3} = \frac{8}{3}$

Der größte gemeinsame Teiler in diesem Beispiel ist also die Zahl 3.

Der Algorithmus von Euklid lautet:

Zur Bestimmung des größten gemeinsamen Teilers zweier natürlicher Zahlen `x` und `y` mache Folgendes[7]:

Solange `x` ungleich `y` ist, wiederhole:
    Wenn `x` größer als `y` ist, dann:
        Ziehe `y` von `x` ab und weise das Ergebnis `x` zu.
    Andernfalls:
        Ziehe `x` von `y` ab und weise das Ergebnis `y` zu.
Wenn `x` gleich `y` ist, dann:
    `x` (bzw. `y`) ist der gesuchte größte gemeinsame Teiler.

Man erkennt in diesem Beispiel Folgendes:

- Es gibt eine Menge von Objekten, mit denen etwas passiert: `x` und `y`. Diese Objekte `x` und `y` haben am Anfang beliebig vorgegebene Werte, am Schluss enthalten sie den größten gemeinsamen Teiler.

[7] Die Arbeitsweise dieses Algorithmus für die Zahlen `x` gleich `24` und `y` gleich `9` wird durch die Tabelle 1-1 in Kapitel 1.2.4 verdeutlicht.

- Es gibt gewisse Grundoperationen, die hier nicht weiter erläutert werden, da sie implizit klar sind: vergleichen, abziehen und zuweisen.
- Es handelt sich um eine sequenzielle Folge von **Anweisungen** (**Operationen**), d. h., die Anweisungen werden der Reihe nach hintereinander ausgeführt.
- Es gibt aber auch bestimmte Konstrukte, welche die einfache sequenzielle Folge (**Sequenz**, **Hintereinanderausführung**) gezielt verändern: eine Auswahl zwischen Alternativen (**Selektion**) und eine Wiederholung von Anweisungen (**Iteration**).

Es gibt auch Algorithmen zur Beschreibung von parallelen Aktivitäten, die zum gleichen Zeitpunkt nebeneinander ausgeführt werden. Diese Algorithmen werden unter anderem bei Betriebssystemen oder in der Prozessdatenverarbeitung benötigt. In den folgenden Unterkapiteln werden bewusst nur sequenzielle Abläufe behandelt, bei denen zu einem Zeitpunkt nur eine einzige Operation durchgeführt wird.

1.2.2 Beschreibung sequenzieller Abläufe

Die **Abarbeitungsreihenfolge von Anweisungen** wird auch als **Kontrollfluss** bezeichnet.

Den Prozessor stört es überhaupt nicht, wenn eine Anweisung einen Sprungbefehl zu einer anderen Anweisung enthält. Solche Sprungbefehle werden in manchen Programmiersprachen beispielsweise mit dem Befehl `GOTO` und Marken wie z. B. `100` realisiert, wie das folgende Beispiel zeigt:

```
     IF(a > b) GOTO 100
     Anweisungen2
     GOTO 300
100  Anweisungen1
300  Anweisungen3
```

In Worten lauten diese Anweisungen an den Prozessor: "Vergleiche die Werte von `a` und `b`. Wenn[8] `a` größer als `b` ist, springe an die Stelle mit der Marke `100`. Führe an der Stelle mit der Marke `100` die Anweisungen `Anweisungen1` aus. Fahre dann mit den Anweisungen `Anweisungen3` fort. Ist aber die Bedingung `a > b` nicht erfüllt, so arbeite die Anweisungen `Anweisungen2` ab. Springe dann zu der Marke `300` und führe die Anweisungen `Anweisungen3` aus."

Will jedoch ein Programmierer ein solches Programm lesen, so verliert er durch die Sprünge sehr leicht den Zusammenhang und damit das Verständnis. Für den menschlichen Leser ist es am besten, wenn ein Programm einen einfachen und damit überschaubaren Kontrollfluss hat. Während typische Programme der sechziger Jahre des letzten Jahrhunderts noch zahlreiche Sprünge enthielten, bemühen sich die Programmierer seit Dijkstras grundlegendem Artikel "Go To Statement Considered Harmful" [Dij72], möglichst einen Kontrollfluss ohne Sprünge zu ent-

[8] "Wenn" wird ausgedrückt durch das Schlüsselwort `IF` der hier verwendeten Programmiersprache FORTRAN.

werfen. Beispielsweise kann der oben mit `GOTO` beschriebene Ablauf auch folgendermaßen realisiert werden:

```
IF (a > b)
   Anweisungen1
ELSE
   Anweisungen2
ENDIF
Anweisungen3
```

Hierbei ist wieder `IF (a > b)` die Abfrage, ob `a` größer als `b` ist. Ist dies der Fall, so werden die Anweisungen `Anweisungen1` ausgeführt. Ist die Bedingung `a > b` nicht wahr, d. h. nicht erfüllt, so werden die Anweisungen `Anweisungen2` des `ELSE`-Zweigs durchgeführt. Mit `ENDIF` ist die Fallunterscheidung zu Ende. Unabhängig davon, welcher der beiden Zweige der Fallunterscheidung abgearbeitet wurde, werden nun die Anweisungen `Anweisungen3` durchgeführt. Dieser Ablauf ist in Bild 1-3 grafisch veranschaulicht:

Bild 1-3 Grafische Darstellung einer Verzweigung

Unter einer **Kontrollstruktur** versteht man eine Anweisung, welche die **Abarbeitungsreihenfolge** von Anweisungen **beeinflusst**.

Betrachtet man nur sequenzielle Abläufe[9], so gibt es Kontrollstrukturen für

- die **Selektion**,
- die **Iteration**
- und die **Sequenz**.

Zu den Kontrollstrukturen gehört die **Fallunterscheidung** (**Selektion**), bei der in Abhängigkeit davon, ob eine Bedingung erfüllt ist oder nicht, entweder die eine oder die andere Anweisung abgearbeitet wird, oder eine **Wiederholung** (**Iteration**) einer An-

[9] Also keine parallelen Abläufe

weisung. Zu den Kontrollstrukturen gehört auch die sogenannte **Sequenz**. Eine Sequenz ist eine Anweisungsfolge (ein sogenannter Block), die eine sequenzielle Folge von Anweisungen enthält, wobei die ganze Anweisungsfolge von der Sprachsyntax her als eine einzige Anweisung zu werten ist.

Im Beispiel des Algorithmus von Euklid stellt

"Solange x ungleich y ist, wiederhole:

. . . . . "

eine **Iteration** dar, die in freier Sprache ausgedrückt ist.

"Wenn x größer als y ist, dann:

. . . . .

andernfalls:

. . . . . "

stellt eine **Fallunterscheidung** (**Selektion**) in freier Sprache dar.

1.2.3 Strukturierte Programmierung

Die Ideen von Dijkstra und anderen fanden ihren Niederschlag in den Regeln für die **Strukturierte Programmierung**. Die Strukturierte Programmierung ist ein programmiersprachenunabhängiges Konzept. Es umfasst die Zerlegung eines Programms in Teil-Programme (Haupt-[10] und Unterprogramme) sowie die folgenden Regeln für den Kontrollfluss. Danach gilt, dass in einer **Sequenz** eine Anweisung nach der anderen – d. h. in einer linearen Reihenfolge – abgearbeitet wird. Man geht über einen einzigen Eingang (engl. **single entry**), nämlich von der davor stehenden Anweisung in eine Anweisung hinein und geht über einen einzigen Ausgang (engl. **single exit**) aus der Anweisung heraus und kommt direkt zur nächsten Anweisung. Dies soll durch das folgende Bild symbolisiert werden:

Bild 1-4 Single entry und single exit bei der Sequenz

[10] Mit dem Hauptprogramm wird das Programm gestartet. Das Hauptprogramm kann Unterprogramme aufrufen. Unterprogramme können ebenso Unterprogramme aufrufen.

Haben Kontrollstrukturen für die **Selektion** und **Iteration** die gleichen Eigenschaften wie einzelne sequentielle Anweisungen, nämlich single entry und single exit, so erhält man für alle Anweisungen einen linearen und damit überschaubaren Programmablauf. Programme, die nur Kontrollstrukturen mit dieser Eigenschaft aufweisen, gehorchen den Regeln der **Strukturierten Programmierung** und können mit Hilfe von **Nassi-Shneiderman-Diagrammen** visualisiert werden (siehe Kapitel 1.3).

Bei der **Strukturierten Programmierung** wird ein Programm in Teil-Programme zerlegt und es werden nur **Kontrollstrukturen** mit **einem einzigen Eingang** und **einem einzigen Ausgang** verwendet.

Mit der Anweisung `GOTO MARKE`, d. h. einer Sprunganweisung, wäre es möglich, die Ausführung eines Programms an einer ganz anderen Stelle, nämlich an der Stelle, an der die Marke `MARKE` steht, fortzusetzen. Dies ist aber in der Strukturierten Programmierung nicht zulässig.

1.2.4 Variablen und Zuweisungen

Die durch den Algorithmus von Euklid behandelten Objekte sind natürliche Zahlen. Diese Zahlen sollen jedoch nicht von vornherein festgelegt werden, sondern der Algorithmus soll für die Bestimmung des größten gemeinsamen Teilers beliebiger natürlicher Zahlen verwendbar sein. Anstelle der Zahlen werden daher Namen verwendet, die einem Speicherplatz im Arbeitsspeicher zugeordnet sind und als **variable Größen** oder kurz **Variablen** bezeichnet werden. Den Variablen werden im Verlauf des Algorithmus konkrete Werte zugewiesen. Diese Wertzuweisung an Variablen ist eine der grundlegenden Operationen, die ein Prozessor ausführen können muss. Auf Variablen wird noch ausführlicher in Kapitel 1.5 eingegangen.

Der im obigen Beispiel beschriebene Algorithmus kann auch von einem menschlichen "Prozessor" ausgeführt werden – andere Möglichkeiten hatten die Griechen in der damaligen Zeit nicht. Als Hilfsmittel braucht man dazu Papier und Bleistift, um die Zustände der Objekte – im obigen Beispiel die Zustände der Objekte `x` und `y` – zwischen den Verarbeitungsschritten festzuhalten. Man erhält dann eine Tabelle, die auch **Trace-Tabelle**[11] genannt wird und für die Zahlen `x` gleich `24` und `y` gleich `9` das folgende Aussehen hat:

Verarbeitungsschritt	Werte von x	Werte von y
Initialisierung		
`x = 24, y = 9`	24	9
`x = x - y`	15	9
`x = x - y`	6	9
`y = y - x`	6	3
`x = x - y`	3	3
Ergebnis: ggT = 3		

Tabelle 1-1 Trace der Variableninhalte für Initialwerte `x` == `24`, `y` == `9`

[11] Mit einer Trace-Tabelle verfolgt man die Zustände (Werte) der Variablen.

Beachten Sie, dass in der Unterschrift von Tabelle 1-1 "`x == 24`" zu lesen ist als "x ist gleich 24". Damit werden hier wie in Java zwei Gleichheitszeichen direkt hintereinander als **Gleichheitssymbol** verwendet. Tabelle 1-1 zeigt, dass die Variablen über den Verlauf des Algorithmus hinweg unterschiedliche Werte repräsentieren.

Zu Beginn eines Algorithmus werden den **Variablen definierte Anfangs- oder Startwerte** zugewiesen.

Diesen Vorgang bezeichnet man als **Initialisierung** der Variablen.

Eine **Werteänderung** erfolgt durch sogenannte **Zuweisungen**.

Solche Zuweisungen sind beispielsweise in den Verarbeitungsschritten von Tabelle 1-1 beschrieben. Als **Zuweisungssymbol** wird hier das Gleichheitszeichen (`=`) benutzt, wie es in Java üblich ist.

Für eine andere Ausgangssituation sieht die Trace-Tabelle beispielsweise so aus:

	Werte von	
Verarbeitungsschritt	`x`	`y`
Initialisierung `x = 5, y = 3`	5	3
`x = x - y`	2	3
`y = y - x`	2	1
`x = x - y`	1	1
Ergebnis: ggT = 1		

Tabelle 1-2 Trace der Variableninhalte für Initialwerte `x == 5, y == 3`

Die Schreibweise `x = x - y` ist zunächst etwas verwirrend. Diese Schreibweise ist nicht als mathematische Gleichung zu sehen, sondern meint etwas ganz anderes: Auf der rechten Seite des Gleichheitszeichens steht ein arithmetischer Ausdruck, dessen Wert zuerst berechnet werden soll. Dieser so berechnete Wert wird dann in einem zweiten Schritt der Variablen zugewiesen, deren Name auf der linken Seite steht. Im Beispiel also:

- Nimm den aktuellen Wert von `x`. Nimm den aktuellen Wert von `y`. Ziehe den Wert von `y` vom Wert von `x` ab.
- Der neue Wert von `x` ist die soeben ermittelte Differenz von `x` und `y`.

Eine Zuweisung verändert den Wert der Variablen, also den Zustand der Variablen, auf der linken Seite.

Bei einer **Zuweisung** wird zuerst der Ausdruck rechts vom Zuweisungsoperator berechnet und der Wert dieses Ausdrucks der Variablen auf der linken Seite des Zuweisungsoperators zugewiesen.

Variablen speichern Werte. Der von einer Variablen gespeicherte Wert kann durch eine Zuweisung mit einem neuen Wert überschrieben werden.

Die Beispiele in diesem Kapitel zeigen, wie ein Algorithmus sequenzielle Abläufe und Zustandstransformationen seiner Variablen beschreibt. Wird derselbe Algorithmus zweimal durchlaufen, wobei die Variablen am Anfang unterschiedliche Werte haben, dann erhält man in aller Regel auch unterschiedliche Abläufe. Sie folgen aber ein und demselben Verhaltensmuster, das durch den Algorithmus beschrieben ist.

1.2.5 Vom Algorithmus zum Programm

Wenn ein **Algorithmus** derart formuliert ist, dass seine Ausführung durch einen bestimmten Prozessor möglich ist, dann spricht man auch von einem **Programm** für diesen Prozessor.

Bei einem Computerprogramm müssen alle Einzelheiten bis ins kleinste Detail festgelegt sein und die Sprachregeln sind absolut einzuhalten.

Ein Prozessor macht eben haarscharf nur das, was durch das Programm festgelegt ist, und nicht das, was noch zwischen den Zeilen steht. Hingegen muss ein Koch bei einem Rezept Erfahrungen mit einbringen und beispielsweise den Topf mit der Milch vom Herd nehmen, bevor die Milch überläuft.

Generell kann man bei Sprachen zwischen **natürlichen Sprachen** wie der Umgangssprache oder den Fachsprachen einzelner Berufsgruppen und **formalen Sprachen** unterscheiden. Formale Sprachen sind beispielsweise die Notenschrift in der Musik, die Formelschrift in der Mathematik oder Programmiersprachen beim Computer. Nur das, was durch eine formale Sprache – die Programmiersprache – festgelegt ist, ist für den Prozessor verständlich.

1.3 Entwurf mit Nassi-Shneiderman-Diagrammen

Zur Visualisierung des Kontrollflusses von Programmen – das heißt, zur grafischen Veranschaulichung ihres Ablaufes – wurden 1973 von Nassi und Shneiderman [Nas73] grafische Strukturen, die sogenannten **Struktogramme** (siehe [DIN 66261]), vorgeschlagen. Diese Struktogramme werden nach ihren Urhebern oftmals auch als **Nassi-Shneiderman-Diagramme** bezeichnet.

Nassi-Shneiderman-Diagramme enthalten kein `GOTO`, sondern nur die Sprachmittel der **Strukturierten Programmierung**, insbesondere die **Sequenz**, **Iteration** und **Selektion**.

Entwirft man Programme mit Nassi-Shneiderman-Diagrammen, so genügt man automatisch den Regeln der Strukturierten Programmierung. Nassi und Shneiderman

schlugen ihre Struktogramme als **Ersatz** für die bis dahin üblichen **Flussdiagramme** (siehe [DIN 66001]) vor. Traditionelle Flussdiagramme erlauben einen Kontrollfluss mit beliebigen Sprüngen in einem Programm. Spezifiziert und programmiert man strukturiert, so geht der Kontrollfluss eines solchen Programmes einfach von oben nach unten – eine Anweisung folgt der nächsten. Wilde Sprünge, welche die Übersicht erschweren, sind nicht zugelassen.

Das wichtigste Merkmal der Struktogramme ist, dass jeder **Verarbeitungsschritt** durch ein **rechteckiges Sinnbild** dargestellt wird. Ein Verarbeitungsschritt kann dabei eine Anweisung oder eine Gruppe von zusammengehörigen Anweisungen sein.

Das folgende Symbol zeigt das Sinnbild für einen Verarbeitungsschritt:

Bild 1-5 Sinnbild für einen Verarbeitungsschritt

Die obere Linie des Rechtecks bedeutet den Beginn eines Verarbeitungsschrittes, die untere Linie bedeutet das Ende dieses Verarbeitungsschrittes. Jedes Sinnbild erhält eine Innenbeschriftung, die den Verarbeitungsschritt näher beschreibt.

1.3.1 Diagramme für die Sequenz

Bei der Sequenz folgen Verarbeitungsschritte sequenziell hintereinander.

Dies wird durch ein Nassi-Shneiderman-Diagramm für die Verarbeitungsschritte `V1` und `V2` wie folgt dargestellt:

Bild 1-6 Nassi-Shneiderman-Diagramm für die Sequenz

Eine Kontrollstruktur für die Sequenz ist der Block. Der **Block** stellt eine Folge logisch zusammenhängender Verarbeitungsschritte dar. Er kann einer **Methode oder Funktion**[12] in einer Programmiersprache entsprechen, kann aber auch nur einfach **mehrere Verarbeitungsschritte** unter einem Namen zusammenfassen. Das folgende Bild visualisiert den Block:

[12] Anweisungsfolgen, die unter einem Namen aufgerufen werden können, heißen in der objektorientierten Programmierung "Methoden", in der prozeduralen Programmierung "Funktionen" wie z. B. in C oder auch "Prozeduren".

Bild 1-7 Sinnbild für einen Block

Wie in Bild 1-7 zu sehen ist, wird der Name des Blockes im Diagramm den Verarbeitungsschritten vorangestellt.

Das Diagramm in Bild 1-8 stellt das "Hello, world"-Programm aus Kapitel 1.1 in grafischer Form dar:

Bild 1-8 Einfaches Beispiel eines Struktogramms

Aus der Darstellung ist zu entnehmen, dass die Details einer Programmiersprache auf dieser Abstraktionsebene keine Rolle spielen.

1.3.2 Diagramme für die Selektion

Bei den Kontrollstrukturen für die Selektion kann man zwischen

- der einfachen Alternative,
- der bedingten Verarbeitung
- und der mehrfachen Alternative

unterscheiden.

Siehe hierzu Bild 1-9 bis Bild 1-11.

Die **einfache Alternative** stellt eine Verzweigung im Programmablauf dar. Das entsprechende Struktogramm ist in Bild 1-9 zu sehen:

Bild 1-9 Struktogramm für die einfache Alternative

Bei der **einfachen Alternative** wird überprüft, ob ein **boolescher Ausdruck**[13] wie z. B. `a > b` wahr ist oder nicht.

[13] Ein **Ausdruck** ist eine Verknüpfung von Operanden durch Operatoren und runden Klammern (siehe Kapitel 7).

Ein **boolescher Ausdruck** kann die Wahrheitswerte `TRUE` bzw. `FALSE` annehmen. Ein solcher boolescher Ausdruck wird auch als **Bedingung** bezeichnet.

Ist der Ausdruck wahr, so wird der Zweig für `TRUE` ausgewählt und der Verarbeitungsschritt `V1` ausgeführt. Ist der Ausdruck nicht wahr, so wird der `FALSE`-Zweig ausgewählt und der Verarbeitungsschritt `V2` durchgeführt. Jeder dieser Zweige kann einen Verarbeitungsschritt bzw. einen Block von Verarbeitungsschritten enthalten.

Die einfache Alternative wurde bereits in Kapitel 1.2.2 ausführlich besprochen. Hier noch einmal der entsprechende Pseudocode für eine solche Verzweigung:

```
if (a > b) V1
else V2
```

Bei der **bedingten Verarbeitung** (siehe Bild 1-10) wird der `TRUE`-Zweig ausgewählt, wenn der Ausdruck wahr ist. Ansonsten wird direkt zu dem nächsten Verarbeitungsschritt übergegangen. Das folgende Bild visualisiert die bedingte Verarbeitung:

Bild 1-10 Struktogramm für die bedingte Verarbeitung

Der Pseudocode für die bedingte Verarbeitung entspricht dem der einfachen Alternative, allerdings fehlt der sogenannte `else`-Teil:

```
if (a > b) V1
```

Bei der **mehrfachen Alternative** (siehe Bild 1-11) wird geprüft, ob ein **arithmetischer Ausdruck**[14] einen von n vorgegebenen Werten c_1 ... c_n annimmt. Ist dies der Fall, so wird der entsprechende Zweig ausgeführt, ansonsten wird direkt zu dem nächsten Verarbeitungsschritt übergegangen. Das folgende Bild visualisiert die mehrfache Alternative:

Bild 1-11 Struktogramm für die mehrfache Alternative

[14] Bei einem arithmetischen Ausdruck werden arithmetische Operatoren auf die Operanden angewandt wie z. B. der Minusoperator im Ausdruck `6 - 2` auf die Operanden `6` und `2`.

Der entsprechende Pseudocode ist relativ komplex. Daher wird auf eine Darstellung hier im Einleitungskapitel verzichtet.

1.3.3 Diagramme für die Iteration

Bei der Iteration kann man drei Fälle von Kontrollstrukturen unterscheiden:

- Wiederholung mit vorheriger Prüfung,
- Wiederholung mit nachfolgender Prüfung
- und Wiederholung ohne Prüfung.

Bei der **Wiederholung mit vorheriger Prüfung** (**abweisende Schleife**) wird zuerst eine Bedingung geprüft. Solange diese Bedingung erfüllt ist, wird der Verarbeitungsschritt V wiederholt. Ist diese Bedingung bereits zu Anfang nicht erfüllt, wird der Verarbeitungsschritt V nicht ausgeführt – die Ausführung der Schleife wird abgewiesen. Das Struktogramm einer abweisenden Schleife ist in Bild 1-12 dargestellt:

Bild 1-12 Struktogramm der Wiederholung mit vorausgehender Bedingungsprüfung

In einem Pseudocode kann man eine abweisende Schleife folgendermaßen darstellen:

```
WHILE (Bedingung) DO V
```

Hat zu Beginn der Schleife die Bedingung `Bedingung` den Wert `TRUE`, dann müssen die Verarbeitungsschritte in der Schleife dafür sorgen, dass der Wert der Bedingung irgendwann `FALSE` wird, sonst entsteht eine **Endlos-Schleife**[15]. Die `FOR`-Schleife (siehe auch Kapitel 8.3.2) ist ebenfalls eine abweisende Schleife. Sie stellt eine spezielle Ausprägung der `WHILE`-Schleife dar. `FOR`-Schleifen bieten eine syntaktische Beschreibung des Startzustandes und der Iterationsschritte (z. B. Hoch- oder Herunterzählen einer Laufvariablen, welche die einzelnen Iterationsschritte durchzählt).

Bei der **Wiederholung mit nachfolgender Prüfung** (**annehmende Schleife**) erfolgt die Prüfung der Bedingung erst am Ende. Das zugehörige Struktogramm ist in Bild 1-13 dargestellt:

Bild 1-13 Struktogramm der Wiederholung mit nachfolgender Bedingungsprüfung

[15] Eine **Endlos-Schleife** ist eine Schleife, deren Ausführung nie abbricht.

Die annehmende Schleife kann man in einem Pseudocode folgendermaßen darstellen:

```
DO V WHILE (Bedingung)
```

Die annehmende Schleife wird mindestens einmal durchgeführt. Erst dann wird die Bedingung bewertet. Die `DO-WHILE`-Schleife wird typischerweise dann benutzt, wenn der Wert der Bedingung erst in der Schleife entsteht, beispielsweise wie in der folgenden Anwendung "Lies Zahlen ein, solange keine `0` eingegeben wird". Hier muss zuerst eine Zahl eingelesen werden. Erst dann kann geprüft werden, ob sie `0` ist oder nicht.

Prüfungen müssen nicht immer notwendigerweise zu Beginn oder am Ende stattfinden. Eine Bedingung muss manchmal auch in der Mitte der Verarbeitungsschritte einer Schleife geprüft werden. Zu diesem Zweck gibt es die Wiederholung ohne Prüfung, bei der die Prüfung in den Verarbeitungsschritten "versteckt" ist. Das zugehörige Struktogramm ist in Bild 1-14 dargestellt:

Bild 1-14 Struktogramm der Wiederholung ohne Bedingungsprüfung

In einem Pseudocode kann die Schleife ohne Bedingungsprüfung folgendermaßen angegeben werden:

```
LOOP V
```

Die **Schleife ohne Bedingungsprüfung** wird verlassen, wenn in einem der Verarbeitungsschritte V eine `BREAK`-**Anweisung** ausgeführt wird.

Eine `BREAK`-Anweisung ist eine spezielle Sprunganweisung und sollte nur eingesetzt werden, damit bei einer Schleife ohne Wiederholungsprüfung keine Endlos-Schleife entsteht. Die Regel, dass eine Kontrollstruktur nur einen einzigen Eingang und einen einzigen Ausgang hat, wird dadurch nicht verletzt, sondern der zunächst fehlende Ausgang wird erst durch die `BREAK`-Anweisung zur Verfügung gestellt.

Bild 1-15 zeigt das Sinnbild für eine solche Abbruchanweisung:

Bild 1-15 Abbruchanweisung

Im Falle der Programmiersprache Java sind die Kontrollstrukturen der Wiederholung mit vorheriger Prüfung und mit nachfolgender Prüfung als Sprachkonstrukt vorhanden, d. h., es gibt in Java Anweisungen für diese Schleifen. Eine Wiederholung ohne Prüfung kann in Java auch formuliert werden, aber es gibt keine spezielle Anweisung

dafür. Bild 1-16 stellt ein Beispiel für eine Schleife ohne Wiederholungsprüfung mit Abbruchanweisung dar:

Bild 1-16 Struktogramm einer Schleife ohne Wiederholungsprüfung mit Abbruchbedingung

Nach der Ausführung des Verarbeitungsschritts `V1` wird die Bedingung geprüft. Hat die Bedingung nicht den Wert `TRUE`, so wird der Verarbeitungsschritt `V2` abgearbeitet und dann die Schleife mit dem Verarbeitungsschritt `V1` beginnend wiederholt. Der Durchlauf der Schleife mit der Reihenfolge "Ausführung `V1`", "Bedingungsprüfung", "Ausführung `V2`" wird solange wiederholt, bis die Bedingung den Wert `TRUE` hat. In diesem Fall wird die Schleife durch die Abbruchbedingung verlassen.

1.3.4 Algorithmus von Euklid als Nassi-Shneiderman-Diagramm

Mit den Mitteln der Struktogramme kann nun der Algorithmus von Euklid, der in Kapitel 1.2.1 eingeführt wurde, in grafischer Form dargestellt werden:

Bild 1-17 Struktogramm des Algorithmus von Euklid

1.4 Zeichen

Wenn ein Programm mit Hilfe eines Texteditors geschrieben wird, werden Zeichen über die Tastatur eingegeben. Einzelne oder mehrere aneinandergereihte Zeichen haben hierbei eine spezielle Bedeutung. So repräsentieren die Zeichen `x` und `y` bei der Implementierung[16] des Algorithmus von Euklid die Namen von Variablen.

[16] Implementierung bedeutet Realisierung, Umsetzung, Verwirklichung.

Ein **Zeichen** ist ein von anderen Zeichen unterscheidbares Objekt, welches in einem bestimmten Zusammenhang eine definierte Bedeutung trägt.

Zeichen können beispielsweise Symbole, Bilder oder Töne sein. Zeichen derselben Art sind **Elemente** eines **Zeichenvorrats**. So sind beispielsweise die Zeichen I, V, X, L, C, M Elemente des Zeichenvorrats der römischen Zahlen. Eine **Ziffer** ist ein Zeichen, das die Bedeutung einer Zahl hat.

Von einem **Alphabet** spricht man, wenn der Zeichenvorrat eine strenge Ordnung aufweist.

So stellt beispielsweise die geordnete Folge der Elemente

0, 1	das **Binäralphabet**,
a, b, c ... z	die **Kleinbuchstaben** ohne Umlaute und ohne ß,
0, 1, ... 9	das **Dezimalalphabet**

dar.

Rechnerinterne Darstellung von Zeichen

Zeichen sind zunächst **Buchstaben**, **Ziffern** oder **Sonderzeichen**. Zu diesen Zeichen können noch **Steuerzeichen** hinzukommen. Ein Steuerzeichen ist beispielsweise ^C, das durch gleichzeitiges Anschlagen der Taste Strg (Steuerung) und der Taste C erzeugt wird. Die Eingabe von ^C kann dazu dienen, ein Programm abzubrechen.

Rechnerintern werden die Zeichen durch Bits dargestellt. Ein **Bit**[17] kann den Wert 0 oder 1 annehmen. Das bedeutet, dass man mit einem Bit zwei verschiedene Fälle darstellen kann. Mit einer Gruppe von zwei Bits hat man 2 * 2 = 4 Möglichkeiten, mit einer Gruppe von drei Bits kann man 2 * 2 * 2 = 8 verschiedene Fälle darstellen und so fort. Mit drei Bits sind die im folgenden Bild gezeigten Kombinationen möglich:

Bild 1-18 Bitkombinationen aus 3 Bits

Jeder dieser Bitgruppen kann man je ein Zeichen zuordnen, das heißt, jede dieser Bitkombinationen kann ein Zeichen repräsentieren. Man braucht nur eine eindeutig umkehrbare Zuordnung (z. B. erzeugt durch eine Tabelle) und kann dann jedem Zeichen eine Bitkombination und jeder Bitkombination ein Zeichen zuordnen. Mit anderen Worten, man bildet die Elemente eines Zeichenvorrats auf die Elemente eines anderen Zeichenvorrats ab. Diese Abbildung bezeichnet man als **Codierung**.

[17] Abkürzung für binary digit (engl.) = Binärziffer.

Begriff eines Codes

Nach DIN 44300 ist ein **Code** eine Vorschrift für die eindeutige Zuordnung oder Abbildung der Zeichen eines Zeichenvorrats auf die Zeichen eines anderen Zeichenvorrats, die sogenannte Bildmenge.

Dieser Begriff des Codes wird aber nicht eindeutig verwendet.

Oftmals wird unter Code auch der Zeichenvorrat der Bildmenge verstanden.

Relevante Codes für Rechner

Für die Codierung von Zeichen im Binäralphabet gibt es viele Möglichkeiten. Für Rechner besonders relevant sind Codes, die ein Zeichen durch 7 bzw. 8 Bits repräsentieren. Mit 7 Bits kann man 128 verschiedene Zeichen codieren, mit 8 Bits 256 Zeichen. Zu den am häufigsten verwendeten Zeichensätzen gehören:

- Der **ASCII**[18]**-Zeichensatz** mit 128 Zeichen – die US-nationale Variante des ISO-7-Bit-Code (ISO 646), die aber weit verbreitet ist.
- Der erweiterte ASCII-Zeichensatz mit 256 Zeichen.
- Der **Unicode**, der jedem Zeichen aller bekannten Schriftkulturen und Zeichensysteme eine Bitkombination zuordnet. Der Unicode definiert Codierungen verschiedener Länge. Java verwendet für Zeichen die ursprüngliche UTF-16-Repräsentation, bei der jedes Zeichen einer Bitkombination einer Gruppe von 16 Bits entspricht. Die ersten 128 Zeichen des UTF-16-Codes sind die Zeichen des 7-Bit ASCII-Zeichensatzes.

1.5 Variablen

Bei **imperativen Sprachen** – zu dieser Klasse von Sprachen gehört Java – besteht ein Programm aus einer Folge von Anweisungen, wie z. B. "Wenn `x` größer als `y` ist, dann:" oder "ziehe `y` von `x` ab und weise das Ergebnis `x` zu". Wesentlich an diesen Sprachen ist das Variablenkonzept – Eingabewerte werden in Variablen gespeichert und weiterverarbeitet.

Eine **Variable** ist eine benannte Speicherstelle. Über den **Variablennamen** kann der Programmierer auf die entsprechende Speicherstelle zugreifen.

Variablen braucht man, um in ihnen Werte abzulegen. Im Gegensatz zu einer Konstanten ist eine Variable eine veränderliche Größe. In ihrem Speicherbereich kann bei Bedarf der Wert einer Variablen verändert werden. Der **Wert** einer Variablen muss der Variablen **manuell vom Programmierer oder automatisch** zugewiesen werden. Ansonsten ist ihr Wert undefiniert. Variablen, die nicht automatisch initiali-

[18] Siehe Anhang A

siert werden (siehe Kapitel 10.4.1), müssen vom Programmierer selbst initialisiert werden. Ansonsten resultiert ein Kompilierfehler.

Der Programmierer darf es nicht versäumen, den **nicht automatisch initialisierten** Variablen die gewünschten **Startwerte** (**Initialwerte**) zuzuweisen, d. h. die Variablen zu **initialisieren**.

Variablen liegen während der Programmausführung in Speicherzellen des Arbeitsspeichers. Die Speicherzellen des Arbeitsspeichers (siehe Bild 1-19) sind durchnummeriert. In der Regel ist beim PC eine Speicherzelle 1 Byte[19] groß.

Die Nummern der Speicherzellen werden **Adressen** genannt.

Eine Variable kann natürlich mehrere Speicherzellen einnehmen. Dies visualisiert das folgende Bild:

Bild 1-19 Variable im Arbeitsspeicher

Während man in der Programmiersprache C sowohl über den Variablennamen als auch über die Adresse auf den in einer Variablen gespeicherten Wert zugreifen kann, kann man in Java nur über den Namen einer Variablen ihren Wert aus den Speicherzellen auslesen und verändern.

Damit wird ein häufiger Programmierfehler in C – der Zugriff auf eine falsche Adresse – verhindert.

Physische Adressen, d. h. Adressen des Arbeitsspeichers, werden in Java **vor dem Programmierer verborgen**.

[19] Ein **Byte** stellt eine Folge von 8 zusammengehörigen Bits dar.

Eine Variable hat in Java drei Kennzeichen:

- Variablennamen,
- Datentyp
- und Wert.

1.6 Datentypen

Ein **Datentyp** ist der Bauplan für eine Variable. Der Datentyp einer Variablen legt fest, welche Operationen auf einer Variablen möglich sind und wie die Darstellung (Repräsentation) der Variablen im Speicher des Rechners erfolgt.

Mit der Darstellung wird festgelegt, wie viele Bytes die Variable im Speicher einnimmt und welche Bedeutung jedes Bit der Darstellung hat.

1.6.1 Einfache Datentypen

Die Sprache Java stellt selbst standardmäßig einige Datentypen bereit, wie die einfachen Datentypen

- `int` zur Darstellung von ganzen Zahlen
- oder `float` zur Darstellung von Gleitpunktzahlen[20].

Kennzeichnend für einen **einfachen Datentyp** ist, dass sein **Wert** einfach im Sinne von **atomar** ist. Ein einfacher Datentyp kann nicht aus noch einfacheren Datentypen zusammengesetzt sein.

Datentypen, die der Compiler zur Verfügung stellt, sind **Standardtypen**.

Ein **Compiler** ist hierbei ein Programm, das Programme aus einer Sprache in eine andere Sprache übersetzt. Ein C-Compiler übersetzt z. B. ein in C geschriebenes Programm in Anweisungen eines sogenannten Maschinencodes, die der Prozessor direkt versteht.

1.6.1.1 Der Datentyp `int`

Der Datentyp `int` vertritt in Java-Programmen die **ganzen Zahlen** (**Integer-Zahlen**). Es gibt in Java jedoch noch weitere Ganzzahl-Datentypen. Sie unterscheiden sich vom Datentyp `int` durch ihre Repräsentation und damit auch durch ihren Wertebereich. Die `int`-Zahlen umfassen auf dem Computer einen endlichen Zahlenbereich, der nicht überschritten bzw. unterschritten werden darf. Dieser Bereich ist in Bild 1-20 dargestellt:

[20] **Gleitpunktzahlen** dienen zur näherungsweisen Darstellung von reellen Zahlen.

Bild 1-20 `int`-Zahlen

-2^{31}, d. h. `-2147483648`, und $2^{31} - 1$, d. h. `2147483647`, sind die Grenzen der `int`-Werte für Java auf jeder Maschine. Somit gilt für jede Zahl `x` vom Typ `int`:

`x` ist eine ganze Zahl, $-2147483648 \leq x \leq 2147483647$

Rechnet man mit einer Variablen `x` vom Typ `int`, so ist darauf zu achten, dass beim Rechnen nicht die Grenzen des Wertebereichs für `int`-Zahlen über- bzw. unterschritten werden. Wird beispielsweise `2 * x` berechnet und ist das Ergebnis `2 * x` größer als `2147483647` oder kleiner als `-2147483648`, so kommt es bei der Multiplikation zu einem Fehler, dem sogenannten **Zahlenüberlauf**. Hierauf muss der Programmierer selbst achten.

Ein **Zahlenüberlauf** wird in Java nicht durch eine Fehlermeldung oder eine Warnung angezeigt.

Meist wird in der Praxis so verfahren, dass ein Datentyp gewählt wird, der für die gängigen Anwendungen einen ausreichend großen Wertebereich hat. Sollte der Wertebereich dennoch nicht ausreichen, kann ein Gleitkomma-Datentyp oder die Klasse `BigInteger` eingesetzt werden. Die Klasse `BigInteger` ermöglicht beliebig lange Ganzzahlen. Sie wird als Bibliotheksklasse (siehe Kapitel 1.6.2) von Java zur Verfügung gestellt.

Die **Variablen** vom Typ `int` haben als Werte ganze Zahlen. Die Darstellung von `int`-Zahlen umfasst in Java 32 Bits. Dies entspricht einem Wertebereich von -2^{31} bis $+2^{31} - 1$.

1.6.1.2 Der Datentyp `float`

`float`-Zahlen entsprechen den rationalen und reellen Zahlen der Mathematik. Im Gegensatz zur Mathematik ist auf dem Rechner jedoch der Wertebereich endlich und die Genauigkeit der Darstellung begrenzt. `float`-Zahlen werden auf dem Rechner in der Regel als **Exponentialzahlen** in der **Form Mantisse * Basis** ^{Exponent} dargestellt (siehe Kapitel 6.2.3). Dabei wird sowohl die Mantisse als auch der Exponent mit Hilfe ganzer Zahlen dargestellt, wobei die Basis auf dem jeweiligen Rechner eine feste Zahl wie z. B. 2 oder 16 ist. Während in der Mathematik die reellen Zahlen unendlich dicht auf dem Zahlenstrahl liegen, haben die `float`-Zahlen, welche die reellen Zahlen auf dem Rechner vertreten, tatsächlich diskrete Abstände voneinander. Es ist im Allgemeinen also nicht möglich, Brüche, Dezimalzahlen, transzendente Zahlen oder die übrigen nicht-rationalen Zahlen wie z. B. die Quadratwurzel aus 2, $\sqrt{2}$,

exakt darzustellen. Werden `float`-Zahlen benutzt, so kommt es also in der Regel zu Rundungsfehlern. Wegen der Exponentialdarstellung werden die Rundungsfehler für große Zahlen größer, da die Abstände zwischen den im Rechner darstellbaren `float`-Zahlen zunehmen. Addiert man beispielsweise eine kleine Zahl `y` zu einer großen Zahl `x` und zieht anschließend die große Zahl `x` wieder ab, so erhält man meist nicht mehr den ursprünglichen Wert von `y`.

Die Variablen vom Typ `float` haben als Werte **reelle Zahlen**

Außer dem Typ `float` gibt es in Java noch einen weiteren Typ von reellen Zahlen, nämlich den Typ `double` mit erhöhter Rechengenauigkeit.

1.6.1.3 Operationen auf einfachen Datentypen

Allgemein ist ein einfacher Datentyp wie `int` oder `float` definiert durch seine **Wertemenge** und die **zulässigen Operationen** auf Ausdrücken dieses Datentyps. Im Folgenden soll der Datentyp `int` betrachtet werden. Der Wertebereich der `int`-Zahlen erstreckt sich über alle ganzen Zahlen von -2^{31} bis $2^{31} - 1$. Die für `int`-Zahlen möglichen Operationen sind:

Operator	Operanden	Ergebnis
Vorzeichenoperatoren +, – (unär)[21]	int	→ int
Binäre arithmetische Operatoren +, -, *, /, %	(int, int)	→ int
Vergleichsoperatoren ==, <, <=, >, >=, !=	(int, int)	→ boolean (Wahrheitswert)
Wertzuweisungsoperator =	int	→ int

Tabelle 1-3 Operationen für den Typ `int`

Die Bedeutung von Tabelle 1-3 wird am Beispiel der Operation

Operator	Operanden	Ergebnis
Vorzeichenoperatoren + (binär)	(int, int)	→ int

Tabelle 1-4 Binäres Plus

erklärt. Dieses Beispiel ist folgendermaßen zu lesen: Der **binäre Operator** + verknüpft zwei `int`-Werte als Operanden zu einem `int`-Wert als Ergebnis. In Tabelle 1-3 ist / der Operator der ganzzahligen Division, % der **Modulo-Operator**, der den Rest bei der ganzzahligen Division angibt, == der Vergleichsoperator "ist gleich", <= der "kleiner gleich"-Operator und != der Operator "ungleich". Das unäre + und – sind Vorzeichenoperatoren.

[21] Ein **unärer Operator** hat nur einen einzigen Operanden (siehe Kapitel 7.1).

1.6.2 Selbst definierte Datentypen

Neben den einfachen Datentypen gibt es in modernen Programmiersprachen auch sogenannte **selbst definierte Datentypen**. Selbst definierte Datentypen sind dem Compiler standardmäßig nicht bekannt. Wenn die Programmiersprache hierfür die Sprachmittel anbietet, so ist es dem Programmierer möglich, eigene Datentypen zu erfinden, die für die Modellierung einer Anwendung von Bedeutung sind, und diese dem Compiler bekannt zu machen.

Java bietet für selbst definierte Datentypen das Sprachkonstrukt der **Klasse** (`class`) oder des Aufzählungstyps (`enum`). Eine Klasse bildet ein Objekt der realen Welt in ein Schema ab, das der Compiler versteht, wobei ein Objekt beispielsweise ein Haus, ein Vertrag oder eine Firma sein kann – also prinzipiell jeder konkrete oder abstrakte Gegenstand, der für einen Menschen eine Bedeutung hat und den er sprachlich beschreiben kann. Will man beispielsweise eine Software für das Personalwesen einer Firma schreiben, so ist es zweckmäßig, einen selbst definierten Datentyp Mitarbeiter, d. h. eine Klasse `Mitarbeiter`, einzuführen.

Es gibt bereits eine große Anzahl vordefinierter Datentypen, die in der Java-Klassenbibliothek als sogenannte **Bibliotheksklassen** zur Verfügung gestellt werden.

Ein selbst definierter Datentyp kann durch eine Bibliothek zur Verfügung gestellt oder von einem Programmierer eingeführt werden.

1.6.3 Von den einfachen Datentypen zu den Klassen

Dieses Kapitel soll kurz aufzeigen, wie sich in der Geschichte der Programmiersprachen die Entwicklung von den einfachen Datentypen zu den Klassen vollzogen hat. Ein Programmieranfänger, der keine anderen Programmiersprachen kennt und sich allein auf das Erlernen der objektorientierten Programmierung mit Java fokussieren möchte, kann dieses Kapitel problemlos überspringen.

Einen Punkt mit ganzzahligen Koordinaten `x` und `y` kann man beschreiben durch zwei einzelne `int`-Variablen:

```
int x;
int y;
```

Geht man so vor, so muss sich der Programmierer natürlich im Kopf merken, dass `x` und `y` zum selben Punkt gehören. Kernighan und Ritchie führten in C den Begriff Struktur (`struct`) für einen zusammengesetzten, selbst definierten Datentyp ein. Damit war es in C möglich, durch

```
struct Punkt
{
   int x;  // Komponentenvariable x
   int y;  // Komponentenvariable y
}
```

einen Datentyp `struct Punkt` zu definieren, der die Komponenten `x` und `y` enthält. Jede Komponente eines Punktes kann dabei prinzipiell ihren eigenen Datentyp haben.

Eine Variable dieses Typs `struct Punkt` kann eingeführt werden durch

```
struct Punkt p;
```

`p` ist eine Strukturvariable, welche die beiden Komponenten `x` und `y` hat. Eine Festlegung der Koordinaten des Punktes kann erfolgen, indem man den Komponenten Werte zuweist, z. B. durch

```
p.x = 1;
p.y = 1;
```

Mit einer Struktur kann man einen Punkt `p` als Ganzes beschreiben. Der Zugriff auf die Komponenten `x` und `y` eines Punktes `p` erfolgt im Programm durch die Punktnotation `p.x` bzw. `p.y`.

Selbstverständlich kann man Variablen vom Datentyp `struct Punkt` mit Funktionen bearbeiten. Dabei werden in C die Funktionen jedoch außerhalb der Struktur definiert. Eine Funktion `get_x()` beispielsweise wird außerhalb der Struktur definiert durch

```
int get_x (struct Punkt p)
{
   return p.x;
}
```

Diese Funktion soll an dieser Stelle nicht komplett diskutiert werden. Wichtig ist jedoch, dass eine Funktion `get_x()`, die außerhalb der Struktur definiert werden muss, natürlich nicht wissen kann, auf welchem der Punkte sie arbeiten soll. Daher muss der zu bearbeitende Punkt in der Parameterliste – hier (`struct Punkt p`) – an die Funktion übergeben werden.

Den nächsten Fortschritt brachte **C mit Klassen**, das später dann **C++** hieß. In C mit Klassen wurde es möglich, dass eine Struktur nicht nur Daten, sondern auch Funktionen enthielt. Das obige Beispiel sieht dann so aus[22]:

```
struct Punkt
{
   int x;
   int y;

   int get_x()
   {
      return x;
   }
```

[22] Aus Symmetriegründen wurde hier auch eine Funktion `get_y()` aufgenommen.

```
    int get_y()
    {
        return y;
    }
}
```

Eine Variable `p` vom Typ `struct Punkt` wird durch

```
struct Punkt p;
```

vereinbart. Eine solche Variable repräsentiert in C++ ein Objekt. Dieses Objekt `p` hat als Komponenten zum einen die Komponentenvariablen `x` und `y` – also Daten – und zum anderen die Funktionen `get_x()` und `get_y()`.

In C++ werden die Daten-Komponenten **Attribute** genannt. In Java werden die Daten-Komponenten jedoch als **Datenfelder** bezeichnet.

Daher wird im Folgenden der Begriff Datenfeld verwendet. Funktions-Komponenten tragen in der objektorientierten Programmierung generell den Namen **Methoden**.

Bitte beachten Sie, dass `get_x()` jetzt keinen Übergabeparameter mehr benötigt. Warum? Die Methode `get_x()` wird jetzt innerhalb der Struktur definiert – das ist in C++ neu gegenüber C und typisch für die Objektorientierung[23] – und damit haben die Methoden automatisch Zugriff auf die Datenfelder `x` und `y`.

Eine klassische Struktur in C kann nur Daten als Komponenten haben. Eine Struktur in C++ kann Daten und Funktionen (Methoden) als Komponenten haben. Grundsätzlich gilt in der Objektorientierung, dass ein Objekt Daten und Methoden als Komponenten hat.

Das folgende Bild visualisiert Datenfelder und Methoden als Komponenten eines Objekts:

Bild 1-21 Methoden und Datenfelder als Komponenten eines Objektes

[23] In der Objektorientierung befinden sich Methoden und Datenfelder in der Kapsel einer Klasse.

Bild 1-21 zeigt ein Objekt des Datentyps `struct Punkt` mit den Datenfeldern `x` und `y` und den Methoden `get_x()`, `get_y()`, `set_x()` und `set_y()`. Die Methoden `set_x()` und `set_y()` dienen zum Setzen der Werte der Datenfelder `x` und `y`.

In der Objektorientierung haben die Methoden eines Objektes automatisch Zugriff auf die Datenfelder des Objektes.

Der Aufruf der Methoden `get_x()` und `get_y()` erfolgt wie beim Zugriff auf die Datenfelder über die Punktnotation, z. B. durch

```
p.get_x();
```

Schreibt man statt `struct Punkt` nun `class Punkt`:

```
class Punkt
{
   int x;
   int y;

   int get_x()
   {
      return x;
   }

   int get_y()
   {
      return y;
   }
}
```

so ist man bereits bei den Klassen angelangt. Der Unterschied zwischen `struct Punkt` und `class Punkt` in C++ ist nur der, dass für den Zugriff auf Komponenten für `struct` und `class` verschiedene Default-Zugriffsrechte gelten. Default-Zugriffsrechte sind die Zugriffsrechte, die automatisch bestehen, wenn nicht explizit bestimmte Zugriffsrechte angegeben werden.

Als Datentyp ist in C++ neben `struct Punkt` bzw. `class Punkt` auch einfach `Punkt` möglich, wobei der Datentyp `Punkt` identisch zum Datentyp `class Punkt` ist. Das hier vorgestellte Konzept der Klassen wird auch in Java übernommen. Klassen stellen einen Datentyp dar. Bei der Bildung von Variablen einer Klasse gibt es jedoch Unterschiede zwischen C++ und Java, die aber hier nicht diskutiert werden müssen.

1.7 Zusammenfassung

In Java kann man nur objektorientiert programmieren. Alle Programme in Java basieren von ihrem Aufbau her komplett auf Klassen. Nach dem ersten Kapitel soll man Programme editieren, kompilieren und ausführen können (siehe Kapitel **1.1**).

Ein Algorithmus einer Programmiersprache besteht aus präzise definierten Anweisungen, die vom Prozessor ausgeführt werden (siehe Kapitel **1.2**). Die Abarbeitungsreihenfolge von Anweisungen wird auch als Kontrollfluss bezeichnet. Unter einer Kontrollstruktur versteht man eine Anweisung, welche die Abarbeitungsreihenfolge von Anweisungen beeinflusst. Die Kontrollstrukturen der sequentiellen Programmierung sind Selektion, Iteration und Sequenz. Kontrollstrukturen, die der Strukturierten Programmierung genügen, haben einen einzigen Eingang und einen einzigen Ausgang.

Das Konzept der Strukturierten Programmierung umfasst die Modularisierung eines Programms in Haupt- und Unterprogramme und die Regel, dass jede Anweisung nur einen einzigen Ein- und Ausgang hat.

Variablen speichern Werte. Der von einer Variablen gespeicherte Wert kann durch eine Zuweisung mit einem neuen Wert überschrieben werden. Bei einer Zuweisung wird zuerst der Ausdruck rechts vom Zuweisungsoperator berechnet und der Wert dieses Ausdrucks der Variablen auf der linken Seite des Zuweisungsoperators zugewiesen.

Nassi-Shneiderman-Diagramme (siehe Kapitel **1.3**) erlauben die Spezifikation eines Programmes in Diagrammform, welches den Regeln der Strukturierten Programmierung genügt. Nassi-Shneiderman-Diagramme enthalten kein GOTO, sondern nur die Sprachmittel der Strukturierten Programmierung, insbesondere die Sequenz, Iteration und Selektion.

Zeichen und ihre rechnerinterne Darstellung in Form von Bits betrachtet das Kapitel **1.4**. Ein Zeichen ist ein von anderen Zeichen unterscheidbares Objekt, welches in einem bestimmten Zusammenhang eine definierte Bedeutung trägt. Von einem Alphabet spricht man, wenn der Zeichenvorrat eine strenge Ordnung aufweist.

Nach DIN 44300 ist ein Code eine Vorschrift für die eindeutige Zuordnung oder Abbildung der Zeichen eines Zeichenvorrats auf die Zeichen eines anderen Zeichenvorrats, die sogenannte Bildmenge.

In Kapitel **1.5** wird das Variablenkonzept der imperativen Programmiersprachen betrachtet. Eine Variable ist eine benannte Speicherstelle. Über den Variablennamen kann der Programmierer auf die entsprechende Speicherstelle zugreifen. Der Programmierer darf es nicht versäumen, den nicht automatisch initialisierten Variablen die gewünschten Startwerte (Initialwerte) zuzuweisen, d. h. die Variablen zu initialisieren. Physische Adressen, d. h. Adressen des Arbeitsspeichers, werden in Java vor dem Programmierer verborgen.

Eine Variable hat in Java drei Kennzeichen:

- Variablennamen,
- Datentyp
- und Wert.

In Kapitel **1.6** werden einfache Datentypen und Klassen in Java eingeführt.

Ein Datentyp ist der Bauplan für eine Variable. Der Datentyp einer Variablen legt fest, welche Operationen auf einer Variablen möglich sind und wie die Darstellung (Repräsentation) der Variablen im Speicher des Rechners erfolgt.

Die Variablen vom Typ `int` haben als Werte ganze Zahlen. Die Darstellung von `int`-Zahlen umfasst in Java 32 Bits. Dies entspricht einem Wertebereich von -2^{31} bis $+2^{31} - 1$. Die Variablen vom Typ `float` haben als Werte reelle Zahlen. Außer dem Typ `float` gibt es in Java noch einen weiteren Typ von reellen Zahlen, nämlich den Typ `double` mit erhöhter Rechengenauigkeit.

Ein selbst definierter Datentyp kann durch eine Bibliothek zur Verfügung gestellt oder von einem Programmierer eingeführt werden. Eine Klasse hat Datenfelder und Methoden. In der Objektorientierung haben die Methoden eines Objektes automatisch Zugriff auf die Datenfelder des Objektes.

1.8 Übungen

Aufgabe 1.1: Verständnisfragen

1.1.1 Was wird unter einem **Algorithmus** verstanden?
1.1.2 Was wird unter einer **Kontrollstruktur** verstanden?
1.1.3 Welche Kontrollstrukturen gibt es bei sequenziellen Abläufen?
1.1.4 Was ist ein **Datentyp**? Was ist eine **Variable**? Wie hängen Datentyp und Variablen zusammen?
1.1.5 Was ist der Unterschied zwischen einem zusammengesetzten Datentyp und einem einfachen Datentyp? Nennen Sie je ein Beispiel.
1.1.6 Welche Schritte sind notwendig, um vom Quellcode zu einem lauffähigen Java-Programm zu gelangen?
1.1.7 Was beinhalten `.class`-Dateien?
1.1.8 Was wird unter einem **Code** verstanden?

Aufgabe 1.2: Nassi-Shneiderman-Diagramme

1.2.1 Nassi-Shneiderman-Diagramm "Quadratzahlen"

Vervollständigen Sie das unvollständige Nassi-Shneiderman-Diagramm

für ein Programm, welches in einer (**äußeren**) **Schleife** Integer-Zahlen in eine Variable n einliest. Die Reaktion des Programms soll davon abhängen, ob der in die Variable eingelesene Wert positiv, negativ oder gleich Null ist. Treffen Sie die folgende Fallunterscheidung:

- Ist die eingelesene Zahl `n` größer als Null, so soll in einer inneren Schleife folgende Ausgabe erzeugt werden:

```
Zahl            Quadratzahl
1                  1
2                  4
.                  .
.                  .
.                  .
n                  n*n
```

- Ist die eingelesene Zahl `n` kleiner als Null, so soll ausgegeben werden: `Negative Zahl`.
- Ist die eingegebene Integer-Zahl `n` gleich Null, so soll das Programm (**die äußere Schleife**) abbrechen.

1.2.2 Nassi-Shneiderman-Diagramm "aufsteigende Ausgabe des Wertes zweier ganzer Zahlen"

Eine Funktion SORT soll die Werte zweier übergebener ganzer Zahlen sofort so ausgeben, dass der kleinere Wert immer links von der größeren Zahl ausgeben wird. Dies soll so lange gehen, bis für beide Zahlen der Wert `0` eingegeben wird. Dann soll der Abbruch erfolgen. Zeichnen Sie das Nassi-Shneiderman-Diagramm der Funktion SORT.

Kapitel 2

Objektorientierte Konzepte

2.1 Modellierung mit Klassen und Objekten
2.2 Kapselung, Abstraktion und Information Hiding
2.3 Individuelle und geteilte Daten von Objekten in Java
2.4 Abstraktion und Brechung der Komplexität
2.5 Erstes Programmbeispiel mit Objekten
2.6 Flughafen-Projekt
2.7 Zusammenfassung
2.8 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_2

2 Objektorientierte Konzepte

Menschen denken objektorientiert. Sie denken in Gegenständen wie Tisch, Stuhl, Person und so weiter. Objektorientierte Programmiersprachen bilden das menschliche Denken nach und enthalten Abstraktionen von Gegenständen.

Entscheidend für den objektorientierten Ansatz ist nicht das objektorientierte Programmieren allein, sondern das **Denken in Objekten** vom Start des Projektes an.

Dies wird dadurch erleichtert, dass bei den objektorientierten Techniken nicht nur in der Systemanalyse, sondern auch beim Systementwurf in Abstraktionen von Gegenständen der realen Welt gedacht wird. Der Kunde wird deshalb nicht durch das Denken in computertechnischen Konstrukten wie beispielsweise Haupt- und Unterprogrammen verwirrt. Selbst beim Programmieren spiegelt sich die Welt des Kunden in den Objekten noch wider.

Kapitel 2.1 stellt die Modellierung mit Klassen und Objekten vor. Bei Klassen wird zwischen den Klassen der Systemanalyse (Entity-Klassen, Kontrollklassen, Interface-Klassen) und den zusätzlichen Klassen des Systementwurfs, den technischen Klassen, differenziert. Entity-Klassen tragen Daten. Kontrollklassen sind prozedurale Klassen in Form von sogenannten Wrapper-Klassen[24]. Interface-Klassen kapseln gegenüber den Entity-Objekten und Kontrollobjekten die Aktoren, d. h. die Schnittstellen zu Fremdsystemen oder Geräten bzw. Nutzern. Ferner wird die Bedeutung von "makroskopischen" und "mikroskopischen" Zuständen für eine Anwendung erklärt sowie die Zusammenarbeit zwischen Objekten analysiert. Kapitel 2.2 diskutiert die für die Objektorientierung zentralen Begriffe der Kapselung, der Abstraktion und des Information Hiding. Kapitel 2.3 erläutert individuelle und geteilte Daten von Objekten. In Kapitel 2.4 wird allgemein auf die Abstraktion und Brechung der Komplexität eingegangen, welche Voraussetzung dafür sind, damit komplexe Projekte überhaupt erst beherrschbar werden. Ein erstes Programmbeispiel in Kapitel 2.5 verdeutlicht den Umgang mit Objekten. Abschließend stellt Kapitel 2.6 das Flughafen-Projekt vor, welches für einen anschaulichen und selbstständigen Einstieg in die objektorientierte Modellierung und Programmierung geeignet ist und in den weiteren Kapiteln des Buchs entsprechend dem Lernfortschritt weiter ausgebaut wird.

2.1 Modellierung mit Klassen und Objekten

Bevor ein neues System mit Hilfe einer Programmiersprache realisiert wird[25], braucht man die **Anforderungen** (**Requirements**) an das System und analysiert, welche

---

[24] Mit Hilfe einer Wrapper-Klasse wird ein nicht objektorientiertes Konstrukt – hier eine Prozedur – in die Gestalt einer Klasse gebracht (eingehüllt = engl. wrapped).

[25] Der Einfachheithalber wird hier vorausgesetzt, dass spezifikationsorientiert und nicht agil entwickelt wird.

Objekte bzw. Klassen[26] für das System aus Verarbeitungssicht von Bedeutung sind, um die Aufgaben des Systems zu erfüllen.

Man sucht in der Systemanalyse Entity-Klassen, Kontrollklassen und Interface-Klassen.

Dabei sucht man in der **Systemanalyse** zuerst die **Entity-Klassen**, dann die **Kontrollklassen** und schließlich die **Interface-Klassen**. Was Entity-Klassen, Kontrollklassen und Interface-Klassen sind, wird in diesem Kapitel noch erläutert.

Im **Systementwurf** macht man sich Gedanken darüber, wie man ein System in Systemteile gliedern kann, damit die durchzuführenden Aufgaben überschaubarer werden, entwickelt dabei die **Architektur des Systems** und sucht **technische Klassen**, die die Anwendung braucht, um lauffähig zu werden.

Durch das Schneiden eines Systems in Teilsysteme soll die Komplexität eines Teilsystems verringert werden.

Man legt dabei fest, welche Objekte bzw. Klassen welchem Systemteil zugeordnet werden und wie die Kommunikation von Systemteilen bzw. von Objekten/Klassen aussieht.

All diese Überlegungen werden in einer Systemspezifikation festgehalten und erst dann beginnt man, das System mit Hilfe einer Programmiersprache umzusetzen.

2.1.1 Problem- und Lösungsbereich

Kapitel 2.1.1.1 befasst sich mit dem Problembereich und Kapitel 2.1.1.2 mit dem Lösungsbereich eines Systems. Der Problembereich wird in der Systemanalyse behandelt und der Lösungsbereich beginnt beim Systementwurf.

In der **Systemanalyse** sucht man zunächst die Entity-Objekte bzw. die Entity-Klassen, die eine Anwendung braucht.

Dann studiert man die Anwendungsfälle eines Systems aus operationeller Sicht. Das heißt, man studiert, welche Verarbeitungen der eigentliche Nutzer – nicht der Systemadministrator – durchführen können muss. Die Leistungen eines Systems für einen Nutzer werden als **Anwendungsfälle** oder **Use Cases** der Anwendung bezeichnet. Aus der Betrachtung der Abläufe der einzelnen Anwendungsfälle entstehen Kontrollobjekte. Ein Kontrollobjekt steuert den Ablauf eines Anwendungsfalls.

[26] Eine Klasse stellt einen Datentyp dar, wobei ein Objekt eine Instanz (oder Variable oder Exemplar) der entsprechenden Klasse ist.

Entity-Klassen und Kontrollklassen leben auch beim Systementwurf weiter.

Interface-Objekte werden noch in der Systemanalyse zwischen Entity-/Kontrollobjekten und Aktoren eingeschoben.

Darauf wird im Folgenden noch eingegangen.

Aus einem Interface-Objekt der Systemanalyse wie beispielsweise einem MMI-Interface-Objekt können beim Entwurf viele technische Klassen entstehen wie z. B. verschiedene Swing- oder JavaFX-Klassen für die Mensch-Maschine-Schnittstelle. Es treten aber auch Objekte weiterer **technischer Klassen beim Systementwurf neu** auf wie z. B. Klassen für den Start-Up und Shut-Down des Systems oder Klassen für die Informationssicherheit.

2.1.1.1 Problembereich

Bei der objektorientierten Modellierung denkt man während der Systemanalyse im **Problembereich** (**Anwendungsbereich**, engl. problem domain) – also in der Begriffswelt des Kunden. Dies hat den großen Vorteil, dass der Kunde die Projektunterlagen leicht verstehen kann. Diese Harmonie wird in folgendem Bild visualisiert:

Bild 2-1 Kunde und Entwickler sprechen dieselbe Sprache. Sie verstehen sich!

Im Rahmen der **Systemanalyse** eines Systems wird zuerst auf Basis der Forderungen studiert, welche **Aufgaben** aus Verarbeitungssicht in dem System durchgeführt werden sollen und welche Objekte bzw. Klassen beim Ausführen der Anwendungsfälle des Systems eine Rolle spielen.

In der **Systemanalyse** ist man in der Welt der Logik der zu betrachtenden Aufgaben und betrachtet dabei noch gar keinen Rechner. Man befindet sich in der Systemanalyse also in einer **idealen Welt**, in der alles unendlich schnell abläuft und in der es keine technischen Fehler gibt.

Meistens kommen zu einem späteren Zeitpunkt der Systementwicklung noch neue Klassen hinzu, an die man zu Beginn der Entwicklung gar nicht gedacht hat. Das ist aber gar nicht so schlimm, da objektorientierte Systeme sehr stabil gegen nachträgliche Änderungen sind.

Entity-Klassen

Denkt man z. B. an ein Fluglotsensystem, so sind Klassen, auf die es ankommt, ganz bestimmt die einzelnen Flugzeuge, die Start- und Landebahnen und die Parkpositionen der Flugzeuge. Klassen haben Eigenschaften. Zu einer Startbahn gehört z. B. ihre Länge, der Belegt-Status und der Zeitraum der Belegung, die relative Windrichtung zur Startbahn und die entsprechende Windstärke. Mit Klassen bzw. ihren Objekten kann man in der Realität und im Programm etwas anfangen. So kann ein Objekt der Klasse `Parkposition` belegt oder freigegeben werden.

Bei der Modellierung des Problembereichs im Rahmen der Systemanalyse werden also **selbst definierte Datentypen** wie Flugzeug, Start- und Landebahn, Parkposition, Fluggesellschaft, Lotse etc. in der Form von **Klassen** eingeführt. Eine solche **Klasse** entspricht einem **Typ eines Gegenstands der realen Welt**. Dabei beschränkt man sich nicht nur auf konkrete Dinge. Auch **Wesen** wie z. B. Mitarbeiter und **Konzepte** wie z. B. Verträge werden darunter gesehen. Als Oberbegriff für konkrete Gegenstände, Wesen und Konzepte wird das Wort **Entität** verwendet.

Eine **Entität** hat im Rahmen des betrachteten Problems eine definierte Bedeutung. Sie kann einen Gegenstand, ein Wesen oder ein Konzept darstellen.

Es ist ein großer Vorteil der Objektorientierung, in Entitäten, d. h. in **Abstraktionen** von Gegenständen, Wesen oder Konzepten der realen Welt, zu denken. Eine Entität entspricht einem Objekt. Der Begriff **Entitätstyp** ist eine Analogie zum Begriff der Klasse. Etwas schwieriger ist, dass in der deutschen Sprache Instanzen (Objekte) und Typen gleich benannt werden. So wird ein konkretes Flugzeug gleichermaßen wie der Typ, d. h. die Klasse, einfach Flugzeug genannt.

Der Ansatz der Objektorientierung basiert darauf, **Objekte der realen** Welt mit Hilfe softwaretechnischer Mittel als **Entity-Objekte** abzubilden.

Eine **Entity-Klasse** ist der Typ von gleichartigen Objekten, die Entitäten der Realität entsprechen. Entity-Klassen sind Daten tragend, da sie Eigenschaften haben wie Länge, Gewicht oder Farbe. Entity-Klassen findet man durch "Peilen" in die Realität.

Bei der Abbildung einer Entität der Realität auf ein Objekt muss aber eine Abstraktion stattfinden, bei der das Unwesentliche weggelassen wird. Die typischen Eigenschaften bleiben übrig. Die unwesentlichen Eigenschaften werden ignoriert (Abstraktion).

Kontrollklassen

Entity-Klassen aber reichen nicht aus.

Beim Studium der Abläufe der einzelnen Anwendungsfälle treten in der Systemanalyse neue Klassen auf, die **Kontrollklassen**, welche die Abläufe der **Anwendungsfälle steuern**. Kontrollklassen sieht man nicht durch "Peilen" in die Realität.

Ein Kontrollobjekt besorgt sich die Daten von den Entity-Objekten. Ein Kontroll-Objekt stellt ein prozedurales Verhalten in Form des Typs einer Klasse dar.

Interface-Klassen

Damit Entity- und Kontrollobjekte unabhängig von der speziellen Ausprägung der Aktoren sind, werden kurz vor dem Übergang zum Systementwurf noch im Rahmen der Systemanalyse sogenannte **Interface-Objekte**, also Objekte von Interface-Klassen, zwischen den Aktoren und den Entity-/Kontrollobjekten eingeschoben.

Dieses Einschieben visualisiert das folgende Bild:

Bild 2-2 Interface-Objekte kapseln die Akteure

2.1.1.2 Lösungsbereich

Mit dem technischen System selbst befasst man sich erst beim **Systementwurf**.

Beim **Systementwurf** betritt der Entwickler den **Lösungsbereich**. Von Vorteil ist, dass die Entity- und Kontrollobjekte dabei unverändert in den Lösungsbereich übernommen werden können.

Wie schon erwähnt, entwickeln sich aus den Interface-Objekten der Systemanalyse technische Klassen. Das gilt z. B. für die Schnittstelle zum MMI, zur Datenbank oder für die Kommunikation der verschiedenen Rechner.

Es liegt aber auf der Hand, dass zu den Klassen des Problembereichs beim Eintritt in den Lösungsbereich eine ganze Reihe neuer technischer Klassen hinzutreten müssen. So werden spezielle technische Klassen benötigt, um das Programmsystem zu

starten oder zu stoppen, oder beispielsweise neue Klassen für die Informationssicherheit.

Beim **Systementwurf** treten **technische Klassen** zu den Entity- und Kontrollklassen der Systemanalyse hinzu.

Wie schon in Kapitel 2.1.1 erwähnt, können aus einer einzigen Interface-Klasse der Systemanalyse verschiedene Klassen beim Entwurf resultieren. So entstehen aus einer einzigen MMI-Interface-Klasse der Systemanalyse beim Entwurf beispielsweise zahlreiche Klassen verschiedener JavaFX-Komponenten für die Oberfläche.

2.1.2 Klassen als Baupläne für Objekte

Die Daten eines Systems werden in den sogenannten **Datenfeldern** von Klassen bzw. Objekten abgelegt. Datenfelder sind Komponentenvariablen einer Klasse bzw. eines Objektes, also Variablen, die ein Bestandteil, d. h. eine Komponente, einer Klasse bzw. eines Objektes sind. Was man mit einer Klasse bzw. einem Objekt alles tun kann, wird durch Methoden beschrieben.

Klassen stellen die **Baupläne für Objekte** dar. **Klassen** sind die **Datentypen**, die **Objekte** sind die **Variablen** (**Instanzen**) dieser Datentypen. Die Objekte werden gemäß den in den Klassen abgelegten Bauplänen erzeugt.

Im Folgenden soll eine Klasse `Punkt` betrachtet werden. Das folgende Bild zeigt die Klasse `Punkt` in der Notation von UML:

Punkt	
x : int	Klassenname `Punkt`
y : int	Datenfeld `x` vom Typ `int`
	Datenfeld `y` vom Typ `int`
zeichne( )	Methode `zeichne()`
verschiebe( )	Methode `verschiebe()`
loesche( )	Methode `loesche()`

Bild 2-3 Klasse `Punkt`

Die Klasse `Punkt` wird für einen grafischen Editor benötigt, der dazu dienen soll, Punkte und Figuren zu zeichnen. Ein Punkt, der gezeichnet wird, stellt ein Objekt oder eine Instanz der Klasse `Punkt` dar.

Eine Klasse trägt stets einen Klassennamen. Der Klassenname lautet hier `Punkt`. In einer Ebene hat ein Punkt 2 Koordinaten. Wählt man kartesische Koordinaten, so hat man die Abszisse auf der x-Achse, `x` genannt, und die Ordinate auf der y-Achse, `y` genannt. Punkte in der realen Welt sind beispielsweise `p1(1|1)`, `p2(2|3)` oder `p3(0|0)`. Was man mit einem Punkt alles anfangen kann, beschreiben die Methoden des Punktes. Man kann einen Punkt zeichnen (Methode `zeichne()`), einen

Punkt verschieben (Methode `verschiebe()`) und einen Punkt löschen (Methode `loesche()`).

Eine Klasse trägt einen **Klassennamen** und enthält **Datenfelder** und die **Methoden**, die auf diese Datenfelder zugreifen.

Bei der **Modellierung** werden aus den Punkten der realen Welt Objekte der Modellierung wie im Falle des Punktes `p1`, der in Bild 2-4 dargestellt ist:

Bild 2-4 Objekt `p1` der Klasse `Punkt`

Der Punkt `p1` ist ein bestimmtes konkretes Objekt der Klasse `Punkt`. Dies wird in UML notiert durch `p1 : Punkt`.[27] Dass die Koordinaten vom Typ `int` sind, wird angegeben durch `x : int` und `y : int`.

Ist der Objektname gefolgt von Doppelpunkt und Klassennamen in der Darstellung **unterstrichen**, so handelt es sich nach UML um ein **konkretes Objekt**. Ist der Objektname gefolgt von Doppelpunkt und Klassenname **nicht unterstrichen**, so handelt es sich nach UML um ein Objekt, an dessen Stelle viele konkrete Objekte treten können. Ein solches Objekt wird dann in UML als **Rolle oder Stellvertreter** betrachtet. An die Stelle einer Rolle bzw. eines Stellvertreters können all diejenigen Objekte treten, die der Rolle genügen.

In der Objektorientierung werden die Daten eines Objektes und die Methoden, welche die Daten eines Objektes ein- und ausgeben und miteinander verknüpfen, als eine zusammengehörige Einheit – als ein sogenanntes **Objekt** – betrachtet.

Ein Objekt der Klasse `Punkt` enthält alles, was man für die Darstellung eines Punktes braucht – seine Daten und seine Methoden – quasi in einer Kapsel. Die Methoden, welche auf die Daten zugreifen dürfen, stehen bei ihren Daten und "bewachen" diese. Damit werden die Daten vor einem direkten unbefugten Zugriff von außen geschützt.

Ein Objekt kann sinnbildlich mit einer Burg verglichen werden. Dies zeigt das folgende Bild:

[27] Da es sich um einen konkreten Punkt handelt, wird der Name des Punktes unterstrichen.

Bild 2-5 Daten stellen einen bewachten Goldschatz einer Burg dar

Die Daten stellen den Goldschatz der Burg dar. Die Daten werden durch die Wächter – die Methoden – bewacht und verwaltet. Eine Änderung der Daten oder ein Abfragen der Datenwerte kann in der Regel nur durch einen Auftrag an die Wächter, d. h. die Methoden, erfolgen.

Möchte also eine Methode auf die Daten eines anderen Objektes zugreifen, so kann sie dies im Allgemeinen nicht direkt tun. Sie muss eine der Methoden des anderen Objektes bitten, auf die gewünschten Daten zuzugreifen.

Die Methoden eines Objektes stellen also die Schnittstellen eines Objektes zu seiner Außenwelt dar. Dies zeigt das folgende Bild:

Bild 2-6 Daten und Methoden – die Bestandteile von Objekten

Methoden, die nach außen sichtbar sind, heißen auch **Export-Methoden**.

Der interne Aufbau eines Objektes, der in der Regel nach außen nicht sichtbar ist, besteht aus privaten Daten, privaten Methoden und der Implementierung der Rümpfe der nach außen sichtbaren Methodenschnittstellen. Private Methoden dienen als **Hilfsmethoden** (**Service-Methoden**) und können durch eine nach außen sichtbare Methode oder eine private Methode aufgerufen werden.

Die **Methoden** erfüllen die **Aufgaben**:

- **Werte** der Datenfelder eines Objektes **auszugeben**,
- **Werte** der Datenfelder zu **verändern**
- und mit Hilfe der in den Datenfeldern gespeicherten Werte **neue Ergebnisse zu berechnen**.

Die Methoden beschreiben, was man mit einem Objekt anfangen kann, d. h. wie sich ein Objekt zu seiner Umgebung verhält. Ein Objekt enthält damit auch sein **Verhalten**.

Datenfelder definieren die **Datenstruktur** der Objekte, die **Methoden** bestimmen das **Verhalten** der Objekte.

2.1.3 Zustände von Objekten

Ein Objekt hat – wie schon gesagt – einen Satz von Variablen (Datenfeldern) und Methoden, die zu ihm gehören.

Jedes Datenfeld hat Werte aus dem Wertebereich seines Typs. Der **Zustand** eines Objektes ist festgelegt durch den momentanen Wert seiner Datenfelder. Verändert werden kann der Zustand eines Objektes durch die Methoden des betreffenden Objektes. Diese Methoden führen ein Objekt von einem Zustand in einen anderen über.

Der Begriff des Zustandes eines Objektes kann noch etwas präzisiert werden:

Jede Kombination von Datenfeldwerten stellt einen Zustand dar, der in diesem Buch als **mikroskopischer Zustand** eines Objektes bezeichnet wird.

Ein Objekt kann sehr viele mikroskopische Zustände haben. Wichtig bei der Modellierung sind jedoch diejenigen Zustände eines Objektes, die für eine Anwendung eine Bedeutung haben. Diese Zustände werden in diesem Buch **makroskopische Zustände** genannt.

Um den Unterschied zwischen mikroskopischen und makroskopischen Zuständen zu erläutern, soll ein Fahrstuhl betrachtet werden. Im Fahrstuhl soll es einen Gewichtssensor geben, welcher das Gewicht der im Fahrstuhl befindlichen Personen feststellt. Jeder einzelne Wert des Gewichtssensors entspricht dann einem **mikroskopischen Zustand**. Für die Anwendung ist jedoch nur interessant, ob der Fahrstuhl überladen ist oder nicht. "Überladen" oder "nicht überladen" sind zwei **makroskopische Zustände**. Ist der Fahrstuhl "überladen", so fährt er nicht los. Es müssen so lange Fahrgäste aussteigen, bis er nicht mehr "überladen" ist. Weitere Beispiele für makroskopische Zustände eines Objektes Fahrstuhl sind "Warten auf Knopfdruck", "Türen schließen sich", "Fahren", "Türen öffnen sich". Generell sind makroskopische Zustände von Bedeutung, wenn man Zustandsübergänge von Objekten betrachtet,

z. B. dass ein Objekt Fahrstuhl beim Drücken eines Knopfes vom Zustand "Warten auf Knopfdruck" in den Zustand "Türen schließen sich" übergeht.

Ein makroskopischer Zustand resultiert aus Wechselwirkungen mit der Umgebung – im Falle eines Fahrstuhls z. B. aus Wechselwirkungen mit der Mechanik bei "Türen öffnen sich" oder mit dem Motor beim "Fahren". Andere Wechselwirkungen mit der Umgebung können das Warten auf ein Ereignis aus der Umgebung sein oder das Eintreten bestimmter Bedingungen.

In welchem Zustand sich ein Objekt dabei befindet, wird durch entsprechende Datenfelder des Objektes festgehalten.

2.1.4 Zusammenarbeit von Objekten

Objektorientierte Systeme erbringen ihre Leistungen im Wesentlichen durch

- einzelne Objekte
- oder durch das Zusammenwirken von Objekten – eine sogenannte Kollaboration.

Bild 2-7 symbolisiert einen Austausch von Nachrichten zwischen den Objekten:

Bild 2-7 Informationsaustausch zwischen Objekten in der Systemanalyse

Bitte beachten Sie, dass dieses Bild nur der Veranschaulichung dient und keineswegs UML[28]-konform ist. Die in Bild 2-7 eingezeichneten **Nachrichten** (Botschaften) der Systemanalyse entsprechen beim Entwurf und bei der Programmierung dem Aufruf von Java-Methoden.

[28] Die UML (Unified Modeling Language) stellt eine grafische Spezifikationssprache dar, um objektorientierte Systeme zu modellieren.

Eine funktionale Leistung eines Geschäftsprozesses auf einem Rechner zum Nutzen eines Anwenders wird als **Anwendungsfall** (**Use Case**) des Systems bezeichnet. Ein System kann viele Anwendungsfälle haben.

Natürlich gibt es auch Anwendungsfälle für Teilsysteme oder sogar für Klassen. Ein Anwendungsfall ist ein Service, den ein System oder ein Zerlegungsprodukt eines Systems einem Anwender zur Verfügung stellt.

Ganz zu Beginn der Systemanalyse eines Systems betrachtet man zunächst die Geschäftsprozesse des Systems und fällt dann die Entscheidung, welche Geschäftsprozesse bzw. welche Anteile davon durch den Rechner unterstützt werden sollen und welche nicht. Die Leistungen der Geschäftsprozesse oder Teilen von Geschäftsprozessen, die auf dem Rechner laufen und einem Anwender zu Gute kommen, sind – wie bereits erwähnt – die **Anwendungsfälle**. Im Rahmen der Systemanalyse bildet man die Wechselwirkungen zwischen den Objekten auf einen Nachrichtenaustausch zwischen den Objekten ab. Bildlich gesprochen heißt dies, dass die Objekte miteinander "reden".

Für jeden Use Case[29] wird beim Studium der Abläufe ein sogenanntes Kommunikations- oder Sequenzdiagramm erstellt. Kommunikationsdiagramme und Sequenzdiagramme können sowohl bei der Systemanalyse als auch beim Systementwurf erfolgreich zur Beschreibung von Abläufen verwendet werden.

Mit dem Aufruf einer Methode wird die Kontrolle an das Objekt, an das die Nachricht gerichtet ist, übergeben. Wie das Objekt handelt, ist Sache des Objektes.

Ein **Aufruf über Nachrichten bzw. Methoden** stellt eine **schwache Kopplung** zwischen Objekten dar, wenn ein Objekt nicht über zu viele Kanäle Nachrichten austauscht.

Eine schwache Kopplung hat den Vorteil, dass bei Änderungen eines Objektes die Rückwirkungen auf ein anderes Objekt gering bleiben. Aufgrund der schwachen Kopplung sind objektorientierte Systeme in der Regel auch leicht erweiterbar, sodass kein Schaden entsteht, wenn man zu Beginn eines Projektes nicht alle Klassen des Problembereichs sofort findet. Bei **stark gekoppelten Systemen** – wie z. B. bei einer Kopplung über globale Variablen in der klassischen prozeduralen Programmierung – führen Änderungen an einer Stelle oftmals zu einer Vielzahl von unliebsamen Folgeänderungen.

Die Menge der Nachrichten, auf die ein Objekt antworten kann, legt seine **Aufrufschnittstelle** fest. Sichtbar nach außen sind von einem Objekt nur seine Nachrichten bzw. seine Methodenköpfe, in anderen Worten seine Aufrufschnittstelle.

[29] Der Begriff Use Case wurde durch Ivar Jacobson [Jac92] weltweit bekannt gemacht.

2.2 Kapselung, Abstraktion und Information Hiding

Hinter den Mechanismen der objektorientierten Programmierung verbirgt sich ein Denkmodell, das sich von dem Modell der funktionsorientierten bzw. prozeduralen Programmierung (z. B. in der Sprache C) sehr stark unterscheidet. Dieses Denkmodell beruht im Kern darauf, dass man Daten und die Methoden, die auf ihnen arbeiten, nicht mehr getrennt behandelt, sondern als Einheit betrachtet. Die Begriffe Kapselung, Abstraktion und Information Hiding sind hierbei miteinander eng verwandt:

- **Kapselung**

  Der Begriff der Kapselung ist eines der wichtigsten Konzepte der objektorientierten Programmierung.

  Methoden und Daten eines Objekts verschmelzen bei der Kapselung zu einem einzigen Objekt.

  Daten und Methoden befinden sich also zusammen in einer Kapsel vom Typ einer Klasse. Der Begriff der Kapselung umfasst sowohl die Realisierung von Information Hiding zum Verbergen der inneren Details, d. h. der Daten und der Implementierung des Verhaltens, als auch die Realisierung von Abstraktion zur Sichtbarmachung und zum Ansprechen des Verhaltens der Kapsel durch eine Schnittstelle nach außen.

- **Abstraktion**

  Eine Schnittstelle einer Export-Methode, die außerhalb des Objekts aufrufbar ist, abstrahiert das Verhalten eines Objekts.

  Ein Objekt implementiert sein Verhalten in Schnittstellenmethoden, die außerhalb des Objekts als Abstraktion dieser Methoden sichtbar sind.

  Ein Objekt sollte nur über wohl definierte Schnittstellenmethoden (Export-Methoden) mit seiner Umwelt in Kontakt treten. Die Methodenköpfe der Exportmethoden eines Objekts stellen die Schnittstellen eines Objekts zur Außenwelt dar.

  Außer Export-Methoden kann es auch noch Methoden eines Objekts bzw. einer Klasse geben, die nach außen nicht sichtbar sind. Sie dienen als **Hilfsmethoden** (**Service-Methoden**) und können nur durch eine Methode des gleichen Objekts/der gleichen Klasse aufgerufen werden

- **Information Hiding**

  Die Daten einer Kapsel, die Rümpfe der Schnittstellenmethoden und die Service-Methoden sollen nach außen nicht direkt sichtbar sein.

Diese inneren Eigenschaften sollen vor der Außenwelt verborgen sein. Nur die Aufrufschnittstelle der Schnittstellenmethoden soll exportiert werden. Man spricht daher auch von **Information Hiding** oder **Geheimnisprinzip**. Dieses Prinzip bedeutet, dass ein Teilsystem (hier ein Objekt) nichts von den Implementierungsentscheidungen eines anderen Teilsystems wissen darf. Damit wird vermieden, dass ein Teilsystem von der Implementierung eines anderen Teilsystems abhängt.

Kapselung, Abstraktion und Information Hiding sind im folgenden Bild dargestellt:

Bild 2-8 Daten und Methoden – die Bestandteile von Objekten

Die Operationen bzw. Methoden auf den Daten eines Objekts werden als eine "innere Fähigkeit" eines Objekts betrachtet. Wie die Operationen im Detail ablaufen, ist von außen nicht sichtbar. Von außen ist nur sichtbar, wie eine Methode aufgerufen werden kann.

Die Prinzipien der Kapselung, der Abstraktion und des Information Hiding sind also eng miteinander verknüpft. Die Außenwelt soll keine Möglichkeit haben, Daten im Inneren eines Objektes direkt zu lesen oder sogar zu verändern und so möglicherweise unzulässige Zustände herbeizuführen.

Das **Verstecken sämtlicher Daten und der Implementierung der Methoden** in einer "**Kapsel**" und die Durchführung der Kommunikation mit der Außenwelt durch eigene **Schnittstellenmethoden** bzw. Export-Methoden bringt dem **Programmierer** den **Vorteil**, dass er bei der Implementierung der Algorithmen in den Methoden und bei den Datenstrukturen des Objektes sehr viele Freiheiten hat. Dem **Benutzer** bringt dies im Gegenzug den Vorteil, dass er sich nicht um interne Details (Datenstrukturen, Algorithmen) kümmern muss.

Ein Benutzer kann dann immer die neueste Version eines Objektes verwenden, da er nicht vom speziellen inneren Aufbau des Objektes abhängig ist. Der Programmierer einer Klasse kann deren inneren Aufbau immer wieder optimieren, ohne Komplikationen befürchten zu müssen. Nur die Schnittstellen müssen gleich bleiben. Bereits an dieser Stelle kann man erkennen, wie wichtig die Schnittstellen sind. Es ist also unbedingt nötig, diese sorgfältig zu entwerfen.

Ein Objekt darf also mit einem anderen Objekt nur über wohl definierte Schnittstellen Informationen austauschen und keine Kenntnisse über den inneren Aufbau seines Partners haben. Damit haben Änderungen im Inneren eines Objektes keine Auswirkungen auf andere Objekte, solange die Schnittstellen stabil bleiben. Um trotzdem ein Höchstmaß an Flexibilität zu gewährleisten, ist es jedoch immer noch möglich, Teile eines Objektes so zu vereinbaren, dass sie ohne Weiteres direkt von außen zugänglich sind. Zumindest für die Schnittstellenmethoden (Export-Methoden) muss diese Eigenschaft in jedem Fall zutreffen.

2.3 Individuelle und geteilte Daten von Objekten in Java

Bild 2-9 zeigt drei Objekte der Klasse `Punkt` und die Klasse `Punkt` selbst. Die Namen der Referenzen auf diese Objekte lauten `p1`, `p2` und `p3`. Ein jeder Punkt trägt seine individuellen Koordinaten. So hat der durch `p1` referenzierte Punkt in diesem Beispiel die Koordinaten `x` ist gleich `1` und `y` ist gleich `1`. Diese Werte sind "persönliche" Eigenschaften dieses Punktes, sie gehören zu dem Punkt-Objekt, in anderen Worten zu der Punkt-Instanz.

"Persönliche" Variablen eines Punktes werden als **Instanzvariablen** bezeichnet.

Bei jedem Punkt-Objekt gibt es die Methoden `zeichne()`, `verschiebe()` und `loesche()`. Diese Methoden arbeiten auf den Instanzvariablen und werden als **Instanzmethoden** bezeichnet. Hier das bereits erwähnte Bild:

Bild 2-9 Drei Instanzen der Klasse `Punkt` und die Klasse `Punkt` selbst

Die Anzahl der erzeugten Punkte ist jedoch keine Eigenschaft eines individuellen Punktes, sondern ist eine Eigenschaft der Menge aller Punkte. Die Variable `punktAnzahl` kann aus Symmetriegründen bei keinem der Objekte der Klasse `Punkt` untergebracht werden, da alle Punkte von Haus aus vollkommen gleichberechtigt sind. Es liegt deshalb nahe, dass eine Variable, die eine Eigenschaft aller Instanzen

derselben Klasse darstellt, in der Klasse selbst angelegt wird – im betrachteten Beispiel also in der Klasse `Punkt`. Dass der Speicherort einer Variablen die Klasse selbst ist, wird durch den Namen **Klassenvariable** zum Ausdruck gebracht. Die Klassenmethode `getPunktAnzahl()` gibt bei ihrem Aufruf den Wert der Klassenvariablen `punktAnzahl` an den Aufrufer zurück. Klassenvariablen und Klassenmethoden werden nach UML unterstrichen.

Klassenvariablen stellen globale Variablen für alle Objekte (Instanzen) einer Klasse dar.

Im Rahmen der Objektorientierung werden Variablen, die allen Instanzen einer Klasse gemeinsam sind, als **Klassenvariablen** bezeichnet. Klassenvariablen werden in der Klasse selbst als einfach vorhandene Variablen, die jedoch allen Objekten dieser Klasse zur Verfügung stehen, angelegt. Variablen, die bei jedem Objekt – also bei jeder Instanz – angelegt werden und bei jedem Objekt eine individuelle Ausprägung annehmen können, werden als **Instanzvariablen** bezeichnet.

Üblicherweise arbeiten **Instanzmethoden** auf **Instanzvariablen**. **Klassenmethoden** sind dazu da, um auf **Klassenvariablen** zu arbeiten. Da Klassenvariablen globale Variablen für alle Instanzen einer Klasse sind, kann eine **Instanzmethode** nicht nur auf Instanzvariablen, sondern **auch auf Klassenvariablen zugreifen**.

2.4 Abstraktion und Brechung der Komplexität

Die Kunst der Abstraktion ist, das Wesentliche zu erkennen und das Unwesentliche wegzulassen. Abstraktion ist in allen Entwicklungsphasen ein effizientes Mittel, um die Komplexität eines Systems zu verringern.

Abstraktion bedeutet immer Konzentration auf das Wesentliche und das Weglassen des Unwesentlichen.

In den Kapiteln 2.4.1 bis 2.4.4 sollen verschiedene Formen der Abstraktion betrachtet werden.

2.4.1 Abstraktion zur Abgrenzung des Problembereichs

Abstrahieren beginnt schon beim Projektstart, bei dem man entscheiden muss, welcher Ausschnitt der realen Welt den **Problembereich** darstellt, der analysiert werden soll. Der sogenannte **Problembereich** (engl. **problem domain**) ist der Bereich der zu untersuchenden Aufgaben. Er ist derjenige Teil der realen Welt, der später durch die zu realisierende Software abgedeckt werden soll. Den Ausschnitt des Problembereichs symbolisiert das folgende Bild:

Bild 2-10 Problembereich als relevanter Ausschnitt der realen Welt

2.4.2 Abstraktion zum Finden der Objekte des Problembereichs

Hat man den Problembereich umrissen, so gilt es, die Objekte des Problembereichs zu finden. Das sind diejenigen Objekte der realen Welt, die auf Objekte der Programmiersprache abgebildet werden sollen. Ob ein Objekt des Problembereichs in ein Objekt des Programmsystems überführt werden soll, wird durch Beantwortung der Frage entschieden, ob es notwendig ist, Informationen über dieses Objekt im System zu führen. Die Beschränkung auf das wirklich Notwendige bedeutet wieder eine Abstraktion.

2.4.3 Abstraktionen für Verhalten und Eigenschaften von Klassen

Hat man erkannt, welche Objekte bzw. Klassen man braucht, so ist zu abstrahieren, welche Informationen über eine Klasse erforderlich sind oder nicht, und über welche Methoden man auf die Daten eines Objekts bzw. einer Klasse zugreifen will.

Dies hängt ganz entscheidend von der jeweiligen Anwendung ab! So wird etwa ein Autohersteller zu einem Objekt der Klasse `Autotyp` alle Komponenten des betreffenden Autotyps speichern, um beispielsweise die richtigen Ersatzteile liefern zu können. Das Finanzamt, das auf die Autos die Kraftfahrzeugsteuer erhebt, interessiert sich für wesentlich weniger Merkmale eines Autotyps, nämlich nur für diejenigen, die steuerrelevant sind wie z. B. der Schadstoffausstoß.

2.4.4 Abstraktion zur Bildung von Hierarchien

Abstraktion und Information Hiding sind effiziente Mittel, um mit der Komplexität fertig zu werden. Ein weiteres Mittel, um Übersicht zu schaffen, ist die Bildung von Hierarchien. Die Bildung von Hierarchien hat aber auch mit der Bildung von Abstraktionen zu tun. Dabei gibt es in der Objektorientierung zwei wichtige Hierarchien:

- die **Vererbungshierarchie**[30] (auch **"kind of"-Hierarchie** oder **"is a"-Hierarchie** genannt)
- und die **Zerlegungshierarchie** (auch **"part of"-Hierarchie** genannt).

2.4.4.1 Vererbungshierarchie

Bei der **Vererbungshierarchie** (siehe Bild 2-11) werden die Klassen **in Abstraktionsebenen** angeordnet.

[30] Das Konzept der Vererbung wird detailliert in Kapitel 11 besprochen. Bei der Vererbung erbt eine abgeleitete Klasse die Daten und Methoden ihrer Basisklasse.

Geht man in einer Vererbungshierarchie von unten nach oben, so spricht man von **Generalisierung**, geht man von oben nach unten, so kommt man zu spezielleren Klassen – man spricht von **Spezialisierung**.

Die jeweils tiefer stehende Klasse ist eine Spezialisierung der Klasse, von der sie abgeleitet ist. So sind in Bild 2-11 die Klassen `B`, `C` und `D` verschiedene Spezialisierungen ihrer Basisklasse `A`, in anderen Worten, jede der Klassen `B`, `C` und `D` ist von der Klasse `A` abgeleitet. Hier das bereits erwähnte Bild:

Bild 2-11 Vererbungshierarchie

Typisch für solche Vererbungshierarchien sind die Klassifikationsschemen[31] der Tiere und Pflanzen in der Biologie. Siehe hierzu Bild 2-12:

Bild 2-12 Klassifikation von Spinnentieren

Eine Kreuzspinne, Hausspinne und Springspinne ist eine Spinne. Ein Skorpion, eine Milbe und eine Spinne ist ein Spinnentier. Spinnentiere wiederum gehören zum Stamm der Gliederfüßler und stehen auf einer Ebene mit Krebstieren, Tausendfüßlern und Insekten [Sch64].

[31] Während man in der Objektorientierung nur von Klassen redet, werden in der Biologie die Namen Stamm, Klasse, Unterklasse, Ordnung, Familie, Gattung, Art und Rasse verwendet.

2.4.4.2 Zerlegungshierarchie

Bei der **Zerlegungshierarchie** (siehe Bild 2-13) hat man auch verschiedene **Abstraktionsebenen**. Sieht man nur das Ganze, so ist man eine Ebene höher, als wenn man die Teile betrachtet. Ein Objekt kann als Datenfelder andere Objekte in Form einer **Komposition** oder einer **Aggregation** enthalten.

Komposition und Aggregation unterscheiden sich bezüglich der **Lebensdauer** des zusammengesetzten Objektes und seiner Komponenten. Bei einer **Komposition** ist die Lebensdauer des zusammengesetzten Objektes identisch zur Lebensdauer der Komponenten. Das zusammengesetzte Objekt ist im Falle der Komposition für den Life-Cycle (create, delete) des eingeschlossenen Objektes verantwortlich. Bei einer **Aggregation** sind die Lebensdauern eines aggregierenden Objekts und der aggregierten Objekte entkoppelt – somit können beispielsweise die Teile auch länger leben als das Ganze.

Bei einer Komposition hat ein Teil einen einzigen "Besitzer", bei einer Aggregation können mehrere aggregierende Objekte auf ein und dasselbe aggregierte Objekt zeigen.

Ein Beispiel für eine Aggregation ist ein Ordner. Ein Ordner und sein Inhalt, die aggregierten Seiten, können verschieden lange leben. Man kann die Seiten, d. h. die Komponenten, früher oder später als den Ordner wegwerfen oder den Ordner samt Inhalt zur selben Zeit. Ein anderes Beispiel ist ein Auto und seine Räder. Die Räder sind nur aggregiert. Man kann die Felgen mit den Sommerreifen gegen die Felgen mit den Winterreifen austauschen oder die Winterreifen und Felgen verkaufen, wenn das Auto verschrottet wird. Die Räder können dann an ein anderes Auto montiert werden.

Bei einer Komposition sind "Groß"- und "Klein"-Objekt fest "verschweißt". Ein Beispiel für eine Komposition ist ein Buch. Hier sind die Seiten und der Umschlag fest verklebt. Das Ganze, das Buch und die Komponenten, die Seiten, leben gleich lange. Sie können nur gemeinsam vernichtet werden.

Programmiertechnisch kann man eine Aggregation in Java mit Hilfe von Referenzen erzeugen. Ein "Groß"-Objekt enthält dann als Datenfeld eine Variable, die auf das "Klein"-Objekt zeigt. Eine solche Variable heißt **Referenzvariable**, da sie einen Verweis (eine Referenz) enthält.

Eine **Referenz** auf ein Objekt **enthält als Wert die logische Adresse** des Objektes, auf das die Referenz zeigt.

Eine Komposition kann man programmiertechnisch erzeugen, wenn ein "Groß"-Objekt als Datenfeld ein "Klein"-Objekt fest enthält. Datenfelder sind bekanntermaßen mit ihrem Objekt untrennbar verbunden. Während C++ sowohl die Komposition als auch die Aggregation zulässt, erlaubt Java nur die Aggregation, solange man nicht

mit sogenannten geschachtelten Klassen arbeitet[32]. Das folgende Bild zeigt eine Aggregationshierarchie:

Bild 2-13 Aggregationshierarchie

Ein Objekt der Klasse `A` aggregiert Objekte der Klassen `B`, `C` und `D`. Wird ein Objekt der Klasse `A` zerstört, so können dennoch die Objekte der Klassen `B`, `C` und `D` weiterleben. Damit ist die Lebensdauer von aggregierenden Objekten und aggregierten Objekten entkoppelt. Dies wird in Bild 2-13 durch die nicht ausgefüllte Raute ausgedrückt. Eine Komposition im Gegensatz zu einer Aggregation würde in Bild 2-13 durch eine ausgefüllte Raute veranschaulicht werden.

2.5 Erstes Programmbeispiel mit Objekten

In diesem Kapitel soll wie in Kapitel 2.1.2 wieder die Klasse `Punkt` als Anwendungsbeispiel herangezogen werden. Dabei soll eine Anwendung geschrieben werden, in der Punkte, d. h. Objekte der Klasse `Punkt` erzeugt werden. Betrachtet man den Punkt als Objekt, so beschreiben dessen Methoden, was man mit dem Objekt alles anfangen kann. So kann man beispielsweise

- die Koordinaten eines Punktes festlegen,
- die Koordinaten eines Punktes abfragen,
- einen Punkt auf dem Bildschirm zeichnen,
- einen Punkt auf dem Bildschirm verschieben
- und einen Punkt auf dem Bildschirm löschen.

Damit hat ein zweidimensionaler Punkt mit ganzzahligen Koordinaten die Datenfelder

```
int x,
int y
```

und die Methoden

```
setX(),
setY(),
getX(),
getY(),
zeichne(),
verschiebe()
```
und `loesche()`.

[32] Siehe Kapitel 15

Diese Eigenschaften – Datenfelder und Methoden – gelten für jeden beliebigen Punkt. Diese Gemeinsamkeit wird in der Objektorientierung durch die Klasse `Punkt` dargestellt. Ein jeder Punkt wird nach dem Bauplan dieser Klasse `Punkt` gebaut.

Um möglichst einfach zu beginnen, sollen die Methoden `zeichne()`, `verschiebe()` und `loesche()` hier außer Acht gelassen werden. Unter dieser Annahme hat die Klasse `Punkt` das im Folgenden gezeigte Aussehen, wobei es hier an dieser Stelle nicht wichtig ist, Schlüsselwörter wie `public`, `return`, `void` usw. zu verstehen. Vielmehr soll auf den **prinzipiellen Aufbau** geachtet werden.

Hier die Klasse `Punkt`:

```
// Datei: Punkt.java

// Deklaration der Klasse Punkt. Dem Compiler wird gesagt, dass es
// eine Klasse Punkt gibt.
public class Punkt
{
   private int x;                    // Datenfelder für die x- und
   private int y;                    // y-Koordinate vom Typ int

   public int getX()                 // eine Methode, um den Wert
   {                                 // von x abzuholen
      return x;
   }

   public int getY()                 // eine Methode, um den Wert
   {                                 // von y abzuholen
      return y;
   }

   public void setX (int i)          // eine Methode, um den Wert
   {                                 // von x zu setzen
      x = i;
   }

   public void setY (int i)          // eine Methode, um den Wert
   {                                 // von y zu setzen
      y = i;
   }

   // Mit main() beginnt eine Java-Anwendung ihre Ausführung.
   public static void main (String[] args)
   {
      Punkt p = new Punkt();         // Punkt erzeugen
      p.setX (3);                    // Aufruf der Methode setX()
      p.setY (2);                    // Aufruf der Methode setY()

      System.out.println ("Die Koordinaten des Punktes sind: ");
      System.out.println (p.getX()); // Wert von x wird ausgegeben
      System.out.println (p.getY()); // Wert von y wird ausgegeben
   }
}
```

Die Ausgabe des Programmes ist:

```
Die Koordinaten des Punktes sind:
3
2
```

Beachten Sie, dass eine Erläuterung hinter einem **Doppelschrägstrich** ein sogenannter **Kommentar** ist. Ein Kommentar dient nur zur Dokumentation und hat auf die Ausführung eines Programms keinen Einfluss.

Es ist nicht das Ziel dieses Beispiels, alle Details zu betrachten. Zuallererst soll zum Ausdruck kommen, dass eine Klasse einen **Klassennamen** – hier `Punkt` – hat und aus **Datenfeldern** und **Methoden** aufgebaut ist. Ferner wird hier gezeigt, dass zu einer **Java-Anwendung** eine Methode `main()` gehört.

In der Methode `main()` werden Objekte geschaffen und Methoden aufgerufen.

Hat man eine Anwendung aus mehreren Klassen, deren Objekte gemeinsam die Anwendung realisieren, so schreibt man normalerweise eine separate Klasse mit der Methode `main()`, um die Anwendung zu starten.

Da Objekte erst in der Methode `main()` selbst angelegt werden, kann die Methode `main()` nicht zu einem Objekt, sondern muss zu der Klasse selbst gehören.

Das bedeutet, dass `main()` eine Klassenmethode ist.

Klassenvariablen und **Klassenmethoden** erhalten bei Java das Schlüsselwort `static`.

Eine Klassenmethode kann auch aufgerufen werden, ohne dass ein Objekt dieser Klasse existiert (siehe Kapitel 2.3).

Die Methode `main()` muss auch stets `public`, d. h. für alle sichtbar, sein. Sonst kann der Java-Interpreter, der das Programm starten und ausführen soll, nicht auf `main()` zugreifen.

Methoden sind in der Regel `public` und Datenfelder `private`.

Das Schlüsselwort `public` ist ein sogenannter **Zugriffsmodifikator**, der angibt, dass diese Methode ungeschützt ist und von allen anderen Klassen bzw. Objekten aus aufgerufen werden kann. Die Datenfelder eines Objektes sollen in der Regel

nicht ungeschützt der Außenwelt ausgeliefert sein. Daher werden sie in der Regel mit dem Modifikator `private` versehen. Ein Außenstehender wie z. B. eine Testklasse kann nur "geordnet", d. h. über speziell vorgesehene Methoden, auf die Datenfelder zugreifen.

Eine **Klasse** kann als eine abstrakte Beschreibung eines Objektes angesehen werden. Wird mit Hilfe dieser Beschreibung ein Objekt im Arbeitsspeicher angelegt, so spricht man von der **Instanziierung einer Klasse**. Man erhält ein arbeitsfähiges Exemplar im Speicher. Ein Objekt stellt eine **Instanz** oder ein **Exemplar** dar.

Mit der Klassenbeschreibung wird ein Schema zur Bildung von Objekten dieser Klasse vereinbart. Dieses Schema enthält:

- den **Namen** der Klasse,
- die **Datenfelder** dieser Klasse
- und die **Methoden** dieser Klasse.

Ein Objekt ist eine **Instanz** einer Klasse. Ein Objekt hat eigene, individuelle Werte für die Datenfelder. Ein Objekt teilt Datenfeldnamen und Methodennamen mit anderen Objekten der Klasse.

Alle Objekte besitzen eine eigene **Identität**, sind also eigene Wesen, selbst wenn ihre Datenwerte (Werte der Instanzvariablen) zufälligerweise identisch sind.

Ein Objekt hat in sich eine implizite Referenz, d. h. einen versteckten Verweis, auf seine eigene Klasse. Es "weiß", zu welcher Klasse es gehört.

2.6 Flughafen-Projekt

Dieses Kapitel richtet sich zum einen an diejenigen, die daran interessiert sind, wie man eigentlich auf die Klassen kommt, die in den Programmen verwendet werden. Zum anderen wendet es sich an all jene, die das hier vorgestellte Lernprojekt Flughafen selbst durchführen wollen. Alle Teilschritte des Projektes befinden sich auf der Internetseite des Verlags unter www.springer.com. Die Aufgaben der einzelnen Teilschritte zur Realisierung des Flughafen-Informationssystems sind in den Übungskapiteln zu finden. So kann das Flughafen-Informationssystem begleitend zum Studium des Buchs erstellt werden.

Im folgenden Beispiel soll prinzipiell gezeigt werden, wie Klassen gefunden werden können. Es wird angenommen, dass Sie – der Leser – Mitglied eines Teams von Systemanalytikern sind, die von einem Flughafenbetreiber die in Kapitel 2.6.1 dargestellte Ausschreibung[33] erhalten würden. Ausgehend von dieser Ausschreibung sollen dann in Kapitel 2.6.2 erste Klassen gefunden werden, und zwar Klassen, die Abstraktionen von Entitäten der realen Welt sind.

[33] Wie Sie wissen, ist das "echte Leben" stets wesentlich komplexer als einfache Schulungsaufgaben. Sehen Sie bitte großzügig darüber hinweg, dass die "Ausschreibung" erhebliche Lücken aufweist.

2.6.1 Ausschreibung für ein neues Flughafen-Informationssystem

Im Folgenden eine Skizze des Flughafens:

Bild 2-14 Skizze des Flughafens

Der Flughafen verfügt über 4 getrennte Bahnen, wovon eine jede als Lande- oder als Startbahn benutzt werden kann. Die Steuerung der Flugzeuge soll über Sprechfunk vom Lotsen an die Piloten erfolgen. Die Steuerung über Sprechfunk ist ein eigenständiges System und nicht Bestandteil des neuen Informationssystems. Auf dem Vorfeld werden die Flugzeuge durch ein "Follow me"-Fahrzeug geleitet. Die Verwaltung des Vorfeldes ist ebenfalls nicht Teil des neuen Systems.

Das neue Informationssystem für einen Flughafen soll es den **Fluglotsen** ermöglichen, die Landung und den Start von Flugzeugen rechnergestützt zu überwachen. Ferner sollen die **Angestellten der Flughafenverwaltung** bei der Erhebung der Start- und Landegebühren unterstützt werden.

Das **Informationssystem** soll Daten über die Position der Flugzeuge von verschiedenen Sensoren erhalten. Diese **Sensoren** sind in folgender Tabelle enthalten:

Radar des Flughafens	liefert Positionsdaten der Flugzeuge in der Luft
Positionsmelder des Flugzeugs	liefert über eine automatisierte Funkschnittstelle die Positionsdaten eines Flugzeugs in der Luft
Flugplatzsensoren	liefern Positionen der Flugzeuge am Boden
Parksensoren	liefern Belegungsstatus der Parkpositionen

Bild 2-15 Sensoren des Flughafen-Informationssystems

Um die Bahnkurve eines Flugzeugs grafisch darstellen zu können, sollen alle Positionsdaten eines Flugzeugs im System geführt werden. Für ein Objekt der Klasse `Flugzeug` selbst sollen Soll-Zeitpunkt der Landung, Ist-Zeitpunkt der Landung, Soll-Zeitpunkt des Starts, Ist-Zeitpunkt des Starts, Landebahn, Parkposition, Startbahn, betreuender Lotse, Flugzeugtyp und Fluggesellschaft gespeichert werden können. Jede Start-/Landebahn sowie jede Parkposition soll im System gespeichert werden und von den Lotsen belegt werden können. Damit soll sichergestellt werden, dass

ein solches Objekt von den Lotsen nicht zur selben Zeit mehrfach zugeteilt wird. Alle Fluggesellschaften sollen ebenfalls im System geführt werden.

Das System soll einen Lotsen bei den folgenden Aufgaben unterstützen:

- **Luftlage anzeigen**

  Positionsdaten von Flugzeugen in der Luft am Bildschirm kontrollieren.

- **Bodenlage anzeigen**

  Positionsdaten von Flugzeugen am Boden am Bildschirm kontrollieren.

- **Zeitplan anfordern**

  Starts und Landungen mit Hilfe eines Zeitplans planen.

- **Landung durchführen**

  Ein Flugzeug kann sich bei einem Lotsen für eine Landung anmelden. Wird die Landung nicht wegen schlechten Wetters verweigert, so trägt der Lotse Flugzeugtyp, Fluggesellschaft und sich selbst als betreuender Lotse in das System ein.

  Ist die Fluggesellschaft des sich im Landeanflug befindenden Flugzeugs dem System noch nicht bekannt, so wird sie vom Lotsen in das System aufgenommen (Name, Adresse). Der Lotse verschafft sich einen Überblick über die Start-/Landebahnen und trägt die von ihm zugeteilte Landebahn und den Soll-Zeitpunkt der Landung in das System ein. Der Lotse schaut sich die vergebenen und freien Parkpositionen an und trägt die von ihm vergebene Parkposition in das System ein. Ist keine Landebahn oder Parkposition frei, wird eine Warteschleife angeordnet, ansonsten wird der Landeanflug freigegeben. Hat das Flugzeug die Parkposition erreicht, so trägt der Lotse den Ist-Zeitpunkt der Landung in das System ein und gibt die Landebahn wieder frei.

  Treten während der Landung zwingende Gründe für einen Abbruch der Landung ein, so hat der Lotse den Abbruch unter Angabe von Gründen in das System einzutragen.

- **Start durchführen**

  Im Rahmen der Startzuweisung lässt sich der Lotse die vergebenen und freien Start-/Landebahnen darstellen und trägt die zugeteilte Startbahn und den Soll-Zeitpunkt des Starts in das System ein. Hat das Flugzeug beim Starten die Startbahn erreicht, so gibt der Lotse die Parkposition wieder frei. Hat das Flugzeug den Flughafen verlassen – dies ist ersichtlich aus den Positionsmeldungen des Radars –, gibt der Lotse die Startbahn wieder frei und trägt den Ist-Zeitpunkt des Starts in das System ein. Mit der Freigabe ist eine automatische Buchung verbunden. Dabei wird aufgrund des Flugzeugtyps eine bestimmte Gebühr für Start und Landung gemäß der Gebührenliste dem Rechnungskonto der Fluggesellschaft zugeordnet.

- **Flugzeug verlegen**

  Gelandete Flugzeuge, die auf das separate Parkfeld oder auf die Werft gebracht werden, sind im System entsprechend zu kennzeichnen.

- **Positionsdaten fusionieren**

  Das System fragt zyklisch die Positionsdaten eines Flugzeugs beim Radar und beim Positionsmelder des Flugzeugs ab. Weichen die vom Radar und vom Positionsmelder des Flugzeugs gemeldeten Daten um mehr als eine vorgegebene Toleranz voneinander ab, so wird automatisch ein akustischer Alarm mit der Hupe generiert.

- **Rechnungen erstellen**

  Jeweils am Ersten eines Monats werden automatisch die Rechnungen an die Fluggesellschaften ausgedruckt und von den Angestellten der Verwaltung weitergeleitet.

- **Bezahlte Rechnungen buchen**

  Die Angestellten buchen von den Fluggesellschaften bezahlte Rechnungen auf das Konto der Fluggesellschaften.

- **Gebühren ändern**

  Die Angestellten der Verwaltung können die Preisliste der Start- und Landegebühren abändern.

Das System soll in Java mit einer Swing- oder JavaFX-Oberfläche realisiert werden. Die Bedienung soll einfach sein. Fehler der Software sollen an die Anwender gemeldet werden, damit diese im Fehlerfall auf ein manuelles Verfahren umsteigen können. Das System soll modular aufgebaut sein, um Änderungen leicht durchführen zu können.

Die hier vorgestellte Ausschreibung wird von dem Auftragnehmer geordnet in eine Anforderungs-Spezifikation mit sogenannten funktionalen und nicht funktionalen Anforderungen übertragen und mit dem Auftraggeber abgestimmt. Diese Anforderungs-Spezifikation kann aus Platzgründen hier nicht wiedergegeben werden. Sie befindet sich auf der Internetseite des Verlags.

2.6.2 Systemanalyse

Die **Entity-Klassen** des Problembereichs – die Entitätstypen der realen Welt entsprechen – sind implizit im Text der Ausschreibung enthalten. Diese Klassen zu finden, ist eine der Aufgaben der Systemanalyse. Natürlich gibt es in der Literatur verschiedene Möglichkeiten, Entity-Klassen den Texten zu entnehmen. Erwähnt sei die Analyse aller Hauptwörter [Hru98] oder die CRC-Methode [Cun86]. In Kapitel 2.6.2.4 wird ein Use Case-basierter Ansatz vorgestellt, der nach Ansicht der Autoren am effizientesten ist, da er systematisch einen Use Case nach dem anderen behandelt und damit automatisch dafür sorgt, dass bei der Analyse alle Use Cases erfasst werden. Beachten Sie, dass ein Use Case (auf Deutsch: Anwendungsfall) nichts anderes ist als eine Grundfunktion eines Systems, die einem Anwender zu Gute kommt.

Kontrollobjekte bzw. **Kontrollklassen** werden beim Studium von Abläufen gefunden. In der Systemanalyse findet man Entity-Klassen, Kontroll-Klassen und **Interface-Klassen**[34]. Das folgende Bild soll den Stellenwert der Systemanalyse symbolisieren:

Bild 2-16 Die Objekte der Systemanalyse sind die Basis des Systems

2.6.2.1 Kontextdiagramm

Im ersten Schritt der Systemanalyse soll das Kontextdiagramm für das Flughafen-Informationssystem gezeichnet werden. Ein Kontextdiagramm ordnet ein System in seine Umgebung ein und zeigt durch sogenannte Datenflüsse die Wechselwirkung des Systems mit seiner Umgebung. Der Name Kontextdiagramm rührt daher, dass dieses Diagramm das System in seiner Umgebung, d. h. seinem Kontext, zeigt. Zur Umgebung gehören bestehende Fremdsysteme und Geräte, die Daten mit dem System elektronisch austauschen, aber es ist auch möglich, anstelle der Geräte die Bediener des Systems anzugeben oder sogar sie zusätzlich zu nennen. Generell findet sich all das, was neu gebaut wird, im System wieder. Vorhandene Fremdsysteme, Geräte oder die Nutzer des Systems stehen außerhalb des zu bauenden Systems und werden als sogenannte **Aktoren** gezeichnet. Die Aktoren werden in der Regel als Strichmännchen, versehen mit dem Namen der Rolle, des Geräts oder des Fremdsystems, gezeichnet. Ein Datenfluss wird stets dargestellt durch einen Pfeil, der den Namen der ausgetauschten Daten trägt.

Zu beachten ist, dass in Bild 2-17 die folgenden Gruppendatenflüsse eingeführt wurden, um das Kontextdiagramm übersichtlich zu gestalten:

- **Landung Flugzeug** := Initialisierung Landeanflug + Reservierung Start-/Landebahn + Ist-Zeitpunkt Landung + Reservierung Parkposition + Freigabe Start-/Landebahn + [Gründe für Abbruch]

  mit Initialisierung Landeanflug :=
  betreuender Lotse + Flugzeugtyp + Fluggesellschaft + Soll-Zeitpunkt Landung

- Start Flugzeug :=
  Reservierung Start-/Landebahn + Soll-Zeitpunkt Start + Freigabe Parkposition + Freigabe Start-/Landebahn + Ist-Zeitpunkt Start

[34] Interface-Klassen bilden die Schnittstelle zu Fremdsystemen, Geräten und Rollen außerhalb eines Systems, den sogenannten **Aktoren**.

So entspricht der eingezeichnete Gruppendatenfluss "Landung Flugzeug" im Falle einer erfolgreichen Landung den Datenflüssen "Initialisierung Landeanflug" (wiederum untergliedert in "betreuender Lotse", "Flugzeugtyp", "Fluggesellschaft" und "Soll-Zeitpunkt Landung"), "Reservierung Start-/Landebahn", "Ist-Zeitpunkt Landung", "Reservierung Parkposition" und "Freigabe Start-/Landebahn". Im Falle eines Abbruchs umfasst der Gruppendatenfluss "Landung Flugzeug" auch die Gründe für den Abbruch. Wird die Landung erfolgreich durchgeführt, so entfallen die Gründe für den Abbruch.

Das folgende Bild zeigt das Kontextdiagramm für das Flughafen-Informationssystem in der Notation von UML:

Bild 2-17 Kontextdiagramm für das System "Flughafen-Informationssystem"

Das Programm "Flughafen-Informationssystem" erhält Daten von "Radarsensor", "Positionsmelder Flugzeug", "Flugplatzsensor", "Parksensor", "Tastatur Lotse", "Tas-

tatur Flughafenangestellter" sowie "Betriebssystem" und erzeugt Ausgaben für "Hupe", "Bildschirm Lotse" und "Drucker Flughafenangestellter".

2.6.2.2 Use Case-Diagramme

Im nächsten Schritt werden die Use Case-Diagramme für den Bedienertyp Lotse und Angestellter der Verwaltung gezeichnet. Eine Besonderheit dieser Anwendung ist, dass in diesem Beispiel die Use Cases der Lotsen und der Angestellten der Verwaltung streng voneinander getrennt sind. Bei anderen Anwendungen hingegen können **mehrere Rollen in einem einzigen Use Case** zusammenarbeiten. Ein Use Case-Diagramm enthält alle Use Cases und die an dem jeweiligen Use Case beteiligten Aktoren. Die an einem Use Case beteiligten Aktoren sind mit einer Linie[35] mit dem entsprechenden Use Case (in Form einer Ellipse) verbunden.

Es ist bereits bekannt, dass ein Use Case eine Leistung ist, die ein System für einen Nutzer zur Verfügung stellt. Leistungen eines Systems können

- asynchron (ereignisorientiert) angefordert werden,
- zeitgesteuert erfolgen
- oder fortlaufend aktiv sein.

Fortlaufend aktive Prozesse laufen ab, solange das System in Betrieb ist. So etwas kommt selten vor. Das Normale bei einem Informationssystem ist die asynchrone und die zeitgesteuerte Anforderung. Eine asynchrone Anforderung liegt beispielsweise vor, wenn ein Benutzer das Generieren und Ausdrucken einer Liste vom System anfordert, wenn er sie braucht. Ein zeitgesteuerter Use Case liegt vor, wenn beispielsweise eine bestimmte Liste, die einen Tag bilanziert, jede Nacht um 24 Uhr automatisch erzeugt und ausgedruckt wird.

Das folgende Bild zeigt die Use Cases des Angestellten der Verwaltung[36]:

Bild 2-18 Use Case-Diagramm "Angestellter der Verwaltung"

Die Angestellten der Verwaltung führen die Use Cases "Rechnungen erstellen", "bezahlte Rechnungen buchen" und "Gebühren ändern" rechnergestützt durch. Der Use Case "Rechnungen erstellen" wird zeitgesteuert angestoßen. Die Use Cases "be-

[35] Das ist eine sogenannte Assoziation.

[36] Da das System sehr einfach ist, kann man die Use Cases der Angestellten der Verwaltung und der Lotsen trennen. In komplexen Systemen sind i. Allg. mehrere Rollen an einem Use Case beteiligt.

zahlte Rechnungen buchen" und "Gebühren ändern" werden auf Anforderung durch den Bediener, d. h. ereignisorientiert, durchgeführt.

Ein Lotse führt ereignisorientiert im Dialog mit dem System die Use Cases "Luftlage anzeigen", "Bodenlage anzeigen", "Zeitplan anfordern", "Landung durchführen", "Start durchführen" und "Flugzeug verlegen" durch. Das folgende Bild zeigt die Use Cases des Lotsen:

Bild 2-19 Use Case-Diagramm "Lotse"

Der Use Case "Positionsdaten fusionieren" kommt ohne Zutun des Lotsen zustande. Er läuft stets zyklisch im Hintergrund mit. Dieser Use Case liest die Positionsdaten des Positionsmelders und des Radars ein und prüft, ob die zulässigen Toleranzen überschritten werden. Er löst einen Alarm mit der Hupe aus, wenn die vorgegebene Toleranz zwischen den Positionsdaten des Radars und denen des Positionsgebers im Flugzeug überschritten wird. Es ist dann Aufgabe des Lotsen, sich um das Problem zu kümmern. Auch wenn der Lotse diesen Use Case nicht anstößt, so wird er dennoch dem Lotsen zugeordnet, da er dem Lotsen einen Nutzen bringt.

2.6.2.3 Use Case-Beschreibungen

Zu einem jeden Use Case ist eine Use Case-Beschreibung[37] in textueller Form zu erstellen, die mit dem Auftraggeber abgestimmt wird. Für den Use Case "Landung durchführen" kann diese Beschreibung wie im Folgenden dann dargestellt aussehen.

Aus Platzgründen wird die Beschreibung jedoch nur bis zum Aufsetzen des Flugzeugs durchgeführt. Es liegt auf der Hand, wie die Beschreibung weitergeht.

Use Case: "Landung durchführen"

Initiator: Lotse

Beteiligte Aktoren: Bildschirm / Tastatur Lotse, Parkposition, Start-/Landebahn

Basisablauf: Meldet sich ein Flugzeug zur Landung an, so nimmt der Lotse das Flugzeug in das System auf. Er schaut dann nach, welche Landebahn wann frei ist und weist dem Flugzeug eine Landebahn zu. Der Soll-Zeitpunkt der Landung wird in das System aufgenommen. Der Lotse schaut nach, welche Parkposition frei ist – dies wird über die Parksensoren gemeldet – und weist dem Flugzeug eine Parkposition zu. Ist das Flugzeug gelandet, so trägt der Lotse den Ist-Zeitpunkt der Landung ein. Ist das Flugzeug an der Parkposition angekommen, so gibt der Lotse die Start-/Landebahn wieder frei.

Alternativablauf: Bei schlechtem Wetter wird die Landung abgebrochen. Der Lotse muss den Grund für den Abbruch in das System eintragen.

Alternativablauf: Ist die Fluggesellschaft dem System nicht bekannt, so müssen Name und Adresse der Gesellschaft in das System aufgenommen werden.

Alternativablauf: Alle Landebahnen belegt. Der Lotse ordnet eine Warteschleife an.

Alternativablauf: Alle Parkpositionen belegt. Der Lotse ordnet eine Warteschleife an.

2.6.2.4 Klassendiagramm der konzeptionellen Sicht der Systemanalyse

Ein Klassendiagramm der konzeptionellen Sicht enthält nur solche Klassen, deren Objekte im Alltag sichtbar sind. Im Rahmen der Objektorientierten Modellierung sind das die Entity-Klassen, die Entitätstypen der realen Welt entsprechen.

Für das Aufstellen eines Klassendiagramms der konzeptionellen Sicht wird hier folgendermaßen vorgegangen: Ein Bediener des Systems wird im Rahmen der konzeptionellen Sicht als Klasse dargestellt[38]. Für jeden **ereignisgesteuerten Use Case** werden alle Klassen identifiziert und die statischen Beziehungen (Assoziationen) zwischen den gefundenen Klassen gezeichnet. Eine Assoziation ist hierbei eine Verbindungslinie zwischen den Klassen, die einen Namen trägt. Der Name charakte-

[37] Es handelt sich um die sogenannte Langbeschreibung eines Use Case. Sie kann noch um Vor- und Nachbedingungen erweitert werden.

[38] Eigentlich ist er nur ein Aktor, aber jeder Aktor darf nur im Rahmen seiner Zugriffsberechtigung handeln. Jeder Zugriff ist zu überprüfen. Daher wird der Aktor zur Klasse.

risiert hierbei diese Beziehung. In entsprechender Weise sind für die **zeitgesteuerten Use Cases** und für **fortlaufend aktive Use Cases** die erforderlichen Klassen und ihre Beziehungen zu ermitteln.

Im Folgenden werden die in Bild 2-18 und Bild 2-19 gefundenen Use Cases eingeteilt in ereignisorientierte Use Cases, die asynchron vom Benutzer angefordert werden, und zeitgesteuerte Use Cases, die vom System selbst ausgelöst werden. Fortlaufend aktive Use Cases kommen beim Flughafen-Informationssystem nicht vor.

Ereignisorientierte Use Cases sind:

- Luftlage anzeigen,
- Bodenlage anzeigen,
- Zeitplan anfordern,
- Landung durchführen,
- Start durchführen,
- Flugzeug verlegen,
- Gebühren ändern,
- bezahlte Rechnungen buchen.

Zeitgesteuerte Use Cases sind:

- Positionsdaten fusionieren,
- Rechnungen erstellen.

Für jeden Use Case werden nun die beteiligten Klassen und ihre statischen Beziehungen gezeichnet. Durch Überlagerung der für die einzelnen Use Cases gefundenen Teil-Klassendiagramme kommt man zum gesamten Klassendiagramm des Systems.

Es folgen nun die Teil-Klassendiagramme für die einzelnen Use Cases:

- **Luftlage anzeigen**

Bild 2-20 Klassen und Beziehungen für den Use Case "Luftlage anzeigen"

Ein Lotse betrachtet Flugzeuge in der Luft. Zu jedem Flugzeug gehört eine Position Positionsmelder und eine Position Radar.

Beachten Sie bitte die Multiplizitäten an den Beziehungen. So hat in Bild 2-20 ein Flugzeug 1..*, d. h. 1 oder mehrere Positionsdaten, die vom Radar bzw. vom Positionsgeber gemeldet werden. Das ist klar, schließlich verändert ein Flugzeug seine Position mit der Zeit und es soll, wenn das Flugzeug zum ersten Mal auf dem Bildschirm auftaucht, seine Bahnkurve am Bildschirm dargestellt werden können.

- **Bodenlage anzeigen**

Bild 2-21 Klassen und Beziehungen für den Use Case "Bodenlage anzeigen"

Der Lotse kann sich im Lageplan des Flughafens anzeigen lassen, an welcher Stelle des Flughafens sich aktuell welches Flugzeug befindet.

- **Zeitplan anfordern**

Bild 2-22 Klassen und Beziehungen für den Use Case "Anforderung Zeitplan"

Der Zeitplan wird dadurch generiert, indem alle für den Start bzw. die Landung angemeldeten Flugzeuge – dies wird ausgedrückt durch die Bedingung [Flugzeug angemeldet] – aufgelistet werden.

- **Landung durchführen**

Bild 2-23 Klassen und Beziehungen für den Use Case "Landung durchführen"

Beachten Sie bitte den Lesepfeil bei "gehört". In UML wird ein Lesepfeil durch ein Dreieck dargestellt. Standardmäßig werden die Beziehungen von links nach rechts gelesen und von oben nach unten. Beispiele hierfür sind in Bild 2-23 "Lotse betreut Flugzeug" oder "Flugzeug benutzt Parkposition". Die Beziehung "Flugzeug gehört Fluggesellschaft" ist von unten nach oben zu lesen. Daher der eingezeichnete Lesepfeil.

- **Start durchführen**

Bild 2-24 Klassen und Beziehungen für den Use Case "Start durchführen"[39]

Eine Besonderheit ist hier zu sehen. Die Klasse Gebühr stellt eine **Assoziations-klasse** dar. Eine Assoziationsklasse gehört zu einer Beziehung (Assoziation) und spezifiziert diese näher.

Entfällt eine Beziehung, so muss auch die zugeordnete Assoziationsklasse wegfallen.

Da keine Gebühr anfällt, wenn die Start-/Landebahn nicht benutzt wird, muss die Klasse `Gebühr` also eine Assoziationsklasse sein.

- **Flugzeug verlegen**

Bild 2-25 Klassen und Beziehungen für den Use Case "Flugzeug verlegen"

[39] Nach UML müssten Assoziation und Assoziationsklasse gleich heißen.

- **Gebühren ändern**

Bild 2-26 Klassen und Beziehungen für den Use Case "Gebühren ändern"

- **Bezahlte Rechnungen buchen**

Bild 2-27 Klassen und Beziehungen für den Use Case "Bezahlte Rechnungen buchen"

Durch die Multiplizität wird zum Ausdruck gebracht, dass für ein bestimmtes Konto nur ein einziger Angestellter ermächtigt ist zu buchen. Ein Angestellter ist aber für mehrere Konten zuständig.

- **Positionsdaten fusionieren**

Hier werden zyklisch die Flugzeugpositionsdaten des Radars und des Positionsgebers abgefragt und geprüft, ob sie im Rahmen einer vorgegebenen Toleranzgrenze übereinstimmen. Der Vergleich wird realisiert durch ein **Kontrollobjekt**[40], das sich zyklisch die Positionsdaten vom Radar und vom Positionsgeber beschafft, den Vergleich durchführt, die Positionsdaten vom Radar und vom Positionsgeber im System anlegt – wenn die Toleranzgrenze nicht überschritten wird – und bei Überschreiten der Toleranzgrenze einen Alarm auslöst. In der konzeptionellen Sicht leistet dieser Use Case keinen neuen Beitrag zum Klassendiagramm, die Klassen `Position` `Positionsmelder` und `Position` `Radar` sind bereits bekannt.

- **Rechnungen erstellen**

Durch ein **Kontrollobjekt**, welches auf die Uhr schaut, werden in automatischer Weise jeweils am Ersten eines Monats die Rechnungen automatisch generiert und ausgedruckt. Die Angestellten müssen die Rechnungen noch kuvertieren und versenden. In der konzeptionellen Sicht gibt dieser Use Case keinen Beitrag zum Klassendiagramm, da in der konzeptionellen Sicht nur Entity-Objekte betrachtet werden. Die Rechnungen sind keine Entity-Objekte, d. h. Objekte von Entity-Klassen, da sie nicht im System gespeichert werden. Sie werden erzeugt und sofort ausgedruckt.

[40] Kontrollobjekte haben kein Gegenstück in der realen Welt.

Gesamtes Klassendiagramm

Durch Zusammensetzen der für die einzelnen Use Cases gefundenen Teil-Klassendiagramme, d. h. von Bild 2-20 bis Bild 2-27, erhält man das gesamte **Klassendiagramm**:

Bild 2-28 Gesamtes Klassendiagramm "Flughafen-Informationssystem"

2.7 Zusammenfassung

Objekte eines Systems der realen Welt werden in die Modellierung und später in die Programme als Entity-Objekte übernommen. Der Typ gleichartiger Objekte ist die sogenannte Klasse, hier eine Entitätsklasse. Bei der Übernahme aus der Realität findet eine Abstraktion, d. h. eine Konzentration auf die wesentlichen Eigenschaften statt. Entity-Klassen haben stets Eigenschaften und sind deshalb Daten tragend. Man kommt auf Entity-Objekte durch "Peilen" in die Realität und schaut, welche konkreten Dinge (wie z. B. Tisch oder Stuhl etc.) und abstrakten Dinge (wie etwa ein Vertrag) zu Objekten der Anwendung werden sollen.

Kontrollklassen findet man nicht durch "Peilen" in die Realität. Um Kontrollklassen zu finden, müssen Abläufe betrachtet werden.

Interface-Klassen kapseln die Aktoren, insbesondere ihre Geräteabhängigkeit, damit die Entity- und Kontroll-Objekte unabhängig von der speziellen Ausprägung eines Aktors werden. Interface-Klassen haben kein Gegenstück im Problembereich. Interface-Objekte werden zwischen den betrachteten Entity-/Kontroll-Objekten und den Akteuren eingeschoben.

Entity-, Kontroll- und Interface-Klassen werden in der Systemanalyse gefunden. Sie leben beim Entwurf weiter. Allerdings wird ein Interface-Objekt der Systemanalyse auf Objekte mehrerer technischen Klassen des Entwurfs abgebildet. Beim Systementwurf kommen jedoch noch weitere Objekte technischer Klassen hinzu wie beispielsweise für den Start-up bzw. Shut-down des Systems.

Der in diesem Buch verwendete Begriff des mikroskopischen Zustands eines Objekts sieht einen mikroskopischen Zustand als Punkt in einem n-dimensionalen Zustandsraum eines Objekts, der von den Attributwerten eines Objekts aufgespannt wird. Ein makroskopischer Zustand hingegen muss für eine Anwendung eine Bedeutung haben. Am einfachsten macht man sich den Unterschied klar, indem man einen Fahrstuhl betrachtet. Je nach Auflösung des Gewichtssensors kann es für den mikroskopischen Zustand des Gewichts aller Personen im Fahrstuhl sehr viele Werte geben. Für den makroskopischen Zustand ist nur entscheidend, ob der Fahrstuhl überladen ist oder nicht. (Siehe Kapitel **2.1**.)

Kapitel **2.2** betrachtet Kapselung, Abstraktion und Information Hiding (Geheimnisprinzip). Kapselung, Abstraktion und Information Hiding führen zu schmalen Schnittstellen zwischen den einzelnen Objekten. Die Begriffe Kapselung, Abstraktion und Information Hiding sind eng miteinander verwandt. Der Begriff der Kapselung ist eines der wichtigsten Konzepte der objektorientierten Programmierung. Methoden und Daten verschmelzen bei der Kapselung zu einem Objekt. Daten und Methoden befinden sich also zusammen in einer Kapsel vom Typ einer Klasse. Der Begriff der Kapselung umfasst sowohl die Realisierung von Information Hiding zum Verbergen der inneren Details, d. h. der Daten und der Implementierung des Verhaltens, als auch die Realisierung von Abstraktion zur Sichtbarmachung und zum Ansprechen des Verhaltens der Kapsel durch eine Schnittstelle nach außen.

In Kapitel **2.3** wird unterschieden zwischen Instanzvariablen bzw. -methoden und Klassenvariablen bzw. -methoden. Instanzvariablen existieren für jedes Objekt einer Klasse in einer individuellen Ausprägung. Eine Klassenvariable liegt als zentrale und nur ein einziges Mal vorhandene (unikate) Variable für alle Objekte der entsprechenden Klasse vor. Im Normalfall arbeiten Instanzmethoden auf Instanzvariablen und Klassenmethoden auf Klassenvariablen.

Kapitel **2.4** befasst sich mit der Abstraktion und der damit verbundenen Brechung der Komplexität. Da die Welt zu komplex ist, braucht man wiederholt Abstraktionen, um die Dinge zu vereinfachen und mit der Komplexität fertig zu werden.

In den Kapiteln **2.5** und **2.6** werden die erlernten Themen anhand praktischer Beispiele umgesetzt und näher erläutert.

2.8 Übungen

Aufgabe 2.1: Modellierung mit Klassen und Objekten

2.1.1 Was versteht man unter Problembereich und unter Lösungsbereich?
2.1.2 Was sollte bei einer Klasse im Idealfall komplett verborgen sein?
2.1.3 Was ist beim objektorientierten Ansatz entscheidend?
2.1.4 Worauf basiert der objektorientierte Ansatz?
2.1.5 Aus welchen Bestandteilen besteht eine Klasse?
2.1.6 Erläutern Sie, was eine Methode ist und welche Aufgaben sie erfüllt.
2.1.7 Wodurch wird der Zustand eines Objektes festgelegt?
2.1.8 Was ist der Unterschied zwischen einer Klassenmethode und einer Instanzmethode?
2.1.9 Wann werden Klassenvariablen verwendet?
2.1.10 Wann werden Instanzvariablen verwendet?
2.1.11 Kann eine Instanzmethode auf eine Klassenvariable zugreifen? Begründung!
2.1.12 Kann eine Klassenmethode auf eine Instanzvariable zugreifen? Begründung!
2.1.13 Erläutern Sie die Begriffe Generalisierung und Spezialisierung.
2.1.14 Erläutern Sie die Begriffe Aggregation und Komposition.

Aufgabe 2.2: Kapselung, Information Hiding und Abstraktion

2.2.1 Erläutern Sie den Begriff der Kapselung bei einem Objekt.
2.2.2 Was ist Information Hiding bei einem Objekt?
2.2.3 Was versteht man unter Abstraktion bei einem Objekt?

Kapitel 3

Einführung in die Programmiersprache Java

3.1 Sprachkonzepte von Java
3.2 Eigenschaften von Java
3.3 Die Java-Plattform
3.4 Programmerzeugung und -ausführung
3.5 Das Java Development Kit
3.6 Begriffe im Java-Umfeld
3.7 Zusammenfassung
3.8 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_3

3 Einführung in die Programmiersprache Java

Im Jahre 1991 befasste sich ein Mitarbeiterteam von Sun Microsystems (inzwischen Oracle) mit der Entwicklung von Set-Top-Boxen für Fernsehgeräte mit dem Ziel des interaktiven Fernsehens. Für die Software dieser Boxen wurde zunächst C++ verwendet – die damals gängige Sprache für objektorientierte technische Anwendungen. Die Unzufriedenheit über die Komplexität von C++ und die daraus resultierende mangelnde Sicherheit veranlasste das Team, ausgehend von C und C++ eine neue Sprache – die Sprache Java – zu entwickeln.

In Kapitel 3.1 wird gezeigt, wie vorhandene Programmiersprachen das Sprachkonzept von Java beeinflussten. Java ist einfach, stabil, objektorientiert, verteilbar, sicher und portierbar. Diese Eigenschaften werden in Kapitel 3.2 hervorgehoben. In Kapitel 3.3 wird die Java-Plattform vorgestellt. Zur Java-Plattform gehören außer der Sprache Java selbst auch Werkzeuge wie der Java-Compiler `javac`, die Java-Virtuelle Maschine und eine umfassende Klassenbibliothek. Bei der Erklärung der Java-Plattform wird speziell auf die Java-Virtuelle Maschine mit Bytecode-Interpreter und Just-in-time-Compiler (kurz: JIT-Compiler) eingegangen wie auch auf die Java-Klassenbibliothek. In Kapitel 3.4 werden die einzelnen Schritte der Programmerzeugung und -ausführung erklärt. Kapitel 3.5 zeigt, von wo das Java Development Kit bezogen werden kann und wie es installiert werden muss. Zuletzt wird in Kapitel 3.6 auf Begriffe eingegangen, die häufig verwechselt werden.

3.1 Sprachkonzepte von Java

Bei der Spezifikation der Programmiersprache Java wurden von verschiedenen Programmiersprachen herausragende Konzepte übernommen und in einer bisher nicht bekannten Kombination zusammengefügt. Dies visualisiert das folgende Bild:

Bild 3-1 Die Väter von Java

Die **Sprachkonstrukte** für das "**Programmieren im Kleinen**" wie die einfachen Datentypen und die Operatoren sind an **C++** angelehnt.

Das Konzept der virtuellen Maschine zur Ausführung des **Bytecodes** hat sich in Smalltalk bewährt. Die Idee der virtuellen Maschine wurde z. B. auch von N. Wirth in dessen Pascal-Compiler umgesetzt. Dieser Pascal-Compiler übersetzte in den sogenannten P-Code und dieser wurde dann interpretiert. Der Gedanke, der dahinter steckt, ist, **unabhängig von der** jeweiligen **Rechner-Plattform** zu werden.

Unter **Rechner-Plattform** wird die **Kombination** von **Betriebssystem** und zugehöriger **Rechner-Hardware** verstanden.

Um die Unabhängigkeit des Bytecodes von der Rechner-Plattform zu ermöglichen, braucht man ein Stück Software, das den Bytecode auf die jeweilige Rechner-Plattform abbilden kann. Diese Aufgabe wird von der **virtuellen Maschine** wahrgenommen.

Dies zeigt das folgende Bild:

Bild 3-2 Abbildung des Byte-Codes auf ein Betriebssystem

Die virtuelle Maschine bietet einem auszuführenden Programm stets dieselbe Schnittstelle, egal welche Rechner-Plattform von der virtuellen Maschine angesteuert wird. Damit kann ein Programm weltweit auf jeder Rechner-Plattform, die über eine virtuelle Maschine verfügt, laufen und verhält sich auch immer gleich.

Mit Hilfe der virtuellen Maschine kann die Idee "**write once, run anywhere**", d. h. "einmal schreiben, überall ausführen", umgesetzt werden.

Das Konzept der **Applets**[41] in Java, die bei Bedarf geladen werden, wurde von den Modulen in **Oberon**[42] übernommen.

[41] **Java-Applets** sind kleine Programme, die nur innerhalb eines Webbrowsers oder eines Applet-Viewer ausgeführt werden können. Sie sind in einer HTML-Seite eingebettet.

[42] **Oberon** ist eine objektorientierte Programmiersprache, die wie Pascal und Modula von N. Wirth entwickelt wurde.

Pakete dienen zur Gruppierung von inhaltlich zusammengehörigen Klassen und Schnittstellen. Ein Paket stellt eine Bibliothek für einen bestimmten Zweck dar. Pakete als Gruppierung kleinerer Bausteine wurden bereits in Ada verwendet.

Die Idee des **JavaBeans[43]-Komponentenmodells** wurde von OpenDoc beeinflusst. OpenDoc wurde von den Firmen Apple, Novell, Borland, WordPerfect und IBM entwickelt und ist eine offene Dokumentenarchitektur, die es ermöglicht, beliebige Dokumententeile aufzunehmen. Ein Komponentenmodell soll die Verwendung von vorgefertigten Software-Bausteinen auf einfache Weise ermöglichen. Erstellte JavaBeans-Komponenten, die sich an die JavaBeans-Spezifikation halten, können in einfacher Weise wiederverwendet werden und als fertige Bausteine mit Hilfe eines Werkzeugs – eines sogenannten Application Builders – in eine neue Anwendung integriert werden.

Enterprise JavaBeans sind Teil der Java Enterprise Edition-Spezifikation. Sie haben mit JavaBeans nichts zu tun, außer dass es sich auch hier um ein Komponentenmodell handelt. Sie dienen jedoch speziell zur Realisierung von Server-Architekturen.

3.2 Eigenschaften von Java

Java-Programme sind einfach, stabil, objektorientiert, verteilbar, sicher und portierbar.

Im Folgenden werden diese Eigenschaften beschrieben:

- **Einfachheit und Stabilität**

  Zur Erhöhung der Einfachheit der Programmiersprache und der Stabilität der Programme wurden in Java gegenüber C und C++ unsichere Sprachkonstrukte wie z. B. die Zeigerarithmetik weggelassen. Natürlich wurde die Sprache Java neu formuliert.

- **Objektorientiertheit**

  Mit Java wurde eine echte objektorientierte Sprache entworfen, die den Programmierer zwingt, objektorientiert zu programmieren. Es ist generell nicht möglich, Methoden von einer Klasse bzw. von den Objekten zu trennen.

- **Verteilbarkeit**

  Java wurde von Anfang an für die Verteilung von Programmen auf verschiedene Rechner entworfen. Infolge einer umfangreichen Unterstützung durch die Java-Klassenbibliothek ist Java nahezu optimal für die Client/Server-Programmierung geeignet. Des Weiteren wurde durch das Konzept der Applets, die von einem zentralen **Web-Server** auf einen anderen Rechner über das Netz geladen werden können, das Problem der Verteilung bei Software-Updates elegant gelöst.

[43] JavaBeans werden in dieser Einführung nicht behandelt.

- **Sicherheit**

  Java hat ein mehrstufiges Sicherheitskonzept, das die Ausführung von kritischen oder für das System gefährlichen Operationen verhindert.

- **Portierbarkeit**

  Durch die Kompilierung in einen von der Rechner-Plattform unabhängigen Bytecode, der von einer virtuellen Maschine ausgeführt wird, ist Java unabhängig vom jeweiligen Betriebssystem und der zugehörigen Rechner-Hardware. Software-Entwickler müssen ihre Programme nicht für jede Rechner-Plattform speziell anpassen. Durch die exakte Definition aller Datentypen in Länge und Aussehen und das Vorhandensein von Threads als Sprachmittel für parallel ausführbare Programme ist ein Java-Programm ohne Probleme von einer Rechner-Plattform auf die andere portierbar (übertragbar). Je geringer die Zahl der Eingriffe in ein Programm ist, um es auf einer anderen Rechner-Plattform ausführen zu können, desto höher ist die Portabilität. Bei Java sind in einem Programm keine Eingriffe erforderlich, um es auf einer anderen Rechner-Plattform auszuführen. Vorausgesetzt wird aber, dass für jede Rechner-Plattform eine virtuelle Maschine existiert. Versuchte man das Gleiche zur Zeit der Entstehung von Java in C++, einer Sprache, die auf allen gängigen Maschinen vertreten ist, so geriet man bereits bei der unterschiedlichen Darstellung der einfachen Datentypen in Schwierigkeiten. Darüber hinaus gab es vor C++11 keine Sprachmittel für parallele Programme, sodass das Erreichen von Parallelität früher in C++ betriebssystemabhängig war. Die Portierbarkeit erstreckt sich in Java auch auf die grafische Oberfläche, welche traditionsgemäß früher komplett betriebssystemabhängig war.

3.3 Die Java-Plattform

Meistens ist – wenn man über Java spricht – die Programmiersprache Java gemeint. Java ist aber viel mehr als eine Programmiersprache.

Zu einer **Java-Plattform** gehören:

- die Programmiersprache Java,
- Werkzeuge wie zum Beispiel der Java-Compiler (`javac`),
- die **Java-Virtuelle Maschine** (JVM)
- und eine umfassende **Klassenbibliothek**.

Die **Java-Plattform** existiert:

- als Standard Edition (bekannt unter dem Namen **Java SE**),
- als Enterprise Edition (**Java EE**),
- als Micro Edition (**Java ME**)
- und als **Java Card** für Chipkarten (Smartcards), allerdings mit einem etwas eingeschränkten Sprachumfang.

Die einzelnen Ausführungen unterscheiden sich im Wesentlichen durch Art und Umfang der Klassenbibliothek und durch die verfügbaren Werkzeuge. Die Java SE umfasst eine Grundmenge an Klassenbibliotheken und wird für den allgemeinen Einsatz

auf PCs und Servern verwendet. Die Java EE, Java ME und Java Card bauen hierbei auf der Java SE auf und spezialisieren diese für Server-Anwendungen (Java EE), für den Einsatz auf mobilen Endgeräten (Java ME) und auf Chipkarten (Java Card). Dabei ist zu erwähnen, dass die Bedeutung der Java ME zurückgeht, zum einen weil moderne Geräte aufgrund ihrer Leistungsfähigkeit heutzutage eine normale JVM unterstützen können, zum anderen weil Android mit einem eigenen Bytecode und einer eigenen Virtual Machine inzwischen eine herausragende Stellung am Handy- bzw. Smartphone-Markt hat.

Das **Java Development Kit** (JDK) beinhaltet eine Java-Plattform für ein spezielles Betriebssystem. Will man mit Java Programme entwickeln, so benötigt man also ein JDK.

Hierbei gibt es für die verschiedenen Java Editionen (Java SE, Java EE und Java ME) jeweils ein JDK für jede gängige Rechner-Plattform. Seit Java Card 3.0 gibt es für die Java Card-Plattform die sogenannte Classic Edition und die Connected Edition mit verschiedenen Development Kits.

Eine **Java Runtime Environment** (JRE) beinhaltet nur diejenigen Bestandteile eines JDK, welche zum Ausführen von Java-Programmen benötigt werden.

Eine JRE besteht somit aus einer Java-Virtuellen Maschine für das jeweilige Betriebssystem, der Java-Klassenbibliothek und einigen Tools für die Administration von Java bzw. für den Betrieb von Java-Programmen.

Auf die Java-Virtuelle Maschine wird in Kapitel 3.3.1 und auf die Java-Klassenbibliothek der Standard Edition in Kapitel 3.3.2 noch näher eingegangen.

3.3.1 Die Java-Virtuelle Maschine

Die Java-Virtuelle Maschine ist die Schnittstelle zwischen der Anwendung und dem Rechner, auf dem eine Anwendung zur Ausführung gebracht werden soll.

Bei Java wird bei der Kompilierung aus dem Quellcode nicht direkt Maschinencode sondern ein **Bytecode** (Zwischencode) erzeugt. Hierbei ist ein **Maschinencode** eine prozessorspezifische Programmiersprache, die ein spezieller Prozessor direkt versteht.

Die Java-Virtuelle Maschine ist unter anderem für die Verarbeitung des Zwischencodes verantwortlich.

Für die Verarbeitung des Zwischencodes verfügt die Java-Virtuelle Maschine über eine

- **Ausführungseinheit** (engl. **execution engine**)
- und einen **Klassenlader** (engl. **class loader**).

Ein **Bytecode-Interpreter** und ein **JIT-Compiler**[44] sind Bestandteile der **Ausführungseinheit**. Die Ausführungseinheit kann man sich als eine Art virtueller Prozessor vorstellen. Welche der beiden Komponenten der Ausführungseinheit wann verwendet wird, ist abhängig von diversen Faktoren wie der Größe des Codes, der Nutzungshäufigkeit von Codeabschnitten etc. Standardmäßig entscheidet die Java-Virtuelle Maschine selbstständig über die Art der Ausführung.

Beim Start einer in Java geschriebenen Anwendung sucht der **Klassenlader** nach dem Einstiegspunkt – der Methode `main()`, in der beim Starten der VM angegebenen Startklasse.

Ist der Einstiegspunkt gefunden, wird im Allgemeinen der Zwischencode in einer sogenannten "Warm Up"-Phase erst einmal vom Interpreter interpretiert. Dabei sammelt der JIT-Compiler Informationen, welche Code-Bereiche, sogenannte Hot Spots, häufig ausgeführt werden. Daraufhin beginnt der JIT-Compiler, den Bytecode zu optimieren, indem er z. B. die Methodenaufrufe durch den Code der aufgerufenen Methode ersetzt (inlining) und dann daraus Maschinencode erzeugt.

Bitte beachten Sie, dass im Folgenden in diesem Buch aus Gründen der Einfachheit nur vom Bytecode-Interpreter gesprochen wird und der JIT-Compiler nicht erwähnt wird.

Das folgende Bild zeigt, dass es für jede Rechnerplattform eine eigene JVM gibt:

Bild 3-3 Die Java-Virtuelle Maschine (JVM)

Durch das Bytecode-Konzept ist jedes Java-Programm auf jeder Rechner-Plattform mit einer Java-Virtuellen Maschine ausführbar.

[44] Just-in-time-Compiler erzeugen Maschinencode (Maschinensprache), während sich ein Programm in der Ausführung befindet.

Neben der **Ausführungseinheit** und dem **Klassenlader** besitzt die Java-Virtuelle Maschine noch drei weitere wichtige Komponenten:

- Die **Speicherverwaltung** (engl. garbage collection) sorgt für das Aufräumen von nicht weiter benötigten Objekten,
- Der **Bytecode-Verifizierer** (engl. bytecode verifier) ist für die Prüfung des geladenen Bytecodes verantwortlich.
- Die **Java-Standardbibliotheken** liefern die Basisfunktionalitäten für jede Anwendung.

Die Java-Virtuelle Maschine kann nicht nur Java, sondern auch andere Sprachen ausführen wie Groovy, JRuby oder Scala.

3.3.2 Die Java-Klassenbibliothek

Erleichtert wird das Programmieren von Java-Anwendungen durch eine umfangreiche Klassenbibliothek, die **Java-API**[45].

Die Klasse `java.lang.Object` ist die Wurzel des Java-Klassenbaums, also auch der Java-API.

Eine ausführliche Beschreibung der Klassen und Schnittstellen kann der **Java-API-Dokumentation** entnommen werden. Die Java-API-Dokumentation ist zu finden unter der Java-Webseite von Oracle:

http://docs.oracle.com/javase/8/docs/api/

Die Java-Klassenbibliothek der Standard Edition lässt sich in drei Bereiche aufteilen:

- **Java Base Libraries**

  Diese Bibliotheken umfassen Klassen und Schnittstellen, welche die grundlegende Funktionalität der Java-Plattform bereitstellen. So sind darin unter anderem Klassen für die String-Verarbeitung, Ein-/Ausgabe, Sicherheit, Netzwerk-Kommunikation und Internationalisierung definiert.

- **Java Integration Libraries**

  Diese Bibliotheken setzen sich aus Klassen und Schnittstellen zusammen, welche für die systemübergreifende Kommunikation benötigt werden. Sie bestehen unter anderem aus der **Java Database Connectivity** (**JDBC**) **API** für den Zugriff auf Datenbanken, der Remote Method Invocation (RMI) API für den entfernten Methodenaufruf zwischen unterschiedlichen Java-Virtuellen Maschinen und der **Java Naming and Directory Interface** (**JNDI**) **API** für das Suchen von Klassen innerhalb eines Namensraumes.

- **Java User Interface Libraries**

  Mit Hilfe dieser Bibliotheken können verschiedenste Benutzerschnittstellen programmiert werden. Beispiele hierfür sind Druckerschnittstellen, Tonausgabe, Bildverarbeitung und die Entwicklung von grafischen Benutzerschnittstellen.

[45] API = Application Programming Interface.

Die Java-API enthält mehrere tausend Klassen und Schnittstellen. Diese können nicht alle in diesem Buch erklärt werden. Vielmehr erfolgt eine Beschränkung auf die Beschreibung und Erläuterung der grundlegenden Spracheigenschaften von Java. Dabei werden einige wenige ausgewählte Klassen der Klassenbibliothek vorgestellt.

3.4 Programmerzeugung und -ausführung

Das folgende Bild zeigt das Erzeugen und Starten einer Java-Anwendung:

Bild 3-4 Ablauf und Erzeugnisse beim Kompilieren und Laden

Die Erzeugung und das Starten von Java-Programmen unterscheidet sich in einigen Punkten grundlegend von anderen Programmiersprachen. Im Wesentlichen kann man sagen, dass kein **ausführbares Programm**[46] (engl. **executable program**) erzeugt wird, sondern dass Bytecode interpretiert wird, wobei benötigte Klassen zur **Laufzeit** durch die Java-Virtuelle Maschine nachgeladen werden.

[46] Ein ausführbares Programm besteht aus Maschinencode und kann nach seinem Aufruf selbstständig auf dem Prozessor laufen.

Mit dem Begriff "zur Laufzeit" ist gemeint, dass eine Aktion wie beispielsweise das Laden einer Klasse in die Java-Virtuelle Maschine ausgeführt wird, während das Programm läuft.

In Java wird kein ausführbares Programm erzeugt.

Der Quelltext eines Programms wird mit einem **Editor**, einem Werkzeug zur Erstellung von Texten, geschrieben und auf der Festplatte des Rechners unter einem Dateinamen als Datei abgespeichert. Da eine solche Datei Quellcode enthält, wird sie auch als **Quellcode-Datei** (oder **Quelldatei**) bezeichnet. Durch Kompilieren erhält man beispielsweise aus der Quellcode-Datei `Bsp1.java` in Bild 3-4 die drei Bytecode-Dateien `Bsp1.class`, `Bsp2.class` und `Bsp3.class`. Dabei werden die Bestandteile des Dateinamens hinter dem Punkt als **Extension** (**Dateinamenserweiterung**) bezeichnet. Für jede Klasse in der Quellcode-Datei wird also eine Bytecode-Datei erzeugt, die den Namen der Klasse trägt und die Extension `class` hat.

Die Klasse `Bsp1` aus Bild 3-4 soll eine `main()`-Methode haben und als **Startklasse** dienen. Wird diese Klasse vom **Interpreter** aufgerufen, so wird sie vom **Klassenlader** geladen und ihre `main()`-Methode ausgeführt.

Klassen, die eine `main()`-Methode haben, können zum Starten einer Java-Anwendung verwendet werden.

Werden nun innerhalb der erwähnten `main()`-Methode Klassenvariablen bzw. Klassenmethoden der beiden anderen Klassen `Bsp2` oder `Bsp3` angesprochen oder Objekte dieser Klassen angelegt und ihre Komponenten verwendet, so müssen diese Klassen vom Klassenlader dynamisch zur Laufzeit vom aktuellen Rechner, einem Intranet oder dem Internet in die Java-Virtuelle Maschine nachgeladen werden.

Ein "Programm" in Java besteht aus einem losen Verbund von einzelnen `.class`-Dateien, die bei Bedarf in die Java-Virtuelle Maschine geladen werden. **Dynamisches Laden** bedeutet, dass die Klassen, wenn sie das erste Mal benötigt werden, in die Java-Virtuelle Maschine nachgeladen werden.

3.4.1 Kompilieren

Am Anfang der Programmerzeugung stehen eine oder mehrere Quellcode-Dateien. Diese Dateien enthalten den in Java geschriebenen Programmtext und werden durch den Compiler **`javac`** in **Bytecode** übersetzt. Für jede Klasse wird beim Kompilieren eine interpretierbare `.class`-Datei erzeugt. Der Bytecode ist unabhängig von der Rechner-Plattform, auf welcher der Programmcode entwickelt bzw. kompiliert worden

ist. Man kann den erzeugten Bytecode auch als Maschinencode für einen virtuellen Prozessor, genauer die Java-Virtuelle Maschine, bezeichnen.

Der **Bytecode** ist ein **Zwischencode**. Er stellt noch nicht den Maschinencode eines existierenden physischen Prozessors dar. Er stellt die **Maschinensprache für** eine abstrakte Maschine, die **Java-Virtuelle Maschine**, dar.

Der Kompiliervorgang durchläuft die vier Bearbeitungsschritte

- lexikalische Analyse,
- Syntaxanalyse,
- Semantikanalyse
- und Codeerzeugung.

3.4.1.1 Lexikalische Analyse

Bei der **lexikalischen Analyse** wird versucht, in der Folge der Zeichen eines Programms die Wörter der Sprache – das sind die kleinsten Einheiten einer Sprache, die eine Bedeutung besitzen – zu erkennen.

Die Wörter einer Sprache werden auch **Symbole** genannt. Beispiele für Symbole sind Namen, Schlüsselwörter oder Operatoren. Zwischenräume und Kommentare dienen dem Compiler dazu, zu erkennen, wo ein Wort zu Ende ist. Ansonsten haben sie keine Bedeutung für den Compiler und werden überlesen. Das folgende Bild symbolisiert, dass Wörter anhand von Trennern wie Zwischenräumen erkannt werden:

Bild 3-5 Erkennen von Wörtern

3.4.1.2 Syntaxanalyse

Für alle modernen Sprachen existiert ein Regelwerk, welches formal die zulässigen Folgen von Symbolen (Wörtern) festlegt.

Im Rahmen der **Syntaxanalyse** wird geprüft, ob die bei der lexikalischen Analyse ermittelte Symbolfolge eines zu übersetzenden Programms zu der Menge der **zulässigen Symbolfolgen** gehört.

3.4.1.3 Semantische Analyse

Die **semantische Analyse** versucht, die **Bedeutung der Wörter** herauszufinden.

Die Bedeutung in einem Programm bezieht sich im Wesentlichen auf dort vorkommende Namen, also muss die semantische Analyse herausfinden, was ein Name, der im Programm vorkommt, bedeutet. Jeder Name wird mit einer Bedeutung versehen, d. h. an eine Deklaration des Namens im Programm gebunden. Grundlage hierfür sind die Sichtbarkeits-, Gültigkeits- und Typregeln einer Sprache.

Neben der Überprüfung der Verwendung von Namen im Rahmen ihrer Gültigkeitsbereiche spielt die **Überprüfung von Typverträglichkeiten bei Ausdrücken** eine Hauptrolle bei der semantischen Analyse.

Auf Gültigkeitsbereiche wird in Kapitel 9.1.5 eingegangen.

Ein wesentlicher Anteil der semantischen Analyse befasst sich also mit der Erkennung von Programmfehlern, die durch die Syntaxanalyse nicht erkannt werden konnten, wie z. B. die Addition von zwei Werten mit unterschiedlichem und nicht verträglichem Typ. So ist es beispielsweise unzulässig, boolesche Werte und Zahlen zu addieren.

Nicht alle semantischen Regeln einer Programmiersprache können durch den Compiler abgeprüft werden.

Man unterscheidet zwischen der **statischen Semantik** (durch den Compiler prüfbar) und der **dynamischen Semantik** (erst zur Laufzeit eines Programmes prüfbar). Die Prüfungen der dynamischen Semantik werden von der Java-Virtuellen Maschine durchgeführt.

3.4.1.4 Erzeugung von Bytecode

Während lexikalische Analyse, Syntaxanalyse und semantische Analyse sich nur mit der **Analyse** des zu übersetzenden Quellcodes befassen, kommen bei der **Codegenerierung** die Eigenschaften der Java-Virtuellen Maschine mit ins Spiel.

Bei der **Codeerzeugung** wird kein Maschinencode für einen physischen Prozessor einer Plattform, sondern Maschinencode – nämlich der Bytecode – für die Java-Virtuelle Maschine erzeugt.

Das Konzept des Bytecodes wurde schon seit den siebziger Jahren eingesetzt, um Programme leichter portieren zu können. Beispiele hierfür sind Bytecode-Systeme für die Sprachen BCPL, LISP, Prolog, aber auch für Pascal. Dabei war beispielsweise ein Bytecode-System für Pascal nicht die Regel, sondern die Ausnahme. Für

Smalltalk und Java ist der Bytecode jedoch Bestandteil der Architektur der Sprache und damit zwingend vorgeschrieben.

Der Bytecode hat die folgenden Eigenschaften:

- Er besteht **aus Instruktionen** (**Befehlen**) für eine **virtuelle Maschine**.
- Eine **Instruktion** ist **ein Byte lang** – daher auch die Bezeichnung Bytecode für den Zwischencode.
- Der Bytecode ist **unabhängig von einer Rechner-Plattform** und damit unabhängig vom Betriebssystem und der Rechner-Hardware.
- Der Bytecode ist **maschinennah**. Die Phasen lexikalische Analyse, Syntaxanalyse, Semantikanalyse und Codegenerierung zum Bytecode sind bereits durchlaufen, sodass man näher am Maschinencode als am Quellcode ist.
- Der Bytecode spiegelt in der Regel die **Eigenschaften der Programmiersprache** wider. So kann es zum Beispiel spezielle Instruktionen für den Zugriff auf die Komponenten von Objekten geben.
- Der Bytecode wird durch einen **Bytecode-Interpreter** ausgeführt, der die Befehle der virtuellen Maschine ausführen kann, mit anderen Worten, der die virtuelle Maschine emuliert.
- Da ein Bytecode für die jeweilige Programmiersprache optimiert ist, ist das resultierende **Programm sehr kompakt**. Mit anderen Worten, Bytecode-Programme eignen sich gut für die Übertragung in Netzen.
- Bytecode ist **portabel**. Programme, die in Bytecode übersetzt wurden, sind auf jeder Rechner-Plattform lauffähig, für die ein Bytecode-Interpreter zur Verfügung steht.
- Da der **Interpreter** eine virtuelle Maschine emuliert und in die Emulation die Zugriffe auf physische Betriebsmittel wie Drucker oder Dateien eingebunden sind, kann in den Emulator sehr leicht eine **Sicherheitsschicht** eingezogen werden, die unter gewissen Umständen Zugriffe auf bestimmte Betriebsmittel verwehrt.

Das Konzept des Bytecodes stellt insgesamt gesehen eine Zwischenstufe zwischen der reinen Interpretation eines Quellcodes und der Ausführung von Instruktionen eines speziellen Prozessortyps dar.

Wird – wie bei anderen Sprachen – der reine Quellcode interpretiert, so stellen die Anweisungen im Quellcode die Instruktionen für eine virtuelle Maschine auf höherem Niveau dar.

Letztendlich wird der Zugewinn an Portabilität und Sicherheit durch den erhöhten Zeitaufwand für den Interpreter erkauft. Um diesen Nachteil zu kompensieren, kann man den Interpreter selbst als Hardware – als **Java-Prozessor** – realisieren, der direkt die Bytecode-Instruktionen ausführt, anstatt als Programm auf einem herkömmlichen Prozessor.

3.4.2 Die Java-Virtuelle Maschine

Die Java-Virtuelle Maschine übernimmt Aufgaben wie die Speicherverwaltung des Programms zur Laufzeit und ist für Ein-/Ausgabe-Operationen und für Interaktionen mit dem Betriebssystem zuständig. Zu den Aufgaben der Java-Virtuellen Maschine gehören alle Prüfungen der dynamischen Semantik, kurz eine ganze Reihe von

Fehlerroutinen. An besonderen Sprachmerkmalen wie Threads (parallele Prozesse) oder Exceptions (Ausnahmen) ist die Java-Virtuelle Maschine ebenfalls beteiligt. Die Sicherheitsverwaltung der Java-Virtuellen Maschine entscheidet zum Beispiel, auf welche Ressourcen – wie z. B. eine Festplatte – ein in Ausführung befindliches Programm zugreifen darf und auf welche nicht.

Die **Java-Virtuelle Maschine** bildet die **Laufzeitumgebung** (das **Laufzeitsystem**) für das auszuführende Programm. Sie stellt einem Programm zusätzliche Funktionen (Routinen) zur Verfügung, welche für die Ausführung des Programms benötigt werden. Dazu gehören unter anderem Funktionen zur Speicheranforderung oder Fehlererkennung.

Das **Laden der Klassen** in die Java-Virtuelle Maschine erfolgt ebenfalls durch die Java-Virtuelle Maschine. Beim Starten eines Programms wird die **Startklasse** dem Interpreter bekannt gemacht, der den **Klassenlader** verwendet, um die Startklasse in den Arbeitsspeicher zu laden. Dies geschieht durch den Aufruf von `java Klassenname` auf der Kommandozeile. Da ein Java-Programm nicht wie die meisten herkömmlichen Programme aus einer einzelnen ausführbaren Datei, sondern aus mehreren `.class`-Dateien besteht, werden die anderen Java-Klassen zur Laufzeit, also zu dem Zeitpunkt, an dem sie gebraucht werden, durch die Java-Virtuelle Maschine nachgeladen. Trifft also der Java-Interpreter auf eine Klasse, die nicht im Speicher der Java-Virtuellen Maschine ist, so benutzt er den **Klassenlader**, um diese Klasse in den Arbeitsspeicher nachzuladen.

3.4.3 Ausführen von Bytecode

Die Java-Virtuelle Maschine setzt den Java-Bytecode in Maschinencode des jeweiligen Prozessors um. Erfordert eine Bytecode-Instruktion den Aufruf einer Betriebssystemroutine, so wird ein Sprungbefehl an die Adresse, an der sich die Betriebssystemroutine im Speicher befindet, generiert. Alle Betriebssystemroutinen liegen dabei fertig kompiliert im entsprechenden Maschinencode bereit.

3.5 Das Java Development Kit

Die offizielle Quelle für das Java Development Kit (JDK), das Werkzeuge für die Entwicklung von Java-Programmen bereitstellt, ist die Firma Oracle. Im Internet ist diese erreichbar über die Adresse

`http://www.oracle.com/technetwork/java/index.html`

Von dort können alle Editionen wie z. B. die Standard Edition oder Enterprise Edition kostenlos bezogen werden. Für die Standard Edition kann dort unter der Rubrik Java SE (Standard Edition) für alle gängigen Rechner-Plattformen das JDK auf den eigenen Rechner geladen werden. Außerdem führen von diesen Seiten Links zu Herstellern professioneller Entwicklungssysteme oder Herstellern des JDK für weniger bekannte Rechner-Plattformen.

Um mit Java arbeiten zu können, muss es für das entsprechende Betriebssystem ein Java Development Kit geben.

Um weitere Beispiele zur Programmiersprache Java bzw. Lösungen für spezielle Probleme zu bekommen, stehen im Internet zahlreiche Foren und Webseiten zur Verfügung.

Die Größe des Arbeitsspeichers des Entwicklungsrechners sollte bei Windows 7 und Windows 8 128 MB nicht unterschreiten. Für die Komplettinstallation der JDK-Version 8 werden mindestens 400 MB Plattenplatz benötigt. Für ältere Versionen wie z. B. Java 7 oder Java 6 ist teilweise weniger Plattenplatz erforderlich.

Das JDK-Installations-File entpackt sich bei der Installation unter Microsoft Windows selbst. Standardmäßig werden die Dateien in das Verzeichnis `Java` im Programmverzeichnis von Windows installiert (z. B. `C:\Programme\Java`). Das Verzeichnis kann aber im Installationswizard angepasst werden. Ebenfalls kann dort das Installieren der Java-Sourcen[47] für die Java-API unterdrückt werden. Bei der Installation werden standardmäßig zwei Java Runtime Environments (JREs) installiert. Das eine JRE, das sogenannte **private JRE**, wird von den Tools des JDK wie z. B. dem Java-Compiler benötigt und innerhalb des JDK-Verzeichnisses installiert. Das andere JRE, das sogenannte **öffentliche JRE**, wird parallel zum JDK in das Programmverzeichnis installiert und kann von anderen Java-Applikationen genutzt werden. Um Plattenplatz zu sparen und Verwirrung wegen zu vieler Java-Installationen zu vermeiden, kann im Installationswizard die Installation des öffentlichen Java Runtime Environment (JRE) verhindert werden.

Ein Java Runtime Environment enthält außer der JVM alle notwendigen Programmdateien und Bibliotheken für eine Java-VM. Die Entwicklungstools wie z. B. `javac.exe` sind nicht in der JRE enthalten.

Nach der Standardinstallation befinden sich im Programmverzeichnis von Windows die beiden Verzeichnisse `jre8` und `jdk1.8.0_xx` für die Version 8. Das `xx` steht hierbei stellvertretend für die Update-Version, wie beispielsweise `51`. Im Verzeichnis `jdk1.8.0_xx` werden mehrere Verzeichnisse angelegt, unter anderem ein Verzeichnis namens `bin`. In diesem Verzeichnis befinden sich alle Programmdateien wie zum Beispiel `javac.exe` oder `java.exe` des JDK. Im Unterverzeichnis `db` befindet sich eine Installation einer Apache Derby Datenbank, die ausschließlich in Java geschrieben ist und Standard-SQL und JDBC unterstützt. Im Unterverzeichnis `include` sind Header-Dateien abgelegt, die zur Einbindung von C-Programmen in Java-Programme benötigt werden. In **Archive**[48] zusammengefasste Java-Klassendateien befinden sich in dem Unterverzeichnis `lib`. Im Basisverzeichnis des JDK

---

[47] Die Java-Bibliotheken mit den `class`-Files der API werden aber immer installiert.

[48] **Archive** dienen zur Zusammenstellung mehrerer Dateien – in gepackter oder ungepackter Form – in einer einzigen Datei. Archivdateien haben die Dateierweiterung `jar`. `jar`-Dateien können außer den Bytecode-Daten der Java-Klassen beispielsweise auch Bilder und `xml`-Dateien enthalten. Es können aber auch weitere `jar`-Dateien in einer `jar`-Datei verpackt sein. Häufig enthält ein Archiv auch eine Manifest-Datei, in der beispielsweise Informationen wie die Manifest-Version oder der Name der Start-Klasse enthalten sind sowie bei signierten Archiven die Signatur-Informationen für Klassen.

liegt die Datei `src.zip`, die alle Java-Sourcen der öffentlichen Klassen[49] der VM enthält. Diese Java-Sourcen ermöglichen dem Programmierer, bei Bedarf die Implementierung der Klassen zu betrachten.

3.5.1 Setzen der Umgebungsvariablen `JAVA_HOME`

Die Umgebungsvariable `JAVA_HOME` wird von vielen Programmen verwendet, die auf dem JDK aufbauen, um das Installationsverzeichnis des JDK, d. h. das Verzeichnis, in dem der Java Compiler und der Java Interpreter bzw. die von der VM benötigten Bibliotheken bei der Installation abgelegt wurden, herausfinden zu können.

Die Umgebungsvariable `JAVA_HOME` zeigt immer auf das Basisverzeichnis des JDK, beispielsweise auf

`C:\Programme\Java\jdk1.x.0_xx.`

Unter **Windows Betriebssystemen** ist hierbei eine neue Umgebungsvariable `JAVA_HOME` anzulegen. Bei den aktuellen Versionen der Windows Betriebssysteme (ab Windows 7) muss in der Systemsteuerung zuerst `System und Sicherheit`, gefolgt von `System` ausgewählt werden. Anschließend ist links auf `Erweiterte Systemeinstellungen` zu klicken, um dann unter `Umgebungsvariablen` eine neue Variable für `JAVA_HOME` anzulegen.

Unter **UNIX/Linux** kann die Umgebungsvariable `JAVA_HOME` beispielsweise wie folgt gesetzt werden:

```
export JAVA_HOME=/usr/java/jdk1.x.0_xx
```

3.5.2 Setzen der Umgebungsvariablen `PATH`

Die `PATH`-Variable wird für alle Java-Tools benötigt, die im `bin`-Verzeichnis des JDK vorhanden sind. Dazu gehören `java`, `javac`, `javah`, `keytool`, `rmic` etc.

Um diese **Java-Programmdateien direkt von der Kommandozeile aufrufen zu können**, muss der Suchpfad für ausführbare Dateien, der in der `PATH`-Variable hinterlegt ist, um das entsprechende `bin`-Verzeichnis erweitert werden. Unter Windows geschieht dies wie folgt:

[49] Mit öffentlichen Klassen sind die Klassen gemeint, die z. B. unter den Packages `java` und `javax` liegen.

- **Temporäre Erweiterung des Suchpfads**

  Will man die `PATH`-Variable temporär – also nur in der aktuellen Sitzung des Kommandozeilen-Interpreters – um den Pfad zu den Java-Programmdateien erweitern, muss der Befehl

  ```
  set PATH=%path%;C:\Programme\Java\jdk1.x.0_xx\bin
  ```

  in der Kommandozeile eingegeben werden, wobei `x` durch `6` für die Version 6, `7` für die Version 7 bzw. `8` für die Version 8 ersetzt werden muss. `xx` muss durch die installierte Update-Version ersetzt werden, z. B. `51`. Nach dem Schließen des Fensters der Kommandoebene, d. h. der Shell in UNIX/Linux bzw. der Windows-Konsole unter Windows, ist diese Einstellung wieder ungültig.

- **Permanente Erweiterung des Suchpfads**

  Um den Pfad zu den Java-Programmdateien permanent in die `PATH`-Variable aufzunehmen, muss diese erweitert werden. Hierfür muss – wie schon beim Setzen von `JAVA_HOME` – der Dialog zum Editieren von Umgebungsvariablen geöffnet werden. Dort kann unter `Systemvariablen` die Variable `PATH` ausgewählt und der Eintrag

  ```
  ;%JAVA_HOME%\bin
  ```

  am Ende hinzufügt werden. Durch diese Änderung steht nun der neue Wert allen Benutzern zur Verfügung. Soll die `PATH`-Variable nur für den aktuellen Benutzer verändert werden, so muss der Wert unter `Benutzervariablen` gesetzt werden.

Es empfiehlt sich, ein eigenes Arbeitsverzeichnis zu erstellen, damit die installierten Programme des JDK und die eigenen Anwendungsprogramme nicht gemischt abgelegt werden. Damit lässt sich die Erstellung von Sicherungskopien des eigenen Programmcodes in einfacher Weise durchführen.

3.5.3 Setzen der Umgebungsvariablen `CLASSPATH`

Die Umgebungsvariable `CLASSPATH` wird vom Compiler `javac` des JDK sowie vom Interpreter `java` des Java-Laufzeitsystems JRE benutzt, um den Weg zu **benutzerdefinierten Klassen**, Sourcen und zu anderen erforderlichen Dateien zu finden, die nicht Teil der Java-API sind.

Auf die Umgebungsvariable `CLASSPATH` wird in Kapitel 12.5.3 detaillierter eingegangen.

Wird der `CLASSPATH` nicht explizit gesetzt, wird das aktuelle Verzeichnis durchsucht. Die Verzeichnisse, die die Klassen der Java-API enthalten, werden vom Compiler `javac` sowie vom Interpreter `java` grundsätzlich durchsucht. Deshalb ist es normalerweise nicht erforderlich, den `CLASSPATH` explizit zu setzen. Sollen jedoch Klassen oder Klassenbibliotheken benutzt werden, die sich nicht innerhalb der aktuellen Verzeichnisstruktur des aktuellen Verzeichnisses befinden, so ist es erforderlich, den

`CLASSPATH` explizit zu setzen. Wie der `CLASSPATH` zu setzen ist, hängt vom jeweiligen Betriebssystem ab. Im Folgenden zwei Beispiele:

```
export CLASSPATH=/wrk:.            (UNIX/Linux)[50]
set CLASSPATH=C:\wrk;.             (Windows)
```

Der Punkt `.` hinter dem Doppel- bzw. Strichpunkt steht dabei für das aktuelle Verzeichnis. Es ist auch möglich, mehrere verschiedene Pfade im `CLASSPATH` anzugeben. Dann kann der Compiler bzw. der Interpreter nach den Klassen in verschiedenen Verzeichnissen suchen.

Unter Windows werden alternative Pfadangaben durch ein Semikolon getrennt, unter UNIX-Systemen durch einen Doppelpunkt.

Der `CLASSPATH` kann auch direkt beim Kompilieren beziehungsweise beim Aufruf des Interpreters für ein Java-Programm angegeben werden. Das folgende Beispiel zeigt dies für das Kompilieren:

```
javac -classpath C:\wrk;. HelloWorld.java
```

3.6 Begriffe im Java-Umfeld

Im Folgenden werden einige wichtige Java-Begriffe, die oft verwechselt werden, vorgestellt:

- **Java-Anwendungen** sind Programme, die in Java geschrieben sind und von der Kommandozeile eines Rechners aus gestartet werden können.
- **Java-Applets** sind Programme, die in einer **HTML-Seite** enthalten sind und zusammen mit einer HTML-Seite von einem **Web-Server** zu einem Rechner geschickt werden. Sie werden von einem Java-fähigen Webbrowser auf der Maschine des Nutzers ausgeführt, wenn die HTML-Seite geladen wird. Der Aufbau von Java-Anwendungen und Applets ist prinzipiell verschieden. Applets haben an Bedeutung verloren, seitdem es Servlets und JavaServer Pages[51] gibt.
- **Java-Servlets** sind Programme, die auf einem Web-Server laufen und dynamisch HTML-Seiten für einen Webbrowser erzeugen.
- **JavaServer Pages** (**JSP**) ermöglichen das Einbinden von Java Code in HTML-Seiten. Dadurch kann der Seiteninhalt ähnlich wie bei der Verwendung von Servlets dynamisch generiert werden. JSPs bauen auf der Servlet-Technologie auf, sind aber einfacher in der Anwendung.
- **JavaServer Faces** (**JSF**) ist ein serverseitiges Framework zur Programmierung von Webanwendungen und ihren Benutzeroberflächen.
- **JavaScript** ist eine von Java unabhängige Skriptsprache. Es stellt eine Erweiterung von HTML zum Aufbau von Internet-Seiten dar und sollte nicht mit Java verwechselt werden.

[50] Dieses Kommando gilt für die Bourne-Shell (sh).
[51] Eine Alternative zu JSP sind die JavaServer Faces (JSF).

3.7 Zusammenfassung

Die Sprachkonzepte von Java (siehe Kapitel **3.1**) haben sich bereits in anderen Programmiersprachen bewährt. Neu ist ihre Zusammenstellung und Ausprägung.

Die Sprachkonstrukte für das "Programmieren im Kleinen" wie die einfachen Datentypen und die Operatoren sind an C++ angelehnt. Der Gedanke, der hinter der Virtuellen Maschine steckt, ist, unabhängig von der jeweiligen Rechner-Plattform zu werden. Unter Rechner-Plattform wird hierbei die Kombination von Betriebssystem und zugehöriger Rechner-Hardware verstanden. Pakete als Gruppierung kleinerer Bausteine wurden bereits in Ada verwendet.

Java-Programme sind einfach, stabil, objektorientiert, verteilbar, sicher und portierbar (siehe Kapitel **3.2**).

Zu einer Java-Plattform (siehe Kapitel **3.3**) gehören:

- die Programmiersprache Java,
- Werkzeuge wie zum Beispiel der Java-Compiler (javac),
- die Java-Virtuelle Maschine (JVM) für eine Rechner-Plattform
- und eine umfassende Klassenbibliothek.

In Kapitel **3.3** werden die unterschiedlichen Ausprägungen der Java-Plattform, d. h. die unterschiedlichen Plattformen für verschiedene Betriebssysteme und zum anderen die unterschiedlichen Editions Java SE, Java EE, Java ME und Java Card vorgestellt. Es wird auf die Java-Virtuelle Maschine mit Bytecode-Interpreter und JIT-Compiler eingegangen wie auch auf die Grobstruktur der Java-Klassenbibliothek.

Ein Bytecode-Interpreter und ein Just-in-Time-Compiler sind Bestandteile der Ausführungseinheit der virtuellen Maschine. Weitere wichtige Komponenten der JVM sind:

- der Klassenlader,
- die Speicherverwaltung,
- der Bytecode-Verifizierer
- und die Java-Standardbibliothek.

In Java wird kein ausführbares Programm erzeugt. Bei Bedarf werden die erforderlichen Klassen durch den Klassenlader geladen.

Beim Kompilieren werden die Phasen

- Lexikalische Analyse,
- Syntaxanalyse,
- Semantische Analyse
- und Erzeugung von Bytecode

durchlaufen.

Bei der lexikalischen Analyse wird versucht, in der Folge der Zeichen eines Programms die Wörter der Sprache – das sind die kleinsten Einheiten einer Sprache, die eine Bedeutung besitzen – zu erkennen.

Im Rahmen der Syntaxanalyse wird geprüft, ob die bei der lexikalischen Analyse ermittelte Symbolfolge eines zu übersetzenden Programms zu der Menge der zulässigen Symbolfolgen gehört.

Die semantische Analyse versucht, die Bedeutung der Wörter herauszufinden. Neben der Überprüfung der Verwendung von Namen im Rahmen ihrer Gültigkeitsbereiche[52] spielt die Überprüfung von Typverträglichkeiten bei Ausdrücken eine Hauptrolle.

Bei der Codegenerierung wird Bytecode für die Java-Virtuelle Maschine erzeugt. Durch das Bytecode-Konzept ist jedes Java-Programm auf jeder Rechner-Plattform mit einer Java-Virtuellen Maschine ausführbar. Die Java-Virtuelle Maschine bildet die Laufzeitumgebung (das Laufzeitsystem) für das auszuführende Programm. Sie stellt einem Programm zusätzliche Funktionen (Routinen) zur Verfügung, welche für die Ausführung des Programms benötigt werden. Dazu gehören unter anderem Funktionen zur Speicheranforderung oder Fehlererkennung.

Das Java-Development Kit (siehe Kapitel **3.4**) stellt Werkzeuge für die Entwicklung von Java-Programmen bereit.

In Kapitel **3.5** wird beschrieben, von wo das Java Development Kit bezogen werden kann und wie es installiert werden muss. Kapitel **3.6** schließlich geht auf Begriffe im Java-Umfeld ein, die häufig durcheinander gebracht werden.

3.8 Übungen

Aufgabe 3.1: Verständnisfragen

3.1.1 Was sind die besonderen Eigenschaften von Java?
3.1.2 Was gehört zur Java-Plattform?
3.1.3 Erläutern Sie das Konzept des Zwischencodes (Bytecodes).
3.1.4 Nennen Sie die drei Bereiche, in welche die Java-Klassenbibliothek der Standard Edition eingeteilt wird.
3.1.5 Nennen Sie die vier Schritte des Kompiliervorgangs.
3.1.6 Was macht der Java-Compiler bei der lexikalischen Analyse?
3.1.7 Was passiert bei der Syntaxanalyse?
3.1.8 Was wird bei der Codegenerierung generiert?

Aufgabe 3.2: Erste Programmierversuche

3.2.1 Beschreibung der Java-Klassen

Eine Beschreibung aller Java-Klassen befindet sich auf der Homepage von Oracle. Diese Dokumentation kann heruntergeladen werden, steht aber auch online zur Verfügung.

a) Aufrufen der Java-API-Dokumentation

Die Java-API-Dokumentation finden Sie unter der folgenden Adresse:

`http://download.oracle.com/javase/index.html`

[52] Siehe Anhang B

Starten Sie Ihren Browser und rufen Sie die oben genannte Seite auf. Unter dem Link `API Documentation` wird die API-Spezifikation der jeweiligen Java-Version aufgerufen. Zum Zeitpunkt des Erscheinens dieses Buches ist dies die Version 8. Der folgende Link führt Sie direkt zur API-Spezifikation:

http://download.oracle.com/javase/8/docs/api/index.html

Machen Sie sich mit der Klassendokumentation etwas vertraut, indem Sie etwas herumspielen.

b) `System.out.println()`

Um die Zeichenkette `Hello World` auf der Kommandozeile auszugeben, wird `System.out.println()` benutzt:

```
System.out.println ("Hello World");
```

Finden Sie mit Hilfe der Klassendokumentation schrittweise heraus, was sich hinter dieser Anweisung verbirgt. Hierzu folgende Hinweise:

- `System` ist eine Klasse im Paket `java.lang`.
- `out` ist der Name einer Klassenvariablen in der Klasse System. Die Referenz `out` zeigt auf ein Objekt der Klasse `PrintStream`. Die Klasse `PrintStream` befindet sich im Paket `java.io`. Dieses Objekt der Klasse `PrintStream` besitzt die Fähigkeit, in die Standardausgabe zu schreiben, die in der Regel auf den Bildschirm zeigt.
- `println()` ist eine Methode der Klasse `PrintStream`.

Beantworten Sie folgende Frage:

Welche weiteren Methoden gibt es in der Klasse `PrintStream` – außer der Methode `println()`?

3.2.2 Die Klasse `String`

Die Klasse String `befindet` sich genauso wie die Klasse `System` im Paket `java.lang`. Beantworten Sie die folgenden Fragen mit Hilfe der Klassendokumentation:

a) Welche Methode liefert einen Rückgabewert vom Typ `char` für einen bestimmten Index eines Strings? Hinweis zum Index eines Strings: Im String `"Hallo"` hat das Zeichen `'H'` den Indexwert `0`.
b) Welche Methode gibt die Länge eines Strings zurück?
c) Welche Aufgabe erfüllt die Methode `trim()`?
d) Welche Aufgabe erfüllt die Methode `replace()`?

3.2.3 Erste Versuche mit dem Programm "Hello, world"

a) Installieren Sie als erstes das Java Development Kit, so wie in Kapitel 3.5 erläutert.
b) Schreiben Sie das "Hello, world"-Programm aus Kapitel 1.1 in einem von Ihnen gewählten Editor.
c) Kompilieren Sie die Java-Datei mit Hilfe des Java-Compilers `javac`.
d) Führen Sie die kompilierte `.class`-Datei aus.

Kapitel 4

Einfache Beispielprogramme

4.1 Lokale Variablen, Ausdrücke und Schleifen
4.2 Erzeugen von Objekten
4.3 Initialisierung von Objekten mit Konstruktoren
4.4 Schreiben von Instanzmethoden
4.5 Zusammengesetzte Objekte
4.6 Selbst definierte Untertypen durch Vererbung
4.7 Ein- und Ausgabe am Bildschirm
4.8 Zusammenfassung
4.9 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_4

4 Einfache Beispielprogramme

Mit dem Programm "Hello, world" in Kapitel 1.1 und dem Programm "Punkt" in Kapitel 2.1 haben Sie bereits erste Erfahrungen im Programmieren erworben. Programmieren kann viel Spaß bereiten. Im Folgenden sollen deshalb weitere kurze, aussagekräftige Programme vorgestellt werden, damit Sie sich zunächst spielerisch voranarbeiten, um dann auch Augen und Ohren für die erforderliche Theorie zu haben. Alle Programme des Buches befinden sich auch auf der Internetseite des Verlags unter www.springer.com. Sie brauchen also die Programme nicht abzutippen.

Als Einstieg werden in Kapitel 4.1 einfache Programmbeispiele vorgestellt, die jedem C-Programmierer bekannt sind. Sie wurden in ihrer Formulierung in C durch Kernighan und Ritchie [Ker78] weltberühmt. Die Sprache Java ist zwar an C++ angelehnt, wobei diese Sprache wiederum auf C aufsetzt, dennoch sind die in diesem Kapitel aufgeführten Beispiele für die Programmierung in Java eigentlich untypisch. Es ist aber trotzdem sinnvoll, solche einfachen Beispiele zu betrachten, um mit **Variablen**, **Konstanten**, **Schleifen**, **Berechnungen** und der **Ein- und Ausgabe** vertraut zu werden. Die Wucht solch klassischer Beispiele ist so groß, dass mit der Version Java 5 die Methode `printf()` mit Formatelementen zur Steuerung der Ausgabe analog zu C eingeführt wurde (siehe Kapitel 4.7).

Kapitel 4.2 erläutert, was beim Erzeugen von Objekten passiert, und Kapitel 4.3, wie **Objekte mit** Hilfe von **Konstruktoren initialisiert** werden. Da bis zu dieser Stelle Methoden stets gebrauchsfertig vorgegeben waren, wird in Kapitel 4.4 erklärt, wie eine **Methode definiert** und **aufgerufen** wird und wie Parameter an Methoden übergeben werden können, d. h., wie **formale und aktuelle Parameter** zusammenhängen. In Kapitel 4.5 wird gezeigt, wie durch eine **Aggregation** "Groß"-Objekte auf "Klein"-Objekte zeigen können. Kapitel 4.6 befasst sich schließlich mit dem Subtyping von Klassen durch den Vererbungsmechanismus, d. h. mit dem Schreiben von Klassen, die Spezialisierungen anderer Klassen sind. Kapitel 4.7 behandelt das Einlesen von Zeichen über die Tastatur. Zudem wird die Methode `printf()` für Ausgaben sowie die Klasse `Scanner` für die Verarbeitung von Eingaben vorgestellt. Anhand eines Beispiels wird das Einlesen von Zeichenketten und von ganzen Zahlen mit Hilfe der Klasse `Scanner` erläutert.

4.1 Lokale Variablen, Ausdrücke und Schleifen

Als erstes Programm wird das durch C berühmte Temperaturwandlungsprogramm von Kernighan und Ritchie nun in Java vorgestellt. Dieses Programm soll eine Temperaturtabelle zur Umrechnung von Fahrenheit-Graden in Celsius-Grade erzeugen. Dieses Programm vermittelt erste Erfahrungen mit einer **Schleife** und mit der Berechnung von **Ausdrücken**. In der ersten Variante dieses Programms werden **symbolische Konstanten**[53] für die untere Grenze, die obere Grenze und die Schritt-

[53] **Symbolische Konstanten** sind Konstanten, die einen Namen tragen. An die Stelle eines Namens setzt der Compiler dann die der symbolischen Konstanten zugeordnete literale Konstante ein, also z. B. eine "nackte" Zahl wie die Zahl 10.

weite in Fahrenheit verwendet. Für die Zwischenspeicherung der Temperaturwerte in Celsius und Fahrenheit werden **Variablen** verwendet, um zu einem Wert in Fahrenheit jeweils den entsprechenden Celsius-Wert zu berechnen.

Hier das Programm:

```
// Datei: Fahrenheit.java

// Klasse zur Wandlung von Temperaturen von Fahrenheit nach Celsius
public class Fahrenheit
{
   public static void main (String[] args)
   {
      // Konstanten

      final int UPPER = 300;    // obere Grenze
                                // UPPER ist eine symbol. Konstante
                                // 300 ist eine literale Konstante
      final int LOWER =   0;    // untere Grenze
      final int STEP =   20;    // Schrittweite

      // Variablen

      int fahr;                 // Definition der lokalen Variablen
                                // fahr für die Temperatur in
                                // Fahrenheit
      int celsius;              // Definition der lokalen Variablen
                                // celsius für die Temperatur in
                                // Celsius

      // Anweisungen

      fahr = LOWER;             // als Anfangswert wird fahr
                                // der Wert 0 zugewiesen
      while (fahr <= UPPER)
      {
         celsius = 5 * (fahr - 32) / 9;
                                // nach dieser Formel berechnet sich
                                // der Celsius-Wert aus einem
                                // Fahrenheit-Wert

         System.out.print (fahr);
                                // der Wert von fahr wird auf
                                // den Bildschirm ausgegeben
         System.out.print ("     ");
                                // Leerzeichen in derselben Zeile
         System.out.println (celsius);
                                // Der Wert von Celsius wird in
                                // derselben Zeile ausgegeben.
                                // Anschließend springt der Cursor
                                // zum Anfang der nächsten Zeile.

         fahr = fahr + STEP;    // Der nächste Wert von fahr
                                // wird berechnet
      }
   }
}
```

Hier ein Auszug der Programmausgabe:

```
0      -17
20      -6
40       4
```

Die in Java erzeugte Tabelle hat nicht dieselbe schön formatierte Form wie in C, wo in der Tabelle Einer unter Einern und Zehner unter Zehnern stehen. Der Grund hierfür ist, dass die Ein- und Ausgabemöglichkeiten in Java schwerpunktmäßig zunächst auf eine grafische Benutzerschnittstelle ausgerichtet wurden, die fensterorientiert arbeitet. Eine zeilenorientierte Ein- und Ausgabe auf den ganzen Bildschirm galt ursprünglich nicht mehr als Stand der Technik und hatte aus diesem Grund für die Entwickler von Java in den ersten Jahren von Java keine Priorität. Daher gibt es für die Methode `println()` keine Formatierungsmöglichkeiten. Da dies jedoch bemängelt wurde, wurde mit der Version Java 5 durch die Einführung der Methode `printf()` (siehe Kapitel 4.7) Abhilfe geschaffen.

Die Größen `LOWER`, `UPPER` und `STEP` sind **symbolische Konstanten**. Namen symbolischer Konstanten werden **üblicherweise in Großbuchstaben** geschrieben.

Bitte beachten Sie:

- Die Division `5/9` ergibt Null (**ganzzahlige Division ohne Rest**). Daher wird die Zahl `5` zunächst mit `(fahr - 32)` multipliziert, damit vor der Division eine große Zahl entsteht. Im zweiten Schritt wird dann durch die Zahl `9` geteilt.
- Der **Zuweisungsoperator** ist das Zeichen `=`. Für den **Vergleichsoperator** "ist gleich" muss die Notation `==` verwendet werden.
- Die `while`-Schleife wird abgearbeitet, solange die Bedingung `fahr <= UPPER` den Wert `true` hat.
- Die Methode `println()` positioniert im Gegensatz zu `print()` im Anschluss an die Ausgabe des auszugebenden Ausdrucks den Cursor des Bildschirms an den Anfang der nächsten Zeile.
- Mit `println()` bzw. `print()` kann nicht nur – wie im Falle von `"Hello, world"` – Text ausgegeben werden, sondern es können auch die Werte von Variablen ausgegeben werden.

Die Methoden `println()` und `print()` sind **überladen**. Überladen bedeutet, dass es zum selben Methodennamen verschiedene Methoden gibt, die sich im Typ ihrer Parameter, aber nicht im Rückgabetyp unterscheiden. Für jeden Typ gibt es eine eigene Methode und automatisch wird die richtige Methode aufgerufen. So können an die Methoden `println()` bzw. `print()` beispielsweise auch `int`- oder `float`-Variablen übergeben werden wie z. B. die `int`-Variable `fahr` in `System.out.print (fahr)`.

Im Folgenden werden einige andere Varianten dieses Programms vorgestellt. Dabei soll in der nächsten Variante **ohne symbolische Konstanten** und nur mit **Integer-Größen**, d. h. mit **ganzzahligen Variablen und Ausdrücken** gearbeitet werden.

Hier die zweite Variante des Programms:

```
// Datei: Fahrenheit2.java

public class Fahrenheit2
{
   public static void main (String[] args)
   {
      int fahr;

      for (fahr = 0; fahr <= 300; fahr = fahr + 20)
      {
         System.out.print (fahr);
         System.out.print ("    ");
         System.out.println (5 * (fahr - 32) / 9);
      }
   }
}
```

Beachten Sie hierbei die folgenden Punkte:

- In der `for`-Schleife stellt `fahr = 0` den Beginn der Schleife dar, `fahr <= 300` die Bedingung, wie lange die Schleife durchgeführt wird, und `fahr = fahr + 20` den nächsten Wert, für den die Schleife durchgeführt wird.
- Die Verwendung des Ausdrucks `5 *(fahr - 32)/9` in der `println()`-Methode anstelle der Variablen `celsius` wie im vorherigen Beispiel ist beispielhaft für die allgemeine Regel:

> In jedem Zusammenhang, in dem der Wert einer Variablen eines bestimmten Typs stehen kann, kann auch ein komplizierter Ausdruck von diesem Typ stehen.

In der letzten Variante soll mit `celsius` als `float`-**Variable** und mit einer `while`-**Schleife** gearbeitet werden:

```
// Datei: Fahrenheit3.java

public class Fahrenheit3
{
   public static void main (String[] args)
   {
      // Konstanten
      final int UPPER = 300;    // obere Grenze
      final int LOWER =   0;    // untere Grenze
      final int STEP =   20;    // Schrittweite

      int fahr;
      float celsius;

      fahr = LOWER;

      while (fahr <= UPPER)
      {
         celsius = (float) (5.0 / 9) * (fahr - 32);
```

```
            System.out.println (fahr + "      " +  celsius);
            fahr = fahr + STEP;
        }
    }
}
```

Hier ein Auszug der Programmausgabe:

```
0       -17.777779
20       -6.666667
. . . . .
280       137.77779
300       148.8889
```

Beachten Sie:

- Die Konstante `5.0` ist vom Typ `double`. Damit ist

  `(5.0 / 9) * (fahr - 32)`

  vom Typ `double` – der Compiler muss die Zahl `9` und den Ausdruck `(fahr - 32)` ohne eine Anweisung des Programmierers **implizit** in den Gleitpunkttyp `double` wandeln. Man spricht von **impliziter Typkonvertierung**, wenn die Konvertierung vorgenommen wird, ohne dass der Programmierer dies explizit in der Programmiersprache formulieren muss. Die Konvertierung erfolgt automatisch, wenn ein Wert eines schmäleren Typs wie z. B. die Zahl `9` mit einem Wert eines breiteren Typs wie z. B. der Zahl `5.0` verknüpft wird (siehe Kapitel 7.7).

- Bei der Zuweisung

  `celsius = (float) (5.0 / 9) * (fahr - 32)`

  muss eine **explizite Typkonvertierung** von `double` nach `float` stattfinden, da die Variable `celsius` vom Typ `float` ist und das Ergebnis des Ausdrucks `(5.0 / 9) * (fahr - 32)` vom Typ `double` ist. Es soll also ein breiterer Typ in einen schmäleren Typ gewandelt werden. Diese Typkonvertierung erfolgt mit Hilfe des sogenannten **cast-Operators** (siehe Kapitel 7.7.1), der von der Form `(datentypname)` ist. Dabei wird der hinter dem cast-Operator stehende Ausdruck in den Datentyp `datentypname` des cast-Operators gewandelt.

- Mit `println()` bzw. `print()` werden auch Zeichenketten, die mit Variablen durch den **Verkettungsoperator** + verknüpft sind, ausgegeben.

  So kann anstelle von

  ```
  System.out.print (fahr);
  System.out.print ("     ");
  System.out.println (celsius);
  ```

  in knapper Form geschrieben werden:

  ```
  System.out.println (fahr + "     " + celsius);
  ```

Ist eine Variable mit einem String durch den Verkettungsoperator + verknüpft, so wird der **Wert der Variablen automatisch in einen String gewandelt** und an den **vorhandenen String angehängt**.

Bild 4-1 visualisiert die Programmstruktur für die drei behandelten Fahrenheit-Programme am Beispiel der Klasse `Fahrenheit3`:

Bild 4-1 Visualisierung der Programmstruktur

In allen drei Programmvarianten liegt jeweils nur eine Wrapper-Klasse[54] für `main()` vor, d. h., die Methode `main()` wird in eine Klasse eingehüllt. Die Methode `main()` in Bild 4-1 ist unterstrichen, da es eine sogenannte Klassenmethode ist (siehe Kapitel 10.2).

4.2 Erzeugen von Objekten

Im Beispiel der Klasse `Punkt` in Kapitel 2.5 wurde ein Punkt erzeugt, indem in der Methode `main()` der Klasse `Punkt` die folgende Anweisung geschrieben wurde:

```
Punkt p = new Punkt();        // hiermit wird ein Punkt erzeugt
```

Die Erzeugung lässt sich auch in 2 Schritten durchführen:

```
Punkt p;
p = new Punkt();
```

Da

```
Punkt p;
```

im Methodenrumpf steht, bedeutet dies, dass in der Methode eine Variable `p` vom Typ der Klasse `Punkt` angelegt wird. Eine solche Variable ist nur innerhalb ihres Blocks, der charakterisiert wird durch eine öffnende und eine schließende geschweifte Klammer, und ggf. in sogenannten inneren Blöcken ihres Blocks sichtbar.[55] Sie wird als **lokale Variable** bezeichnet.

Eine Variable `p` einer Klasse `Punkt` ist in Java eine Referenz auf ein Objekt der Klasse `Punkt`. Eine Referenz ist nichts anderes als ein Zeiger. In Java spricht man aber nicht von Zeigern, sondern von Referenzen, weil man mit Zeigern in den Programmiersprachen C und C++ zum Teil sehr schlechte Erfahrungen gemacht hat.

---

[54] Mit Hilfe einer Wrapper-Klasse wird ein nicht objektorientiertes Konstrukt – hier die Methode `main()` – in die Gestalt einer Klasse gebracht (eingehüllt = engl. wrapped).

[55] Blöcke können geschachtelt werden.

Referenzen in Java unterscheiden sich von den Zeigern in C und C++ dadurch, dass **Javas Referenzen**[56] **nicht** der **physischen Adresse** im Speicher entsprechen und dass diese **keine Zeigerarithmetik** haben.

Es ist nur möglich, auf ein selbst erzeugtes Objekt zu zeigen. Es ist aber nicht möglich, wie in C oder C++ den Zeiger zu verändern und zu beginnen, von der entsprechenden Stelle an beliebig über den Arbeitsspeicher zu laufen und auf andere Speicherzellen zuzugreifen.

Mit

```
Punkt p;
```

wird in Java eine **Referenzvariable** mit dem Namen `p` angelegt.

Diese Referenzvariable kann nur auf Objekte der Klasse `Punkt` zeigen, nicht aber auf Objekte anderer Klassen wie z. B. der Klassen `Pferd` oder `Blume`. Nun zur Anweisung

```
p = new Punkt();
```

Der `new`-Operator plus runde Klammern wird hier verwendet, um Objekte zu erzeugen. Mit

```
new Punkt
```

erfährt der `new`-Operator, dass er ein Objekt der Klasse `Punkt` erzeugen soll. Dies ist der erste Schritt der Anweisung

```
p = new Punkt();
```

Der Aufruf des `new`-Operators mit seinem Parameter `Punkt` wurde zur besseren Übersicht unterstrichen. Im 2. Schritt wird der Konstruktor[57] ohne Parameter, der sogenannte parameterlose Konstruktor[58,59] aufgerufen:

```
p = new Punkt();
```

Der 2. Schritt ist ebenfalls unterstrichen. Das Elegante an dieser Notation ist, dass `Punkt` zum einen ein Parameter für den `new`-Operator ist, damit dieser weiß, was für ein Objekt er überhaupt erzeugen soll, und dass zum anderen mit Absicht der Name des Konstruktors gleich dem Klassennamen gewählt wurde und damit `Punkt()` dem Aufruf des parameterlosen Konstruktors entspricht.

Durch `new Punkt()` wird als Rückgabewert die Referenz auf die Stelle des Speichers zurückgegeben, an der das Objekt erzeugt wurde. Dieser Rückgabewert wird

---

[56] Referenzen entsprechen aus Sicherheitsgründen logischen Adressen. Erst durch die Java-Virtuelle Maschine wird aus einer logischen Referenz eine physische Adresse im Arbeitsspeicher.

[57] Ein Konstruktor dient meist zur Initialisierung.

[58] Ein parameterloser Konstruktor hat keine Parameter. Er dient zur Initialisierung eines Objekts.

[59] Auf die Unterschiede zwischen einem Default-Konstruktor und einem selbstgeschriebenen parameterlosen Konstruktor wird in Kapitel 10.4.4.3 eingegangen.

im Rahmen einer Zuweisung der Referenzvariablen `p` zugewiesen. Damit kann man im weiteren Programmverlauf mit Hilfe der Referenz `p` auf das Objekt zugreifen.

4.3 Initialisierung von Objekten mit Konstruktoren

Das folgende **Beispiel** zeigt die Klasse `Punkt2` und die Klasse `TestPunkt2`. Bei professionellen Programmen ist es üblich, für jede Klasse eine eigene Testklasse zu schreiben. Die Klasse `TestPunkt2` dient also dazu, die Klasse `Punkt2` auszutesten. Die Testklasse `TestPunkt2` ist wieder eine Wrapper-Klasse, die eine einzige Methode, die Methode `main()` enthält. Die Klasse `Punkt2` enthält keine `main()`-Methode. Dies bedeutet, dass die Klasse `Punkt2` nicht gestartet werden kann. Das ist vollkommen normal. Ein Punkt stellt ein Objekt dar, das benutzt wird, und das nicht von selbst aktiv handelt. Das Erzeugen von Objekten, d. h. Punkten, und der Aufruf von Methoden der Objekte der Klasse `Punkt` findet in der `main()`-Methode der Testklasse statt.

In der Regel wird jede Klasse in einer eigenen Datei gespeichert.

Daher liegt die Klasse `Punkt2` in der Datei `Punkt2.java` und die Klasse `TestPunkt2` in der Datei `TestPunkt2.java`.

Es ist aber durchaus möglich, dass eine Datei mehrere Klassen enthält. Allerdings kann nur eine einzige dieser Klassen `public` sein.

Hier das Programm:

```
// Datei: Punkt2.java
public class Punkt2
{
   private int x;

   public Punkt2()                // Dieser Konstruktor wird
   {                              // noch erklärt
      System.out.print ("selbst geschriebener ");
      System.out.println ("parameterloser Konstruktor:");
      x = 1;
      System.out.println ("x = " + x);
   }

   public Punkt2 (int u)          // Dieser Konstruktor wird noch
   {                              // erklärt
      System.out.print ("Konstruktor mit einem Parameter:");
      System.out.println (" x = " + u);
      x = u;
   }

   public void print()
   {
      System.out.println ("x = " + x);
   }
}
```

```
// Datei: TestPunkt2.java
public class TestPunkt2
{
   public static void main (String[] args)
   {
      Punkt2 p1 = new Punkt2();   // Erzeugen eines Punktes.
                                  // x wird durch den selbst
                                  // geschriebenen parameterlosen
                                  // Konstruktor auf 1 gesetzt
      Punkt2 p2 = new Punkt2 (3); // Erzeugen eines Punktes.
                                  // x wird auf 3 gesetzt
      Punkt2 p3 = new Punkt2 (10);// Erzeugen eines Punktes.
                                  // x wird auf 10 gesetzt

      System.out.println ("Koordinate des Punktes, auf den p1
                          zeigt:");
      p1.print();

      System.out.println ("Koordinate des Punktes, auf den p2
                          zeigt:");
      p2.print();

      System.out.println ("Koordinate des Punktes, auf den p3
                          zeigt:");
      p3.print();
   }
}
```

Die Ausgabe des Programms ist:

```
selbst geschriebener parameterloser Konstruktor: x = 1
Konstruktor mit einem Parameter: x = 3
Konstruktor mit einem Parameter: x = 10
Koordinate des Punktes, auf den p1 zeigt:
x = 1
Koordinate des Punktes, auf den p2 zeigt:
x = 3
Koordinate des Punktes, auf den p3 zeigt:
x = 10
```

Beachten Sie die Methode `Punkt2()`:

```
public Punkt2()
{
   System.out.print ("selbst geschriebener ");
   System.out.println ("parameterloser Konstruktor:");
   . . .
}
```

Diese Methode ist etwas Besonderes. Sie trägt den gleichen Namen wie die Klasse und hat keinen Rückgabewert. Eine solche Methode heißt **Konstruktor**.

Eine Methode, die gleich heißt wie die Klasse, wird **Konstruktor** genannt. Ein Konstruktor dient meist zur Initialisierung eines Objektes.

Da kein Übergabeparameter angegeben ist, handelt es sich um einen sogenannten **parameterlosen Konstruktor**. Jeder Punkt, der mit dem angegebenen parameterlosen Konstruktor initialisiert wird, hat in der Regel dieselben Koordinaten, es sei denn im Konstruktor werden die Initialisierungswerte interaktiv eingelesen.

Mit Hilfe eines **parameterlosen Konstruktors** kann man jeden Punkt nur dann individuell initialisieren, wenn im parameterlosen Konstruktor interaktiv die Werte eingegeben werden.

Der parameterlose Konstruktor wird aufgerufen, wenn ein Objekt mit Hilfe des `new`-Operators erzeugt wird und die Parameterliste leer ist, d. h. zwischen den runden Klammern des Konstruktors nichts steht wie in folgendem Beispiel:

```
Punkt2 p1 = new Punkt2();
```

Die Klasse `Punkt2` enthält noch einen zweiten Konstruktor, einen **Konstruktor mit Parametern**:

```
public Punkt2 (int u)
{
   System.out.print ("Konstruktor mit einem Parameter:");
   System.out.println (" x = " + u);
   x = u;
}
```

Dieser Konstruktor wird aufgerufen, wenn ein Objekt mit Hilfe des `new`-Operators erzeugt wird und in der Parameterliste ein Parameter des Typs `int` steht wie in folgendem Beispiel:

```
Punkt2 p2 = new Punkt2 (3);
```

Mit Hilfe eines **Konstruktors mit Parametern** ist es möglich, einen jeden Punkt individuell zu initialisieren.

Der Compiler hat keine Schwierigkeiten damit, dass es zwei Methoden mit demselben Namen gibt. An den Typen der Parameter sieht der Compiler, welche der beiden Methoden er zu nehmen hat. Man sagt, der Konstruktor ist überladen[60].

Wenn man **selbst überhaupt keinen Konstruktor** schreibt, dann stellt der Compiler einen vordefinierten **Default-Konstruktor** als parameterlosen Konstruktor bereit, der es ermöglicht, dass beispielsweise ein Aufruf

```
Punkt2 p1 = new Punkt2();
```

kompilierbar ist.

[60] Überladen von Methoden wird in Kapitel 9.4 behandelt.

Bild 4-2 symbolisiert die Programmstruktur. Neben der Wrapper-Klasse `TestPunkt2` für die Methode `main()` existieren 3 Objekte der Klasse `Punkt`:

Bild 4-2 Wrapper-Klasse `TestPunkt2` erzeugt 3 Punkte

4.4 Schreiben von Instanzmethoden

Mit Methoden kann man Objekte bearbeiten. Im Folgenden wird wieder eine Klasse für einen Punkt behandelt. Die Punkt-Objekte sollen in einer zweidimensionalen Ebene liegen und sollen durch ihre kartesischen Koordinaten `x` und `y` charakterisiert werden. Dabei soll eine Methode geschrieben werden, um ein Punkt-Objekt zu verschieben. Der Punkt `P1` mit den Koordinaten `x` und `y` soll um den Vektor (`deltaX`, `deltaY`), wie in Bild 4-3 gezeigt, verschoben werden:

Bild 4-3 Verschieben eines Punktes (Translation)

Konkret soll der Punkt `P1(1,2)` um den Vektor `(4,1)` nach `P2(5,3)` verschoben werden. Die bisherige Klasse `Punkt3` lautet:

```
// Datei: Punkt3.java

public class Punkt3   // Deklaration der Klasse Punkt3
{
   private int x;     // Datenfeld für die x-Koordinate vom Typ int
   private int y;     // Datenfeld für die y-Koordinate vom Typ int
```

```
   public int getX()                // eine Methode, um den x-Wert
   {                                // abzuholen
      return x;
   }

   public int getY()                // eine Methode, um den y-Wert
   {                                // abzuholen
      return y;
   }

   public void setX (int i)         // eine Methode, um den x-Wert
   {                                // zu setzen
      x = i;
   }

   public void setY (int i)         // eine Methode, um den y-Wert
   {                                // zu setzen
      y = i;
   }
}

// Datei: TestPunkt3.java

public class TestPunkt3
{
   public static void main (String[] args)
   {
      Punkt3 p = new Punkt3();      // hiermit wird ein Punkt
                                    // erzeugt
      p.setX (1);                   // Aufruf der Methode setX()
      p.setY (2);                   // Aufruf der Methode setY()

      System.out.println ("Die Koordinaten des Punktes sind: ");
      System.out.println (p.getX());
      System.out.println (p.getY());
   }
}
```

Die Ausgabe des Programms ist:

```
Die Koordinaten des Punktes sind:
1
2
```

Im Folgenden soll die **Realisierung der Methode `verschiebe()`** betrachtet werden. Methoden müssen in der Lage sein, für verschiedene Parameter zu funktionieren. Im vorliegenden Beispiel muss eine Verschiebung eines beliebigen Punktes um einen beliebigen Vektor realisiert werden können. Ein beliebiger Punkt hat das Datenfeld `x`, welches einen konkreten Zahlenwert für die Abszisse trägt, und das Datenfeld `y` für die Ordinate. Damit lautet der Algorithmus:

Nimm den aktuellen Wert von `x` und addiere den Wert von `deltaX`.
Nimm den aktuellen Wert von `y` und addiere den Wert von `deltaY`.

Dieser Algorithmus wird programmtechnisch auf jeweils eine Zuweisung abgebildet. Der Algorithmus lautet:

```
x = x + deltaX;
y = y + deltaY;
```

Damit ist der Rumpf der Methode fast schon fertig. Ein **Rumpf einer Methode** hat aber immer eine öffnende und eine schließende geschweifte Klammer:

```
{
   x = x + deltaX;
   y = y + deltaY;
}
```

Es ist üblich, die Anweisungen gegenüber den geschweiften Klammern einzurücken, damit man die Blockgrenzen[61] besser erkennt.

Aufgerufen wird dieser Algorithmus über seinen Namen. Das ist der **Methodenname**. Er lautet hier `verschiebe`. Er steht im **Methodenkopf**, welcher **vor dem Methodenrumpf**, d. h. dem Block, angeordnet ist. Einen Rückgabewert hat diese Methode nicht. Dies wird durch das Schlüsselwort `void`[62] vor dem Methodennamen beschrieben. Die Methode soll von außerhalb der Klasse aufrufbar sein. Also erhält sie den Zugriffsmodifikator `public`. Damit ist die Definition der Methode schon fast fertig. Der momentane Zwischenstand ist:

```
public void verschiebe
{
   x = x + deltaX;
   y = y + deltaY;
}
```

Was noch fehlt, ist der Übergabemechanismus. Schließlich sollen `deltaX` und `deltaY` als **Parameter an die Methode übergeben** werden. Zu diesem Zweck dient die Liste der Übergabeparameter, welche im Methodenkopf hinter dem Methodennamen steht. In runden Klammern wird hierbei der Typ und Name eines jeden Übergabeparameters aufgeführt. Damit lautet die vollständige Definition der Methode `verschiebe()`:

```
public void verschiebe (int deltaX, int deltaY)
{
   x = x + deltaX;
   y = y + deltaY;
}
```

Nun zum Aufruf der Methode `verschiebe()`. Eine Instanzmethode wird für eine Referenz auf einen konkreten Punkt aufgerufen. In der Methode `main()` der Klasse `TestPunkt3` gibt es die Referenz `p`, die auf einen konkreten Punkt zeigt:

---

[61] Ein Block enthält Anweisungen, die zwischen geschweiften Klammern als Blockbegrenzer stehen.
[62] `void` bedeutet "leer".

```
public static void main (String[] args)
{
   Punkt3 p = new Punkt3(); // hiermit wird ein Punkt erzeugt
   p.setX (1);              // Aufruf Methode setX() für die
                            // Referenz p auf den Punkt
   p.setY (2);              // Aufruf Methode setY() für die
                            // Referenz p auf den Punkt

   System.out.println ("Die Koordinaten des Punktes sind: ");
   System.out.println (p.getX());
   System.out.println (p.getY());
}
```

Die Koordinaten des Punktes, auf den `p` verweist, werden mit Hilfe der Methoden `setX()` und `setY()` auf `x` gleich `1` und `y` gleich `2` gesetzt. Eine Verschiebung um den Vektor `(4, 1)` erhält man durch den Aufruf:

```
p.verschiebe (4, 1);
```

Hier das komplette Programm:

```
// Datei: Punkt4.java
public class Punkt4   // Deklaration der Klasse Punkt4
{
   private int x;     // Datenfeld für die x-Koordinate vom Typ int
   private int y;     // Datenfeld für die y-Koordinate vom Typ int
   public int getX()                // eine Methode, um den x-Wert
   {                                // abzuholen
      return x;
   }

   public int getY()                // eine Methode, um den y-Wert
   {                                // abzuholen
      return y;
   }

   public void setX (int i)         // eine Methode, um den x-Wert
   {                                // zu setzen
      x = i;
   }

   public void setY (int i)         // eine Methode, um den y-Wert
   {                                // zu setzen
      y = i;
   }

   public void verschiebe (int deltaX, int deltaY)
   {
      x = x + deltaX;
      y = y + deltaY;
   }
}
```

Die folgende Klasse `TestPunkt` soll testen, ob das Verschieben erfolgreich war:

```
// Datei: TestPunkt4.java

public class TestPunkt4
{
   public static void main (String[] args)
   {
      Punkt4 p = new Punkt4();   // hiermit wird ein Punkt erzeugt

      p.setX (1);                // Aufruf der Methode setX()
      p.setY (2);                // Aufruf der Methode setY()

      System.out.println ("Die Koordinaten des Punktes sind " +
         "vor der Verschiebung: ");
      System.out.println (p.getX());
      System.out.println (p.getY());

      p.verschiebe (4, 1);
      System.out.println ("Die Koordinaten des Punktes sind " +
         "nach der Verschiebung: ");
      System.out.println (p.getX());
      System.out.println (p.getY());
   }
}
```

Die Ausgabe des Programms ist:

```
Die Koordinaten des Punktes sind vor der Verschiebung:
1
2
Die Koordinaten des Punktes sind nach der Verschiebung:
5
3
```

Was passiert nun beim Aufruf von `verschiebe (4, 1)`? Es werden die beiden lokalen Variablen `deltaX` und `deltaY` angelegt. `deltaX` und `deltaY` werden auch als **formale Parameter** bezeichnet. Diese werden mit den Werten der **aktuellen Parameter**, d. h. mit dem aktuellen Parameter `4` und dem aktuellen Parameter `1` initialisiert. Die Variable `deltaX` erhält also den Wert `4` und `deltaY` den Wert `1`.

Ein formaler Parameter einer Methode wird durch den übergebenen, aktuellen Parameter initialisiert.

Was programmtechnisch beim Aufruf abläuft, kann man sich vereinfacht folgendermaßen veranschaulichen:

```
int deltaX = 4;          // Anlegen der Variablen deltaX und
                         // Zuweisung der 4
int deltaY = 1;          // Anlegen der Variablen deltaY und
                         // Zuweisung der 1
```

Durch diesen Übergabemechanismus erhalten die lokalen Variablen `deltaX` und `deltaY` definierte Werte für die Abarbeitung des Methodenrumpfes.

4.5 Zusammengesetzte Objekte

Das folgende Bild zeigt ein Kreiseck:

Bild 4-4 Kreiseck mit Mittelpunkt im Ursprung des kartesischen Koordinatensystems

Unter einem Kreiseck wird hier ein Quadrat – ein rechtwinkliges Viereck mit vier gleich langen Seiten – verstanden, welches von einem Kreis so ausgefüllt ist, dass die Seiten des Quadrats Tangenten an den Kreis sind. Mit anderen Worten, der Kreis soll einen Inkreis darstellen. Der Mittelpunkt des Kreisecks soll im Ursprung eines kartesischen Koordinatensystems liegen.

In Java kann ein Kreiseck erzeugt werden mit Hilfe einer Klasse, die einen Kreis und ein Eck (Quadrat) **aggregiert**. Das Eck selbst stellt eine Aggregation von 4 Punkten dar. Sind die Klassen `Eck` und `Kreis` schon bekannt, so kann ein Objekt der Klasse `Kreiseck` aus Objekten der schon bekannten Klassen `Kreis` und `Eck` zusammengebaut werden. Siehe hierzu das folgende Bild:

Bild 4-5 "Groß"-Objekt `:Kreiseck` referenziert "Klein"-Objekte `:Kreis` und `:Eck`

Methodenaufrufe einer Anwendung gehen an das zusammengesetzte Objekt der Klasse `Kreiseck`, z. B. "`skaliere (2)`", was eine Vergrößerung um den Faktor `2` bedeuten soll. Die entsprechende Methode dieses Objektes leitet diese Botschaft dann weiter an das aggregierte Objekt der Klasse `Kreis` und das aggregierte Objekt der Klasse `Eck` durch Aufruf deren Skalierungsmethoden.

Das "Groß"-Objekt delegiert den Aufruf, der an das "Groß"-Objekt selbst gerichtet war, weiter an die "Klein"-Objekte. Dieses Prinzip wird als **Delegationsprinzip** bezeichnet.

Das folgende Java-Programm enthält die Klassen `Punkt5`, `Eck`, `Kreis`, `KreisEck` und `KreisEckTest`:

```
// Datei: Punkt5.java
public class Punkt5
{
   private double x;
   private double y;

   public Punkt5 (double x, double y)
   {
      this.x = x;
      this.y = y;
   }

   public double getX()
   {
      return x;
   }

   public void setX (double u)
   {
      x = u;
   }

   public double getY()
   {
      return y;
   }

   public void setY (double v)
   {
       y = v;
   }
}

// Datei: Eck.java
public class Eck
{
   private Punkt5 p1;
   private Punkt5 p2;
   private Punkt5 p3;
   private Punkt5 p4;

   public Eck (double u)   // u soll eine halbe Seitenlänge
   {                       // des Quadrats darstellen
      System.out.println ("Viereck wird erzeugt aus 4 Eckpunkten");
      p1 = new Punkt5 (u, u);
      p2 = new Punkt5 (-u, u);
      p3 = new Punkt5 (u, -u);
      p4 = new Punkt5 (-u, -u);
   }
```

```
   public void skaliere (double u)
   {
      p1.setX (p1.getX() * u);
      p1.setY (p1.getY() * u);
      p2.setX (p2.getX() * u);
      p2.setY (p2.getY() * u);
      p3.setX (p3.getX() * u);
      p3.setY (p3.getY() * u);
      p4.setX (p4.getX() * u);
      p4.setY (p4.getY() * u);
   }

   public double berechneFlaeche()
   {
      return (2 * p1.getX()) * (2 * p1.getY());
   }
}

// Datei: Kreis.java
public class Kreis
{
   private double radius;
   static final double PI = 3.1415; // PI ist eine konstante
                                    // Klassenvariable

   public Kreis (double u)
   {
      radius = u;
   }

   public void skaliere (double u)
   {
      radius = radius * u;
   }

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

   public double getRadius()
   {
      return radius;
   }
}

// Datei: Kreiseck.java
public class Kreiseck
{
   private Kreis kreisref;
   private Eck eckref;

   public Kreiseck (double alpha) // alpha ist der Radius des
                                  // Inkreises
   {
      kreisref = new Kreis (alpha);
      eckref = new Eck (alpha);
   }
```

```
   public void skaliere (double u)
   {
      kreisref.skaliere (u); // Delegationsprinzip. Der Aufruf
      eckref.skaliere (u);   // skaliere (u) wird an die "Klein"-
   }                         // Objekte weitergeleitet.

   public double flaechendifferenz()
   {
      return (eckref.berechneFlaeche() -
              kreisref.berechneFlaeche());
   }

   public Kreis getKreis()
   {
      return kreisref;
   }
}

// Datei: KreiseckTest.java
public class KreiseckTest
{
   public static void main (String[] args)
   {
      Kreiseck kreiseckref = new Kreiseck (1);
      System.out.println ("Radius = " +
                          kreiseckref.getKreis().getRadius());
      System.out.println ("Die Flächendifferenz ist " +
                          kreiseckref.flaechendifferenz());
      kreiseckref.skaliere (2);
      System.out.println ("Es wurde um den Faktor 2 skaliert");
      System.out.println ("Radius = " +
                          kreiseckref.getKreis().getRadius());
      System.out.println ("Die Flächendifferenz ist " +
                          kreiseckref.flaechendifferenz());
   }
}
```

Die Ausgabe des Programms ist:

```
Viereck wird erzeugt aus 4 Eckpunkten
Radius = 1.0
Die Flächendifferenz ist 0.8584999999999998
Es wurde um den Faktor 2 skaliert
Radius = 2.0
Die Flächendifferenz ist 3.4339999999999993
```

Beachten Sie, dass im Konstruktor des Kreisecks die Referenzen `kreisref` und `eckref` des Kreiseck-Objektes initialisiert werden müssen. Die Initialisierung erfolgt durch Erzeugen eines Objektes der Klasse `Kreis` bzw. der Klasse `Eck` mit Hilfe des `new`-Operators. Dabei wird der Referenz `kreisref` die Referenz auf das vom `new`-Operator plus Konstruktor erzeugte Objekt der Klasse `Kreis` zugewiesen und der Referenz `eckRef` die Referenz auf das vom `new`-Operator plus Konstruktor erzeugte Objekt der Klasse `Eck`. Genauso müssen im Konstruktor von `Eck` die Referenzen auf die vier Eckpunkte durch Erzeugen der Eckpunkte initialisiert werden.

4.6 Selbst definierte Untertypen durch Vererbung

Das folgende Programm soll eine erste Einführung in die Vererbung sein.

Eine Klasse erbt von einer sogenannten **Basisklasse**, von der sie abgeleitet wird, den gesamten Code, der aus Datenfeldern und Methoden besteht, und kann diesen Code durch zusätzliche Datenfelder und Methoden ergänzen. Man spricht dann davon, dass die **abgeleitete Klasse** die **Struktur** und das **Verhalten der Basisklasse erbt** und die Basisklasse erweitert.

Dies dient dazu, um von einer bereits vorhandenen allgemeineren Klasse eine spezialisierte Klasse abzuleiten. Mit anderen Worten, Vererbung dient dem **Subtyping**, d. h. dem Bilden eines Untertyps für einen vorhandenen Typ. Als Beispiel für einen Typ soll die Klasse `Person` dienen. Ein Untertyp der Klasse `Person` ist die Klasse `Student`. Ein Objekt eines Untertyps (einer abgeleiteten Klasse) muss auch als Objekt des allgemeineren Typs (der Basisklasse) auftreten können. So tritt ein Student in gewissen Fällen als Student und in anderen Fällen als eine normale Person auf. Dies kommt auch darin zum Ausdruck, dass die Beziehung zwischen den beiden Klassen als **"is a"-Beziehung** bezeichnet wird. Eine "is a"-Beziehung ist in Bild 4-6 zu sehen. Sie wird dargestellt durch einen Pfeil von der abgeleiteten Klasse zu der Basisklasse, wobei die Pfeilspitze ein nicht ausgefülltes Dreieck ist.

Abgeleitet werden Klassen. Objekte können nicht abgeleitet werden.

Durch Ableitung wird ein Untertyp geschaffen. Dies nennt man auch **Subtyping**.

Im Folgenden wird die bereits erwähnte Ableitung visualisiert:

Bild 4-6 Ableitung der Klasse `Student` von der Klasse `Person`

Der Mechanismus der Vererbung hat auch den Vorteil, dass dem Programmierer beim Erstellen eines Untertyps ein fehlerträchtiges "Copy and Paste" der Datenfelder und Methoden des allgemeineren Typs in den Subtyp erspart bleibt.

Nun zum Beispiel des Studenten. Ein Student ist bekanntermaßen eine Person, die studiert. Wenn man studieren möchte, muss man immatrikuliert werden und eine Matrikelnummer erhalten. Kurz, wer eine Matrikelnummer hat, ist eingeschrieben und ist somit ein Student. Also kann man einen Studenten beschreiben als eine Person, die eine Matrikelnummer hat. Das folgende Codestück zeigt die Klassen `Person` und `Student`:

```
// Datei: Person.java

import java.util.Scanner;

public class Person
{
   private String name;       // Namen sind konstante Zeichenketten
   private String vorname;    // und können in einer Variable vom
                              // Typ String gespeichert werden. Die
                              // Klasse String ist eine Bibliotheks-
                              // klasse
   public Person()
   {
      System.out.print ("\nNachnamen eingeben ");
      System.out.print ("(Ende mit <CR>): ");
      name = input();

      System.out.print ("Vornamen eingeben ");
      System.out.print ("(Ende mit <CR>): ");
      vorname = input();
   }

   public String input()      // bitte überlesen Sie diese Methode
   {
      Scanner eingabe = new Scanner (System.in);
      return eingabe.next();
   }

   public void print()
   {
      System.out.print ("\nNachname: " + name);
      System.out.print ("\nVorname: " + vorname);
   }
}

// Datei: Student.java

public class Student extends Person
{
   private String matrikelnummer;

   public Student()
   {
      super();       // Aufruf des Konstruktors der Vaterklasse
```

```
      System.out.print ("Matrikelnummer eingeben ");
      System.out.print ("(Ende mit <CR>): ");
      matrikelnummer = input();
   }

   public void printMatrikelnummer()
   {
      System.out.print ("\nMatrikelnummer : " + matrikelnummer);
   }

   public static void main (String[] args)
   {
      System.out.print ("\nErfasse Person");
      Person pers1 = new Person();

      System.out.print ("\nErfasse Student");
      Student stud1 = new Student();

      System.out.print ("\nAusgabe Person");
      pers1.print();

      System.out.print ("\n\nAusgabe Student");
      stud1.print();
      stud1.printMatrikelnummer();
   }
}
```

Die Ausgabe des Programms ist:

```
Erfasse Person
Nachnamen eingeben (Ende mit <CR>): Schmidt
Vornamen eingeben (Ende mit <CR>): Georg

Erfasse Student
Nachnamen eingeben (Ende mit <CR>): Meiser
Vornamen eingeben (Ende mit <CR>): Myriam
Matrikelnummer eingeben (Ende mit <CR>): 512346

Ausgabe Person
Nachname: Schmidt
Vorname: Georg

Ausgabe Student
Nachname: Meiser
Vorname: Myriam
Matrikelnummer : 512346
```

Beachten Sie, dass der Compiler automatisch den parameterlosen Konstruktor der Vaterklasse als erste Anweisung im Konstruktor der Sohnklasse aufruft, wenn es der Programmierer nicht durch den Aufruf `super()` selbst tut.

Der Aufruf des parameterlosen Konstruktors der Vaterklasse dient dazu, im Sohnobjekt die Datenfelder, die vom Vater geerbt sind, zu initialisieren.

In Kapitel 11.3.2 wird der Aufruf von Konstruktoren einer Basisklasse ausführlich behandelt.

4.7 Ein- und Ausgabe am Bildschirm

Kapitel 4.7.1 erläutert das Einlesen von Zeichen, die über die Tastatur eingegeben werden. Kapitel 4.7.2 behandelt die Methode `printf()` und die Klasse `Scanner`, die Methoden bietet, um Text oder die Werte einfacher Datentypen von der Tastatur einzulesen.

4.7.1 Zeichen von der Tastatur einlesen

Im Folgenden wird ein Zeichenzählprogramm gezeigt, welches die Zeichen zählt, die von der Tastatur eingegeben werden. Der Programmablauf wird solange beim Aufruf der Funktion

```
c = System.in.read();
```

blockiert, bis der Bediener seine Eingabe von beliebig vielen Zeichen mit der `<RETURN>`-Taste abgeschlossen hat. Dann liest das Programm aus dem **Tastaturpuffer**[63] die eingegebenen Zeichen solange aus, bis das Zeichen `<RETURN>` gelesen wird. Danach kann der Bediener, wenn er will, weitere Zeichen eingeben. Die Eingabe von Zeichen kann mit der Tastenkombination `<Strg>`[64] `+ Z`, also das gleichzeitige Drücken der `<Strg>`-Taste und der Taste mit dem Zeichen `Z`, abgebrochen werden. Diese Tastenkombination erzeugt das Steuerzeichen `^Z`. Unter Linux muss man die Tastenkombination `<STRG> + D` anstelle von `<STRG> + Z` betätigen.

Hier das Programm:

```
// Datei: Zeichen.java

public class Zeichen
{
   // beachten Sie die Deklaration der Methode main() nicht
   public static void main (String[] args) throws Exception
   {
      int c = 0;
      int anzahl = 0;

      System.out.print ("Bitte eine Folge von Zeichen eingeben ");
      System.out.println ("und mit <RETURN> abschliessen");
```

[63] Von der Tastatur eingegebene Zeichen kommen zuallererst in den sogenannten Tastaturpuffer.
[64] Die Abkürzung Strg steht für Steuerung.

```
      do
      {
         // System.in.read() gibt einen int-Wert im Bereich 0 bis
         // 255 zurück. -1 wird zurückgegeben, wenn kein Zeichen
         // mehr im Dateipuffer steht.
         c = System.in.read();

         // Mit (char) c wird die int-Variable c
         // in ein Zeichen gewandelt.
         System.out.println (
            "ASCII-Code: " + c + " Zeichen: " + (char) c);
         anzahl = anzahl + 1;

      } while (c != -1);
      System.out.println ("Anzahl: " + anzahl);
   }
}
```

Es wurde eingegeben:

```
FHTE<RETURN>
<STRG> + Z
```

Im Protokoll ist zu sehen, dass auf einem Windows-Rechner bei der Eingabe eines `<RETURN>` das Zeichen `'\r'` mit dem ASCII-Wert `13` und das Zeichen `'\n'` mit dem ASCII-Wert `10` erzeugt wird[65]. Die eingegebenen Zeichen – hier `'F' 'H' 'T' 'E'` – werden im **Tastaturpuffer** zwischengespeichert und werden erst mit der Eingabe von `<RETURN>` an den sogenannten Dateipuffer des Programms übergeben. Die Tastenkombination `<STRG> + Z` erzeugt das Steuerzeichen `^Z`. Dieses Steuerzeichen hat den ASCII-Code `-1`.

Die Ein- und Ausgabe des Programms ist:

```
FHTE
ASCII-Code: 70 Zeichen: F
ASCII-Code: 72 Zeichen: H
ASCII-Code: 84 Zeichen: T
ASCII-Code: 69 Zeichen: E
ASCII-Code: 13 Zeichen:
ASCII-Code: 10 Zeichen:

^Z
ASCII-Code: -1 Zeichen: ?
Anzahl: 7
```

Beachten Sie,

- dass `throws Exception` hier noch nicht erklärt werden kann und an dieser Stelle einfach unbesehen verwendet werden soll,

65 Bei einem UNIX-Rechner wird durch Eingabe von `<RETURN>` nur das Zeichen `'\n'` erzeugt, nicht jedoch zusätzlich das Zeichen `'\r'`. Im ASCII-Zeichensatz (siehe Tabelle 5-5) trägt das Steuerzeichen `'\r'` den Namen CR (Carriage Return) und das Steuerzeichen `'\n'` den Namen NL (New Line) oder LF (Line Feed).

- dass die Notation `!=` "ungleich" bedeutet, `==` bedeutet "gleich",
- dass die Methode `System.in.read()` den Wert `-1` zurückgibt, wenn das Dateiende – erzeugt durch `<CTRL> + Z` – erreicht ist.

Die Methode `System.in.read()` legt bei jedem Aufruf als Rückgabewert das nächste Zeichen des Eingabestroms in der Variablen `c` ab. Durch diese Zuweisung wird der vorhandene Wert von `c` durch den Wert des nächsten Zeichens überschrieben. Nach dem Aufruf der Methode `System.in.read()` steht der Lesezeiger im Dateipuffer ein Zeichen weiter als vor dem Aufruf. Der **Eingabestrom** der Zeichen wird als **Eingabedatei** (**Standard-Input**) gesehen. Deshalb spricht man statt vom **Lesezeiger** auch vom **Dateizeiger**.

Es soll jetzt angenommen werden, über die Tastatur wären die Zeichen `'F' 'H' 'T' 'E'` mit abschließendem `<RETURN>` eingegeben worden. Mit der Eingabe `<RETURN>` wird der **Inhalt des Tastaturpuffers** `'F' 'H' 'T' 'E' <RETURN>` **als Dateipuffer** an das Programm übergeben. Der Dateizeiger steht dabei im Dateipuffer vor dem `'F'` von `'F' 'H' 'T' 'E' '\r' '\n'` (siehe Bild 4-7). Nach dem Aufruf der Methode `System.in.read()` steht der Dateizeiger ein Zeichen weiter, d. h. vor dem `'H'`. Dies wird im folgenden Bild visualisiert:

Bild 4-7 Veranschaulichung des Wanderns des Dateizeigers beim Lesevorgang mit ***`System.in.read()`***

Liest man mit `System.in.read()` von der Tastatur ein, so werden die eingegebenen Zeichen zunächst **im Tastaturpuffer zwischengespeichert** (gepuffert), solange kein `<RETURN>` eingegeben wird. Der Inhalt des Tastaturpuffers wird erst dann in den Dateipuffer übergeben, wenn der Benutzer ein `<RETURN>` eingibt.

Bild 4-8 zeigt ein Beispiel für die **zeilenweise**[66] **Pufferung** im Tastaturpuffer und die zeilenweise Übergabe des Inhalts des Tastaturpuffers in den Dateipuffer:

[66] Eine Zeile ist definiert durch eine Folge von Zeichen bis zum ersten `<RETURN>`, dem Zeilenende.

Bild 4-8 Zeilenweise Pufferung im Tastaturpuffer

Bild 4-9 visualisiert die Programmstruktur:

Bild 4-9 Visualisierung der Programmstruktur

Auch hier liegt nur eine Wrapper-Klasse für `main()` vor.

4.7.2 Die Methode `printf()` und die Klasse `Scanner`

Das folgende Programm `Fahrenheit4` verwendet anstelle der Methode `print()` die Methode `printf()`.

Die Methode `printf()` erwartet als ersten Parameter einen **Formatstring**, der von doppelten Hochkommata begrenzt ist. Innerhalb des Formatstrings stehen **Formatelemente**, welche **die Formatierung der Ausgabe** regeln. Formatelemente erkennt man an dem Zeichen `%`. Nach dem Formatstring kommen als weitere Parameter die auszugebenden Variablen bzw. Ausdrücke, jeweils getrennt durch ein Komma. Für jeden auszugebenden Wert muss im Formatstring ein Formatelement vorhanden sein.

Vor dem Programm sollen zuerst zwei Beispiele für den Aufruf von `printf()` gezeigt werden:

```
System.out.printf ("%3d", 10);
System.out.printf ("%6.2f   %6.2f", 1.0f, 2.2f);
```

Im ersten Beispiel wird die Zahl `10` rechtsbündig in ein Feld, das 3 Zeichen breit ist, ausgegeben. Das `d` steht für dezimal. Da die Zahl `10` nicht die volle Feldbreite ausfüllt, wird links mit einem Leerzeichen aufgefüllt. Im zweiten Beispiel werden die `float`-Zahlen `1.0` und `2.2` mit jeweils 2 Stellen hinter dem Punkt ausgegeben. Für den Punkt selbst wird eines der genannten 6 Zeichen verbraucht. Damit verbleiben vor dem Punkt noch 3 Zeichen, die von links mit Leerzeichen aufgefüllt werden. Wie aus dem Formatstring ersichtlich ist, stehen zwischen den beiden Feldern für die beiden Zahlen genau 3 Leerzeichen. Und nun das Programm `Fahrenheit4`:

```
// Datei: Fahrenheit4.java

public class Fahrenheit4
{
   // Klassenmethode main() zur Ausgabe der Temperaturtabelle
   public static void main (String[] args)
   {
      // Konstanten
      final int UPPER = 300;    // obere Grenze
                                // UPPER ist eine symbol. Konstante
                                // 300 ist eine literale Konstante
      final int LOWER =   0;    // untere Grenze
      final int STEP =   20;    // Schrittweite

      // Variablen
      int fahr;                 // Definition der lokalen Variablen
                                // fahr für die Temperatur in
                                // Fahrenheit
      int celsius;              // Definition der lokalen Variablen
                                // celsius für die Temperatur in
                                // Celsius
      // Anweisungen
      fahr = LOWER;             // als Anfangswert wird fahr
                                // der Wert 0 zugewiesen
      while (fahr <= UPPER)
      {
         // nach dieser Formel berechnet sich der Celsius-Wert aus
         // einem Fahrenheit-Wert
         celsius = 5 * (fahr - 32) / 9;

         // die Werte von fahr und celsius werden jeweils rechtsbün-
         // dig in einem 3 Zeichen breiten Feld ausgegeben
         System.out.printf ("\n%3d   %3d", fahr, celsius);
         fahr = fahr + STEP;    // nächsten Wert von fahr berechnen
      }
   }
}
```

Hier ein Auszug der Programmausgabe:

```
  0   -17
 20    -6
 40     4
  . . .
280   137
300   148
```

Neben der einfachen Ausgabemöglichkeit mit `printf()` gibt es seit Java 5 auch eine einfache Möglichkeit für die Eingabe.

Die Klasse **`Scanner`** kann man aus einer Instanz vom Typ `InputStream`, aber auch aus einer Datei oder einer Variablen vom Typ `String` lesen.

Dazu bietet die Klasse `Scanner` verschiedene Methoden, mit denen Text und Werte primitiver Datentypen eingelesen werden können.

Die Klasse `InputStream` wird in Kapitel 19.3.2.1 erklärt.

Darüber hinaus bietet die Klasse `Scanner` die Möglichkeit, die Eingabedaten mit Hilfe von regulären Ausdrücken[67] nach bestimmten Zeichenmustern zu durchsuchen. Auf reguläre Ausdrücke kann an dieser Stelle nicht näher eingegangen werden. Es wird hier auf die API-Dokumentation verwiesen.

Das folgende Beispiel zeigt das Einlesen einer Zeichenkette und von ganzen Zahlen mit den Methoden `next()` und `nextInt()` der Klasse `java.util.Scanner`:

```
// Datei: EingabeTest.java
import java.util.Scanner;

public class EingabeTest
{
   public static void main (String[] args)
   {
      // Erzeugen eines Objektes der Klasse Scanner, um von
      // der Standard-Eingabe (Tastatur) einzulesen.
      Scanner eingabe = new Scanner (System.in);
      System.out.print ("Geben Sie Ihren Namen ein: ");
      String name = eingabe.next();
      System.out.println ("Hallo " + name +
                  "! Heute wollen wir zwei Zahlen addieren.");

      System.out.print (name + ", gib die erste Zahl ein: ");
      int zahl1 = eingabe.nextInt();
      System.out.print ("OK. Und nun die zweite Zahl: ");
      int zahl2 = eingabe.nextInt();
      System.out.println ("Das Ergebnis ist: " + zahl1 +
                    " + " + zahl2 + " = " + (zahl1 + zahl2));
   }
}
```

Die Ausgabe des Programms ist:

```
Geben Sie Ihren Namen ein: Martin
Hallo Martin! Heute wollen wir zwei Zahlen addieren.
Martin, gib die erste Zahl ein: 5
OK. Und nun die zweite Zahl: 4
Das Ergebnis ist: 5 + 4 = 9
```

[67] Ein **regulärer Ausdruck** ist eine aus einer Menge von Zeichen nach syntaktischen Regeln erstellte Zeichenkette, die dazu dient, beliebige Zeichenketten daraufhin zu überprüfen, ob diese den regulären Ausdruck enthalten.

Wie aus dem Programm ersichtlich ist, werden Zeichenketten mit der Methode `next()` eingelesen, `int`-Werte mit der Methode `nextInt()`. Analog gibt es weitere Methoden wie `nextFloat()`, `nextDouble()` und `nextByte()`.

Die vollständigen Möglichkeiten der Klasse `java.util.Scanner` können der API-Dokumentation entnommen werden.

4.8 Zusammenfassung

Kapitel **4.1** dient unter anderem zur Erprobung von lokalen Variablen, Ausdrücken und Schleifen in einem Programmbeispiel und zur Erprobung der Ausgabe.

Die Methoden `println()` und `print()` sind überladen. Überladen bedeutet, dass es zum selben Methodennamen verschiedene Methoden gibt, die sich im Typ ihrer Parameter unterscheiden. Für jeden Typ gibt es eine eigene Methode und automatisch wird die richtige Methode aufgerufen. So können an `println()` bzw. `print()` beispielsweise auch `int`- oder `float`-Variablen übergeben werden wie z. B. die `int`-Variable `fahr` in `System.out.print (fahr)`. Ist eine Variable mit einem String durch den Verkettungsoperator `+` verknüpft, so wird der Wert der Variablen automatisch in einen String gewandelt und an den vorhandenen String angehängt.

Kapitel **4.2** untersucht das Erzeugen von Objekten und stellt den `new`-Operator und den Konstruktor ohne Parameter vor.

Kapitel **4.3** führt dann in einem Programmbeispiel Konstruktoren ohne und mit Parametern ein.

Mit Hilfe eines parameterlosen Konstruktors kann man jeden Punkt nur dann individuell initialisieren, wenn im parameterlosen Konstruktor interaktiv die Werte eingegeben werden. Mit Hilfe des Konstruktors mit Parametern ist es möglich, einen jeden Punkt individuell zu initialisieren.

Kapitel **4.4** erklärt das Schreiben von Instanzmethoden. Ein formaler Parameter einer Methode wird durch einen aktuellen Parameter initialisiert.

Kapitel **4.5** befasst sich anhand eines Programmbeispiels mit "Groß"-Objekten und "Klein"-Objekten.

In Kapitel **4.6** wird das Subtyping durch Vererbung erläutert und in einem Programmbeispiel demonstriert. Abgeleitet werden Klassen. Objekte können nicht abgeleitet werden. Durch Ableitung wird ein Untertyp geschaffen. Dies nennt man auch Subtyping. Eine abgeleitete Klasse erbt die Struktur und das Verhalten ihrer Vaterklasse.

Kapitel **4.7** zeigt Programmbeispiele zur Ein- und Ausgabe am Bildschirm. Als erstes wird das Einlesen von Zeichen von der Tastatur behandelt. Anschließend wird die Methode `printf()` zum Ausgeben und die Verwendung der Klasse `Scanner` für das Einlesen von Strings und Werten primitiver Datentypen besprochen.

4.9 Übungen

Aufgabe 4.1: Schleifen

4.1.1 Summe der Zahlen 1 bis 9

Berechnen Sie die Summe der Zahlen 1 bis 9.

Hinweis: Führen Sie eine Variable `summe` vom Typ `int` ein. Diese Variable soll den Wert der Summe repräsentieren. Setzen Sie den Anfangswert von `summe` auf `0`. Erhöhen Sie dann den Wert von `summe` bei jedem Schleifendurchgang um den Wert der Schleifenvariablen. Geben Sie bei jedem Durchlauf den Wert der Summe aus, um zu verfolgen, ob Ihr Programm richtig rechnet.

4.1.2 Fakultät von n berechnen

Schreiben Sie ein Programm, das die Fakultät von `n` in einer Schleife berechnet. Die Variable `n` soll dabei vom Typ `int` sein. Für `n` dürfen Sie einen beliebigen Wert auswählen. Geben Sie das Ergebnis am Bildschirm aus.

Hinweis: n! = 1 * 2 * 3 * … * (n-1) * n

4.1.3 Fibonacci-Folge

Schreiben Sie ein Programm, das die Fibonacci-Folge auf der Konsole ausgibt. Führen Sie hierzu 3 Variablen des Typs `int` ein. Die Berechnung der Folgezahlen soll in einer Schleife gemacht werden.

Hinweis: Für die ersten beiden Zahlen sollen die Werte `0` und `1` angenommen werden. Jede weitere Zahl ist die Summe ihrer beiden Vorgänger. Eine Fibonacci-Folge sieht wie folgt aus: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

4.1.4 Größter gemeinsamer Teiler

In Kapitel 1.2.1 wurde der Algorithmus von Euklid zur Bestimmung des größten gemeinsamen Teilers vorgestellt. Entwickeln Sie eine Anwendung, welche diesen Algorithmus implementiert. Berechnen Sie den größten gemeinsamen Teiler von:

`24` und `9`

Beachten Sie, dass das in Kapitel 1.3.4 vorgestellte Nassi-Shneiderman-Diagramm bei der Implementierung eine zusätzliche Hilfe darstellt.

Aufgabe 4.2: Zeichen

4.2.1 Zeichen zählen

Erweitern Sie das Programm `Zeichen` aus Kapitel 4.7.1, sodass die Anzahl der eingegebenen Leerzeichen ermittelt wird.

4.2.2 Zeilen zählen

Erweitern Sie das Programm `Zeichen` aus Kapitel 4.7.1, sodass die Anzahl der eingegebenen Zeilen sowie die Anzahl der Zeichen pro Zeile ermittelt wird.

Aufgabe 4.3: Klassen und Objekte

4.3.1 Erste Versuche mit der Klasse `Person`

Schreiben Sie eine Klasse `Person` mit den Datenfeldern:

```
private String name;
private String vorname;
```

und mit den Methoden:

```
public void setName (String name)
public String getName()
public void setVorname (String vname)
public String getVorname()
```

Testen Sie diese Klasse mit der Klasse `TestPerson`:

```
// Datei: TestPerson.java

public class TestPerson
{
   public static void main (String[] args)
   {
      String vorname;
      String name;
      Person schoettle;

      schoettle = new Person();

      schoettle.setName ("Schöttle");
      schoettle.setVorname ("Lothar");

      name = schoettle.getName();
      vorname = schoettle.getVorname();

      System.out.println ("Vorname: " + vorname);
      System.out.println ("Name:    " + name);
   }
}
```

4.3.2 Definition einer Klasse für Schuhe

a) Definieren Sie eine Klasse `Schuh` mit den folgenden Eigenschaften:

- einem Datenfeld `groesse` vom Typ `int`,
- je einem Datenfeld `hersteller` und `modellbezeichnung` vom Typ `String`,
- den folgenden Methoden zum Setzen und Lesen der Datenfelder:

```
public void setGroesse (int groesse)
public void setHersteller (String hersteller)
public void setModellbezeichnung (String modell)
public int getGroesse()
public String getHersteller()
public String getModellbezeichnung()
```

Beachten Sie, dass **kein** direkter Zugriff auf die Datenfelder eines Objektes durch Methoden außerhalb der Klasse erfolgen darf.

b) Schreiben Sie eine Testklasse, die nur eine Methode `main()` enthält. Innerhalb dieser Methode soll ein Objekt der Klasse `Schuh` erzeugt werden, auf das mit der Referenz `s` gezeigt wird. Setzen Sie nun in der Testklasse die Daten des Schuhs mit Hilfe obiger Methoden auf folgende Daten (Größe `42`, Hersteller "`Mike`" und Modellbezeichnung "`Air Ultramatic`"). Erweitern Sie zusätzlich die Klasse `Schuh` um die Methode `print()`, um alle Datenfelder eines Schuhs auf dem Bildschirm auszugeben.

Aufgabe 4.4: Konstruktoren

4.4.1 Konstruktoren für Schuhe

Ergänzen Sie die Klasse `Schuh` um einen selbst geschriebenen **parameterlosen Konstruktor**. Fügen Sie zur Kontrolle des Konstruktoraufrufs eine Protokoll-Ausgabe ein. Überladen Sie den parameterlosen Konstruktor mit einem (oder mehreren) **Konstruktor(en) mit Parametern**. Fügen sie wiederum Protokoll-Ausgaben ein und überzeugen Sie sich, welcher Konstruktor bei der Erzeugung der Schuhe (z. B. durch `new Schuh()`, `new Schuh (44)`, `new Schuh (41, "Panther")`) aufgerufen wird.

4.4.2 Überladene Konstruktoren für die Klasse `Punkt2`

Schreiben Sie für die Klasse `Punkt2`:

- einen Konstruktor ohne Parameter, der die Koordinaten auf `x` gleich 0 und `y` gleich 0 initialisiert,
- einen Konstruktor mit einem Parameter, der den `x`-Wert des Punktes mit dem übergebenen Wert initialisiert und die `y`-Komponente auf 0 setzt,
- einen Konstruktor mit 2 Parametern, der den `x`-Wert und den `y`-Wert des Punktes mit den übergebenen Werten initialisiert.

Hier das zu ergänzende Programm (fehlende Stellen sind dabei durch `. . . . .` markiert):

```
// Datei: Punkt2.java

public class Punkt2 {
   private int x;
   private int y;

   public Punkt2()
   {
      System.out.println ("Konstruktor ohne Parameter");
      . . . . .
   }

   public Punkt2 (int u)
   {
```

```
      System.out.println
         ("Konstruktor mit einem Parameter: x = " + u);
      . . . . .
   }

   public Punkt2 (int u, int v)
   {
      System.out.print ("Konstruktor mit zwei Parametern: ");
      System.out.println ("x = " + u + " y = " + v);
      . . . . .
   }


   public int getX()
   {
      return x;
   }

   public int getY()
   {
      return y;
   }

   public void print()
   {
      System.out.println ("x = " + x);
      System.out.println ("y = " + y);
   }
}
```

Schreiben Sie eine Testklasse `TestPunkt2`, die 3 Punkte in der Methode `main()` erzeugt. Der erste Punkt soll mit dem Konstruktor ohne Parameter, der zweite Punkt mit dem Konstruktor mit einem Parameter und der dritte Punkt mit dem Konstruktor mit zwei Parametern initialisiert werden. Anschließend sind die Koordinaten aller drei Punkte auszugeben.

Aufgabe 4.5: Klasse `Scanner`

4.5.1 Durchschnittsberechnung

Schreiben Sie ein Programm, welches mit Hilfe der Klasse `Scanner` drei `float`-Zahlen von der Tastatur einliest und dann den Durchschnitt dieser Zahlen auf dem Bildschirm ausgibt.

4.5.2 Flächenberechnung – Konstruktor mit Parametern

Schreiben Sie eine Klasse `Rechteck` mit den Instanz-Variablen `a` und `b` und eine weitere Klasse `Quadrat` mit der Instanz-Variablen `a`. Beide Klassen sollen eine Methode zum Berechnen der Fläche namens `getFlaeche()` haben.

In einer `main()`-Methode soll der Benutzer nun die Möglichkeit haben, für ein Objekt der Klasse `Rechteck` und ein Objekt der Klasse `Quadrat` Werte einzugeben. Die Eingabe soll mittels der bereits bekannten Java-Klasse `Scanner` erfolgen. Die eingegebenen Werte sollen dem jeweiligen Objekt beim Konstruktoraufruf übergeben werden.

In der `main()`-Methode sollen getrennt die beiden Flächeninhalte sowie der gesamte Flächeninhalt am Bildschirm ausgegeben werden. Hier das Programm:

```
// Datei: Flaechenberechnung.java

import java.util.Scanner;

class Quadrat
{
   // Hier sind Sie an der Reihe
   . . . . .
}

class Rechteck
{
   // Hier sind Sie gefragt
   . . . . .
}

public class Flaechenberechnung
{
    public static void main (String[] args)
    {
        System.out.println ("Flaechenprogramm\n");

         // Hier sind Sie am Zug
         . . . . .
    }
}
```

Aufgabe 4.6: Instanzmethoden

4.6.1 Ein einfacher Taschenrechner

Schreiben Sie einen einfachen Taschenrechner, der die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division unterstützt. Beim Starten des Taschenrechners wird der Benutzer gefragt, welche Rechenoperation er ausführen möchte. Die Auswahl der Rechenoperation erfolgt durch die Eingabe einer ganzen Zahl zwischen `1` und `4`.

Anschließend werden vom Taschenrechner zwei Zahlen `a` und `b` vom Typ `int` eingelesen, die als Operanden für die ausgewählte Rechenoperation verwendet werden.

Zum Bestimmen des Ergebnisses müssen Sie ein Objekt der Klasse `Rechner` erzeugen und dort die entsprechende Instanzmethode aufrufen.

Der Taschenrechner wird solange ausgeführt, bis der Benutzer die Frage zum Beenden des Taschenrechners mit der Zahl `0` beantwortet.

Achten Sie bei der Division darauf, ob das richtige Ergebnis zurückgeliefert wird.

Fehlende Stellen sind im Programmcode mit `. . . . .` gekennzeichnet:

```
// Datei: Taschenrechner.java
import java.util.Scanner;

public class Taschenrechner
{
   public static void main (String[] args)
   {
      Scanner eingabe = . . . . . ;
      int . . . . . ;
      float . . . . . ;

      do
      {
         System.out.println ("\n=======================");
         System.out.println ("Einfacher Taschenrechner");
         System.out.println ("========================\n");
         System.out.println ("\nFolgende Operationen sind "
                             + "verfuegbar:");
         System.out.println ("Addition...............[1]");
         System.out.println ("Subtraktion............[2]");
         System.out.println ("Multiplikation.........[3]");
         System.out.println ("Division...............[4]");
         System.out.print ("Treffen Sie Ihre Auswahl: ");
         auswahl = eingabe.nextInt();

         System.out.print ("Bitte geben Sie die "
                           + "erste Zahl ein: ");
         a = eingabe.nextInt();
         System.out.print ("Bitte geben Sie die "
                           + "zweite Zahl ein: ");
         b = eingabe.nextInt();

         Rechner rechner = . . . . . ;

         if (auswahl == 1)
         {
            ergebnis = . . . . . ;
            System.out.println( a + " + " + b + " = " + ergebnis);
         }
         else if (auswahl == 2)
         {
            ergebnis = . . . . . ;
            System.out.println( a + " - " + b + " = " + ergebnis);
         }
         else if (auswahl == 3)
         {
            ergebnis = . . . . . ;
            System.out.println( a + " * " + b + " = " + ergebnis);
         }
         else if (auswahl == 4)
         {
            ergebnis = . . . . . ;
            System.out.println( a + " / " + b + " = " + ergebnis);
         }
         else
         {
            System.out.println("\nUnbekannte Auswahl getroffen !");
         }
```

```
         System.out.println
               ("\nWollen Sie den Taschenrechner beenden?");
         System.out.print ("So geben Sie '0' ein: ");
         prgEnde = eingabe.nextInt();
      } while (. . . . .);
   }
}

class Rechner
{
   public int addition (. . . . .)
   {
      . . . . .
   }

   public int subtraktion (. . . . .)
   {
      . . . . .
   }

   public int multiplikation (. . . . .)
   {
      . . . . .
   }

   public float division (. . . . .)
   {
      . . . . .
   }
}
```

4.6.2 Ein erweiterter Taschenrechner

Erweitern Sie den Taschenrechner aus Aufgabe 4.6.1 um die Funktionalität zur Berechnung von Zinseszinsen. Verwenden Sie dazu folgende Formel:

K_0: Anfangskapital (≥ 0)
K_n: Endkapital nach n Jahren

p: Zinsatz (≥ 0 %)
n: Laufzeit der Verzinsung nach n ganzen Jahren (≥ 0)

$K_n = K_0 * (1 + p/100)^n$

Beachten Sie folgende Punkte:

- Sie müssen das Menü der verfügbaren Rechenoperationen erweitern.
- Die Berechnung der Zinseszinsen benötigt 3 anstatt 2 Operanden, d. h., die Anzahl der einzulesenden Zahlen ist abhängig von der ausgewählten Rechenoperation.
- Die obige Zinseszins-Formel enthält einen Term mit einer Potenz. Verwenden Sie zum Potenzieren NICHT die entsprechende Methode der Java-Bibliotheksklasse `Math`. Erweitern Sie stattdessen die Klasse `Rechner` um eine einfache Hilfsmethode, welche die Potenz mit Hilfe einer Schleife berechnet. Die Hilfsmethode muss keine Sonderfälle abdecken, es genügt, wenn sie mit positiven Exponenten und Basen funktioniert. Tipp: $(\ldots)^3 = (\ldots) * (\ldots) * (\ldots)$.

- Die Klasse `Rechner` muss um eine Instanzmethode zum Berechnen von Zinseszinsen erweitert werden. Diese Methode verwendet die oben genannte Hilfsmethode.

4.6.3 Ein einfacher Temperaturkonverter

Verwenden Sie den einfachen Taschenrechner aus Aufgabe 4.6.1 als Vorlage und schreiben Sie das Programm so um, dass daraus ein einfacher Temperaturkonverter entsteht. Der Temperaturkonverter liest einen Temperaturwert (Datentyp `float`) von der Kommandozeile ein. Anschließend kann der Benutzer aus einem Menü auswählen, ob die Temperatur von Grad Celsius nach Grad Fahrenheit oder umgekehrt umgerechnet werden soll. Nach der Umrechnung wird das Ergebnis wieder auf der Kommandozeile ausgegeben.

Die Temperaturumrechnung erfolgt gemäß den folgenden Formeln:

°C = (°F - 32) * 5/9
°F = °C * 1,8 + 32

Beachten Sie folgende Punkte:

- Das Auswahlmenü der verfügbaren Umrechnungen soll so oft angezeigt werden, bis eine gültige Option ausgewählt wird.
- Die Klasse `Rechner` wird durch eine Klasse `Konverter` ersetzt, welche die oben genannten Temperaturumrechnungen in Form von Instanzmethoden bereitstellt.
- Nach der Ausgabe des Ergebnisses wird der Benutzer gefragt, ob das Programm beendet werden soll. Dieses Mal soll das Programm nicht durch Eingabe der Zahl `0` beendet werden, sondern durch Eingabe des Zeichens `'j'`.

Kapitel 5

Lexikalische Konventionen

5.1 Zeichenvorrat von Java
5.2 Der Unicode
5.3 Lexikalische Einheiten
5.4 Zusammenfassung
5.5 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_5

5 Lexikalische Konventionen

"Lexikalisch" bedeutet "ein Wort (eine Zeichengruppe) betreffend", ohne den Textzusammenhang (Kontext), in dem dieses Wort steht, zu berücksichtigen. Im Folgenden werden also die "Rechtschreibregeln", um Wörter in der Programmiersprache Java zu bilden, besprochen.

Wer nur für "die Schule programmiert" und viel Zeit hat, kann sich auch vom Compiler belehren lassen. Da Fehlermeldungen in manchen Fällen wie das Orakel von Delphi klingen können, kann die Fehlerbeseitigung durchaus zu einer spannenden Geschichte werden. Wer industriell programmieren möchte und von vornherein möglichst keine Fehler machen will, wird sich aus Effizienzgründen zuerst mit den Regeln befassen. Es bleiben ihm dann einige langwierige Diskussionen mit dem Compiler erspart.

Zu Beginn befasst sich Kapitel 5.1 mit dem Zeichenvorrat von Java. Anschließend wird in Kapitel 5.2 der Unicode-Zeichensatz erläutert, auf dem die Programmiersprache Java basiert. Die lexikalischen Einheiten von Java, d. h. die Wörter der Sprache Java, wie beispielsweise reservierte Wörter, werden in Kapitel 5.3 vorgestellt.

5.1 Zeichenvorrat von Java

Ein Java-Programm wird als Programmtext aus einer Folge von lexikalischen Einheiten geschrieben. In der Regel wird ein solches Programm am Bildschirm erstellt. Es erhält damit zwangsläufig eine Zeilenstruktur[68].

Eine jede **Anweisung** endet mit einem **Strichpunkt**. Normalerweise schreibt man in eine Zeile nur eine einzige Anweisung. Eine Anweisung kann sich aber auch über mehrere Zeilen erstrecken oder mehrere Anweisungen können in einer einzigen Zeile stehen.

Jede lexikalische Einheit darf nur Zeichen aus dem **Zeichenvorrat** (dem **Zeichensatz**) der Sprache umfassen.

Java benutzt den **Unicode**-Zeichensatz.

Der Zeichenvorrat von Java umfasst:

- **Buchstaben**
  lateinische Buchstaben nach ASCII und ISO-Latin-1[69] als Groß- und Kleinbuchstaben:
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  a b c d e f g h i j k l m n o p q r s t u v w x y z

[68] Eine **Zeile** wird vom Programmierer durch Betätigung der `<RETURN>`-Taste abgeschlossen.
[69] ISO-Latin-1 ist ein 8-Bit-Zeichensatz.

den Unterstrich _, das Dollarzeichen $, das Pfundzeichen £ und weitere Währungssymbole sowie weitere Buchstaben aus dem Unicode (siehe Kapitel 5.2) wie z. B. griechische Symbole

- **Ziffern**
  Ziffern nach ASCII und ISO-Latin-1
  `0 1 2 3 4 5 6 7 8 9`
  und weitere Ziffern aus dem Unicode wie z. B. thailändische Ziffern

- **das Leerzeichen** (**blank**)

- **die Steuerzeichen**
  Zeilenende-Zeichen, horizontaler Tabulator und Seitenvorschub

- die **Sonderzeichen** für die **Satzzeichen**
  `() {} [] ; , .`

- die **Sonderzeichen** für die **Operatoren** wie z. B.
  `= > < ! ~ ? : & | + - * / ^ %`

- für **Ersatzdarstellungen** das **Sonderzeichen**
  `\`

- für **Zeichen** das einfache Hochkomma
  `'`

- und für **Strings** (**konstante Zeichenketten**) das **doppelte Anführungszeichen**
  `"`

In Java gehören der Unterstrich _ und das $-Zeichen sowie weitere Buchstaben aus dem Unicode mit zu den **Buchstaben** des Zeichensatzes von Java. Das Semikolon `;` dient als Satzzeichen und wird hauptsächlich zum Abschluss einer Anweisung verwendet. Die Sonderzeichen für die **Operatoren** werden gebraucht, um Operatoren darzustellen wie z. B. den Zuweisungsoperator `=` oder das logische UND, welches durch den Operator `&&` bzw. den Operator[70] `&` dargestellt wird. Verschiedene Sonderzeichen finden sich sowohl bei den Operatoren als auch bei den **Satzzeichen**. Ein Beispiel hierfür sind die runden Klammern, die als Operator für einen Methodenaufruf und als Satzzeichen zum Einschließen der Bedingung bei einer Selektion Verwendung finden. Zeichen werden begrenzt durch einfache Hochkommata, wie z. B. `'a'`. Konstante Zeichenketten werden begrenzt durch Anführungszeichen wie z. B. in `"Zeichenkette"`. Das Sonderzeichen `\` wird für die Ersatzdarstellungen benötigt, die im Folgenden vorgestellt werden. Dieses Sonderzeichen wird als **Backslash** (**Gegenschrägstrich**) bezeichnet.

Java-Programme kann man vollständig in ASCII-Zeichen schreiben. Ist ein gewünschtes Unicode-Zeichen nicht auf der Tastatur verfügbar – eine Tastatur hat üblicherweise einen sehr eingeschränkten Zeichensatz – so kann dieses Zeichen durch eine **Ersatzdarstellung** der Form $\backslash u x_1 x_2 x_3 x_4$ mit Hilfe von ASCII-Zeichen dargestellt werden.

[70] Siehe Kapitel 7.6.5

Der Backslash \, das u und die Zeichen x_1, x_2, x_3 und x_4 sind ASCII-Zeichen. x_1, x_2, x_3 und x_4 sind dabei hexadezimale Zeichen, d. h. Zeichen aus dem Wertevorrat 0, 1, 2, ... , 9, A, B, ... , F. Die Buchstaben A ... F sind Abkürzungen. A entspricht der 10, B der 11, usw. bis F schließlich der 15 entspricht. Es spielt dabei keine Rolle, ob die Buchstaben A ... F groß oder klein geschrieben werden. a ist äquivalent zu A, b ist äquivalent zu B, usw.

Die Ersatzdarstellung

$\backslash u x_1 x_2 x_3 x_4$

entspricht dem Unicode-Zeichen, welches an der Position mit der dezimalen Nummer

$x_1 * 16^3 + x_2 * 16^2 + x_3 * 16^1 + x_4 * 16^0$

des Unicode-Zeichensatzes steht.

Groß- und Kleinschreibung

Java ist **case sensitiv**. Das bedeutet, dass Groß- und Kleinbuchstaben in Java streng unterschieden werden.

Alle reservierten Wörter müssen klein geschrieben werden. Namen, die sich nur durch Groß- bzw. Kleinschreibung unterscheiden, stellen verschiedene Namen dar. So ist beispielsweise der Name alpha ein anderer Name als Alpha.

5.2 Der Unicode

Java basiert auf einem Zeichensatz, der geeignet ist, viele Zeichen aufzunehmen. Der gewählte Zeichensatz ist der sogenannte Unicode. Unter der Adresse

http://www.unicode.org

kann man im Internet Informationen zu diesem Code abrufen[71].

Der Unicode war ursprünglich ein 16-Bit-Code, bestehend aus 2 Bytes. Der Unicode-Standard erlaubt in der Zwischenzeit auch Zeichen, deren Darstellung mehr als 16 Bits erfordert. Der Zahlenraum des Unicodes erstreckt sich seit dem Unicode-Standard 2.0 von 0 bis $10FFFF_{16}$, wodurch insgesamt 1.114.112 Zeichen möglich sind. Der Unicode-Standard definiert drei Unicode-Codierformen, nämlich UTF-8[72], UTF-16 und UTF-32. Die Abkürzung **UTF** steht für **Unicode Transformation Format** und die Zahl jeweils für die minimale Anzahl Bits, die im Speicher benötigt werden. Java verwendet für Zeichen die UTF-16-Darstellung, wobei Zeichen, die einem Wert größer als FFFF entsprechen, als ein Paar von Zeichen dargestellt werden. Das heißt, dass Java mit 16-Bit-Unicode-Zeichen arbeitet. Die ersten 128 Zeichen des

[71] Der Unicode Standard, Version 5.0, ist als Buch erhältlich [Uni07]. Aktuell ist der Unicode in der Version 8.0.

[72] UTF = Unicode Transformation Format.

verwendeten klassischen Unicodes sind die Zeichen des 7-Bit-ASCII-Zeichensatzes und seine ersten 256 Zeichen die Zeichen des Zeichensatzes ISO-Latin-1.

Der Unicode erlaubt es, wesentlich mehr Zeichen zu verwenden als beispielsweise der Code ISO Latin-1 und zahlreiche andere Codes.

Für die Kodierung der Zeichen gibt es die Formate UTF-8, UTF-16 und UTF-32. Unabhängig davon, welche der Kodierungsarten gewählt wird, steht immer der volle Unicode-Zeichensatz zur Verfügung.

Java verwendet die Kodierung im Format UTF-16.

5.3 Lexikalische Einheiten

Ein Programm besteht für einen Compiler zunächst nur aus einer Folge von Zeichen. Der **Scanner**-Anteil des Compilers hat die Aufgabe, Zeichengruppen zu finden. Zeichengruppen werden gefunden, indem man nach den **Trennern** sucht, beispielsweise einem **Leerzeichen** (**Whitespace-Zeichen**) oder einem Kommentar. Whitespace-Zeichen werden in Kapitel 5.3.1.1, Kommentare in Kapitel 5.3.1.2 behandelt. Stehen zwischen zwei Trennern noch weitere Zeichen, die keine Trenner enthalten, so ist eine **lexikalische Einheit** (**Token**) gefunden.

Die Wörter oder **Zeichengruppen**, aus denen ein Programmtext aufgebaut ist, werden als **lexikalische Einheiten** bzw. als **Token** bezeichnet.

Diese lexikalischen Einheiten werden dann vom **Parser** auf die Einhaltung der Syntax geprüft. Lexikalische Einheiten sind die Wörter einer Sprache.

Lexikalische Einheiten sind:

- **Namen**,
- **Schlüsselwörter** (**reservierte Wörter**),
- **Konstanten**,
- **Satzzeichen**
- und **Operatoren**.

Ersatzdarstellungen $\backslash u x_1 x_2 x_3 x_4$ werden erkannt und in Unicode-Zeichen umgesetzt.

Bei der Kompilierung wird vor der Ermittlung der lexikalischen Einheiten eine vorliegende Java-Quelltext-Datei, die z. B. in ASCII oder ISO-Latin-1 geschrieben wurde, in Unicode gewandelt.

Zu beachten ist, dass in Java wie in C **Operatoren und Satzzeichen auch** als **Trenner** wirken. Dies hat zum Beispiel die Konsequenz, dass Operatoren und Operanden zusammengeschrieben werden können, wie z. B. `a+b`.

Der Compiler bildet immer die größtmöglichen Token. So wird `a--b` stets als `a --b` erkannt (wie später noch erklärt wird, ist `--` der Dekrement-Operator) und nicht als `a - -b` (`a` abzüglich minus `b`), selbst wenn aus dieser Interpretation ein Kompilierfehler resultiert.

5.3.1 Trenner

Eine lexikalische Einheit wird gefunden, indem man die Trenner findet, die sie begrenzen.

Trenner (**Separatoren**) sind **Zwischenraum** (**Whitespace-Zeichen**), **Kommentare**, **Satzzeichen** und **Operatoren**.

Für den Compiler ist beispielsweise `A&&B` das logische UND (`&&`) zwischen `A` und `B`, da Operatoren Trenner sind. Denkt man an den menschlichen Leser des Programms, so empfiehlt es sich, nicht die Trenner-Eigenschaft der Operatoren zu verwenden, sondern nach jeder lexikalischen Einheit Leerzeichen einzugeben, damit das Programm leichter lesbar ist. Im genannten Beispiel ist es also besser, `A && B` zu schreiben!

5.3.1.1 Whitespace-Zeichen

Zu den **Whitespace-Zeichen** gehören **Leerzeichen**, **horizontaler Tabulator**, **Zeilentrenner** und **Seitenvorschub**.

Zwischen zwei aufeinanderfolgenden lexikalischen Einheiten kann eine beliebige Anzahl an Whitespaces eingefügt werden. Damit hat man die Möglichkeit, ein Programm optisch so zu gestalten, dass die Lesbarkeit verbessert wird. Üblicherweise wird vor jeder Methode mindestens eine Leerzeile eingefügt oder innerhalb eines Blocks[73] etwas eingerückt.

5.3.1.2 Kommentare

Java hat drei verschiedene Arten von Kommentaren:

- Kommentarblock,
- Zeilenkommentar
- und Dokumentationskommentar.

[73] Ein **Block** wird begrenzt durch die Blockbegrenzer `{` und `}`. Die Definition eines Blockes wird in Kapitel 9 behandelt.

Kommentare dienen dazu, die Bedeutung von Anweisungen und Programmeinheiten schriftlich direkt an der entsprechenden Stelle im Quellcode festzuhalten. Da Kommentare Trenner sind, dürfen sie nicht innerhalb von Zeichenkonstanten (siehe Kapitel 5.3.5.4) oder konstanten Zeichenketten (siehe Kapitel 5.3.5.5) auftreten.

Kommentarblock

Ein Kommentarblock wird durch die Zeichen `/*` eingeleitet und durch die Zeichen `*/` beendet. Der ganze Block zwischen `/*` und `*/`, der auch über mehrere Zeilen gehen kann, wird vom Compiler als Trenner betrachtet und ignoriert. Kommentarblöcke dürfen nicht verschachtelt werden.

Beispiele:

```
/* dies ist ein Kommentar                    */
/* dieser Unfug /* ist auch ein Kommentar    */
```

Zeilenkommentar

Alle Zeichen von `//` bis zum Zeilenende werden vom Compiler ignoriert.

Beispiele:

```
// Hier ist die ganze Zeile Kommentar
int x; // Hier ist nur ein Teil der Zeile Kommentar
```

Dokumentationskommentar

Mit Hilfe eines Dokumentationskommentars kann man im Gegensatz zum Kommentarblock und zum Zeilenkommentar nicht nur den Quellcode kommentieren, sondern es besteht auch die Möglichkeit, den Dokumentationskommentar unter Verwendung eines Werkzeugs zu einer richtigen Dokumentation in Form von HTML-Dateien aufbereiten zu lassen. Das Werkzeug **`javadoc`** zur online-Dokumentation von Programmen, das Teil des JDK ist, filtert alle Informationen, die sich zwischen den Kommentarsymbolen `/**` und `*/` befinden, heraus und legt diese Information in HTML-Dateien ab. Diese HTML-Dateien können dann mit einem Webbrowser angesehen werden.

Dokumentationskommentare können nur direkt **vor** einer **Klassendefinition**, einem **Datenfeld**, einer **Methode** oder einem **Konstruktor** stehen[74]. Das folgende Beispiel zeigt eine Klasse, die mit Dokumentationskommentaren versehen ist:

```
// Datei: DocuTest1.java

/** Ich bin ein Kommentar und erläutere die Klasse DocuTest1 */
public class DocuTest1
{
   /** Ich bin ein Kommentar und erläutere das Datenfeld x   */
   public int x;
```

[74] Sie können auch vor Schnittstellen stehen. Schnittstellen (siehe Kapitel 14) sind an dieser Stelle noch nicht bekannt.

```
    /** Ich bin ein Kommentar und erläutere die Methode
      * meth()
      */

public void meth()
{
       // Weitere Anweisungen
    }
}
```

Das folgende Bild zeigt einen Ausschnitt aus den mit **`javadoc`** generierten HTML-Dateien in einem Webbrowser:

Package Class Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

Class DocuTest1

java.lang.Object
DocuTest1

```
public class DocuTest1
extends java.lang.Object
```

Ich bin ein Kommentar und erläutere die Klasse DocuTest1

Field Summary

Fields

Modifier and Type	Field and Description
int	x Ich bin ein Kommentar und erläutere das Datenfeld x

Constructor Summary

Constructors

Constructor and Description
DocuTest1()

Method Summary

Methods

Modifier and Type	Method and Description
void	meth() Ich bin ein Kommentar und erläutere die Methode meth()

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Bild 5-1 Ausschnitt aus den erzeugten HTML-Dateien

Um aus Dokumentationskommentaren HTML-Dateien zu erzeugen, wie sie in Bild 5-1 ausschnittsweise dargestellt sind, geht man am besten wie im folgenden Beispiel vor:

- In das Verzeichnis wechseln, in dem die `.java`-Datei liegt – hier die Datei `DocuTest1.java`.
- In der Kommandozeile `javadoc DocuTest1.java` eingeben.

Es ist zu beachten, dass nur aus den Quellcode-Dateien `*.java` Dokumentationskommentare extrahiert werden können, da in den `.class`-Dateien diese Information gar nicht mehr vorhanden ist.

Es werden mehrere HTML-Dateien erzeugt, die alle in das aktuelle Verzeichnis gelegt werden. Es lohnt sich auf jeden Fall, alle diese Dateien mit einem Webbrowser anzuschauen. Die wesentliche Information steht in der Datei `Dateiname.html` – hier also in der Datei `DocuTest1.html`. Ein Ausschnitt der Datei `DocuTest1.html` in einem Webbrowser wurde in Bild 5-1 gezeigt.

Zusätzlich zu den Kommentaren von Klassen, Datenfeldern, Methoden und Konstruktoren können noch Übergabeparameter, Rückgabewerte und vieles mehr genauer durch sogenannte Tags beschrieben werden. Diese Tags werden innerhalb der **Dokumentationskommentare für Klassen**, **Datenfelder**, **Methoden** und **Konstruktoren** verwendet. Die folgende Tabelle zeigt einen Ausschnitt der vorhandenen Tags und beschreibt, mit welchem **Kommentartyp** (**Klasse**, **Datenfeld** oder **Methode**) diese eingesetzt werden können:

Tag	Bedeutung	Kommentartyp
@see	erstellt einen Link zu anderen Klassen	Klasse, Datenfeld, Methode
@version	gibt die Version an	Klasse
@author	gibt den Autor an	Klasse
@param	beschreibt einen Parameter näher	Methode
@return	beschreibt den Rückgabewert	Methode
@exception	beschreibt die Exception näher	Methode
@deprecated	markiert ein Element als deprecated[75]	Klasse, Datenfeld, Methode

Tabelle 5-1 Tags zum Einsatz mit Dokumentationskommentaren

Eine ausführliche Beschreibung der oben genannten und eine Auflistung aller verfügbaren Tags kann auf der `javadoc`-Homepage

`http://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/index.html`

nachgelesen werden.

Im folgenden Beispiel werden einige der aufgelisteten Tags verwendet:

[75] Engl. für "überholt". Beschreibt, dass das markierte Element nicht mehr verwendet werden soll, da es in neueren Versionen der Bibliothek nicht mehr vorhanden sein muss.

```
// Datei: DocuTest2.java
/** Ich bin ein Kommentar und erläutere die Klasse DocuTest2
  * @version 1.0
  * @author  Rainer Brang
  */
public class DocuTest2
{
   /** Ich bin ein Kommentar und erläutere das Datenfeld x */
   public int x;

   /** Erläuterung der Methode meth()
     * @param para Hier die Beschreibung des Parameters
     * @return  Kein Rückgabewert
     */
   public void meth (int para)
   {
      // Anweisungen
   }
}
```

Dem Werkzeug `javadoc` können beim Aufruf einige Optionen mitgegeben werden, mit deren Hilfe sich die Ausgabe in die HTML-Dateien steuern lässt. Wird für obiges Beispiel einfach `javadoc DocuTest2.java` eingegeben, so enthalten die erzeugten HTML-Dateien die Informationen über Autor und Version nicht. Durch Eingabe von `javadoc -version -author DocuTest2.java` werden auch diese Informationen extrahiert. Wer mehr über die Optionen erfahren möchte, gibt auf der Kommandozeile einfach nur `javadoc` ein. Dann wird eine Auflistung der zulässigen Optionen und deren Beschreibung ausgegeben.

5.3.2 Namen

Namen bezeichnen in Java:

- Klassen,
- Methoden,
- Konstruktoren,
- Datenfelder,
- lokale Variablen,
- Parameter (einer Methode, eines Konstruktors oder eines `catch`-Blocks[76]),
- Schnittstellen
- und Pakete.

Ein **Name** (**Bezeichner**) besteht aus einer beliebig langen Zeichenfolge aus Buchstaben und Ziffern, die mit einem Buchstaben beginnt. In Java zählen – wie bereits erwähnt – auch der Unterstrich _ und das `$`-Zeichen sowie andere Währungssymbole zu den Buchstaben.

Zu den bereits vom ASCII-Code bekannten Zeichen kommen die Unicode-spezifischen Buchstaben und Ziffern hinzu.

[76] Siehe Kapitel 13

Generell wird in Java und natürlich auch bei Namen zwischen Groß- und Kleinbuchstaben unterschieden. Reservierte Wörter (siehe Kapitel 5.3.4) und die literalen Konstanten `true`, `false` und `null` dürfen nicht als Namen verwendet werden.

5.3.3 Styleguide für die Programmierung

Eine **Gestaltungsrichtlinie** (engl. **style guide**) für das Erstellen von Programmen umfasst Darstellungsregeln, welche die Lesbarkeit der Programme erleichtern sollen. Solche Regeln beruhen immer auf einer gegenseitigen Übereinkunft (Konvention)

Die Anwendung dieser Regeln ist nicht zwingend, da nicht die Korrektheit, sondern die Lesbarkeit der Programme von ihrer Einhaltung abhängt. Ein Programmierer, der sich an den im Projekt vereinbarten Styleguide hält, arbeitet teamorientiert, da seine Programme übersichtlich sind.

Zum Styleguide gehört auch eine Konvention über das Einrücken in Blöcken oder die Konvention, dass vor der Definition einer Methode eine Leerzeile stehen soll, damit man leichter erkennen kann, dass jetzt eine neue Methode kommt.

Für Namen hat sich in Java der folgende **Programmierstil** durchgesetzt:

Name	Konvention	Beispiel
Variablennamen	Kleinbuchstaben	`variable`
Datenfeldnamen	Kleinbuchstaben	`vorname`
Methodennamen	Kleinbuchstaben	`methode()`
Klassennamen	1. Buchstaben groß, Rest klein	`Person`
symbolische Konstanten[77]	alle Buchstaben groß	`MAXIMUM`

Tabelle 5-2 Styleguide-Konventionen

Aus mehreren Wörtern zusammengesetzte Namen werden ohne Unterstrich geschrieben. Dabei wird ab dem zweiten Wort jeweils der erste Buchstabe eines Wortes groß geschrieben, so dass jedes Wort eines Nanems leicht erkannt werden kann (**CamelCase-Notation**). Für das erste Wort gilt die normale Konvention. Beispiele hierfür sind `dritteWurzel` als Variablenname oder `verschiebeSchwerpunkt()` als Methodenname. Im Falle der symbolischen Konstanten werden, da alle Buchstaben groß geschrieben sind, zur optischen Trennung der Wörter wie im Falle von `MAX_VALUE` Unterstriche verwendet.

5.3.4 Reservierte Wörter

Die **Schlüsselwörter bzw. reservierten Wörter** in der folgenden Tabelle sind in Java reserviert. Sie müssen stets klein geschrieben werden. Die Bedeutung dieser Schlüsselwörter ist festgelegt und kann nicht verändert werden.

[77] Siehe Kapitel 5.3.5

Eine vollständige Erklärung dieser Schlüsselwörter kann erst in späteren Kapiteln erfolgen.

Hier die Schlüsselwörter:

`abstract`	Dient zur Deklaration abstrakter Klassen und Methoden.
`assert`	Hiermit lassen sich Zusicherungen (Assertions) gezielt überprüfen.
`boolean`	Einfacher Java-Datentyp, der einen booleschen Wahrheitswert enthält.
`break`	Zum Herausspringen aus Schleifen oder der `switch`-Anweisung.
`byte`	Einfacher Java-Datentyp, der eine 8-Bit-Zahl enthält.
`case`	Auswahl-Fall in der `switch`-Anweisung.
`catch`	Leitet einen Programmblock zur Ausnahmebehandlung ein.
`char`	Einfacher Java-Datentyp, der ein 16-Bit-Unicode-Zeichen enthält.
`class`	Dient zur Deklaration einer Klasse.
`const`	Dieses Schlüsselwort ist reserviert, wird aber nicht benutzt.
`continue`	Starten eines neuen Durchgangs in einer Schleife.
`default`	Standard-Einsprungmarke in einer `switch`-Anweisung.
`do`	Teil einer Schleifen-Anweisung.
`double`	Einfacher Java-Datentyp, der eine 64-Bit-Fließkommazahl enthält.
`else`	Teil einer bedingten Anweisung.
`enum`	Dient zur Definition eines Aufzählungstyps.
`extends`	Dient zur Angabe der Basisklasse bei der Klassendeklaration.
`final`	Modifikator für Klassen, Methoden, Datenfelder und Variablen.
`finally`	Dient zur Einleitung des `finally`-Blocks einer `try`-Anweisung.
`float`	Einfacher Java-Datentyp, der eine 32-Bit-Fließkommazahl enthält.
`for`	Schleifenanweisung.
`goto`	Dieses Schlüsselwort ist reserviert, wird aber nicht benutzt.
`if`	Teil einer bedingten Anweisung.
`implements`	Gibt bei der Klassendeklaration an, welche Schnittstelle implementiert wird.
`import`	Dient zur Bekanntgabe von Klassen und Schnittstellen aus anderen Paketen.
`instanceof`	Operator, der überprüft, ob eine Referenz auf ein Objekt zeigt, das vom Typ einer bestimmten Klasse ist.
`int`	Einfacher Java-Datentyp, der eine 32-Bit-Ganzzahl enthält.
`interface`	Dient zur Deklaration einer Schnittstelle.
`long`	Einfacher Java-Datentyp, der eine 64-Bit-Ganzzahl enthält.
`native`	Dient als Modifikator für Methoden, die in einer anderen Sprache als Java implementiert sind.
`new`	Erzeugt ein neues Objekt auf dem Heap.
`package`	Dient zur Deklaration eines Paketes.
`private`	Zugriffsmodifikator für Methoden, Konstruktoren, Datenfelder und Elementklassen.
`protected`	Zugriffsmodifikator für Methoden, Konstruktoren, Datenfelder und Elementklassen.
`public`	Zugriffsmodifikator für Methoden, Konstruktoren, Datenfelder, Elementklassen sowie Klassen und Schnittstellen.
`return`	Anweisung für den Rücksprung aus einer Methode zur aufrufenden Methode.
`short`	Einfacher Java-Datentyp, der eine 16-Bit-Ganzzahl enthält.

`static`	Dient als Modifikator für Methoden, Datenfelder und Klassen und wird auch beim statischen Initialisierungsblock verwendet.
`strictfp`	Kennzeichnet eine Methode, die Zwischenwerte von Operationen nach IEEE 754 (siehe [IEEE 754]) berechnet.
`super`	Erlaubt den Zugriff im eigenen Objekt auf überschriebene Methoden und verdeckte Datenfelder, die von der Basisklasse geerbt wurden. Weiterhin wird mit `super` der Aufruf des Konstruktors der Vaterklasse ermöglicht.
`switch`	Auswahlanweisung.
`synchronized`	Dient zur Thread-Synchronisation.
`this`	Bezeichnet eine Referenz auf das eigene Objekt oder dient zum Aufruf eines klasseneigenen Konstruktors.
`throw`	Dient zum Werfen einer Ausnahme.
`throws`	Dient zur Auflistung der Ausnahmen bei der Deklaration von Methoden.
`transient`	Kennzeichnung für Datenfelder, die bei der Serialisierung nicht berücksichtigt werden sollen.
`try`	Kennzeichnet einen Programmblock, in dem eine Ausnahme auftreten kann.
`void`	Dient zur Anzeige, dass eine Methode keinen Rückgabewert hat.
`volatile`	Kennzeichnet ein Datenfeld, das gleichzeitig von mehreren Threads verändert werden kann.
`while`	Schleifenanweisung.

5.3.5 Literale und symbolische Konstanten

In Java gibt es zwei Arten von Konstanten:

- symbolische (benannte) Konstanten
- und literale Konstanten.

Überall da, wo von der Syntax her Konstanten oder konstante Ausdrücke erlaubt sind, kann man literale Konstanten oder symbolische Konstanten einsetzen.

Überall da, wo von der Syntax her an einer Stelle im Programm Konstanten stehen können, können auch konstante Ausdrücke[78] notiert werden. Dies liegt daran, dass ein konstanter Ausdruck ein Ausdruck ist, an dem nur Konstanten beteiligt sind.

Symbolische Konstanten

Symbolische Konstanten haben einen Namen, der ihren Wert repräsentiert.

[78] Ein **konstanter Ausdruck** ist eine Verknüpfung von Konstanten mit Operatoren und runden Klammern. Konstante Ausdrücke spielen beispielsweise eine Rolle bei den `case`-Marken der `switch`-Anweisung (siehe Kapitel 8.2.3).

Symbolische Konstanten sind Variablen, **die nach ihrer Initialisierung nicht verändert werden dürfen**.

Zur Definition von symbolischen Konstanten wird in Java das Schlüsselwort `final` verwendet. Im folgenden Beispiel wird die symbolische Konstante `UPPER` definiert:

```
final int UPPER = 300;
```

Das Schlüsselwort `final`, welches die Konstanz von Variablen gewährleistet, kann auf Klassenvariablen, Instanzvariablen und lokale Variablen (siehe Kapitel 9.1.3) in Methoden angewandt werden.

Symbolische Konstanten, die zusammengehören und oft gebraucht werden, werden in der Regel in einer Klasse gruppiert wie in folgendem Beispiel:

```
public final class Math[79]
{
   public static final double PI = 3.141592654;
   public static final double E = 2.718281828;
   . . . . .
}
```

Da es unerwünscht ist, dass symbolische Konstanten als konstante Instanzvariablen angelegt werden – Konstanten sind für jedes Objekt einer Klasse gleich –, werden symbolische Konstanten als konstante Klassenvariablen angelegt. Damit werden sie nur ein einziges Mal erzeugt, wie viele Objekte es auch immer von dieser Klasse gibt. Klassenvariablen werden in Java mit Hilfe des Schlüsselwortes `static` erzeugt.

Literale Konstanten

Literale Konstanten – oft auch nur **Konstanten** oder **Literale** genannt – haben keinen Namen, sie werden durch ihren Wert dargestellt.

So ist im letzten Beispiel die Zahl `3.141592654` eine literale Konstante.

Es gibt verschiedene Arten von literalen Konstanten:

- **ganzzahlige Konstanten**,
- **Gleitpunktkonstanten**,
- **boolesche Konstanten**,
- **Aufzählungskonstanten** (bei **enum**, siehe Kapitel 6.6),
- **Zeichenkonstanten**,
- **String-Konstanten** (**konstante Zeichenketten**)
- und die **Nullkonstante**.

[79] Die Klasse `Math` stellt eine Bibliotheksklasse dar.

Jede dieser Konstanten hat einen definierten Datentyp. Aufzählungskonstanten werden in Kapitel 6.6 besprochen, die anderen werden im Folgenden betrachtet.

5.3.5.1 Ganzzahlige Konstanten

Ganzzahlige Konstanten wie zum Beispiel die Zahl `1234` sind vom Typ `int`.

Wenn der Typ-Suffix `l` oder `L` an eine Konstante angehängt ist, so bedeutet das, dass die Konstante vom Typ `long`[80] ist.

Zahlensysteme

Ganzzahlige Konstanten können in verschiedenen Zahlensystemen aufgeschrieben werden. Neben der normalen **Dezimaldarstellung** ist auch die Angabe als oktale **Konstante** (Basis 8) oder **hexadezimale Konstante** (Basis 16) sowie seit Java 7 als **binäre Konstante** (Basis 2) möglich.

Eine ganzzahlige **dezimale** Konstante ist die Konstante `0`, sowie Konstanten, die mit einer Ziffer zwischen `1` und `9` beginnen.

Konstanten, die mit einer `0` beginnen und weitere Ziffern haben, werden als **oktale** Zahlen interpretiert. Oktalzahlen haben die Ziffern `0`, `1`, `2`, ..., `7`.

Beginnt eine Konstante mit `0X` oder `0x`, also einer Null, gefolgt von einem großen oder einem kleinen `x`, so wird die Zahl **hexadezimal** interpretiert. Hexadezimale Ziffern sind: `0`, `1`, ..., `9`, `a` (oder `A`), `b` (oder `B`), `c` (oder `C`), `d` (oder `D`), `e` (oder `E`) und `f` (oder `F`).

Seit Java 7 können ganzzahlige Konstanten als **binäre Konstanten** aufgeschrieben werden. Beginnt eine Konstante mit `0b` oder `0B`, so wird die Zahl **binär** interpretiert. Binäre Ziffern sind `0` und `1`.

Eine allein stehende `0` wird vom Compiler als Dezimalzahl betrachtet, `00` als Oktalzahl, `0x0` als Hexadezimalzahl, `0b0` als Binärzahl.

Alle vier Zahlen haben denselben Wert. Ansonsten gilt natürlich, dass mehrstellige, gleichlautende Ziffernfolgen in den vier Zahlensystemen unterschiedliche Werte haben.

Dezimale Konstanten sind immer **positiv oder null**. **Binäre**, **oktale und hexadezimale Konstanten** können **positiv**, **null oder negativ** sein.

[80] Siehe Kapitel 6.2.2

Beispiele für ganzzahlige Konstanten sind:

`14` `int`-Konstante in Dezimaldarstellung mit dem Wert `14` dezimal
`-14` positive `int`-Konstante, auf die der Vorzeichenoperator `-` angewandt wird
`014` `int`-Konstante in Oktaldarstellung mit dem Wert `12` dezimal
`0x14` `int`-Konstante in Hexadezimaldarstellung mit dem Wert `20` dezimal
`14L` `long`-Konstante in Dezimaldarstellung mit dem Wert `14`

Auf eine Konstante kann der negative **Vorzeichenoperator** angewendet werden. Das Ergebnis dieser Operation ist der mit (-1) multiplizierte Wert der Konstanten.

Der Hintergrund, warum oktale und hexadezimale Konstanten negativ werden können, dezimale Konstanten aber nicht, ist, dass `int`-Werte in der sogenannten Zweierkomplementdarstellung berechnet werden, wobei der Compiler für binäre, oktale und hexadezimale Konstanten beliebige Werte zulässt, solange sie sich mit 32 Bits darstellen lassen, und bei dezimalen Konstanten nur die Werte von `0` bis zur größten positiven Zahl `2147483647`.

Die binäre, oktale und hexadezimale Schreibweise findet ihren Einsatz bei Bitmustern, ansonsten ist die dezimale Schreibweise vorzuziehen.

Den Wert einer binären, oktalen bzw. hexadezimalen Zahl im Dezimalsystem kann man mit Hilfe einer **Stellenwert-Tabelle** berechnen. Solange das oberste Bit – bei einer `int`-Konstanten das 32. Bit, bei einer `long`-Konstanten das 64. Bit – auf `0` gesetzt ist, ist die Berechnung des dezimalen Wertes einer binären, oktalen bzw. hexadezimalen Konstanten ganz einfach, so wie im folgenden Beispiel einer Hexadezimalzahl:

16^2	16^1	16^0
2	0	a

Tabelle 5-3 Stellenwert-Tabelle für Hexadezimalzahlen

So berechnet sich der dezimale Wert der in Tabelle 5-3 dargestellten Hexadezimalzahl `20a` zu 2*162+0*161+10*160 = 522. In entsprechender Weise kann auch bei einer anderen Basis als 16 wie z. B. 8 vorgegangen werden.

Bits werden üblicherweise von rechts nach links durchnummeriert. Das Bit ganz rechts wird als Bit 0, das Bit ganz links in Bild 5-2 als Bit 31 bezeichnet. Ist das oberste Bit in der Zweierkomplementdarstellung auf 1 gesetzt, so ist es hilfreich, eine Wandlung in Bits durchzuführen. Hier das bereits erwähnte Bild 5-2:

Bild 5-2 Die Konstante `0xffffffff`

Eine Ziffer einer hexadezimalen Zahl ist eine Folge von 4 Bits. Eine Folge von 4 Bits wird auch als **Halbbyte** bezeichnet. Die hexadezimale Zahl `0xf` entspricht dem Bitmuster `1111`. Die hexadezimale Zahl `0xffffffff` ist ein Bitmuster aus 8 Halbbytes, welches – wie in Bild 5-2 gezeigt – eine Folge von 32 Einsern ist.

Der Stellenwert von Bit 31 des Typs `int` hat im Rahmen der Zweierkomplementdarstellung ein negatives Vorzeichen. Der Stellenwert ist -2^{31}. Alle anderen Bits haben den üblichen positiven Stellenwert.

Wie man aus dem Beispiel mit 8 Bits in Kapitel 6.2.2 ableiten kann, ist der Wert des angegebenen Bitmusters gleich minus eins (-1).

Mit Java 7 wird die Lesbarkeit von ganzzahligen Konstanten (dezimal, binär, hex und oktal) vereinfacht. Als visuelles Trennglied kann der Unterstrich _ an jeder Stelle in einer Ziffernfolge verwendet werden. So ist die folgende Schreibweise beispielsweise seit Java 7 zulässig:

```
int personalNummer = 50_775_654;
```

Durch die Schreibweise mit Unterstrich kann die Lesbarkeit für den Menschen erhöht werden.

5.3.5.2 Gleitpunktkonstanten

Eine **Gleitpunktkonstante** (**Fließkommakonstante**) wird in Dezimalform angegeben und kann aus

- einer **Mantisse**,
- einem **Exponential-Anteil**
- und aus einer angehängten Typkennung für einen Gleitpunkttyp (**Typ-Suffix**)

bestehen, wie das im folgenden Bild dargestellte Beispiel zeigt:

Bild 5-3 Darstellung einer Gleitpunktkonstanten

Beispiele für Gleitpunktkonstanten sind:

```
500.   .5   -500.6   5E2   5.e2   .5E3   1f
```

Der Teil einer Fließkommazahl vor dem `E` bzw. dem gleichbedeutenden `e` ist die **Mantisse**. Der Exponential-Anteil besteht aus einem `e` oder `E` gefolgt von einer ganzen Zahl, die den Exponenten darstellt. Wird ein **Exponent** angegeben, so ist die Mantisse mit $10^{Exponent}$ zu multiplizieren. Der Dezimalpunkt der Mantisse, der Exponent und der Typ-Suffix können fehlen, aber nicht alle drei zugleich. Entweder der ganzzahlige Anteil vor dem Punkt oder der Dezimalbruch-Anteil nach dem Dezimalpunkt darf fehlen, aber nicht beide zugleich.

Eine Gleitpunktkonstante hat den Typ `double`. Durch die Angabe eines **Typ-Suffixes** `f` oder `F` wird sie zu `float`.

Es ist auch möglich, den Typ-Suffix `d` oder `D` für `double` anzuhängen wie z. B. bei der Zahl `1D`. Die Gleitpunktkonstanten `float` und `double` werden nach dem Format IEEE 754 repräsentiert[81], wobei der Datentyp `float` 32 Bits und der Datentyp `double` 64 Bits hat.

Symbolische Gleitpunkt-Konstanten

In den Wrapper-Klassen[82] `Float` und `Double` gibt es die in Tabelle 5-4 aufgeführten symbolischen Gleitpunkt-Konstanten. In der Wrapper-Klasse `Float` sind die symbolischen Gleitpunkt-Konstanten dabei vom Typ `float` und in der **Wrapper-Klasse** `Double` vom Typ `double`. Dass ein und dieselbe Konstante das eine Mal vom Typ `float` und das andere Mal vom Typ `double` ist, darf nicht verwirren. Angesprochen werden die Konstanten über den Namen der jeweiligen Klasse, also z. B. als `Float.MIN_VALUE` bzw. `Double.MIN_VALUE`. Damit gibt es keine Doppeldeutigkeit. Die folgende Tabelle zeigt symbolische Konstanten für den Typ `float` und `double`:

Konstanten-Name	Bedeutung	Wert
`MIN_VALUE`	Kleinster positiver Wert des Typs `float` bzw. `double`, der von Null verschieden ist.	1.4E-45f für `float` 5E-324 für `double`
`MAX_VALUE`	Größter positiver endlicher Wert des Typs `float` bzw. `double`.	3.4028235E+38f für `float` 1.7976931348623157E+308 für `double`
`NEGATIVE_INFINITY`	Negativ Unendlich des Typs `float` bzw. `double`.	-1.0f/0.0f für `float` -1.0/0.0 für `double`
`POSITIVE_INFINITY`	Positiv Unendlich des Typs `float` bzw. `double`.	1.0f/0.0f für `float` 1.0/0.0 für `double`
`NaN`	Not-a-Number, d. h. eine Zahl außerhalb des Wertebereichs der `float`- bzw. `double`-Zahlen.	0.0f/0.0f für `float` 0.0/0.0 für `double`

Tabelle 5-4 Symbolische Konstanten

Der Wert `NaN` kann beispielsweise verwendet werden, um einen nicht zulässigen Wert anzuzeigen.

5.3.5.3 Boolesche Konstanten

Boolesche Konstanten werden nach dem englischen Mathematiker George Boole (1815-1864), dem Begründer der mathematischen Logik, benannt.

[81] Siehe Kapitel 6.2.3
[82] Siehe Kapitel 6.8

Boolesche Konstanten sind vom Typ `boolean`. **`true`** und **`false`** sind **boolesche Konstanten**. Sie werden als **Wahrheitswert** bezeichnet.

Logische Ausdrücke wie z. B. `a > b` (`a` größer `b`) können prinzipiell nur zwei Werte annehmen. Entweder ist ein logischer Ausdruck wahr oder er ist nicht wahr, d. h., er ist falsch.

In Java werden die literalen Konstanten `true` und `false` als Wahrheitswerte verwendet. Die Konstanten `true` und `false` tragen keinen numerischen Wert.

5.3.5.4 Zeichenkonstanten

Eine **Zeichenkonstante** – auch **Zeichenliteral** genannt – ist ein Zeichen, eingeschlossen in einfache Anführungszeichen.

In Java ist eine Zeichenkonstante vom Typ `char`.

Mit Zeichenkonstanten kann man rechnen wie mit ganzen Zahlen. Man kann sie in ganzzahligen Ausdrücken verwenden. So hat beispielsweise das Zeichen `'0'` im Unicode-Zeichensatz den Wert `48`. Meistens verwendet man Zeichenkonstanten jedoch, um Zeichen zu vergleichen.

Das folgende **Beispiel** zeigt die Ausgabe von Zeichen:

```
// Datei: Zeichen.java

public class Zeichen
{
   public static void main (String[] args)
   {
      char c;
      int d = 0;
      c = '3';
      System.out.println (c); // Ausgabe des Zeichens 3
      c = 49;
      System.out.println (c); // Ausgabe des Zeichens 1
      System.out.println (d); // Ausgabe der int-Zahl 0
      d = d + c;
      System.out.println (d); // Ausgabe der int-Zahl 49
   }
}
```

Die Ausgabe des Programms ist:

```
3
1
0
49
```

In Java repräsentiert ein Zeichenliteral immer genau ein Zeichen. Zeichenkonstanten dürfen das Zeichen `'` sowie Zeilentrenner nicht enthalten. Mit Hilfe von **Ersatzdarstellungen** kann man auch nicht darstellbare Zeichen aufschreiben. So entspricht `\n` einem Zeilentrenner. Ein Zeilentrenner ist ein auf dem Papier unsichtbares Zeichen (white space). `\n` sorgt dafür, dass die Ausgabe am linken Rand fortgesetzt wird. Das `n` in `\n` kommt von **n**ew line = Zeilenende-Zeichen[83].

Ersatzdarstellungen in Zeichenkonstanten und konstanten Zeichenketten

Ersatzdarstellungen – auch Fluchtzeichenfolgen (engl. Escape-Sequences) genannt – wie `\n` können in Zeichenkonstanten und in konstanten Zeichenketten verwendet werden.

Ersatzdarstellungen werden stets mit Hilfe eines Backslashs `\` (Gegenschrägstrich) konstruiert. Mit solchen Ersatzdarstellungen kann man Steuerzeichen oder Zeichen, die auf dem Eingabegerät nicht vorhanden oder nur schwer zu erhalten sind, darstellen. Die folgende Tabelle zeigt Ersatzdarstellungen für Zeichenkonstanten und konstante Zeichenketten:

	Bezeichnung	ASCII-Zeichensatz		Erläuterung
		Char.	Dez.	
`\n`	Zeilentrenner (New Line bzw. Line Feed)	NL (LF)	10	
`\t`	Tabulatorzeichen (Horizontal Tabulator)	HT	9	
`\b`	Backspace	BS	8	
`\r`	Wagenrücklauf (Carriage Return)	CR	13	
`\f`	Seitenvorschub (Form Feed)	FF	12	
`\\`	Gegenschrägstrich (Backslash)	\	92	
`\'`	Anführungszeichen (Einfaches Hochkomma)	'	39	wird gebraucht für das Zeichen einfaches Hochkomma
`\"`	Doppelanführungszeichen (Doppeltes Hochkomma)	"	34	wird gebraucht für das Zeichen doppeltes Hochkomma in Zeichenketten
$\backslash o_1 o_2 o_3$	oktale Zahl			o1, o2, o3 aus {0,...,7}

Tabelle 5-5 Ersatzdarstellungen

Die Ersatzdarstellungen in Tabelle 5-5 werden zwar als **zwei Zeichen oder mehr im Programmcode** hingeschrieben, werden aber **vom Compiler wie ein einziges Zeichen behandelt**.

[83] Je nach Betriebssystem unterscheiden sich die erforderlichen Zeichenkonstanten, um einen Zeilenumbruch (New Line) plus einen Wagenrücklauf (Carriage Return) durchzuführen. Bei Betriebssystemen basierend auf Windows wird die Kombination `\r\n` benötigt, während bei Betriebssystemen basierend auf Linux lediglich `\n` verwendet werden muss. Bei dem Betriebssystem iOS hingegen muss `\r` verwendet werden.

Das **erste Zeichen** muss immer ein **Backslash** sein. Das **zweite** bzw. die weiteren Zeichen legen die **Bedeutung** der Ersatzdarstellung fest.

Die Ersatzdarstellung `\'` stellt ein einfaches Hochkomma dar und die Ersatzdarstellung $\backslash o_1 o_2 o_3$ besteht aus einem Gegenschrägstrich `\` gefolgt von 1, 2 oder 3 Oktalziffern, die als Wert des gewünschten Zeichens interpretiert werden. Auf diese Art kann eine Zeichenkonstante direkt über ihre oktale Zahlendarstellung angegeben werden. Werden drei Oktalziffern angegeben, so kann die erste Ziffer nur eine 0, 1, 2 oder 3 sein. Damit ist 255 die größte durch eine Fluchtzeichenfolge darstellbare Zahl.

Unicode-Ersatzdarstellungen

Die Unicode-Ersatzdarstellung $\backslash u x_1 x_2 x_3 x_4$ – bestehend aus einem Gegenschrägstrich `\` gefolgt von dem ASCII-Zeichen `u` und 4 hexadezimalen Ziffern – wird als Wert des gewünschten Unicode-Zeichens interpretiert.

Solche Unicode-Ersatzdarstellungen sind nicht beschränkt auf Zeichenkonstanten oder konstante Zeichenketten. Sie können auch in Namen (Bezeichnern) auftreten.

Die Ersatzdarstellung $\backslash u x_1 x_2 x_3 x_4$ ermöglicht es, dass jedes beliebige Java-Programm vollständig durch ASCII-Zeichen geschrieben werden kann.

5.3.5.5 Konstante Zeichenketten

Konstante Zeichenketten (**String-Konstanten**, **String-Literale**) sind Folgen von Zeichen, die in doppelten Anführungszeichen eingeschlossen sind.

Die **Doppelanführungszeichen** sind nicht Teil der Zeichenketten, sondern **begrenzen** sie nur.

Beispiele für konstante Zeichenketten sind etwa `"Max"` oder `"Moritz"`.

In Java sind konstante Zeichenketten Instanzen der Klasse `String`.

Wird also eine konstante Zeichenkette in Programmen angegeben, so wird bei der Abarbeitung des Programms ein Objekt der Klasse `String` erzeugt, das mit den Zeichen der konstanten Zeichenkette initialisiert wird. Innerhalb einer Zeichenkette dürfen Zeichen sowie Ersatzdarstellungen stehen.

Mit Hilfe des **Verkettungsoperators** (**Konkatenationsoperators**) + kann man mehrere Zeichenketten verketten. So hat der Ausdruck

```
"Max & " + "Moritz"
```

die gleiche Wirkung wie

```
"Max & Moritz"
```

Ist eine Zeichenkettenkonstante länger als eine Zeile, so stellt man sie als Summe ihrer Bestandteile verknüpft durch den Verkettungsoperator `+` dar und stellt jeden Bestandteil in eine eigene Zeile.

In Zeichenketten sollten **bei oktalen Ersatzdarstellungen stets drei Ziffern** angegeben werden, z. B. `\033` statt `\33`. Dadurch wird vermieden, dass eine eventuell nachfolgende Ziffer zur Ersatzdarstellung gerechnet wird.

5.3.5.6 Nullkonstante

Die **Nullkonstante** (**Nullreferenz**) `null` ist vom Typ `null`.

`null` ist – wie bereits erwähnt – eine Referenz und kann jeder Referenzvariablen zugewiesen werden. Es ist nicht möglich, `null` in einen primitiven Typ[84] (wie zum Beispiel in den Datentyp `int` für ganze Zahlen) zu wandeln.

`null` wird meist dafür verwendet, um auszudrücken, dass eine Referenzvariable auf kein Objekt verweist.

Wie noch behandelt wird (siehe Kapitel 10.4.1), initialisiert Java automatisch die Datenfelder von Objekten und Klassen mit Default-Werten. Handelt es sich bei den Datenfeldern um Referenzen, so werden sie mit der `null`-Referenz initialisiert. Da `null` nicht auf ein Objekt verweist, wird eine Exception vom Typ `NullPointerException` geworfen, wenn versucht wird, über `null` auf eine Komponente wie eine Methode eines Objekts zuzugreifen.

5.3.6 Satzzeichen

Ein **Satzzeichen** (**Interpunktionszeichen**) ist ein Zeichen, das keine Operation spezifiziert. Es hat eine unabhängige syntaktische und semantische Bedeutung.

Dasselbe Symbol kann auch als Operator oder Teil eines Operators vorkommen. In Java gibt es die folgenden Satzzeichen:

---

[84] **Primitive Typen** enthalten einen einfachen Wert, sie sind nicht aus mehreren Werten zusammengesetzt.

```
[]   ()   {}   ;   ,   .
```

Tabelle 5-6 Satzzeichen der Sprache Java

Die Satzzeichen `[]`, `()` und `{}` treten dabei stets in Paaren auf. So werden beispielsweise die eckigen Klammern für die Definition der Größe eines Arrays benötigt. Die runden Klammern werden beispielsweise für das Aufnehmen einer Bedingung in einer `if`-Anweisung gebraucht. Die geschweiften Klammern werden als Blockbegrenzer und für Initialisierungslisten eingesetzt. Das Komma dient als Trenner von Listenelementen, so etwa bei der Deklaration der Liste der formalen Parameter von Methoden oder bei der Initialisierungsliste von Arrays. Der Strichpunkt hingegen wird beispielsweise als Ende einer Anweisung und der Punkt z. B. zur Trennung eines Paketnamens von einem Unterpaketnamen bzw. einem Klassen- oder Schnittstellennamen benötigt. Satzzeichen wirken als Trenner.

5.3.7 Operatoren

Operatoren werden auf Operanden angewandt, um Operationen durchzuführen.

Operanden können beispielsweise Konstanten, Variablen oder auch komplizierte Ausdrücke sein. Durch Operationen werden in der Regel Werte gebildet, aber auch sogenannte Nebeneffekte (siehe Kapitel 7.3) durchgeführt. In Java gibt es die folgenden Operatoren:

```
=    >    <    !    ~    ?    :
==   >=   <=   !=   &&   ||   ++   --   ()   []   .
+    -    *    /    &    |    ^    %    <<   >>   >>>
+=   -=   *=   /=   &=   |=   ^=   %=   <<=  >>=  >>>=
(type) instanceof new
```

Tabelle 5-7 Operatoren der Programmiersprache Java

Operatoren wirken als Trenner. Operatoren werden detailliert in Kapitel 7 besprochen.

5.4 Zusammenfassung

Nach der Darstellung des Zeichenvorrats von Java, d. h. der in Java verwendbaren Zeichen, in Kapitel **5.1** wird in Kapitel **5.2** auf die Repräsentation der Zeichen durch Java als Unicode-Zeichen eingegangen.

Kapitel **5.3** betrachtet die lexikalischen Einheiten von Java. Lexikalische Einheiten sind die Wörter einer Sprache. Lexikalische Einheiten in Java sind: Namen, Schlüsselwörter (reservierte Wörter), Konstanten, Satzzeichen (Interpunktionszeichen) und Operatoren.

Ein Name (Bezeichner) besteht aus einer beliebig langen Zeichenfolge aus Buchstaben und Ziffern, die mit einem Buchstaben beginnt. In Java zählen auch der Unterstrich _ und das `$`-Zeichen sowie andere Währungssymbole zu den Buchstaben.

Schlüsselwörter bzw. reservierte Wörter sind in Java reserviert. Sie müssen stets klein geschrieben werden. Die Bedeutung dieser Schlüsselwörter ist festgelegt und kann nicht verändert werden.

Symbolische Konstanten haben einen Namen, der ihren Wert repräsentiert. Symbolische Konstanten sind Variablen, die nach ihrer Initialisierung nicht verändert werden dürfen.

Literale Konstanten – oft auch nur Konstanten oder Literale genannt – haben keinen Namen, sie werden durch ihren Wert dargestellt

Überall da, wo von der Syntax her Konstanten oder konstante Ausdrücke erlaubt sind, kann man literale Konstanten oder symbolische Konstanten einsetzen. Überall da, wo von der Syntax her an einer Stelle im Programm Konstanten stehen können, können auch konstante Ausdrücke stehen. Dies liegt daran, dass ein konstanter Ausdruck ein Ausdruck ist, an dem nur Konstanten beteiligt sind.

Ein Satzzeichen (Interpunktionszeichen) ist ein Zeichen, das keine Operation spezifiziert. Es hat eine unabhängige syntaktische und semantische Bedeutung. Dasselbe Symbol kann auch als Operator oder Teil eines Operators vorkommen.

Operatoren werden auf Operanden angewandt, um Operationen durchzuführen.

5.5 Übungen

Aufgabe 5.1: Zeichensatz von Java

5.1.1 Auf welchem Zeichensatz basiert Java?

5.1.2 Welche Zeichen verbergen sich hinter folgenden Werten des Unicodes?
- a) dezimal 65
- b) dezimal 122
- c) dezimal 33

5.1.3 Wie kann ein Zeichen eingegeben werden, welches nicht auf der Tastatur vorhanden ist?

5.1.4 Wie können in Java die folgenden Zeichen in einem String dargestellt werden?
- a) `<RETURN>`-Steuerzeichen zum Sprung des Cursors in eine neue Zeile
- b) `<TAB>`-Steuerzeichen zur Positionierung des Cursors auf die nächste Tabulatorposition
- c) doppeltes Hochkomma, d. h. das Zeichen `"`
- d) Gegenschrägstrich (Backslash), d. h. das Zeichen `\`
- e) eine oktale Zahl

Aufgabe 5.2: Lexikalische Einheiten

5.2.1 Nennen Sie verschiedene Arten von literalen Konstanten und jeweils ein Beispiel.

5.2.2 Was sind symbolische Konstanten?

5.2.3 Nennen Sie alle möglichen Satzzeichen und erklären Sie kurz, was ein Satzzeichen ist.
5.2.4 Nennen Sie die zwei literalen Konstanten für Wahrheitsausdrücke.
5.2.5 Nennen Sie fünf Schlüsselwörter (reservierte Wörter).
5.2.6 Welche drei Arten von Kommentaren gibt es in Java?
5.2.7 In Kapitel 4.5 wurde das Programm Kreiseck vorgestellt. Alle Klassen dieses Programms sollen in dieser Aufgabe mit **Dokumentationskommentaren** versehen werden. Was ein Dokumentationskommentar ist, wurde in Kapitel 5.3.1.2 anhand eines Beispiels erläutert.

Nachdem Sie die einzelnen Klassen und ihre Methoden dokumentiert haben, erzeugen Sie mit Hilfe des Werkzeugs `javadoc` eine HTML-Hilfe. Der Aufruf sollte wie folgt aussehen:

```
javadoc Punkt5.java Eck.java Kreis.java Kreiseck.java
        KreiseckTest.java
```

Kapitel 6

Datentypen und Variablen

6.1 Klassifikation der Datentypen von Java
6.2 Einfache Datentypen
6.3 Der Typ einer Klasse
6.4 Variablen
6.5 Array-Typ
6.6 Aufzählungstyp
6.7 Zeichenketten
6.8 Wandlung von Datentypen
6.9 Annotationen
6.10 Zusammenfassung
6.11 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_6

6 Datentypen und Variablen

Datentypen stellen den **Bauplan für Variablen** dar. Alle Variablen eines Datentyps haben dieselbe Darstellung im Arbeitsspeicher, d. h. dieselbe Anzahl von Speicherzellen und dieselbe Interpretation der einzelnen Bits. Verschiedene Variablen können dabei individuelle Werte tragen.

In Java muss der Datentyp einer Variablen, d. h. einer benannten Speicherstelle, zur Übersetzungszeit bekannt sein. Namen und Typ einer Variablen werden in Java durch eine Vereinbarungsanweisung (engl. declaration) dem Compiler bekannt gegeben. Man sagt: **Java** ist **statisch typisiert**.

Bei einer dynamisch typisierten Sprache kann sich der Typ einer Variablen in Abhängigkeit vom Inhalt dieser Variablen zur Laufzeit ändern. Der Typ wird hier zur Laufzeit überprüft.

Kapitel 6.1 erläutert die Klassifikation von Datentypen in Java. Kapitel 6.2 stellt die einfachen Datentypen vor. Anschließend wird in Kapitel 6.3 der Typ einer Klasse erörtert, während Kapitel 6.4 statische und dynamische Variablen untersucht. Array-Typen werden in Kapitel 6.5 behandelt. Kapitel 6.6 diskutiert Aufzählungstypen und Kapitel 6.7 Zeichenketten. Abschließend wird in Kapitel 6.8 auf die Wandlung von Datentypen eingegangen sowie in Kapitel 6.9 auf Annotationen.

6.1 Klassifikation der Datentypen von Java

Zuerst eine Übersicht über die Datentypen von Java:

Bild 6-1 Klassifikation der Datentypen[85]

[85] Der `null`-Typ wird hier nicht betrachtet.

Die **Datentypen** in Java können, wie in Bild 6-1 dargestellt, klassifiziert werden. Es wird generell zwischen **konkreten** und **generischen Datentypen** unterschieden. Generische Datentypen werden an dieser Stelle nur der Vollständigkeit halber erwähnt. Generische Datentypen gibt es in Java erst seit der Version Java 5. In diesem Buch stehen zunächst die konkreten Datentypen im Vordergrund. Auf generische Datentypen – und damit auch auf den Unterschied zu den konkreten Datentypen – wird erst in Kapitel 17 eingegangen.

Bei den **konkreten Datentypen** unterscheidet Java zwischen:

- **einfachen** (**elementaren**, **primitiven**) **Datentypen**
- und **Referenztypen**.

Einfache Datentypen sind in Java durch die Sprache vorgegeben. Selbst definierte einfache Datentypen gibt es in Java nicht.

In Java gibt es die folgenden **einfachen Datentypen**:

- die **ganzzahligen Typen** `byte`, `short`, `int`, `long`, `char`,
- die **Gleitpunkttypen** `float` und `double`
- und den **logischen Typ** `boolean`.

Das Schlüsselwort **`void`** stellt keinen Datentyp dar, sondern ist lediglich eine Kennzeichnung für Methoden, die keinen Rückgabewert haben.

Während die einfachen Datentypen in Java durch die Sprache vorgegeben werden, sind die **Referenztypen** in Java in der Regel sogenannte selbst definierte Datentypen. Eine Ausnahme bildet hierbei nur der in Bild 6-1 nicht erwähnte **`null`-Typ**, der auch ein Referenztyp ist, aber nur die **`null`-Referenz** als Wert zulässt. Die `null`-Referenz wird durch die **Nullkonstante** `null` repräsentiert.

Referenztypen sind in Java – bis auf den Typ `null` – selbst definierte Datentypen. Sie können vom Programmierer **selbst geschrieben** sein oder in **Bibliotheken** enthalten sein.

Auf generische Datentypen wird – wie bereits erwähnt – erst in Kapitel 17 eingegangen. Bei den Referenztypen können Klassen-Typ, Schnittstellen-Typ, Aufzählungstyp und Array-Typ unterschieden werden.[86] Nach der Erläuterung der einfachen Datentypen in Kapitel 6.2 erfolgt in Kapitel 6.3 eine erste Einführung in die Definition von Klassen-Typen. Schnittstellen-Typen werden erst in Kapitel 14 behandelt. Der Array-Typ wird in Kapitel 6.5 und der Aufzählungstyp in Kapitel 6.6 erläutert.

[86] Der `null`-Typ wird hier nicht betrachtet.

6.2 Einfache Datentypen

Tabelle 6-1 fasst die einfachen Datentypen von Java zusammen und gibt den zulässigen Wertebereich für jeden Typ an:

Typ	Inhalt	Wertebereich
`boolean`	`true` oder `false`	`true` und `false`
`char`	**UTF16 Zeichen**	0 bis 65535
`byte`	**8-Bit-Ganzzahl** mit Vorzeichen	-2^7 bis $+2^7-1$
`short`	**16-Bit-Ganzzahl** mit Vorzeichen	-2^{15} bis $+2^{15}-1$
`int`	**32-Bit-Ganzzahl** mit Vorzeichen	-2^{31} bis $+2^{31}-1$
`long`	**64-Bit-Ganzzahl** mit Vorzeichen	-2^{63} bis $+2^{63}-1$
`float`	**Gleitpunkttyp** mit Vorzeichen	$-3.4*10^{38}$ bis $+3.4*10^{38}$
`double`	**Gleitpunkttyp** mit Vorzeichen	$-1.7*10^{308}$ bis $+1.7*10^{308}$

Tabelle 6-1 Einfache Datentypen

6.2.1 Der logische Typ `boolean`

Logische Variablen sind in Java vom Typ `boolean`. Der Typ `boolean` hat die beiden Werte `true` und `false`.

Diese Werte stellen Konstanten dar, keine Schlüsselwörter.

6.2.2 Die Integer-Typen `byte`, `short`, `int`, `long`, `char`

In Java werden die Datentypen `byte`, `short`, `int`, `long`, `char` auf allen Rechnern gleich dargestellt.

Die Typen `byte`, `short`, `int` und `long` sind ganze Zahlen in der **Zweierkomplementdarstellung** und umfassen 8, 16, 32 bzw. 64 Bits.

Der Datentyp `char` umfasst 16 Bits. Er hat als einziger ganzzahliger Datentyp kein Vorzeichenbit und dient zur Darstellung von Unicode-Zeichen.

Das höchste Bit einer **Zweierkomplement**-Zahl gibt das Vorzeichen an. Ist es 0, so ist die Zahl positiv, ist es 1, so ist die Zahl negativ.

Zur Erläuterung soll folgendes Beispiel einer Zweierkomplement-Zahl der Größe 1 Byte dienen:

Bitmuster	MSB							LSB
	1	0	1	0	0	1	1	1
Stellen-wertigkeit	-2^7	$+2^6$	$+2^5$	$+2^4$	$+2^3$	$+2^2$	$+2^1$	$+2^0$
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Bild 6-2 Zweierkomplementdarstellung

Beachten Sie, dass Bit 0 das sogenannte **least significant bit** (LSB) ist. Das höchste Bit wird als **most significant bit** (MSB) bezeichnet.

Der Wert des dargestellten Bitmusters errechnet sich aufgrund der Stellenwertigkeit zu:

$$-1*2^7 + 0*2^6 + 1*2^5 + 0*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 1*2^0 =$$
$$-128 + 0 + 32 + 0 + 0 + 4 + 2 + 1 = -89$$

Die dem Betrag nach größte positive Zahl in der Zweierkomplement-Darstellung ist:

$(0111\ 1111)_2 = 64 + 32 + 16 + 8 + 4 + 2 + 1 = 127$

Die dem Betrag nach größte negative Zahl in dieser Darstellung ist:

$(1000\ 0000)_2 = -128$

Die tief gestellte 2 bedeutet, dass es sich bei dieser Zahl um ein Bitmuster, welches bekanntlich die Basis 2 hat, handelt.

Eine andere (äquivalente) Rechenvorschrift zur Berechnung des Wertes negativer Zahlen ist:

Schritt 1: Da das höchste Bit 1 ist, ist die Zahl negativ.
Schritt 2: Invertiere alle Bits.
Schritt 3: Addiere die Zahl 1.
Schritt 4: Berechne die Zahl in der üblichen Binärdarstellung mit den Stellenwerten 2^7 ... 2^0 und füge anschließend das negative Vorzeichen (von Schritt 1) hinzu.

Wendet man diese Rechenvorschrift auf das obige Beispiel an, so erhält man:

Schritt 1: Zahl ist negativ.
Schritt 2: 01011000
Schritt 3: 01011001
Schritt 4: -(26 + 24 + 23 + 1) = - (64 + 16 + 8 + 1) = - 89

6.2.3 Die Gleitpunkttypen `float` und `double`

Gleitpunktzahlen sind das computergeeignete Modell der in der Mathematik vorkommenden reellen Zahlen.

Nach [IEEE 754] wird die folgende interne Darstellung für `float`-Zahlen verwendet:

`float`: 1 Vorzeichenbit (Bit 31)
8 Bits für Exponenten (Bit 23 - 30)
23 Bits für Mantisse (Bit 0 - 22)

Bild 6-3 Darstellung einer `float`-Zahl (IEEE-Format)

Bei `double`-Zahlen hat man nach [IEEE 754]:

- 1 Vorzeichenbit,
- 11 Bits für Exponenten
- und 52 Bits für Mantisse.

Das **Vorzeichenbit** hat für negative Zahlen den Wert 1, sonst den Wert 0.

Der Wertebereich der `float`-Zahlen liegt zwischen -10^{38} und 10^{38}, der von `double`-Zahlen zwischen -10^{308} und 10^{308}. Die Genauigkeit beträgt 7 Stellen bei `float`-Zahlen und 15 Stellen bei `double`-Zahlen.

6.3 Der Typ einer Klasse

Bevor auf die Definition von Klassen in Kapitel 6.3.1 eingegangen wird, soll in Kapitel 6.3.1 das Konzept eines abstrakten Datentyps erläutert werden. Dieses Kapitel kann beim ersten Lesen übersprungen werden.

6.3.1 Abstrakte Datentypen und Klassen

Ein großer Fortschritt in der Geschichte der Programmiersprachen war das **Konzept der Datenabstraktion**. Mit diesem Konzept wurde das Ziel verfolgt, die Einzelheiten der Datendarstellung von den Beschreibungen der Operationen auf den Daten zu trennen, um eine gesteigerte Übertragbarkeit und Wartbarkeit sowie eine höhere Sicherheit zu erreichen. Bei diesem Konzept ist dem Programmierer die Darstellung der Daten verborgen, er kennt nur die Operationen zum Zugriff auf die Daten.

Ein **abstrakter Datentyp** (ADT) entspricht dem **Konzept der Datenabstraktion** in vollem Maße.

Ein abstrakter Datentyp basiert auf der Formulierung abstrakter Operationen auf abstrakt beschriebenen Daten. Die erlaubten Operationen müssen spezifiziert werden, d. h. sie benötigen jeweils eine Signatur und eine Semantik.

Eine Signatur gibt den entsprechenden Operationen einen Namen und definiert die Übergabeparameter. Was die Operation tun soll, wird als die Semantik der Operation bezeichnet. Diese Spezifikation kann auf unterschiedliche Weise erfolgen.

Bertrand Meyer [Mey97] symbolisierte einen abstrakten Datentyp (ADT) durch einen Eisberg, von dem man nur den Teil über Wasser – sprich die Aufrufschnittstellen der Operationen – sieht. "Unter Wasser" und damit im Verborgenen liegen die Repräsentation der Daten und die Implementierung der Operationen. Dies symbolisiert das folgende Bild:

Bild 6-4 Verbergen der Implementierung eines abstrakten Datentyps

Erst das **Klassenkonzept** der objektorientierten Programmiersprachen ermöglichte es, dass Daten und die Operationen, die mit diesen Daten arbeiten, zu einem gemeinsamen **Datentyp** – der Klasse – zusammengefasst werden konnten.

Eine Klasse implementiert einen abstrakten Datentyp in einer speziellen objektorientierten Programmiersprache.

Ein Ersteller eines objektorientierten Programms konzipiert Klassen, die seine Anwendungswelt widerspiegeln.

Ein Programmierer kann im Idealfall nicht direkt auf die Daten eines Objektes zugreifen, sondern nur über die Methoden des Objektes. Zu einer Klasse gehören die Methoden, die beschreiben, was man mit einem Objekt der Klasse tun kann. Dabei kann man nur auf diejenigen Daten zugreifen, für die explizit eine Methode zur Verfügung gestellt wird. Daten, für die es keine Export-Methode gibt, dienen zu internen Berechnungen und bleiben nach außen verborgen.

Beispiel für einen abstrakten Datentyp

Als Beispiel für einen abstrakten Datentyp soll ein **Stack** von Elementen betrachtet werden. Ein Stack von Elementen ist eine lineare Datenstruktur. Man kann ihn sich am besten am Beispiel eines Bücherstapels veranschaulichen. Auf dem Stapel liegt eine bestimmte Anzahl von Büchern übereinander. Mit der Operation `put()`[87] legt man ein weiteres Buch oben auf den Stapel. Mit der Operation `get()`[88] kann man ein Buch oben an der Spitze des Stapels entnehmen. Ein Buch aus der Mitte des Stapels oder von ganz unten herauszuziehen, ist jedoch nicht erlaubt.

Mathematisch formulieren kann man den abstrakten Datentyp eines Stacks durch: STACK[G]. Hierbei kann G ein Element irgendeines beliebigen Typs sein. Die Operation `put()` lässt sich formulieren durch:

`put()`: STACK[G] x G -> STACK[G]

In Worten ausgedrückt bedeutet dies: Die Operation `put()` hat zwei Parameter, den Stack aus Instanzen von G und eine weitere Instanz von G. Als Resultat der Operation (rechts vom Pfeil) resultiert ein neuer Stack. Entsprechendes gilt für `get()`.

Eine Klasse, die den abstrakten Datentyp STACK implementiert, muss die Methoden `put()` und `get()` in ausprogrammierter Form zur Verfügung stellen, genauso wie die Datenstruktur eines Stacks, die beispielsweise durch ein Array oder eine verkettete Liste realisiert wird. Nach außen werden nur die Aufrufschnittstellen der Methoden angeboten. Die Implementierung, d. h. die Datenstruktur und die Methodenrümpfe, sind verborgen, sodass der Aufrufer gar nicht wissen kann, ob der Stack als Array oder verkettete Liste implementiert ist.

6.3.2 Definition einer Klasse

Wie in Kapitel 6.3.1 erläutert, implementiert eine Klasse einen **abstrakten Datentyp**. Die Realisierung eines abstrakten Datentyps beschreibt man in einer **Klassendefinition**.

Eine Klassendefinition gibt den Namen eines neuen Datentyps bekannt und definiert zugleich dessen Methoden und Datenfelder.

Eine Klassendefinition umfasst die Deklaration der Klasse, d. h. des Namens der Klasse, plus den Rumpf der Klasse.

Das Schlüsselwort zur Deklaration einer neuen Klasse ist **`class`**. Auf das Schlüsselwort `class` folgt der Klassenname. Der **Klassenname** stellt den Namen für den neuen Datentyp dar. Er muss ein gültiger Java-Namen sein. Er sollte – wie in der Java-Welt durch Konventionen allgemein üblich – mit einem **Großbuchstaben** beginnen.

---

[87] Statt `put()` wird oft der Name `push()` verwendet.
[88] Wird `push()` verwendet, so tritt `pop()` an die Stelle von `get()`.

Hier ein **Beispiel**:

```
class Punkt       // Deklaration des neuen Klassennamens Punkt
{                 // Der Klassenrumpf enthält
   . . . . .      // Datenfelder
}                 // und Methoden
```

Eine **Deklaration** gibt dem Compiler einen neuen Namen bekannt. Die **Definition** einer Klasse umfasst die Deklaration des Klassennamens plus die Festlegung der Datenfelder und Methoden innerhalb der geschweiften Klammern des **Klassenrumpfes**.

6.3.2.1 Methoden

Eine Methode ist eine Anweisungsfolge, die unter einem Namen abgelegt ist und über diesen aufrufbar ist.

Eine Methode muss in einer Klasse definiert werden. Eine Methode besteht aus der **Methodendeklaration** und dem **Methodenrumpf**. Die Methodendeklaration gibt dem Compiler die Aufrufschnittstelle der Methode bekannt. Die Methodendeklaration wird auch als **Methodenkopf** bezeichnet. Hier die Definition einer Methode als Pseudocode:

```
Methodendeklaration   // Methodenkopf
{                     //
                      // Methodenrumpf
}                     //
```

Die Methodendeklaration beinhaltet im Minimalfall den Namen der Methode, dahinter eine öffnende und eine schließende runde Klammer und vor dem Methodennamen den Rückgabetyp der Methode oder das Schlüsselwort `void`.

Im folgenden Beispiel wird die Definition der Klasse `Punkt` betrachtet:

```
public class Punkt
{                     // Mit der öffnenden geschweiften Klammer
                      // beginnt der Klassenrumpf

   private int x;     // x-Koordinate vom Typ int

   public int getX()  // getX ist der Name der Methode. Die runden
                      // Klammern ohne Inhalt besagen, dass die
                      // Methode ohne Übergabeparameter aufgerufen
                      // wird. Das vor den Methodennamen gestellte
                      // int bedeutet, dass die Methode an der Stelle
                      // ihres Aufrufs einen int-Wert zurückliefert.

   {                  // Der Methodenrumpf beginnt mit einer öffnen-
                      // den geschweiften Klammer.
                      // Zwischen den geschweiften Klammern, die den
```

```
                        // Beginn und das Ende des Methodenrumpfes
                        // bilden, stehen die Anweisungen der Methode.

      return x;         // Die einzige Anweisung hier ist: return x.
                        // return x gibt an den Aufrufer der Methode
                        // den Wert des Datenfeldes x zurück.

   }                    // Der Methodenrumpf endet mit der
                        // schließenden geschweiften Klammer.

    . . . . .           // Die weiteren Methoden dieser Klasse werden
                        // hier nicht betrachtet.

}                       // Mit der schließenden geschweiften Klammer
                        // endet der Klassenrumpf.
```

Die **Methoden eines Objektes** haben direkten **Zugriff auf** die **Datenfelder und Methoden desselben Objektes**.

Das folgende Beispiel zeigt den Zugriff einer Methode eines Objektes auf eine Methode desselben Objektes, nämlich den Zugriff der Methode `print()` auf die Methode `printSterne()`, und den Zugriff einer Methode eines Objektes auf ein Datenfeld desselben Objektes, nämlich den Zugriff auf das Datenfeld `x` durch die Methoden `getX()` und `setX()`[89]:

```
// Datei: Punkt6.java
public class Punkt6

{
   private int x;           // x-Koordinate vom Typ int

   public int getX()
   {                        // Zugriff der Methode getX() auf das
      return x;             // Datenfeld x desselben Objektes.
   }

   public void setX (int i)// Eine Methode, um den x-Wert zu setzen.
   {
      x = i;                // Zugriff der Methode setX() auf das
                            // Datenfeld x desselben Objektes.
   }

   public void printSterne()
   {
      System.out.println ("**********************************");
   }

   public void print()
   {
      printSterne();        // Zugriff der Methode print() auf die
                            // Methode printSterne() desselben
                            // Objektes.
```

[89] Werden Methoden im Text zitiert, so können die runden Klammern `()` nach dem Methodennamen leer bleiben, obwohl die Methode Parameter hat.

```
        System.out.println ("Die Koordinate des Punktes ist: " + x);
        printSterne();
    }
}

// Datei: Punkt6Test.java

public class Punkt6Test
{
    // mit main() beginnt eine Java-Anwendung ihre Ausführung.
    public static void main (String[] args)
    {
        Punkt6 p = new Punkt6(); // Hiermit wird ein Punkt erzeugt.
        p.setX (3);              // Setzen der x-Koordinate auf 3.
        p.print();               // Aufruf der Methode print().
    }
}
```

Die Ausgabe des Programmes ist:

```
***********************************
Die Koordinate des Punktes ist: 3
***********************************
```

In Java werden konventionsgemäß die **Namen von Methoden** klein geschrieben. Bei zusammengesetzten Namen beginnt jedes Wort bis auf das erste mit einem Großbuchstaben[90]. Wird das Schlüsselwort `void` anstelle des Rückgabetyps angegeben, so gibt die Methode nichts zurück und deshalb ist kein `return` notwendig[91], ansonsten muss immer ein Wert mit Hilfe einer `return`-Anweisung zurückgegeben werden.

Klassen-Typen sind Referenztypen (siehe Bild 6-1).

Eine **Variable einer Klasse** ist eine Referenz auf ein Objekt dieser Klasse.

Referenztypen haben als Variablen Referenzen auf Objekte.

Von einem fremden Objekt aus wird eine **Methode ohne Parameter** eines anderen Objektes aufgerufen, indem das fremde Objekt auf eine **Referenz auf das andere Objekt** den **Punktoperator** anwendet und den **Methodennamen** gefolgt von einem **leeren Klammerpaar** () angibt wie etwa `ref.meth()`.

Die Methode `print()` eines Objekts wird über die Referenzvariable `p`, die auf dieses Objekt zeigt, folgendermaßen aufgerufen:

```
p.print();
```

[90] CamelCase-Notation (siehe Kapitel 5.3.3)

[91] Eine `return`-Anweisung ist nicht erforderlich, aber möglich. Die `return`-Anweisung gibt hier aber keinen Wert zurück, sondern bedeutet nur einen Rücksprung (siehe Kapitel 9.2.3).

Die Notation `p.print()` bedeutet, dass die Methode `print()` des Objektes, auf das die Referenz `p` zeigt, aufgerufen wird.

Von einem fremden Objekt aus wird eine **Methode mit Parametern**[92] eines anderen Objektes aufgerufen, indem das fremde Objekt auf eine **Referenz auf das andere Objekt** den **Punktoperator** anwendet und den **Methodennamen** gefolgt von den **benötigten aktuellen Parametern der Methode in runden Klammern** () angibt wie etwa `ref.meth (aktPar)`.

Jede Operation auf einer Referenz erfolgt tatsächlich auf dem referenzierten Objekt.

6.3.2.2 Datenfelder

Die Datenfelder (Variablen) einer Klasse werden im Klassenrumpf definiert. Die Definition kann an jeder Stelle des Klassenrumpfes erfolgen. Die Vereinbarung einer Variablen im Klassenrumpf erfolgt durch:

```
datentyp name;
```

In Java werden konventionsgemäß die **Namen von Variablen** klein geschrieben. Bei zusammengesetzten Namen beginnt jedes Wort bis auf das erste mit einem Großbuchstaben (CamelCase-Notation, siehe auch Kapitel 5.3.3).

Der Zugriff auf ein Datenfeld eines fremden Objektes erfolgt ebenfalls mit der Punktnotation wie der Zugriff auf Methoden. So sei `p` eine Referenz auf ein Objekt der Klasse `Punkt`. Der Zugriff auf das Datenfeld `x` des Objektes, auf das die Referenz `p` zeigt, erfolgt dann mit[93]:

```
p.x;
```

6.4 Variablen

Prinzipiell unterscheidet man bei Programmiersprachen zwischen **statischen**[94] und **dynamischen** Variablen. Im Folgenden wird sowohl eine **statische** als auch eine **dynamische** Variable in Java gezeigt:

```
// Datei: Punkt7.java

public class Punkt7            // Deklaration der Klasse Punkt7
{
```

[92] Auf die Parameter von Methoden wird ausführlich in Kapitel 9.2 eingegangen.
[93] Unter der Voraussetzung, dass der Zugriff erlaubt ist (siehe Kapitel 12.7).
[94] Statisch im Sinne des Unterschieds zwischen statisch und dynamisch hat überhaupt nichts mit den `static`-Variablen (Klassenvariablen) von Java zu tun. In diesem Kapitel kommen statische Variablen nur in ihrer allgemeinen Bedeutung als Gegensatz zu dynamischen Variablen vor.

```
   private int x;              // Datenfeld für die x-Koordinate
                               // vom Typ int

   public int getX()           // eine Methode, um den Wert
   {                           // von x abzuholen
       return x;
   }

   public void setX (int i)    // eine Methode, um den Wert
   {                           // von x zu setzen
      x = i;
   }
}

// Datei: TestPunkt7.java
public class TestPunkt7
{
   public static void main (String[] args)
   {
      int x = 3;               // x ist eine statische Variable
                               // eines einfachen Datentyps
      Punkt7 p;                // Die Referenzvariable p ist
                               // eine statische Variable

      p = new Punkt7();        // Erzeugen einer dynamischen
                               // Variablen mit dem new-Operator
      p.setX (x);              // Aufruf der Methode setX()
      System.out.println ("Die Koordinate des Punktes, auf den p " +
         "zeigt ist: ");
      System.out.println (p.getX());
   }
}
```

Die Ausgabe des Programmes ist:

```
Die Koordinate des Punktes, auf den p zeigt ist:
3
```

Im obigen Beispiel stellen die benannten **lokalen Variablen**[95] `x` und `p` **statische Variablen dar**. Mit `new Punkt7()` wird auf dem Heap[96] ein **namenloses Objekt** der Klasse `Punkt7` als **dynamische Variable** mit Hilfe des `new`-Operators und des Default-Konstruktors erzeugt. Der Rückgabewert bei der Erzeugung ist eine Referenz auf die angelegte dynamische Variable auf dem Heap. Die zurückgegebene Referenz wird der statischen **Referenzvariablen** mit dem Namen `p` zugewiesen. Die statische Referenzvariable `p` stellt im obigen Beispiel die einzige Möglichkeit dar, auf das namenslose Objekt vom Typ `Punkt7` zuzugreifen.

Eine Variable, die in einer Methode definiert wird, ist sowohl eine lokale Variable als auch eine statische Variable.

[95] **Lokale Variablen** sind Variablen, die innerhalb von Methoden definiert werden.

[96] Der **Heap** ist ein von der Java-Virtuellen Maschine verwalteter Speicherbereich, in welchem die mit dem `new`-Operator dynamisch erzeugten Objekte abgelegt werden (siehe Kapitel 6.4.5.2).

Eine statische Variable hat immer einen Typ und einen Namen (Bezeichner).

Bei der Definition einer statischen Variablen – hier wird die Variable angelegt – muss der Typ und der **Variablenname** wie in folgendem Beispiel angegeben werden:

```
int x;
```

Nach ihrer Definition kann auf eine statische Variable über ihren Namen zugegriffen werden. Eine solche Variable heißt **statisch**, weil ihr **Gültigkeitsbereich und** ihre **Lebensdauer durch die statische Struktur des Programms** festgelegt ist. Der **Gültigkeitsbereich** einer statischen Variablen umfasst alle Stellen im Programm, an denen ihr Name durch die Vereinbarung bekannt ist.

Die **Lebensdauer** einer lokalen, statischen Variablen erstreckt sich über den Zeitraum der **Abarbeitung der Methode bzw. des Blocks**[97], zu dem sie gehört. Das heißt, während dieser Zeit ist für sie Speicherplatz vorhanden.

Die **Gültigkeit und Lebensdauer** einer **dynamischen Variablen** wird **nicht durch die statische Struktur des Programms**, wie z. B. die Blockgrenzen, **bestimmt**.

Dynamische Variablen erscheinen nicht explizit in einer Definition. Sie tragen keinen Namen. Daher kann auf sie nicht über einen Bezeichner zugegriffen werden. Dynamische Variablen werden in der Regel mit dem Operator `new` im Heap angelegt[98]. Der Zugriff auf dynamische Variablen erfolgt mit Hilfe der **Referenz** auf das neu erzeugte Objekt, welches der `new`-Operator zurückgibt.

Statische Variablen sind entweder Variablen einfacher Datentypen oder Referenzvariablen. Dynamische Variablen sind in Java immer Objekte.

Statische Variablen können sein:

- lokale Variablen,
- Instanzvariablen
- und Klassenvariablen.

Auf statische Variablen wird in Kapitel 6.4.4 eingegangen.

---

[97] Ein **Block** (siehe Kapitel 9) stellt eine zusammengesetzte Anweisung dar. Als Blockbegrenzer dienen die geschweiften Klammern. In jedem Block können Variablen definiert werden.

[98] Die andere Form der Erzeugung mit `new Instance ()` soll hier nicht betrachtet werden. Siehe Kapitel 17.5.1

6.4.1 Variablen einfacher Datentypen

Von einfachen Datentypen kann man eine Variable erzeugen, die einen einfachen Wert in der ihr zugeteilten Speicherstelle aufnehmen kann. So enthält eine Variable vom Typ `int` genau einen `int`-Wert wie z. B. die Zahl 3.

Eine Definition einer Variablen

- legt den **Namen** und die **Art einer Variablen fest**
  – nämlich ihren **Typ**
  – sowie **Modifikatoren** wie `public`, `static` etc.
- und sorgt gleichzeitig für die **Reservierung des Speicherplatzes** für die Variable.

Mit einer **Definition** ist stets auch eine **Deklaration** verbunden.

Die **Deklaration** einer Variablen umfasst den Namen einer Variablen, ihren Typ und ggf. ihren **Typmodifikator**. Mit der **Deklaration** wird **dem Compiler bekanntgegeben**, mit welchem **Typ** und mit welchem **Typmodifikator** er einen **Namen** verbinden muss.

Kurz und bündig ausgedrückt, bedeutet dies:

Definition = Deklaration + Reservierung des Speicherplatzes.

In Java ist es nicht möglich, Variablen nur zu deklarieren und sie an anderer Stelle zu definieren, wohl aber in der Programmiersprache C.

Eine einzige Variable eines einfachen Datentyps wird definiert zu

```
datentyp name;
```

Hier ein **Beispiel**:

```
int x;
```

Mehrere Variablen vom selben Typ können i**n einer einzigen Vereinbarung definiert** werden, indem man wie im folgenden **Beispiel** die Variablennamen durch Kommata trennt:

```
int x, y, z;
```

Die Namen der Variablen müssen den Namenskonventionen (siehe Kapitel 5.3.2) genügen. Ein Variablenname darf nicht identisch mit einem Schlüsselwort sein.

6.4.2 Referenzvariablen

Referenzvariablen ermöglichen den **Zugriff auf Objekte** im Heap. Als Wert enthalten sie die Adresse[99], an der sich das entsprechende Objekt im Heap befindet.

Referenzvariablen zeigen in Java entweder auf:

- Objekte
- oder nichts, wenn sie die `null`-Referenz als Wert enthalten.

Referenzvariablen können vom Typ einer Klasse, aber auch von einem Arraytyp, einem Aufzählungstyp oder einer Schnittstelle sein.[100] Arrays werden in Kapitel 6.5 vorgestellt, Aufzählungstypen in Kapitel 6.6 und Schnittstellen in Kapitel 14.

Eine Referenzvariable kann als Wert enthalten:

- Die Adresse eines Objektes, dessen Typ zuweisungskompatibel[101] zum Typ der Referenzvariablen ist.
- Die **null-Referenz**.

  In Java gibt es – wie bereits erwähnt – den sogenannten **`null`-Typ**. Von diesem Typ gibt es nur einen einzigen Wert, die Konstante `null`. Diese Konstante wird verwendet als `null`-Referenz für Referenzvariablen, die noch auf kein Objekt zeigen. Die Referenz `null` ist eine vordefinierte Referenz, deren Wert sich von allen regulären Referenzen unterscheidet. Wird einer Referenzvariablen, die auf ein gültiges Objekt im Speicher zeigt, die `null`-Referenz zugewiesen, so können keine Methoden und keine Datenfelder über diese Referenzvariable mehr angesprochen werden. Eine `null`-Referenz ist zu allen anderen Referenztypen zuweisungskompatibel, d. h., jeder Referenzvariablen kann die Referenz `null` zugewiesen werden.

Objekte und Referenzvariablen haben einen Datentyp. Will man mit einer Referenzvariablen auf ein Objekt zeigen, so muss die Klasse des Objektes zuweisungskompatibel zum Typ der Referenzvariablen sein. Vereinfacht ausgedrückt bedeutet dies: Ist ein Objekt vom Typ `Klassenname`, so braucht man eine Referenzvariable vom Typ `Klassenname`, um auf dieses Objekt zeigen zu können.

Eine Referenzvariable wird formal wie eine einfache Variable definiert:

```
Klassenname referenzName;
```

Diese Definition wird von rechts nach links gelesen zu: "`referenzName` ist vom Typ `Klassenname` und ist eine Referenz auf ein Objekt der Klasse `Klassenname`".

[99] Es handelt sich hierbei nicht um die physische Adresse im Arbeitsspeicher des Rechners, sondern um eine **logische Adresse**, die von der Java-Virtuellen Maschine in die physische Adresse umgesetzt wird. Dass die Referenz nicht die physische Adresse enthält, hat Sicherheitsgründe.

[100] Der `null`-Typ wird hier nicht betrachtet.

[101] **Zuweisungskompatibilität** bei Klassen wird in Kapitel 11.3.1 erläutert.

Durch diese Definition wird eine Referenzvariable `referenzName` vom Typ `Klassenname` definiert, wobei der Compiler für diese Referenzvariable Platz vorsehen muss. **Beispiele** für die Definition von Referenzvariablen sind:

```
ClassA refA;
ClassB refB;
ClassC blubb; //damit niemand meint, es müsse immer ref heißen
```

Durch die Definition sind Referenz und zugeordneter Typ miteinander verbunden. Durch die Definition einer Referenzvariablen wird noch kein Speicherplatz für ein Objekt vorgesehen, sondern nur für die Referenzvariable. Ebenso wie bei jeder anderen lokalen Variablen ist der Wert einer lokalen Referenzvariablen nach der Variablendefinition zunächst unbestimmt. Diese Variable muss vor ihrer Verwendung initialisiert werden, sonst resultiert ein Kompilierfehler.

Wie bei einfachen Datentypen kann man **mehrere Referenzvariablen vom selben Typ** in einem einzigen Schritt definieren, indem man in der Definition eine Liste von Variablennamen angibt, wobei die verschiedenen Variablennamen durch Kommata voneinander getrennt sind wie im folgenden **Beispiel**:

```
Punkt p1, p2, p3;
```

Eine **Referenzvariable** ist also in Java eine Variable, die eine Verknüpfung zu einem im Speicher befindlichen **Objekt** beinhaltet. Die **Verknüpfung mit dem referenzierten Objekt** erfolgt durch einen **logischen Namen**. Eine Referenzvariable enthält als Variablenwert also einen logischen Namen, der auf das entsprechende Objekt verweist. Dieser logische Name wird von der Java-Virtuellen Maschine in eine physische Adresse umgesetzt. Von Java aus sind also die physischen Adressen des Arbeitsspeichers nicht direkt sichtbar. In Java kann damit die physische Adresse einer Variablen nicht vom Anwender ermittelt werden.

Referenzen gibt es in Java nur auf Objekte, nicht auf Variablen einfacher Datentypen.

Das folgende Bild zeigt eine Referenzvariable und ein Objekt im Arbeitsspeicher:

Bild 6-5 Referenzvariablen können auf Objekte zeigen

Bruce Eckel [Eck00] verwendet für Referenzen ein treffendes Beispiel. Er vergleicht ein Objekt mit einem Fernseher und die **Referenz** auf dieses Objekt mit der **Fernsteuerung**, die auf den Fernseher zugreift. Will man den Fernseher bedienen, so bedient man direkt die Fernsteuerung und damit indirekt den Fernseher.

Während man jedoch bei Fernsehgeräten oftmals auch ohne Fernsteuerung auskommen und den Fernsehapparat direkt einstellen kann, ist dies bei Objekten in Java nicht möglich. Objekte tragen in Java keinen Namen. Werden sie erzeugt, so erhält man eine Referenz auf das entsprechende Objekt. Diese Referenz muss man einer Referenzvariablen zuweisen, um den Zugriff auf das Objekt nicht zu verlieren.

Objekte können in Java nicht direkt manipuliert werden. Sie können nur "ferngesteuert bedient" werden. Mit anderen Worten, **man kann auf Objekte nur indirekt mit Hilfe von Referenzen zugreifen**.

Eine Referenzvariable muss nicht immer auf dasselbe Objekt zeigen. Der Wert einer Referenzvariablen kann durch eine erneute Zuweisung auch verändert werden. Das soll im Folgenden diskutiert werden.

Bei der **Zuweisung**

```
a = b;   // a und b sollen Variablen vom selben Typ sein
```

findet im Falle von einfachen Datentypen ein **Kopieren des Wertes** von `b` in die Variable `a` statt. Sind `a` und `b` Referenzvariablen, so wird der Wert der Referenzvariablen `b` in die Referenzvariable `a` **kopiert**. Nach einer solchen Zuweisung zeigen die Referenzvariablen `a` und `b` auf dasselbe Objekt.

Das folgende **Beispiel** zeigt die Zuweisung des Werts einer Referenzvariablen `p1` an eine andere Referenzvariable `p2`. Nach dieser Zuweisung zeigt die Referenzvariable `p2` auf dasselbe Objekt wie `p1`, um dann nach der Zuweisung `p2 = p3` auf dasselbe Objekt wie `p3` zu zeigen.

Zuerst

```
p2 = p1;
```

Dies wird in folgendem Bild visualisiert:

Bild 6-6 Speicherabbild nach der Zuweisung `p2 = p1`

Und nun:

```
p2 = p3;
```

Die Auswirkung dieser Zuweisung zeigt das folgende Bild:

Bild 6-7 Speicherabbild nach der Zuweisung `p2 = p3`

Und jetzt das bereits erwähnte **Beispiel**:

```
// Datei: Punkt8.java

public class Punkt8
{
   private int x;
   public int getX()
   {
      return x;
   }
   public void setX (int u)
   {
      x = u;
   }
}

// Datei: TestPunkt8.java

public class TestPunkt8
{
   public static void main (String[] args)
   {
      Punkt8 p1 = new Punkt8(); // Anlegen eines Punkt-Objektes
      p1.setX (1);              // Dieses enthält den Wert x = 1
      Punkt8 p2;                // Anlegen einer Referenzvariablen
                                // vom Typ Punkt
      Punkt8 p3 = new Punkt8(); // Anlegen eines Punkt-Objektes
      p3.setX (3);              // x wird 3

      p2 = p1;                  // Nun zeigt p2 auf dasselbe Objekt
                                // wie p1
      System.out.println ("p1.x hat den Wert " + p1.getX());
      System.out.println ("p2.x hat den Wert " + p2.getX());
      System.out.println ("p3.x hat den Wert " + p3.getX());

      p2 = p3;                  // Nun zeigt p2 auf dasselbe
                                // Objekt wie p3
```

```
        System.out.println ("p2.x hat den Wert " + p2.getX());
        p2.setX (20);
        System.out.println ("p2.x hat den Wert " + p2.getX());
        System.out.println ("p3.x hat den Wert " + p3.getX());
    }
}
```

Die Ausgabe des Programms ist:

```
p1.x hat den Wert 1
p2.x hat den Wert 1
p3.x hat den Wert 3
p2.x hat den Wert 3
p2.x hat den Wert 20
p3.x hat den Wert 20
```

6.4.3 Dynamische Variablen – Objekte

Kapitel 6.4.3.1 befasst sich mit der Erzeugung von Objekten und Kapitel 6.4.3.2 mit dem Zugriff auf Objekte über lokale Variablen.

6.4.3.1 Erzeugung von Objekten

Referenzen auf Objekte – die Referenzvariablen – werden in Java als statische Variablen angelegt. Die **Objekte** selbst werden mit Hilfe des **`new`-Operators** als **dynamische Variablen** auf dem **Heap** erzeugt.

Ein **Objekt** wird in Java erzeugt durch die Anweisung:

```
new Klassenname();
```

Ein **Objekt** wird vom Laufzeitsystem der JVM als **dynamische Variable** auf dem **Heap**, der ein Speicherreservoir für dynamische Variablen darstellt, angelegt.

Ist nicht genug Platz zum Anlegen eines Objektes vorhanden, so muss das **Laufzeitsystem** versuchen, über eine Speicherbereinigung (Garbage Collection) Platz zu gewinnen. Schlägt dies fehl, so wird eine Ausnahme (Exception) vom Typ `OutOfMemoryError`[102] ausgelöst.

Dynamische Variablen erscheinen nicht in einer Variablendefinition. Dynamische Variablen tragen keinen Namen. Der Zugriff auf dynamische Variablen erfolgt in Java mit Hilfe von Referenzen, den Referenzvariablen.

[102] Siehe Kapitel 13.4

Die Definition einer Referenzvariablen und die Erzeugung eines Objektes lassen sich in einem einzigen Schritt wie folgt durchführen:

```
Klassenname var = new Klassenname();
```

Dabei gibt der `new`-Operator den Zeiger auf das erzeugte Objekt zurück und dieser Zeiger wird der Referenzvariablen `var` zugewiesen.

Darauf wird in diesem Kapitel noch genauer eingegangen.

Oftmals – wenn es nicht so genau darauf ankommt, oder wenn man mit der Sprache etwas nachlässig ist – verwendet man statt "Referenz auf ein Objekt" auch das Wort "Objekt". Liest man dann an einer Stelle das Wort "Objekt", so muss man aus dem Zusammenhang erschließen, ob das Objekt oder die Referenz auf das Objekt gemeint ist – denn woher soll man wissen, ob sich der Autor gerade "locker" oder präzise ausdrückt.

Bei einer exakten Sprechweise werden "Referenz auf ein Objekt" und "Objekt" unterschieden.

Objekte werden in der Regel mit `new` auf dem Heap angelegt. Es gibt noch einen **zweiten Weg**, **Objekte zu schaffen**. Dies erfolgt mit Hilfe der **Methode `newInstance()` der Klasse `Class<T>`** und wird in Kapitel 17.5.1 erklärt. Im Weiteren soll jedoch die Erzeugung eines Objektes mit dem `new`-Operator am **Beispiel** der Klasse `Punkt` betrachtet werden.

Hier die Klasse `Punkt`:

```
public class Punkt
{
   private int x;             // x-Koordinate vom Typ int
   public static void main (String[] args)
   {
      Punkt p = null;
      p = new Punkt();        // hiermit wird ein Punkt erzeugt
      // weitere Anweisungen
   }
}
```

Die Definition `Punkt p;` erzeugt die Referenzvariable `p`, die auf ein Objekt der Klasse `Punkt` zeigen kann. Mit `new Punkt()` wird ein Objekt ohne Namen auf dem Heap angelegt. `new Punkt()` gibt eine Referenz auf das erzeugte Objekt zurück. Diese Referenz wird der Referenzvariablen `p` zugewiesen. Dies zeigt das folgende Bild:

Bild 6-8 `null`-Referenz und Referenz auf ein Objekt

6.4.3.2 Zugriff auf Objekte über lokale Variablen

In dem Beispiel dieses Kapitels soll nicht auf das Bild 6-8 Bezug genommen werden.

Eine **Referenzvariable als lokale Variable** in einer Methode wird vom Compiler **nicht automatisch initialisiert**.

Wird versucht, mit dem Punktoperator auf eine nicht initialisierte lokale Referenzvariable zuzugreifen, so meldet der Compiler einen Fehler. Dies soll anhand des folgenden **Beispiels** erläutert werden:

```
// Datei: CompilerTest.java

class Punkt
{
   private int x;           // Das Datenfeld x wird vom Compiler mit
                            // dem Default-Wert 0 initialisiert
   public void print()
   {
      System.out.println ("x: " + x);
   }
}

public class CompilerTest
{
   public static void main (String[] args)
   {
      // Anlegen der nicht initialisierten lokalen Variablen p
      Punkt p;

      // Zugriff auf die nicht initialisierte lokale Variable p
      p.print();
   }
}
```

Der Aufruf des Compilers lautet:

```
javac CompilerTest.java
```

Die Ausgabe des Compilers ist:

```
CompilerTest.java:21: variable p might not have
been initialized
      p.print();
      ^
```

Um das obige Beispielprogramm für den Compiler akzeptabel umzuschreiben, wird im folgenden **Beispiel** die lokale Variable `p` vom Typ `Punkt` mit der `null`-Referenz initialisiert, so wie in Bild 6-8 auf der linken Seite.

Hier das Programm:

```
// Datei: CompilerTest2.java
class Punkt
{
   private int x;          // Das Datenfeld x wird vom Compiler
                           // mit dem Default-Wert 0 initialisiert

   public void print()
   {
      System.out.println ("x: " + x);
   }
}

public class CompilerTest2
{
   public static void main (String[] args)
   {
      // Anlegen einer mit null initialisierten lokalen Variablen
      Punkt p = null;

      // Zugriff auf die mit null initialisierte lokale Variable p
      p.print();
   }
}
```

Wird das Programm nach der erfolgreichen Übersetzung gestartet, generiert das Laufzeitsystem jedoch eine Exception[103] vom Typ `NullPointerException`. Dies zeigt die Ausgabe:

Die Ausgabe des Programms ist:

```
Exception in thread "main"
     java.lang.NullPointerException
      at CompilerTest2.main(CompilerTest2.java:21)
```

Das Programm ist also immer noch falsch. Es gibt zwar keinen Kompilierfehler mehr, aber einen Laufzeitfehler. Was ist los? Die Antwort ist klar. Es wurde vergessen, der Variablen `p` eine Referenz auf ein Objekt der Klasse `Punkt` zuzuweisen.

[103] Exceptions werden in Kapitel 13 behandelt.

Beim Zugriff auf eine Methode oder ein Datenfeld einer mit `null` initialisierten Referenzvariablen erzeugt die Laufzeitumgebung eine Exception vom Typ `NullPointerException`. Diese Exception führt zu einem Programmabsturz. Ein Zugriff auf eine mit `null` initialisierte Referenzvariable stellt einen häufigen Programmierfehler dar.

Wird durch

```
p = new Punkt();
```

die Referenzvariable `p` mit einer Referenz auf ein Objekt initialisiert – wie in Bild 6-8 auf der rechten Seite –, so kann der Aufruf

```
p.print();
```

problemlos durchgeführt werden. Mit dieser Korrektur läuft dann das Programm ohne Fehler.

Hier die Ausgabe:

Die Ausgabe des Programms ist:

```
x: 0
```

6.4.4 Klassenvariablen, Instanzvariablen und lokale Variablen

Neben der Unterscheidung zwischen statischen und dynamischen Variablen (den erzeugten Objekten) ist die Unterscheidung der statischen Variablen in lokale Variablen, Instanzvariablen und Klassenvariablen von Bedeutung.

In Java gibt es die folgenden Arten von statischen Variablen:

- **Klassenvariablen**,
- **Instanzvariablen**
- und **lokale Variablen**.

Klassenvariablen werden für jede Klasse ein einziges Mal angelegt. **Instanzvariablen** gibt es für jede Instanz einer Klasse, also für jedes Objekt. **Lokale Variablen** gibt es in Methoden. Die Gültigkeit einer lokalen Variablen kann sich auf den Methodenrumpf oder auf einen inneren Block – zum Beispiel den Block einer `for`-Schleife – erstrecken.

Übergabeparameter sind spezielle lokale Variablen. **Übergabeparameter** gibt es bei **Methoden**, **Konstruktoren** (siehe Kapitel 10.4.4) und bei **`catch`-Blöcken**[104]. Übergabeparameter werden noch ausführlich in Kapitel 9.2.5 behandelt.

[104] `catch`-Blöcke dienen zur Behandlung von Ausnahmen (siehe Kapitel 13.5).

Das folgende **Beispiel** zeigt die Verwendung von Klassenvariablen, Instanzvariablen und lokalen Variablen:

```
// Datei: VariablenTypen.java

public class VariablenTypen
{
   private int x;            // dies ist eine Instanzvariable
   private static int y;     // dies ist eine Klassenvariable

   public void print()
   {
      int z = 0;             // dies ist eine lokale Variable
   }
}
```

In Java werden Klassenvariablen und Instanzvariablen (Datenfelder eines Objektes) automatisch mit einem Default-Wert initialisiert (siehe Kapitel 10.4.1). Lokale Variablen hingegen werden nicht automatisch initialisiert.

Der Compiler prüft, ob lokale Variablen initialisiert wurden. Eine lokale Variable muss dabei entweder **manuell initialisiert** oder mit Hilfe einer Zuweisung vor ihrer Verwendung mit einem Wert belegt werden.

Zunächst ein **Beispiel** für eine manuelle **Initialisierung**:

```
int x = 3;
```

Und nun ein **Beispiel** für eine **Zuweisung** des Werts vor Verwendung der Variablen:

```
int x;
. . . . .
x = 3;
a = x + 2;      // hier wird x verwendet
```

Werden lokale Variablen verwendet, bevor sie initialisiert wurden, so erzeugt der Compiler eine Fehlermeldung.

Lokale Variablen sind **statische Variablen** und werden auf dem **Stack** angelegt. **Instanzvariablen** bilden einen Teil eines Objektes und sind damit Bestandteil einer **dynamischen Variablen**. Sie liegen deshalb auf dem **Heap**. **Klassenvariablen** sind **statische Variablen** und werden in der **Method-Area** abgelegt.

Dabei kann eine **lokale Variable**, eine **Instanzvariable** und eine **Klassenvariable** entweder einen **einfachen Datentyp** haben oder eine **Referenzvariable** darstellen. Die Speicherbereiche für Variablen – Stack, Heap und Method-Area – werden im nächsten Kapitel genauer erläutert.

6.4.5 Speicherbereiche für Variablen

Die drei Variablenarten – **lokale Variablen**, **Instanzvariablen** und **Klassenvariablen** – werden in verschiedenen Speicherbereichen abgelegt. Diese Speicherbereiche – **Stack**, **Heap** und **Method-Area** – werden alle von der Java-Virtuellen Maschine verwaltet.

In den nächsten drei Unterkapiteln erfolgt eine kurze Vorstellung von Stack, Heap und Method-Area.

6.4.5.1 Der Stack

Als **Stack** wird ein Speicherbereich bezeichnet, auf dem Informationen temporär abgelegt werden können. Ein Stack wird auch als Stapel bezeichnet. Ganz allgemein ist das Typische an einem Stack, dass auf die Information, die zuletzt abgelegt worden ist, als erstes wieder zugegriffen werden kann. Denken Sie z. B. an einen Bücherstapel. Sie beginnen mit dem ersten Buch, legen darauf das zweite, dann das dritte und so fort. In diesem Beispiel soll beim fünften Buch Schluss sein. Beim Abräumen nehmen Sie zuerst das fünfte Buch weg, dann das vierte, dann das dritte und so weiter, bis kein Buch mehr da ist. Bei einem Stack ist es nicht erlaubt, Elemente von unten oder aus der Mitte des Stacks wegzunehmen.

Die Datenstruktur eines Stacks wird als **LIFO-Datenstruktur** bezeichnet. LIFO bedeutet "**Last-In-First-Out**", d. h. das, was als Letztes abgelegt wird, wird wieder als Erstes entnommen. Das Ablegen eines Elementes auf dem Stack wird als **`push()`**-Operation, das Wegnehmen eines Elementes als **`pop()`**-Operation bezeichnet.

Ein Stack wird damit durch seine beiden Operationen `push()` und `pop()` gekennzeichnet und durch die Einschränkung, dass die Zahl der Elemente auf dem Stack nicht kleiner als Null werden kann und auch nicht höher als die Stackgröße. Das folgende Bild symbolisiert einen Stack aus Büchern:

Bild 6-9 Auf- und Abbau eines Bücherstapels

In Programmen wird eine solche Datenstruktur dazu benutzt, um die Daten eines Programms zu organisieren. Auf einem Programmstack werden zum Beispiel lokale Variablen einer Methode gespeichert. Ruft eine Methode eine weitere Methode auf, so muss auch der Inhalt des Befehlszeigers der aufrufenden Methode auf dem Stack zwischengespeichert werden, damit – wenn die aufgerufene Methode fertig ist – an der richtigen Stelle der aufrufenden Methode weiter gearbeitet werden kann. Dass auch die Übergabewerte für eine Methode sowie der Rückgabewert einer aufgerufenen Methode vorübergehend auf dem Stack abgelegt werden, soll hier nur beiläufig erwähnt und nicht vertieft werden.

Der Stack dient bei Programmen als Speicherbereich, um Daten zu organisieren. Bei einem Methodenaufruf werden auf dem Stack die lokalen Variablen einer Methode, der Rückgabewert und die Rücksprungadresse einer Methode hinterlegt, die durch den Aufruf einer anderen Methode in ihren eigenen Anweisungen unterbrochen wurde.

6.4.5.2 Der Heap

Aufgabe des **Heap** ist es, **Speicherplatz für die Schaffung dynamischer Variablen** bereitzuhalten.

Der `new`-Operator, der vom Anwendungsprogramm aufgerufen wird, um eine Variable auf dem Heap anzulegen, gibt dem Anwendungsprogramm eine Referenz auf eine im Heap erzeugte dynamische Variable zurück.

Dynamische Variablen tragen keine Namen. Die erhaltene Referenz ermöglicht den Zugriff auf die entsprechende dynamische Variable im Heap. An welcher Stelle des Heap die dynamische Variable angelegt wird, entscheidet nicht der Programmierer, sondern die Java-Virtuelle Maschine.

Die dynamischen Variablen stehen von ihrer Erzeugung bis zum Programmende zur Verfügung, es sei denn, der Programmierer benötigt diese Variablen nicht mehr. Dann kann er die Referenz aufheben – dies erfolgt in Java, indem der Referenzvariablen die `null`-Referenz zugewiesen wird – und damit wird dem Garbage Collector ein Zeichen gegeben, dass er das nicht mehr referenzierte Objekt entfernen darf, wenn keine weitere Referenz auf dieses Objekt zeigt. Ob dies der Garbage Collector auch umsetzt, ist eine andere Sache.

Wird Speicher freigegeben, so kann der Speicherplatz im Heap für andere dynamische Variablen benutzt werden. Die Größe des Heap ist beschränkt. Daher kann es zu einem Überlauf des Heap kommen, wenn ständig nur Speicher angefordert und nichts zurückgegeben wird. Ein solcher Überlauf resultiert in einer Ausnahme (Exception) vom Typ `OutOfMemoryError` (siehe Kapitel 13.4).

In Java werden Objekte im Heap nicht explizit freigegeben. Es wird vielmehr in unregelmäßigen Abständen durch die Java-Virtuelle Maschine der sogenannte **Garbage Collector** aufgerufen. Der Garbage Collector kann den Speicherplatz, der nicht mehr referenziert wird, freigeben.

6.4.5.3 Die Method-Area

In der Method-Area befinden sich die Klassenvariablen. Klassenvariablen sind statische Variablen – denn sie tragen einen Namen und ihr Gültigkeitsbereich und ihre Lebensdauer ist durch die statische Struktur des Programms bestimmt. Der Gültigkeitsbereich einer Klassenvariablen hängt von ihrem Zugriffsmodifikator (siehe Kapitel 6.4.7) ab.

Die Lebensdauer einer Klassenvariablen beginnt mit dem Laden der Klasse und endet, wenn die Klasse vom Programm nicht mehr benötigt wird.

Nicht nur Klassenvariablen liegen in der Method-Area, sondern ebenso der gesamte Programmcode einer Klasse. Damit der Programmcode einer Klasse ausgeführt werden kann, muss die Klasse erst einmal in die Method-Area geladen werden.

Den Speicherbereich, in den die Java-Virtuelle Maschine den **Programmcode einer Klasse** und die **Klassenvariablen** ablegt, bezeichnet man als **Method-Area**.

6.4.6 Konstante Variablen

Mit dem Modifikator `final` kann jede Klassenvariable, Instanzvariable und lokale Variable – unabhängig davon, ob es nun eine Referenzvariable oder eine Variable eines einfachen Datentyps ist – konstant gemacht werden.

Das heißt, der Wert einer Variablen mit dem Modifikator `final` ist konstant und kann nicht mehr verändert werden.

Wird mit dem Schlüsselwort `final` eine Variable eines einfachen Datentyps zur Konstanten gemacht, so ist ihr Wert immer konstant.

Mit

```
final int konstantVar = 1;
```

wird beispielsweise eine Variable vom Typ `int` angelegt. Nach der Initialisierung mit dem Wert `1` kann keine weitere Zuweisung an die konstante Variable `konstantVar` erfolgen. Das Gleiche gilt für Referenzvariablen.

Wird eine Referenzvariable mit Hilfe des Modifikators `final` zu einer konstanten Referenzvariablen gemacht, so muss diese Referenz immer auf das Objekt zeigen, mit dessen Adresse die Referenzvariable initialisiert wurde.

Die folgende Codezeile legt eine konstante Referenz `p` an, die immer auf dasselbe Objekt der Klasse `Punkt` zeigt, mit dessen Referenz es initialisiert wurde:

```
final Punkt p = new Punkt();
```

Die **Inhalte eines Objektes**, **auf das eine konstante Referenz zeigt**, **können problemlos verändert werden**, da ja nur die Referenz konstant ist. Es gibt in Java **keine Möglichkeit**, **ein Objekt konstant zu machen**.

6.4.7 Modifikatoren

Bei der Deklaration von Datenfeldern können zusätzlich **Modifikatoren** (engl. **modifiers**) angegeben werden. Es gibt aber nicht nur Modifikatoren für Datenfelder, sondern auch für Methoden, Konstruktoren, Klassen und Schnittstellen.

Im Folgenden werden alle Modifikatoren aufgelistet:

- `public`, `private`, `protected` für die Zugriffsrechte (siehe Kapitel 12.7)),
- `static` für Klassenvariablen, Klassenmethoden, geschachtelte Klassen und Schnittstellen,
- `final` für benannte (symbolische) Konstanten,
- `transient` für Datenfelder, die nicht serialisiert werden sollen (siehe Kapitel 19.7.3),
- `volatile` für Datenfelder, die von mehreren Threads gleichzeitig verändert werden können,
- `abstract` für die Kennzeichnung von abstrakten Klassen und Methoden,
- `native` für die Kennzeichnung von Methoden, die in einer anderen Sprache als Java implementiert sind,
- `synchronized` für den wechselseitigen Ausschluss von Methoden bzw. Blöcken (siehe Kapitel 16).

Die Definition einer konstanten Klassenvariablen könnte zum Beispiel folgendermaßen aussehen:

```
final static float PI = 3.14f;
```

Die folgende Tabelle zeigt, welcher Modifikator mit einem Datenfeld, einer Methode, einem Konstruktor, einer Klasse bzw. einer Schnittstelle eingesetzt werden darf:

	Datenfeld	Methode	Konstruktor	Klasse	Schnittstelle
abstract		ja		ja	
final	ja	ja		ja	
native		ja			
private	ja	ja	ja	ja[105]	ja[106]
protected	ja	ja	ja	ja[105]	ja[106]
public	ja	ja	ja	ja	ja
static	ja	ja		ja[105]	ja[106]
synchronized		ja			
transient	ja				
volatile	ja				

Tabelle 6-2 Verwendung von Zugriffsmodifikatoren

6.5 Array-Typ

Eine **Variable eines Array-Typs** ist eine Referenz auf ein Array-Objekt. Arrays sind in Java immer Objekte – es geht gar nicht anders!

Ein **Array** ist ein **Objekt**, das **aus Komponenten** (**Elementen**) zusammengesetzt ist, wobei jedes Element eines Arrays vom selben Datentyp sein muss.

Das folgende Bild zeigt ein Beispiel für ein Array:

Bild 6-10 Ein Array aus 5 `int`-Elementen

Man kann in Java Arrays aus **Elementen** eines **einfachen Datentyps** oder aus Elementen eines **Referenztyps** anlegen. Ein Element eines Arrays kann auch selbst wieder ein Array sein. Dann entsteht ein mehrdimensionales Array.

Im Folgenden werden zunächst **eindimensionale Arrays** betrachtet. Mehrdimensionale Arrays werden in Kapitel 6.5.4 besprochen. Die **Länge** oder **Größe** eines Arrays legt die Anzahl der Elemente des Arrays fest. Die Länge muss als Wert immer eine positive ganze Zahl haben. Ist `laenge` die Länge eines Arrays, so werden die Elemente von `0` bis `laenge - 1` durchgezählt. Die Nummer beim Durchzählen wird als **Index des Arrays** bezeichnet. Über den Array-Index kann man auf ein Element eines Arrays zugreifen. Der Zugriff auf das i-te Element des Arrays mit dem Namen `arrayName` erfolgt durch `arrayName [i - 1]`.

105 Gilt nur bei geschachtelten Klassen.

106 Gilt nur bei Schnittstellen innerhalb von Klassen.

Der Zugriff auf ein Element eines Arrays erfolgt über den Array-Index. Hat man ein **Array mit n Elementen** definiert, so ist darauf zu achten, dass in Java die **Indizierung der Arrayelemente mit 0 beginnt und bei n - 1 endet**.

Der Vorteil von Arrays gegenüber mehreren einfachen Variablen ist, dass Arrays sich leicht mit Schleifen bearbeiten lassen, da der Index einer Array-Komponente eine Variable sein kann und als Laufvariable in einer Schleife benutzt werden kann.

In Java sind **Arrays** stets **Objekte**, auch wenn man Arrays aus einfachen Datentypen als Komponenten anlegt. Arrays werden zur Laufzeit im Heap angelegt.

Dabei kann die Länge des anzulegenden Arrays zur Laufzeit berechnet werden. Ist das Array angelegt, so kann seine Länge nicht mehr verändert werden. Der Zugriff auf die Komponenten eines Arrays erfolgt über den Arraynamen, der eine **Referenz** auf das **Array-Objekt** darstellt.

Die Definition einer **Array-Variablen** bedeutet in Java nicht das Anlegen eines Arrays, sondern die Definition einer **Referenzvariablen**, die auf ein **Array-Objekt** zeigen kann. Dieses Array-Objekt muss **im Heap angelegt** werden.

Die allgemeine Form der Definition einer Referenzvariablen zum Zugriff auf ein eindimensionales Array ist:

```
Typname[] arrayName;
```

Ein konkretes **Beispiel** hierfür ist:

```
int[] alpha;
```

wobei `alpha` eine Referenzvariable ist, die auf ein Array aus Elementen (Komponenten) vom Typ `int` zeigen kann. Im folgenden Bild zeigt eine Referenzvariable `alpha` auf ein Array aus `int`-Komponenten:

Bild 6-11 Ein Array-Objekt im Heap, auf das die Referenzvariable `alpha` zeigt

Die Referenzvariable `alpha` kann auf ein Array-Objekt aus beliebig vielen Komponentenvariablen vom Typ `int` verweisen.

Die Definition

```
int[] alpha;
```

wird von rechts nach links gelesen: `alpha` ist eine Referenzvariable, die auf ein Array-Objekt aus Elementen vom Typ `int` zeigen kann.

Für Arrays gibt es in Java kein spezielles Schlüsselwort. Der Java-Compiler erkennt ein **Array** an den **eckigen Klammern**.

Arrays werden **in 3 Schritten angelegt**:

Schritt 1: **Definition** einer **Referenzvariablen**, die auf das Array-Objekt zeigen soll.
Schritt 2: **Erzeugen des Arrays**, d. h. eines Array-Objektes, welches aus Komponenten (Elementen) besteht.
Schritt 3: Belegen der Array-Elemente mit Werten, d. h. **Initialisierung des Arrays**.

Wie in Kapitel 7.6.1 gezeigt wird, können diese Schritte auch zusammengefasst werden.

Eine weitere Eigenschaft von Arrays in Java ist, dass eine genaue **Überwachung der Grenzen des Arrays durchgeführt** wird.

Es ist in Java nicht möglich, über die Grenzen eines Arrays hinaus andere Speicherbereiche zu überschreiben oder auszulesen. Bei einem solchen Versuch wird sofort eine Ausnahme (Exception) vom Typ `ArrayIndexOutOfBoundsException` geworfen.

Ausnahmen (Exceptions) werden in Kapitel 13 behandelt.

6.5.1 Arrays aus Elementen eines einfachen Datentyps

Zunächst muss eine Referenzvariable für ein Array-Objekt definiert werden. Dies erfolgt, **ohne die Länge** des Arrays anzugeben wie in folgendem Beispiel:

```
byte[] bArray;
```

Hier wird eine Referenzvariable `bArray` angelegt. Das folgende Bild zeigt die Referenzvariable `bArray`:

Bild 6-12 Die Referenzvariable `bArray`

Eine **Array-Variable** ist eine **Referenz auf ein Array-Objekt**. Mit der Definition einer Array-Variablen ist aber das Array-Objekt selbst noch nicht angelegt.

Erzeugen eines Array-Objektes

Zum Erzeugen eines Array-Objektes gibt es zwei Möglichkeiten:

- Die erste Möglichkeit ist, das Array **mit `new` zu erzeugen** und anschließend die Elemente des Arrays mit den gewünschten Werten zu initialisieren.
- Die andere Möglichkeit ist, das Array **über eine Initialisierungsliste anzulegen und gleichzeitig zu initialisieren**.

Beim Erzeugen eines Array-Objektes wird die Länge des Arrays festgelegt. Die Länge kann danach nicht mehr geändert werden.

Wenn ein Array angelegt ist, kann man über das **Datenfeld `length`**, das jedes Array besitzt, dessen Länge ermitteln.

Im Folgenden werden die beiden Möglichkeiten, ein Array-Objekt zu schaffen, vorgestellt:

- **Erzeugung mit dem `new`-Operator**

  Zunächst die erste Möglichkeit, d. h. die Verwendung von `new`, anhand eines **Beispiels**:

  ```
  bArray = new byte [4];
  ```

  Mit `new byte [4]` wird ein neues Array-Objekt erstellt, das Werte vom Typ `byte` aufnehmen kann. Es hat vier Komponenten, die beim Erstellen des Objektes mit dem Default-Wert[107] `0` initialisiert werden. Dies zeigt das folgende Bild:

Bild 6-13 Mit `0` initialisiertes `byte`-Array

[107] Bei Array-Komponenten gelten dieselben **Default-Werte** wie bei Datenfeldern (siehe Kapitel 10.4.1).

Es ist auch möglich, das Anlegen der Referenz und das Erzeugen des Objekts in einer einzigen Anweisung durchzuführen:

```
byte[] bArray = new byte [4];
```

Die Länge eines Arrays kann auch durch eine Variable angegeben werden. Damit kann die Länge des anzulegenden Arrays zur Laufzeit wie in folgendem **Beispiel** festgelegt werden:

```
int i = 5;
byte[] bArray = new byte [i];
```

Der Wert der Variablen `i` könnte somit auch von der Tastatur eingegeben oder mit Hilfe einer Berechnung bestimmt werden.

Initialisierung

Durch Zuweisung von Werten an die Komponenten können dann die Default-Werte der Komponenten mit sinnvollen Werten überschrieben werden, z. B. durch:

```
bArray [2] = 6;
```

- **Implizites Erzeugen über eine Initialisierungsliste**

Die andere Möglichkeit, ein Array anzulegen, ist, das Array implizit über eine Initialisierungsliste zu erzeugen und gleichzeitig zu initialisieren.

Hierfür ein **Beispiel**:

```
byte[] bArray = {1, 2, 3, 4};
```

Das Erzeugen des Array-Objektes wird hier vom Compiler im Verborgenen durchgeführt. Hierbei wird also die Definition der Referenzvariablen `bArray`, das Anlegen des Array-Objektes und die Initialisierung der Array-Elemente in einem einzigen Schritt durchgeführt. Bild 6-14 zeigt das Speicherabbild für das diskutierte Beispiel:

Bild 6-14 Mit einer Initialisierungsliste erzeugtes und initialisiertes `byte`-Array

Hervorzuheben ist, dass die Initialisierungsliste auch Ausdrücke und Variablen enthalten darf wie in folgendem **Beispiel**:

```
byte i = 1;
byte[] bArray = {i, i + 1, i + 2, i * 4};
```

6.5.2 Arrays aus Referenzen

Zunächst muss die Referenzvariable, die auf das noch anzulegende Array-Objekt zeigen soll, definiert werden. Dies erfolgt, ohne die Länge des Arrays anzugeben. Siehe hierzu das folgende **Beispiel**:

```
Klasse[] kArray;
```

Hier wird eine Referenzvariable `kArray` angelegt. Das Array-Objekt selbst ist jedoch noch nicht angelegt. Wenn das Array-Objekt erzeugt ist, kann man über das Datenelement `length`, welches jedes Array-Objekt hat, dessen Länge ermitteln. Das folgende Bild visualisiert die Referenzvariable `kArray`:

Bild 6-15 Die Referenzvariable `kArray`

Erzeugen des Array-Objektes

Auch hier gibt es die beiden schon bei den Arrays aus einfachen Datentypen gezeigten Möglichkeiten, nämlich die Array-Elemente mit `new` zu erzeugen oder eine Initialisierungsliste zu verwenden:

- **Erzeugen mit dem `new`-Operator**

  Zunächst die Verwendung von `new` anhand eines **Beispiels**:

  ```
  kArray = new Klasse [4];
  ```

  Mit `new Klasse [4]` wird ein neues Array-Objekt erstellt. Die vier Komponenten dieses Arrays sind Referenzvariablen vom Typ `Klasse`. Hier das Speicherabbild:

Bild 6-16 Mit `null` initialisiertes Array aus Referenzvariablen

  Jede angelegte Referenzvariable vom Typ `Klasse` wird mit dem Default-Wert `null` initialisiert.

  Es ist auch möglich, das Anlegen der Referenz und die Erstellung des Array-Objektes auf einmal durchzuführen:

```
Klasse[] kArray = new Klasse [4];
```

Initialisierung

Durch Zuweisung von Werten an die Komponenten können die Default-Werte der Komponenten des Arrays wie im folgendem **Beispiel** mit sinnvollen Werten überschrieben werden:

```
Klasse refAufObj = new Klasse();
. . . . .
kArray [2] = refAufObj;
. . . . .
```

Das folgende Bild zeigt eine mit einer Referenz auf ein Objekt der Klasse `Klasse` initialisierte Referenzvariable `kArray [2]` des betrachteten Arrays:

Bild 6-17 Initialisierung der Referenzvariablen `kArray [2]`

Das folgende **Programmbeispiel** zeigt die Initialisierung desselben Arrays mit Hilfe von Referenzen auf Objekte der Klasse `Klasse`:

```
for (int lv = 0; lv < kArray.length; lv = lv + 1)
       kArray [lv] = new Klasse();
```

Hierzu das Speicherabbild:

Bild 6-18 Mit Referenzen auf Objekte initialisiertes Array aus Referenzvariablen

- **Implizites Erzeugen über eine Initialisierungsliste**

  Die andere Möglichkeit, ein Array implizit über eine Initialisierungsliste anzulegen und gleichzeitig zu initialisieren, funktioniert auch bei Arrays aus Referenzvariablen:

```
Klasse[] kArray = {refK1, refK2, refK3, new Klasse()};
           // dabei müssen refK1, refK2 und refK3 Referenzen
           // auf vorhandene Objekte vom Typ Klasse sein
```

  In der Initialisierungsliste können entweder Referenzvariablen angegeben oder direkt Objekte eines bestimmten Typs mit Hilfe des `new`-Operators erzeugt werden. In folgendem Programmausschnitt soll das Anlegen eines Arrays über eine Initialisierungsliste und der Zugriff auf die in den Array-Elementen referenzierten Objekte demonstriert werden:

```
// Die Referenzen p1 und p2 sollen auf Objekte der Klasse
// Person zeigen. Die folgende Codezeile legt eine Referenz-
// variable arr für ein Array von Personen an. Es wird ein
// Array-Objekt mit 2 Elementen auf dem Heap angelegt und
// mit den Referenzvariablen p1 und p2 initialisiert.
Person[] arr = {p1, p2};

// Die folgende Codezeile zeigt einen Aufruf der Methode
// print() für das erste Array-Element.
arr [0].print();
```

6.5.3 Objektcharakter von Arrays

Variablen vom Typ eines Arrays sind Referenzen auf Array-Objekte, die zur Laufzeit des Programms dynamisch auf dem Heap angelegt werden. Jeder Array-Typ wird implizit, d. h. ohne eine explizite Angabe des Programmierers, von der Klasse `Object` abgeleitet. Damit beinhaltet jeder Array-Typ automatisch alle Methoden der Klasse `Object`. Zusätzlich enthält jeder Array-Typ das konstante Datenfeld `length` vom Typ `int`, das die Anzahl der Array-Elemente als Wert beinhaltet.

Das folgende Bild soll visualisieren, dass Array-Typen von der Klasse `Object` abgeleitet sind:

Bild 6-19 Arrays als implizite Subklassen der Klasse `Object`

Das folgende Beispiel demonstriert den Aufruf der Methode `equals()` der Klasse `Object` für Array-Typen sowie die Verwendung des Datenfeldes `length`. Die Methode `equals()` der Klasse `Object` hat die Schnittstelle

```
public boolean equals (Object ref)
```

Diese Methode gibt bei einem Aufruf

```
x.equals (y)
```

`true` zurück, wenn `x` und `y` Referenzen auf dasselbe Objekt sind.

Beispiel:

```
// Datei: Arrays.java

public class Arrays
{
   public static void main (String[] args)
   {
      int[] alpha = new int [2];
      int[] beta;
      beta = alpha;           // beta zeigt auf dasselbe
                              // Array-Objekt wie alpha
      System.out.println ("alpha equals beta ist " +
                          alpha.equals (beta));
      System.out.println ("alpha hat " + alpha.length +
                          " Komponenten");
   }
}
```

Die Ausgabe des Programms ist:

```
alpha equals beta ist true
alpha hat 2 Komponenten
```

Arrays aus Basisklassen dienen zur flexiblen Speicherung von Objekten verschiedenster abgeleiteter Klassen. Arrays aus Basisklassen werden in Kapitel 11.4.2 behandelt.

6.5.4 Mehrdimensionale Arrays

Mehrdimensionale Arrays stellen Arrays aus Arrays dar und werden wie in folgendem **Beispiel** erzeugt:

```
int[][][] dreiDimArray = new int[10][20][30];
```

Es können auch sogenannte **offene Arrays** erzeugt werden. Offene Arrays sind Arrays, bei denen die Länge einzelner Dimensionen nicht angegeben wird. Hierfür lässt man einfach bei der Speicherplatz-Allokierung mit `new` die eckigen Klammern leer. Dies ist jedoch **nur bei mehrdimensionalen Arrays möglich**, da der ersten Dimension eines Arrays immer ein Wert zugewiesen werden muss. Es ist allerdings

nicht erlaubt, nach einer leeren eckigen Klammer noch einen Wert in einer der folgenden Klammern anzugeben.

So ist beispielsweise

```
int[][][][] matrix = new int[5][3][][];
```

erlaubt, aber

```
int[][][][] matrix = new int[5][][][4];
```

nicht und genauso wenig die folgende Codezeile:

```
int[][][][] matrix = new int[][][][];
```

Mehrdimensionale Arrays müssen nicht unbedingt rechteckig sein. Es spricht nichts dagegen, die Elemente der ersten Dimension eines mehrdimensionalen Arrays in den verschiedenen Dimensionen mit unterschiedlich langen Arrays zu initialisieren.

Das folgende **Beispielprogramm** wendet dies an. Es legt ein dreiecksförmiges Array an, füllt es mit den Werten des pascalschen Dreiecks bis zur zehnten Ebene und gibt dieses am Bildschirm aus.

```
// Datei: PascalDreieck.java

public class PascalDreieck
{
   public static void main (String[] args)
   {
      final int EBENE = 10;
      int i, j;
      int[][] binom = new int[EBENE][];

      for (i = 0; i < binom.length; i++)
      {
         // Anlegen eines Arrays mit der Größe der entsprechenden
         // Ebene.
         binom[i] = new int[i + 1];

         // Erstes Element einer Ebene mit 1 belegen.
         binom[i][0] = 1;

         // Letztes Element einer Ebene mit 1 belegen.
         binom[i][binom[i].length - 1] = 1;
         System.out.printf ("%1d ", binom[i][0]);

         for (j = 1; j < binom [i].length - 1; j++)
         {
            binom[i][j] = binom[i - 1][j - 1] + binom[i - 1][j];
            System.out.printf ("%3d ", binom[i][j]);
         }

         if (i > 0)
         {
```

```
            // Für alle Ebenen ausser der ersten wird zum Schluss
            // noch eine 1 ausgegeben.
            System.out.printf ("%3d", binom[i][binom[i].length-1] );
         }

         // Ausgabe eines Zeilenumbruchs nach jeder Ebene.
         System.out.println();
      }
   }
}
```

Die Ausgabe des Programms ist:

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5  10  10   5   1
1   6  15  20  15   6   1
1   7  21  35  35  21   7   1
1   8  28  56  70  56  28   8   1
1   9  36  84 126 126  84  36   9   1
```

Mit `int[][] binom = new int[EBENE][]` wird in diesem Beispiel eine Referenz auf ein Array mit 2 Dimensionen angelegt. Dabei sind in der ersten Dimension 10 Elemente vorhanden. Die zweite Dimension wird noch nicht mit Elementen belegt. Bild 6-20 zeigt diesen Sachverhalt:

Bild 6-20 Ein zweidimensionales Array mit 10 Elementen in der ersten Dimension

Man beachte, dass in der zweiten Dimension `int`-Arrays mit unterschiedlichen Größen angelegt werden können. Da diese aber noch nicht erzeugt wurden, zeigen alle Array-Elemente der ersten Dimension auf `null`.

Danach wird das Array – auf das `binom` zeigt – in der ersten Dimension vom ersten Element an durchlaufen und jedem dieser Elemente wird mit der Anweisung `binom[i] = new int[i + 1];` ein `int`-Array von der Größe der Variablen `i + 1` zugewiesen. Das eindimensionale `int`-Array, das dem Element `binom[0]` zugewiesen wird, hat also die Größe 1. Nach dem zweiten Schleifendurchlauf sieht der Sachverhalt folgendermaßen aus:

Bild 6-21 Von der Referenzvariablen `binom` referenziertes Array nach dem zweiten Schleifendurchlauf

Entsprechend werden die weiteren `int`-Arrays für die zweite Dimension angelegt.

6.5.5 Schreibweise von Arrays

Java erlaubt mehrere Syntax-Varianten bei der Definition von Referenzvariablen auf Arrays.

Wie die folgenden Beispiele zeigen, können Array-Klammern entweder hinter oder vor dem Namen der Referenzvariablen angegeben werden:

```
int    zahl[];
char[] buffer;
```

Die zweite Variante bedeutet zwar eine Umstellung zu der von C gewohnten Schreibweise, sie entspricht aber der von den einfachen Datentypen bekannten syntaktischen Form:

```
datentypname variablenname;
```

Es ist auch eine gemischte Schreibweise möglich:

```
byte[] zeile, spalte, matrix[];
```

Hier ist also `matrix` ein zweidimensionales Array aus Bytes.

Von der gemischten Schreibweise ist jedoch aus Gründen der Übersichtlichkeit abzuraten.

6.6 Aufzählungstyp

Nach der Erklärung der Typsicherheit von Aufzählungstypen in Kapitel 6.6.1 wird in Kapitel 6.6.2 erläutert, wie Java Aufzählungstypen modelliert. Aufzählungskonstanten sind als konstante Referenzvariablen auf Objekte zu betrachten (siehe Kapitel 6.6.3). Nach der Vorstellung einiger Standardmethoden für Aufzählungstypen in Kapitel 6.6.4 wird in Kapitel 6.6.5 dargelegt, dass Aufzählungstypen auch Datenfelder und Methoden aufweisen können. Nach der Vorstellung von Konstruktoren für Aufzählungstypen in Kapitel 6.6.6 wird in Kapitel 6.6.7 das Überschreiben von Methoden für Aufzählungskonstanten erläutert.

6.6.1 Typsicherheit von Aufzählungstypen

In Java sind mit Java 5 Aufzählungstypen hinzugekommen, die in der Vergangenheit schmerzlich vermisst wurden. Aufzählungstypen sind vom Prinzip her einfache Datentypen. Zur Erläuterung, was ein Aufzählungstyp prinzipiell ist, hier ein Beispiel in der Programmiersprache Pascal:

```
type werktag = (Mo, Di, Mi, Dn, Fr, Sa);
var x: werktag;
```

Mit dem Typ `werktag` wird festgelegt, dass eine Variable `x` dieses Typs als Wert genau eine der **Aufzählungskonstanten** `Mo` (steht für Montag), `Di` (Dienstag) etc. annehmen kann. Die Zuweisung anderer Werte, als in der Liste der Aufzählungskonstanten aufgeführt, an eine Variable des Aufzählungstyps wird vom Compiler abgelehnt. Damit wird die **Typsicherheit bereits beim Kompilieren** garantiert. Hätte man stattdessen die Wochentage durch ganze Zahlen beschrieben, könnten einer solchen Variablen beliebige ganze Zahlen zugewiesen werden, die keinen Wochentagen entsprechen würden – und der Compiler hätte keine Möglichkeit, solche Fehler zu verhindern.

Eine Variable eines Aufzählungstyps enthält nur einen einzigen Wert, der sich nicht aus weiteren Werten zusammensetzt. Mit anderen Worten, der Wert einer Variablen eines Aufzählungstyps ist atomar. Der einzige Wert muss einem Wert aus der Liste der Aufzählungskonstanten entsprechen.

Aufzählungstypen – in Java auch **enums** genannt – sind Datentypen, die als Wertebereich eine endliche geordnete Menge von Konstanten zulassen. Da die Elemente einer endlichen geordneten Menge abzählbar sind, erhielten diese Datentypen den Namen Aufzählungstypen (engl. **enumeration types**, abgekürzt **enums**). Die Konstanten werden als Aufzählungskonstanten (engl. **enumeration constants**, abgekürzt **enum constants**) bezeichnet. Eine Variable eines Aufzählungstyps kann als Wert eine dieser Aufzählungskonstanten besitzen.

Aufzählungstypen in ihrer ursprünglichen Form wie in Pascal oder C sind einfache Datentypen.

6.6.2 Aufzählungstypen in Java

In Java werden Aufzählungstypen jedoch in Form von Klassen angelegt.

Damit gehören sie in Java zu den Referenztypen und sind im Unterbaum Referenztypen in Bild 6-1 eingeordnet.

Enums wurden in Java mit der Version Java 5 eingeführt. Sie funktionieren mit älteren Versionen des Compilers nicht.

Ein Aufzählungstyp trägt einen Namen. Bei der Definition des Typs werden die Aufzählungskonstanten in Form einer Liste wie im folgenden **Beispiel** angegeben:

```
enum AmpelFarbe {ROT, GELB, GRUEN}
```

`AmpelFarbe` ist hier der **Name des Aufzählungstyps** und `ROT`, `GELB` und `GRUEN` sind Aufzählungskonstanten. Durch die Angabe der Aufzählungskonstanten in Form einer Liste entsteht eine **Reihenfolge der Aufzählungskonstanten**. Ein Aufzählungstyp ist daher ein ordinaler Datentyp. Das bedeutet, dass den Aufzählungskonstanten vom Compiler Werte in aufsteigender Reihenfolge zugeordnet werden.

Um auf eine **Aufzählungskonstante eines Aufzählungstyps** zuzugreifen, wird der entsprechende Aufzählungstyp mittels der Punktnotation der Aufzählungskonstanten vorangestellt.

Hierfür ein **Beispiel**:

```
AmpelFarbe.GELB
```

Hier ein **Programmbeispiel**, das den Aufzählungstyp `AmpelFarbe` definiert und verwendet:

```
// Datei: Ampel.java

public class Ampel
{
   // Der Aufzählungstyp wird hier in der Klasse definiert, in der
   // er verwendet wird. Die Ampel kann ROT, GELB oder GRUEN sein.
   public enum AmpelFarbe {ROT, GELB, GRUEN}

   // Instanzvariable des Aufzählungstyps Ampelfarbe
   private AmpelFarbe farbe;

   public void setAmpelFarbe (AmpelFarbe ampelFarbe)
   {
      farbe = ampelFarbe;
   }
```

```
   public AmpelFarbe getAmpelFarbe()
   {
      return farbe;
   }

   // main()-Methode zum Testen
   public static void main (String[] args)
   {
      Ampel ampel = new Ampel();

      ampel.setAmpelFarbe (AmpelFarbe.ROT);  // funktioniert
      // Das Folgende geht nicht! Es können nur die Aufzählungs-
      // konstanten des Aufzählungstyps AmpelFarbe verwendet werden
      // ampel.setAmpelFarbe (3);

      System.out.println ("Die Ampel ist: "
                          + ampel.getAmpelFarbe());
      System.out.println ("Die Ordinalzahl ist: "
                          + ampel.getAmpelFarbe().ordinal());

      ampel.setAmpelFarbe (AmpelFarbe.GELB);
      System.out.println ("Der Name ist: "
                          + ampel.getAmpelFarbe().name());
      System.out.println ("Die Ordinalzahl ist: "
                          + ampel.getAmpelFarbe().ordinal());
   }
}
```

Die Ausgabe des Programms ist:

```
Die Ampel ist: ROT
Die Ordinalzahl ist: 0
Der Name ist: GELB
Die Ordinalzahl ist: 1
```

Wie im obigen Beispiel zu sehen ist, sind Aufzählungstypen typsicher, d. h., der Compiler lässt keine ungültigen Zuweisungen zu.

Einer Variablen eines Aufzählungstyps können nur Werte aus der Menge der Aufzählungskonstanten zugewiesen werden.

6.6.3 Aufzählungskonstanten als konstante Referenzvariablen

Die einfache Notation

```
public enum AmpelFarbe {ROT, GELB, GRUEN}
```

setzt der Compiler in eine Notation der folgenden Art um:

```
public final class AmpelFarbe extends Enum
{
   public static final AmpelFarbe ROT = new AmpelFarbe ("ROT", 0);
   public static final AmpelFarbe GELB = new AmpelFarbe ("GELB", 1);
   public static final AmpelFarbe GRUEN =new AmpelFarbe ("GRUEN",2);
```

```
    private AmpelFarbe (String s, int i)
    {
       super (s, i);
    }
}
```

In Java ist ein **Aufzählungstyp eine Klasse** und die **Aufzählungskonstanten sind konstante Referenzvariablen auf Objekte des Aufzählungstyps**. Da ein Aufzählungstyp eine Klasse ist, können einem Aufzählungstyp auch Methoden und Datenfelder hinzugefügt werden.

Wird eine Referenzvariable als `final` deklariert, so kann dieser Variablen kein anderer Wert zugewiesen werden und damit zeigt diese Referenzvariable immer auf dasselbe Objekt. Des Weiteren ist zu beachten, dass der Compiler für jede Aufzählungskonstante genau ein Objekt des Aufzählungstyps erzeugt.

Es ist nicht möglich, dass der Programmierer selbst mit Hilfe des `new`-Operators weitere Objekte eines Aufzählungstyps anlegt.

Der Programmierer kann nur Referenzvariablen des Aufzählungstyps anlegen, die auf eine der definierten Aufzählungskonstanten zeigen können.

Jede Aufzählungskonstante zeigt auf ein **Objekt des Aufzählungstyps**, welches den **Namen** der Aufzählungskonstanten als String und auch den **Ordinalwert** der Aufzählungskonstanten als Instanzvariable enthält.

Da für jede Aufzählungskonstante nur ein einziges Objekt existiert, kann der Operator == verwendet werden, um Aufzählungskonstanten zu vergleichen.

6.6.4 Standardmethoden von Aufzählungstypen

Die Klasse `Enum` ist in der Java-Klassenbibliothek im Paket `java.lang` zu finden. In der Klasse `Enum` werden auch die Methoden `ordinal()` und `name()` definiert, die damit zu jedem Aufzählungstyp aufgerufen werden können.

Der Aufruf der Methode **`ordinal()`** für eine **Aufzählungskonstante** liefert ihre **Ordinalzahl**.

Die Ordinalzahl spiegelt die Reihenfolge der Definition der Aufzählungskonstanten beginnend bei dem Wert 0 wieder. Liefert also beispielsweise eine Aufzählungskonstante den Wert 4, bedeutet dies, dass diese Aufzählungskonstante als fünftes Element definiert wurde und mindestens vier weitere Aufzählungskonstanten vorhanden sein müssen.

Der Aufruf der Methode **`name()`** für eine Aufzählungskonstante liefert den exakt definierten **Namen der Aufzählungskonstanten** als ein Objekt der Klasse `String` zurück.

Neben den schon vorgestellten Instanzmethoden `name()` und `ordinal()`, die jeder Aufzählungstyp von der Klasse `Enum` erbt, werden vom Compiler beim Übersetzen eines Aufzählungstyps automatisch die folgenden Klassenmethoden hinzugefügt:

```
public static E[] values()
public static E valueOf (String name)
```

Hierbei ist `E` der Name des betrachteten Aufzählungstyps.

Hier sofort einige **Beispiele**:

- Der Aufruf

  ```
  AmpelFarbe.values();
  ```

  liefert ein Array von Referenzvariablen zurück, die auf alle Aufzählungskonstanten zeigen, die für den Aufzählungstyp `AmpelFarbe` definiert sind. Hierbei entspricht die Reihenfolge der Referenzvariablen im Array der Reihenfolge, wie die Aufzählungskonstanten für den Aufzählungstyp definiert wurden.

- Der Aufruf

  ```
  AmpelFarbe.valueOf ("ROT");
  ```

  gibt die Aufzählungskonstante `ROT` des Aufzählungstyps `AmpelFarbe` zurück.

- Das folgende **Beispiel** zeigt die Verwendung der Methoden `values()` und `valueOf()` am Beispiel des Aufzählungstyps `AmpelFarbe`:

  ```
  // Datei: AmpelFarbeTest.java
  enum AmpelFarbe {ROT, GELB, GRUEN}

  public class AmpelFarbeTest
  {
     public static void main (String[] args)
     {
        // Abfragen aller in AmpelFarbe definierten Konstanten
        AmpelFarbe[] alleFarben = AmpelFarbe.values();
        System.out.println
           ("Folgende Konstanten sind in AmpelFarbe definiert:");
        // Ausgabe aller Farben auf dem Bildschirm
        // mit Hilfe einer for-Schleife.
        for (int i = 0; i < alleFarben.length; i++)
        {
           System.out.println (alleFarben[i].name());
        }
        // Beschaffen der Konstante ROT des Aufzählungstyps
        // AmpelFarbe
        AmpelFarbe rot = AmpelFarbe.valueOf ("ROT");
        System.out.println (rot + " ist in AmpelFarbe definiert.");
  ```

```
        try
        {
           // Bei einem ungültigen Wert wird eine Exception vom Typ
           // IllegalArgumentException geworfen
           AmpelFarbe ungueltig = AmpelFarbe.valueOf ("BLAU");
        }

        catch (IllegalArgumentException e)
        {
           System.err.println ("Ampelfarbe ist ungültig!");
        }
     }
  }
```

Die Ausgabe des Programms ist:

```
Folgende Konstanten sind in AmpelFarbe definiert:
ROT
GELB
GRUEN
ROT ist in AmpelFarbe definiert.
Ampelfarbe ist ungültig!
```

6.6.5 Datenfelder und Methoden von Aufzählungstypen

Da ein Aufzählungstyp eine Klasse darstellt, können die Aufzählungstypen auch Datenfelder und Methoden haben.

Dies zeigt das folgende **Beispiel**:

```
// Datei: Name1.java
public enum Name1
{
   // Definition der Aufzählungskonstanten
   PETER,
   HANS,
   JULIA,
   ROBERT;

   // Datenfeld
   private int note;

   // Methoden, um auf das Datenfeld zuzugreifen
   public int getNote()
   {
      return note;
   }

   public void setNote (int var)
   {
      note = var;
   }
}
```

Die folgende Klasse `NameTest` holt die Aufzählungskonstante `HANS` des Aufzählungstyps `Name1` und setzt die Note dieses Objektes. Anschließend wird die Note ausgelesen und ausgegeben:

```
// Datei: NameTest.java
public class NameTest
{
   public static void main (String[] args)
   {
      // Zuweisen des Elementes HANS aus dem Aufzählungstyp Name1
      // an die Variable des Aufzählungstyps Name1.
      Name1 name = Name1.HANS;

      // Aufrufen von Methoden des Objektes des Aufzählungstyps
      name.setNote (2);
      System.out.println ("Hans hat die Note: " + name.getNote());
   }
}
```

Die Ausgabe des Programms ist:

```
Hans hat die Note: 2
```

6.6.6 Konstruktoren von Aufzählungstypen

Ebenso wie normale Methoden können die Aufzählungstypen auch Konstruktoren enthalten. Um dies zu zeigen, wird dem obigen **Beispiel** noch ein Konstruktor hinzugefügt und entsprechend verwendet:

```
// Datei: Name2.java
public enum Name2
{
   // Definition der Aufzählungskonstanten
   PETER (2),  // In den runden Klammern steht der Parameter,
   HANS (4),   // der an den Konstruktor übergeben wird.
   JULIA (1),
   ROBERT (2);

   // Datenfeld
   private int note;

   // Konstruktor
   Name2 (int var)
   {
      note = var;
   }

   // Methoden, um auf das Datenfeld zuzugreifen
   public int getNote()
   {
      return note;
   }
```

```
   public void setNote (int var)
   {
      note = var;
   }
}
```

Wird der Konstruktor – wie oben gezeigt – verwendet, wird das Datenfeld `note` für alle Elemente der Menge entsprechend initialisiert. Es gelten dieselben Regeln wie bei normalen Klassen: Wird ein eigener Konstruktor bereitgestellt, wird dadurch der parameterlose Konstruktor, der durch den Compiler bereitgestellt wird, unsichtbar und kann nicht mehr verwendet werden.

Ein Konstruktor von Aufzählungstypen darf weder `public` noch `protected` sein.

6.6.7 Überschreiben von Methoden

Jede Aufzählungskonstante kann mit Hilfe einer anonymen Klasse die Methoden, welche im Aufzählungstyp definiert sind, überschreiben. Dies ist zunächst etwas gewöhnungsbedürftig. Das folgende Beispiel soll den Sachverhalt veranschaulichen: Hans möchte immer der Beste sein und behauptet daher immer, eine 1 geschrieben zu haben. Hier dieses **Beispiel**:

```
// Datei: Name3.java

public enum Name3
{
   // Definition der Aufzählungskonstanten
   PETER (1),
   // Überschreiben der Methode getNote()
   // für die Aufzählungskonstante HANS
   HANS (5)
   {
      public int getNote()
      {
         return 1;
      }
   },
   JULIA (1),
   ROBERT (2);

   // Datenfeld
   private int note;

   Name3 (int var)
   {
      note = var;
   }

   // Methoden, um auf das Datenfeld zuzugreifen
   public int getNote()
   {
```

```
      return note;
   }

   public void setNote (int var)
   {
      note = var;
   }
}
```

In diesem Fall wird der Methodenaufruf `getNote()` für Hans immer `1` zurückliefern.

Werden Methoden eines Aufzählungstyps als `abstract` definiert, müssen sie von allen Aufzählungskonstanten des Aufzählungstyps überschrieben werden.

Methoden können bei Aufzählungstypen für jede einzelne Aufzählungskonstante überschrieben werden. Dies geschieht an derjenigen Stelle, an der die Aufzählungskonstanten erzeugt werden.

Das folgende **Beispielprogramm** verwendet den oben angegebenen Aufzählungstyp `Name3`:

```
// Datei: NameTest2.java

public class NameTest2
{
   public static void main (String[] args)
   {
      // Zuweisen der Objekte JULIA und HANS aus der Menge der Auf-
      // zählungskonstanten an lokale Variablen des Aufzählungstyps.
      Name3 julia = Name3.JULIA;
      Name3 hans = Name3.HANS;
      // Beide bekommen ihre Note mitgeteilt
      System.out.println ("Professor: Julia hat die Note 2");
      julia.setNote (2);
      System.out.println ("Professor: Hans hat die Note 5");
      hans.setNote (5);
      // Beide werden nach ihren Noten gefragt
      System.out.println ("Julia: Ich habe eine "+ julia.getNote());
      System.out.println ("Hans: Ich habe eine " + hans.getNote());
   }
}
```

Die Ausgabe des Programms ist:

```
Professor: Julia hat die Note 2
Professor: Hans hat die Note 5
Julia: Ich habe eine 2
Hans: Ich habe eine 1
```

Da für die Aufzählungskonstante `HANS` die Methode `getNote()` überschrieben ist, wird immer die Wunschnote von Hans, die `1`, zurückgegeben.

6.7 Zeichenketten

Strings sind in Java – wie auch in anderen Programmiersprachen – Zeichenketten, d. h. Folgen von Zeichen. In Java sind Strings Objekte.

In Java gibt es drei verschiedene String-Klassen:

- die Klasse `String` für konstante Zeichenketten
- und die Klasse `StringBuffer` sowie die Klasse `StringBuilder` für variable Zeichenketten.

Diese Datentypen sollen in den nächsten Unterkapiteln näher betrachtet werden.

6.7.1 Konstante Zeichenketten

Eine konstante Zeichenkette ist eine Folge von Zeichenkonstanten, die nicht abgeändert werden kann. Sie kann also nur gelesen werden (engl. read only). Tabelle 6-3 zeigt exemplarisch den Aufbau einiger konstanter Zeichenketten aus Zeichenkonstanten:

konstante Zeichenkette (Objekt)	Enthält
`"alpha"`	`'a''l''p''h''a'`
`"Pia"`	`'P''i''a'`

Tabelle 6-3 Beispiel für den Aufbau konstanter Zeichenketten

Konstante Zeichenketten sind in Java Objekte der Klasse `String`. Die Klasse `String` repräsentiert eine Zeichenkette mit folgenden Eigenschaften:

- Die Länge eines Strings steht fest und kann auch nicht verändert werden.
- Der Inhalt des Strings kann nicht verändert werden.

Kurz und gut, ein **String** ist eine Konstante. Ziel dieser zwei Eigenschaften ist es, ein ungewolltes Überschreiben von Speicherinhalten in Programmen zu vermeiden und die Programme dadurch sicherer zu machen.

Eine konstante Zeichenkette `"Peter"` ist ein Ausdruck und hat als Rückgabewert eine Referenz auf das Objekt der Klasse `String`, das den Inhalt `'P''e''t''e''r'` hat.

Um das Ende eines Strings bei vorgegebener Anfangsposition des Strings zu finden, gibt es zwei prinzipielle Möglichkeiten:

- Erstens, sich die Länge des Strings zu merken. Dann weiß man, an welcher Position das letzte Zeichen des Strings steht.
- Zweitens, ein besonderes Zeichen zu verwenden, das unter den Buchstaben und Ziffern des verwendeten Alphabets nicht vorkommt und das an das letzte Zeichen des Strings angehängt wird, um das Ende des Strings anzuzeigen.

Diese beiden prinzipiellen Möglichkeiten visualisiert das folgende Bild:

Bild 6-22 Erkennen des Stringendes mit Stringlänge oder speziellem Endezeichen

In Java wird die erste Methode angewandt.

Das Datenfeld, in welchem die Länge eines Strings abgelegt ist, lässt sich mit der **Methode `length()`** der Klasse `String` abfragen. Beim Aufruf gibt `length()` die Anzahl der Zeichen des Strings zurück.

6.7.1.1 Erzeugung von Strings

Für die Erzeugung von Strings gibt es zwei prinzipielle Möglichkeiten:

- **Erzeugung eines Objektes vom Typ `String` mit dem `new`-Operator und Initialisierung mit einem Konstruktor**[108]

  Durch die Erzeugung eines Strings mit `new` wird ein neues Objekt vom Typ `String` im Heap angelegt. Zur Initialisierung bietet die Klasse `String` verschiedene Möglichkeiten. So kann das Objekt vom Typ `String` mit einer konstanten Zeichenkette initialisiert werden wie in folgendem **Beispiel**:

  ```
  String name1 = new String ("Anja");
  ```

  Der `new`-Operator gibt dabei eine Referenz auf das auf dem Heap angelegte `String`-Objekt `"Anja"` zurück, welche in der Referenzvariablen `name1` abgespeichert wird. In Analogie dazu zeigt die Referenzvariable `name2` auf das `String`-Objekt `"Herbert"` auf dem Heap:

  ```
  String name2 = new String ("Herbert");
  ```

  Objekte vom Typ `String` können nicht abgeändert werden.

  Durch die Zuweisung

  ```
  name1 = name2;
  ```

  zeigt nun die Referenzvariable `name1` auch auf das `String`-Objekt `"Herbert"`.

[108] Konstruktor siehe Kapitel 10.4.4

Die folgenden beiden Bilder zeigen diese Situation:

Bild 6-23 Referenzen und String-Objekte vor der Zuweisung `name1 = name2`

Bild 6-24 Referenzen und String-Objekte nach der Zuweisung `name1 = name2`

Einer Referenzvariablen vom Typ `String` kann eine Referenz auf ein anderes String-Objekt zugewiesen werden. Die auf dem Heap angelegten String-Objekte sind unveränderlich, den Referenzvariablen vom Typ `String` können jedoch neue Werte zugewiesen werden.

Im nächsten **Beispiel** erfolgt die Initialisierung mit Hilfe einer Referenzvariablen, die auf ein Array von Zeichen zeigt, das mit Hilfe einer Initialisierungsliste angelegt wurde:

```
char[] data = {'A', 'n', 'j', 'a'};  // Array von Zeichen
String name = new String ("Anja");
String gleicherName = new String (data);
```

Für mit `new` erzeugte Strings wird immer ein neues `String`-Objekt im Heap erzeugt.

Hierfür das Speicherabbild:

Bild 6-25 Heap nach der Erzeugung zweier Strings mit `new`

- **Implizites Erzeugen eines Objektes vom Typ `String`**

  Strings können – wie Arrays – auch ohne expliziten Aufruf des `new`-Operators erzeugt werden. Die Erzeugung von Strings erfolgt implizit, wenn im Programm eine konstante Zeichenkette verwendet wird. Allerdings wird dabei nicht immer im Heap ein neues Objekt vom Typ `String` angelegt.

  Bei optimierenden Compilern können Objekte vom Typ `String`, die den gleichen Inhalt haben und implizit erzeugt werden, wiederverwendet werden.

  Durch diese Wiederverwendung werden die `.class`-Dateien kleiner und es ist eine Einsparung von Speicher im Heap möglich.

  Im folgenden Beispiel werden zwei Referenzvariablen `name` und `gleicherName` vom Typ `String` angelegt und mit `"Anja"` initialisiert:

```
String name = "Anja";
String gleicherName = "Anja";
```

  Der Compiler kann beiden Referenzvariablen die Adresse auf dasselbe Objekt vom Typ `String` zuweisen. Dies zeigt das folgende Bild:

Bild 6-26 Heap nach der impliziten Erzeugung zweier Strings

Da die `.class`-Datei kleiner wird, ist sie für eine eventuelle Übertragung über ein Netzwerk optimiert. Nicht nur der Compiler kann Optimierungen durchführen, auch der Interpreter hat die Möglichkeit, eine ähnliche Speicherplatzoptimierung durchzuführen.

Während der **Compiler** die **Optimierung einer einzelnen `.class`-Datei** durchführt, kann der **Interpreter zur Laufzeit klassenübergreifend optimieren**.

Werden in zwei unterschiedlichen Klassen die gleichen konstanten Zeichenketten verwendet, hat der Compiler keine Möglichkeit zu optimieren. Dagegen kann der Interpreter beim Laden einer Klasse prüfen, ob schon ein Objekt vom Typ `String` mit dieser konstanten Zeichenkette existiert. Existiert schon ein Objekt vom Typ `String` mit dem Inhalt der konstanten Zeichenkette, muss kein neues Objekt angelegt werden.

6.7.1.2 Vergleichen von Strings

Wenn zwei Referenzvariablen, die auf ein Objekt vom Typ `String` zeigen, mit **Hilfe des Operators** `==` verglichen werden, so werden ihre Werte, d. h. die **Referenzen**, **verglichen**.

Mit anderen Worten, es wird verglichen, ob sie auf das gleiche Objekt vom Typ `String` zeigen. Es wird jedoch nicht geprüft, ob der Inhalt der Objekte übereinstimmt. Würden die in Kapitel 6.7.1.1 angesprochenen Optimierungen in jedem Fall greifen, so würde dieser Vergleich tatsächlich funktionieren. Da jedoch die Optimierung zumindest bei mit `new` erzeugten Strings nicht durchgeführt wird und bei implizit erzeugten Objekten vom Typ `String` nicht zwingend vorgeschrieben ist, muss zum Vergleich des Inhalts zweier Objekte vom Typ `String` die **Methode** `equals()` der Klasse `String` verwendet werden.

Zeigen `name1` und `name2` auf Objekte vom Typ `String`, so wird der Vergleich

```
if (name1.equals (name2))
    . . . . .
```

korrekt durchgeführt, ungeachtet dessen, ob die Objekte vom Typ `String` explizit oder implizit erzeugt wurden.

6.7.1.3 Stringverarbeitung – Methoden der Klasse `String`

Da eine konstante Zeichenkette ein Objekt der Klasse `String` ist, können auch alle Methoden dieser Klasse angewendet werden. Zu beachten ist:

Jede Methode der Klasse `String`, die eine **Veränderung einer Zeichenkette** zur Folge hat, z. B. der Aufruf der Methode `substring()`, liefert ein **neues Objekt** der Klasse `String` zurück und nicht das gleiche Objekt mit einem geänderten Inhalt. **Objekte vom Typ `String` können nicht geändert werden.**

Die Klasse `String` ist im Paket `java.lang` definiert. Eine ausführliche Beschreibung aller Methoden kann in der Dokumentation der Java-API (siehe Kapitel 3.3.2) gefunden werden. Im Folgenden sind kurz die gebräuchlichsten Methoden und ihre Aufgaben aufgezählt:

- `public int length()`

  Die Methode `length()` gibt die Anzahl der Zeichen einer Zeichenkette zurück.

- `public boolean equals (Object obj)`

  Diese Methode vergleicht zwei Zeichenketten miteinander und gibt `true` zurück, falls die Zeichenketten den gleichen Inhalt haben. Ansonsten wird `false` zurückgegeben.

Die Methode `equals()`, die in der Klasse `String` implementiert ist, unterscheidet sich von der Methode `equals()` der Klasse `Object`. Die Methode `equals()` der Klasse `Object` überprüft nur, ob die beiden Referenzen, die am Vergleich beteiligt sind, auf das gleiche Objekt zeigen. Bei zwei unterschiedlichen Objekten mit gleichem Inhalt gibt diese Methode immer `false` zurück.

- `public String substring (int anfang, int ende)`

  Diese Methode schneidet eine Zeichenkette zwischen `anfang` und `ende - 1` aus und gibt den ausgeschnittenen Teil als neues Objekt vom Typ `String` zurück. Beachten Sie, dass das erste Zeichen den Index `0` hat. Ist der Wert von `anfang` negativ oder geht der Wert von `ende` über die tatsächliche Länge hinaus, so wird eine Ausnahme (Exception) vom Typ `StringIndexOutOfBounds-Exception`[109] geworfen.

- `public String trim()`

  Die Methode `trim()` entfernt alle Leerzeichen am Anfang und am Ende der Zeichenkette und gibt den bearbeiteten String als neues Objekt vom Typ `String` zurück.

Im folgenden **Beispiel** werden diese Methoden verwendet:

```
// Datei: Zeichenkette.java
public class Zeichenkette
{
   public static void main (String[] args)
   {
      String buchtitel = "Java als erste Programmiersprache";
      String buchtitelAnfang;
      System.out.println (buchtitel);
      System.out.println ("Anzahl der Zeichen des Buchtitels: "
                          + buchtitel.length());

      // Zuweisung eines Teilstrings an buchtitelAnfang
      buchtitelAnfang = buchtitel.substring (0, 5);
      System.out.println ("Anzahl der Zeichen des Buchtitel"
              + "anfangs vor trim(): " + buchtitelAnfang.length());

      // Entfernen der Leerzeichen von beiden Enden des Strings
      buchtitelAnfang = buchtitelAnfang.trim();
      System.out.println ("Anzahl der Zeichen des Buchtitel"
              + "anfangs nach trim(): " + buchtitelAnfang.length());
      if (buchtitelAnfang.equals ("Java"))
      {
         System.out.println ("Buchtitel fängt mit Java an");
      }
   }
}
```

[109] Exceptions siehe Kapitel 13

Die Ausgabe des Programms ist:

```
Java als erste Programmiersprache
Anzahl der Zeichen des Buchtitels: 33
Anzahl der Zeichen des Buchtitelanfangs vor trim(): 5
Anzahl der Zeichen des Buchtitelanfangs nach trim(): 4
Buchtitel fängt mit Java an
```

6.7.2 Variable Zeichenketten mit der Klasse `StringBuffer`

Die Klasse `StringBuffer` gehört auch zum Paket `java.lang` und repräsentiert eine Zeichenkette mit den folgenden Eigenschaften:

- Die Länge der Zeichenkette in einem Objekt der Klasse `StringBuffer` ist nicht festgelegt.
- Die Länge vergrößert sich automatisch, wenn in einem Objekt der Klasse `StringBuffer` weitere Zeichen angefügt werden und der vorhandene Platz nicht ausreicht.
- Der Inhalt eines Objektes der Klasse `StringBuffer` lässt sich verändern.

Die Länge der Zeichenkette in einem Objekt vom Typ `StringBuffer` wird – wie auch bei der Klasse `String` – in einem zusätzlichen Datenfeld des Objektes abgelegt.

6.7.2.1 Erzeugung eines Objektes der Klasse `StringBuffer`

Im Gegensatz zur Klasse `String` gibt es bei der Klasse `StringBuffer` **nicht die Möglichkeit, ein Objekt implizit zu erzeugen**. Die Erzeugung ist nur mit dem Operator `new` möglich. Die Initialisierung eines Objektes der Klasse `StringBuffer` erfolgt durch Aufruf des Konstruktors, der im Anschluss an den Operator `new` folgt und durch den Klassennamen, die runden Klammern und ihren Inhalt gegeben ist. Im Folgenden werden die Konstruktoren

```
StringBuffer()
StringBuffer (int length)
StringBuffer (String str)
```

vorgestellt. Dem ersten Konstruktor werden keine Parameter zur Initialisierung übergeben. Es wird ein Objekt der Klasse `StringBuffer` auf dem Heap angelegt, das 16 Zeichen aufnehmen kann. Hierbei wird – für den Programmierer unsichtbar – innerhalb des Konstruktors ein zweites Mal mit dem Operator `new` Speicher auf dem Heap allokiert und die Referenz auf diesen Speicher dem privaten Datenfeld `value` vom Typ `private char[]` des Objektes der Klasse `StringBuffer` zugewiesen:

```
value = new char [16];
```

Ein Objekt der Klasse `StringBuffer`, das mit folgender Anweisung

```
StringBuffer str1 = new StringBuffer();
```

erzeugt wird, ist in folgendem Bild zu sehen:

Bild 6-27 Mit dem parameterlosen Konstruktor initialisiertes Objekt der Klasse `StringBuffer`

Wird als aktueller Parameter dem Konstruktor ein `int`-Wert übergeben, so wird ein Objekt der Klasse `StringBuffer` der Länge des übergebenen `int`-Wertes angelegt. Die Anweisung

```
StringBuffer str2 = new StringBuffer (10);
```

legt also ein Objekt der Klasse `StringBuffer` an, das 10 Zeichen aufnehmen kann. Man beachte, dass auch hier innerhalb des Konstruktors noch einmal vom Operator `new` Gebrauch gemacht wird, um ein `char`-Array der entsprechenden Länge auf dem Heap zu allokieren. Das folgende Bild zeigt das genannte Objekt:

Bild 6-28 Erzeugtes Objekt der Klasse `StringBuffer`*, das 10 Zeichen aufnehmen kann*

Objekte der Klasse `StringBuffer`, die mit dem Konstruktor ohne Parameter bzw. mit dem Konstruktor, der die Länge eines Objektes der Klasse `StringBuffer` entgegennimmt, initialisiert werden, haben noch keine Zeichenkette, die sie beinhalten. Um ein Objekt der Klasse `StringBuffer` nach der Erzeugung mit einer Zeichenkette zu füllen, existieren die Methoden `append()` und `insert()` in unterschiedlichen Ausprägungen in der Java-API. Genauso kann nachträglich der schon bestehende Inhalt eines Objektes der Klasse `StringBuffer` durch Methoden wie `insert()`, `delete()` und `setCharAt()` verändert werden. Der volle Umfang der Methoden der Klasse `StringBuffer` und deren detaillierte Beschreibung kann der Dokumentation der Java-Klassenbibliothek entnommen werden.

Wird als aktueller Parameter des Konstruktors eine konstante Zeichenkette angegeben, so wird damit das Objekt der Klasse `StringBuffer` initialisiert. Mit der Codezeile

```
StringBuffer name = new StringBuffer ("Anja");
```

wird also ein Objekt der Klasse `StringBuffer` auf dem Heap angelegt und zusätzlich mit der konstanten Zeichenkette `"Anja"` initialisiert. Dies ist im folgenden Bild dargestellt:

Bild 6-29 `StringBuffer`-Objekt und `String`-Objekt mit `"Anja"` initialisiert

Auch hier wird innerhalb des Konstruktors vom Operator `new` Gebrauch gemacht und ein entsprechendes `char`-Array auf dem Heap angelegt, das mit den Zeichen `'A' 'n' 'j' 'a'` initialisiert wird. Dabei wird in das private Datenfeld `count` die Länge der Zeichenkette `"Anja"`, d. h. die Zahl 4, eingetragen.

Da bei der Erzeugung des Objektes der Klasse `StringBuffer` eine Referenz auf ein Objekt vom Typ `String` übergeben wird, kommt die Zeichenkette `"Anja"` zweimal im Heap vor, einmal als Objekt vom Typ `String` und einmal als Objekt vom Typ `StringBuffer`. Die Darstellung eines Strings im Arbeitsspeicher wurde bis Bild 6-29 noch nicht vorgestellt.

Obwohl Objekte vom Typ `String` konstant und Objekte vom Typ `StringBuffer` variabel sind, sind ihre Datenfelder gleich, allerdings sind ihre Methoden verschieden.

Bei Objekten vom Typ `StringBuffer` wird **in keinem Fall** eine **Speicherplatzoptimierung** wie bei Objekten vom Typ `String` vorgenommen.

Es wird also stets ein neues Objekt vom Typ `StringBuffer` im Heap angelegt.

6.7.2.2 Vergleichen von Objekten vom Typ `StringBuffer`

Da für jede Zeichenkette ein neues Objekt der Klasse `StringBuffer` erzeugt wird, sollte man meinen, dass auch in diesem Fall ein Vergleich zweier Zeichenketten mit der Methode `equals()` erfolgt.

Die Klasse `StringBuffer` erbt zwar wie jede Klasse die Methode `equals()` von der Klasse `Object`, allerdings wird sie jedoch nicht wie bei der Klasse `String` überschrieben und kann daher nicht sinnvoll eingesetzt werden.

Wird die Methode `equals()` für den Vergleich von Objekten der Klasse `StringBuffer` verwendet, so liefert sie ein unbrauchbares Ergebnis.

Ein Vergleich der Zeichenketten zweier Objekte vom Typ `StringBuffer` ist nur über den Umweg der Konvertierung beider Objekte in zwei Objekte vom Typ `String` möglich.

Die Konvertierung von Objekten vom Typ `StringBuffer` in Objekte vom Typ `String` erfolgt mit der Methode `toString()`. Dies wird in folgendem **Beispiel** vorgestellt:

```
StringBuffer name1 = new StringBuffer ("Anja");
StringBuffer name2 = new StringBuffer ("Peter");
String name1String = name1.toString();
String name2String = name2.toString();
if (name1String.equals (name2String)). . . . .
```

6.7.3 Verkettung von Objekten der Klassen `String` und `StringBuffer`

Die Verkettung von Zeichenketten aus Objekten der Klassen `String` und `StringBuffer` wird in diesem Kapitel zusammengefasst, da die Verkettung von Objekten der Klasse `String` auf der Verkettung von Objekten der Klasse `StringBuffer` aufsetzt. Kapitel 6.7.3.1 zeigt das Anhängen von Zeichenketten an ein Objekt der Klasse `StringBuffer`. Kapitel 6.7.3.2 diskutiert die Verkettung von Objekten der Klasse `StringBuffer` und Kapitel 6.7.3.3 die Verkettung von Strings.

6.7.3.1 Anhängen von Zeichenketten an ein Objekt der Klasse `StringBuffer`

Das Anhängen einer konstanten Zeichenkette an ein Objekt der Klasse `StringBuffer` erfolgt mit der Methode `append()` der Klasse `StringBuffer` wie in folgendem **Beispiel**:

```
StringBuffer name = new StringBuffer ("Anja");
name.append (" Christina");
```

Das folgende Bild zeigt diese Situation:

Bild 6-30 Anhängen einer Zeichenkette

Die beiden Objekte `"Anja"` und `" Christina"` der Klasse `String` werden, sofern sie nicht von anderen Stellen in der Java-Virtuellen Maschine referenziert werden, zur gegebenen Zeit durch den Garbage Collector entfernt.

6.7.3.2 Verkettung von Objekten der Klasse `StringBuffer`

Die Verkettung von Objekten der Klasse `StringBuffer` erfolgt analog zu Kapitel 6.7.3.1, wie im folgenden **Beispiel** gezeigt wird:

```
StringBuffer name1 = new StringBuffer ("Anja");
StringBuffer name1 = new StringBuffer (" Christina");
name1.append (name2);
```

6.7.3.3 Verkettung von Objekten der Klasse `String`

Zur Verkettung von Objekten der Klasse `String` gibt es in Java den Operator `+` als **Verkettungsoperator**. Da Objekte der Klasse `String` unveränderlich sind, wird hierbei ein neues `String`-Objekt erzeugt, welches die neue, verkettete Zeichenkette aufnimmt. Dies ist im folgenden **Beispiel** zu sehen:

```
String name = "Anja";
name = name + " Christina";
```

Lassen Sie sich hier nicht verblüffen. Objekte vom Typ `String` können tatsächlich nicht verändert werden. Mit `name + " Christina"` wird ein neues Objekt vom Typ `String` geschaffen. Mit `name = name + " Christina"` wird die Referenzvariable `name` auf das neu erzeugte Objekt gerichtet.

Der Operator `+` wird dabei vom Compiler in einen `append()`-Aufruf der Klasse `StringBuffer` übersetzt. Die zwei Codezeilen des vorherigen **Beispiels** werden dabei sinngemäß in die folgenden Anweisungen übersetzt:

```
String name = "Anja";
StringBuffer b = new StringBuffer (name);
b.append (" Christina");
name = b.toString();
```

Im Heap werden mit diesen Anweisungen die in Bild 6-31 gezeigten `String`- und `StringBuffer`-Objekte angelegt:

Bild 6-31 Zeichenketten im Heap vor und nach der Verkettung

Es werden also zwei Objekte vom Typ `String` erzeugt und miteinander über Umwege zu einem neuen Objekt vom Typ `String` verkettet. Die `String`-Objekte `"Christina"` und `"Anja"` sowie das `StringBuffer`-Objekt `"Anja Christina"` können, wenn sie nicht mehr durch eine andere Referenzvariable referenziert werden, vom Garbage Collector entfernt werden. Dieses Vorgehen ist, da die Erzeugung von Objekten viel Rechenzeit verbraucht, nicht besonders effizient. **In Schleifen sollte** deshalb **die Verkettung von Strings vermieden werden**. Stattdessen sollten besser Objekte der Klasse `StringBuffer` verwendet werden. Das folgende **Beispiel** zeigt, wie in einer `for`-Schleife der bestehende String jedes Mal um ein Sternchen erweitert wird:

```
String s = "*";

for (int i = 0; i < 5000; i++)
{
   s = s + "*";  // Dies sollte vermieden werden
}
```

Für das vorangehende **Beispiel** empfiehlt sich daher die folgende Optimierung durch die Verwendung von Objekten der Klasse `StringBuffer`:

```
String s = "*";
StringBuffer tmp = new StringBuffer (s);

for (int i = 0; i < 5000; i++)
{
   tmp.append ("*");  // So ist es besser
}

s = tmp.toString();
```

Durch dieses Vorgehen ergibt sich eine Geschwindigkeitserhöhung, da das ineffiziente Erzeugen von Objekten in der Schleife vermieden wird.

6.7.4 Variable Zeichenketten mit der Klasse `StringBuilder`

Seit Java 5 stellt das Paket `java.lang` auch eine Klasse `StringBuilder` zur Verfügung. Bisher war die Klasse `StringBuffer` die einzige Möglichkeit, um Zeichenketten zu verändern.

Die **Klasse `StringBuilder`** ist eine neue Klasse mit einer **höheren Performanz**, die jedoch **nicht verwendet** werden darf, wenn eine Zeichenkette von mehreren **Threads** parallel bearbeitet wird.

Threads werden in Kapitel 16 beschrieben.

Die Klassen `StringBuffer` und `StringBuilder` werden hauptsächlich dazu verwendet, Zeichenketten aufzubauen oder abzuändern. Bestehende Zeichenketten, die in einem Objekt der Klasse `StringBuilder` oder `StringBuffer` gespeichert sind, werden dabei durch Anfügen oder Einfügen von neuen Zeichen oder Abändern von bestehenden Zeichen modifiziert.

Zu diesem Zweck bietet die Klasse `StringBuilder` die Methoden `append()`, `insert()` und `setCharAt()` an. Diese Methoden sind überladen, sodass sie Parameter verschiedener Datentypen entgegennehmen können. Die übergebenen Parameter werden gegebenenfalls in Strings konvertiert. Die Methode `append()` hängt die String-Repräsentation ihres Parameters an das Ende einer Zeichenkette an. Im Gegensatz dazu wird der Parameter der Methode `insert()` an einer definierbaren Stelle in eine Zeichenkette eingefügt. Mit der Methode `setCharAt()` kann an einer bestimmten Stelle in der Zeichenkette ein Zeichen verändert werden. Die Methode erwartet als ersten Parameter einen `int`-Wert, der die Stelle in der Zeichenkette angibt, und als zweiten Parameter das neu zu setzende Zeichen vom Typ `char`.

Hierzu ein **Beispiel**:

```
// Datei: StringBuilderTest.java

public class StringBuilderTest
{
   public static void main (String[] args)
   {
      StringBuilder sb = new StringBuilder ("Wilhelm Röntgen");
      System.out.println (sb);

      sb.insert (7, " Konrad");
      sb.append (", Matrikelnummer: ");
      sb.append (123456);
      System.out.println (sb);
      System.out.println();

      // Ein neues StringBuilder-Objekt erzeugen
      sb = new StringBuilder ("Tasse");
      System.out.println (sb);

      sb.setCharAt (0, 'K');
      System.out.println (sb);
```

```
        sb.setCharAt (0, 'M');
        System.out.println (sb);
    }
}
```

Die Ausgabe des Programms ist:

```
Wilhelm Röntgen
Wilhelm Konrad Röntgen, Matrikelnummer: 123456

Tasse
Kasse
Masse
```

Die Konstruktoren und Methoden der Klasse `StringBuilder` entsprechen den Konstruktoren und Methoden der Klasse `StringBuffer` (siehe Kapitel 6.7.2).

6.8 Wandlung von Datentypen

In diesem Kapitel steht im Fokus:

- die Wandlung von Variablen eines einfachen Datentyps in Variablen eines Klassen-Typs und umgekehrt
- sowie die Wandlung von Variablen beliebiger Datentypen in Zeichenketten und umgekehrt.

Für die Wandlung von Variablen eines einfachen Datentyps in Variablen eines Klassen-Typs werden Ihnen Wrapper-Klassen sowie die Bedeutung von Boxing und Unboxing in Kapitel 6.8.1 vorgestellt. Die Konvertierung zwischen Strings und anderen Datentypen ist systematisch in Kapitel 6.8.2 zusammengestellt.

Typkonvertierungen

- einfacher Datentypen in andere einfache Datentypen
- sowie Typkonvertierungen von Referenztypen in andere Referenztypen

werden gesondert in den Kapiteln 7.7 und 11.3.1 behandelt.

6.8.1 Wrapper-Klassen mit Boxing und Unboxing

Wrapper-Klassen sind Klassen, die dazu dienen, eine **Anweisungsfolge** oder einen **einfachen Datentyp** in die **Gestalt einer Klasse** zu bringen.

Wrapper-Klassen betten ein nicht-objektorientiertes Konstrukt in die Form einer Klasse ein.

Wrapper-Klassen, die nur die Methode `main()` enthalten, wurden in Kapitel 4 vorgestellt. In diesem Kapitel werden Wrapper-Klassen, die geschaffen wurden, um einfache Datentypen aufzunehmen, betrachtet. Für alle einfachen Datentypen gibt es in Java im Paket `java.lang` die folgenden Wrapperklassen:

Einfache Datentypen	Wrapper-Klassen
`char`	`Character`
`boolean`	`Boolean`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`double`	`Double`
`float`	`Float`

Tabelle 6-4 Einfache Datentypen und die zugehörigen Wrapper-Klassen

Darüber hinaus gibt auch eine **Wrapper-Klasse `Void`**, obwohl `void` in Java kein Datentyp ist. Da `void` eine leere Menge bedeutet und keinerlei Informationen enthält, besitzt die Klasse `Void` weder Konstruktoren noch Methoden.

6.8.1.1 Methoden der Wrapper-Klassen

Die in Tabelle 6-4 aufgeführten Wrapper-Klassen stellen Methoden zur Bearbeitung der entsprechenden einfachen Datentypen bereit. **Beispiele** hierfür sind Methoden zum **Umwandeln von Zahlen in Strings** und **von Strings in Zahlen** (siehe hierzu auch Kapitel 6.8.2). Viele der vorhandenen Methoden sind `static`, sodass sie auch benutzt werden können, ohne dass ein Objekt einer Wrapper-Klasse gebildet werden muss.

Wandlung eines Strings in einen einfachen Datentyp

Die Wrapper-Klassen `Float`, `Double`, `Byte`, `Short`, `Integer` und `Long` stellen Klassenmethoden bereit, um ein Objekt vom Typ `String` in die einfachen Datentypen `float`, `double`, `byte`, `short`, `int` und `long` zu wandeln.

So stellt die Wrapper-Klasse `Integer` die Klassenmethode **`parseInt()`** zur Verfügung, die den Inhalt eines Objektes vom Typ `String` als `int`-Wert zurückgibt.

Beispiel zur Wandlung eines Strings in einen einfachen Datentyp

Das folgende **Beispielprogramm** zeigt, wie aus einem `String`-Objekt mit Hilfe der Klassenmethode `parseInt()` der Wrapper-Klasse `Integer` eine Wandlung eines `String`-Objektes in den Datentyp `int` durchgeführt werden kann:

```
// Datei: WrapperTest.java

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

```
    {
        String s1 = "100";
        System.out.println ("100 im Dezimalsystem hat den Wert: "
                           + Integer.parseInt (s1));
        System.out.println ("100 im Oktalsystem hat den Wert: "
                           + Integer.parseInt (s1, 8));
        System.out.println ("100 im Hexadezimalsystem hat den Wert: "
                           + Integer.parseInt (s1, 16));
    }
}
```

Die Ausgabe des Programms ist:

```
100 im Dezimalsystem hat den Wert: 100
100 im Oktalsystem hat den Wert: 64
100 im Hexadezimalsystem hat den Wert: 256
```

Einpacken von einfachen Datentypen in Objekte von Wrapper-Klassen

Die Werte von einfachen Datentypen können in Objekte von Wrapper-Klassen verpackt werden. Entsprechend gibt es auch eine Möglichkeit, Werte von einfachen Datentypen, die in Objekten von Wrapper-Klassen verpackt sind, wieder auszupacken.

Um einen Ausdruck eines einfachen Datentyps in ein Objekt vom Typ einer Wrapper-Klasse zu verpacken, kann entweder der `new`-Operator oder die Klassenmethode **`valueOf()`** der entsprechenden Wrapper-Klasse verwendet werden.

Die folgenden Zeilen zeigen **Beispiele** für die beiden Alternativen:

```
Integer j = new Integer (2);      // Äquivalente Alternativen
Integer i = Integer.valueOf (2); // zur Objekterzeugung
```

Auspacken von einfachen Datentypen aus Objekten von Wrapperklassen

Jede Wrapper-Klasse besitzt eine Klassenmethode `valueOf()`, die eine Referenz auf ein Objekt vom Typ der jeweiligen Wrapper-Klasse zurückgibt. Zum anderen kann der `new`-Operator verwendet werden.

Um einen Wert wieder aus einem Objekt einer Wrapper-Klasse "auszupacken", stellen Wrapper-Klassen entsprechende Methoden bereit. Im Falle der Wrapper-Klasse `Integer` ist dies die Methode `intValue()`, die den einfachen Datentyp `int` als Rückgabewert liefert.

Beispiel zum "Verpacken" und "Auspacken"

Das folgende **Beispielprogramm** zeigt das "Verpacken" und das "Auspacken". In diesem Programm sind beide Möglichkeiten für das "Verpacken" zu sehen. Mit den

"ausgepackten" Werten der einfachen Datentypen werden dann Berechnungen durchgeführt. Hier das erwähnte Programm:

```
// Datei: Wrapper.java

public class Wrapper
{
   public static void main (String[] args)
   {
      Integer i1 = new Integer (1);
      Integer i2 = Integer.valueOf (2);
      Double d = new Double (2.2);

      int summe = i1.intValue() + i2.intValue();
      double produkt = d.doubleValue() * 2;

      System.out.println ("Wert der Variablen summe: " + summe);
      System.out.println ("Wert der Variablen produkt: " + produkt);
   }
}
```

Die Ausgabe des Programms ist:

```
Wert der Variablen summe: 3
Wert der Variablen produkt: 4.4
```

6.8.1.2 Boxing und Unboxing

Vor der Version Java 5 war es nicht möglich, mit Objekten von Wrapper-Klassen zu rechnen wie mit den Variablen einfacher Datentypen. Auf Variablen einfacher Datentypen kann man zum Beispiel die Operatoren + und − anwenden. Früher musste man, um mit den Werten, die in Objekten von Wrapper-Klassen gespeichert sind, rechnen zu können, diese wie im vorangegangenen Beispielprogramm zuerst "auspacken".

Seit Java 5 wird das **"Verpacken" und "Auspacken" automatisch** durch den Compiler durchgeführt.

Das **"Verpacken"** bezeichnet man als **Boxing**, das **"Auspacken"** als **Unboxing**.

Boxing (in eine Schachtel packen) bedeutet, dass ein Wert eines einfachen Typs in ein Objekt einer Wrapper-Klasse umgewandelt wird. Bildlich gesehen stellt ein Objekt einer Wrapper-Klasse eine Box dar, in welche die Variable eines einfachen Typs hineingelegt wird.

Unboxing ist genau das Gegenteil von Boxing. Bildlich gesehen wird hier die Variable eines einfachen Typs aus der Schachtel (dem Objekt der Wrapper-Klasse) wieder herausgenommen.

Da der ganze Mechanismus von **Boxing** und **Unboxing** vom Compiler automatisch umgesetzt wird, spricht man auch von **Auto-Boxing** und **Auto-Unboxing**. Durch das Auto-Boxing wird es auch möglich, fast alle Operatoren, die bisher nur auf einfache numerische Datentypen anwendbar waren, auch für die entsprechenden Wrapper-Klassen zu verwenden.

Die folgende Tabelle zeigt **Beispiele für Boxing und Unboxing** jeweils manuell und automatisch für den einfachen Datentyp `int` und die entsprechende Wrapper-Klasse `Integer`. Dabei ist die Variable `i` vom Typ `int` und die Variable `wi` vom Typ `Integer`. Hier die Tabelle:

Boxing		**Unboxing**	
Manuell	Automatisch	Manuell	Automatisch
`Integer wi = new Integer(i);`	`Integer wi = i;`	`int i = wi.intValue();`	`int i = wi;`

Tabelle 6-5 Beispiele für Boxing und Unboxing

Das folgende Programm zeigt, dass beispielsweise die Werte zweier Objekte der Wrapper-Klasse `Integer` mit dem Operator `+` addiert werden können, ohne sie vorher manuell auszupacken.

Programmbeispiel zur Addition zweier Objekte der Wrapper-Klasse `Integer`:

```
// Datei: AutoBoxing2.java

public class AutoBoxing2
{
   public static void main (String[] args)
   {
      // Anlegen von zwei Objekten der Wrapper-Klasse Integer
      // und Zuweisen von Werten eines einfachen Typs (der Compiler
      // fügt Boxing-Code ein).
      Integer i1 = 10;
      Integer i2 = 5;

      // Auto-Unboxing von i1 und i2, Addieren der Werte, danach
      // Auto-Boxing des Wertes für die Zuweisung an i3.
      Integer i3 = i1 + i2;

      // Ausgeben der einzelnen Werte und des Ergebnisses
      System.out.println (i1 + " + " + i2 + " = " + i3);
   }
}
```

Die Ausgabe des Programms ist:

```
10 + 5 = 15
```

6.8.1.3 Operatoren und Wrapper-Klassen

Durch das Auto-Unboxing können Referenzvariablen auf Objekte von Wrapper-Klassen numerischer Datentypen auch als Operanden der Postfix- und Präfix-Operatoren `++` und `--` verwendet werden, genauso wie als Operanden der binären arithmetischen Rechenoperationen (`+`, `-`, `/`, `*`, `%`) oder der bitweisen Operationen (`&`, `^`, `|`, `>>`, `<<`, `>>>`).

Referenzvariablen auf Objekte der Wrapper-Klasse `Boolean` können Operanden von bestimmten relationalen Operationen werden oder in die Bedingung des Bedingungsoperators (siehe Kapitel 7.6.7) eingesetzt werden. Genauso können die logischen Operatoren nicht nur auf boolesche Ausdrücke, sondern auch auf Referenzvariablen, die auf Objekte vom Typ der Wrapper-Klasse `Boolean` zeigen, angewendet werden.

Zahlreiche Operatoren, die auf Variablen einfacher Datentypen angewandt werden können, können auch auf Referenzvariablen, die auf Objekte von Wrapper-Klassen zeigen, angewandt werden.

Werden die relationalen Operatoren `==` und `!=` bei Referenzvariablen, die auf Objekte von Wrapper-Klassen zeigen, verwendet, so werden aber – wie bei Referenzvariablen üblich – die Referenzen verglichen und nicht die Inhalte!

6.8.1.4 Beispielprogramm für Boxing und Unboxing

Das folgende Beispiel zeigt die Verwendung von Wrapper-Klassen, wenn an eine **Methode Referenzen** als Parameter übergeben werden müssen, der Aufrufer aber einen **einfachen Zahlenwert** übergeben möchte. Die Methode `push()` der folgenden Klasse `EasyStack` erwartet als Übergabeparameter eine Referenzvariable der Wrapper-Klasse `Integer`. Beim Aufruf der Methode `push()` wird allerdings eine Variable des einfachen Datentyps `int` übergeben. Der Compiler fügt automatisch den Boxing-Code ein. Entsprechend gibt die Methode `pop()` eine Referenz vom Typ der Wrapper-Klasse `Integer` zurück. Die zurückgegebene Referenz wird automatisch in den einfachen Datentyp `int` gewandelt. Hier das bereits erwähnte Beispiel:

```
// Datei: EasyStack.java

public class EasyStack
{
   // Anlegen eines Integer-Arrays, das 3 Elemente aufnehmen kann.
   private Integer[] stack = new Integer [3];
   private int top = -1;

   public void push (Integer i) // Methode zum Ablegen eines Elemen-
   {                            // tes auf dem Stack.
      top = top + 1;
      stack [top] = i;
   }

   public Integer pop()         // Methode, um ein Element vom Stack
   {                            // abzuholen.
      top = top - 1;
      return stack [top + 1];
   }
}

// Datei: EasyStackTest.java

public class EasyStackTest
{
   public static void main (String[] args)
   {
      // Ein Objekt der Klasse EasyStack erzeugen.
      EasyStack arr = new EasyStack();

      arr.push (1);           // Stack befüllen. Der Compiler
      arr.push (2);           // fügt automatisch den Boxing-Code
      arr.push (3);           // ein.

      int wert1 = arr.pop();  // Daten vom Stack abholen.
      int wert2 = arr.pop();  // Der Compiler fügt automatisch
      int wert3 = arr.pop();  // Unboxing-Code ein.

      // Daten ausgeben
      System.out.println ("Wert 1. Element: " + wert1);
      System.out.println ("Wert 2. Element: " + wert2);
      System.out.println ("Wert 3. Element: " + wert3);
   }
}
```

Die Ausgabe des Programms ist:

```
Wert 1. Element: 3
Wert 2. Element: 2
Wert 3. Element: 1
```

6.8.1.5 Vor- und Nachteile von Wrapper-Klassen

Die Stärke der Wrapper-Klassen in der Kombination mit dem automatischen Boxing und Unboxing liegt darin, dass man Variablen einfacher Datentypen an Methoden übergeben kann, die als Übergabeparameter einen Referenztyp erwarten.

Das Auto-Boxing kann jedoch auch Probleme verursachen: Existieren beispielsweise in einer Klasse zwei überladene Methoden[110], deren Methodenköpfe sich nur dadurch unterscheiden, dass einmal einfache Typen verwendet werden und einmal die entsprechenden Wrapper-Klassen, ist es für den Compiler nicht klar, welche Methode aufgerufen werden soll. Dies wird durch einen Fehler beim Kompilieren angezeigt. Das folgende **Beispiel** zeigt diese Situation:

```
public void testMethode (int x)
{
   ... // nicht relevant
}
public void testMethode (Integer x)
{
   ... // nicht relevant
}

// Welche Methode soll der Compiler aufrufen? Durch Auto-
// Boxing ist der folgende Aufruf nicht mehr eindeutig.

testMethode (1)
```

Solche Mehrdeutigkeiten treten vor allem in einem Code auf, der vor der Version Java 5 geschrieben wurde, da dort noch kein Auto-Boxing enthalten war und sich viele Entwickler mit entsprechenden überladenen Methoden behalfen. Durch eine explizite Typkonvertierung beim Aufruf kann dieses Problem umgangen werden.

Bei der Deklaration von überladenen Methoden muss darauf geachtet werden, dass keine Mehrdeutigkeiten auftreten. Beim Überladen einer Methode sollte kein einfacher Typ in der Parameterliste durch einen Typ der Wrapper-Klasse ersetzt werden (und umgekehrt).

Auch bei Kontrollflusskonstrukten (siehe Kapitel 8.2 und 8.3) kann dieser neue Mechanismus hilfreich sein. Es besteht nun auch die Möglichkeit, bei `if`-Anweisungen als Ausdruck eine Referenzvariable der Klasse `Boolean` zu verwenden. Gleiches gilt ebenfalls für `while`-, `do-while`- oder `for`-Schleifen, die nun in ihrem Ausdruck auch eine Referenzvariable der Klasse `Boolean` akzeptieren.

Der Ausdruck der `switch`-Anweisung (siehe Kapitel 8.2.3) kann nicht nur wie früher vom Typ `char`, `byte`, `short`, `int` sein, sondern inzwischen auch vom Typ `String`, `Character`, `Byte`, `Short` oder `Integer`.

[110] Was das Überladen einer Methode genau bedeutet, wird in Kapitel 9.4 noch ausführlich behandelt.

6.8.2 Konvertierung zwischen Strings und anderen Datentypen

Kapitel 6.8.2.1 stellt die Konvertierung von primitiven Typen in Strings und Kapitel 6.8.2.2 die Konvertierung von Referenztypen in Strings systematisch dar. Kapitel 6.8.2.3 beschreibt die implizite Konvertierung bei Verwendung der Methode `println()` und Kapitel 6.8.2.4 erörtert das Konvertieren von Strings in primitive Typen.

6.8.2.1 Konvertieren von Referenztypen in Strings

Die statischen, überladenen Methoden `String.valueOf()` liefern die String-Darstellung von primitiven Werten.

Die Methode `valueOf()` akzeptiert als Übergabeparameter die verschiedenen primitiven Typen (siehe Tabelle 6-6). Innerhalb der Methode `valueOf()` wird auf der dem Übergabeparameter zugehörigen Wrapper-Klasse die statische Methode `toString()` aufgerufen.

Das folgende **Beispiel** veranschaulicht, wie die **Konvertierung in der Methode `value-Of()`** für einen Übergabeparameter vom Typ `double` durchgeführt wird:

Zuerst wird die Methode `valueOf()` der Klasse `String` betrachtet:

```
public static String valueOf (double d)
{
   return Double.toString(d);
}
```

und dann die Methode `toString()` der Klasse `Double`:

```
public static String toString(double d)
{
   return new FloatingDecimal(d).toJavaFormatString();
}
```

Auf der Wrapper-Klasse `Double` wird also die statische Methode `toString()` aufgerufen.

Dies betrifft jedoch nicht die beiden primitiven Typen `char` und `boolean`. Diese bilden eine **Ausnahme**, da sie unmittelbar von der `String.valueOf()`-Methode in den entsprechenden String gewandelt werden. Zwar besitzen beide Typen eine entsprechende Wrapper-Klasse, welche die statische Methode `toString()` definiert, allerdings wird diese von der statischen Methode `String.valueOf()` nicht verwendet.

Das folgende **Beispiel** beschreibt den gewöhnlichen Fall und dient zur Verdeutlichung:

```
int i = 30;
String s = String.valueOf (i) // ergibt 30 als Zeichenkette
```

Innerhalb der statischen Methode `String.valueOf (int)` wird die statische Methode `Integer.toString (int)` mit dem Parameter `i` aufgerufen. Das Resultat wird der Variablen `s` dann zugewiesen.

Bis auf die erwähnten Ausnahmen verhalten sich alle primitiven Datentypen analog. Der Unterschied liegt lediglich in den entsprechenden Varianten der Methoden `valueOf()` und `toString()`.

Die folgende Tabelle gibt einen Überblick über die primitiven Datentypen, die der überladenen Methode `valueOf()` übergeben werden können, und den darin aufgerufenen `toString()`-Methoden der jeweiligen Wrapperklassen:

primitiver Datentyp	zugehörige Wrapper-Klasse	Zugehörige überladene Methode
boolean	Boolean	public static String toString (boolean)
char	Character	public static String toString (char)
int, byte, short	Integer	public static String toString (int)
long	Long	public static String toString (long)
float	Float	public static String toString (float)
double	Double	public static String toString (double)

Tabelle 6-6 Primitive Typen, Wrapper-Klassen und Stringkonvertierungsroutinen

6.8.2.2 Konvertieren von Referenztypen in Strings

Die Wandlung von Referenztypen in Strings wird von der statischen Methode `String.valueOf (Object obj)` durchgeführt. Innerhalb dieser Methode wird die Methode `toString()` des übergebenen Objekts aufgerufen. Die Methode `toString()` der Klasse `Object` muss überschrieben werden, sofern eine spezielle Darstellung gewünscht ist. Andernfalls wird immer die Implementierung der `toString()`-Methode der Klasse `Object` aufgrund der Vererbung verwendet. Diese gibt den Namen der Klasse, das @-Zeichen und den Hashwert[111] des Objektes aus, für das die Methode aufgerufen wurde.

Das nachfolgende **Beispiel** zeigt die Konvertierung eines Referenztyps der Klasse `Date` der Java-Standardbibliothek:

```
Date d = new Date();
String s = String.valueOf (d);
// s = "Tue Aug 06 10:11:54 CET 2013"
```

[111] Der Hashwert ist das Ergebnis einer Einwegfunktion, die eine beliebige Eingangsgröße aus einer Menge M1 in eine Ausgangsgröße einer Menge M2 abbildet.

Die Klasse `Date` hat – wie das Beispiel zeigt – eine überschriebene `toString()`-Methode.

6.8.2.3 Implizite Konvertierung bei der Methode `println()`

In Java existieren Mechanismen, die es ermöglichen, Variablen beliebiger Datentypen in Zeichenketten zu wandeln und auszugeben. Bei vielen gezeigten Programmausgaben wurde dieser Mechanismus bereits verwendet, ohne dass näher darauf eingegangen wurde. Dies soll an dieser Stelle nachgeholt werden. Hierfür müssen zuerst die Methoden `print()` und `println()` der Klasse `PrintStream` erläutert werden. Beide Methoden sind mehrfach überladen, um alle primitiven Datentypen und die beiden Referenztypen `Object` und `String` bedienen zu können. Die Methode `print()` ist für die eigentliche Ausgabe eines Inhaltes (Übergabeparameter) auf dem jeweiligen Ausgabestream zuständig. Für die Umwandlung übergebener Parameter in eine Zeichenkette greift die Methode `print()` auf die bereits bekannte statische Methode `String.valueOf()` zu. Die Methode `println()` ruft intern ebenfalls die Methode `print()` auf, geht aber anschließend in eine neue Zeile. Hierfür wird die private Methode `newLine()` aufgerufen. Die Methoden `write()` und `newLine()` innerhalb der `println()`-Methode bilden eine atomare Operation, d. h. sie werden immer gemeinsam ausgeführt.

Die folgenden beiden Programmzeilen dienen der Ausgabe des Wertes eines primitiven Datentyps:

```
int x = 42;
System.out.println (x);
```

Diese beiden Programmzeilen verursachen den folgenden Ablauf:

- Die Variable `x` wird an die Methode `println (int i)` der Klasse `PrintStream` übergeben.
- Diese Methode ruft die statische Methode `String.valueOf (int i)` auf. Die Methode `valueOf()` wurde bereits im Kapitel 6.8.2.1 erläutert. Das von der statischen Methode zurückgelieferte Objekt vom Typ `String` wird an die Methode `print()` weitergeleitet und innerhalb dieser Methode an die private `write()`-Methode delegiert. Diese Methode sorgt für die Darstellung der Zahl auf der Standardausgabe. Abschließend wird die Zeile mit der privaten Methode `newLine()` terminiert.

Referenztypen verhalten sich im Ablauf ähnlich, rufen allerdings die für Referenztypen überladenen Methoden auf. Im nachfolgenden **Beispiel** ist die Ausgabe eines Referenztyps dargestellt:

```
Date d = new Date();
System.out.println (d);
```

6.8.2.4 Konvertieren von String in primitiven Typen

Die Umwandlung von Strings in primitive Datentypen erfolgt mit Hilfe der für den primitiven Datentyp zugeordneten Wrapper-Klasse. Diese Wrapper-Klassen besitzen jeweils eine statische Methode, die diese Konvertierung vornimmt.

Eine **Ausnahme** stellt der primitive Datentyp `char` dar. Seine Wrapper-Klasse `Character` bietet keine Umwandlungsroutine an. Da ein String eine Verkettung mehrerer einzelner Zeichen ist und ein Zeichen der Zeichenkette durch den primitiven Typ `char` repräsentiert wird, ist die entsprechende Konvertierungsroutine in der Klasse `String` untergebracht. Die Methode `toCharArray()` wandelt eine Zeichenkette in ein Array von Typ `char` um.

Tabelle 6-7 gibt einen Überblick:

primitiver Datentyp	zugehörige Wrapper-Klasse	zugehörige Methode
`char`	`Character`	-- nicht vorhanden --
`boolean`	`Boolean`	`public static boolean parseBoolean (String)`
`byte`	`Byte`	`public static byte parseByte (String)`
`short`	`Short`	`public static byte parseShort (String)`
`int`	`Integer`	`public static int parseInt (String)`
`long`	`Long`	`public static long parseLong (String)`
`double`	`Double`	`public static double parseDouble (String)`
`float`	`Float`	`public static float parseFloat (String)`

Tabelle 6-7 Primitive Typen, Wrapper-Klassen und Typkonvertierungsroutinen

Will man also beispielsweise eine Zeichenkette, die einen Wert vom Typ `double` repräsentiert, in den primitiven Datentyp `double` umwandeln, so kann man die folgenden Programmzeilen verwenden:

```
String s = "1.2";
double d = Double.parseDouble (s);
```

6.9 Annotationen

Seit Java 5 gibt es Annotationen (engl. annotations).

Annotationen sind Metadaten zum Programmcode, die zur Kommunikation des Programms mit dem Compiler, der Laufzeitumgebung oder anderen Werkzeugen dienen, und vom Programmierer erstellt werden.

Dem Compiler kann beispielsweise mittels Annotationen mitgeteilt werden, dass bestimmte Warnungen unterdrückt werden sollen. Die Laufzeitumgebung kann bei-

spielsweise Annotationen mit Hilfe der Reflection-API[112] auslesen und für Code-Injection[113] nutzen. Werkzeuge können Annotationen verwenden, um notwendige Informationen zur Codegenerierung auszulesen.

Annotationen zeichnen sich durch das vorangestellte `@` aus. Das Schlüsselwort hinter einem `@` drückt den Typ einer Annotation aus. Der Typ einer Annotation wird auch **Annotationstyp** genannt.

Hierzu ein erstes **Beispiel**:

```
@Override        // Der Compiler soll prüfen, ob die annotierte
void print()     // Methode eine Methode einer Vaterklasse
                 // überschreibt bzw. eine Methode eines
                 // Interface implementiert
```

Die Methode `print()` wird im obigen Beispiel durch die Annotation `@Override` mit einer zusätzlichen Information versehen, die vom Compiler ausgewertet wird.

Annotationen können in einem Programm an jeder beliebigen Vereinbarung stehen, d. h. an Vereinbarungen von Konstruktoren, Datenfeldern, lokalen Variablen, Methoden, Paketen[114], Parametern, Klassen, Enums, Schnittstellen und Annotationen. Dabei können Annotationen und Modifikatoren in beliebiger Reihenfolge notiert werden.[115] So kann beispielsweise

```
@Override public void print()
```

oder

```
public @Override void print()
```

geschrieben werden. Es ist jedoch allgemein üblich, die Annotation in einer eigenen Zeile anzugeben:

```
@Override
public void print()
```

Vor allem in Java EE werden seit EJB 3 Annotationen ausgiebig eingesetzt. Hier wurde früher in Deployment-Deskriptoren (XML) beschrieben, was eine Bean ist und wie sie sich verhält. Das ist heute immer noch möglich, bevorzugt wird aber die Beschreibung mit Annotationen. Annotationen gewinnen also zunehmend an Bedeutung, da sie das sogenannte "Ease of Development" unterstützen, d. h. die Entwicklung von Programmen vereinfachen.

[112] Reflection wird in diesem Einsteigerbuch nicht behandelt.
[113] Code-Injection erlaubt es, zur Laufzeit Code dynamisch einzufügen.
[114] Im Fall von Paketen muss die Annotation in einer Datei namens `package-info.java` stehen.
[115] Durch Konventionen kann die Reihenfolge beeinflusst werden.

6.9.1 Marker-Annotationen

Marker-Annotationen dienen lediglich dazu, Sprachkonstrukte von Java (z. B. Methoden oder Datenfelder) zu markieren. Es ist nicht möglich, eine Marker-Annotation mit zusätzlichen Informationen zu versehen. Ein **Beispiel** für eine Marker-Annotation ist die bereits vorgestellte Annotation `@Override`:

```
@Override
public void print()
```

6.9.2 Annotationen mit einem einzigen Element

Elemente können Annotationen mit zusätzlichen Informationen versehen. Diese Elemente tragen vordefinierte Namen. Bei der Erstellung einer Annotation werden die Elemente festgelegt. Die Erstellung von Annotationen wird in dieser Einführung nicht behandelt. Elemente einer Annotation können wie die Variablen einer Klasse mit einem Wert belegt werden. Besitzt eine Annotation nur ein einziges Element, spricht man auch von einer **Single-Value-Annotation**. Einem Element einer Annotation kann ein Wert folgendermaßen zugewiesen werden:

```
@SuppressWarnings (value = "unchecked")
void print{}
```

Der hier gezeigte Annotationstyp `SuppressWarnings` besitzt ein Element namens `value`, dem im vorangegangenen Beispiel die Zeichenkette `"unchecked"` zugewiesen wird.

Besitzt ein Annotationstyp nur ein einziges Element und dieses Element trägt den Namen `value`, so kann bei der Verwendung der Annotation der Name des zu setzenden Elementes weggelassen werden:

```
@SuppressWarnings ("unchecked")
void print{}
```

Hierbei wird dem Element namens `value` der Annotation `@SuppressWarnings` der Wert `"unchecked"` zugewiesen.

6.9.3 Annotationen mit mehreren Elementen

Annotationen können auch mehrere Elemente besitzen. Diese Elemente werden dann – analog zu der Parameterliste bei Methodenaufrufen – durch Kommata getrennt aufgelistet, wie im folgenden **Beispiel** zu sehen ist:

```
@Verfasser(vorname = "Peter", nachname = "Maier")
class MeineKlasse{...}
```

6.9.4 Verwendung mehrerer verschiedener Annotationen

Soll ein Sprachkonstrukt von Java mit mehreren Annotationen versehen werden, so werden diese üblicherweise wieder jeweils in eine eigene Zeile geschrieben:

```
@Titel("Mein Titel")
@Verfasser(vorname = "Peter", nachname = "Maier")
class MeineKlasse{...}
```

6.9.5 Wiederholende Annotationen

Seit Java 8 können sogenannte wiederholende Annotationen (engl. repeating annotations) verwendet werden, d. h. ein Sprachkonstrukt von Java wird mit mehreren Annotationen desselben Annotationstyps versehen. Hierfür ein **Beispiel**:

```
@Verfasser(vorname = "Peter", nachname = "Maier")
@Verfasser(vorname = "Max", nachname = "Mueller")
class MeineKlasse{...}
```

6.9.6 Typ-Annotationen

Eine weitere Neuerung in Java 8 sind die sogenannten Typ-Annotationen (engl. type annotations). Bis Java 7 konnten Annotationen nur an Vereinbarungen wie beispielsweise einer Klasse stehen. Die in Java 8 eingeführten Typ-Annotationen erlauben es nun, die Verwendung von Klassen oder Schnittstellen zu annotieren. Einige Beispiele, an denen Typ-Annotationen stehen können, sind die Erzeugung von Instanzen mittels `new`-Operator, `throws`-Klauseln[116] oder der cast-Operator. Die genannten **Beispiele** könnten folgendermaßen aussehen:

```
// Annotation @InEntwicklung nach new-Operator
new @InEntwicklung MeineKlasse() {...}

// Annotation @Kritisch nach throws-Klausel
void dauerBerechnen()
     throws @Kritisch DauerUeberzogenException {...}[117]

// Annotation @NichtNull in cast-Operator
meineAdresse = (@NichtNull String) adresse;
```

Typ-Annotationen erlauben eine verbesserte Analyse von Java-Programmen durch eine stärkere Typprüfung. Durch Typ-Annotationen ist es möglich, Frameworks zur Typprüfung selbst zu schreiben oder auf existierende Frameworks zuzugreifen, die eine Typprüfung mit Hilfe von Typ-Annotationen durchführen.

Das Checker Framework[118] der Universität Washington z. B. definiert eine Typ-Annotation namens `@NonNull`. Es können mit dieser Annotation u. a. Typen der Übergabeparameter von Methoden versehen werden:

---

[116] Siehe Kapitel 13.5.5
[117] Exceptions werden erst in Kapitel 13 behadelt.
[118] http://types.cs.washington.edu/checker-framework/

```
void setName(@NonNull String name){...}
```

Das Framework überprüft im Beispiel beim Kompilieren des Java-Codes, ob die Methode `setName()` mit der `null`–Referenz als Übergabeparameter aufgerufen wird. Ist das an irgendeiner Stelle des Java-Codes der Fall, so wird vom Compiler eine Warnung ausgegeben.

6.9.7 Vordefinierte Annotationstypen

Die Laufzeitumgebung von Java stellt fünf vordefinierte Annotationstypen im Paket `java.lang` zur Verfügung, die genutzt werden können. Zwei davon wurden bereits in den vorigen Beispielen verwendet, nämlich die Annotationstypen `Override` und `SuppressWarnings`. Darüber hinaus existieren im Paket `java.lang` die Annotationstypen `Deprecated`, `SafeVarargs` und seit Java 8 `FunctionalInterface`. Die Bedeutung dieser vordefinierten Annotationstypen des Pakets `java.lang` wird im Folgenden kurz erklärt:

- `@Deprecated` markiert eine Methode als veraltet und weist den Programmierer darauf hin, diese Methode nicht mehr zu verwenden. Wird eine mit `@Deprecated` annotierte Methode verwendet oder überschrieben, so gibt der Compiler eine Warnung aus.
- `@FunctionalInterface` markiert eine Schnittstelle als sogenannte funktionale Schnittstelle[119] im Sinne der Java Language Specification. Wird eine Methode mit `@FunctionalInterface` gekennzeichnet, die keine funktionale Schnittstelle im Sinne der Java Language Specification ist, so wird vom Compiler eine Fehlermeldung ausgegeben.
- `@Override` markiert eine Methode, die eine Methode einer Basisklasse überschreibt bzw. eine Methode einer Schnittstelle implementiert. Wenn eine mit `@Override` gekennzeichnete Methode keine Methode einer Basisklasse überschreibt bzw. keine Methode einer Schnittstelle implementiert, wird vom Compiler eine Fehlermeldung ausgegeben.
- `@SafeVarargs` weist den Compiler an, für die markierten Methoden und Konstruktoren keine Warnungen auszugeben, die im Zusammenhang mit möglichen Typunsicherheiten bei generischen Parameterlisten variabler Länge stehen.
- `@SuppressWarnings` weist den Compiler an, für das markierte Sprachkonstrukt von Java und alle darin enthaltenen Sprachkonstrukte keine Warnungen auszugeben. Dabei müssen ein oder mehrere Namen von Warnungen angeben werden, die unterdrückt werden sollen. Wird z. B. eine Klasse mit `@SuppressWarnings("unused")` gekennzeichnet, so werden Warnungen für sämtliche in der Klasse enthaltenen privaten Datenfelder, Methoden oder Konstruktoren, die nicht benutzt werden, unterdrückt. Es ist deshalb ratsam, nur die am tiefsten verschachtelten Sprachkonstrukte mit `@SuppressWarnings` zu kennzeichnen.

Neben der Nutzung vordefinierter Annotationstypen können auch selbst definierte Annotationstypen erstellt werden. Selbst definierte Annotationen können z. B. von selbst geschriebenen Tools ausgewertet werden. Das wird von vielen Frameworks

[119] Eine Schnittstelle wird als funktionale Schnittstelle bezeichnet, wenn sie genau eine abstrakte Methode besitzt.

genutzt. Hibernate beispielsweise hat selbst definierte Annotationen, durch die etwa erkannt wird, wann eine Datenbanktransaktion zu öffnen und wieder zu schließen ist. Das Framework Spring wiederum nutzt selbst definierte Annotationen, um eine aspektorientierte Programmierung zu realisieren und zur Laufzeit Code zu injizieren.

Zur Erstellung selbst definierter Annotationstypen können wiederum Annotationen verwendet werden, die sogenannten **Meta-Annotationen**. Meta-Annotationen sind im Paket `java.lang.annotation` definiert und können nur an Annotationstypen stehen, nicht an anderen Sprachkonstrukten von Java.

Das Paket `java.lang.annotation` beinhaltet die vordefinierten Annotationstypen `Documented`, `Inherited`, `Retention` und `Target`. Die Bedeutung dieser Meta-Annotationen und die Erstellung selbst definierter Annotationstypen kann in dieser Einführung leider nicht durchgeführt werden.

6.10 Zusammenfassung

Java ist statisch typisiert, d. h. der Datentyp einer Variablen muss zur Übersetzungszeit bekannt sein. Kapitel **6.1** gibt eine Übersicht über die konkreten und generischen Datentypen in Java.

In Kapitel **6.2** werden die einfachen Datentypen von Java beschrieben. Einfache Datentypen sind `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` und `double`.

Ein abstrakter Datentyp (siehe Kapitel **6.3**) basiert auf der Formulierung abstrakter Operationen auf abstrakt formulierten Daten, wobei die Semantik der Operationen definiert sein muss. Eine Klasse in einer speziellen objektorientierten Programmiersprache implementiert einen abstrakten Datentyp. Eine Klasse hat einen Namen und enthält Datenfelder und Methoden.

Kapitel **6.4** beschreibt statische und dynamische Variablen. Nach ihrer Definition kann auf eine statische Variable über ihren Namen zugegriffen werden. Eine solche Variable heißt statisch, weil ihr Gültigkeitsbereich und ihre Lebensdauer durch die statische Struktur des Programms festgelegt ist. Der Gültigkeitsbereich einer statischen Variablen umfasst alle Stellen im Programm, an denen ihr Name durch die Vereinbarung bekannt ist. Statische Variablen sind lokale Variablen, Instanzvariablen und Klassenvariablen. Dynamische Variablen sind Objekte auf dem Heap. Sie sind nicht über einen Namen greifbar. Die Gültigkeit und die Lebensdauer einer dynamischen Variablen wird nicht durch die statische Struktur des Programms – in anderen Worten durch die Blockgrenzen – bestimmt. Zum Zugriff auf dynamische Variablen – sie liegen auf dem Heap – benötigt man Referenzvariablen, d. h. statische Variablen vom Typ einer Klasse.

Kapitel **6.5** behandelt Arrays. Ein Array ist ein Objekt, das aus Komponenten (Elementen) zusammengesetzt ist, wobei jedes Element eines Arrays vom selben Datentyp sein muss. Man kann in Java Arrays aus Elementen eines einfachen Datentyps oder aus Elementen eines Referenztyps anlegen. Ein Element eines Arrays kann auch selbst wieder ein Array sein. Dann entsteht ein mehrdimensionales Array. Da

Arrays implizit von der Klasse `Object` abgeleitet sind, weisen sie beispielsweise die Methode `equals()` der Klasse `Object` auf.

Kapitel **6.6** behandelt Aufzählungstypen. Eine Variable eines Aufzählungstyps enthält nur einen einzigen Wert, der sich nicht aus weiteren Werten zusammensetzt. Mit anderen Worten, der Wert einer Variablen eines Aufzählungstyps ist atomar. Dieser einzige Wert muss einem Wert aus der Liste der Aufzählungskonstanten entsprechen. Einer Variablen eines Aufzählungstyps können nur Werte aus der Menge der Aufzählungskonstanten zugewiesen werden.

In Java werden Aufzählungstypen in Form von Klassen angelegt. Ein Aufzählungstyp ist in Java eine Klasse und die Aufzählungskonstanten sind konstante Referenzvariablen auf Objekte des Aufzählungstyps. Dem Aufzählungstyp können deshalb auch Methoden und Datenfelder hinzugefügt werden.

Jede Aufzählungskonstante zeigt auf ein Objekt des Aufzählungstyps, welches den Namen der Aufzählungskonstanten als String und auch den Ordinalwert der Aufzählungskonstanten als Instanzvariable enthält.
Da ein Aufzählungstyp eine Klasse darstellt, können die Aufzählungstypen auch Datenfelder und Methoden haben. Methoden können bei Aufzählungstypen für jede einzelne Aufzählungskonstante überschrieben werden. Dies geschieht an derjenigen Stelle, an der die Aufzählungskonstanten erzeugt werden.

Kapitel **6.7** befasst sich mit Zeichenketten in Java. In Java gibt es drei verschiedene String-Klassen:

- die Klasse `String` für konstante Zeichenketten
- und die Klasse `StringBuffer` sowie die Klasse `StringBuilder` für variable Zeichenketten.

Für mit `new` erzeugte Strings wird immer ein neues `String`-Objekt im Heap erzeugt. Bei optimierenden Compilern können Objekte vom Typ `String`, die den gleichen Inhalt haben und implizit erzeugt werden, wiederverwendet werden. Objekte vom Typ `String` können nicht abgeändert werden.

Die Klasse `StringBuffer` gehört auch zum Paket `java.lang` und repräsentiert eine Zeichenkette mit den folgenden Eigenschaften:

- Die Länge der Zeichenkette in einem Objekt der Klasse `StringBuffer` ist nicht festgelegt.
- Die Länge vergrößert sich automatisch, wenn in einem Objekt der Klasse `StringBuffer` weitere Zeichen angefügt werden und der vorhandene Platz nicht ausreicht.
- Der Inhalt eines Objektes der Klasse `StringBuffer` lässt sich verändern.

Objekte vom Typ `StringBuffer` können nicht implizit angelegt werden.

Seit Java 5 stellt das Paket `java.lang` auch eine Klasse `StringBuilder` zur Verfügung. Bis dahin war die Klasse `StringBuffer` die einzige Möglichkeit, um Zeichenketten zu verändern. Die Klasse `StringBuilder` ist eine neue Klasse mit

einer höheren Performanz, die jedoch nicht verwendet werden darf, wenn eine Zeichenkette von mehreren Threads parallel bearbeitet wird.

In Kapitel **6.8** steht die Wandlung von Variablen eines einfachen Datentyps in Variablen eines Klassen-Typs und umgekehrt sowie die Wandlung von Variablen beliebiger Datentypen in Zeichenketten im Fokus.

Kapitel **6.9** behandelt Annotationen. Annotationen sind Metadaten zum Programmcode, die zur Kommunikation des Programms mit dem Compiler, der Laufzeitumgebung oder anderen Werkzeugen dienen. Es gibt in der Java API vordefinierte Annotationen, es können aber auch vom Programmierer eigene Annotationen erstellt werden.

6.11 Übungen

Aufgabe 6.1: Arrays

6.1.1 `int`-Array als Stack

Die Klasse `Stack` soll ein `int`-Array kapseln, welches als Stack dienen soll. Die Funktionsweise eines Stacks wird in Kapitel 6.4.5.1 erklärt. Für den Zugriff auf den Stack sollen die Methoden

```
public void push (int u)
public int pop()
```

bereitgestellt werden.

Die Methode

```
public boolean isEmpty()
```

überprüft, ob der Stack leer ist, und liefert in diesem Fall `true` zurück, ansonsten wird `false` zurückgeliefert. Die Methode

```
public void stackPrint()
```

soll zu Testzwecken dienen und den Inhalt des gesamten Stacks ausgeben. Die Größe des Stacks soll dem Konstruktor übergeben werden können. Testen Sie die Klasse `Stack` mit Hilfe der folgenden Wrapper-Klasse:

```
// Datei: TestStack.java

public class TestStack
{
   public static void main (String[] args)
   {
      Stack stackRef = new Stack (5);
      stackRef.push (7);
      stackRef.push (3);
      stackRef.push (4);
      stackRef.push (9);
      stackRef.push (1);
```

```
      stackRef.stackPrint();

      System.out.println ("\nAusgabe der Zahlen: ");
      while (stackRef.isEmpty() == false)
      {
         int rückgabe;

         // oberste Zahl des Stacks wird
         // mit pop() vom Stack geholt
         rückgabe = stackRef.pop();
         System.out.println ("Die Zahl war " + rückgabe);
      }
   }
}
```

6.1.2 Array mit einfachen Datentypen – die Klasse `FloatQueue`

Die selbst geschriebene Klasse `FloatQueue` ist eine Warteschlange für `float`-Werte. In dieser Warteschlange können sich mehrere `float`-Werte befinden. Es kann jeweils nur ein Element gleichzeitig in die Warteschlange (hinten) eingereiht werden (`enqueue()`-Methode) oder aus der Warteschlange (vorne) entnommen werden (`dequeue()`-Methode). Im Gegensatz zu einem Stapelspeicher (Stack) handelt es sich bei einer Warteschlange um einen **FIFO**-Speicher ("**First In – First Out**").

Die Klasse `FloatQueue` soll folgende Methoden beinhalten:

- **Konstruktor:** **`public FloatQueue (int laenge)`**

  Der Übergabeparameter `int laenge` gibt die Anzahl der maximalen Speicherstellen der Warteschlange an.

- **In Warteschlange einfügen:** **`public void enqueue (float wert)`**

  Diese Methode fügt den Wert am Ende der Warteschlange ein.

- **Aus Warteschlange entnehmen:** **`public float dequeue()`**

  Diese Methode entfernt den ersten Wert aus der Warteschlange und gibt diesen an den Aufrufer zurück. Ist die Warteschlange leer, so wird der Wert `-1` zurückgegeben.

- **Ausgabe des Inhalts der Warteschlange:** **`public void queuePrint()`**

  Diese Methode gibt alle in der Warteschlange enthaltenen Werte aus.

- **Überprüfen, ob die Warteschlange leer ist:** **`public boolean isEmpty()`**

  Diese Methode liefert `true` zurück, falls die Warteschlange leer ist. Andernfalls gibt die Methode `false` zurück.

- **Leeren der Warteschlange:** **`public void clear()`**

  Diese Methode löscht alle in der Warteschlange enthaltenen Werte.

Testen Sie die Klasse `FloatQueue` mit Hilfe folgender Testklasse:

```
// Datei: TestFloatQueue.java

public class TestFloatQueue
{
   public static void main (String[] args)
   {
      FloatQueue queue = new FloatQueue(5);
      queue.enqueue (2.45f);
      queue.enqueue (1.29f);
      queue.enqueue (4.31f);
      queue.enqueue (7.85f);

      queue.queuePrint();

      System.out.println ("\nAusgabe der Zahlen: ");

      while (queue.isEmpty() == false )
      {
         float rueckgabe;
         rueckgabe = queue.dequeue();
         System.out.println ("Die Zahl war " + rueckgabe);
      }

      queue.enqueue (1.11f);
      queue.queuePrint();
      queue.clear();
      queue.queuePrint();
   }
}
```

Aufgabe 6.2: Strings

6.2.1 Performance

Führen Sie die folgende Klasse `TestString` aus, welche zum Testen der Performance des Verkettungsoperators `+` von Strings dient. Die Zeit, welche die `for`-Schleife benötigt, wird in Millisekunden gemessen. Die Zeitmessung erfolgt mit Hilfe der Klasse `System` (siehe Anhang C). Hier die Klasse `TestString`:

```
// Datei: TestString.java

public class TestString
{
   public static void main (String[] args)
   {
      String s = "Hello";
      System.out.println ("Starte Schleife, Bitte warten");
      long startTime = System.currentTimeMillis();

      for (int n = 0; n < 10000; n++)
      {
         s += "World";
      }

      long endTime = System.currentTimeMillis();

      System.out.println ("Mit dem + Operator braucht man " +
```

```
                                (endTime-startTime) +
                                " Millisekunden");

        System.out.println ("Der zusammengesetzte String hat " +
                                "eine Länge von " + s.length () +
                                " Zeichen");
    }
}
```

Die gemessene Zeit erscheint recht hoch. Es wird allerdings ein Vergleich benötigt. Fügen Sie einen Block an, in welchem der String `"Hello"` in einem Objekt der Klasse `StringBuffer` steht und der String `"World"` nicht über den Verkettungsoperator, sondern über die Methode `append()` der Klasse `StringBuffer` hinzugefügt wird. Sie können natürlich auch andere oder weitere Möglichkeiten programmieren und die Zeit messen.

Um welchen Faktor unterscheiden sich die Laufzeiten der beiden Möglichkeiten? Geben Sie eine Erklärung für die Laufzeitunterschiede an.

6.2.2 Dateinamen ermitteln

Benutzen Sie die Methoden der Klasse `String`, um eine Klasse `Parser` zu schreiben. Diese Klasse hat die Aufgabe, aus einem vollständigen Pfad in Form eines Strings das Verzeichnis, den Dateinamen und die Extension der Datei zu ermitteln. Lautet zum Beispiel der gesamte Pfad:

```
C:\Eigene Daten\Javatest\Beispiel.java
```

dann soll das Programm Folgendes extrahieren:

```
Extension:    java

Dateiname:    Beispiel

Verzeichnis:  C:\Eigene Daten\Javatest
```

Aufgabe 6.3: Aufzählungstypen

6.3.1 Wochentage

Definieren Sie einen Aufzählungstyp `Wochentag`, der die Tage der Woche repräsentiert, und eine Klasse `WochentagAusgabe`. In der `main()`-Methode der Klasse `WochentagAusgabe` soll die Methode `values()` des Aufzählungstyps verwendet werden, um alle Wochentage auszugeben. Zu jedem Wochentag soll die jeweilige Ordinal-Zahl ausgegeben werden. Die Ausgabe soll folgendermaßen aussehen:

```
MONTAG ist der 1. Tag der Woche.
DIENSTAG ist der 2. Tag der Woche.
MITTWOCH ist der 3. Tag der Woche.
DONNERSTAG ist der 4. Tag der Woche.
FREITAG ist der 5. Tag der Woche.
SAMSTAG ist der 6. Tag der Woche.
SONNTAG ist der 7. Tag der Woche.
```

6.3.2 Rechenmaschine

Definieren Sie einen Aufzählungstyp `Operation` mit den Aufzählungskonstanten `PLUS`, `MINUS`, `TIMES` und `DIVIDE`. Der Aufzählungstyp soll die Methode `eval (double arg0, double arg1)` haben, die für jede Aufzählungskonstante entsprechend überschrieben werden muss. Implementieren Sie eine Klasse `Rechenmaschine`, die ein privates Datenfeld vom Typ `Operation` hat. Die Rechenmaschine soll so funktionieren, dass zuerst eine Operation gesetzt wird, dann werden zwei Parameter vom Typ `double` übergeben. Abschließend wird die Methode `ausfuehren()` aufgerufen, die das Ergebnis berechnet und ausgibt. Schreiben Sie eine `main()`-Methode, um die Klasse `Rechenmaschine` und den Aufzählungstyp zu testen. Nutzen Sie die Methode `values()` des Aufzählungstyps, um alle Operationen in einer Schleife zu testen.
Die Ausgabe soll folgendermaßen aussehen:

```
Die Operation PLUS ergibt für die Parameter  2.0 und 3.0 das
Ergebnis 5.0.
Die Operation MINUS ergibt für die Parameter 2.0 und 3.0 das
Ergebnis -1.0.
Die Operation TIMES ergibt für die Parameter 2.0 und 3.0 das
Ergebnis 6.0.
Die Operation DIVIDE ergibt für die Parameter 2.0 und 3.0 das
Ergebnis 0.6666666666666666.
```

6.3.3 Münzen

Ergänzen Sie die Klasse `Muenze`, die einen Aufzählungstyp darstellt, mit dem alle Münzen der Euro-Währung abgebildet werden können. Jeder definierten Aufzählungskonstanten von diesem Typ soll als Wert der entsprechende Münzbetrag in Cent zugewiesen werden, d. h. der Aufzählungskonstanten für das Ein-Cent-Stück der Wert 1, der Aufzählungskonstanten für das Zwei-Cent-Stück der Wert 2 usw. Hier die Klasse `Muenze`:

```
// Datei: Muenze.java
public enum Muenze
{
   // Definition der Aufzählungskonstanten
   EinCent . . . . .,
   ZweiCent . . . . . .,
   FuenfCent . . . . .,
   ZehnCent . . . . .,
   ZwanzigCent . . . . .,
   FuenfzigCent . . . . .,
   EinEuro . . . . .,
   ZweiEuro . . . . .;

   // Datenfeld
   private int wert;

   // Konstruktor
   Muenze (int wert)
   {
      this.wert = wert;
   }
```

```
   // Methode zum Auslesen des Wertes
   public int value()
   {
      return wert;
   }
}
```

Definieren Sie außerdem einen Aufzählungstyp, der alle Metalle enthält, aus denen die Euro-Münzen hergestellt sind. Die Ein-, Zwei- und Fünf-Cent-Münzen bestehen auf Kupfer, die Zehn-, Zwanzig- und Fünfzig-Cent-Münzen bestehen aus Messing und die Ein- und Zwei-Euro-Münzen sind aus den Metallen Messing und Nickel zusammengesetzt. Schreiben Sie hierzu den Aufzählungstyp `Metall`, in welchem die Aufzählungskonstanten `Kupfer`, `Messing` und `MessingUndNickel` definiert sind.

Die beiden Aufzählungstypen `Muenze` und `Metall` werden mit der Testklasse `Kleingeld` getestet. Ergänzen Sie die fehlenden Stellen im Quellcode der Testklasse `Kleingeld`:

```
// Datei: Kleingeld.java
public class Kleingeld
{
   public static void main (String[] args)
   {
      System.out.println ("Es gibt die folgenden Muenzen:");
      Muenze[] euroMuenzen = . . . . .;
      for (int i = 0; i < euroMuenzen.length; i++)
      {
         System.out.print (euroMuenzen [i] + " ");
      }
      System.out.println();

      for (int i = 0; i < euroMuenzen.length; i++)
      {
         switch (. . . . .)
         {
            case . . . . .:
            case . . . . .:
            case . . . . .:
            {
               System.out.println ("Die Muenze " +
               euroMuenzen[i]. . . . .  + " ist aus " +
               Metall.Kupfer. . . . . + " und hat den Wert " +
               euroMuenzen[i]. . . . . + " Cent");
               break;
            }
            case . . . . .:
            case . . . . .:
            {
               System.out.println ("Die Muenze " +
               euroMuenzen[i]. . . . . + " ist aus " +
               Metall.Messing. . . . . +" und hat den Wert " +
               euroMuenzen[i]. . . . . +  " Cent");
               break;
            }
```

```
            case . . . . .:
            case . . . . .:
            {
               System.out.println ("Die Muenze " +
               euroMuenzen[i]. . . . . + " ist aus " +
               Metall.MessingUndNickel. . . . . +
               " und hat den Wert " +
               euroMuenzen[i]. . . . . + " Cent");
               break;
            }
         }
      }
   }
}
```

Die Ausgabe der Testklasse `Kleingeld` sieht folgendermaßen aus:

```
Es gibt die folgenden Münzen:
EinCent ZweiCent FuenfCent ZehnCent ZwanzigCent FuenfzigCent EinEuro
ZweiEuro
Die Muenze EinCent ist aus Kupfer und hat den Wert 1 Cent
Die Muenze ZweiCent ist aus Kupfer und hat den Wert 2 Cent
Die Muenze FuenfCent ist aus Kupfer und hat den Wert 5 Cent
Die Muenze ZehnCent ist aus Messing und hat den Wert 10 Cent
Die Muenze ZwanzigCent ist aus Messing und hat den Wert 20 Cent
Die Muenze FuenfzigCent ist aus Messing und hat den Wert 50 Cent
Die Muenze EinEuro ist aus MessingUndNickel und hat den Wert 100
Cent
Die Muenze ZweiEuro ist aus MessingUndNickel und hat den Wert 200
Cent
```

Aufgabe 6.4: Boxing und Unboxing

6.4.1 Auto-Boxing und Auto-Unboxing von aktuellen Parametern

Erstellen Sie eine Klasse `BoxingUnboxing` mit zwei Methoden. Die eine Methode soll einen Übergabeparameter vom Typ `int` und die andere einen Übergabeparameter vom Typ `Integer` haben. Erstellen Sie eine `main()`-Methode, in der sie eine Variable vom Typ `int` und eine andere Variable vom Typ `Integer` anlegen. Rufen Sie die Methoden so auf, dass der Compiler Auto-Boxing bzw. Auto-Unboxing durchführen muss.

6.4.2 Operatoren mit Auto-Boxing und Auto-Unboxing

Erstellen Sie eine Klasse `BoxingUnboxing2`, die eine `main()`-Methode enthält. Legen Sie in dieser Methode zwei Variablen vom Typ `Integer` an und initialisieren Sie diese mit Hilfe von Auto-Boxing.

Ändern Sie den Wert dieser beiden Variablen mit Hilfe der unären Operatoren `++` und `--`.

Legen Sie eine dritte Variable vom Typ `int` an und initialisieren Sie diese mit der Differenz der Werte von Variable 1 und Variable 2.

Vergleichen Sie den Wert zweier Variablen vom Typ `Integer` mit Hilfe der relationalen Operatoren. Überlegen Sie, welche relationalen Operatoren nicht verwendet werden dürfen, da damit nicht die Werte verglichen werden.

Nutzen Sie einen Bit-Operator, um den Wert einer der Variablen vom Typ `Integer` zu verdoppeln.

Legen Sie eine Variable vom Typ `Boolean` an und verwenden Sie diese mit dem Bedingungsoperator `?:` (siehe Kapitel 7.6.7), um den jeweiligen Wert mit den Strings "wahr" oder "falsch" auszugeben.

Schreiben Sie eine `switch`-Anweisung, wobei Sie nach dem Wert einer Variablen vom Typ `Character` unterscheiden.
Die Ausgabe des Programms soll folgendermaßen aussehen:

```
Der Wert von i3 ist: 3
i1 > i2 : true
i1 < i2 : false
i1 == i2 : false
i1 != i2 : true
i1 vor der Bit-Operation: 4
i1 nachher: 8
b ist wahr
Der Ausdruck der switch-Anweisung hat den Wert 'c'.
```

Kapitel 7

Ausdrücke und Operatoren

X = (A + B) * C

7.1 Operatoren und Operanden
7.2 Ausdrücke und Anweisungen
7.3 Nebeneffekte
7.4 Auswertungsreihenfolge
7.5 L-Werte und R-Werte
7.6 Zusammenstellung der Operatoren
7.7 Konvertierung von Datentypen
7.8 Zusammenfassung
7.9 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_7

7 Ausdrücke und Operatoren

Ein **Ausdruck** ist in Java im einfachsten Falle der Bezeichner (Name) einer Variablen oder einer Konstanten. Meist interessiert der Wert eines Ausdrucks. So hat eine Konstante einen Wert, eine Variable kann einen Wert liefern, aber auch der Aufruf einer Instanz- oder Klassenmethode. Der Wert eines Ausdrucks wird oft auch als **Rückgabewert** des Ausdrucks bezeichnet.

Alles das, was einen Wert zurückliefert, stellt einen Ausdruck dar.

Verknüpft man Operanden – ein Operand ist selbst ein Ausdruck – durch Operatoren und gegebenenfalls auch runde Klammern, so entstehen **komplexe Ausdrücke**.

Runde Klammern beeinflussen dabei die **Auswertungsreihenfolge** der Operatoren. Das Ziel dieser Verknüpfungen ist die Berechnung neuer Werte oder auch das Erzeugen von gewollten Nebeneffekten (siehe Kapitel 7.3).

Kapitel 7.1 stellt die Einteilung von Operatoren in Java als unäre, binäre und ternäre Operatoren vor. In Kapitel 7.2 wird der Begriff eines Ausdrucks vom Begriff einer Anweisung abgegrenzt. Wenn Inhalte des Speichers, meist einer statischen Variablen, verändert werden, wird von Nebeneffekten gesprochen. Nebeneffekte werden in Kapitel 7.3 diskutiert. Hierbei wird insbesondere auf den Ausführungszeitpunkt von Nebeneffekten eingegangen. Kapitel 7.4 erklärt die Auswertungsreihenfolge der Operatoren eines Ausdrucks und die Bewertungsreihenfolge der Operanden. Anschließend wird in Kapitel 7.5 auf die Verwendbarkeit von L-Werten und R-Werten beim Schreiben von Zuweisungen eingegangen. Kapitel 7.6 gibt eine Beschreibung der verschiedenen Operatoren in Java. Kapitel 7.7 stellt den cast-Operator für die Konvertierung von Datentypen vor und behandelt die Konvertierung einfacher Datentypen.

7.1 Operatoren und Operanden

Um Verknüpfungen mit Operanden durchzuführen, braucht man **Operatoren** (siehe Kapitel 5.3.7).

Es gibt in Java die folgenden Arten von Operatoren:

- **einstellige** (**unäre**, **monadische**),
- **zweistellige** (**binäre**, **dyadische**)
- und einen einzigen **dreistelligen** (**ternären**, **triadischen**), nämlich den Bedingungsoperator `? :`

Unäre Operatoren

Ein einstelliger Operator – auch **unärer Operator** genannt –, hat einen einzigen Operanden.

Ein Beispiel hierfür ist der Minusoperator als Vorzeichenoperator, der auf einen einzigen Operanden wirkt und das Vorzeichen des Wertes des Operanden ändert. So kehrt in folgendem Bild der Minusoperator das Vorzeichen des Wertes von `a` um.

Bild 7-1 Ein unärer Operator angewandt auf einen Operanden

Unäre Operatoren können vor oder hinter ihren Operanden stehen. Darauf soll im Folgenden eingegangen werden.

Unäre Operatoren als Postfix-Operatoren

Als Beispiel soll der Postfix-Operator `++` betrachtet werden. Der Ausdruck

```
u++
```

stellt die Anwendung des Postfix-Operators `++` auf seinen Operanden `u` dar.

Auf den Rückgabewert und Nebeneffekt des Postfix-Operators `++` wird in Kapitel 7.3 eingegangen.

Postfix-Operatoren sind **unäre** Operatoren, die **hinter** (**post**) **ihrem Operanden** stehen.

Unäre Operatoren als Präfix-Operatoren

Ein Beispiel für einen Präfix-Operator ist das unäre Minus (Minus als Vorzeichen), ein anderes Beispiel ist der Präfix-Operator `++`, siehe folgendes Beispiel:

```
++u
```

Präfix-Operatoren sind **unäre** Operatoren, die **vor** (**prä**) **ihrem Operanden** stehen.

Auf den Rückgabewert und Nebeneffekt des Präfix-Operators `++` wird in Kapitel 7.3 eingegangen.

Binäre Operatoren

Benötigt ein Operator 2 Operanden für die Verknüpfung, so spricht man von einem **zweistelligen** (**binären**) **Operator**.

Ein vertrautes Beispiel für einen binären Operator ist der Additionsoperator, der in folgendem Beispiel zur Addition der beiden Zahlen `3` und `4` verwendet werden soll:

Bild 7-2 Ein binärer Operator verbindet zwei Operanden zu einem Ausdruck

Binäre Operatoren kann man auch nach ihrer Wirkungsweise klassifizieren. So gibt es außer den **arithmetischen Operatoren** beispielsweise auch **logische Operatoren**, **Zuweisungsoperatoren** oder **Vergleichsoperatoren** (**relationale Operatoren**).

Ternärer Operator

Der einzige dreistellige Operator ist der Bedingungsoperator. Der Bedingungsoperator wird in Kapitel 7.6.7 behandelt.

7.2 Ausdrücke und Anweisungen

Anweisungen und Ausdrücke sind nicht das Gleiche. Sie unterscheiden sich durch den Rückgabewert.

Ausdrücke haben in Java stets einen **Rückgabewert**. Alles das, was einen Wert zurückliefert, stellt einen Ausdruck dar. **Anweisungen** haben **keinen Rückgabewert**.

Was ist aber nun genau der Rückgabewert? Das soll anhand des Ausdrucks `3 + 4` erklärt werden. Durch die Anwendung des Additionsoperators `+` auf seine Operanden `3` und `4` ist der Rückgabewert des Ausdrucks `3 + 4` eindeutig festgelegt. Aus den Typen der Operanden ergibt sich immer eindeutig der **Typ des Rückgabewertes**. Da beide Operanden vom Typ `int` sind, ist der Rückgabewert der Addition ebenfalls vom Typ `int` und hat den Wert `7`.

Der Wert eines Ausdrucks wird auch als sein **Rückgabewert** bezeichnet. **Jeder Rückgabewert hat** auch **einen Typ**.

Es werden die folgenden Anweisungen unterschieden:

- Selektionsanweisungen (siehe Kapitel 8.2),
- Iterationsanweisungen (siehe Kapitel 8.3),
- Sprunganweisungen (siehe Kapitel 8.4),
- die leere Anweisung (siehe Kapitel 9.1.2)),
- die `try`-Anweisung (siehe Kapitel 13),
- die `throw`-Anweisung (siehe Kapitel 13.3),
- die `assert`-Anweisung (siehe Kapitel 13.7.1),
- die `synchronized`-Anweisung (siehe Kapitel 16)
- und Ausdrucksanweisungen.

Ausdrucksanweisungen werden sogleich im Folgenden behandelt.

Ausdrucksanweisungen

In Java kann man bei bestimmten Arten von Ausdrücken durch das Anhängen eines Semikolons an den vorliegenden Ausdruck erreichen, dass dieser Ausdruck zu einer Anweisung wird. Man spricht dann von einer sogenannten **Ausdrucksanweisung**.

In einer solchen **Ausdrucksanweisung** wird der **Rückgabewert** eines Ausdrucks **nicht verwendet**.

Lediglich wenn **Nebeneffekte** zum Tragen kommen, ist eine **Ausdrucksanweisung sinnvoll und zulässig**.

Die folgenden Ausdrücke können in Java zu einer Anweisung werden:

- Zuweisungen, d. h. `=` und kombinierte Zuweisungsoperatoren wie z. B. `+=`, da (kombinierte) Zuweisungungsoperatoren einen Nebeneffekt haben.
- Postfix- und Präfix-Inkrement- und -Dekrementoperator, d. h. `++` und `--` angewandt auf eine Variable, da diese Operatoren einen Nebeneffekt haben.
- Methodenaufrufe, unbenommen davon, ob sie einen Rückgabewert haben oder nicht, da Methodenaufrufe Nebeneffekte haben können.
- Ausdrücke, die mit `new` ein Objekt erzeugen. Begründung: Der Rückgabewert des `new`-Operators ist die Referenz auf das erzeugte Objekt. Der Nebeneffekt ist, dass ein Objekt im Heap existiert.

Das folgende Beispiel zeigt eine zulässige und eine unzulässige Ausdrucksanweisung:

```
. . . . .
int c = 0;
// 5 * 5; // nicht zulässige Ausdrucksanweisung.
          // Kein Nebeneffekt
. . . . .
c++;      // zulässige Ausdrucksanweisung. Hat Nebeneffekt
```

7.3 Nebeneffekte

Bei **Nebeneffekten** werden **Inhalte des Speichers verändert**, meist der Wert einer statischen Variablen.

Ein anderes Beispiel ist das Anlegen eines Objektes durch den `new`-Operator. Hier ist – wie bereits erwähnt – der Rückgabewert des `new`-Operators der Rückgabewert auf das im Heap erzeugte Objekt. Der Nebeneffekt ist, dass ein Objekt im Heap existiert.

Nebeneffekte werden auch als **Seiteneffekte** oder **Nebenwirkungen** bezeichnet.

Es gibt Operatoren, die eine schnelle und kurze Programmierschreibweise erlauben und es ermöglichen, während der Auswertung eines Ausdrucks Programmvariablen nebenbei zu verändern. Ein Beispiel hierfür ist:

```
int u = 1;
int v;
v = u++;
```

Der Rückgabewert des Ausdrucks `u++` ist der Wert `1`. Mit dem Zuweisungsoperator wird der Variablen `v` der Rückgabewert von `u++`, d. h. der Wert `1`, zugewiesen. Die Zuweisung `v = u++` ist ebenfalls ein Ausdruck und `v = u++;` stellt eine **Ausdrucksanweisung** dar. Als Nebeneffekt des Operators `++` wird in diesem Beispiel die Variable `u` inkrementiert und hat nach der Inkrementierung den Wert `2`.

Man sollte mit Nebeneffekten sparsam umgehen, da sie leicht zu unleserlichen und fehlerträchtigen Programmen führen.

Der Rückgabewert des Ausdrucks `++u` ist `u+1`. Zusätzlich wird als Nebeneffekt die Variable `u` inkrementiert und erhält den Wert `u+1`.

In Java gibt es verschiedene **Sorten von Nebeneffekten**:

- Nebeneffekte bei **Zuweisungen**,
- Nebeneffekte von **Operatoren**,
- Nebeneffekte bei **Methoden**, die nicht nur lesend, sondern auch schreibend auf Variablen zugreifen,
- und Nebeneffekte bei Ausdrücken, die **mit `new` ein Objekt erzeugen**.

Ausführungszeitpunkt von Nebeneffekten

Die Berechnung von Ausdrücken kann mit Nebeneffekten verbunden sein.

In Java wird jeder **Operand eines Operators vollständig ausgewertet, bevor irgendein Teil der Operation begonnen wird**. Damit haben (mit Ausnahme der Operatoren `&&`, `||` und `? :`) **vor einer Operation** die **Nebeneffekte der Operanden** stattgefunden.

In Java werden die **Operanden eines Operators strikt von links nach rechts ausgewertet**. Dies bedeutet, dass der Nebeneffekt des linken Operanden vor der Bewertung des rechten Operanden erfolgt ist.

In Java werden die **aktuellen Parameter eines Methodenaufrufs von links nach rechts bewertet**. Dies bedeutet, dass nach der Bewertung eines aktuellen Parameters ein Nebeneffekt dieses aktuellen Parameters stattgefunden hat und erst dann der rechts davon stehende aktuelle Parameter bewertet wird.

Ein Beispiel für die Auswertungsreihenfolge der aktuellen Parameter bei einem Methodenaufruf wird in Kapitel 9.2.6 gegeben.

Ein Nebeneffekt hat stattgefunden nach der Auswertung der folgenden Ausdrücke:

- Initialisierungsausdruck einer manuellen Initialisierung,
- Ausdruck in einer Ausdrucksanweisung,
- Bedingung in einer `if`-Anweisung (siehe Kapitel 8.2.1),
- Selektionsausdruck in einer `switch`-Anweisung (siehe Kapitel 8.2.3),
- Bedingung einer `while`- oder `do while`-Schleife (siehe Kapitel 8.3.5)),
- Initialierungsklausel in Form eines einzelnen Ausdrucks oder einer Ausdrucksliste sowie boolescher Ausdruck und Aktualisierungs-Ausdrucksliste der `for`-Schleife (siehe Kapitel 8.3.2),
- Ausdruck einer `return`-Anweisung (siehe Kapitel 9.2.3)).

7.4 Auswertungsreihenfolge

Wie in der Mathematik spielt es auch bei Java eine Rolle, in welcher Reihenfolge ein Ausdruck berechnet wird. Genau wie in der Mathematik gilt auch in Java die Regel "Punkt vor Strich", weshalb 5 + 2 * 3 gleich 11 und nicht gleich 21 ist. Allerdings gibt es in Java sehr viele Operatoren. Daher muss für alle Operatoren eindeutig festgelegt werden, welcher im Zweifelsfall Priorität hat.

7.4.1 Einstellige und mehrstellige Operatoren

Die Auswertung eines Ausdrucks mit Operatoren[120] wie `++`, `+`, `*` etc. wird nach folgenden Regeln durchgeführt:

[120] Methodenaufruf-, Array-Index- und Punktoperator werden hier noch nicht betrachtet.

1. Wie in der Mathematik werden als erstes **Teilausdrücke in Klammern** ausgewertet. Der Wert und Typ eines Ausdrucks ändert sich nicht, wenn er in Klammern gesetzt wird. So sind beispielsweise die beiden Zuweisungen `a = b` und `a = (b)` identisch.
2. Anschließend werden **Ausdrücke mit unären Operatoren** ausgewertet. Unäre Operatoren werden **von rechts nach links** angewendet. Dies bedeutet, dass `-~x` gleichbedeutend ist mit `-(~x)`. Anzumerken ist, dass der hier verwendete unäre Operator `~` alle Bits seines Operanden invertiert.
3. Abschließend werden Teilausdrücke mit **mehrstelligen Operatoren** ausgewertet.

Unäre Operatoren haben alle dieselbe Priorität. Die Abarbeitung mehrstelliger Operatoren erfolgt nach der Prioritätstabelle der Operatoren, wenn Operatoren verschiedener Prioritäten nebeneinander stehen.

Die Prioritätstabelle der Operatoren ist in Kapitel 7.6.8 zu finden.

Bei Operatoren verschiedener Priorität erfolgt zuerst die Abarbeitung der Operatoren mit höherer Priorität. Bei gleicher Priorität entscheidet die **Assoziativität der Operatoren**, ob die Verknüpfung von links nach rechts oder von rechts nach links erfolgt.

Der Begriff der Assoziativität wird in Kapitel 7.4.2 behandelt.

Durch das Setzen von Klammern (Regel 1) kann man von der festgelegten Reihenfolge abweichen.

7.4.2 Mehrstellige Operatoren gleicher Priorität

Unter **Assoziativität** versteht man die Reihenfolge, wie Operatoren und Operanden **verknüpft** werden, wenn Operanden durch Operatoren **derselben Priorität** (**Vorrangstufe**) verknüpft werden.

Ist ein Operator **rechtsassoziativ** so wird eine Verkettung von Operatoren und Operanden von rechts nach links abgearbeitet, bei **Linksassoziativität** dementsprechend von links nach rechts. Das folgende Bild visualisiert die Verknüpfungsreihenfolge bei einem linksassoziativen Operator:

Bild 7-3 Verknüpfungsreihenfolge bei einem linksassoziativen Operator `op`

Im Beispiel von Bild 7-3 wird also zuerst der linke Operator `op` auf die Operanden `A` und `B` angewendet, als zweites wird dann die Verknüpfung `op` mit `C` durchgeführt.

Da Additions- und Subtraktionsoperator linksassoziativ sind und dieselbe Priorität haben, wird **beispielsweise** der Ausdruck `a - b + c` wie `(a - b) + c` verknüpft und nicht wie `a - (b + c)`.

Es gibt **zwei Möglichkeiten** für die Verknüpfung des Ausdrucks `a - b + c`:

Fall 1: `a - b + c` wird verknüpft wie `(a - b) + c`.

Also erst `a` und b verknüpfen zu `a - b`, dann `(a - b)` und `c` verknüpfen zu `(a - b) + c`. Damit kam der linke Operator vor dem rechten an die Reihe. Die Linksassoziativität wurde nicht verletzt.

Fall 2: `a - b + c` wird verknüpft wie `a - (b + c)`.

Hier werden zuerst die Operanden `b` und `c` durch den Additionsoperator verknüpft. Die Linksassoziativität ist damit verletzt, da als erstes der Operator – hätte dran kommen müssen.

Einige der in Java vorhandenen Operatoren sind jedoch nicht links-, sondern rechtsassoziativ (siehe Zuweisungsoperator).

7.4.3 Bewertungsreihenfolge von Operanden

In Java werden die **Operanden eines Operators** strikt von **links nach rechts ausgewertet**.

Da in Java die Bewertungsreihenfolge von Operanden definiert ist, ist in Java auch ein Ausdruck

```
a++ - a
```

deterministisch und damit zulässig. Vor der binären Operation muss der linke Operand vollständig bewertet sein, d. h., der Nebeneffekt muss stattgefunden haben. Dass der linke Operand vollständig bewertet sein muss, bedeutet, dass sein Wert zwischengespeichert werden muss, um anschließend in einer Operation – hier der Subtraktion – verwendet zu werden. Der Rückgabewert des Operanden `a++` ist `a`, nach Abarbeitung des Nebeneffekts ist der Wert von `a` um `1` erhöht. Dies bedeutet, dass `a++ - a` den Wert `-1` hat.

In Java ist festgelegt, dass jeder Operand eines Operators mit Ausnahme der Operatoren `&&`, `||` und `? :` vollständig ausgewertet wird, bevor irgendein Teil der Operation begonnen wird.

7.5 L-Werte und R-Werte

Die Begriffe L-Wert und R-Wert sind in C geläufig. Gosling [Gos97] spricht in Java anstelle des **L-Werts** von **Variablen**, statt **R-Wert** einfach von **Wert**. Aus Gründen der Präzision werden in diesem Buch die Begriffe L- und R-Wert beibehalten.

Einen Ausdruck, der eine Variable im Speicher bezeichnet, nennt man einen **L-Wert** (**lvalue** oder **left value**).

Ein **L-Wert** zeichnet sich dadurch aus, dass er einen Speicherplatz irgendwo **im Arbeitsspeicher** besitzt.

In Java stellt der Name `var` einer lokalen Variablen einen solchen Ausdruck dar. Andere Möglichkeiten für L-Werte sind der Name einer Instanzvariablen oder einer Klassenvariablen, der Zugriff auf ein Arrayelement, eine Variable eines Schnittstellentyps oder eine Variable eines Aufzählungstyps.

Das 'L' steht für links (left) und deutet darauf hin, dass dieser Ausdruck links vom Zuweisungsoperator `=` stehen kann. Natürlich kann ein L-Wert auch rechts vom Zuweisungsoperator stehen wie in

```
a = b
```

wobei `a` und `b` Variablen sind.

Steht ein Variablenname rechts neben dem Zuweisungsoperator, so wird über den Variablennamen der Wert an der entsprechenden Speicherstelle ausgelesen, d. h., es interessiert hier nur sein R-Wert. Links neben dem Zuweisungsoperator muss immer ein L-Wert stehen, da man eine Speicherstelle benötigt, die den Wert der Zuweisung aufnehmen kann.

Ist ein Ausdruck kein L-Wert, so ist er ein **R-Wert** (**rvalue** oder **right value**) und kann nicht links, sondern nur rechts vom Zuweisungsoperator stehen. Einem R-Wert kann man keinen Wert zuweisen, da er keine feste Speicherstelle besitzt.

Das folgende Bild gibt Beispiele für L- und R-Werte:

Bild 7-4 Beispiele für L- und R-Werte

Des Weiteren wird zwischen modifizierbarem und **nicht modifizierbarem L-Wert** unterschieden.

Die L-Werte in Bild 7-4 sind modifizierbar.

Ein Ausdruck, welcher eine `final`-Variable bezeichnet, ist ein spezieller L-Wert, der, solange er noch nicht initialisiert wurde, zu den modifizierbaren L-Werten zählt und nach der Initialisierung einen nicht modifizierbaren L-Wert repräsentiert.

Auf der linken Seite einer Zuweisung darf nur ein modifizierbarer L-Wert stehen, jedoch weder ein R-Wert noch ein nicht modifizierbarer L-Wert. Bestimmte Operatoren können nur auf modifizierbare L-Werte angewendet werden wie z. B. der Inkrementoperator `++` oder der Dekrementoperator `--`. `5++` ist falsch, `i++`, wobei `i` eine Variable darstellt, ist jedoch korrekt.

7.6 Zusammenstellung der Operatoren

In den folgenden Kapiteln wird ein unärer Operator stets mit seinem Operanden bzw. werden binäre und tertiäre Operatoren mit ihren Operanden gezeigt. Es wird also stets die ganze Operation vorgestellt.

7.6.1 Einstellige arithmetische Operatoren

Im Folgenden werden die einstelligen (unären) **Operatoren**

- positiver Vorzeichenoperator: `+A`
- negativer Vorzeichenoperator: `-A`
- Postfix-Inkrementoperator: `A++`
- Präfix-Inkrementoperator: `++A`
- Postfix-Dekrementoperator: `A--`
- Präfix-Dekrementoperator: `--A`

anhand von Beispielen vorgestellt. Die Inkrement- und Dekrementoperatoren können seit der Version Java 5 auch auf Referenzen auf Objekte numerischer Wrapper-Klassen angewendet werden.

Positiver Vorzeichenoperator: `+A`

Der positive Vorzeichenoperator wird selten verwendet, da er lediglich den Wert seines Operanden wiedergibt. Es gibt keine Nebeneffekte.

Beispiel:

```
+a                    // +a hat denselben Rückgabewert wie a.
```

Negativer Vorzeichenoperator: `-A`

Will man den Wert eines Operanden mit umgekehrtem Vorzeichen erhalten, so ist der negative Vorzeichenoperator von Bedeutung. Es gibt keine Nebeneffekte.

Beispiel:

```
-a                    // -a hat vom Betrag denselben Rückgabe-
                      // wert wie a. Der Rückgabewert hat aber
                      // das umgekehrte Vorzeichen.
```

Postfix-Inkrementoperator: `A++`

Der Rückgabewert ist der unveränderte Wert des Operanden. Als **Nebeneffekt** wird der Wert des Operanden um 1 inkrementiert. Der Inkrementoperator kann auf modifizierbare L-Werte eines ganzzahligen oder eines Gleitpunkt-Typs – nicht jedoch auf nicht modifizierbare L-Werte und R-Werte – angewandt werden.

Beispiele:

```
a = 1
b = a++               // Erg.: b = 1, Nebeneffekt: a = 2
```

Präfix-Inkrementoperator: `++A`

Der Rückgabewert ist der um 1 inkrementierte Wert des Operanden. Als **Nebeneffekt** wird der Wert des Operanden um 1 inkrementiert. Der Inkrementoperator kann nur auf modifizierbare L-Werte eines ganzzahligen oder eines Gleitpunkt-Typs angewandt werden.

Beispiele:

```
a = 1
b = ++a               // Erg.: b = 2, Nebeneffekt: a = 2
```

Postfix-Dekrementoperator: `A--`

Der Rückgabewert ist der unveränderte Wert des Operanden. Als **Nebeneffekt** wird der Wert des Operanden um 1 dekrementiert. Der Dekrementoperator kann nur auf modifizierbare L-Werte eines ganzzahligen oder eines Gleitpunkt-Typs angewandt werden.

Beispiele:

```
a = 1
b = a--               // Erg.: b = 1, Nebeneffekt: a = 0
```

Präfix-Dekrementoperator: `--A`

Der Rückgabewert ist der um 1 dekrementierte Wert des Operanden. Als **Nebeneffekt** wird der Wert des Operanden um 1 dekrementiert. Der Dekrementoperator kann nur auf modifizierbare L-Werte eines ganzzahligen oder eines Gleitpunkt-Typs angewandt werden.

Beispiele:

```
a = 1
b = --a                // Erg.: b = 0, Nebeneffekt: a = 0
```

7.6.2 Zweistellige arithmetische Operatoren

Im Folgenden werden die zweistelligen **Operatoren**

- Additionsoperator: `A + B`
- Subtraktionsoperator: `A - B`
- Multiplikationsoperator: `A * B`
- Divisionsoperator: `A / B`
- Restwertoperator: `A % B`

anhand von Beispielen vorgestellt. Seit der Version Java 5 gelten diese Operatoren auch für Referenzen auf Objekte numerischer Wrapper-Klassen, da hier beim Anwenden der arithmetischen Operatoren ein automatisches Unboxing erfolgt.

Additionsoperator: `A + B`

Wendet man den zweistelligen Additionsoperator auf seine Operanden an, so ist der Rückgabewert die Summe der Werte der beiden Operanden. Es gibt hier keine Nebeneffekte.

Beispiele:

```
6 + (4 + 3)
a + 1.1E1
PI + 1           // PI ist eine symbolische Konstante.
ref.meth() + 1   // Hier muss der Aufruf der Methode meth()
                 // einen arithmetischen Wert zurückgeben.
```

Subtraktionsoperator: `A - B`

Wendet man den zweistelligen Subtraktionsoperator auf die Operanden `A` und `B` an, so ist der Rückgabewert die Differenz der Werte der beiden Operanden. Es gibt keine Nebeneffekte.

Beispiel:

```
6 - 4
```

Multiplikationsoperator: `A * B`

Es wird die Multiplikation des Wertes von `A` mit dem Wert von `B` durchgeführt. Es gelten hier die "üblichen" Rechenregeln, d. h. Klammerung vor Punkt und Punkt vor Strich. Deshalb wird im Beispiel `3 * (5 + 3)` zuerst der Ausdruck `(5 + 3)` ausgewertet, der dann anschließend mit `3` multipliziert wird. Es gibt keine Nebeneffekte.

Beispiele:

```
3 * 5 + 3        // Erg.: 18
3 * (5 + 3)      // Erg.: 24
```

Divisionsoperator: `A / B`

Bei der Verwendung des Divisionsoperators mit ganzzahligen Operanden ist das Ergebnis wieder eine ganze Zahl. Der Nachkommateil des Ergebnisses wird abgeschnitten.

In Java führt eine Division durch `0` nicht wie in vielen anderen Sprachen zum Absturz des Programms. Bei der Ganzzahldivision durch `0` wird eine Ausnahme[121] vom Typ **`ArithmeticException`** ausgelöst. Bei der Gleitpunktdivision durch `0.0` wird als Ergebnis `Infinity` mit Berücksichtigung des Vorzeichens geliefert.

Ist bei einer **ganzzahligen Division** entweder der Zähler oder der Nenner negativ, so ist das Ergebnis negativ. Dabei bestimmt sich der Quotient vom Betrag her nach der Vorschrift, dass der Quotient die größtmögliche Ganzzahl ist, für die gilt: |Quotient * Nenner| <= |Zähler|. Wird also `-7` durch `2` geteilt, so ist das Ergebnis `-3` mit dem Rest `-1`, da `|-3 * 2| <= |-7|` ist.

Ist mindestens ein Operand eine `double`- oder `float`-Zahl, d. h. eine Gleitpunktzahl, so ist das Ergebnis eine **Gleitpunktzahl**. Es gibt keine Nebeneffekte.

Beispiele:

```
5 / 5           // Erg.: 1
5 / 3           // Erg.: 1
11.0 / 5        // Erg.: 2.2
```

Beispiel für die Division durch `0` bei Gleitpunktzahlen:

```
// Datei: DivisionTest.java

public class DivisionTest
{
   public static void main (String[] args)
   {
      float ergebnis;
      float nenner;
      float zaehler;
      zaehler = 20;
      nenner = 0;
      ergebnis = zaehler / nenner;
      System.out.println (zaehler + " / " + nenner + " = "
                          + ergebnis);
      zaehler = 0;
      nenner = 0;
      ergebnis = zaehler / nenner;

      System.out.println (zaehler + " / " + nenner + " = "
                          + ergebnis);
   }
}
```

[121] Ausnahmen (Exceptions) werden in Kapitel 13 behandelt.

Die Ausgabe des Programms ist:

```
20.0 / 0.0 = Infinity
0.0 / 0.0 = NaN
```

Die Division durch `0` ergibt mathematisch korrekt unendlich (`Infinity`). Das Ergebnis der Division von `0` durch `0` ergibt keine Zahl (`NaN` – Not a Number).

Restwertoperator: `A % B`

Der **Restwertoperator** oder **Modulo-Operator** gibt für ganzzahlige Operanden `A` und `B` den Rest bei der ganzzahligen Division des Operanden `A` durch den Operanden `B` an. Das Ergebnis der Restwert-Operation `A % B` ergibt sich aus: `A - (A / B) * B`. Es gibt keine Nebeneffekte.

Beispiele:

```
  5  %   3     =  2
  8  %   4     =  0
  3  %   7     =  3
(-7) %   2     = -1      // denn (-7) / 2 ergibt -3
(-7) % (-2)    = -1      // denn (-7) / (-2) ergibt 3
  7  % (-2)    =  1      // denn 7 / (-2) ergibt -3
  7  %   2     =  1      // denn 7 / 2 ergibt 3
```

Die Restwertbildung für den Nenner `0` führt nicht zum Absturz des Programms. Es wird eine Ausnahme[122] vom Typ `ArithmeticException` ausgelöst.

In Java gibt es den Restwertoperator nicht nur für ganzzahlige Operanden, sondern auch für Gleitpunktoperanden. Hierfür wird auf [Gos97] verwiesen.

7.6.3 Zuweisungsoperatoren

Zu den Zuweisungsoperatoren gehören

der **einfache Zuweisungsoperator**: `A = B`

sowie die **kombinierten Zuweisungsoperatoren**:

Additions-Zuweisungsoperator:	`A += B`
Subtraktions-Zuweisungsoperator:	`A -= B`
Multiplikations-Zuweisungsoperator:	`A *= B`
Divisions-Zuweisungsoperator:	`A /= B`
Restwert-Zuweisungsoperator:	`A %= B`

[122] Ausnahmen (Exceptions) werden in Kapitel 13 behandelt.

Bitweises-UND-Zuweisungsoperator:	`A &= B`
Bitweises-ODER-Zuweisungsoperator:	`A |= B`
Bitweises-Exklusiv-ODER-Zuweisungsoperator:	`A ^= B`
Linksschiebe-Zuweisungsoperator:	`A <<= B`
Rechtsschiebe-Zuweisungsoperator:	`A >>= B`
Vorzeichenloser Rechtsschiebe-Zuweisungsoperator:	`A >>>= B`

Dabei darf zwischen den Zeichen eines kombinierten Zuweisungsoperators kein Leerzeichen stehen. Die Operanden eines kombinierten Zuweisungsoperators müssen einen einfachen Datentyp haben oder Referenzen auf Objekte numerischer Wrapper-Klassen sein. Die einzige Ausnahme ist der Operator `+=`. Hier kann der linke Operand vom Typ `String` und – in diesem Fall – der rechte Operand von jedem beliebigen Typ sein.

Zuweisungsoperator `A = B`

Der Zuweisungsoperator wird in Java als binärer Operator betrachtet und liefert als **Rückgabewert** den **Wert des rechten Operanden** – es handelt sich bei einer **Zuweisung** also um einen **Ausdruck**. Zuweisungen können wiederum in Ausdrücken weiter verwendet werden.

Bei einer Zuweisung wird zusätzlich zur Erzeugung des Rückgabewertes – und das ist der **Nebeneffekt** – dem linken Operanden der Wert des rechten Operanden zugewiesen.

Sonst wäre es ja auch keine Zuweisung! Im Übrigen muss der linke Operand `A` ein modifizierbarer L-Wert sein. Wie zu sehen ist, sind dadurch auch **Mehrfachzuweisungen** möglich. Da der Zuweisungsoperator rechtsassoziativ ist, wird der Ausdruck `a = b = c` von rechts nach links verknüpft. Er wird also abgearbeitet wie `a = (b = c)`. Dies wird im Folgenden gezeigt:

1. Schritt: `a = (b = c)`

   Rückgabewert `c`
   Nebeneffekt: In der Speicherstelle `b` wird der Wert von `c` abgelegt, d. h., `b` nimmt den Wert von `c` an.

2. Schritt: `a = c`

   Rückgabewert `c`
   Nebeneffekt: In der Speicherstelle `a` wird der Wert von `c` abgelegt.

Zuweisungsoperatoren haben eine geringe Priorität, sodass man beispielsweise bei einer Zuweisung `b = x + 3` den Ausdruck `x + 3` nicht in Klammern setzen muss. Erst erfolgt die Auswertung des arithmetischen Ausdrucks, dann erfolgt die Zuweisung.

Die Prioritäten von Operatoren werden in Kapitel 7.6.8 behandelt.

Der Ausdruck rechts des Zuweisungsoperators wird **implizit in den Typ der Variablen links des Zuweisungsoperators gewandelt**, es sei denn, die Typen sind identisch oder die implizite Typkonvertierung ist nicht möglich.

Die implizite Typkonvertierung wird in Kapitel 7.7.2 behandelt.

Beispiele:

```
b = 1 + 3
c = b = a                // Mehrfachzuweisung
Math.abs (x = 1.4)       // Zuweisung als aktueller Parameter
                         // beim Aufruf der Klassenmethode
                         // abs() der Klasse Math
```

Additions-Zuweisungsoperator: `A += B`

Der Additions-Zuweisungsoperator ist – wie der Name schon verrät – ein zusammengesetzter Operator. Zum einen wird die Addition `A + (B)` durchgeführt. Der Rückgabewert dieser Addition ist `A + (B)`. Zum anderen erhält die Variable `A` als Nebeneffekt den Wert dieser Addition zugewiesen. Damit entspricht der Ausdruck `A += B` semantisch genau dem Ausdruck `A = A + (B)`. Die Klammern sind nötig, da `B` selber ein Ausdruck wie z. B. `b = 3` sein kann. Es wird also zuerst der Ausdruck `B` ausgewertet, bevor `A + (B)` berechnet wird.

Beispiel:

```
a += 1                   // hat den gleichen Effekt wie ++a
```

Wie zuvor erwähnt, kann der Additions-Zuweisungsoperator auf Referenzen auf Objekte der Klasse `String` angewandt werden. Es können Ausdrücke einfacher Datentypen wie `int`, `float` oder `boolean` über den Operator `+=` mit einer Referenz auf ein `String`-Objekt verknüpft werden.

Beispiele:

```
s1 = "Hallo "            // s1 zeigt auf den String "Hallo "
s2 = "Myriam "           // s2 zeigt auf den String "Myriam "
s1 += s2                 // s1 zeigt jetzt auf den neuen
                         // String "Hallo Myriam ".
s1 += 2                  // s1 zeigt jetzt auf den neuen
                         // String "Hallo Myriam 2".
```

Sonstige kombinierte Zuweisungsoperatoren

Für die sonstigen kombinierten Zuweisungsoperatoren gilt das Gleiche wie für den Additions-Zuweisungsoperator. Außer der konventionellen Schreibweise:

```
A = A op (B)
```

gibt es die zusammengesetzte kurze Schreibweise:

```
A op= B
```

Beispiele:

```
a -= 1                  //  a = a - 1
b *= 2                  //  b = b * 2
c /= 5                  //  c = c / 5
d %= 5                  //  d = d % 5
a &= 8                  //  a = a & 8   Bitoperator
b |= 4                  //  b = b | 4   Bitoperator
c ^= d                  //  c = c ^ d   Bitoperator
a <<= 1                 //  a = a << 1  Bitoperator
b >>= 1                 //  b = b >> 1  Bitoperator
b >>>= 5                //  b = b >>> 5 Bitoperator
```

Bit-Operatoren werden in Kapitel 7.6.6 besprochen.

7.6.4 Relationale Operatoren

In diesem Kapitel werden anhand von Beispielen die folgenden zweistelligen relationalen **Operatoren** vorgestellt:

Gleichheitsoperator:	`A == B`
Ungleichheitsoperator:	`A != B`
Größeroperator:	`A > B`
Kleineroperator:	`A < B`
Größergleichoperator:	`A >= B`
Kleinergleichoperator:	`A <= B`

Relationale Operatoren werden auch als **Vergleichsoperatoren** bezeichnet. Nebeneffekte treten bei Vergleichsoperationen nicht auf.

Die Priorität der Operatoren `==` und `!=` ist kleiner als die der Operatoren `>`, `>=`, `<` und `<=`.

Besitzen die Operanden unterschiedliche, aber verträgliche Datentypen, werden implizite Typkonvertierungen durchgeführt. Bei den Vergleichsoperatoren `>`, `>=`, `<` und `<=` ist darauf zu achten, dass der Typ der Operanden nur ein numerischer Typ bzw. eine numerische Wrapper-Klasse sein darf. Hat einer der Operanden einen anderen Typ, so gibt der Compiler eine Fehlermeldung aus. Der Rückgabewert von Vergleichsoperationen ist immer vom Datentyp `boolean`. Wenn ein Vergleich falsch ist, ist der Rückgabewert `false`, wenn er wahr ist, ist er `true`.

Gleichheitsoperator: `A == B`

Mit dem Gleichheitsoperator wird überprüft, ob der Wert des linken Operanden mit dem Wert des rechten Operanden übereinstimmt. Verglichen werden können zwei Operanden von einem numerischen Typ, zwei Operanden vom Typ `boolean` oder zwei Operanden eines Referenztyps bzw. vom Typ `null`.

Bei Referenztypen (auch bei Referenzen auf Objekte von Wrapper-Klassen) wird verglichen, ob die Referenzen gleich sind – mit dem Vergleichsoperator lässt sich also nicht prüfen, ob zwei Objekte inhaltlich gleich sind.

Im Falle von Aufzählungstypen können Aufzählungskonstanten mit dem Gleichheitsoperator verglichen werden.

Ist ein Vergleich wahr, hat der Rückgabewert den Wert `true`. Andernfalls, d. h., wenn ein Vergleich falsch ist, hat der Rückgabewert den Wert `false`.

Beispiele:

```
3 == 3          // Erg.: true
2 == 3          // Erg.: false
```

Ein folgenschwerer Fehler ist in Java, statt des Gleichheitsoperators `==` versehentlich den Zuweisungsoperator `=` anzuschreiben. Ein solches Programm ist oft kompilier- und lauffähig, erzeugt aber andere Ergebnisse als erwartet.

Programmiert man aber **defensiv** und schreibt bei einem Vergleich einer Konstanten mit einer Variablen die Konstante stets links und die Variable rechts, also z. B.

```
true == a
```

so merkt der Compiler den Fehler, da einer Konstanten kein Wert zugewiesen werden kann, weil sie kein L-Wert ist.

Die **defensive Programmierung** beschreibt einen Ansatz, um die Qualität des Quellcodes zu verbessern. Dabei setzt man sich das Ziel, die Robustheit des Codes zu erhöhen und zufällige Fehler bei Programmänderungen zu verhindern.

Zur defensiven Programmierung gehört auch, dass unzulässige Benutzereingaben konsequent abgewiesen werden, was ein Abstürzen verhindert (Robustheit), und dass man Code-Konstrukte so formuliert, dass sie nicht fehlerträchtig sind und leicht erweitert werden können. So sollte bei Selektionen und Iterationen stets ein Block verwendet werden, auch, wenn der Block nur eine einzige Anweisung enthält. Damit sind Erweiterungen um Anweisungen nicht fehlerträchtig, da die geschweiften Klammern schon da sind. Ein Beispiel für eine Selektion befindet sich im Kapitel 8.2.1.

Ungleichheitsoperator: `A != B`

Mit dem Ungleichheitsoperator wird überprüft, ob der Wert des linken Operanden ungleich dem Wert des rechten Operanden ist. Es können dieselben Operanden wie

im Falle des Gleichheitsoperators verwendet werden. Bei Ungleichheit hat der Rückgabewert den Wert `true`. Andernfalls hat der Rückgabewert den Wert `false`.

Beispiele:

```
5 != 5          // Erg.: false
3 != 5          // Erg.: true
```

Größeroperator: `A > B`

Mit dem Größeroperator wird überprüft, ob der Wert des linken Operanden größer als der Wert des rechten Operanden ist. Ist der Vergleich wahr, so hat der Rückgabewert den Wert `true`. Andernfalls hat der Rückgabewert den Wert `false`.

Beispiele:

```
5 > 3           // Erg.: true
3 > 3           // Erg.: false
```

Kleineroperator: `A < B`

Mit dem Kleineroperator wird überprüft, ob der Wert des linken Operanden kleiner als der Wert des rechten Operanden ist. Ist der Vergleich wahr, hat der Rückgabewert den Wert `true`. Andernfalls hat der Rückgabewert den Wert `false`.

Beispiel:

```
5 < 5           // Erg.: false
```

Größergleichoperator: `A >= B`

Der Größergleichoperator ist aus den Zeichen > und = zusammengesetzt. Der Größergleichoperator liefert genau dann den Rückgabewert `true`, wenn entweder der Wert des linken Operanden größer als der Wert des rechten Operanden ist oder der Wert des linken Operanden dem Wert des rechten Operanden entspricht.

Beispiele:

```
2 >= 1          // Erg.: true
1 >= 1          // Erg.: true
```

Kleinergleichoperator: `A <= B`

Der Kleinergleichoperator ist aus den Zeichen < und = zusammengesetzt. Der Kleinergleichoperator liefert genau dann den Rückgabewert `true`, wenn entweder der Wert des linken Operanden kleiner als der Wert des rechten Operanden ist oder der Wert des linken Operanden dem Wert des rechten Operanden entspricht. Ansonsten ist der Rückgabewert `false`.

Beispiele:

```
10 <= 11        // Erg.: true
11 <= 11        // Erg.: true
```

7.6.5 Logische Operatoren

In diesem Kapitel werden anhand von Beispielen die folgenden logischen **Operatoren** vorgestellt:

- Operatoren für das logische UND: `A && B` bzw. `A & B`
- Operatoren für das logische ODER: `A || B` bzw. `A | B`
- Logischer Exklusiv-ODER-Operator: `A ^ B`
- Logischer Negationsoperator (unär): `!A`

Die Operatoren für das logische UND/ODER sowie das logische Exklusiv-ODER sind zweistellig, der logische Negationsoperator ist einstellig. Mit diesen Operatoren lassen sich logische Verknüpfungen von Ausdrücken durchführen. Wie schon erwähnt, können die Operanden selber zusammengesetzte Ausdrücke sein. Von den logischen Operatoren hat der Negationsoperator die höchste Priorität, der Operator `||` für das logische ODER die geringste (siehe Kapitel 7.6.8).

Die logischen Operatoren `&&`, `||`, `!` können nur auf Operanden vom Typ `boolean` – und seit der Version Java 5 auch auf Operanden vom Typ `Boolean` – angewandt werden. Andere Typen führen zu einer Fehlermeldung des Compilers. Der Ergebnistyp ist ebenfalls vom Typ `boolean`.

Die Operatoren `&`, `|`, `^` können als logische Operatoren auf Operanden vom Typ `boolean` (bzw. `Boolean`) und als logische Bit-Operatoren (siehe Kapitel 7.6.6.1) auch auf Operanden numerischer Datentypen (bzw. numerischer Wrapper-Klassen) angewandt werden.

Operatoren für das logische UND: `A && B` und `A & B`

Java bietet zwei verschiedene Operatoren für das logische UND an: den Operator `&&` und den Operator `&`. Beide Operatoren haben eine identische Wahrheitstabelle. Sie liefern genau dann den Rückgabewert `true`, wenn beide Operanden den Wahrheitswert `true` haben. Ansonsten ist der Rückgabewert `false`. Die folgende Tabelle zeigt die Wahrheitstabelle für das logische UND:

A	B	A UND B
false	false	false
false	true	false
true	false	false
true	true	true

Tabelle 7-1 Wahrheitstabelle für das logische UND

Die Wahrheitstabelle in Tabelle 7-1 wird wie folgt interpretiert: Der logische Ausdruck `A UND B` ist nur dann `true`, wenn der Ausdruck `A` und der Ausdruck `B` gleich `true` ist.

Beispiele:

```
true && false        // Erg.: false
true && true         // Erg.: true
```

Wird der Operator `&` zwischen zwei Operanden verwendet, so wird der rechte Operand immer ausgewertet, egal ob der linke Operand `true` oder `false` ist. Wird dagegen der Operator `&&` verwendet, so wird der rechte Ausdruck nur dann ausgewertet, wenn der linke Ausdruck `true` ist. Dies ist zu beachten, wenn die Operanden Nebeneffekte beinhalten.

Operatoren für das logische ODER: `A || B` und `A | B`

Java bietet zwei verschiedene Operatoren für das logische ODER an: den Operator `||` und den Operator `|`. Beide Operatoren haben eine identische Wahrheitstabelle. Ein Operator für das logische ODER liefert genau dann den Rückgabewert `true`, wenn der linke oder der rechte Operand oder beide Operanden den Wahrheitswert `true` haben. Ansonsten ist der Rückgabewert `false`. Die folgende Tabelle zeigt die Wahrheitstabelle für das logische ODER:

A	B	A ODER B
false	false	false
false	true	true
true	false	true
true	true	true

Tabelle 7-2 Wahrheitstabelle für das logische ODER

Beispiele:

```
false || true          // Erg.: true
false || false         // Erg.: false
```

Wird der Operator `|` zwischen zwei Operanden verwendet, so wird der rechte Operand immer ausgewertet, egal ob der linke Operand `true` oder `false` ist. Wird dagegen der Operator `||` verwendet, so wird der rechte Ausdruck nur dann ausgewertet, wenn der linke Ausdruck `false` ist. Dies ist zu beachten, wenn die Operanden Nebeneffekte beinhalten.

Logischer Exklusiv-ODER-Operator: `A ^ B`

Ein Operator für das logische Exklusiv-ODER liefert genau dann den Rückgabewert `true`, wenn entweder der linke oder der rechte Operand den Wert `true` haben. Haben beide Operanden denselben Wert (beide `true` oder beide `false`), so ist das Ergebnis `false`. Im Folgenden die Wahrheitstabelle für das logische Exklusiv-ODER:

A	B	A ^ B
false	false	false
false	true	true
true	false	true
true	true	false

Tabelle 7-3 Wahrheitstabelle für das logische Exklusiv-ODER `A ^ B`

Beispiele:

```
false ^ true          // Erg.: true
false ^ false         // Erg.: false
true  ^ true          // Erg.: false
```

Logischer Negationsoperator: `!A`

Mit dem einstelligen Negationsoperator werden **Wahrheitswerte negiert**, d. h., aus `true` wird `false` und aus `false` wird `true`. Wird der Negationsoperator zweimal auf seinen Operanden angewendet, bleibt der Wahrheitswert unverändert. Im Folgenden die Wahrheitstabelle für die Negation:

A	!A
true	false
false	true

Tabelle 7-4 Wahrheitstabelle für die Negation

Die Wahrheitstabelle wird folgendermaßen interpretiert: Der logische Ausdruck `!A` ist nur dann `true`, wenn der Ausdruck `A` `false` ist.

Beispiele:

```
!false                //  Erg.: true
!!true                //  Erg.: true
```

Priorität der logischen Operatoren

Die Operatoren für das logische UND/ODER haben eine sehr geringe Bindekraft. Die Vergleichsoperatoren haben eine höhere Priorität als die logischen Operatoren.

Deshalb sind Klammern für die Bewertung der Ausdrücke oft nicht notwendig. So entspricht `(a < b) && (c == d)` dem Ausdruck `a < b && c == d`. Die Klammern erhöhen lediglich die Übersichtlichkeit der Programme.

Verknüpfungsreihenfolge

Ausdrücke, die durch den UND-Operator `&&` verknüpft sind, werden von links nach rechts zusammengefasst. Dasselbe gilt für Ausdrücke, die durch den ODER-Operator `||` verknüpft werden.

Werden jedoch sowohl UND-Operator als auch ODER-Operator gemeinsam in einem Ausdruck verwendet, ist der UND-Operator höher priorisiert als der ODER-Operator. Dieses Verhalten ist auch für den bitweisen UND-Operator `&` und ODER-Operator `|` gültig.

Nebeneffekte

Nebeneffekte des rechten Operanden kommen bei den Operatoren `&` und `|` immer zum Tragen, bei den Operatoren `&&` und `||` nur, wenn der rechte Operand ausgewertet wird. Das kann dazu führen, dass Nebeneffekte der rechts stehenden Ausdrücke nicht mehr ausgeführt werden. Das zeigt das folgende Beispiel:

```
1 < 0 && 2 < a++  // a++ wird nie ausgeführt, da die
                  // Auswertung vorher beendet ist.
```

7.6.6 Bit-Operatoren

Java besitzt auch Operatoren zur Bit-Manipulation. Im Folgenden werden die vier **logischen Bit-Operatoren** und die drei **Shift-Operatoren** für Bits anhand von Beispielen vorgestellt.

Die vier logischen **Bit-Operatoren** sind:

- UND-Operator für Bits: `A & B`
- ODER-Operator für Bits: `A | B`
- Exklusiv-ODER-Operator für Bits: `A ^ B`
- Negationsoperator für Bits (unär): `~A`

Die drei **Shift-Operatoren** für Bits sind:

- Vorzeichenbehafteter Rechtsshift-Operator: `A >> B`
- Vorzeichenloser Rechtsshift-Operator: `A >>> B`
- Linksshift-Operator: `A << B`

Seit Einführung der Version Java 5 können diese Operatoren auch auf Referenzen auf Objekte vom Typ einer numerischen Wrapper-Klasse angewendet werden. Dabei findet ein automatisches Unboxing und Boxing statt.

7.6.6.1 Logische Bit-Operatoren

Bit-Operationen finden auf allen Bits der Operanden statt. Bei den Bit-Operationen werden jeweils die Bits der entsprechenden Position miteinander verknüpft.

Bits können bekanntermaßen zwei Zustände annehmen: 0 oder 1. Die 1 wird bei Bits in Java als `true` interpretiert, die 0 als `false`. Nebeneffekte treten bei den logischen Bit-Operatoren nicht auf.

UND-Operator für Bits: `A & B`

Die Operation "bitweises UND" findet auf allen Bits der Operanden statt. Dabei werden jeweils die Bits der entsprechenden Position miteinander verknüpft.

Die folgende Tabelle zeigt die entsprechenden Wahrheitswerte:

Bit n von `A`	Bit n von `B`	Bit n von `A & B`
0	0	0
0	1	0
1	0	0
1	1	1

Tabelle 7-5 Wahrheitstabelle für das bitweise UND

Diese Wahrheitstabelle wird folgendermaßen interpretiert: Bei der UND-Verknüpfung ist die 0 dominant, d. h., ist mindestens eines der Bits (Bit n von `A` oder Bit n von `B`) eine 0, so ist das Ergebnis 0 (`false`). Damit kann man **Bits in Bitmustern ausblenden**. Der logische UND-Operator für Bits hat eine höhere Priorität als der logische ODER-Operator für Bits.

Beispiele:

```
0 & 1                   //  0 & 1 = 0
14 & 1                  //  1110 & 0001 = 0000
var & var               //  var & var = var
```

ODER-Operator für Bits: `A | B`

Die Operation "bitweises ODER" findet auf allen Bits der Operanden statt, dabei werden jeweils die Bits der entsprechenden Position miteinander verknüpft.

Die folgende Tabelle zeigt die Wahrheitstabelle für das bitweise ODER:

Bit n von `A`	Bit n von `B`	Bit n von `A | B`
0	0	0
0	1	1
1	0	1
1	1	1

Tabelle 7-6 Wahrheitstabelle für das bitweise ODER

Die Wahrheitstabelle wird folgendermaßen interpretiert: Bei der ODER-Verknüpfung ist die 1 dominant, d. h., ist mindestens eines der Bits (Bit n von `A` oder Bit n von `B`) eine 1, so ist das Ergebnis 1 (`true`). Damit kann man Bits in Bitmustern einblenden.

Beispiele:

```
  0 | 1                 //  0 | 1 = 1
 14 | 1                 //  1110 | 0001 = 1111 = 15
var | 0                 //  var | 0 = var
```

Exklusiv-ODER-Operator für Bits: `A ^ B`

Die Operation "bitweises Exklusiv-ODER" findet auf allen Bits der Operanden statt, dabei werden jeweils die Bits der entsprechenden Position miteinander verknüpft.

Die folgende Tabelle zeigt die Wahrheitstabelle für das bitweise Exklusiv-ODER:

Bit n von `A`	Bit n von `B`	Bit n von `A ^ B`
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 7-7 Wahrheitstabelle für das bitweise Exklusiv-ODER

Die Wahrheitstabelle wird folgendermaßen interpretiert: Bei der Exklusiv-ODER-Verknüpfung ist das Ergebnis 1 (`true`), wenn entweder Bit n von Operand `A` oder Bit n von Operand `B` eine 1 ist. Haben beide zu vergleichende Bits denselben Wert (beide 0 oder beide 1), so ist das Ergebnis der Exklusiv-ODER-Verknüpfung gleich 0.

Beispiele:

```
0 ^ 1                  //  0 ^ 1 = 1
14 ^ 1                 //  1110 ^ 0001 = 1111 = 15
var ^ 0                //  var ^ 0 = var
14 ^ 3                 //  1110 ^ 0011 = 1101 = 13
                       //  Bit 0 und Bit 1 von 1110 invertieren
```

Negationsoperator für Bits: ~A

Die Operation einer "bitweisen Negation" findet auf allen Bits des Operanden statt.

Die folgende Tabelle zeigt die Wahrheitstabelle für die bitweise Negation:

Bit n von `A`	Bit n von `~A`
0	1
1	0

Tabelle 7-8 Wahrheitstabelle für die bitweise Negation

Die Wahrheitstabelle wird folgendermaßen interpretiert: Bei der Negation für Bits wird jedes Bit invertiert. Aus der 0 wird durch Negation eine 1 und aus der 1 eine 0.

Beispiel:

```
int a = 9;     // a = 00000000 00000000 00000000 00001001
int b = ~a;    // b = 11111111 11111111 11111111 11110110
               // b hat den Wert -10
```

7.6.6.2 Shift-Operatoren für Bits

Shift-Operatoren (**Verschiebeoperatoren**) können nur auf ganzzahlige Werte bzw. auf Objekte der entsprechenden Wrapper-Klassen angewandt werden. Mit dem Shift-Operator `<<` werden Bits nach links, mit dem Shift-Operator `>>` nach rechts unter Beachtung des Vorzeichens verschoben (engl. shift). Der Operator `>>>` wurde in Java eingeführt, um die Verschiebung nach rechts ohne Beachtung des Vorzeichens durchzuführen.

Der **linke Operand** eines Shift-Operators ist stets **der zu verschiebende Wert**. Der **rechte Operand** gibt die **Anzahl der Stellen an**, **um die verschoben werden soll**.

Obwohl Verschiebeoperatoren binär sind, wird auf ihre Operanden nicht die Typanpassung für binäre Operatoren (siehe Kapitel 7.7.3.4), sondern die Typanpassung für unäre Operatoren (siehe Kapitel 7.7.3.3) in impliziter Weise angewandt. Der Rückgabetyp eines Shift-Ausdrucks ist der angepasste Typ des linken Operanden.

Wenn der (implizit angepasste) Typ des linken Operanden der Typ `int` ist, so werden nur die **5 niederwertigsten Bits des rechten Operanden als Verschiebe-Distanz interpretiert**.

Dies wird in Bild 7-5 visualisiert.

Mit den 5 niederwertigsten Bits kann maximal die Zahl 32 dargestellt werden, denn 2^5 ergibt 32. Daher kann nur um 0 bis 31 Stellen verschoben werden. Wird als Verschiebung beispielsweise -1 angegeben, so wird tatsächlich um ($2^0 + 2^1 + 2^2 + 2^3 + 2^4$) = 31 Stellen verschoben. Dies bedeutet, dass alle Verschiebungen – auch bei Angabe negativer Zahlen – um ganzzahlige positive Stellen von Bits erfolgen. Das folgende Bild visualisiert die Verschiebedistanz:

Bild 7-5 Verschiebealgorithmus

Ist der angepasste Typ des linken Operanden der Typ `long`, so werden die **niedersten 6 Bits des rechten Operanden als Verschiebe-Distanz interpretiert**.

Mit anderen Worten, es kann zwischen 0 und 63 Stellen verschoben werden (2^4 ergibt 64). Die **Verschiebe-Operationen** werden **auf der Basis** der **Zweierkomplement-Darstellung des linken Operanden** durchgeführt.

Vorzeichenbehafteter Rechtsshift-Operator: `A >> B`

Mit dem Rechtsshift-Operator `A >> B` werden `B` Bitstellen von `A` nach rechts geschoben. Dabei gehen die `B` niederwertigen Bits von `A` verloren. Ist die Zahl `A` positiv, so werden von links Nullen nachgeschoben, ist `A` negativ, werden Einsen nachgeschoben.

Beispiel:

```
int a;
                                                    verloren
a = 8;          // 00000000 00000000 00000000 00001000
                   ------------------------------------->
a = a >> 3;     // 00000000 00000000 00000000 00000001
                   aufgefüllt
                                                    verloren
a = -7;         // 11111111 11111111 11111111 11111001
                   ------------------------------------->
a = a >> 3;     // 11111111 11111111 11111111 11111111
                   aufgefüllt
```

Bild 7-6 Verschiebung der Bits nach rechts mit dem >>-Operator

Für nicht negative Werte entspricht eine Verschiebung um 3 Bits nach rechts einer abschneidenden Ganzzahl-Division durch $2^3 = 8$.

Vorzeichenloser Rechtsshift-Operator: `A >>> B`

Mit dem Rechtsshift-Operator `A >>> B` werden `B` Bitstellen von `A` nach rechts geschoben. Dabei gehen die `B` niederwertigen Bits von `A` verloren. Es werden stets Nullen von links nachgeschoben, egal ob die Zahl negativ oder positiv ist.

Beispiel:

```
int a;
                                                    verloren
a = 8;          // 00000000 00000000 00000000 00001000
                   ------------------------------------->
a = a >>> 3;    // 00000000 00000000 00000000 00000001
                   aufgefüllt
                                                    verloren
a = -7;         // 11111111 11111111 11111111 11111001
                   ------------------------------------->
a = a >>> 3;    // 00011111 11111111 11111111 11111111
                   aufgefüllt
```

Bild 7-7 Verschiebung der Bits nach rechts mit dem >>>-Operator

Linksshift-Operator: `A << B`

Bei dem Linksshift-Operator `A << B` werden `B` Bitstellen von `A` nach links geschoben. Dabei gehen die `B` höherwertigen Bits von `A` verloren.

Durch das arithmetische Linksverschieben kann sich das Vorzeichen des Operanden `A` verändern, da das MSB durch den Wert an der Stelle MSB – `B` ersetzt wird.

Beispiel:

```
int a;
                        verloren
a = 8;          //  000 00000 00000000 00000000 00001000
                    ◄-----------------------------------
a = a << 3;     //  00000000 00000000 00000000 01000 000
                                                   aufgefüllt
```

Bild 7-8 Verschieben um drei Bits nach links

Die Verschiebung um 3 Bits nach links entspricht (auch bei einem Überlauf) einer Multiplikation mit 2^3. Wenn ein Überlauf entsteht, wird das vom Prozessor erkannt. Egal ob multipliziert oder verschoben wird, bei einem Überlauf ist das Ergebnis nicht mehr korrekt, aber bei beiden Operationen identisch.

7.6.7 Der Bedingungsoperator: `A ? B : C`

Eine echte "Rarität" ist der Bedingungsoperator. Er ist nämlich der einzige Operator, der drei Operanden verarbeitet. In einem **bedingten Ausdruck** `A ? B : C` wird zuerst der boolesche Ausdruck `A` ausgewertet. Der Ausdruck `A` kann seit Java 5 auch eine Referenz auf ein Objekt der Wrapper-Klasse `Boolean` sein.

Ist in `A ? B : C` der Rückgabewert von Ausdruck `A` gleich `true`, also wahr, so wird der Ausdruck `B` ausgewertet. Das Ergebnis von `B` ist dann der Rückgabewert des Bedingungsoperators. Ist jedoch der Ausdruck `A` gleich `false`, also "falsch", so wird der Ausdruck `C` ausgewertet.

Die Ausdrücke `B` und `C` müssen beide von einem numerischen Typ, beide vom Typ `boolean` oder beide jeweils entweder von einem Referenztyp oder vom `null`-Typ sein. Die Typen der Ergebnisausdrücke `B` bzw. `C` müssen zueinander zuweisungskompatibel (siehe Kapitel 7.7.2) sein.

Eine Methode kann (siehe Kapitel 9.2.3) mit der `return`-Anweisung einen Wert an den Aufrufer zurückliefern. Soll je nach dem Wahrheitswert von `A` der Wert von `B` bzw. `C` zurückgegeben werden, so kann statt

```
if (A)
   return B;
else
   return C;
```

knapper

```
return A ? B : C;
```

geschrieben werden.

Der Typ des bedingten Ausdrucks `A ? B : C` ist – unabhängig davon, ob der Rückgabewert dieses Ausdrucks `B` oder `C` ist – stets der breitere Typ (siehe Kapitel 7.7.3) der beiden Ausdrücke `B` und `C`.

So ist beispielsweise der Rückgabetyp von

```
(3 > 4) ? 5.0 : 6
```

vom Typ `double` und der Rückgabewert ist `6.0`.

Bedingte Ausdrücke enthalten Ausdrücke, die selbst wieder bedingt sein können.

Die Abarbeitungsreihenfolge ist von rechts her (Rechtsassoziativität).

So wird

```
A ? B : C ? D : E ? F : G
```

abgearbeitet wie

```
A ? B : (C ? D : (E ? F : G))
```

7.6.8 Prioritätentabelle der Operatoren

Die in Tabelle 7-9 gezeigte Vorrangtabelle enthält die Priorität (Rangfolge) und die Assoziativität der Operatoren:

Priorität	Operatoren	Bedeutung	Assoziativität
Priorität 1	`[]`	Array-Index	links
	`()`	Methodenaufruf	links
	`.`	Komponentenzugriff	links
	`++`	Postinkrement	links
	`--`	Postdekrement	links
Priorität 2	`++`	Präinkrement	rechts
	`--`	Prädekrement	rechts
	`+ -`	Vorzeichen (unär)	rechts
	`~`	bitweises Komplement	rechts
	`!`	logischer Negationsoperator	rechts
Priorität 3	`(type)`	Typ-Umwandlung	rechts
	`new`	Erzeugung	rechts
Priorität 4	`* / %`	Multiplikation, Division, Rest	links
Priorität 5	`+ -`	Addition, Subtraktion	links
	`+`	Stringverkettung	links
Priorität 6	`<<`	Linksshift	links
	`>>`	Vorzeichenbehafteter Rechtsshift	links
	`>>>`	Vorzeichenloser Rechtsshift	links
Priorität 7	`< <=`	Vergleich kleiner, kleiner gleich	links
	`> >=`	Vergleich größer, größer gleich	links
	`instanceof`	Typüberprüfung eines Objektes	links
Priorität 8	`==`	Gleichheit	links
	`!=`	Ungleichheit	links
Priorität 9	`&`	bitweises/logisches UND	links
Priorität 10	`^`	bitweises/logisches Exclusiv-ODER	links
Priorität 11	`|`	bitweises/logisches ODER	links
Priorität 12	`&&`	logisches UND	links
Priorität 13	`||`	logisches ODER	links
Priorität 14	`? :`	Bedingungsoperator	rechts
Priorität 15	`=`	Wertzuweisung	rechts
	`*= /= %= += -= <<= >>= >>>= &= ^= |=`	kombinierter Zuweisungsoperator	rechts

Tabelle 7-9 Priorität und Assoziativität der Operatoren von Java

Grau hinterlegt in Tabelle 7-9 sind die unären Operatoren.

Priorität 1 ist die höchste Priorität. So hat beispielsweise der Multiplikations- bzw. der Divisionsoperator eine höhere Priorität als der Additions- bzw. der Subtraktionsoperator. Durch gezielte Klammerungen `( )` lässt sich die Abarbeitungsreihenfolge verändern. Das wird in den nächsten Beispielen ersichtlich:

`5 * (3 + 4)`	das Ergebnis ist 35
`A && (B || C)`	dieser Ausdruck ist `true`, wenn die Bedingung `A` und `B` erfüllt ist, oder wenn `A` und `C` erfüllt ist.

Wie man der Tabelle 7-9 entnehmen kann, gilt die folgende Aussage bezüglich der Assoziativität:

Rechtsassoziativ sind Zuweisungsoperatoren, der Bedingungsoperator und unäre Operatoren. Alle anderen Operatoren sind linksassoziativ.

7.7 Konvertierung von Datentypen

In Java ist es nicht notwendig, dass die Operanden eines arithmetischen Ausdrucks vom selben Typ sind. Genauso wenig muss bei einer Zuweisung der Typ der Operanden übereinstimmen[123]. In solchen Fällen kann der Compiler selbsttätig **implizite** (**automatische**) **Typkonvertierungen** durchführen, die nach einem von der Sprache vorgeschriebenen Regelwerk ablaufen. Diese Regeln sollen in diesem Kapitel vorgestellt werden. Wenn man selbst dafür sorgt, dass solche Typverschiedenheiten nicht vorkommen, braucht man sich um die implizite Typkonvertierung nicht zu kümmern. Insbesondere kann man auch selbst mit Hilfe des cast-Operators **explizite Typkonvertierungen** durchführen.

7.7.1 Der cast-Operator

Eine **explizite Typumwandlung** eines beliebigen Ausdrucks kann man mit dem **cast-Operator** (**Typkonvertierungsoperator**) durchführen. Das englische Wort cast heißt unter anderem "in eine Form gießen". Durch `(Typname) Ausdruck` wird der Wert des Ausdrucks in den Typ gewandelt, der in den Klammern eingeschlossen ist. Der **Typkonvertierungsoperator** hat **einen** einzigen Operanden und ist damit ein **unärer Operator**.

Es kann nicht jeder Typ eines Operanden explizit in einen beliebigen anderen Typ gewandelt werden. Möglich sind Wandlungen

- zwischen numerischen Datentypen
- und zwischen Referenztypen.

Die explizite Typkonvertierung soll anhand eines **Beispiels** veranschaulicht werden:

```
int a = 1;          // a hat den Wert 1
double b = 3.5;     // b hat den Wert 3.5
a = (int) b;        // Explizite Typkonvertierung in den Typ int
```

Der Ausdruck `(int) b` hat den Rückgabewert `3` (die `0.5` wird abgeschnitten). Der Variablen `a` wird dann der Rückgabewert `3` zugewiesen.

[123] Auch bei der Übergabe von Werten an Methoden und bei Rückgabewerten von Methoden (siehe Kapitel 9.2.3) kann der Typ der übergebenen Ausdrücke bzw. des rückzugebenden Ausdrucks vom Typ der formalen Parameter bzw. vom Rückgabetyp verschieden sein.

Weitere **Beispiele** sind:

```
a = (int) 4.1  // a bekommt den Wert 4 zugewiesen.
a = (int) 4.9  // a bekommt ebenfalls den Wert 4 zugewiesen.
```

7.7.2 Implizite und explizite Typkonvertierungen

Eine implizite Typumwandlung hat dasselbe Resultat wie die entsprechende explizite Typumwandlung. Allerdings sind bei Zuweisungen, wenn auf der rechten Seite Variablen stehen, **nur implizite Typumwandlungen in** einen "**breiteren**" **Typ** möglich. **Mit** dem **cast-Operator** sind **auch Wandlungen** in einen "**schmäleren**" **Typ** möglich. Allerdings sind solche Wandlungen potentiell sehr gefährlich, da nicht nur die Genauigkeit, sondern auch das Vorzeichen und die Größe verloren gehen können (für Beispiele siehe Kapitel 7.7.4).

Typkonvertierungen erfolgen in Java prinzipiell **nur zwischen verträglichen Datentypen**. Zwischen nicht verträglichen Datentypen gibt es keine Umwandlungen. Hier muss der Compiler bzw. das Laufzeitsystem einen Fehler melden.

Kann ein Ausdruck in den Typ einer Variablen durch Zuweisung umgewandelt werden, so ist der Typ des Ausdrucks **zuweisungskompatibel** mit dem Typ der Variablen. Es findet eine implizite Typkonvertierung statt.

Implizite Typkonvertierungen gibt es:

- zwischen einfachen, numerischen (arithmetischen) Typen,
- zwischen Referenztypen,
- bei Verknüpfungen von Objekten der Klasse `String` mit Operanden anderer Datentypen
- und seit der Version Java 5 durch das automatische Boxing bzw. Unboxing zwischen einfachen numerischen Typen und Referenztypen numerischer Wrapper-Klassen.

Explizite Umwandlungen funktionieren wie implizite Umwandlungen, allerdings können mit expliziten Typumwandlungen auch Wandlungen durchgeführt werden, die implizit nicht zulässig sind.

Das folgende Kapitel 7.7.3 behandelt die Typkonvertierung von einfachen Datentypen. Typkonvertierungen bei der Verknüpfung von String-Objekten mit Operanden anderer Datentypen wurden bereits in Kapitel 6.8.2 behandelt. Das explizite und implizite Casten – d. h. die explizite und implizite Typumwandlung – bei **Referenzen** wird in Kapitel 11.3.1 **behandelt.**

7.7.3 Typkonvertierungen bei einfachen Datentypen

Zu den einfachen Datentypen gehören der Typ `boolean` und die numerischen Datentypen. Zwischen dem Typ `boolean` und den numerischen Datentypen kann weder explizit noch implizit gecastet werden. Somit kann eine Typkonvertierung von

einfachen Datentypen nur innerhalb der numerischen Datentypen erfolgen. Typumwandlungen in einen "breiteren" Typ bzw. mit anderen Worten "erweiternde Umwandlungen" sind in Bild 7-9 dargestellt:

Bild 7-9 Erweiternde Umwandlungen numerischer Datentypen

Bei erweiternden Umwandlungen ist der Wert immer darstellbar. Allerdings kann man an Genauigkeit verlieren, z. B. bei der Wandlung von `int` nach `float`, da die Gleitpunktzahlen nicht unendlich dicht aufeinander folgen. Typumwandlungen in einen "schmäleren" Typ bzw. mit anderen Worten "einschränkende Umwandlungen" sind in Bild 7-10 Einschränkende Umwandlungen numerischer Datentypendargestellt. Bei Wandlungen in einen "schmäleren" Typ kann es zu Informationsverlusten in der Größe, dem Vorzeichen und der Genauigkeit kommen. **Wandlungen in einen "schmäleren" Typ** sind in der Regel bei der impliziten Typkonvertierung nicht möglich und **müssen explizit mit dem cast-Operator durchgeführt werden**. Das folgende Bild zeigt solche Wandlungen in einen "schmäleren" Typ:

Bild 7-10 Einschränkende Umwandlungen numerischer Datentypen

7.7.3.1 Implizite Typkonvertierungen bei numerischen Datentypen

Welche Wandlung wann vorgenommen wird, hängt davon ab, ob es sich um

- eine Typkonvertierung von numerischen Operanden bei unären Operatoren (siehe Kapitel 7.7.3.3),
- eine Typkonvertierung von numerischen Operanden bei binären Operatoren (siehe Kapitel 7.7.3.4),
- bzw. eine Zuweisung (siehe Kapitel 7.7.3.5)

handelt.

Das Ergebnis einer bestimmten impliziten Typwandlung, die sowohl bei numerischen Operanden als auch bei Zuweisungen vorkommt, ist stets dasselbe. Bei numerischen Operanden gilt generell, dass der "kleinere" ("schmälere") Datentyp in den "größeren" ("breiteren") Datentyp umgewandelt wird. Bei Zuweisungen ist dies auch die Regel, es gibt jedoch einen Fall – siehe Kapitel 7.7.3.5 – wo vom "größeren" in den "kleineren" Datentyp gewandelt wird.

7.7.3.2 Die Integer-Erweiterung

Mit `byte`-, `short`- oder `char`-Werten werden in Java in der Regel keine Verknüpfungen zu Ausdrücken durchgeführt. Operanden dieser Typen werden oftmals vor der Verknüpfung mit einem Operator in den Datentyp `int` konvertiert.

Dies gilt für unäre und binäre Operatoren (siehe Kapitel 7.7.3.3 und Kapitel 7.7.3.4). Dieser Vorgang wird als **Integer-Erweiterung** (**integral promotion**) bezeichnet.

7.7.3.3 Anpassungen numerischer Typen bei unären Operatoren

Die **Integer-Erweiterung** eines einzelnen Operanden wird angewandt:

- auf den Dimensionsausdruck bei der Erzeugung von Arrays (siehe Kapitel 6.5),
- auf den Indexausdruck in Arrays (siehe Kapitel 6.5),
- auf Operanden der unären Operatoren `+` und `-`,
- auf den Operanden des Negationsoperators für Bits `~`,
- separat auf jeden Operanden der Schiebeoperatoren `>>`, `>>>` und `<<`.

7.7.3.4 Anpassungen numerischer Typen bei binären Operatoren

Bei binären Operatoren **mit Ausnahme von Zuweisungen, logischen Operatoren und Bitshift-Operatoren** werden implizite Typkonvertierungen von numerischen Typen durchgeführt mit dem Ziel, einen gemeinsamen numerischen Typ der Operanden des binären Operators zu erhalten, der auch der Typ des Ergebnisses ist. Diese Typkonvertierungen finden bei den folgenden binären Operatoren statt: `*`, `/`, `%`, `+`, `-`, `<`, `<=`, `>`, `>=`, `!=`, `==`, den bitweisen Operatoren `&`, `^` und `|`, sowie in gewissen Fällen (siehe [Gos97]) beim ternären Bedingungsoperator `?:`.

Wird **beispielsweise** eine Temperaturangabe von Grad Fahrenheit – hinterlegt in der Variablen `fahr` – nach Grad Celsius – abzuspeichern in der Variablen `celsius` vom Typ `double` – umgerechnet, wobei die Rechenvorschrift

```
celsius = (5.0 / 9) * (fahr - 32);
```

lautet, so werden bei der Berechnung der rechten Seite der Zuweisung automatisch die `int`-Konstante `9` und der Ausdruck `(fahr - 32)` in die `double`-Darstellung gewandelt, da `5.0` eine `double`-Zahl ist. Dieses Beispiel ist eine Anwendung der folgenden Regel:

Verknüpft ein binärer Operator einen ganzzahligen und einen Gleitpunktoperanden, so erfolgt eine Umwandlung des ganzzahligen Operanden in einen Gleitpunktwert. Anschließend wird eine Gleitpunktoperation durchgeführt.

Allgemeines Regelwerk

Bei binären Operatoren werden – bis auf die bereits genannten Ausnahmen – arithmetische Operanden in einen gemeinsamen Typ umgewandelt. D. h., im Ausdruck

```
Ausdruck1 Operator Ausdruck2
```

werden die Ausdrücke `Ausdruck1` und `Ausdruck2` auf den gleichen Typ gebracht. Von diesem Typ ist auch das Ergebnis. Die Umwandlung erfolgt in den höheren Typ der Hierarchie, wie in Bild 7-11 Wandlungen bei binären Operatoren symbolisiert:

Bild 7-11 Wandlungen bei binären Operatoren

Das allgemeine Regelwerk für diese Konvertierung lautet dabei:

1. Zunächst wird geprüft, ob einer der beiden Operanden vom Typ `double` ist. Ist einer von diesem Typ, dann wird der andere ebenfalls in `double` umgewandelt.
2. Ist dies nicht der Fall, so wird, wenn einer der beiden Operanden vom Typ `float` ist, der andere in den Typ `float` umgewandelt.
3. Ist dies nicht der Fall, so wird, wenn einer der beiden Operanden vom Typ `long` ist, der andere in den Typ `long` umgewandelt.
4. Ist dies nicht der Fall, so werden beide der **Integer-Erweiterung** unterworfen und in den Typ `int` umgewandelt.

Beispiel:

```
2 * 3L + 1.1
```

Die Multiplikation wird vor der Addition ausgeführt. Bevor die Multiplikation durchgeführt wird, wird die literale Konstante `2` in den Typ `long` gewandelt. Das Ergebnis der Multiplikation wird dann in den Typ `double` gewandelt und anschließend wird die Addition ausgeführt.

7.7.3.5 Implizite Typkonvertierung von numerischen Typen bei Zuweisungen, Rückgabewerten und Übergabeparametern von Methoden

Stimmt der Typ der Variablen links des Zuweisungsoperators `=` nicht mit dem Typ des Ausdrucks auf der rechten Seite des Zuweisungsoperators überein, so findet eine implizite Konvertierung statt, wenn die Typen links und rechts "verträglich" sind. Bei nicht verträglichen Typen wird eine Fehlermeldung generiert. Numerische Typen sind verträgliche Typen. **Zulässig** bei einer Zuweisung sind **erweiternde Umwandlungen** in einen "**breiteren**" **Typ**. Eine **implizite Umwandlung** in einen **schmäleren Typ** ist nur zulässig, wenn auf der rechten Seite der Zuweisung ein konstanter Ausdruck vom Typ `int` steht und auf der linken Seite eine Variable vom Typ `byte`, `short` oder `char` und wenn der Wert des Ausdrucks im Typ der Variablen darstellbar ist.

Bei der Zuweisung wird – wenn zulässig – der rechte Operand in den Typ des linken Operanden umgewandelt, d. h., der Resultattyp einer Zuweisung ist der Resultattyp des linken Operanden, und der Wert ist der, der sich nach der Zuweisung im linken Operanden befindet.

Bei Rückgabewerten von Methoden wird der Ausdruck, der mit der Sprunganweisung `return` zurückgegeben wird – wie bei einer Zuweisung – in den Rückgabetyp der Methode umgewandelt.

Dies gilt auch für einen konstanten Ausdruck vom Typ `int` als Rückgabewert, der passend in den Typ `byte`, `short` oder `char` gewandelt wird (sofern der konstante Ausdruck vom Typ `int` im jeweiligen Typ dargestellt werden kann). Im Falle von Übergabeparametern bei Methodenaufrufen ist das jedoch nicht zugelassen.

Verlangt eine Methode einen Parameter vom Typ `byte`, `short` oder `char`, so darf kein konstanter Ausdruck vom Typ `int` übergeben werden. Es ist in diesen Fällen immer eine explizite Typkonvertierung erforderlich.

7.7.4 Konvertierungsvorschriften für einfache Datentypen

Im Folgenden werden die Wandlungsvorschriften zwischen verschiedenen Typen behandelt.

7.7.4.1 Umwandlungen eines vorzeichenbehafteten Integer-Typen in den breiteren Typ

Wird ein Integer-Wert in einen größeren Integer-Typ mit Vorzeichen[124] gewandelt, so bleibt sein Wert unverändert. Es wird dabei links mit Nullen aufgefüllt und das Vorzeichenbit wird passend gesetzt.

[124] Die Integer-Typen `byte`, `short`, `int` und `long` haben ein Vorzeichen, der Typ `char` nicht.

7.7.4.2 Umwandlungen eines vorzeichenbehafteten Integer-Typen in den Typ `char`

Wird ein Integer-Wert vom Typ `short` in den Typ `char` gewandelt, so bleibt das Bitmuster erhalten, jedoch nicht die Bedeutung des Bitmusters. Dies bedeutet, dass eine negative Zahl als positive Zahl interpretiert wird. Ein korrektes Resultat ist deshalb nur für positive Zahlen möglich. Dies zeigt das folgende **Beispiel:**

```
// Datei: Short2Char.java

public class Short2Char
{
   public static void main (String[] args)
   {
      char posChar, negChar;
      short posShort = 1;
      short negShort = -1;
      posChar = (char) posShort; // explizites Casten
      negChar = (char) negShort; // explizites Casten

      // Bei der Ausgabe muss vom Typ char nach int konvertiert wer-
      // den, da sonst ein entsprechendes Zeichen angezeigt wird.
      System.out.println ("positiver Short: " + posShort
             + " hat als Char den Dezimalwert " + (int) posChar);
      System.out.println ("negativer Short: " + negShort
             + " hat als Char den Dezimalwert " + (int) negChar);
   }
}
```

Die Ausgabe des Programms ist:

```
positiver Short: 1 hat als Char den Dezimalwert 1
negativer Short: -1 hat als Char den Dezimalwert 65535
```

Wird ein Integer-Wert vom Typ `byte` in den Typ `char` gewandelt, so wird von links mit Null-Bits aufgefüllt und das Vorzeichen propagiert. Da sich die Interpretation ändert, bleibt der Wert einer negativen Zahl nicht erhalten, jedoch der Wert einer positiven Zahl. Dies ist im folgenden **Programmbeispiel** zu sehen:

```
// Datei: Byte2Char.java
public class Byte2Char
{
   public static void main (String[] args)
   {
      char posChar, negChar;
      byte posByte = 3;
      byte negByte = -1;
      posChar = (char) posByte;
      negChar = (char) negByte;

      // Bei der Ausgabe muss vom Typ char nach int konvertiert wer-
      // den, da sonst ein entsprechendes Zeichen angezeigt wird.
      System.out.println ("positives Byte: " + posByte
                          + " hat als char den Dezimalwert "
                          + (int) posChar);
```

```
      System.out.println ("negatives Byte: " + negByte
                          + " hat als char den Dezimalwert "
                          + (int) negChar);
   }
}
```

Die Ausgabe des Programms ist:

```
positives Byte: 3 hat als char den Dezimalwert 3
negatives Byte: -1 hat als char den Dezimalwert 65535
```

Wird ein Integer-Wert vom Typ `int` oder `long` in den Typ `char` gewandelt, so ist ein korrektes Resultat für große Zahlen nicht gegeben, was in folgendem **Programmbeispiel** demonstriert wird:

```
// Datei: Int2Char.java

public class Int2Char
{
   public static void main (String[] args)
   {
      int wert1 = 65535;
      int wert2 = 65536;
      char wert1Char = (char) wert1;
      char wert2Char = (char) wert2;

      // Bei der Ausgabe muss vom Typ char nach int konvertiert wer-
      // den, da sonst ein entsprechendes Zeichen angezeigt wird.
      System.out.println (wert1 + " hat als char den Dezimalwert "
                          + (int) wert1Char);
      System.out.println (wert2 + " hat als char den Dezimalwert "
                          + (int) wert2Char);
   }
}
```

Die Ausgabe des Programms ist:

```
65535 hat als char den Dezimalwert 65535
65536 hat als char den Dezimalwert 0
```

7.7.4.3 Umwandlungen zwischen Integer- und Gleitpunkt-Typen

Die folgenden Umwandlungen werden betrachtet:

- **Integer nach Gleitpunkt**

  Wenn ein Wert aus einem Integer-Typ in einen Gleitpunkttyp umgewandelt wird, so werden als Nachkommastellen Nullen eingesetzt. In der Realität kann eine solche Zahl jedoch nicht exakt dargestellt werden. Das Resultat ist dann entweder der nächsthöhere oder der nächstniedrigere darstellbare Wert.

- **Gleitpunkt nach Integer**

  Bei der Wandlung einer Gleitpunktzahl in eine Integerzahl werden die Stellen hinter dem Komma abgeschnitten. Bei zu großen Zahlen ist ein korrektes Ergebnis nicht möglich, wie folgendes Beispiel zeigt:

```
// Datei: Double2Int.java

public class Double2Int
{
   public static void main (String[] args)
   {
      double d = 2147483642d;
      int i;

      for (int count = 0; count < 10; count++)
      {
         i = (int) d;
         System.out.println("Double " + d + " ist als int " + i);
         d++;
      }
   }
}
```

Die Ausgabe des Programms ist:

```
Double 2.147483642E9 ist als int 2147483642
Double 2.147483643E9 ist als int 2147483643
Double 2.147483644E9 ist als int 2147483644
Double 2.147483645E9 ist als int 2147483645
Double 2.147483646E9 ist als int 2147483646
Double 2.147483647E9 ist als int 2147483647
Double 2.147483648E9 ist als int 2147483647
Double 2.147483649E9 ist als int 2147483647
Double 2.14748365E9 ist als int 2147483647
Double 2.147483651E9 ist als int 2147483647
```

7.7.4.4 Umwandlungen zwischen Gleitpunkttypen

Wenn ein Gleitpunktwert mit niedriger Genauigkeit in einen Gleitpunkttyp mit einer höheren Genauigkeit umgewandelt wird, so gibt es keine Probleme. Die Größe bleibt selbstverständlich unverändert. Wenn ein Gleitpunktwert mit höherer Genauigkeit in einen Gleitpunkttyp mit einer niedrigeren Genauigkeit umgewandelt wird, so kann – wenn der Wert im zulässigen Wertebereich liegt – der neue Wert wegen der unterschiedlichen Genauigkeit der beteiligten Typen der nächsthöhere oder der nächstniedrigere darstellbare Wert sein. Liegt der Wert nicht im zulässigen Wertebereich, so ist ein korrektes Ergebnis nicht möglich. Dies ist in folgendem **Beispiel** zu sehen:

```
// Datei: Double2Float.java
public class Double2Float
{
   public static void main (String[] args)
   {
      double smallDouble = 9.999999999d;
      double bigDouble =   1.23E145;
```

```
        float smallFloat = (float) smallDouble;
        float bigFloat = (float) bigDouble;

        System.out.println ("kleiner Double-Wert: " + smallDouble
                           +" wird zu " + smallFloat);
        System.out.println ("grosser Double-Wert: " + bigDouble
                           +" wird zu " + bigFloat);
    }
}
```

Die Ausgabe des Programms ist:

```
kleiner Double-Wert: 9.999999999 wird zu 10.0
grosser Double-Wert: 1.23E145 wird zu Infinity
```

7.8 Zusammenfassung

Alles das, was einen Wert zurückliefert, stellt einen Ausdruck dar. Verknüpft man Operanden – ein Operand ist selbst ein Ausdruck – durch Operatoren und gegebenenfalls auch runde Klammern, so entstehen komplexe Ausdrücke. Runde Klammern beeinflussen dabei die Auswertungsreihenfolge der Operatoren. Das Ziel dieser Verknüpfungen ist die Berechnung neuer Werte oder auch das Erzeugen von gewollten Nebeneffekten.

Kapitel **7.1** stellt vor, dass Java

- einstellige (unäre, monadische),
- zweistellige (binäre, dyadische)
- und einen einzigen dreistelligen (ternären, triadischen), nämlich den Bedingungsoperator `? :`

aufweist.

Unäre Operatoren haben einen einstelligen Operanden. Der Operator kann vor (prä) oder hinter (post) seinem Operator stehen. Ein binärer Operator benötigt zwei Operanden und der einzige dreistellige Operator verarbeitet drei Operanden.

Kapitel **7.2** untersucht Ausdrücke und Anweisungen. Ausdrücke haben in Java stets einen Rückgabewert eines bestimmten Typs.

Man unterscheidet die verschiedenen Typen von Anweisungen in Java:

- Selektionsanweisungen,
- Iterationsanweisungen,
- Sprunganweisungen,
- die leere Anweisung,
- die `try`-Anweisung,
- die `throw`-Anweisung,
- die `assert`-Anweisung,
- die `synchronized`-Anweisung
- und Ausdrucksanweisungen.

In Java kann man bei bestimmten Arten von Ausdrücken durch das Anhängen eines Semikolons an den vorliegenden Ausdruck erreichen, dass dieser Ausdruck zu einer Anweisung wird. Man spricht dann von einer sogenannten Ausdrucksanweisung. In einer solchen Ausdrucksanweisung wird der Rückgabewert eines Ausdrucks nicht verwendet. Lediglich wenn Nebeneffekte zum Tragen kommen, ist eine Ausdrucksanweisung sinnvoll und zulässig.

Die folgenden Ausdrücke können in Java zu einer Anweisung werden:

- Zuweisungen, d. h. `=` und kombinierte Zuweisungsoperatoren wie z. B. `+=`, da (kombinierte) Zuweisungungsoperatoren einen Nebeneffekt haben.
- Postfix- und Präfix-Inkrement- und -Dekrementoperator, d. h. `++` und `--` angewandt auf eine Variable, da diese Operatoren einen Nebeneffekt haben.
- Methodenaufrufe, unbenommen davon, ob sie einen Rückgabewert haben oder nicht, da Methodenaufrufe Nebeneffekte haben können.
- Ausdrücke, die mit `new` ein Objekt erzeugen. Begründung: Der Rückgabewert des `new`-Operators ist die Referenz auf das erzeugte Objekt. Der Nebeneffekt ist, dass ein Objekt im Heap existiert.

Nebeneffekte und ihr Ausführungszeitpunkt werden in Kapitel **7.3** besprochen. Bei Nebeneffekten werden Inhalte des Speichers verändert, meist der Wert einer statischen Variablen.

In Java gibt es verschiedene Sorten von Nebeneffekten:

- Nebeneffekte bei Zuweisungen,
- Nebeneffekte von Operatoren,
- Nebeneffekte bei Methoden, die nicht nur lesend, sondern auch schreibend auf Variablen zugreifen,
- und Nebeneffekte bei Ausdrücken, die mit `new` ein Objekt erzeugen.

In Java wird jeder Operand eines Operators vollständig ausgewertet, bevor irgendein Teil der Operation begonnen wird. Damit haben (mit Ausnahme der Operatoren `&&`, `||` und `? :`) vor einer Operation die Nebeneffekte der Operanden stattgefunden.

In Java werden die Operanden eines Operators strikt von links nach rechts ausgewertet. Dies bedeutet, dass der Nebeneffekt des linken Operanden vor der Bewertung des rechten Operanden erfolgt ist.

In Java werden die aktuellen Parameter eines Methodenaufrufs von links nach rechts bewertet. Dies bedeutet, dass nach der Bewertung eines aktuellen Parameters ein Nebeneffekt dieses aktuellen Parameters stattgefunden hat und erst dann der rechts davon stehende aktuelle Parameter bewertet wird.

Die Auswertungsreihenfolge von Operatoren verschiedener und gleicher Priorität wird in Kapitel **7.4** analysiert.

1. Wie in der Mathematik werden als erstes Teilausdrücke in Klammern ausgewertet. Der Wert und Typ eines Ausdrucks ändert sich nicht, wenn er in Klammern gesetzt wird. So sind beispielsweise die beiden Zuweisungen `a = b` und `a = (b)` identisch.

2. Anschließend werden Ausdrücke mit unären Operatoren ausgewertet. Unäre Operatoren werden von rechts nach links angewendet.
3. Abschließend werden Teilausdrücke mit mehrstelligen Operatoren ausgewertet.

Unäre Operatoren haben alle dieselbe Priorität. Die Abarbeitung mehrstelliger Operatoren erfolgt nach der Prioritätstabelle der Operatoren, wenn Operatoren verschiedener Prioritäten nebeneinander stehen.

Bei Operatoren verschiedener Priorität erfolgt zuerst die Abarbeitung der Operatoren mit höherer Priorität. Bei gleicher Priorität entscheidet die Assoziativität der Operatoren, ob die Verknüpfung von links nach rechts oder von rechts nach links erfolgt. Durch das Setzen von Klammern kann man von der festgelegten Reihenfolge abweichen.

Unter Assoziativität versteht man die Reihenfolge, wie Operatoren und Operanden verknüpft werden, wenn Operanden durch Operatoren derselben Priorität (Vorrangstufe) verknüpft werden. Ist ein Operator rechtsassoziativ, so wird eine Verkettung von Operatoren und Operanden von rechts nach links abgearbeitet, bei Linksassoziativität dementsprechend von links nach rechts.

In Java werden die Operanden eines Operators strikt von links nach rechts ausgewertet.

Kapitel **7.5** diskutiert den Unterschied zwischen L- und R-Werten. Ein L-Wert zeichnet sich dadurch aus, dass er einen Speicherplatz irgendwo im Arbeitsspeicher besitzt. Ist ein Ausdruck kein L-Wert, so ist er ein R-Wert (rvalue oder right value) und kann nicht links, sondern nur rechts vom Zuweisungsoperator stehen. Einem R-Wert kann man keinen Wert zuweisen, da er keine feste Speicherstelle besitzt. Des Weiteren wird zwischen modifizierbarem und nicht modifizierbarem L-Wert unterschieden.

Kapitel **7.6** stellt alle unären, binären und tertiären Operatoren zusammen und gibt die Prioritätentabelle der Operatoren wieder.

Kapitel **7.7** erläutert die Konvertierung von Datentypen mit Hilfe des cast-Operators und betrachtet die Konvertierung einfacher Datentypen.

7.9 Übungen

Aufgabe 7.1: Operatoren, Ausdrücke und Anweisungen

7.1.1 Verständnisfragen

a) Welche Arten von Operatoren gibt es in Java:

- nach Anzahl der Operatoren?
- nach Art der Wirkungsweise?

b) Was sind Ausdrücke?
c) Was sind Anweisungen?
d) Was sind Nebeneffekte?

7.1.2 Zuordnen von Operatoren

Ordnen Sie die folgenden Operatoren den Operator-Arten zu, nach denen Sie in Aufgabe 7.1.1 a) gefragt wurden:

a) `i++`
b) `if(x && y) . . .`
c) `a?b:c`

Aufgabe 7.2: Verwendung von Operatoren

7.2.1 Zahlen vergleichen mit Vergleichsoperatoren

Gegeben ist eine Klasse `ZahlenVergleich`, welche die Methode `eingabeZahl()` enthält. Diese Methode ermöglicht es Ihnen, eine `int`-Zahl von der Tastatur einzulesen:

```
// Datei: ZahlenVergleich.java

public class ZahlenVergleich
{
      public int eingabeZahl()
      {
         try
         {
            java.util.Scanner scanner =
               new java.util.Scanner (System.in);
            System.out.print ("Gib einen Wert ein: ");
            return scanner.nextInt();
         }
         catch (Exception e)
         {
            System.out.println (e);
         }
         return -1;
      }
      . . . . .
}
```

Ergänzen Sie die fehlende Stelle im Programm (. . . . .), sodass mit Hilfe der Methode `eingabeZahl()` zwei Zahlen von der Tastatur eingelesen werden. Anschließend vergleichen Sie die Zahlen miteinander auf Gleichheit (==). Sind die Zahlen gleich, so wird folgender Text ausgegeben:

```
Die Zahlen sind gleich!
```

Bei Ungleichheit der Zahlen ermitteln Sie die größere Zahl. Eine mögliche Ausgabe würde folgendermaßen aussehen:

```
Die Zahl 5 ist größer als die Zahl 2!
```

7.2.2 Bedingungen auswerten mit dem Bedingungsoperator

Gegeben seien folgende Codezeilen:

```
int x = 5;
int y = 7;
int i = (x == y) ? 1 : 0;
```

a) Welchen Wert hat `i`?
b) Wie würden obige Codezeilen mit Hilfe einer `if-else`-Abfrage aussehen?

7.2.3 Modulo-Operator

Vervollständigen Sie das folgende Code-Fragment, um die Funktion des Modulo-Operators (Restwert-Operator) nachzubilden:

```
// Datei: ZuweisungsOperator.java

public class ZuweisungsOperator
{
   public static void main (String[] args)
   {
      int a = 39;
      int b = 5;

      while (a . . . . . b) a . . . . . b;

      System.out.println ("39 Modulo 5 ist: " + a);
   }
}
```

Aufgabe 7.3: Auswertungsreihenfolge

7.3.1 Erläutern Sie folgende Begriffe:

a) Operatorpriorität
b) Operatorassoziativität
c) Bewertungsreihenfolge

7.3.2 Operatorpriorität und -assoziativität

Vor jeder Anweisung seien folgende Werte gegeben:

```
int a = 2;
int b = 1;
```

Finden Sie ohne Java-Compiler heraus, welchen Wert die Variablen `a` und `b` nach den einzelnen Anweisungen a) bis m) haben. Beachten Sie hierbei genau die Priorität und Assoziativität der entsprechenden Operatoren. Erläutern Sie, wie Sie auf das Ergebnis kommen. Verifizieren Sie Ihr theoretisch ermitteltes Ergebnis gegebenenfalls durch einen Programmlauf:

a) `a = b = 2;`
b) `a = b * 3 + 2;`
c) `a = b * (3 + 2);`
d) `a *= b + 5;`
e) `b %= 2 * a;`
f) `a = --b;`
g) `b = ~a;`

h) `b = b++ * a`
i) `a = - 5 - 5;`
j) `b = b << 2;`
k) `b = (a == b) ? 5 : 7;`
l) `a = --b * b++;`
m) `a = a ^ b;`

Aufgabe 7.4: L-Werte und R-Werte

7.4.1 Was versteht man unter L-Werten und R-Werten?
7.4.2 Ist das folgende Code-Fragment korrekt?

```
. . . . .
int x = 5;
final int y = 2;
y = x;
. . . . .
```

Aufgabe 7.5: Konvertierung von Datentypen

7.5.1 Implizite Konvertierung bei numerischen Datentypen

Welche Ausgabe erhalten Sie von folgendem Programm? Begründen Sie Ihre Antwort! Hier das Programm:

```
// Datei: ImpliziteKonvertierung.java
public class ImpliziteKonvertierung
{
   public static void main (String[] args)
   {
      System.out.println ("Division von 10 durch 12: " + (10/12));
      System.out.println ("Division von 10. durch 12: "+ (10./12));
      System.out.println ("Division von 10 durch 12.: " + (10/12.));
   }
}
```

7.5.2 Explizite Konvertierung bei numerischen Datentypen

Welche Ausgabe erhalten Sie von folgendem Programm? Begründen Sie Ihre Antwort! Hier das Programm:

```
// Datei: ExpliziteKonvertierung.java
public class ExpliziteKonvertierung
{
   public static void main (String[] args)
   {
      int k;
      float f = 1.5f, g;
      k = 10 * (int) f;
      g = 10 * f;

      System.out.println ("Multiplikation (int)1.5f * 10: "
                          + k);
      System.out.println ("Multiplikation 1.5f * 10: " + g);
   }
}
```

Kapitel 8

Kontrollstrukturen

8.1 Blöcke – Kontrollstrukturen für die Sequenz
8.2 Selektion
8.3 Iteration
8.4 Sprunganweisungen
8.5 Zusammenfassung
8.6 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_8

8 Kontrollstrukturen

Kontrollstrukturen steuern den Kontrollfluss eines sequenziellen Programms. Dabei können beispielsweise in Abhängigkeit von der Bewertung von Ausdrücken gewisse Anweisungen übergangen oder ausgeführt werden. Da Kontrollstrukturen in Java einen einzigen Eingang und einen einzigen Ausgang haben, bleibt der Kontrollfluss einer Methode dennoch sequenziell.

Kapitel 8.1 beschäftigt sich mit Kontrollstrukturen für die Sequenz, den sogenannten Blöcken. Kapitel 8.2 geht auf die Möglichkeit der Selektion und Kapitel 8.3 auf die Iteration von Anweisungen ein. Sprunganweisungen werden in Kapitel 8.4 vorgestellt.

8.1 Blöcke – Kontrollstrukturen für die Sequenz

Erfordert die Syntax genau eine einzige Anweisung, so können dennoch mehrere Anweisungen geschrieben werden, wenn man sie in Form eines Blockes[125] zusammenfasst:

```
{
   Anweisung_1
   Anweisung_2
       .
       .
   Anweisung_n
}
```

Die geschweiften Klammern `{` und `}` stellen die Blockbegrenzer dar. Die Anweisungen zwischen den Blockbegrenzern werden sequenziell abgearbeitet. Ein **Block** wird deshalb auch als **Kontrollstruktur für die Sequenz** bezeichnet. Bild 8-1 zeigt mehrere Anweisungen, die zu einem Block gruppiert sind:

Bild 8-1 Ein Block ist eine Sequenz von Anweisungen

Ein **Block** (eine **zusammengesetzte Anweisung**) kann an jeder Stelle stehen, an der eine einzelne Anweisung angeschrieben werden kann.

[125] Blöcke werden in Kapitel 9 behandelt.

8.2 Selektion

Eine Selektion ermöglicht die Abarbeitung von Anweisungen in Abhängigkeit von einer Bedingung. In Java gibt es die bedingte Anweisung, die einfache Alternative mit `if` und `else` und die mehrfache Alternative in den Ausprägungen `else-if` und `switch`.

8.2.1 Bedingte Anweisung und einfache Alternative

Die Syntax der **einfachen Alternative** ist:

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

Dies wird visualisiert durch das folgende Struktogramm:

Bild 8-2 Struktogramm der einfachen Alternative (`if-else`-Anweisung)

Der Ausdruck `Ausdruck` in Klammern wird berechnet und ausgewertet. Trifft diese Bedingung zu (hat also der Ausdruck `Ausdruck` den Wert `true`), so wird die Anweisung `Anweisung1` ausgeführt. Trifft die Bedingung nicht zu (hat also der Ausdruck `Ausdruck` den Wert `false`), so wird die Anweisung `Anweisung2` ausgeführt, falls ein `else`-Zweig vorhanden ist. Soll mehr als eine einzige Anweisung ausgeführt werden, so ist ein **Block** zu verwenden, der **syntaktisch** als **eine einzige Anweisung** zählt. Der `else`-Zweig ist optional. Entfällt der `else`-Zweig, so spricht man von einer bedingten Anweisung. Diese ist im folgenden Bild dargestellt:

Bild 8-3 Struktogramm der bedingten Anweisung mit `if`

Die Syntax der **bedingten Anweisung** ist:

```
if (Ausdruck) Anweisung1
```

Fällt einem jetzt plötzlich ein, dass man eigentlich zwei Anweisungen ausführen sollte, wenn die Bedingung zutrifft, so darf man nicht die zweite Anweisung `Anweisung2` einfach hinter der Anweisung `Anweisung1` notieren. Schreibt man

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

wird nämlich die Anweisung `Anweisung2` stets ausgeführt, auch wenn die Bedingung `Ausdruck` nicht zutrifft. Hier ist ein Block zu verwenden:

```
if (Ausdruck)
{
   Anweisung1
   Anweisung2
}
```

Für eine defensive Programmierung sollten stets geschweifte Klammern verwendet werden, damit der Handlungsablauf leicht um weitere Anweisungen ergänzt werden kann:

```
if (Ausdruck)
{
   Anweisung1
}
```

Auf die defensive Programmierung wird in Kapitel 7.6.4 eingegangen.

Geschachtelte `if-else`-Anweisungen

Da der `else`-Zweig einer `if-else`-Anweisung optional ist, entsteht eine Mehrdeutigkeit, wenn ein `else`-Zweig in einer verschachtelten Folge von `if-else`-Anweisungen fehlt. Der letzte `else`-Zweig wird deshalb immer mit dem letzten `if` verbunden, für das noch kein `else`-Zweig existiert, damit die Eindeutigkeit hergestellt wird. So gehört im folgenden **Beispiel**

```
if (n > 0)
   if (a > b)
      z = a;
   else
      z = b;
```

der `else`-Zweig – wie die Regel oben aussagt – zum letzten, inneren `if`. Eine von Programmierern eventuell versuchte Umgehung der Zuordnung der `if`- und `else`-Zweige durch Einrücken (z. B. mit dem Tabulator) kann der Compiler nicht erkennen, da für ihn Whitespaces nur die Bedeutung von Trennern haben, aber sonst vollkommen bedeutungslos sind. Um eine andere Zuordnung zu erreichen, müssen entsprechende geschweifte Klammern gesetzt und somit Blöcke definiert werden, wie im Folgenden gezeigt wird:

```
if (n > 0)
{
   if (a > b)
      z = a;
}
else
   z = b;
```

8.2.2 Mehrfache Alternative – `else-if`

Die `else-if`-Anweisung ist die allgemeinste Möglichkeit für eine **Mehrfach-Selektion**, d. h., um eine Auswahl unter verschiedenen Alternativen zu treffen. Die Syntax dieser Anweisung ist:

```
if (Ausdruck_1)
   Anweisung_1
else if (Ausdruck_2)
   Anweisung_2
      .
      .
      .
else if (Ausdruck_n)
   Anweisung_n
else                      // der else-Zweig
   Anweisung_else         // ist optional
```

In der angegebenen Reihenfolge wird ein Vergleich nach dem anderen durchgeführt. Bei der ersten Bedingung, die `true` ist, wird die zugehörige Anweisung abgearbeitet und die Mehrfach-Selektion abgebrochen. Dabei kann statt einer einzelnen Anweisung stets auch ein Block von Anweisungen stehen, da ein Block syntaktisch einer einzigen Anweisung gleichgestellt ist. Der letzte `else`-Zweig ist optional. Hier können alle anderen Fälle behandelt werden, die nicht explizit aufgeführt wurden. Ist dies nicht notwendig, so kann der `else`-Zweig entfallen. Dieser `else`-Zweig wird oft zum Abfangen von Fehlern, z. B. bei einer Benutzereingabe, verwendet. Betätigt der Benutzer eine ungültige Taste, kann er etwa in diesem `else`-Teil "höflichst" auf sein Versehen hingewiesen werden.

Das folgende Bild zeigt ein Beispiel für ein Struktogramm:

Bild 8-4 Beispiel für ein Struktogramm der `else-if`-Anweisung

8.2.3 Mehrfache Alternative – `switch`

Für eine **Mehrfach-Selektion**, d. h. eine Selektion unter mehreren Alternativen, kann auch die `switch`-Anweisung verwendet werden. Der Ausdruck `Ausdruck` in der `switch`-Anweisung muss vom Typ `char`, `byte`, `short`, `int`, `Character`, `Byte`, `Short`, `Integer`, einem Aufzählungstyp oder – seit Java 7 – vom Typ `String` sein.

Ferner muss jeder konstante Ausdruck `konstanter_Ausdruck_n` dem Typ von `Ausdruck` zuweisbar sein[126]. Die **Syntax** der **`switch`-Anweisung** lautet:

```
switch (Ausdruck)
{
   case konstanter_Ausdruck_1:
      Anweisungen_1
      break;                          // ist optional
   case konstanter_Ausdruck_2:
      Anweisungen_2
      break;                          // ist optional
     .
     .
     .
   case konstanter_Ausdruck_n:
      Anweisungen_n
      break;                          // ist optional
   default:                           // ist
      Anweisungen_default             // optional
}
```

Das Struktogramm der `switch`-Anweisung sieht wie folgt aus:

Bild 8-5 Prinzipielles Struktogramm einer `switch`-Anweisung

Jeder Alternative geht eine einzige oder eine Reihe von **`case`-Marken** mit ganzzahligen Konstanten oder konstanten Ausdrücken voraus. Ein Beispiel für eine einzige `case`-Marke ist:

```
case 5:
```

Ein Beispiel für eine Reihe von `case`-Marken ist:

```
case 1:  case 3:  case 5:
```

Eine Konstante kann eine literale Konstante oder eine symbolische Konstante sein. Eine **symbolische Konstante** wird meist mit **`final`** deklariert, kann aber auch als **Aufzählungskonstante** festgelegt werden. Eine `final`-Konstante wird in dem folgenden **Beispielprogramm** demonstriert:

[126] Siehe Kapitel 7.7.3.5. Ist beispielsweise `Ausdruck` vom Typ `byte`, so kann `konstanter_Ausdruck_1` z. B. nicht den Wert 1000 annehmen.

```
// Datei: SwitchTest.java

public class SwitchTest
{
   private static final int EINS = 1; //symbolische Konstante mit
                                      //dem Namen EINS
   public void testSwitch (int zahl)
   {
      switch (zahl)
      {
         case EINS:
         {
            System.out.println ("Testergebnis: " + EINS);
            break;
         }
         case 2:
         {
            System.out.println ("Testergebnis: " + 2);
            break;
         }
      }
   }

   public static void main (String[] args)
   {
      SwitchTest test = new SwitchTest();
      test.testSwitch (1);
      test.testSwitch (2);
      test.testSwitch (EINS);
   }
}
```

Die Ausgabe des Programms ist:

```
Testergebnis: 1
Testergebnis: 2
Testergebnis: 1
```

Ist der Wert des Ausdrucks einer `switch`-Anweisung identisch mit dem Wert eines der konstanten Ausdrücke der **`case`-Marken**, wird die Ausführung des Programmes an dieser Stelle weitergeführt. Stimmt keiner der konstanten Ausdrücke im Wert mit dem `switch`-Ausdruck überein, wird zu `default` gesprungen. **`default`** ist optional. Benötigt eine Anwendung keinen `default`-Fall, kann dieser entfallen und das Programm wird beim Nichtzutreffen aller aufgeführten konstanten Ausdrücke nach der `switch`-Anweisung fortgeführt. Die Reihenfolge der `case`-Marken ist beliebig. Auch die `default`-Marke muss nicht als letzte stehen. Am übersichtlichsten ist es allerdings, wenn die `case`-Marken nach aufsteigenden Werten geordnet sind und `default` am Schluss steht.

Soll eine `switch`-Anweisung auf Aufzählungskonstanten angewendet werden, so muss der Ausdruck in der `switch`-Anweisung dem Aufzählungstyp entsprechen. Die konstanten Ausdrücke der `case`-Marken müssen die Aufzählungskonstanten des verwendeten Typs sein, da sie dem Ausdruck zuweisbar sein müssen.

Hierfür ein **Programmbeispiel**:

```
// Datei: Richtungsweiser.java
public class Richtungsweiser
{
   public enum Richtung {LINKS, RECHTS}

   public static void main (String[] args)
   {
      Richtung ref = Richtung.RECHTS;
      switch (ref)
      {
         case LINKS:
            System.out.println ("LINKS");
            break;
         case RECHTS:
            System.out.println ("RECHTS");
            break;
      }
   }
}
```

Die Ausgabe des Programms ist:

```
RECHTS
```

Beachten Sie, dass Aufzählungskonstanten nicht qualifiziert sein dürfen, wenn sie als `case`-Marken verwendet werden.

Im Falle der Marken geht der Typ der Aufzählungskonstanten aus dem Ausdruck der `switch`-Anweisung hervor. Eine wichtige Bedingung für die `switch`-Anweisung ist, dass **alle `case`-Marken unterschiedlich** sein müssen. Vor einer einzelnen Befehlsfolge können – wie bereits erwähnt – mehrere verschiedene `case`-Marken stehen. Dies demonstriert das folgende **Beispiel**:

```
// Datei: ZeichenTester.java
public class ZeichenTester
{
   public void testeZeichen (char c)
   {
      switch (c)
      {
         case '\t':
         case '\n':
         case '\r':
            System.out.println ("Steuerzeichen");
            break;
         default:
            System.out.println ("Kein Steuerzeichen: " + c);
      }
   }
```

```
   public static void main (String[] args)
   {
      ZeichenTester pars = new ZeichenTester();
      pars.testeZeichen ('\t');
      pars.testeZeichen ('A');
      pars.testeZeichen ('\r');
   }
}
```

Die Ausgabe des Programms ist:

```
Steuerzeichen
Kein Steuerzeichen: A
Steuerzeichen
```

Wird in der `switch`-Anweisung eine passende `case`-Marke gefunden, werden hinter dieser Marke die anschließenden Anweisungen bis zu der Sprunganweisung `break` ausgeführt. Die Sprunganweisung `break` bewirkt dann einen Sprung an das Ende der `switch`-Anweisung (siehe auch Kapitel 8.4.2).

Fehlt die **`break`-Anweisung**, so werden die Anweisungen nach der nächsten `case`-Marke abgearbeitet. Dies geht so lange weiter, bis ein `break` gefunden wird oder bis das Ende der `switch`-Anweisung erreicht ist.

Im Gegensatz zur `if`-Anweisung prüft die `switch`-Anweisung nur auf die Gleichheit von Werten. Bei der `if`-Anweisung wird ein logischer Ausdruck ausgewertet.

8.3 Iteration

Eine Iteration ermöglicht das mehrfache (iterative) Ausführen von Anweisungen. In **Java** gibt es **abweisende Schleifen**, **annehmende Schleifen** und die **Endlos-Schleife**.

8.3.1 Abweisende Schleife mit `while`

Im Folgenden das zugehörige Nassi-Shneiderman-Diagramm:

Bild 8-6 Struktogramm der `while`-Schleife

Die Syntax der `while`-Schleife lautet:

```
while (Ausdruck)
   Anweisung
```

In einer `while`-Schleife kann eine Anweisung in Abhängigkeit von der Bewertung eines Ausdrucks wiederholt ausgeführt werden. Da der **Ausdruck vor der Ausführung** der Anweisung **bewertet wird**, spricht man auch von einer "**abweisenden**" **Schleife**. Der Ausdruck wird berechnet und die Anweisung dann und nur dann ausgeführt, wenn der Ausdruck `true` ist. Danach wird die Berechnung des Ausdrucks und die eventuelle Ausführung der Anweisung wiederholt. Um keine **Endlos-Schleife** zu erzeugen, muss daher ein Teil des Bewertungsausdrucks im Schleifenrumpf, d. h. in der Anweisung, manipuliert werden. Sollen mehrere Anweisungen ausgeführt werden, so ist ein Block zu verwenden. Das folgende **Beispiel** zeigt die Manipulation der Abbruchbedingung im Schleifenrumpf:

```
while (i < 100)
{
   . . . . .
   i++; // manipuliert Variable i der Abbruchbedingung i < 100
}
```

8.3.2 Abweisende Schleife mit `for`

Erste Erfahrungen mit der `for`-Schleife wurden bereits in Kapitel 4.1 gewonnen. Ein **Beispiel** für eine einfache `for`-Schleife ist:

```
for (int lv = 1; lv <= 5; lv++)
   System.out.println (lv);
```

Hierbei werden die Zahlen 1 bis 5 nacheinander jeweils in einer eigenen Zeile ausgegeben. Die `for`-Schleife ist wie die `while`-Schleife eine abweisende Schleife, da erst geprüft wird, ob die Bedingung für ihre Ausführung zutrifft. Die Syntax der `for`-Schleife lautet:

```
for (Initialisierungsklausel; BoolescherAusdruck;
                              Aktualisierungs-Ausdrucksliste)
   Anweisung
```

Die `for`-Anweisung ist per Sprachdefinition äquivalent zu[127]:

```
{
   Initialisierungsklausel;
   while (BoolescherAusdruck)
   {
      Anweisung
      Aktualisierungs-Ausdrucksliste
   }
}
```

[127] Vorausgesetzt, der bei der `for`-Schleife optionale Ausdruck `BoolescherAusdruck` ist tatsächlich vorhanden. Die Äquivalenz ist auch gegeben, wenn die optionale Initialisierungsklausel bzw. die Aktualisierungs-Ausdrucksliste fehlen. Fehlt der Ausdruck `BoolescherAusdruck`, so entsteht eine Endlosschleife, da in der äquivalenten `while`-Schleife anstelle des Ausdrucks `Boolescher-Ausdruck` die Konstante `true` tritt.

Beachten Sie die erste und die letzte geschweifte Klammer. Durch den umfassenden Block wird in Java eine Schleifenvariable, die in der Initialisierungsklausel `Initialisierungsklausel` definiert wurde, beim Verlassen der `for`-Schleife ungültig. Das Struktogramm der zur `for`-Anweisung äquivalenten `while`-Anweisung sieht wie folgt aus:

Bild 8-7 Struktogramm der zur `for`-Anweisung äquivalenten `while`-Schleife

Die Initialisierungsklausel

Zu Beginn der Schleife wird einmalig die Initialisierungsklausel `Initialisierungsklausel` zur Initialisierung der Schleife berechnet. In der Regel stellt die Initialisierungsklausel die **Definition und Initialisierung einer lokalen Variablen** dar, die als **Laufvariable**[128] verwendet wird und **nach der Beendigung der `for`-Schleife ungültig ist**. Eine Initialisierungsklausel kann bestehen aus:

- einem einzelnen Ausdruck
- oder einer Ausdrucksliste als Folge von Ausdrücken, getrennt durch Kommata.

Laufvariablen können vor der `for`-Schleife definiert werden. Sind die Laufvariablen alle vom gleichen Typ – z. B. es werden nur Laufvariablen vom Typ `int` verwendet – so können diese innerhalb der Initialisierungsklausel in einer einzelnen Vereinbarung definiert werden, **beispielsweise** wie folgt:

```
for (int i = 0, j = 0; i < 10 && j < 10; i++, j++)
```

BoolescherAusdruck, Anweisung und Aktualisierungs-Ausdrucksliste

Nach der Auswertung der Initialisierungsklausel wird der Ausdruck `BoolescherAusdruck` berechnet. Ist dieser Ausdruck `true`, wird die Anweisung `Anweisung` ausgeführt und anschließend die Aktualisierungs-Ausdrucksliste berechnet. Die Bewertung des Ausdrucks `BoolescherAusdruck`, die Ausführung der Anweisung `Anweisung` und die Berechnung der Aktualisierungs-Ausdrucksliste werden solange wiederholt, bis der Ausdruck `BoolescherAusdruck false` wird.

Gebräuchliche Form der `for`-Schleife in Java

In einer gebräuchlichen Form wird die `for`-Schleife so verwendet, dass die Initialisierungsklausel eine Laufvariable definiert und initialisiert und dass die Aktualisie-

[128] Die Laufvariable kann natürlich auch vor der `for`-Schleife definiert werden.

rungs-Ausdrucksliste ein einzelner Ausdruck in Form einer Zuweisung an die Laufvariable ist. Dies wird im folgenden **Beispiel** gezeigt, in welchem die Komponenten eines `int`-Arrays mit dem jeweiligen Wert der Laufvariablen initialisiert werden:

```
int[] a = new int [20];
for (int i = 0; i < 20; i = i + 1)
   a [i] = i;
```

Die Variable `i` in diesem Beispiel zählt die Zahl der Schleifendurchläufe hoch. Sie wird als **Laufvariable** bezeichnet. Für die Erhöhung des Wertes der Laufvariablen kann statt `i = i + 1` auch genauso gut `i++` oder `++i` geschrieben werden. Alle drei Schreibweisen sind hier äquivalent. Entscheidend ist nur, dass die Laufvariable erhöht wird. Der Rückgabewert der Aktualisierungs-Ausdrucksliste wird ja nicht abgeholt. Natürlich ist es von der Syntax her möglich, dass statt `i++` beispielsweise auch `x = i++`[129] geschrieben wird, wobei `x` eine bereits definierte Variable sein soll. Dann wird ebenfalls der Schleifenindex (die Laufvariable) erhöht, aber darüber hinaus noch der Wert der Variablen `x` verändert. Solche Kunststücke können leicht übersehen werden und machen deshalb das Programm schlecht lesbar.

Ausdruckslisten aus Folgen von Ausdrücken

In Java kann eine Initialisierungsklausel der `for`-Schleife entweder eine einzige Definition mit Initialisierung sein oder ein einzelner Ausdruck oder eine Liste von Ausdrücken, die durch Kommata getrennt sind. Die **Aktualisierungs-Ausdrucksliste** kann ein einzelner Ausdruck sein oder eine Liste von Ausdrücken. Mit Hilfe dieser Listen von Ausdrücken ist es möglich, mehrere Laufvariablen gleichzeitig zu bearbeiten. Im Folgenden einige **Beispiele**:

```
// Datei: Schleifen.java

public class Schleifen
{
   public static void main (String[] args)
   {
      //Dieses Beispiel funktioniert
      int i = 0;
      int j = 0;
      for (i = 0, j = 1; j >= 0; i++, j--) //Liste von Ausdrücken
      {
         System.out.println ("i: " + i);
         System.out.println ("j: " + j);
      }
      // Dieses Beispiel funktioniert auch
      for (int k = 0, l = 1; l >= 0; k++, l--)
      {
         System.out.println ("k: " + k);
         System.out.println ("l: " + l);
      }

      // Dieses Beispiel funktioniert nicht.
      // Es ist nur eine Liste von Ausdrücken zulässig, nicht
      // aber eine Liste von Definitionen von Laufvariablen.
```

[129] `x = i++` stellt ja – wie bereits in Kapitel 7.2 vorgestellt – einen Ausdruck dar.

```
      // for (int m = 0, int n = 2; n >= 0; m++, n--)
      // {
      //    System.out.println ("m: " + m);
      //    System.out.println ("n: " + n);
      // }
   }
}
```

Die Ausgabe des Programms ist:

```
i: 0
j: 1
i: 1
j: 0
k: 0
l: 1
k: 1
l: 0
```

Beachten Sie, dass `int k = 0, l = 1;` eine einzige Definition darstellt. Es entspricht von der Wirkung her aber

```
int k = 0;
int l = 1;
```

Allerdings ist in der `for`-Schleife eine Liste von Definitionen nicht zugelassen.

8.3.3 Abweisende for-each-Schleife

Die `for`-Schleife wird besonders gerne verwendet, um über Arrays oder die Elemente von Collection-Klassen (siehe Kapitel 18) zu iterieren. Bei Arrays war dazu bisher immer die Einführung einer Laufvariablen (meist `i` genannt) notwendig. Mit der **erweiterten `for`-Schleife**, die seit der Version Java 5 Bestandteil von Java ist, wird derselbe Code wesentlich kürzer und prägnanter.

Die for-each-Schleife wird für jedes Element eines Arrays (oder einer Collection) durchlaufen. Es muss also zuerst geprüft werden, ob das Array (oder die Collection) Elemente enthält. Ist die Collection leer oder hat das Array die Länge 0, so wird die for-each-Schleife nicht durchlaufen.

Das folgende **Beispiel** zeigt das Iterieren über ein Array:

```
// Datei: ForEachTest.java

public class ForEachTest
{
   public static void main (String[] args)
   {
      String[] testArray =
                     new String[] {"Hallo", "for-each", "Schleife"};
```

```
        // Array mit Hilfe der erweiterten for-Schleife auslesen.
        for (String element : testArray;
        {
           // Zugriff auf das Element des Arrays
           System.out.println (element);
        }
    }
}
```

Die Ausgabe des Programms ist:

```
Hallo
for-each
Schleife
```

In der erweiterten `for`-Schleife wird zuerst eine Variable vom Typ eines Elements des Arrays definiert, im obigen Beispiel durch `String element`. Nach dem Doppelpunkt steht der Name des zu durchlaufenden Arrays. Das obige Beispiel `for (String element : testArray)` kann gelesen werden als: "Für alle Elemente des Arrays `testarray`, das aus Referenzen auf Objekte vom Typ `String` besteht".

Die erweiterte `for`-Schleife wird auch **for-each Schleife** genannt, da sie immer über alle Elemente eines Arrays läuft. Sie kann durch eine `break`-Anweisung abgebrochen werden. Zudem kann die Reihenfolge, in der über die Elemente iteriert wird, nicht beeinflusst werden. Arrays werden immer in aufsteigender Reihenfolge durchlaufen.

Damit ist die erweiterte `for`-Schleife nicht für Aufgaben geeignet, die eine andere als die aufsteigende Reihenfolge ohne Auslassungen verlangen.

8.3.4 Endlosschleife

Fehlt der Ausdruck `BoolescherAusdruck` in einer `for`-Schleife, so gilt die Bedingung immer als `true` und die Schleife wird nicht mehr automatisch beendet. Durch Weglassen des Ausdrucks `BoolescherAusdruck` kann somit in einfacher Weise eine Endlosschleife programmiert werden. Die geläufigste Form ist dabei, alle drei Ausdrücke wegzulassen, wie im folgenden **Beispiel**:

```
for ( ; ; )       // Endlosschleife
{
    . . . . .
}
```

Beachten Sie hierbei, dass die beiden Semikolons trotzdem hingeschrieben werden müssen. Eine schönere Möglichkeit ist, die `while`-Schleife zu verwenden und die Bedingung auf `true` zu setzen:

```
while (true)      // Endlosschleife
{
```

```
   . . . . .
}
```

8.3.5 Annehmende Schleife mit `do-while`

Die Syntax der `do-while`-Schleife ist:

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

Das folgende Bild zeigt das Struktogramm der `do-while`-Schleife:

Bild 8-8 Struktogramm der `do-while`-Schleife

Die `do-while`-Schleife ist eine "annehmende Schleife". Zuerst wird die Anweisung der Schleife einmal ausgeführt. Danach wird der Ausdruck bewertet. Ist er `true`, wird die Ausführung der Anweisung und die Bewertung des Ausdrucks solange fortgeführt, bis der Ausdruck zu `false` ausgewertet wird.

Die `do-while`-Schleife wird somit auf jeden Fall mindestens einmal durchlaufen, da die Bewertung des Ausdrucks erst am Ende der Schleife erfolgt.

Das folgende **Programmbeispiel** gibt zu einer Zahl in Dezimaldarstellung den entsprechenden Wert in der Binärdarstellung aus:

```
// Datei: BinaerWandler.java
public class BinaerWandler
{
   public static void main (String[] args)
   {
      int zahl = 100;
      String binaer = "";
      // Variable, die den Rest der Division durch 2 speichert
      int rest;

      do
      {
         // Der Rest kann immer nur 1 oder 0 sein.
         rest = zahl % 2;
         zahl = zahl / 2;
         // Zusammensetzen des Strings zur Binärdarstellung
         binaer = rest + binaer;
      }while (zahl > 0);

      System.out.println ("100 dezimal ist: " + binaer + " binär");
```

```
    }
}
```

Die Ausgabe des Programms ist:

```
100 dezimal ist: 1100100 binär
```

8.4 Sprunganweisungen

Mit der `break`-Anweisung (siehe Kapitel 8.4.2) kann eine `while`-, `do-while`-, `for`-Schleife und `switch`-Anweisung abgebrochen werden. Die `continue`-Anweisung (siehe Kapitel 8.4.3) dient zum Sprung in den nächsten Schleifendurchgang bei einer `while`-, `do-while`- und `for`-Schleife. Sowohl bei `break`- als auch bei `continue`-Anweisungen können Marken (siehe Kapitel 8.4.1) verwendet werden. Zu den Sprunganweisungen zählt auch die `return`-Anweisung. Mit `return` springt man aus einer Methode an die aufrufende Stelle zurück. Die `return`-Anweisung wird in Kapitel 9.2.3 behandelt.

8.4.1 Marken

In Java können Anweisungen mit Marken versehen werden:

```
int a = 0;
int b = 1;
marke: if (a < b)
    . . . . .
```

Hierbei trennt ein Doppelpunkt die Marke von der ihr zugeordneten Anweisung. Dass eine Marke vor einer Anweisung steht, ändert nichts an dem Charakter der Anweisung. Ein Programm ist aber besser lesbar, wenn Marken vermieden werden. Für die Syntax einer Marke gelten dieselben Konventionen wie für einen Bezeichner (Namen). Der Gültigkeitsbereich einer Marke ist der Block, in dem sie enthalten ist. Eine Marke in einem äußeren Block darf denselben Namen tragen wie eine Marke in einem inneren Block. Wird zu einer Marke gesprungen, so wird zur innersten Marke mit diesem Namen gesprungen.

8.4.2 Die `break`-Anweisung

Die `break`-Anweisung ohne Marke erlaubt es, eine `for`-, `do-while`- und `while`-Schleife sowie die `switch`-Anweisung vorzeitig zu verlassen.

Im Folgenden wird die Anwendung der `break`-Anweisung bei zwei ineinander verschachtelten `for`-Schleifen gezeigt:

```
for (. . . . .)
{   ...
    for (. . . . . )
    {  ...
       if (Bedingung) break;
       ...
    }
    ...
}
```

Bild 8-9 Beispiel einer `break`-Anweisung bei geschachtelten `for`-Schleifen

Beachten Sie, dass die Abarbeitung des Programms nach der schließenden Klammer der inneren `for`-Schleife fortgesetzt wird.

Bei geschachtelten Schleifen bzw. `switch`-Anweisungen wird jeweils nur die Schleife bzw. `switch`-Anweisung verlassen, aus der mit `break` herausgesprungen wird.

Die Abarbeitung des Programms wird **mit der Anweisung fortgesetzt**, **welche direkt der verlassenen Schleife bzw. `switch`-Anweisung folgt**.

Im folgenden **Beispiel** wird eine Endlosschleife mit Hilfe von `break` verlassen. Der gezeigte Anmeldevorgang ist nur dann erfolgreich, wenn exakt "`Anja`" gefolgt von `<RETURN>` eingegeben wird. Bei korrekter Eingabe wird die Meldung "`Anmeldevorgang erfolgreich!`" ausgegeben. Bei einer Falscheingabe wird der Benutzer aufgefordert, einen erneuten Anmeldeversuch zu starten. Hier das **Beispiel**:

```
// Datei: Login.java
import java.util.Scanner;

public class Login
{
   public static void main (String[] args)
   {
      Scanner scanner = new Scanner (System.in);
      String eingabe = null;

      while (true)
      {
         System.out.print ("Bitte geben Sie Ihr Login ein: ");
         eingabe = scanner.next();

         if (eingabe.equalsIgnoreCase ("Anja"))
         {
            System.out.println ("Anmeldevorgang erfolgreich!");
            break;
         }
         else
         {
            System.out.println ("Falsche Eingabe!");
```

```
            }
        }
    }
}
```

Die Ausgabe des Programms ist:

```
Bitte geben Sie Ihr Login ein: Mathias
Falsche Eingabe!
Bitte geben Sie Ihr Login ein: Anja
Anmeldevorgang erfolgreich!
```

8.4.3 Die `continue`-Anweisung

Die `continue`-Anweisung ist wie die `break`-Anweisung eine Sprung-Anweisung. Im Gegensatz zur Anweisung `break` wird aber eine Schleife nicht verlassen, sondern der Rest der Anweisungsfolge der Schleife übersprungen und ein neuer Schleifendurchgang gestartet. Die `continue`-Anweisung kann auf die `for`-, die `while`- und die `do-while`-Schleife angewandt werden.

Bei `while` und `do-while` wird nach `continue` direkt zum **Bedingungstest der Schleife** gesprungen. Bei der **`for`-Schleife** wird **zuerst noch** die **Aktualisierungs-Ausdrucksliste** (siehe Kapitel 8.3.2) **bewertet**.

Angewandt wird die `continue`-Anweisung zum Beispiel, wenn an einer gewissen Stelle des Schleifenrumpfes mit einem Test festgestellt werden kann, ob der "umfangreiche" Rest noch ausgeführt werden muss. Das folgende Bild zeigt den Kontrollfluss bei der `continue`-Anweisung:

Bild 8-10 Kontrollfluss bei der `continue`-Anweisung für eine `for`-Schleife (a), eine `while`-Schleife (b) und eine `do-while`-Schleife (c)

Es gibt die Möglichkeit, in Verbindung mit der `continue`-Anweisung Marken zu verwenden. Soll nicht zum Bedingungstest des innersten Blocks mit der `continue`-Anweisung gesprungen werden, sondern zum Bedingungstest eines äußeren Blocks, so ist die Anweisung, die den Bedingungstest enthält, mit einer Marke `amarke` zu versehen. Mit der Anweisung `continue amarke` kann dann dieser Bedingungstest angesprungen werden. Da jedoch bei einer disziplinierten Programmierung das Springen an Marken vermieden werden kann, wird hierzu kein Beispiel gezeigt.

Das folgende **Beispiel** zeigt die Verwendung der `continue`-Anweisung in einer `while`-Schleife:

```
// Datei: Login2.java

import java.util.Scanner;

public class Login2
{
   public static void main (String[] args)
   {
      Scanner scanner = new Scanner (System.in);
      String eingabe = null;

      while (true)
      {
         System.out.print ("Bitte geben Sie Ihr Login ein: ");
         eingabe = scanner.next();
         if (!eingabe.equalsIgnoreCase ("Anja"))
         {
            System.out.println ("Falsche Eingabe!");
            continue;
         }
         System.out.println ("Anmeldevorgang erfolgreich!");
         break;
      }
   }
}
```

Es wird hier wiederum – wie im Beispiel mit der `break`-Anweisung – die Eingabe des Benutzers auf die Übereinstimmung mit "`Anja`" überprüft.

Die Ausgabe des Programms ist:

```
Bitte geben Sie Ihr Login ein: anja
Anmeldevorgang erfolgreich!
```

8.5 Zusammenfassung

Blöcke stellen Kontrollstrukturen für die Sequenz dar. Ein Block (eine zusammengesetzte Anweisung) kann an jeder Stelle stehen, an der eine einzelne Anweisung angeschrieben werden kann. Blöcke werden in Kapitel 8.1 besprochen.

Bei der Selektion in Kapitel 8.2 werden

- die bedingte Anweisung und einfache Alternative
- sowie die mehrfache Alternative mit `else-if` und diejenige mit `switch`

analysiert.

Kapitel 8.3 behandelt die Iteration.

Hierbei werden

- die abweisende Schleife mit `while`,
- die abweisende Schleife mit `for`,
- die abweisende for-each-Schleife,
- die Endlosschleife
- und die annehmende Schleife mit `do-while`

untersucht.

Kapitel 8.4 behandelt Sprunganweisungen[130].

Die `break`-Anweisung ohne Marke erlaubt es, eine `for`-, `do`-while- und `while`-Schleife sowie die `switch`-Anweisung vorzeitig zu verlassen.

Die `continue`-Anweisung ist wie die `break`-Anweisung eine Sprung-Anweisung. Im Gegensatz zur Anweisung `break` wird aber eine Schleife nicht verlassen, sondern der Rest der Anweisungsfolge der Schleife übersprungen und ein neuer Schleifendurchgang gestartet. Die `continue`-Anweisung kann auf die `for`-, die `while`- und die `do-while`-Schleife angewandt werden.

8.6 Übungen

Aufgabe 8.1: Iteration

8.1.1 Rechteck zeichnen

Das folgende Programm wurde umständlicherweise nur mit `while`- und `do-while`-Schleifen geschrieben. Schreiben Sie das Programm so um, dass es übersichtlicher wird. Verwenden Sie hierzu die `for`-Schleife.

Hier das Programm:

```
// Datei: RechteckZeichnen.java

public class RechteckZeichnen
{
   static final int BREITE = 20;
   static final int HOEHE = 10;
```

[130] Anzumerken ist, dass die Sprunganweisung `return` in Kapitel 9.2.3 behandelt wird.

```
public static void main (String[] args)
   {
      int hoehe;    // Zaehlvariable für die Hoehe
      int breite;   // Zaehlvariable für die Breite
      breite = 0;

      do
      {
         System.out.print ("*");
         breite++;
      }
      while (breite < BREITE);

      System.out.println();
      hoehe = 0;
      while (hoehe < HOEHE - 2)
      {
         System.out.print ("*");
         breite = 1;

         do
         {
            System.out.print (" ");
            breite++;
         }
         while (breite < BREITE - 1);

         System.out.println ("*");
         hoehe++;
      }

      breite = 0;
      while (breite < BREITE)
      {
         System.out.print ("*");
         breite++;
      }
      System.out.println();
   }
}
```

8.1.2 Dreieck zeichnen

Schreiben Sie ein Programm, das ein gefülltes Dreieck auf dem Bildschirm ausgibt. Geben Sie hierzu in jeder Zeile mit Hilfe einer Schleife zuerst die entsprechende Anzahl von Leerzeichen aus. Verwenden Sie dann eine zweite Schleife, um die entsprechende Anzahl von Sternchen '*' auszugeben. Verwenden Sie zur Ausgabe der einzelnen Zeichen die Methode `System.out.print()`.

Eine Beispielausgabe könnte z. B. so aussehen:

```
    *
   ***
  *****
 *******
*********
```

Aufgabe 8.2: Selektion

8.2.1 Mehrfache Alternative

Analysieren Sie das unten stehende Programm. Was erwarten Sie als Ausgabe? Schreiben Sie das Programm so um, dass es anstatt der `if-else`-Anweisungen eine `switch`-Anweisung verwendet.

Hier das Programm:

```
// Datei: Zahlen.java
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Zahlen
{
   // Verwenden Sie die Methode eingabeZahl(),
   // ohne sie genauer zu studieren
   public static int eingabeZahl()
   {
      int wert = -1;
      try
      {
         java.util.Scanner scanner =
            new java.util.Scanner (System.in);
         System.out.print ("Gib einen Wert ein: ");
         wert = scanner.nextInt();
      }
      catch (Exception e)
      {
         e.printStackTrace();
         System.exit(-1);
      }
      return wert;
   }

   public static void main (String[] args)
   {
      int zahl = eingabeZahl();

      System.out.print ("Die eingegebene Zahl ist ");
      if (zahl == 1)
         System.out.println ("EINS");
      else if (zahl == 2)
         System.out.println ("ZWEI");
      else if (zahl == 3)
         System.out.println ("DREI");
      else if (zahl == 4)
         System.out.println ("VIER");
      else if (zahl == 5)
         System.out.println ("FUENF");
      else
      {
         System.out.println ("UNBEKANNT");
      }
   }
}
```

8.2.2 Vereinfachen einer `if-else`-Anweisung

Wie lassen sich folgende Codezeilen vereinfachen?

```
if (wert > 0)
{
   if (wert < 5)
   {
      System.out.println ("Der Wert ist innerhalb 0 und 5");
   }
   else
   {
      System.out.println ("Der Wert ist ausserhalb 0 und 5");
   }
}
else
{
   System.out.println ("Der Wert ist ausserhalb 0 und 5");
}
```

Aufgabe 8.3: Sprunganweisungen

8.3.1 Endlosschleife

Ein Programmierer hat im folgenden Programmcode einen Fehler eingebaut, wodurch das Programm in einer Endlosschleife hängen bleibt. Eigentlich sollte das Programm beim Erreichen des Werts 10 beendet werden. Beheben Sie den Fehler mit Hilfe einer bedingten Sprunganweisung.

Hier ist das fehlerhafte Programm:

```
// Datei: Endlos.java
public class Endlos
{
   public static void main (String[] args)
   {
      int i = 0;
      while (true)
      {
         i++;
         System.out.println (i);
      }
   }
}
```

8.3.2 Ungerade Zahlen ausgeben

a) Ergänzen Sie das folgende Programmfragment unter Verwendung einer `continue`-Anweisung, sodass nur die ungeraden Zahlen ausgegeben werden.

   Fehlende Stellen sind mit . . . . . markiert. Hier das Programm:

```
// Datei: UngeradeZahlen.java

import java.io.BufferedReader;
import java.io.InputStreamReader;
```

```
public class UngeradeZahlen
{
   // Verwenden Sie die Methode eingabeZahl(),
   // ohne sie genauer zu studieren

   public static int eingabeZahl()
   {
      int wert = -1;
      try
      {
         java.util.Scanner scanner =
            new java.util.Scanner (System.in);
         wert = scanner.nextInt();
      }

// Datei: UngeradeZahlen.java

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class UngeradeZahlen
{
   // Verwenden Sie die Methode eingabeZahl(),
   // ohne sie genauer zu studieren
   public static int eingabeZahl()
   {
      int wert = -1;
      try
      {
         java.util.Scanner scanner =
            new java.util.Scanner (System.in);
         wert = scanner.nextInt();
      }
      catch (Exception e)
      {
         e.printStackTrace();
         System.exit(-1);
      }
      return wert;
   }

   public static void main (String[] args)
   {
      int startwert = -1;
      int endwert = -1;

      System.out.println ("Dieses Programm gibt alle " +
                          "ungeraden Zahlen zwischen " +
                          "dem Startwert und Endwert aus.");
      System.out.print ("Gib den Startwert ein: ");
      startwert = eingabeZahl();

      do
      {
         System.out.print ("Gib den Endwert ein: ");
         endwert = eingabeZahl();
```

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

         while (endwert < startwert);
         for (int i = startwert; i <= endwert; i++)
         {
            . . . . .

            System.out.println
               ("Die Zahl " + i + " ist ungerade!");
         }
      }
   }
```

b) Verändern Sie Ihre Lösung aus Teilaufgabe a), indem Sie die `for`-Schleife durch eine `while`-Schleife ersetzen. Wie verhält sich eine `continue`-Anweisung bei einer `while`-Schleife? An welcher Stelle müssen Sie aufpassen?

Kapitel 9

Blöcke und Methoden

9.1 Blöcke und ihre Besonderheiten
9.2 Methodendefinition und -aufruf
9.3 Polymorphie von Operationen
9.4 Überladen von Methoden
9.5 Parameterliste variabler Länge
9.6 Parameterübergabe beim Programmaufruf
9.7 Iteration und Rekursion
9.8 Zusammenfassung
9.9 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_9

9 Blöcke und Methoden

Ein **Block** ist eine Folge von Anweisungen, die sequenziell hintereinander ausgeführt wird.

Eine **Methode** ist eine Folge von Anweisungen, die unter einem Namen aufgerufen werden kann.

Diese beiden Sätze enthalten bereits die Definition von Block und Methode. Blöcke und Methoden werden in diesem Kapitel detailliert behandelt.

Kapitel 9.1 untersucht Blöcke und ihre Besonderheiten, während Kapitel 9.2 die Methodendefinition und den Methodenaufruf untersucht. Polymorphie von Operationen wird in Kapitel 9.3 behandelt. Das Überladen von Methoden wird in Kapitel 9.4 untersucht. Kapitel 9.5 geht auf Parameterlisten mit variabler Länge ein und Kapitel 9.6 auf die Übergabe von Parametern beim Programmaufruf. Die Prinzipien der Iteration und Rekursion werden in Kapitel 9.7 erläutert.

9.1 Blöcke und ihre Besonderheiten

Der **Block** als **Kontrollstruktur für die Sequenz** wurde bereits in Kapitel 8.1 vorgestellt. Die Anweisungen eines Blockes werden durch **Blockbegrenzer** – in C, C++ und Java sind dies die geschweiften Klammern – zusammengefasst. Statt Block ist auch die Bezeichnung **zusammengesetzte Anweisung** üblich.

Einen Block benötigt man aus zwei Gründen:

- zum einen ist der Rumpf einer Methode ein Block,
- zum anderen gilt ein Block syntaktisch als eine einzige Anweisung. Daher kann ein Block auch da stehen, wo von der Syntax her nur eine einzige Anweisung zugelassen ist wie z. B. im `if`- oder `else`-Zweig einer `if-else`-Anweisung.

Ein Block hat in Java den folgenden Aufbau:

```
{
   Anweisungen
}
```

Nach dem Blockbegrenzer, der schließenden geschweiften Klammer, kommt kein Strichpunkt.

9.1.1 Die Deklarationsanweisung

Während in einem Block in der Programmiersprache C zuerst alle Vereinbarungen – in der Regel zum Anlegen von Variablen – angeschrieben werden mussten und erst dahinter die Anweisungen:

```
{
   Vereinbarungen /* Aufbau eines */
   Anweisungen    /* Blockes in C */
}
```

können seit C++ in einem Block Vereinbarungen, also Deklarationen und Definitionen, sowie Anweisungen "wild" gemischt werden. Möglich wurde dies durch das von Stroustrup – dem Vater von C++ – ausgedachte Konzept der **Deklarationsanweisung**[131].

Mit dem Konzept einer Deklarationsanweisung wird jede Vereinbarung als Anweisung betrachtet und daher ist die Reihenfolge von Vereinbarungen und "echten" Anweisungen nicht mehr fest vorgegeben.

Java folgt hier sinngemäß C++ und daher ist es nicht erforderlich, dass zu Beginn eines Blockes erst alle Vereinbarungen angeschrieben werden müssen. Es ist aber zu beachten, dass Variablen (lokale Variablen, Instanzvariablen und Klassenvariablen) generell in Java nur definiert, aber nicht deklariert werden können.

Beim Anlegen lokaler Variablen sollte immer ein minimaler Gültigkeitsbereich angestrebt werden. Das heißt, Variablen sollten immer möglichst nahe zu ihrer ersten Verwendung angelegt werden. Ist eine Variable nur in einem Block von Nutzen, so sollte sie auch in diesem Block erst angelegt werden. So können die Ressourcen nach dem Verlassen eines Blocks freigegeben werden und ein Leser einer Methode kann sich sicher sein, dass diese Variable danach keine Rolle mehr spielt. Dies erhöht die Wart- und Lesbarkeit des Codes und macht die Programme weniger fehleranfällig.

Ein anderes Beispiel für eine Deklarationsanweisung innerhalb eines Blocks außer dem Anlegen von Variablen ist die Definition einer Klasse oder eines Aufzählungstyps. In Bild 9-1 ist die zulässige Blockstruktur für Java dargestellt:

```
{
   . . . . .
   Deklarationsanweisungen
   . . . . .
   Anweisungen
   . . . . .
   Deklarationsanweisungen
   . . . . .
   Anweisungen
   . . . . .
}
```

Bild 9-1 Zulässige Blockstruktur in Java

[131] Der Begriff "declaration statement" kommt aus dem Englischen. Im Deutschen müsste man eigentlich Vereinbarungsanweisung schreiben. Eine Deklaration einer Variablen gibt es in Java im Gegensatz zu C nicht, sondern nur Definitionen. So wäre auch der Begriff Definitionsanweisung im Deutschen im Falle von Java denkbar.

In Java können an einer beliebigen Stelle innerhalb eines Blockes Variablen mit Hilfe einer Deklarationsanweisung definiert werden.

9.1.2 Die leere Anweisung und der leere Block

Eine sogenannte **leere Anweisung** besteht nur aus einem **leeren Block** `{}` oder einem **Strichpunkt** wie in folgendem Beispiel:

```
// primitive Warteschleife des Programmes
for (int i = 0; i < 100000; i++)
   ;                  // Der Strichpunkt ist fett gedruckt,
                      // damit er auffällt.
```

Ist in einem Programm an einer von der Syntax für eine Anweisung vorgesehenen Stelle keine Anweisung notwendig, so muss dort eine leere Anweisung, d. h. ein Strichpunkt oder ein leerer Block, stehen, um die Syntax zu erfüllen.

Damit man ein Semikolon als leere Anweisung besser erkennt, wird üblicherweise das Semikolon für sich allein in einer eigenen Zeile geschrieben.

Zur **defensiven Programmierung** gehört es, dass bei Selektionen und Iterationen sowie auch bei leeren Anweisungen stets ein **Block** verwendet wird. Damit sind Erweiterungen um Anweisungen nicht fehlerträchtig, da die Klammern schon da sind.

9.1.3 Lokale Variablen

Variablen, die innerhalb eines Blockes vereinbart werden, sind lokal für diesen Block und werden **lokale Variablen** genannt. Sie werden angelegt, wenn der entsprechende Block aufgerufen wird und im Programmcode des Blocks die Definition[132] der Variablen erreicht wird.

Ein Block zählt syntaktisch als eine einzige Anweisung. Im Gegensatz zu einer normalen Anweisung besteht bei einem Block jedoch die Möglichkeit, Block-lokale Variablen einzuführen.

Lokale Variablen werden durch das Laufzeitsystem auf dem Stack angelegt. Beim Verlassen des Blocks, d. h. beim Erreichen der schließenden geschweiften Klammer, werden die lokalen Variablen wieder ungültig und werden auf dem Stack zum Überschreiben freigegeben.

[132] In Java bedeutet die Vereinbarung einer Variablen stets die Definition dieser Variablen.

9.1.4 Schachtelung von Blöcken

Da eine Anweisung eines Blocks selbst wieder ein Block sein kann, können Blöcke geschachtelt werden.

Dies zeigt das folgende Bild:

Bild 9-2 Schachtelung von Blöcken

In Java können in jedem Block – auch in inneren Blöcken – Variablen definiert werden.

In einem inneren Block definierte Variablen sind nur innerhalb dieses Blockes sichtbar, in einem umfassenden Block sind sie unsichtbar. Variablen, die in einem umfassenden Block definiert sind, sind für einen folgenden inneren Block auch sichtbar.

Bei Java sind identische Namen im inneren und äußeren Block nicht zugelassen. Es resultiert ein Kompilierfehler.

Das folgende **Beispiel** demonstriert die **Sichtbarkeit** von Variablen in inneren und äußeren Blöcken. Auf Variablen, die in äußeren Blöcken definiert wurden, kann in inneren Blöcken zugegriffen werden. Dies demonstriert das folgende Beispiel:

```
// Datei: BlockTest.java

public class BlockTest
{
   public void zugriff()
   {
      int aussen = 7;

      if (aussen == 7)
      {
         int innen = 8;
         System.out.print ("Zugriff auf die Variable");
         System.out.println (" des äußeren Blocks: " + aussen);
         System.out.print ("Zugriff auf die Variable");
         System.out.println (" des inneren Blocks: " + innen);
      }
   }
```

```
   public static void main (String[] args)
   {
      BlockTest ref = new BlockTest();
      ref.zugriff();
   }
}
```

Die Ausgabe des Programms ist:

```
Zugriff auf die Variable des äußeren Blocks: 7
Zugriff auf die Variable des inneren Blocks: 8
```

9.1.5 Gültigkeit, Sichtbarkeit und Lebensdauer

Im Folgenden werden neben lokalen Variablen auch Datenfelder betrachtet.

Die **Lebensdauer** ist die Zeitspanne, in der die Java-Virtuelle Maschine einer **Variablen** einen **Platz im Arbeitsspeicher** zur Verfügung stellt. Mit anderen Worten, während ihrer Lebensdauer besitzt eine Variable einen Speicherplatz.

Die **Gültigkeit einer Variablen** bedeutet, dass an einer Programmstelle der **Name einer Variablen dem Compiler** durch eine Vereinbarung **bekannt** ist.

Die **Sichtbarkeit** einer Variablen bedeutet, dass man von einer Programmstelle aus die Variable sieht, das heißt, dass man auf die Variable **über den Variablennamen zugreifen** kann.

Eine Variable kann aber gültig sein und von einer Variablen desselben Namens verdeckt werden und deshalb nicht sichtbar sein. Ein **lokaler Variablenname** kann ein **Datenfeld** mit demselben Namen **verdecken**. Dann ist das **Datenfeld** zwar **gültig**, **aber nicht sichtbar**. Es ist aber möglich, mit Hilfe der `this`-Referenz (siehe Kapitel 10.3) auf eine **verdeckte Instanzvariable** zuzugreifen. Auf eine **verdeckte Klassenvariable** kann über den Klassennamen oder über eine Referenz auf ein Objekt dieser Klasse zugegriffen werden (siehe Kapitel 11.3.3.2).

Das folgende **Programmbeispiel** zeigt den Zugriff auf ein verdecktes Datenfeld:

```
// Datei: Sichtbar.java

public class Sichtbar
{
   private int x = 0;          // Datenfeld x

   public void zugriff()
   {
```

```
      int x = 7;              // lokale Variable x

      // Ausgabe der lokalen Variablen x
      System.out.println ("Lokale Variable x: " + x);

      // this zeigt auf das aktuelle Objekt und damit ist this.x die
      // x-Komponente des aktuellen Objektes
      // Ausgabe des Datenfeldes x
      System.out.println ("Datenfeld x: " + this.x);
   }

   public static void main (String[] args)
   {
      Sichtbar sicht = new Sichtbar();
      sicht.zugriff();
   }
}
```

Die Ausgabe des Programms ist:

```
Lokale Variable x: 7
Datenfeld x: 0
```

Wird das Verdecken von Datenfeldern durch lokale Variablen außer Acht gelassen, sind in Java Sichtbarkeits- und Gültigkeitsbereich identisch, wie in folgender Tabelle zu sehen ist:

Variable	Sichtbarkeits- und Gültigkeitsbereich	Lebensdauer
Lokal	im Block ab Definition und in inneren Blöcken	Block ab Definition
Instanzvariable	im Objekt selbst[133], solange das Objekt existiert	vom Anlegen des Objektes, bis das Objekt nicht mehr referenziert wird
Klassenvariable	in allen Objekten der entsprechenden Klasse und in allen zugehörigen Klassenmethoden[134]	vom Laden der Klasse, bis die Klasse nicht mehr benötigt wird

Tabelle 9-1 Sichtbarkeit, Gültigkeit und Lebensdauer

Bei lokalen Variablen fallen Gültigkeit und Sichtbarkeit zusammen. Bei Datenfeldern muss man prinzipiell zwischen Gültigkeit und Sichtbarkeit unterscheiden.

[133] Bei entsprechenden Zugriffsmodifikatoren kann auch aus anderen Klassen zugegriffen werden. Darauf wird an späterer Stelle eingegangen.

[134] Bei entsprechenden Zugriffsmodifikatoren kann auch aus anderen Klassen zugegriffen werden. Darauf wird an späterer Stelle eingegangen.

9.2 Methodendefinition und -aufruf

Methoden stellen Anweisungsfolgen dar, die unter einem Namen aufgerufen werden können. Methoden werden stets für Objekte – im Falle von **Instanzmethoden** – bzw. Klassen – im Falle von **Klassenmethoden** – aufgerufen.

Wie aus Kapitel 6.3.2.1 bekannt, besteht die Definition einer Methode in Java aus der Methodendeklaration, dem Methodenkopf, und dem Methodenrumpf:

```
Methodendeklaration // Methodenkopf
{                   //
                    // Methodenrumpf
}                   //
```

Methoden können einen Rückgabewert haben. Sie können auch Übergabeparameter besitzen. Der **Methodenrumpf** stellt einen Block dar. Im Methodenrumpf stehen die Anweisungen der Methode.

Die **Methodendeklaration** sieht im allgemeinen Fall folgendermaßen aus:

```
Modifikatoren Rückgabetyp Methodenname (Typ1 formalerParameter1,
                                        Typ2 formalerParameter2,
                                        . . . . .
                                        TypN formalerParameterN)
```

Ein Beispiel für einen Modifikator ist das Schlüsselwort `static`. Ein Beispiel für einen Rückgabetyp ist `int`.

9.2.1 Parameterlose Methoden

Bei parameterlosen Methoden wie z. B.:

```
int getX()      // Deklaration
{               // Definition des Rumpfes
   return x;    // der parameterlosen
}               // Methode getX()
```

folgt in der Deklaration dem Methodennamen ein **leeres Paar runder Klammern**. Der Aufruf erfolgt durch Anschreiben des Methodennamens, gefolgt von einem leeren Paar runder Klammern, z. B.:

```
alpha = ref.getX();     // Aufruf
```

Dabei stellt `ref` eine Referenz auf ein Objekt dar[135].

[135] Klassenmethoden können auch über den Klassennamen aufgerufen werden.

9.2.2 Methoden mit Parametern

Hat eine Methode formale Parameter – das sind die Parameter in den runden Klammern der Deklaration der Methode – so muss der Aufruf mit **aktuellen Parametern** erfolgen (siehe auch Kapitel 9.2.4).

Beispiel:

```
void setX (int var)     // var ist der Name des formalen
{                       // Parameters. Der Typ von var ist int
   x = var;
}
```

Der Aufruf von `setX()` kann beispielsweise erfolgen durch:

```
ref.setX (intAusdruck);
```

Hierbei ist `intAusdruck` der aktuelle Parameter.

9.2.3 Der Rückgabewert – die `return`-Anweisung

Die Methodendeklaration einer Methode[136] beinhaltet im Minimalfall den Rückgabetyp sowie den Namen der Methode und ein Paar runder Klammern. Anstelle eines Rückgabetyps kann auch das Schlüsselwort `void` stehen.

Eine Methode zur Rückgabe des Wertes eines Datenfeldes `x` könnte wie folgt aussehen:

```
int getX()
{
   return x;
}
```

Mit Hilfe der `return`-Anweisung ist es möglich, den Wert eines Ausdrucks an den Aufrufer der Methode zurückzugeben. Nach `return` kann ein Ausdruck stehen:

```
return expression;
```

Im obigen Beispiel steht `x` als Ausdruck hinter `return`. Es kann aber auch ein beliebiger anderer Ausdruck wie beispielsweise `x * x` hinter `return` stehen. Der Typ des zurückzugebenden Wertes steht vor dem Methodennamen. Der zurückgegebene Wert ist im Beispiel also vom Typ `int`. Zurückgegeben wird der Wert des Ausdrucks hinter dem `return`, im Beispiel also der Wert von `x`. Stimmen Rückgabetyp und Typ des zurückzugebenden Ausdrucks nicht überein, so erfolgt eine **implizite Typkonvertierung des rückzugebenden Ausdrucks** in den Rückgabetyp der Methode, wenn die Typen verträglich sind (siehe Kapitel 7.7.3.5). Sind die Typen nicht verträglich, so resultiert ein Kompilierfehler.

---

[136] Der Einfachheit halber wird hier eine parameterlose Methode betrachtet. Die Aussagen zu `return` gelten natürlich auch für Methoden mit Parametern.

Wird das Schlüsselwort `void` statt eines Rückgabetyps angegeben, so ist kein `return` notwendig. Es kann aber jederzeit mit `return` die Abarbeitung einer Methode abgebrochen werden. Damit wird ein sofortiger Rücksprung zur Aufrufstelle bewirkt. In diesem Fall darf mit der `return`-Anweisung kein Wert zurückgegeben werden.

Abholung des Rückgabewerts

Wird keine `return`-Anweisung angegeben, so wird der Methodenrumpf bis zu seinem Ende abgearbeitet. Ist nicht das Schlüsselwort `void`, sondern ein Rückgabetyp angegeben, so ist ein Rücksprung mit `return` erforderlich und es muss immer ein zum Rückgabetyp kompatibler Ausdruck hinter `return` stehen.

Eine `return`-Anweisung ohne einen nachfolgenden Ausdruck beendet die Ausführung einer Methode, liefert aber keinen Wert an den Aufrufer. Gleiches gilt, wenn das Ende des Programmtextes einer Methode – also die abschließende geschweifte Klammer des Methodenrumpfes – erreicht wird.

Eine Methode kann **mit `return`** nur **einen einzigen Wert** zurückgeben. Möchte man mehrere Werte zurückgeben, so kann dies über Referenzen auf Objekte in der Parameterliste gehen oder aber über die Schaffung eines Objektes mit mehreren Datenfeldern, auf das mit `return` eine Referenz zurückgegeben wird.

Gibt die Methode einen Wert zurück, so kann dieser – er muss es aber nicht – abgeholt werden, z. B. indem man den Rückgabewert einer Variablen zuweist, wie in folgendem **Beispiel**

```
alpha = ref.getX();
```

oder aber an eine andere Methode übergibt, beispielsweise durch

```
System.out.println (ref.getX());
```

Ebenso ist es erlaubt, den Rückgabewert wie in folgendem Beispiel zu ignorieren. Die Methodendeklaration soll hierbei

```
boolean insert (String s)
```

lauten. Dabei soll `insert()` die Instanzmethode eines Objektes sein, auf das `ref` zeigt. Der folgende Methodenaufruf ist dann erlaubt:

```
ref.insert ("Hanna");    // hier wird der Rückgabewert nicht
                         // abgeholt
```

An der aufrufenden Stelle darf der Wert, den eine Methode liefert, ignoriert werden. Mit anderen Worten, man kann eine Methode, die einen Rückgabewert hat, einfach aufrufen, ohne den Rückgabewert abzuholen.

9.2.4 Formale und aktuelle Parameter

In der Parameterliste der Methodendeklaration werden sogenannte formale **Parameter** aufgelistet:

```
Modifikatoren Rückgabetyp Methodenname (Typ1 formalerParameter1,
                                        Typ2 formalerParameter2,
                                             . . . . .
                                        TypN formalerParameterN)
```

Mit den **formalen Parametern** wird festgelegt, **wie viele Übergabeparameter** existieren, **von welchem Typ** diese sind **und welche Reihenfolge** sie haben.

Die Bezeichnung "**formal"** soll andeuten, dass die formalen Parameter **zur Beschreibung der Methode verwendet** werden.

Beim Aufruf einer Methode werden den formalen Parametern die Werte der **aktuellen Parameter** des Aufrufs zugewiesen.

Die formalen Parameter sind Variablen, welche die Werte der aktuellen Parameter des Aufrufs entgegennehmen. Mit den Werten der aktuellen Parameter wird dann die Methode ausgeführt.

Beim Aufruf einer Methode mit Parametern finden Zuweisungen statt. Der **Wert** eines **aktuellen Parameters** wird dem entsprechenden **formalen Parameter zugewiesen**. Eine solche Aufrufschnittstelle wird als **call-by-value-Schnittstelle** bezeichnet.

Die Namen der formalen Parameter können völlig frei vereinbart werden. Sie sind nur lokal in der jeweiligen Methode sichtbar. Der formale Parameter kann denselben Namen wie der aktuelle Parameter haben, muss es aber nicht.

Hat beispielsweise die Methode `setX()` den formalen Parameter `newX` vom Typ `int`, wie aus der Methodendeklaration `void setX (int newX)` ersichtlich ist, so wird der aktuelle Parameter `intAusdruck`, der beim Methodenaufruf `ref.setX (intAusdruck)` übergeben wird, dem formalen Parameter `newX` beim Aufruf zugewiesen.

Beim Aufruf wird ein **formaler Parameter als spezielle lokale Variable auf dem Stack angelegt** und **mit dem Wert des aktuellen Parameters** initialisiert. Dies kann man sich für das obige Beispiel so vorstellen, als ob quasi eine manuelle Initialisierung (siehe Kapitel 10.4.2) der lokalen Variablen `newX` bei ihrer Definition durchgeführt würde:

```
int newX = intAusdruck;
```

Ein **formaler Parameter** hat den **Charakter einer lokalen Variablen**. Mit anderen Worten, ein formaler Parameter stellt eine spezielle lokale Variable dar. Dies hat zur Konsequenz, dass eine im Methodenrumpf definierte lokale Variable nicht gleich heißen darf wie ein formaler Parameter.

Ein **formaler Parameter** stellt stets eine **Variable** dar. Ein aktueller Parameter muss keine Variable sein. Ein **aktueller Parameter** ist irgendein **Ausdruck** eines passenden Typs, den ein Aufrufer an den entsprechenden formalen Parameter übergibt.

9.2.5 Übergabe von einfachen Datentypen und Referenzen

Generell finden bei **Übergabeparametern** genauso wie bei **Rückgabewerten Kopiervorgänge** statt. Unabhängig davon, ob es sich um einfache Datentypen oder Referenzen handelt, werden Werte kopiert.

Bei einfachen Datentypen stellen die Werte Zeichen, Zahlen oder boolesche Werte der Anwendung dar, im Falle von Referenzen werden deren logische Adressen kopiert.

Übergabe von einfachen Datentypen

Ist der formale Parameter von einem **einfachen Datentyp**, so wird der Wert des aktuellen Parameters in den formalen Parameter kopiert. Damit sind formaler und aktueller Parameter vollständig entkoppelt. Änderungen am formalen Parameter haben keine Auswirkungen auf den aktuellen Parameter. Da der Wert des aktuellen Parameters zugewiesen wird, braucht der aktuelle Parameter keine Variable zu sein, sondern kann ein beliebiger Ausdruck sein. Da bei einem formalen Parameter eines einfachen Datentyps der Wert übergeben wird, spricht man auch von einem **call-by-value**.

Bei einfachen Datentypen als Übergabeparameter wirken sich Änderungen am Wert des formalen Parameters – genauso wie bei Referenztypen – nur auf die Kopie, d. h. den formalen Parameter, aus. Es gibt keinerlei Rückwirkungen auf das Original, d. h. den aktuellen Parameter.

Übergabe von Referenzen

Auf Objekte wird in Java über **Referenzen** zugegriffen. Beim Aufruf einer Methode wird dem formalen Parameter der Wert des aktuellen Parameters zugewiesen (**call-by-value**), d. h. eine Referenzvariable als formaler Parameter erhält als Kopie die Referenz auf das Objekt, das der aktuelle Parameter referenziert. Es gilt auch hier, dass **formaler und aktueller Parameter entkoppelt** sind. Änderungen am formalen Parameter haben keine Auswirkungen auf den aktuellen Parameter. Der aktuelle

Parameter kann ein Ausdruck sein. Dieser Ausdruck muss aber eine Referenz als Rückgabewert haben.

Da jedoch im Falle von Referenzen Kopie und Original dasselbe Objekt referenzieren, kann aus der Methode heraus über den Zugriff auf den formalen Parameter das Original verändert werden. Das Bild 9-3 zeigt, wie die Referenz `ref` auf ein Objekt der Klasse `RefTyp` zeigt:

Bild 9-3 Der formale Parameter referenziert dasselbe Objekt wie der aktuelle Parameter

Beim Aufruf der Methode `methode2()` im folgenden Beispiel wird dem formalen Parameter `refPara` der Wert des aktuellen Parameters `ref` zugewiesen. Nach dieser Zuweisung zeigen beide Referenzen auf das gleiche Objekt.

Werden Referenzen übergeben, so referenziert der formale Parameter dasselbe Objekt wie der aktuelle Parameter. Eine **Operation auf dem formalen Referenzparameter erfolgt** auf dem Objekt, auf das die Referenz zeigt, in anderen Worten a**uf dem referenzierten Objekt**.

Call-by-value und simulierter call-by-reference

Da eine Referenz kopiert wird und man mit Hilfe dieser Referenz auf ein Objekt zugreifen kann, spricht man auch von einem **simulierten call-by-reference**. Tatsächlich liegt jedoch wie bei einfachen Datentypen eine **call-by-value Schnittstelle** vor, da der Wert des aktuellen Parameters in den formalen Parameter kopiert wird.

Sowohl **einfache Datentypen** (`int`, `char`, etc.) als auch **Referenzen** werden "**by value**" übergeben. Da Referenzen aber auf Objekte zeigen, wird quasi ein "call-by-reference" simuliert.

Beispiel

Dass kopiert wird, soll das nachfolgende Beispielprogramm für Übergabeparameter verdeutlichen. Der formale Parameter `par` der Methode `methode1()` ist von einem einfachen Datentyp, der formale Parameter `refPara` der Methode `methode2()` stellt eine Referenz auf ein Objekt der Klasse `RefTyp` dar. In beiden Fällen wird der Wert des aktuellen Parameters in den formalen Parameter kopiert. Die Änderungen, welche die Methode `methode1()` an dem formalen Parameter `par` vornimmt, haben keine Auswirkung auf den aktuellen Übergabeparameter `var`. Genausowenig hat eine Änderung an der Referenzvariablen `refPara` als formaler Parameter eine Auswirkung auf den aktuellen Parameter, die Referenzvariable `ref`. Das Entscheidende jedoch ist, dass über die kopierte Referenz – hier `refPara` – auf ein

Objekt zugegriffen werden kann und über diese Referenz die Datenfelder dieses Objektes geändert werden können.

Hier das **Beispiel**:

```
// Datei: Parameter.java

class RefTyp
{
   int x;
}

public class Parameter
{
   public static void methode1 (int par)
   {
      par = 2;        // Änderung an der Kopie
   }

   public static void methode2 (RefTyp refPara)
   {
      refPara.x = 2;  // Änderung an dem Datenfeld x des Objektes,
                      // auf das refPara zeigt
   }

   public static void main (String[] args)
   {
      int var = 1;
      RefTyp ref = new RefTyp();
      ref.x = 1;

      System.out.println ("Übergabeparameter ist von einem" +
                          " einfachen Datentyp");
      System.out.println ("aktueller Parameter vor Aufruf : "+ var);
      methode1 (var);
      System.out.println ("aktueller Parameter nach Aufruf: "+ var);

      System.out.println ("Übergabeparameter ist ein Referenztyp");
      System.out.println ("Datenfeld vor Aufruf : " + ref.x);
      methode2 (ref);
      System.out.println ("Datenfeld nach Aufruf: " + ref.x);
   }
}
```

Die Ausgabe des Programms ist:

```
Übergabeparameter ist von einem einfachen Datentyp
aktueller Parameter vor Aufruf : 1
aktueller Parameter nach Aufruf: 1
Übergabeparameter ist ein Referenztyp
Datenfeld vor Aufruf : 1
Datenfeld nach Aufruf: 2
```

9.2.6 Auswertungsreihenfolge der aktuellen Parameter

Die **Auswertung der aktuellen Parameter** in der Parameterliste erfolgt **von links nach rechts**. Die genauen Abläufe beim Aufruf einer Methode sollen am folgenden **Beispiel** erklärt werden:

```
// Datei: Auswertung.java

public class Auswertung
{
   public static void main (String[] args)
   {
      int aktuell = 1;
      methode (aktuell++, aktuell);
      System.out.println ("Nach Methodenaufruf:");
      System.out.println ("Wert von aktuell: " + aktuell);
   }

   public static void methode (int formalA, int formalB)
   {
      System.out.println ("Innerhalb der Methode:");
      System.out.println ("Wert von formalA: " + formalA);
      System.out.println ("Wert von formalB: " + formalB);
   }
}
```

Die Ausgabe des Programms ist:

```
Innerhalb der Methode:
Wert von formalA: 1
Wert von formalB: 2
Nach Methodenaufruf:
Wert von aktuell: 2
```

Beim Aufruf der Methode `methode()` laufen folgende Zuweisungen ab:

```
formalA = aktuell++;
formalB = aktuell;
```

Als aktuelle Werte werden die Rückgabewerte der Ausdrücke `aktuell++` und `aktuell` an die formalen Parameter der Methode `methode()` zugewiesen.

In Java werden die aktuellen Parameter von links nach rechts bewertet.

Zuerst wird also der erste aktuelle Parameter ausgewertet. Der Rückgabewert 1 des Ausdrucks `aktuell++` wird dem ersten formalen Parameter zugewiesen. Nach der Bewertung des ersten aktuellen Parameters hat die Variable `aktuell` den Wert 2. Dieser Wert wird dem zweiten formalen Parameter zugewiesen.

9.2.7 Beispielprogramm für die Verwendung von Methoden

Im Folgenden soll ein größeres Beispiel die Verwendung von verschiedenen Methoden zeigen. Die Klasse `IntArray` hat die Aufgabe, ein `int`-Array zu kapseln und komfortable Methoden bereitzustellen:

- Die beiden Methoden `min()` und `max()` geben jeweils den minimalen bzw. maximalen Wert im Array zurück.
- Die Methode `average()` hat die Aufgabe, den Durchschnitt aller Arraywerte zu berechnen.
- Die Methode `expand()` hat die Aufgabe, das Array zu vergrößern. Die Zahl der zusätzlich anzulegenden Array-Elemente wird durch den Wert des Übergabeparameters festgelegt.
- Die Methode `sort()` hat die Aufgabe, das Array zu sortieren. Der kleinste Wert soll sich nach dem Sortieren im Element mit dem Index 0 befinden. Als Sortierverfahren wird der "**Bubble Sort**"-Algorithmus benutzt. Beim Bubble Sort werden jeweils benachbarte Elemente vertauscht, wenn sie nicht wie gewünscht geordnet sind. Dabei steigt das jeweils größte Element wie eine Blase im Wasser auf, was dem Verfahren seinen Namen gegeben hat.
- Die Methode `swap()` tauscht den Inhalt von zwei Array-Elementen mit gegebenen Indexwerten.

Hier das Programm:

```
// Datei: IntArray.java

public class IntArray
{
   private int[] arrayOfInt = null;

   public IntArray()
   {
      arrayOfInt = new int [1];
   }

   // Erweitern der Arraygröße um anzahlElemente Array-Elemente.
   public void expand (int anzahlElemente)
   {
      int size = arrayOfInt.length;
      // neues größeres Array anlegen
      int[] tmp = new int [size + anzahlElemente];
      // bestehendes zu kleines Array umkopieren
      for (int i = 0; i < size; i++)
      {
         tmp [i] = arrayOfInt [i];
      }
      arrayOfInt = tmp;
   }

   public int max()
   {
      int max = arrayOfInt [0];
      for (int element : arrayOfInt)
```

```
      {
         // Ist ein Element größer als das vorliegende Maximum, so
         // wird sein Wert zum neuen Maximum.
         if (element > max)
            max = element;
      }
      return max;
   }

   public int min()
   {
      int min = arrayOfInt [0];
      for (int element : arrayOfInt)
      {
         if (element < min)
            min = element;
      }
      return min;
   }

   public void put (int index, int newValue)
   {
      // Liegt die Position, an die der neue Wert geschrieben werden
      // soll, außerhalb der aktuellen Dimension, dann muss dass
      // Array vergrößert werden.
      if (arrayOfInt.length <= index)
         expand (index - arrayOfInt.length + 1);
      arrayOfInt [index] = newValue;
   }

   public int get (int index)
   {
      if (arrayOfInt.length > index)
         return arrayOfInt [index];
      // Fehlerfall, der angegebene Index ist zu groß.
      return -1;
   }

   public void swap (int index1, int index2)
   {
      if ((index1 < 0) || (index2 < 0))
         return;
      int size = arrayOfInt.length;
      if ((index1 > size) || (index2 > size))
         return;
      int hilf = arrayOfInt [index1];
      arrayOfInt [index1] = arrayOfInt [index2];
      arrayOfInt [index2] = hilf;
   }

   public float average()
   {
      // Es ist ein Cast erforderlich, da Gleitpunktkonstanten vom
      // Typ double sind.
      float average = (float) 0.0;
      for (int element : arrayOfInt)
```

```
      {
         average += element;
      }

      average = average / arrayOfInt.length;
      return average;
   }

   public void sort()
   {
      // Anmerkung: Zu Beginn des bubblesort-Algorithmus ist die
      // Obergrenze gleich der Dimension des zu sortierenden Arrays,
      // d. h. gleich der Anzahl seiner Elemente
      // Hier der bubblesort-Algorithmus:
      // while Obergrenze > Index des 2. Feldelementes:
      // Gehe in einer Schleife vom 2. bis zum letzten zu sortie-
      // renden Array-Element (dessen Array-Index ist um 1 geringer
      // als die Obergrenze). Wenn ein Element kleiner ist als sein
      // Vorgänger, werden beide vertauscht. (Hinweis: Nach dem
      // ersten Durchlauf steht das größte Element am Ende). Nun
      // wird die Obergrenze um 1 verringert.

      int obergrenze = arrayOfInt.length;
      while (obergrenze > 1)
      {
         for (int lauf = 1; lauf < obergrenze; lauf++)
         {
            if (arrayOfInt [lauf] < arrayOfInt [lauf - 1])
               swap (lauf, lauf - 1);
         }
         obergrenze--;
      }
   }

   public void print()
   {
      System.out.println ("Ausgabe des Array-Inhaltes: ");
      for (int i = 0; i < arrayOfInt.length; i++)

      {
         System.out.print ('\t' + "Index: " + i + " Wert: ");
         System.out.println (arrayOfInt [i]);
      }
   }
}

// Datei: IntArrayTest.java

public class IntArrayTest
{
   public static void main (String[] args)
   {
      int[] array = {4, 19, 20, 7, 36, 18, 1, 5};
      IntArray intArray = new IntArray();

      // Das intArray mit den Werten von array füllen
      for (int i = 0; i < array.length; i++)
         intArray.put (i, array [i]);
      intArray.print();
```

```
        System.out.println ("Minimum: " + intArray.min());
        System.out.println ("Maximum: " + intArray.max());
        System.out.println ("Average: " + intArray.average());
        intArray.sort();
        intArray.print();
    }
}
```

Die Ausgabe des Programms ist:

```
Ausgabe des Array-Inhaltes:
        Index: 0 Wert: 4
        Index: 1 Wert: 19
        Index: 2 Wert: 20
        Index: 3 Wert: 7
        Index: 4 Wert: 36
        Index: 5 Wert: 18
        Index: 6 Wert: 1
        Index: 7 Wert: 5
Minimum: 1
Maximum: 36
Average: 13.75
Ausgabe des Array-Inhaltes:
        Index: 0 Wert: 1
        Index: 1 Wert: 4
        Index: 2 Wert: 5
        Index: 3 Wert: 7
        Index: 4 Wert: 18
        Index: 5 Wert: 19
        Index: 6 Wert: 20
        Index: 7 Wert: 36
```

9.3 Polymorphie von Operationen

Ein Methodenkopf in einer Klasse muss eindeutig sein und darf nicht mehrmals vorkommen. Insbesondere darf es in einer Klasse nicht verschiedene Implementierungen zu demselben Methodenkopf geben. Es ist aber problemlos möglich, dass Methoden in verschiedenen nicht verwandten Klassen[137] mit jeweils gleichen Methodenköpfen existieren und jeweils eine andere Implementierung haben. Dies liegt daran, dass eine Methode ja zu einer Klasse gehört und jede nicht verwandte Klasse[138] einen eigenen Namensraum darstellt.

Eine **Klasse** stellt einen **Namensraum** dar. Damit ist es möglich, dass verschiedene **nicht verwandte Klassen** die **gleiche** Operation implementieren, d. h. den **gleichen** Methodenkopf und die **gleiche** Semantik der Operationen haben.

[137] Es gibt noch eine andere Bedeutung des Begriffs Polymorphie: Methoden in einer **Vererbungshierarchie**, die aus einer überschreibenden und überschriebenen Methode bestehen, werden auch als polymorphe Methoden bezeichnet. Dies wird in Kapitel 11 erklärt.

[138] Klassen, die durch Vererbung auseinander hervorgehen, sind verwandt.

Je nach Klasse kann eine Operation in verschiedenen Implementierungen – sprich in verschiedener Gestalt – auftreten. Man spricht hierbei auch von der **Vielgestaltigkeit** (**Polymorphie**) **von Operationen**.

Ein einfaches Beispiel ist die Methode `print()`. Alle Klassen, die ihren Objekten die Möglichkeit geben wollen, auf dem Bildschirm Informationen über sich auszugeben, stellen eine `print()`-Methode zur Verfügung. Von außen betrachtet macht die `print()`-Methode – unabhängig davon, zu welcher Klasse sie gehört – immer das Gleiche – sie gibt Informationen auf dem Bildschirm aus. Vom Standpunkt der Implementierung aus sind diese Methoden aber grundverschieden, weil jede `print()`-Methode einen für die jeweilige Klasse spezifischen Methodenrumpf hat. Das folgende **Beispiel** zeigt die Polymorphie von Methoden anhand der Klasse `Person2` und der Klasse `Bruch2`:

```
// Datei: Person2.java

public class Person2
{
   private String name;
   private String vorname;
   private int alter;

   // Konstruktur für die Initialisierung der Datenfelder
   public Person2 (String name, String vorname, int alter)
   {
      this.name = name;
      this.vorname = vorname;
      this.alter = alter;
   }

   public void print()
   {
      System.out.println ("Name    : " + name);
      System.out.println ("Vorname : " + vorname);
      System.out.println ("Alter   : " + alter);
   }
}

// Datei: Bruch2.java

public class Bruch2
{
   private int zaehler;
   private int nenner;

   public Bruch2 (int zaehler, int nenner)
   {
      this.zaehler = zaehler;
      this.nenner = nenner;
   }

   public void print()
   {
      System.out.print ("Der Wert des Quotienten von " + zaehler);
```

```
      System.out.print (" und " + nenner + " ist " + zaehler
                        + " / ");
      System.out.println (nenner);
   }
}

// Datei: Polymorphie.java

public class Polymorphie
{
   public static void main (String[] args)
   {
      Bruch2 b;
      b = new Bruch2 (1, 2);
      b.print();

      Person2 p;
      p = new Person2 ("Müller", "Fritz", 35);
      p.print();
   }
}
```

Die Ausgabe des Programms ist:

```
Der Wert des Quotienten von 1 und 2 ist 1 / 2
Name    : Müller
Vorname : Fritz
Alter   : 35
```

Beide Klassen implementieren jeweils eine `print()`-Methode. Die Klasse `Polymorphie` dient als Testklasse. In der `main()`-Methode der Klasse Polymorphie wird jeweils ein Objekt beider Klassen erzeugt und die `print()`-Methode für jedes erzeugte Objekt aufgerufen.

Jedes Objekt trägt die Typinformation, von welcher Klasse es ist, immer bei sich. Das heißt, dass ein **Objekt** immer **weiß**, **zu welcher Klasse es gehört**. Damit kann ein Methodenaufruf für ein Objekt immer der entsprechenden Klasse zugewiesen werden.

9.4 Überladen von Methoden

Methoden, die grundverschieden sind, tragen verschiedene Methodennamen.

Oftmals verrichten verschiedene Methoden dieselbe Aufgabe, allerdings für verschiedene Datentypen der Übergabeparameter. Denken Sie z. B. an eine Ausgabe-Methode, welche die Ausgabe eines Übergabeparameters auf den Bildschirm bewerkstelligt. Je nach Datentyp dieses Parameters braucht man eine andere Methode. Jede der Methoden muss dabei im Detail etwas anderes tun, um die Ausgabe durchzuführen.

Erlaubt eine Sprache das **Überladen von Methoden** (engl. **overloading**), so können Methoden mit verschiedenen Parameterlisten denselben Namen tragen. Der Aufruf der richtigen Methode ist Aufgabe des Compilers.

Anhand der Datentypen der Übergabeparameter erkennt der Compiler, welche der Methoden gemeint ist.

Der Nutzen des Überladens von Methoden ist, dass man gleichartige Methoden mit dem gleichen Namen ansprechen kann. Die Verständlichkeit der Programme kann dadurch erhöht werden.

Überladen wird der Methodenname, da er nun für verschiedene Methoden verwendet wird. Der Methodenname allein ist also mehrdeutig. Überladene Methoden müssen sich deshalb in der Liste ihrer formalen Parameter unterscheiden, um eindeutig identifizierbar zu sein. Mit anderen Worten: Die Signatur einer Methode muss eine Methode eindeutig identifizieren.

Die Signatur setzt sich **in Java** zusammen aus dem Methodennamen und der Parameterliste:

Signatur = Methodenname + Parameterliste

Der Rückgabetyp ist in Java **nicht** Bestandteil der Signatur!

Beachten Sie,

- dass es nicht möglich ist, in der gleichen Klasse zwei Methoden mit gleichem Methodennamen und gleicher Parameterliste – d. h. gleicher Signatur –, aber verschiedenen Rückgabetypen zu vereinbaren.
- dass, wenn keine exakte Übereinstimmung der aktuellen mit den formalen Parametern gefunden wird, vom Compiler versucht wird, die spezifischste Methode zu finden. Besser ist es jedoch stets, selbst für passende aktuelle Parameter zu sorgen, gegebenenfalls durch eine explizite Typkonvertierung.

Dass zwei Methoden mit identischer Signatur und verschiedenem Rückgabetyp in Java beim Überladen nicht zulässig sind, liegt daran, dass der Compiler keine Chance hat, die richtige Methode aufzurufen, wenn der Rückgabewert gar nicht abgeholt wird. Dass der Rückgabewert nicht abgeholt wird, ist zulässig.

Die Methode `static int parse (String var)` kann deshalb nicht in derselben Klasse wie die Methode `static float parse (String var)` vorkommen. Der Compiler könnte an dieser Stelle beispielsweise nicht unterscheiden, ob der Metho-

denaufruf `Klasse.parse ("7.7")` die Methode mit `float` als Rückgabetyp oder die Methode mit `int` als Rückgabetyp bezeichnet.

Beispiele

Als erstes Beispiel soll die in der Klasse `java.lang.Math` in überladener Weise definierte Methode `abs()` zur Ermittlung des Betrags eines arithmetischen Ausdrucks erwähnt werden. Die Methode `abs()` liefert den absoluten Wert im Format des jeweiligen Datentyps zurück. Die Methoden `abs()` sind wie folgt deklariert:

```
public static int    abs (int)
public static float  abs (float)
public static long   abs (long)
public static double abs (double)
```

Das nächste Beispiel zeigt eine Klasse `Parser`, die überladene Methoden mit unterschiedlichen Parameterlisten für das Umwandeln von Strings in `int`-Werte zur Verfügung stellt. Alle diese Methoden sind als Klassenmethoden realisiert, da sie auch ohne die Existenz eines Objektes zur Verfügung stehen sollen. Hier das bereits erwähnte Beispiel:

```
// Datei: Parser.java

public class Parser
{
   // Wandelt den String var in einen int-Wert.
   public static int parseInt (String var)
   {
      return Integer.parseInt (var);
   }

   // Wandelt den Stringanteil von der Position pos
   // bis zum Stringende in einen int-Wert.
   public static int parseInt (String var, int pos)
   {
      var = var.substring (pos);
      return Integer.parseInt (var);
   }

   // Wandelt den Stringanteil von der Position von bis
   // zur Position bis in einen int-Wert.
   public static int parseInt (String var, int von, int bis)
   {
      var = var.substring (von, bis);
      return Integer.parseInt (var);
   }
}

// Datei: TestParser.java

public class TestParser
{
   public static void main (String[] args)
   {
      String[] daten =
          {"Rainer Brang", "Hauptstr. 17", "73732 Esslingen", "25"};
```

```
        System.out.println ("Alter: " + Parser.parseInt (daten [3]));
        System.out.println ("Hausnummer: " +
                            Parser.parseInt (daten [1], 10));
        System.out.println ("Postleitzahl: " +
                            Parser.parseInt (daten [2], 0, 5));
    }
}
```

Die Ausgabe des Programms ist:

```
Alter: 25
Hausnummer: 17
Postleitzahl: 73732
```

9.5 Parameterliste variabler Länge

Methoden konnten in Java bis zu der Version Java 5 nur eine feste Anzahl von Parametern haben. Sollten früher in Java unterschiedlich viele Parameter an eine Methode übergeben werden, so gab es zwei verschiedene Wege:

- Für jede Parametervariante schrieb man eine überladene Methode
- oder man verpackte die zu übergebenden Werte in einem Array oder einer Collection[139] und übergab eine Referenz auf das Array bzw. die Collection als aktuellen Parameter an die Methode, die dadurch Zugriff auf die einzelnen Werte erhielt.

Seit der Version Java 5 ist dies nicht mehr notwendig, da Methoden eine Parameterliste mit variabler Länge – **varargs** genannt – besitzen können. In Anlehnung an die Ellipse in der Programmiersprache C, d. h. die **drei Punkte . . . am Ende der Parameterliste**, führte Java mit der Version Java 5 auch eine variable Parameterliste ein.

Eine **Parameterliste** kann sich in **zwei Teile** aufteilen: in einen Teil **fester Länge** mit einer festen Anzahl von Parametern und ferner einen Teil **variabler Länge**.

Die **variable Parameterliste** muss immer **am Ende der Parameterliste** stehen.

Um eine Parameterliste variabler Länge zu deklarieren, werden in Java drei Punkte . . . an den Datentyp des entsprechenden Parameters angefügt.

Dabei ist zu beachten, dass sich der variable Anteil nur auf einen einzigen spezifizierten Typ beschränkt und stets nach den explizit definierten Parametern der Parameterliste stehen muss.

[139] Collection siehe Kapitel 18

Hier ein **Beispiel** in **Pseudocode**:

```
public void myTestMethod (fester Anteil, variabler Anteil);
```

Dabei gilt:

`fester Anteil`	z. B.: `int a, String b`
`variabler Anteil`	z. B.: `int... c`

Und nun ein erstes **Programmbeispiel** mit einem festen und einem variablen Anteil in Java, wobei der variable Anteil vom Typ `String` ist:

```
// Datei: TestVarargs.java

public class TestVarargs
{
   public static void main (String[] args)
   {
      varPar (1, 2, 3, "Dies", "ist", "ein", "Test!");
   }

   public static void varPar (int a, int b, int c, String... str)
   {
      System.out.printf ("Erster Parameter: %d\n", a);
      System.out.printf ("Zweiter Parameter: %d\n", b);
      System.out.printf ("Dritter Parameter: %d\n", c);
      for (String element : str)
      {
         System.out.println ("Variabler Anteil: " + element);
      }
   }
}
```

Die Ausgabe des Programms ist:

```
Erster Parameter: 1
Zweiter Parameter: 2
Dritter Parameter: 3
Variabler Anteil: Dies
Variabler Anteil: ist
Variabler Anteil: ein
Variabler Anteil: Test!
```

Jetzt ein Beispiel mit einer variablen Liste von Objekten vom Typ `Object`:

```
// Datei: VarargsTest.java
public class VarargsTest
{

   public static void main (String[] args)
   {
      // Ein Beispiel mit 3 Parametern
      printAllObjects ("Jetzt folgen 2 Objekte vom Typ Object",
                       new Integer (10), new Double (2.0));
      // Ein Beispiel mit 4 Parametern
      printAllObjects ("Jetzt folgen 3 Objekte vom Typ Object",
```

```
                        new Integer (10),
                        new Integer (11),
                        new Double (3.0));
   }

   // Definition einer Methode mit einem festen Parameter und
   // einer beliebigen Anzahl von Parametern vom Typ Object.
   // Ein Leerzeichen nach dem Typ (hier Object) ist optional

   public static void printAllObjects (String text,
                                       Object... parameters)
   {
      // Text ausgeben
      System.out.println (text);
      // Parameter ausgeben - dabei wird automatisch die
      // toString()-Methode der Parameter aufgerufen.
      for (Object element : parameters)
      {
         System.out.println (element);
      }
   }
}
```

Die Ausgabe des Programms ist:

```
Jetzt folgen 2 Objekte vom Typ Object
10
2.0
Jetzt folgen 3 Objekte vom Typ Object
10
11
3.0
```

Die **variable Parameterliste** wird innerhalb der Methode als ein **Array des spezifizierten Typs** behandelt.

Der Aufruf einer Methode mit `varargs` bietet gegenüber dem Aufruf einer Methode mit einem Array als Parameter den Vorteil, dass die Übergabewerte direkt im Methodenaufruf angegeben werden können und nicht zuvor ein Array angelegt werden muss.

Generizität bei variablen Parameterlisten wird in Kapitel 17.6 besprochen.

9.6 Parameterübergabe beim Programmaufruf

In Java ist es möglich, Übergabeparameter an ein Programm zu übergeben. Diese Möglichkeit wird durch den Übergabeparameter `String[] args` bereitgestellt:

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

Die Array-Variable `args` ist eine Referenz auf ein Array von Referenzen, die auf die in der Kommandozeile übergebenen String-Objekte zeigen. Die Zahl der übergebenen Parameter kann dem Wert des Datenfeldes `args.length` entnommen werden.

C-Programmierer müssen berücksichtigen, dass sich an der ersten Position des String-**Arrays** `args` bereits **der erste Übergabeparameter** befindet.

Beispiele

Im folgenden **Beispiel** wird getestet, ob ein auf der Kommandozeile als Parameter mitgegebener String der Zeichenkette `"Java"` entspricht. Da die Inhalte der Strings mit der Methode `equals()` verglichen werden, ist der Vergleich `true`, wenn als Übergabe die Zeichenkette `"Java"` übergeben wird.

```
// Datei: StringTest.java

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

   {
      String a = "Java";
      String b = args [0];
      if (a.equals (b))
      {
         System.out.println ("Der String war Java");
      }
      else
      {
         System.out.println ("Der String war nicht Java");
      }
   }
}
```

Die Ausgabe des Programms ist:

```
java StringTest Java
```

Die Ausgabe des Programms ist:

```
Der String war Java
```

Im nächsten **Beispiel** werden Zahlen als Strings übergeben. Sie werden mit Hilfe der Klassenmethode `parseInt()` der Wrapper-Klasse `Integer` in einen `int`-Wert gewandelt. Die Integer-Zahlen werden dann addiert und das Ergebnis ausgegeben:

```
// Datei: AddInteger.java

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

```
      if (args.length != 2)
      {
         System.out.println ("FEHLER: Falsche Parameteranzahl");
         System.out.println ("Bitte zwei Parameter eingeben");
         System.out.println ("AddInteger <int1> <int2>");
      }
      else
      {
         int i1 = Integer.parseInt (args [0]);
         int i2 = Integer.parseInt (args [1]);
         System.out.println (args [0]+" + "+args [1]+" = "+(i1+i2));
      }
   }
}
```

Die Ausgabe des Programms ist:

```
java AddInteger 5 4
```

Die Ausgabe des Programms ist:

```
5 + 4 = 9
```

Weitere Möglichkeit zur Übergabe von Parametern

Eine weitere Möglichkeit, Parameter an eine Java-Applikation zu übergeben, besteht über die Option `-D` der Java-Virtuellen Maschine. Durch die Option -D können Parameter als Schlüssel-Wert-Paare – als sogenannte System-Eigenschaften (System-Properties) – von außen gesetzt werden. Der Vorteil dieses Vorgehens ist, dass die Parameter nicht nur in der Methode `main()` der Startklasse zur Verfügung stehen, sondern auch in allen anderen Klassen, die die System-Properties auslesen dürfen.

Das folgende Beispiel zeigt, wie System-Properties von außen gesetzt werden können und auf diese zugegriffen werden kann:

```
// Datei: SystemPropertiesTest.java

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

  public SystemPropertiesTest()
  {
    System.out.println("user.language="
        + System.getProperty("user.language"));

    System.out.println("my.parameter="
        + System.getProperty("my.parameter"));

  }
}
```

Die Ausgabe des Programms ist:

```
java -Duser.language=en -Dmy.parameter="Hello,
World" SystemPropertiesTest
```

Die Ausgabe des Programms ist:

```
user.language=en
my.parameter=Hello, World
```

Im Konstruktor der Klasse `SystemPropertiesTest` werden die System-Properties `user.language` und `my.parameter` ausgelesen. Wird die Option `-D` für `user.language` nicht angegeben, so wird trotzdem ein Wert für die System-Property `user.language` ausgeben. Das liegt daran, dass es sich bei dieser System-Property um eine Standard-Property der Java-Virtuellen Maschine handelt und von dieser, wenn sie nicht durch die Option `-D` von außen gesetzt wurde, implizit gesetzt wird. Bei `my.parameter` handelt es sich um keine Standard-Property. Wird für diese System-Property nicht die Option `-D` angegeben, so gibt das Beispiel für `my.parameter` den Wert `null` aus.

9.7 Iteration und Rekursion

Ein Algorithmus wird als **iterativ** bezeichnet, wenn bestimmte Abschnitte des Algorithmus innerhalb einer einzigen Ausführung des Algorithmus mehrfach durchlaufen werden.

Ein Algorithmus wird als **rekursiv**[140] bezeichnet, wenn er Abschnitte enthält, die sich selbst direkt oder indirekt aufrufen.

Iteration und Rekursion sind Prinzipien, die oftmals als Alternativen für die Programmkonstruktion erscheinen. Theoretisch sind Iteration und Rekursion äquivalent, weil man jede Iteration in eine Rekursion umformen kann und umgekehrt. In der Praxis gibt es allerdings oftmals den Fall, dass die iterative oder rekursive Lösung auf der Hand liegt, dass man aber auf die dazu alternative rekursive bzw. iterative Lösung nicht so leicht kommt.

Programmtechnisch läuft eine Iteration auf eine Schleife, eine **direkte Rekursion** auf den Aufruf einer Methode durch sich selbst hinaus. Es gibt aber auch eine indirekte Rekursion. Eine **indirekte Rekursion** liegt beispielsweise vor, wenn zwei Methoden sich wechselseitig aufrufen.

Das Prinzip der Iteration und der Rekursion soll an dem folgenden **Beispiel** der Berechnung der Fakultätsfunktion veranschaulicht werden.

[140] lateinisch recurrere = zurücklaufen

Iterative Berechnung der Fakultätsfunktion

Bei der iterativen Berechnung der Fakultätsfunktion geht man aus von der Definition der Fakultät

0! = 1
n! = 1 * 2 * ... * n für n > 0

und beginnt bei den kleinen Zahlen. Der Wert von 0! ist 1, der Wert von 1! ist 0! * 1, der Wert von 2! ist 1! * 2, der Wert von 3! ist 2! * 3, usw.

Nimmt man eine Schleifenvariable `i`, die von 1 bis n durchgezählt wird, so muss innerhalb der Schleife lediglich der Wert der Fakultät vom vorhergehenden Schleifendurchlauf mit dem aktuellen Wert der Schleifenvariablen multipliziert werden.

Das folgende Programm zeigt die iterative Berechnung der Fakultätsfunktion:

```
// Datei: IterativFaku.java
public class IterativFaku
{
   public static long berechneFakultaet (int n)
   {
      long faku = 1;
      for (int i = 1; i <= n; i++)
         faku = faku * i;
      return faku;
   }

   public static void main (String[] args)
   {
      long faku = berechneFakultaet (5);
      System.out.println ("5! = " + faku);
   }
}
```

Die Ausgabe des Programms ist:

```
5! = 120
```

Rekursive Berechnung der Fakultätsfunktion

Bei der rekursiven Berechnung der Fakultätsfunktion geht man ebenfalls von der Definition der Fakultät aus, beginnt aber nicht bei den kleinen Zahlen, sondern bei den großen Zahlen und läuft dann zu den kleinen Zahlen zurück. Dabei gilt:

n! = n * (n-1)! für n > 0
0! = 1

Im Gegensatz zur Iteration schaut man jetzt auf die Funktion f(n) und versucht, diese Funktion durch sich selbst – aber mit anderen Aufrufparametern – darzustellen. Die mathematische Analyse ist hier ziemlich leicht, denn man sieht sofort, dass

f(n) = n * f (n-1)

ist. Damit hat man das Rekursionsprinzip bereits gefunden. Dies ist jedoch nur die eine Seite der Medaille, denn die Rekursion darf nicht ewig gehen! Das **Abbruchkriterium** wurde bereits oben erwähnt. Es heißt:

0! = 1

Durch n! = n * (n-1)! lässt sich also die Funktion f(n) auf sich selbst zurückführen, d. h. f(n) = n * f(n-1). f(n-1) ergibt sich wiederum durch f(n-1) = (n-1) * f(n-2). Nach diesem Algorithmus geht es jetzt solange weiter, bis das Abbruchkriterium erreicht ist. Das Abbruchkriterium ist bei 0! erreicht, da 0! nicht auf (-1)! zurückgeführt werden kann, sondern per Definition gleich 1 ist.

Dieser Algorithmus lässt sich leicht programmieren. Die Methode `berechnefakultaet()` enthält zwei Zweige:

- Der eine Zweig wird angesprungen, wenn die **Abbruchbedingung nicht erfüllt ist**. Hier ruft die Methode sich selbst wieder auf. Hierbei ist zu beachten, dass die Anweisung, welche die Methode aufruft, gar nicht abgearbeitet werden kann, solange die aufgerufene Methode kein Ergebnis zurückliefert.
- Der andere Zweig wird angesprungen, wenn die **Abbruchbedingung erfüllt** ist. In diesem Fall liefert die Methode zum ersten Mal einen Rückgabewert.

Rekursive Berechnung der Fakultätsfunktion als Programm

```
// Datei: RekursivFaku.java

public class RekursivFaku
{
   public static long berechneFakultaet (int n)
   {
      System.out.println ("Aufruf mit: " + n);
      if (n >= 1)  // Abbruchbedingung noch nicht erreicht
         return n * berechneFakultaet (n - 1);
      else // Abbruchbedingung erfüllt, d. h. n ist gleich 0.
         return 1;
   }

   public static void main (String[] args)
   {
      int n = 5;
      long z = berechneFakultaet (n);
      System.out.println ("5! = " + z);
   }
}
```

Die Ausgabe des Programms ist:

```
Aufruf mit: 5
Aufruf mit: 4
Aufruf mit: 3
Aufruf mit: 2
Aufruf mit: 1
Aufruf mit: 0
5! = 120
```

Das folgende Bild veranschaulicht die Berechnung der Fakultät für `n = 3`:

Bild 9-4 Verfolgung der rekursiven Aufrufe für `berechneFakultaet (3)`

Das Bild 9-5 zeigt den **Aufbau des Stacks** durch den rekursiven Aufruf der Methode `berechneFakultaet(3)`, bis das **Abbruchkriterium** erreicht ist. Das Abbruchkriterium liegt dann vor, wenn `berechneFakultaet()` mit `n = 0` aufgerufen wird.

Danach beginnt durch die Beendigung aller wartenden `berechneFakultaet()`-Methoden der **Abbau des Stacks**. Der Abbau des Stacks wird in Bild 9-6 gezeigt.

Aus Platzgründen wurde dort der Methodenaufruf `berechneFakultaet()` mit `faku()` abgekürzt. Das folgende Bild zeigt den Aufbau des Stacks für `faku(3)`:

Bei jedem Aufruf von `faku()` werden die Rücksprungadresse und weitere Verwaltungsinformationen auf einem Stack abgelegt, der durch die virtuelle Maschine verwaltet wird. Auch die übergebenen Parameter (hier nur einer) werden auf diesem Stack abgelegt. Dabei wächst der Stack mit der Rekursionstiefe der Funktion.

Der letzte Aufruf von `faku()` mit dem Parameter `n = 0` bewirkt keine weitere Rekursion, da ja die Abbruchbedingung erfüllt ist.

Bild 9-5 Aufbau des Stacks für `faku (3)`

Der Abbau des Stacks für `faku(3)` geschieht in umgekehrter Reihenfolge. Dies wird im folgenden Bild 9-6 gezeigt:

Beim Beenden der aufgerufenen Methode werden auf dem Stack die lokalen Variablen (übergebene Parameter) freigegeben und die Rücksprungadresse und sonstigen Verwaltungsinformationen abgeholt. Der Rückgabewert wird in diesem Beispiel über ein Register an die aufrufenden Methoden zurückgegeben. Der Rückgabewert kann auf verschiedene Weise an die aufrufende Methode zurückgegeben werden, beispielsweise auch über den Stack. Dies ist vom Compiler abhängig.

Bild 9-6 Abbau des Stacks für `faku (3)`

9.8 Zusammenfassung

Ein Block ist eine Folge von Anweisungen, die sequenziell hintereinander ausgeführt wird. Eine Methode ist eine Folge von Anweisungen, die unter einem Namen aufgerufen werden kann.

Einen Block benötigt man aus zwei Gründen:

- zum einen ist der Rumpf einer Methode ein Block,
- zum anderen gilt ein Block syntaktisch als eine einzige Anweisung. Daher kann ein Block auch da stehen, wo von der Syntax her nur eine einzige Anweisung zugelassen ist, wie z. B. im `if`- oder `else`-Zweig einer `if`-`else`-Anweisung.

Blöcke und ihre Besonderheiten werden in Kapitel **9.1** besprochen. Dies umfasst den Begriff der Deklarationsanweisung, den leeren Block, lokale Variablen und die Schachtelung von Blöcken ebenso wie die Definition von Sichtbarkeit, Gültigkeit und Lebensdauer.

Durch das Konzept der Deklarationsanweisung können in Java an einer beliebigen Stelle innerhalb eines Blockes Deklarationsanweisungen notiert werden.

Zur defensiven Programmierung gehört es, dass bei Selektionen und Iterationen sowie auch bei leeren Anweisungen stets ein Block (ggf. ein leerer Block) verwendet wird. Damit sind Erweiterungen um Anweisungen nicht fehlerträchtig, da die Klammern schon da sind.

Ein Block zählt syntaktisch als eine einzige Anweisung. Im Gegensatz zu einer normalen Anweisung besteht bei einem Block jedoch die Möglichkeit, Block-lokale Variablen einzuführen.

Bei geschachtelten Blöcken können in Java in jedem Block – auch in inneren Blöcken – Variablen definiert werden. In einem inneren Block definierte Variablen sind nur innerhalb dieses Blockes sichtbar, in einem umfassenden Block sind sie unsichtbar. Variablen, die in einem umfassenden Block definiert sind, sind für einen folgenden inneren Block auch sichtbar. Bei Java sind identische Namen im inneren und äußeren Block nicht zugelassen. Es resultiert ein Kompilierfehler.

Die Bereiche der Sichtbarkeit, Gültigkeit und Lebensdauer werden für lokale Variablen, Instanzvariablen und Klassenvariablen betrachtet. Die Lebensdauer ist die Zeitspanne, in der die Java-Virtuelle Maschine einer Variablen einen Platz im Arbeitsspeicher zur Verfügung stellt. Mit anderen Worten, während ihrer Lebensdauer besitzt eine Variable einen Speicherplatz. Die Gültigkeit einer Variablen bedeutet, dass an einer Programmstelle der Name einer Variablen dem Compiler durch eine Vereinbarung bekannt ist. Die Sichtbarkeit einer Variablen bedeutet, dass man von einer Programmstelle aus die Variable sieht, das heißt, dass man auf sie über ihren Namen zugreifen kann.

Bei lokalen Variablen fallen Gültigkeit und Sichtbarkeit zusammen. Bei Datenfeldern muss man prinzipiell zwischen Gültigkeit und Sichtbarkeit unterscheiden.

Kapitel **9.2** befasst sich mit der Methodendefinition und dem Methodenaufruf und behandelt insbesondere formale und aktuelle Parameter sowie die `return`-Anweisung. In Java werden die aktuellen Parameter von links nach rechts bewertet.

Methoden stellen Anweisungsfolgen dar, die unter einem Namen aufgerufen werden können. Methoden werden stets für Objekte – im Falle von Instanzmethoden – bzw. Klassen – im Falle von Klassenmethoden – aufgerufen. Die Methodendeklaration einer Methode beinhaltet den Rückgabetyp sowie den Namen der Methode und ein Paar runder Klammern, die leer sein können oder formale Parameter enthalten können. Anstelle eines Rückgabetyps kann auch das Schlüsselwort `void` stehen. Wird das Schlüsselwort `void` statt eines Rückgabetyps angegeben, so ist keine `return`-Anweisung notwendig. Es kann aber jederzeit mit `return` die Abarbeitung der Methode abgebrochen werden. Damit wird ein sofortiger Rücksprung zur Aufrufstelle bewirkt. In diesem Fall darf mit der `return`-Anweisung kein Wert zurückgegeben werden. Eine Methode kann mit `return` nur einen einzigen Wert zurückgeben. Möchte man mehrere Werte zurückgeben, so kann dies über Referenzen auf Objekte in der Parameterliste gehen oder aber über die Schaffung eines Objektes mit mehreren Datenfeldern, auf das mit `return` eine Referenz zurückgegeben wird. An der aufrufenden Stelle darf der Wert, den eine Methode liefert, ignoriert werden. Mit anderen Worten, man kann eine Methode, die einen Rückgabewert hat, einfach aufrufen, ohne den Rückgabewert abzuholen.

Mit den formalen Parametern wird festgelegt, wie viele Übergabeparameter existieren, von welchem Typ diese sind und welche Reihenfolge sie haben. Beim Aufruf einer Methode mit Parametern finden Zuweisungen statt. Der Wert eines aktuellen Parameters wird dem entsprechenden formalen Parameter zugewiesen. Eine solche Aufrufschnittstelle wird als call-by-value-Schnittstelle bezeichnet. Ein formaler Parameter stellt eine spezielle lokale Variable dar. Dies hat zur Konsequenz, dass eine im Methodenrumpf definierte lokale Variable nicht gleich heißen darf wie ein formaler Parameter. Ein aktueller Parameter muss keine Variable sein. Ein aktueller Parameter ist irgendein Ausdruck eines passenden Typs, den der Aufrufer an den formalen Parameter übergibt.

Generell finden bei Übergabeparametern genauso wie bei Rückgabewerten Kopiervorgänge statt. Unabhängig davon, ob es sich um einfache Datentypen oder Referenzen handelt, werden Werte kopiert. Bei einfachen Datentypen als Übergabeparameter wirken sich Änderungen am Wert des formalen Parameters – genauso wie bei Referenztypen – nur auf die Kopie, d. h. den formalen Parameter, aus. Es gibt keinerlei Rückwirkungen auf das Original, d. h. den aktuellen Parameter. Werden Referenzen übergeben, so referenziert der formale Parameter dasselbe Objekt wie der aktuelle Parameter. Eine Operation auf dem formalen Referenzparameter erfolgt auf dem Objekt, auf das die Referenz zeigt, in anderen Worten auf dem referenzierten Objekt. Sowohl einfache Datentypen (`int`, `char`, ...) als auch Referenzen werden "by value" übergeben. Da Referenzen aber auf Objekte zeigen, wird quasi ein "call-by-reference" simuliert. Die Auswertung der aktuellen Parameter in der Parameterliste erfolgt von links nach rechts.

Kapitel **9.3** befasst sich mit der Polymorphie von Operationen.

Eine Klasse stellt einen Namensraum dar. Damit ist es möglich, dass verschiedene nicht verwandte Klassen die gleiche Operation implementieren, d. h. den gleichen

Methodenkopf und die gleiche Semantik der Operationen haben. Je nach Klasse kann eine Operation in verschiedenen Implementierungen – sprich in verschiedener Gestalt – auftreten. Man spricht hierbei auch von der Vielgestaltigkeit (Polymorphie) von Operationen. Jedes Objekt trägt die Typinformation, von welcher Klasse es ist, immer bei sich. Das heißt, dass ein Objekt immer weiß, zu welcher Klasse es gehört. Da ein Methodenaufruf immer an ein Objekt (im Falle von Instanzmethoden) bzw. an die Klasse (im Falle von Klassenmethoden) gebunden ist, und ein Objekt weiß, zu welcher Klasse es gehört, ist immer eine eindeutige Zuordnung eines Methodenaufrufs zu seiner Klasse möglich.

Kapitel **9.4** thematisiert das Überladen von Methoden.

Oftmals verrichten verschiedene Methoden dieselbe Aufgabe, allerdings für verschiedene Datentypen der Übergabeparameter. Erlaubt eine Sprache das Überladen von Methoden (engl. overloading), so können Methoden mit verschiedenen Parameterlisten denselben Namen tragen. Der Aufruf der richtigen Methode ist Aufgabe des Compilers. Die Signatur einer Methode setzt sich in Java zusammen aus dem Methodennamen und der Parameterliste: Signatur = Methodenname + Parameterliste. Der Rückgabetyp ist in Java nicht Bestandteil der Signatur! Beachten Sie,

- dass es nicht möglich ist, in der gleichen Klasse zwei Methoden mit gleichem Methodennamen und gleicher Parameterliste – d. h. gleicher Signatur – aber verschiedenen Rückgabetypen zu vereinbaren.
- dass, wenn keine exakte Übereinstimmung der aktuellen mit den formalen Parametern gefunden wird, vom Compiler versucht wird, die spezifischste Methode zu finden. Besser ist es jedoch stets, selbst für passende aktuelle Parameter zu sorgen, gegebenenfalls durch eine explizite Typkonvertierung.

Kapitel **9.5** behandelt Parameterlisten variabler Länge.

Um eine Parameterliste variabler Länge zu deklarieren, werden in Java drei Punkte `...` an den Datentyp des entsprechenden Parameters angefügt. Dabei ist zu beachten, dass sich der variable Anteil nur auf einen einzigen spezifizierten Typ beschränkt und stets nach den explizit definierten Parametern der Parameterliste stehen muss. Die variable Parameterliste wird innerhalb der Methode als ein Array des spezifizierten Typs behandelt.

Kapitel **9.6** studiert die Parameterübergabe beim Programmaufruf.

Kapitel **9.7** diskutiert die Verfahren der Iteration und Rekursion. Ein Algorithmus heißt iterativ, wenn bestimmte Abschnitte des Algorithmus innerhalb einer einzigen Ausführung des Algorithmus mehrfach durchlaufen werden. Er heißt rekursiv[141], wenn er Abschnitte enthält, die sich selbst direkt oder indirekt aufrufen.

[141] lateinisch recurrere = zurücklaufen

9.9 Übungen

Aufgabe 9.1: Blöcke

9.1.1 Sichtbarkeit von Variablen

Analysieren Sie das folgende Programm. Was erwarten Sie als Ausgabe?

```
// Datei: SichtbarAufg.java
public class SichtbarAufg
{
   private int wert = 7;

   public int zugriff()
   {
      int wert = 77;
      return wert;
   }

   public static void main (String[] args)
   {
      SichtbarAufg sich = new SichtbarAufg();
      System.out.println (sich.zugriff());
   }
}
```

9.1.2 Gültigkeit von Variablen

Analysieren Sie das folgende Programm. Was erwarten Sie als Ausgabe?

```
// Datei: GueltigkeitAufg.java
public class GueltigkeitAufg
{
   private int wert = 7;

   public int zugriff()
   {
      int tmp = wert;
      int wert = 77;
      return tmp;
   }

   public static void main (String[] args)
   {
      GueltigkeitAufg guelt = new GueltigkeitAufg();
      System.out.println (guelt.zugriff());
   }
}
```

9.1.3 Sichtbarkeit und Verdecken von Instanzvariablen

Ausgangspunkt ist das Programm aus Aufgabe 9.1.1. Wie muss die Methode `zugriff()` verändert werden, damit sie nicht den Wert der lokalen Variablen, sondern der Instanzvariablen zurückgibt? Die lokale Variable soll nicht umbenannt oder entfernt werden! Ergänzen Sie das Programm.

Aufgabe 9.2: Polymorphie von Operationen

9.2.1 Berechnung von Flächeninhalten

Die Flächeninhalte von Quadraten und Kreisen werden unterschiedlich berechnet. Ergänzen Sie das folgende Programmfragment um die polymorphe Operation "berechne Flächeninhalt", sodass der jeweils korrekte Flächeninhalt ermittelt wird. Fehlende Stellen sind durch . . . . . markiert.

```
// Datei: PolymorpheOperation.java

public class PolymorpheOperation
{
   public static void main (String[] args)
   {
      Quadrat quad = new Quadrat (5.0);
      Kreis kreis = new Kreis (3.0);

      //Flaecheninhalt des Quadrats
      System.out.println ("Flaecheninhalt des Quadrats: " +
                          . . . . .);

      //Flaecheninhalt des Kreises
      System.out.println ("Flaecheninhalt des Kreises: " +
                          . . . . .);
   }
}

class Quadrat
{
   private double seitenlaenge;

   public Quadrat(double seitenlaenge)
   {
      this.seitenlaenge = seitenlaenge;
   }
   . . . . .
}

class Kreis
{
   private double pi;
   private double radius;

   public Kreis(double radius)
   {
      pi = 3.14;
      this.radius = radius;
   }
   . . . . .
}
```

9.2.2 Figurgröße skalieren

Ausgangspunkt ist das Programm aus Aufgabe 9.2.1. Erweitern Sie das Programm um die polymorphe Operation "skaliere Figur". Der Operation wird ein Skalierungsfaktor übergeben, mit dem die Größe des Kreises bzw. des Quadrats skaliert wird.

Aufgabe 9.3: Überladen von Methoden

9.3.1 Überladene Methode zur Summenberechnung

Das folgende Programmfragment stellt einen minimalistischen Taschenrechner dar. Überladen Sie die Methode zur Summenberechnung, damit auch Kommazahlen addiert werden können. Fehlende Stellen sind durch `. . . . .` markiert.

```
// Datei: UeberladeneMethoden.java

public class UeberladeneMethoden
{
   public static void main (String[] args)
   {
      int intSumme = 0;
      int a = 1;
      int b = 2;
      double doubleSumme = 0.0;
      double c = 1.5;
      double d = 0.25;

      Taschenrechner tr = new Taschenrechner();
      intSumme = tr.addiere (a, b);
      System.out.println (a + " + " + b + " = " +
                          intSumme);

      doubleSumme = . . . . .
      System.out.println (c + " + " + d + " = " +
                          doubleSumme);
   }
}

class Taschenrechner
{
   public int addiere (int a, int b)
   {
      return a + b;
   }
   . . . . .
}
```

9.3.2 Methoden zur Produktberechnung

Ausgangspunkt ist das Programm aus Aufgabe 9.3.1. Ergänzen Sie den Taschenrechner um eine Methode zum Multiplizieren zweier ganzer Zahlen. Überladen Sie die Methode, sodass auch Kommazahlen miteinander multipliziert werden können.

Aufgabe 9.4: Variable Länge einer Parameterliste

9.4.1 Variable Anzahl von Summanden

Erstellen Sie eine Klasse mit einer Methode, die eine variable Parameterliste vom Typ `int` und einen Parameter vom Typ `String` hat. Die Methode soll zuerst das `String`-Objekt auf der Kommandozeile ausgeben. Anschließend werden die Parameter der variablen Parameterliste addiert und das Ergebnis wird ebenfalls auf der Kommandozeile ausgegeben.

Schreiben Sie auch eine entsprechende `main()`-Methode, um Ihre Methode zu testen.

9.4.2 Ausgabe von Strings

Das folgende Programm enthält eine Methode `printObjects()`, der mittels einer variablen Parameterliste beliebig viele Objekte übergeben werden können. Anschließend gibt diese Methode eine String-Repräsentation der übergebenen Objekte auf der Standardausgabe bzw. Standardfehlerausgabe aus. Dabei hat sich ein Fehler eingeschlichen. Finden und korrigieren sie den Fehler.

```
// Datei: VariableParameterlisteTest2.java

public class VariableParameterlisteTest2
{
   private void printObjects (Object... objects,
                              boolean useStdErr)
   {
      if (useStdErr == true)
      {
         for (Object o : objects)
         {
            System.err.println (o.toString());
         }
      }
      else
      {
         for (Object o : objects)
         {
            System.out.println (o.toString());
         }
      }
   }

   public static void main (String[] args)
   {
      VariableParameterlisteTest2 vpt =
                           new VariableParameterlisteTest2();
      vpt.printObjects (new Integer (1), new Double (3.5),
                        "Hello", "World!", false);
   }
}
```

Aufgabe 9.5: Parameterübergabe beim Programmaufruf

9.5.1 Text parametergesteuert ausgeben

Entwickeln Sie ein einfaches Java-Programm, das beim Programmaufruf einen Parameter vom Typ `String` entgegennimmt, der eine ganze Zahl darstellt. Durch diese Zahl wird festgelegt, wie oft beispielsweise eine Zeile wie "Hallo, Welt!" auf dem Bildschirm ausgegeben wird. Der übergebene String kann, wie im Beispiel `AddInteger` in Kapitel 9.6 gezeigt, mit Hilfe der Methode `parseInt()` der Wrapper-Klasse `Integer` in einen `int`-Wert gewandelt werden.

Ein beispielhafter Programmaufruf könnte folgendermaßen aussehen:

```
java MassenGruss 4
Hallo, Welt!
Hallo, Welt!
Hallo, Welt!
Hallo, Welt!
```

9.5.2 Einfacher Taschenrechner mit Parameterübergabe

Entwickeln Sie einen einfachen Rechner, der die vier Grundrechenarten (+, -, *, /) beherrscht. Dieser Rechner soll durch Parameter beim Programmaufruf gesteuert werden. Hierzu werden dem Programm beim Aufruf zwei Zahlen als Strings sowie ein Schlüsselwort für die durchzuführende Operation übergeben. Diese Zahlen können, wie im Beispiel `AddInteger` in Kapitel 9.6 gezeigt, mit Hilfe der Methode `parseInt()` der Wrapper-Klasse `Integer` in einen `int`-Wert gewandelt werden.

Die Reihenfolge der Parameter ist folgendermaßen definiert:

```
[Zahl1] [Operation] [Zahl2]
```

Verwenden Sie die folgenden Schlüsselworte für die Rechenoperationen:

`add` Addition: Zahl1 + Zahl2
`sub` Subtraktion: Zahl1 – Zahl2
`mul` Multiplikation: Zahl1 * Zahl2
`div` Division: Zahl1 / Zahl2

Ein Aufruf des Programms könnte beispielsweise so aussehen:

```
java Rechner 13 add 7
```

und würde zu folgendem Ergebnis führen:

```
13 add 7 ist 20
```

Aufgabe 9.6: Iteration und Rekursion

9.6.1 Analyse eines rekursiven Algorithmus

Analysieren Sie das folgende Programm. Was wird hier berechnet? Ist Ihnen ein alternativer (nicht rekursiver) Lösungsweg bekannt?

```
// Datei: Rekursion.java
public class Rekursion
{
   public int rekursAufruf (int n)
   {
      if (n > 1)
         return n + rekursAufruf (n - 1);
      return 1;
   }

   public static void main (String[] args)
   {
      Rekursion rek = new Rekursion();
      System.out.println (rek.rekursAufruf (50));
   }
}
```

9.6.2 Analyse eines iterativen Algorithmus

Welche mathematische Formel berechnet das Programm? Wie lautet das Ergebnis?

```
// Datei: Iteration.java

public class Iteration
{
   public int iterativAufruf (int n)
   {
      int wert = 1;

      for (int i = 2; i <= n; i++)
      {
         wert *= i;
      }
      return wert;
   }

   public static void main (String[] args)
   {
      Iteration it = new Iteration();
      System.out.println (it.iterativAufruf (4));
   }
}
```

9.6.3 Berechnung der Potenz a^n

a) Berechnen Sie mit Hilfe einer Rekursion die Potenz a^n. Die Variablen `a` und `n` sollen sich auf Werte größer 0 beschränken und von der Tastatur eingelesen werden. Vervollständigen Sie hierzu im folgenden Programm die mit `. . . . .` gekennzeichneten Stellen:

```
// Datei: PotenzRekursiv.java

public class PotenzRekursiv
{
   public static . . . . . berechnePotenz (. . . . .)
   {
      . . . . .
   }

   public static void main (String[] args)
   {
      java.util.Scanner scanner =
                          new java.util.Scanner (System.in);
      try
      {
         System.out.println
                     ("Gib einen Wert >0 für a ein: ");
         int a = scanner.nextInt();

         System.out.println
                     ("Gib einen Wert >0 für n ein: ");
         int n = scanner.nextInt();

         int ergebnis = berechnePotenz (a, n);
```

```
            System.out.println ("Das Ergebnis ist: " +
                                ergebnis);
        }
        catch (Exception ex)
        {
            System.out.println (ex.toString());
        }
    }
}
```

b) Erweitern Sie die Methode `berechnePotenz()`, sodass sie auch mit den Werten `a = 0` und `n = 0` zurechtkommt. Negative Werte werden weiterhin nicht betrachtet. Beachten Sie, dass eine Potenz für den Exponenten 0 immer 1 liefert. Ausnahme: Falls `a` den Wert 0 hat, ist das Ergebnis immer 0.

c) Schreiben Sie eine neue Klasse `PotenzIterativ`. Diese Klasse implementiert die Lösung aus Aufgabe 9.6.3 b) mit Hilfe einer Iteration. Verwenden Sie dazu eine `for`-Schleife.

Kapitel 10

Klassen und Objekte

10.1 Information Hiding
10.2 Klassenvariablen und Klassenmethoden
10.3 Die `this`-Referenz
10.4 Initialisierung von Datenfeldern
10.5 Instanziierung von Klassen
10.6 Freigabe von Speicher
10.7 Die Klasse `Object`
10.8 Zusammenfassung
10.9 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_10

10 Klassen und Objekte

In Java kann man nur objektorientiert programmieren. Die Klassen und ihre statischen Beziehungen stellen das Skelett eines Programms dar. Die Muskeln, die das Skelett zum Leben und in Bewegung bringen, sind die Methoden, welche die Dynamik, d. h. die Verarbeitung von Daten, beschreiben.

Alle Klassen in Java haben die Klasse `java.lang.Object` als Urvater. Klassen sind vom Programmschreiber entworfene und implementierte Datentypen oder von der Java-Plattform bereitgestellte Bibliotheksklassen. Klassen enthalten Daten und die auf diesen Daten möglichen Operationen in Form von Methoden.

Wenn eine Klasse nicht explizit von einer anderen Klasse abgeleitet wird, so ist die Klasse `java.lang.Object` automatisch ihre Vaterklasse.

Methoden repräsentieren die Schnittstellen eines Objektes bzw. einer Klasse nach außen. Objekte sind Variablen, die nach dem Bauplan der Klasse gebaut sind. **Klassenbezogene Datenfelder** stellen benannte Speicherstellen in einer Klasse, die **Klassenvariablen**, dar. **Objektbezogene Datenfelder** werden in Form von **Instanzvariablen** in jedem einzelnen Objekt individuell angelegt.

Kapitel 10.1 erläutert die Bedeutung des Information Hiding. Anschließend folgt in Kapitel 10.2 die Definition von Klassenvariablen und Klassenmethoden. Kapitel 10.3 erläutert die `this`-Referenz in Java und Kapitel 10.4 die Initialisierung von Datenfeldern. Wie die Instanziierung von Klassen vor sich geht, wird in Kapitel 10.5 betrachtet. Die Freigabe von Speicher folgt in Kapitel 10.6. Zum Schluss wird in Kapitel 10.7 auf die Klasse `Object` eingegangen.

10.1 Information Hiding

Ein Ziel der objektorientierten Programmierung ist es, die Daten und die Implementierung der Methoden eines Objekts zu verbergen.[142] Das bedeutet, dass das Prinzip des Information Hiding angewandt werden soll. Es soll kein Unbefugter die Daten eines Objekts verändern können. Nur die eigenen Methoden eines Objektes sollen auf die Daten des entsprechenden Objektes Zugriff haben. Dies bedeutet, dass das folgende Beispiel zwar syntaktisch korrekt ist, jedoch diesen Zielvorstellungen widerspricht:

```
// Datei: Person.java
public class Person
{
   public String name;     // Der Zugriffsmodifikator public erlaubt
   public String vorname; // einen Zugriff auf die Daten und ermög-
```

[142] Java erfüllt dieses Ziel nur zum Teil. **In Java ist der Zugriffsschutz nur klassenbezogen realisiert**. Dies bedeutet, dass es nur Objekten fremder Klassen verwehrt werden kann, auf ein anderes Objekt zuzugreifen. Ein Objekt einer bestimmten Klasse kann in Java stets auf die Methoden und Datenfelder eines Objekts derselben Klasse zugreifen!

```
   public int alter;       // licht damit eine unzulässige Manipula-
                           // tion von "außen"!
   public void print()
   {
      System.out.println ("Name    : " + name);
      System.out.println ("Vorname : " + vorname);
      System.out.println ("Alter   : " + alter);
   }
}

// Datei: TestPerson.java
public class TestPerson
{
   public static void main (String[] args)
   {
      Person p = new Person();

      // Die Daten der Klasse Person sind nicht geschützt, es kann
      // auf sie problemlos aus einer anderen Klasse heraus zuge-
      // griffen werden!
      p.name = "Müller";
      p.vorname = "Fritz";
      p.alter = 35;
      p.print();
   }
}
```

Die Ausgabe des Programms ist:

```
Name:    Müller
Vorname: Fritz
Alter:   35
```

Aus Gründen des Software Engineering sollte es in Java **keinen direkten Zugriff auf die Daten eines Objektes von anderen Klassen bzw. von Objekten anderer Klassen aus** geben. Direkter Zugriff bedeutet, dass über eine Referenz auf ein Objekt direkt auf ein Datenfeld zugegriffen werden kann.

Der Zugriffsschutz ist in Java klassenbezogen realisiert. Auf die privaten Eigenschaften eines Objekts einer Klasse kann von einem Objekt derselben Klasse aus zugegriffen werden. Nur fremden Klassen kann der Zugriff verwehrt werden.

Das **Verbergen von Daten** erfolgt mit Hilfe des **Schlüsselworts `private`** wie in folgendem **Beispiel**:

```
// Datei: Person.java
public class Person
{
   private String name;
   private String vorname;
   private int alter;
```

```
   public void print()
   {
      System.out.println ("Name    : " + name);
      System.out.println ("Vorname : " + vorname);
      System.out.println ("Alter   : " + alter);
   }

   // Es folgen die anderen Methoden
   . . . . .
}
```

Damit hat man in diesem Beispiel von der Methode `main()` der Klasse `TestPerson` aus keinen direkten Zugriff mehr auf die Daten eines Objektes der Klasse `Person`, sondern nur noch über die Methode `print()` der Klasse `Person`.

Daten zu schützen, ist die generelle Vorgehensweise.

Man erlaubt klassenfremden Methoden in der Regel nicht den Zugriff auf die Daten einer Klasse. Dies ist ausschließlich Aufgabe der klasseneigenen Methoden.

Damit ist auch bei einer fehlerhaften Datenbearbeitung automatisch die Fehlersuche auf die Methoden, die innerhalb der betrachteten Klasse liegen, beschränkt.

Das hier vorgestellte Schlüsselwort `private` ist ein Zugriffsmodifikator, der den Zugriffsschutz regelt. Kapitel 12.7 behandelt auch die Zugriffsmodifikatoren `public` und `protected` und den Fall, dass gar kein Zugriffsmodifikator angegeben wird.

Hier nun ein **Beispiel** für den **Zugriff eines Objekts auf die privaten Daten eines anderen Objekts derselben Klasse**:

```
// Datei: Person2.java

public class Person2
{
   private float geld;

   Person2 (float geld)
   {
      this.geld = geld;
   }

   float getGeld (Person2 p)
   {
      return p.geld;
   }
}

// Datei: TestPerson2.java
public class TestPerson2
{
   public static void main (String[] args)
   {
```

```
        System.out.print ("Das eigene Geld der ersten Person ist: ");
        Person2 p1 = new Person2 (10);
        System.out.println (p1.getGeld (p1));

        System.out.print ("Die erste Person schaut in "
           + "den Geldbeutel der zweiten Person. Die zweite "
           + "Person hat:  ");
        Person2 p2 = new Person2 (20);
        System.out.println (p1.getGeld (p2));
    }
}
```

Die Ausgabe des Programms ist:

```
Das eigene Geld der ersten Person ist: 10.0
Die erste Person schaut in den Geldbeutel der
zweiten Person. Die zweite Person hat: 20.0
```

10.2 Klassenvariablen und Klassenmethoden

Klassenvariablen und Klassenmethoden wurden bereits in Kapitel 2.3 vorgestellt. Klassenvariablen und Klassenmethoden werden in Java mit Hilfe des Schlüsselwortes **`static`** deklariert. Die folgenden Unterkapitel zeigen Beispiele, in denen Klassenvariablen bzw. Klassenmethoden Anwendung finden.

10.2.1 Klassenvariablen

Klassenvariablen, die für alle Objekte einer Klasse als globale Daten zur Verfügung stehen, werden mit Hilfe des **Schlüsselwortes `static`** deklariert.

Beispiel für Klassenvariablen

Das folgende **Programm** behandelt die Schüler einer Schulklasse als Objekte. Die Schülerzahl ist keine Eigenschaft, die einem individuellen Objekt zugeordnet werden kann. Sie ist eine Eigenschaft des Verbunds aus allen Schüler-Individuen. Sie wird daher als Eigenschaft der gesamten Schulklasse betrachtet und damit als Klassenvariable definiert. Hier das bereits zitierte Beispiel:

Klassenvariablen sind nicht Teil von Objekten und werden deshalb nicht auf dem Heap geführt. Klassenvariablen werden bei der Klasse in der sogenannten **Method Area** geführt und sind deshalb für alle Objekte einer Klasse nur ein einziges Mal vorhanden.

Der **Zugriff auf Klassenvariablen** ist ohne die Existenz einer Instanz einer Klasse möglich. Der Zugriff erfolgt dann **über den Klassennamen** wie z. B. `Schueler.klassenStaerke`.

Innerhalb der eigenen Klasse kann der Zugriff auch direkt über den Namen der Klassenvariablen erfolgen, hier also über `klassenStaerke`.

Der **Zugriff auf eine Klassenvariable** kann auch **über eine Referenz auf ein Objekt** der entsprechenden Klasse erfolgen wie z. B. durch Aufruf von `schuelerInKlasse[0].klassenStaerke`, da ein Objekt stets weiß, zu welcher Klasse es gehört.

```
// Datei: Schueler.java

public class Schueler
{
   private int nummerDesSchuelers;
   public static int klassenStaerke = 0;

   public void setzeNummer()
   {
      nummerDesSchuelers = ++klassenStaerke;
   }

   public void abzaehlen()
   {
      System.out.println ("Ich bin die Nr.: " + nummerDesSchuelers);
   }
}

// Datei: SchuelerTest.java

public class SchuelerTest
{
   public static void main (String[] args)
   {
      System.out.println ("Klassenstärke vor der Einschulung: "
                          + Schueler.klassenStaerke);
      // Erzeugung eines Arrays für Schüler
      Schueler[] schuelerInKlasse = new Schueler [10];

      for (int lv = 0; lv < schuelerInKlasse.length; lv++)
      {
         schuelerInKlasse [lv] = new Schueler();
         schuelerInKlasse [lv].setzeNummer();
      }

      // Ausgabe der Schüler
      for (Schueler element : schuelerInKlasse)
      {
         element.abzaehlen();
      }

      System.out.println ("Klassenstärke nach der Einschulung: "
                          + schuelerInKlasse [0].klassenStaerke);
   }
}
```

Die Ausgabe des Programms ist:

```
Klassenstärke vor der Einschulung: 0
Ich bin die Nr.: 1
   . . . . .
Ich bin die Nr.: 10
Klassenstärke nach der Einschulung: 10
```

10.2.2 Klassenmethoden

In Java ist es möglich, eine **Klassenmethode** über die Klasse selbst oder über eine Referenz auf ein Objekt anzusprechen. **Instanzmethoden** sind nur über eine Referenz auf ein Objekt aufrufbar.

Es zeugt jedoch von einem guten Programmierstil, wenn Klassenvariablen und Klassenmethoden nur über den Klassennamen wie beispielsweise

`Klassenname.klassenvariable`

angesprochen werden.

Im folgenden **Beispielprogramm** soll das Programm aus Kapitel 10.2.1 erweitert werden. Es soll die Klassenmethode `holeSchuelerAnzahl()` hinzukommen, welche die aktuelle Klassenstärke – also den Wert der Klassenvariablen `klassenStaerke` – zurückgibt. Die Klassenvariable `klassenStaerke` wird als `private` deklariert. Hier das erweiterte Programm:

```
// Datei: Schueler2.java
public class Schueler2
{
   private int nummerDesSchuelers;

   private static int klassenStaerke = 0;

   public void setzeNummer()
   {
      klassenStaerke++;
      nummerDesSchuelers = klassenStaerke;
   }

   public void abzaehlen()
   {
      System.out.println ("Ich bin die Nr.: " + nummerDesSchuelers);
   }

   public static int holeSchuelerAnzahl()
   {
      return klassenStaerke;
   }
}
```

```
// Datei: Schueler2Test.java

public class Schueler2Test
{
   public static void main (String[] args)
   {
      // Zugriff auf Klassenmethode, ohne dass ein Objekt existiert
      System.out.println ("Klassenstärke vor der Einschulung: "
                          + Schueler2.holeSchuelerAnzahl());

      Schueler2[] schuelerInKlasse = new Schueler2 [10];

      for (int lv = 0; lv < schuelerInKlasse.length; lv++)
      {
         schuelerInKlasse [lv] = new Schueler2();
         schuelerInKlasse [lv].setzeNummer();
      }

      for (Schueler2 element : schuelerInKlasse)
      {
         element.abzaehlen();
      }

      System.out.println("Klassenstärke nach der Einschulung: "
                       + schuelerInKlasse [0].holeSchuelerAnzahl());
   }
}
```

Die Ausgabe des Programms ist:

```
Klassenstärke vor der Einschulung: 0
Ich bin die Nr.: 1
Ich bin die Nr.: 2
   . . . . .
Ich bin die Nr.: 10
Klassenstärke nach der Einschulung: 10
```

An diesem Beispiel ist zu sehen, dass eine Klassenmethode aufgerufen werden kann:

- als Komponente der entsprechenden Klasse wie im Falle von `Schueler2.holeSchuelerAnzahl();`
- oder durch Zugriff über eine Referenz auf ein Objekt wie in folgendem Beispiel: `SchuelerInKlasse [0].holeSchuelerAnzahl();`

10.2.3 Übergabe von Objekten an Klassenmethoden

Bild 10-1 zeigt die UML-Notation für eine Klasse `Dampfer`. Jedes Schiff einer Flotte von Ausflugsdampfern wie z. B. der Dampfer "Michelangelo" oder "Leonardo da Vinci" ist eine Instanz dieser Klasse. Hier die Klasse `Dampfer`:

Bild 10-1 Klasse Dampfer

Bitte beachten Sie, dass **Klassenvariablen** und **Klassenmethoden** in einer Klasse nach UML unterstrichen dargestellt werden. Jeder Dampfer erhält eine laufende Nummer, das Datenfeld `dampferNummer`. Bei jeder Inbetriebnahme eines neuen Dampfers wird die Anzahl der Dampfer, geführt in der Klassenvariablen `anzahlDampfer`, um eins erhöht. Jeder Dampfer hat eine individuelle Sitzplatzkapazität `anzahlSitzplaetze`. Die Datenfelder `dampferNummer` und `anzahlSitzplaetze` sind Instanzvariablen, da sie für jedes Objekt der Klasse `Dampfer` – also für jeden einzelnen Dampfer – angelegt werden. Schließlich hat jeder Dampfer eine eigene Nummer und auch eine bestimmte Anzahl an Sitzplätzen. Die Anzahl der Dampfer einer Flotte gehört jedoch nicht zu einem individuellen Dampfer, sondern ist eine Eigenschaft der gesamten Flotte. Daher ist das Datenfeld `anzahlDampfer` eine **Klassenvariable**.

Die **Instanzmethode** `getAnzahlSitzplaetze()` gibt den Wert der **Instanzvariablen** `anzahlSitzplaetze` eines bestimmten Dampfers, d. h. einer bestimmten Instanz, zurück. Mit Hilfe der Instanzmethode `setAnzahlSitzplaetze()` kann die Sitzplatzanzahl für jeden Dampfer – also für jedes Objekt der Klasse `Dampfer` – individuell gesetzt werden. Durch den Aufruf

```
michelangelo.setAnzahlSitzplaetze (100);
```

wird beispielsweise der Wert der Instanzvariablen `anzahlSitzplaetze` für das `Dampfer`-Objekt `michelangelo` auf `100` gesetzt.

Mit Hilfe der überladenen Klassenmethode `setAnzahlSitzplaetze()` wird im folgenden Beispiel ebenfalls die Anzahl der Sitzplätze eines Dampfers festgelegt[143]. Da diese Klassenmethode aber nicht das Objekt kennt, auf dessen Methoden und Variablen sie arbeiten soll, muss ihr eine Referenz auf das entsprechende Objekt übergeben werden. Sie kann wie folgt aufgerufen werden:

```
Dampfer.setAnzahlSitzplaetze (michelangelo, 60);
```

Weiterhin stellt die Klasse `Dampfer` die Klassenmethode `getAnzahlDampfer()` bereit. Diese liefert den aktuellen Wert der Klassenvariablen `anzahlDampfer` zurück.

[143] Dieses Beispiel dient nur zur Demonstration der technischen Möglichkeiten. Es wird nicht empfohlen, dass Klassenmethoden Instanzvariablen ändern, es sei denn es handelt sich um Hilfsmethoden wie z. B. zum Vertauschen von Objekten.

Instanzmethoden haben stets Zugriff auf **Instanzvariablen und auf Klassenvariablen.**

Eine Instanzmethode kennt ihre Instanzvariablen, da diese zum gleichen Objekt gehören. Sie kennt auch ihre Klassenvariablen, da Klassenvariablen globale Variablen für alle Objekte einer Klasse darstellen.

Eine Klassenmethode kann auch auf den Instanzvariablen und den Instanzmethoden eines bestimmten Objekts arbeiten, wenn ihr explizit eine Referenz auf das entsprechende Objekt übergeben wird[144].

Hier nun das oben beschriebene **Beispielprogramm**:

```
// Datei: Dampfer.java

public class Dampfer
{
   private int dampferNummer = 0;
   private int anzahlSitzplaetze = 0;
   private static int anzahlDampfer = 0;

   public void init()
   {
      anzahlDampfer++;             // Zugriff auf Klassenvariable
      dampferNummer = anzahlDampfer;
      System.out.print ("Dampfer Nr. " + dampferNummer);
      System.out.print (" angelegt, Dampfer insgesamt: ");
      // Zugriff auf Klassenvariable
      System.out.println (anzahlDampfer);
   }

   public static int getAnzahlDampfer()
   {
      // Klassenmethode hat Zugriff auf Klassenvariable
      return anzahlDampfer;
   }

   // Der Klassenmethode wird ein Objekt der eigenen Klasse
   // übergeben
   public static void setAnzahlSitzplaetze (Dampfer dampfer,
                                            int sitzplaetze)
   {
      dampfer.setAnzahlSitzplaetze (sitzplaetze);
   }

   public void setAnzahlSitzplaetze (int sitzplaetze)
   {
      anzahlSitzplaetze = sitzplaetze;
      System.out.println ("Sitzplätze von Dampfer Nr. "
                        + dampferNummer + ": " + anzahlSitzplaetze);
   }
```

[144] Das wird aus Gründen des Software Engineering nicht empfohlen!

```
   public int getAnzahlSitzplaetze()
   {
      return anzahlSitzplaetze;
   }
}

// Datei: DampferTest.java

public class DampferTest
{
   public static void main (String[] args)
   {
      // Zugriff auf Klassenmethode, ohne dass ein Objekt existiert.
      System.out.println ("Dampfer insgesamt: "
                          + Dampfer.getAnzahlDampfer());
      // Zwei Dampfer anlegen
      Dampfer michelangelo = new Dampfer();
      michelangelo.init();
      Dampfer leonardoDaVinci = new Dampfer();
      leonardoDaVinci.init();

      // Sitzplätze festlegen
      michelangelo.setAnzahlSitzplaetze (100);
      leonardoDaVinci.setAnzahlSitzplaetze (150);

      // Leonardo Da Vinci wurde vergrößert
      Dampfer.setAnzahlSitzplaetze (leonardoDaVinci, 170);
      // Michelangelo wurde verkleinert
      Dampfer.setAnzahlSitzplaetze (michelangelo, 60);

      // Zugriff auf Klassenmethode über den Klassennamen.
      System.out.println ("Dampfer insgesamt: "
                          + Dampfer.getAnzahlDampfer());
   }
}
```

Die Ausgabe des Programms ist:

```
Dampfer insgesamt: 0
Dampfer Nr. 1 angelegt, Dampfer insgesamt: 1
Dampfer Nr. 2 angelegt, Dampfer insgesamt: 2
Sitzplätze von Dampfer Nr. 1: 100
Sitzplätze von Dampfer Nr. 2: 150
Sitzplätze von Dampfer Nr. 2: 170
Sitzplätze von Dampfer Nr. 1: 60
Dampfer insgesamt: 2
```

Auch wenn Instanzmethoden auf Klassenvariablen zugreifen können, wird dennoch empfohlen, dass das **Schreiben von Klassenvariablen** nur in Klassenmethoden, Konstruktoren oder sogenannten statischen Initialisierungsblöcken erfolgen soll.

Auf statische Initialisierungsblöcke wird in Kapitel 10.4.3 eingegangen.

Aus Gründen des Software Engineering sollen Klassenmethoden übergebene Instanzvariablen nicht ändern.

Damit eine Klassenmethode auf eine Instanz zugreifen kann, muss diese Instanz als Parameter an diese Klassenmethode übergeben werden. So wird beispielsweise vorgegangen, wenn Klassenmethoden als Hilfsmethoden zum Vertauschen von Objekten oder zum Umwandeln von Objekten von Wrapper-Klassen in Werte einfacher Datentypen verwendet werden.

10.3 Die `this`-Referenz

Jedes Objekt hat seine individuellen Instanzvariablen und in logischer Sicht auch dieselben Methoden wie alle anderen Objekte derselben Klasse. Da diese Methoden jedoch für alle Objekte einer Klasse stets dieselben sind, ist es aus Gründen des Speicherverbrauchs effizienter, diese Methoden an zentraler Stelle im Speicher nur ein einziges Mal abzulegen. Als zentrale Stelle bietet sich natürlich die Klasse selbst an. Dieser Sachverhalt soll an der vereinfachten Klasse `Person1` als **Beispiel** im Folgenden diskutiert werden. Hier die Klasse `Person1`:

```
// Datei: Person1.java

public class Person1
{
   private int alter;

   public void print()
   {
      System.out.println ("Alter: " + alter);
   }

   public void setzeAlter (int alt)
   {
      alter = alt;
   }
}

// Datei: TestPerson1.java

public class TestPerson1
{
   public static void main (String[] args)
   {
      Person1 p = new Person1();
      p.setzeAlter (10);
      p.print();
   }
}
```

Die Ausgabe des Programms ist:

```
Alter: 10
```

Bei der Ausführung eines Programms liegen die **Methoden** in der **Method-Area** der Java-Virtuellen Maschine und die **Objekte** mit ihren Instanzvariablen auf dem **Heap**.

Dies ist in Bild 10-2 zu sehen:

Bild 10-2 Instanzmethoden müssen Zugriff auf die Instanzvariablen im Heap haben

Damit gibt es zwei Probleme:

- zum einen muss ein Objekt seine Klasse finden
- und zum anderen muss eine Methode das Objekt finden, für welches sie aufgerufen wird.

Das erste Problem wird in Kapitel 11.6 behandelt und auf das zweite Problem soll im Folgenden eingegangen werden. **Die Fragestellung ist also, woher die Methode `print()` weiß, wo ein Datenfeld des betreffenden Objektes liegt, wenn sie aufgerufen wird?** Eigentlich bräuchte jede Methode, die auf Instanzvariablen arbeitet, auch eine Referenz auf die im Heap befindlichen Daten, die sie bearbeiten soll. Wenn 100 Personen angelegt werden, muss die `print()`-Methode in der Lage sein, auf die Daten der 100 verschiedenen Objekte zuzugreifen. Sie benötigt also zum Zeitpunkt der Abarbeitung immer eine Referenz genau auf die Daten desjenigen Objektes, zu dem der Methodenaufruf aktiviert wurde.

Also wäre es sinnvoll, wenn jede Instanzmethode einen zusätzlichen Übergabeparameter bekommen würde, an den beim Methodenaufruf die Referenz auf das im Heap befindliche Objekt übergeben werden kann. Über diese Referenz könnte dann auf die Daten zugegriffen werden. Genau auf diese Art und Weise wird der Zugriff einer Instanzmethode auf die entsprechenden Datenfelder zur Laufzeit auch realisiert. Diese ganze Umsetzung erfolgt jedoch für den Programmierer unsichtbar.

Zur Erläuterung wird in folgendem Beispiel so getan, als würde der Compiler diesen zusätzlichen Übergabeparameter einführen. Beachten Sie, dass dies nur fiktiv ist.

Welche Rolle hierbei der Compiler und welche Rolle die Java-Virtuelle Maschine spielt, kann hier nicht untersucht werden. Das folgende Programmbeispiel dient also nur zu Demonstrationszwecken und ist natürlich nicht kompilierbar. Hier das bereits erwähnte **Beispiel**:

```
public class Person
{
   private int alter;

   public void print (Person this)
   {
      System.out.println ("Name: " + this.name);
   }

   public void setzeAlter (Person this, int alt)
   {
      this.alter = alt;
   }
}

public class PersonTest
{
   public static void main (String[] args)
   {
      Person p = new Person();
      Person.setzeAlter (p, 10);
      Person.print (p);
   }
}
```

Auf alle Datenfelder wird nun mit Hilfe der übergebenen Referenz zugegriffen. Diese fiktive Umsetzung veranschaulicht sehr schön, dass eine Methode zur Klasse gehört, denn mit dem Aufruf

```
Person.setzeAlter (p, 10)
```

wird gesagt: Rufe die Methode `setzeAlter()` der Klasse `Person` auf. Dabei soll die Methode auf den Daten des Objektes arbeiten, dessen Referenz als erster Parameter übergeben wurde. Es ist kein Zufall, dass im obigen Beispiel dieser formale Parameter den Namen `this` trägt. Dieser Name ist bewusst gewählt, um nun die **`this`-Referenz** einführen zu können. Das oben Beschriebene ist für den Programmierer zwar unsichtbar realisiert, aber er hat trotzdem die Möglichkeit, die `this`-Referenz in seinen Programmen in folgenden Fällen zu benutzen:

- Der Programmierer **möchte explizit darauf aufmerksam machen**, dass er **auf eine Instanzvariable bzw. auf eine Instanzmethode des eigenen Objektes zugreift**.

  Hierfür ein **Beispiel**:

  ```
  // Datei: Person2.java

  public class Person2
  {
  ```

```
   private int alter;

   public void print()
   {
      this.printSterne();
      System.out.println ("Alter: " + this.alter);
      this.printSterne();
   }

   private void printSterne()
   {
      System.out.println ("*******************");
   }

   public void setzeAlter (int alt)
   {
      this.alter = alt;
   }
}

// Datei: TestPerson2.java

public class TestPerson2
{
   public static void main (String[] args)
   {
      Person2 p = new Person2();
      p.setzeAlter (10);
      p.print();
   }
}
```

Die Ausgabe des Programms ist:

```
*******************
Alter: 10
*******************
```

- **Ein Datenfeld hat den gleichen Namen wie eine lokale Variable.** In diesem Fall kann mit der `this`-Referenz auf das verdeckte Datenfeld zugegriffen werden. Dies wird in folgendem **Beispiel** demonstriert:

```
// Datei: Person3.java

public class Person3
{
   private int alter;
   private String vorname;
   private String name;

   public void print()
   {
      System.out.println ("Name:    " + name);
      System.out.println ("Vorname: " + vorname);
      System.out.println ("Alter:   " + alter);
   }
```

```
   public void setzeDaten (String name, String vorname, int alter)
   {
      // Zugriff auf verdeckte Datenfelder.
      this.name = name;       // Beachten Sie, dass die formalen
      this.vorname = vorname; // Parameter lokale Variable dar-
      this.alter = alter;     // stellen.
   }
}

// Datei: TestPerson3.java

public class TestPerson3
{
   public static void main (String[] args)
   {
      Person3 p = new Person3();
      p.setzeDaten ("Brang", "Rainer", 25);
      p.print();
   }
}
```

Die Ausgabe des Programms ist:

```
Name:    Brang
Vorname: Rainer
Alter:   25
```

- **Eine Referenz auf das aktuelle Objekt soll als Rückgabewert zurückgegeben werden.** Damit können Methodenaufrufe für dasselbe Objekt verkettet werden.

  Dies wird im nächsten **Beispiel** gezeigt:

```
// Datei: Bruch.java
public class Bruch
{
   private int zaehler;
   private int nenner;
   public void print()
   {
      System.out.println (zaehler + "/" + nenner);
   }

   public Bruch setzeWerte (int zaehler, int nenner)
   {
      this.zaehler = zaehler;
      this.nenner = nenner;
      return this;
   }

   public Bruch multipliziere (int faktor)
   {
      setzeWerte (zaehler * faktor, nenner);
      return this;
   }
}
```

```
// Datei: BruchTest.java

public class BruchTest
{
   public static void main (String[] args)
   {
      Bruch b1 = new Bruch();
      System.out.print ("Wert des Bruches b1: ");
      b1.setzeWerte (1, 2).print();

      Bruch b2 = new Bruch();
      System.out.print ("Wert des Bruches b2: ");
      b2.setzeWerte (1, 3).print();

      System.out.print ("Wert des Bruches b1: ");
      b1.multipliziere (10).print();
   }
}
```

Die Ausgabe des Programms ist:

```
Wert des Bruches b1: 1/2
Wert des Bruches b2: 1/3
Wert des Bruches b1: 10/2
```

- Eine **Referenz auf das aktuelle Objekt** soll als **Übergabeparameter an eine Methode** übergeben werden.

  Im folgenden **Beispiel** wird das vorangehende Beispiel der Klasse `Bruch` modifiziert:

```
// Datei: Bruch2.java

public class Bruch2
{
   private int zaehler;
   private int nenner;

   public void print()
   {
      System.out.println (zaehler + "/" + nenner);
   }

   public void setzeWerte (int zaehler, int nenner)
   {
      this.zaehler = zaehler;
      this.nenner = nenner;
   }

   // Eine Klassenmethode zum Erweitern eines Bruches
   public static Bruch2 erweitere (Bruch2 b, int faktor)
   {
      Bruch2 tmp = new Bruch2();
      tmp.setzeWerte (b.zaehler * faktor, b.nenner * faktor);
      return tmp;
   }
```

```
    public void addiere (Bruch2 b)
    {
       // Benutzt zur Berechnung die Klassenmethode erweitere()
       Bruch2 tmp1 = erweitere (b, nenner);
       Bruch2 tmp2 = erweitere (this, b.nenner);
       zaehler = tmp1.zaehler + tmp2.zaehler;
       nenner = tmp1.nenner;
    }
}

// Datei: Bruch2Test.java
public class Bruch2Test
{
    public static void main (String[] args)
    {
       Bruch2 b1 = new Bruch2();
       b1.setzeWerte (1, 2);
       System.out.print ("Wert des Bruches b1: ");
       b1.print();
       Bruch2 b2 = new Bruch2();
       b2.setzeWerte (1,3);
       System.out.print ("Wert des Bruches b2: ");
       b2.print();
       System.out.print ("b1 + b2 = ");
       b1.addiere (b2);
       b1.print();
    }
}
```

Die Ausgabe des Programms ist:

```
Wert des Bruches b1: 1/2
Wert des Bruches b2: 1/3
b1 + b2 = 5/6
```

Am Schluss soll nochmals darauf hingewiesen werden, dass Klassenmethoden keine `this`-Referenz besitzen.

Ein Programmierer kann die `this`-Referenz in folgenden Fällen benutzen:

- Der Programmierer möchte explizit darauf aufmerksam machen, dass er auf eine Instanzvariable bzw. auf eine Instanzmethode des eigenen Objektes zugreift.
- Ein Datenfeld hat den gleichen Namen wie eine lokale Variable.
- Eine Referenz auf das aktuelle Objekt soll als Rückgabewert zurückgegeben werden.
- Eine Referenz auf das aktuelle Objekt soll als Übergabeparameter an eine Methode übergeben werden.

Klassenmethoden können nicht über die `this`-Referenz auf Instanzvariablen bzw. Instanzmethoden zugreifen.

10.4 Initialisierung von Datenfeldern

Dieses Kapitel behandelt, wie Datenfelder von Klassen und Objekten initialisiert werden. Nimmt ein Programmierer keine Initialisierung vor, so werden die vom Compiler zur Verfügung gestellten Default-Initialisierungen **implizit** durchgeführt (siehe Kapitel 10.4.1).

Explizite Initialisierungen des Programmierers können sein:

- eine **manuelle Initialisierung** (siehe Kapitel 10.4.2),
- eine Initialisierung mit einem **Initialisierungsblock** (siehe Kapitel 10.4.3)
- sowie eine Initialisierung mit einem **Konstruktor** im Falle von Objekten (siehe Kapitel 10.4.4). Klassen können jedoch nicht mit Konstruktoren initialisiert werden.

10.4.1 Default-Initialisierungen von Datenfeldern

In Java werden Klassenvariablen und Instanzvariablen, d. h. klassen- und objektbezogene Datenfelder, automatisch mit **Default-Werten** (**Standard-Werten**) initialisiert.

Lokale Variablen werden nicht automatisch initialisiert.

Lokale Variablen müssen von Hand initialisiert werden (siehe Kapitel 6.4.4).

Die folgende Tabelle zeigt, mit welchen Default-Werten Datenfelder bei der **automatischen Initialisierung** (**Default-Initialisierung**) belegt werden:

Typ	Default-Wert
`boolean`	`false`
`char`	`'\u0000'`
`byte`	`0`
`short`	`0`
`int`	`0`
`long`	`0`
`float`	`0.0f`
`double`	`0.0d`
`Referenztyp`	`null`

Tabelle 10-1 Default-Werte für Datenfelder

Dabei ist zu beachten, dass Datenfelder, die Referenzvariablen sind, mit der `null`-Referenz als Default-Wert belegt werden.

10.4.2 Manuelle Initialisierung von Datenfeldern

Will man die Datenfelder mit anderen Werten als den Default-Werten belegen, so kann man die Datenfelder manuell wie im folgenden **Beispiel** initialisieren:

```
// Datei: Punkt2.java

public class Punkt2
{
   // Manuelle Initialisierung von anzahl könnte entfallen, da der
   // Default-Wert auch 0 ist.
   private static int anzahl = 0;
   private int x;                  // Der Default-Wert ist 0
   private int y = 1;              // Manuelle Initialisierung

   public void print()
   {
     System.out.println ("Die Koordinaten des Punktes sind:");
     System.out.println ("x = " + x + ", y = " + y);
   }

   public static void main (String[] args)
   {
      System.out.println ("Anzahl der Punkte: " + anzahl);
      Punkt2 p1 = new Punkt2();   // Anlegen eines Punkt-Objektes
      p1.print();
      anzahl++;    // Eine bessere Lösung wird später gezeigt
      System.out.println ("Anzahl der Punkte: " + anzahl);
   }
}
```

Die Ausgabe des Programms ist:

```
Anzahl der Punkte: 0
Die Koordinaten des Punktes sind:
x = 0, y = 1
Anzahl der Punkte: 1
```

Klassenvariablen – in obigem Beispiel `private static int anzahl = 0;` – werden **beim Laden der Klasse initialisiert**.

Instanzvariablen, hier:

```
private int x;
private int y = 1;
```

werden **beim Anlegen eines Objektes initialisiert**. Wird einer Instanzvariablen kein Wert manuell zugewiesen, d. h. wird sie nicht manuell initialisiert, so erhält sie als Default-Wert den entsprechenden "Null"-Wert (`false`, `'\u0000'`, `0`, `0.0`, `null`) aus Tabelle 10-1.

In Java ist es möglich, bei einer **manuellen Initialisierung** nicht nur Konstanten zur Initialisierung zu verwenden, sondern beliebige Ausdrücke.

Das folgende **Beispiel** zeigt die Möglichkeiten zur Initialisierung, die hierdurch entstehen:

```
public class Init
{
   // Eine Klassenmethode zur Initialisierung aufrufen
   private static int anzahl = Math.abs (-239);
   // Eine Klassenvariable der eigenen Klasse benutzen
   private int x = anzahl + 10;
   // Ein zuvor initialisierte Instanzvariable verwenden
   private int y = x - 100;

   // Ein Objekt mit Hilfe des new-Operators erzeugen
   private String str = new String ("Guten" + " Morgen");
}
```

Klassenvariablen werden beim **Laden der Klasse** initialisiert.

Instanzvariablen werden beim **Erzeugen eines Objektes** initialisiert.

Wird also irgendwo in einem Programmstück entweder auf eine Klassenvariable oder eine Klassenmethode zugegriffen, so wird die Klasse – sofern sie nicht schon früher benutzt wurde – in die Java-Virtuelle Maschine geladen und unmittelbar danach werden die Initialisierungen der Klassenvariablen der Reihe nach durchgeführt. Das Gleiche läuft ab, wenn mit Hilfe des `new`-Operators eine Instanz einer Klasse erzeugt wird. Auch dann wird ggf. zuerst die Klasse in die Java-Virtuelle Maschine geladen, die Initialisierungen der Klassenvariablen werden durchgeführt und erst danach kann das Objekt mit Hilfe des `new`-Operators erzeugt und die manuellen Initialisierungen für die Instanzvariablen durchgeführt werden.

Initialisierungen erfolgen stets der Reihe nach. Deshalb ist es nicht erlaubt, eine Klassenvariable manuell mit dem Wert einer anderen Klassenvariablen zu initialisieren, welche erst später definiert wird. Dasselbe gilt sinngemäß für Instanzvariablen.

Folgendes ist deshalb unzulässig:

```
public class Init
{
   // Benutzt Klassenvariable zur Initialisierung,
   // die erst weiter unten definiert ist.
   // Der Compiler gibt einen Fehler aus.
```

```
   private static int anzahl = stat;
   private static int stat = 999;

   // Benutzt Instanzvariable zur Initialisierung
   // die erst weiter unten definiert ist.
   // Der Compiler gibt einen Fehler aus.
   private int y = x - 100;
   private int x = anzahl + 10;
}
```

Es ist natürlich möglich, bei der Initialisierung einer Instanzvariablen eine Klassenvariable zu benutzen, die erst weiter unten in der Klasse definiert ist. Dies liegt daran, dass Initialisierungen von Klassenvariablen beim Laden der Klasse erfolgen und Instanzvariablen erst nach der Erzeugung eines Objektes initialisiert werden.

Das Folgende ist also korrekt:

```
public class Init
{
   private int x = anzahl + 10;
   private static int anzahl = Math.abs (999);
}
```

10.4.3 Initialisierung mit einem Initialisierungsblock

Eine weitere Möglichkeit zur Initialisierung von Datenfeldern ist die Initialisierung mit Hilfe eines **Initialisierungsblocks**.

Ein Block, der mit dem Schlüsselwort `static` eingeleitet wird, ist ein **statischer Initialisierungsblock**.

Ein solcher Block wird im Rahmen der **Initialisierung von Klassenvariablen**, d. h. beim Laden der Klasse genau einmal ausgeführt. Dies wird in folgendem **Beispiel** demonstriert:

```
// Datei: StaticBlockTest.java

class StaticBlock
{
   // Zählt die erzeugten Objekte von der Klasse.
   public static int anzahl = 0;
   // Anzahl der Aufrufe des statischen Initialisierungsblocks.
   public static int anzahlAufrufeStaticBlock = 0;
   static  //statischer Initialisierungsblock
   {
      System.out.println ("* Betreten des statischen Blocks *");
      anzahlAufrufeStaticBlock++;
   }
}
```

```
public class StaticBlockTest
{
   public static void main (String[] args)
   {
      StaticBlock objekt1 = new StaticBlock();
      StaticBlock.anzahl++;
      System.out.println ("Anzahl erzeugter Objekte: "
                           + StaticBlock.anzahl);
      System.out.println ("Aufrufe stat. Initialisierungsblock: "
                          + StaticBlock.anzahlAufrufeStaticBlock);

      StaticBlock objekt2 = new StaticBlock();
      StaticBlock.anzahl++;
      System.out.println ("Anzahl erzeugter Objekte: "
                          + StaticBlock.anzahl);
      System.out.println ("Aufrufe stat. Initialisierungsblock: "
                          + StaticBlock.anzahlAufrufeStaticBlock);
   }
}
```

Die Ausgabe des Programms ist:

```
* Betreten des statischen Blocks *
Anzahl erzeugter Objekte: 1
Aufrufe stat. Initialisierungsblock: 1
Anzahl erzeugter Objekte: 2
Aufrufe stat. Initialisierungsblock: 1
```

Das Anschreiben des Klassennamens `StaticBlock` in der zweiten Zeile der `main()`-Methode signalisiert der Laufzeitumgebung, dass der Code dieser Klasse nun benötigt wird und in die Java-Virtuelle Maschine geladen werden soll. Dabei werden die Klassenvariablen `anzahl` und `anzahlAufrufeStaticBlock` von der Java-Virtuellen Maschine jeweils mit ihrem Default-Wert `0` initialisiert.

Nachdem diese Initialisierung stattgefunden hat, wird der **statische Initialisierungsblock** abgearbeitet und dessen Anweisungen ausgeführt. Dort wird der Wert der Klassenvariablen `anzahlAufrufeStaticBlock` auf `1` gesetzt.

Nach der Abarbeitung des Blocks ist die Initialisierung der Klasse abgeschlossen und es kann das erste Objekt, auf das die Referenz `objekt1` zeigt, erzeugt werden. Danach wird die Klassenvariable `anzahl` um `1` erhöht. Bei der Erzeugung des zweiten Objektes, auf das die Referenz `objekt2` zeigt, ist der Code der Klasse `StaticBlock` schon in der Java-Virtuellen Maschine vorhanden. Es findet also kein Laden und auch keine Initialisierung der Klassenvariablen mehr statt. Der statische Initialisierungsblock wird also nicht mehr durchlaufen und es kann sofort das Objekt erzeugt werden. Die Klassenvariable `anzahlAufrufeStaticBlock` hat somit immer noch den Wert `1` und wird nicht mehr verändert. Die Klassenvariable `anzahl` hingegen wird erneut um `1` erhöht und hat dann den Wert `2`. Das Besondere an einem statischen Initialisierungsblock ist, dass in ihm wie in jedem anderen Block beliebige Anweisungen stehen können.

Ein **statischer Initialisierungsblock** kann beliebige Anweisungen enthalten. Er wird im Rahmen der Initialisierung von Klassenvariablen genau einmal ausgeführt.

Auch für die **Initialisierung von Instanzvariablen** gibt es einen **Initialisierungsblock**. Dieser ist allerdings nicht statisch.

Dies zeigt das folgende **Beispiel**:

```
// Datei: Punkt3.java

public class Punkt3
{
   public static int anzahl = 0;
   private int x;
   private int y;

   {
      System.out.println ("Wert von x: " + x);
      System.out.println ("Wert von y: " + y);

      anzahl++;
      y = 1;
   }

   public void print()
   {
     System.out.println ("Die Koordinaten des Punktes sind:");
     System.out.println ("x = " + x + ", y = " + y);
   }
}

// Datei: TestPunkt3.java
public class TestPunkt3
{
   public static void main (String[] args)
   {
      System.out.println ("Anzahl der Punkte: " + Punkt3.anzahl);
      Punkt3 p1 = new Punkt3();   // Anlegen eines Punkt-Objektes
      p1.print();
      System.out.println ("Anzahl der Punkte: " + Punkt3.anzahl);
   }
}
```

Die Ausgabe des Programms ist:

```
Anzahl der Punkte: 0
Wert von x: 0
Wert von y: 0
Die Koordinaten des Punktes sind:
x = 0, y = 1
Anzahl der Punkte: 1
```

Der nicht statische Initialisierungsblock wird einfach durch eine geschweifte öffnende Klammer eingeleitet und durch eine geschweifte schließende Klammer beendet. In ihm können wie im statischen Initialisierungsblock beliebige Anweisungen stehen. Ein nicht statischer Initialisierungsblock wird im Zuge der Initialisierung von Instanzvariablen ausgeführt. Im obigen Beispiel ist zu erkennen, dass die Default-Initialisierungen der Instanzvariablen `x` und `y` schon durchgeführt sind, wenn mit der Abarbeitung des Initialisierungsblocks begonnen wird.

Der **nicht statische Initialisierungsblock** wird jedes Mal dann ausgeführt, wenn ein Objekt dieser Klasse angelegt wird.

Damit ist der Initialisierungsblock dafür geeignet, die Klassenvariable `anzahl`, welche die Zahl der erzeugten Objekte der Klasse `Punkt` als Wert enthält, zu erhöhen.

Zuerst werden die Default-Initialisierungen durchgeführt. Manuelle Initialisierungen und Initialisierungen mit einem Initialisierungsblock werden **der Reihe nach** abgearbeitet und überschreiben die entsprechenden Default-Initialisierungen.

Anstelle des nicht statischen Initialisierungsblocks kann auch ein Konstruktor verwendet werden.

Für die Initialisierung der Objekte **anonymer Klassen** wird jedoch der **nicht statische Initialisierungsblock** benötigt, da anonyme Klassen keinen Konstruktor besitzen.

10.4.4 Konstruktoren zur Initialisierung

Konstruktoren sind Initialisierungsroutinen, die automatisch beim Erzeugen eines Objektes ausgeführt werden. Das bedeutet, dass die Initialisierung eines Objektes sofort nach dem Anlegen des Objektes durch Aufruf des Konstruktors erfolgt. Hierzu ist es lediglich erforderlich, dass der **Name der Initialisierungsroutine gleich dem Namen der Klasse** ist. Dies wird in dem folgenden **Beispiel** demonstriert:

```
// Datei: Punkt9.java

public class Punkt9
{
   public static int anzahl = 0; // Anzahl der Punkte.

   private int x;
   private int y = 1;

   public Punkt9()        // Dies ist ein Konstruktor.
   {
      System.out.println ("Anfang des Konstruktors");
      print();
      System.out.println ("Klassenvariable anzahl " +
                          "noch unverändert");
```

```
      System.out.println ("anzahl hat den Wert " + anzahl);

      // Initialisieren von Instanzvariablen
      x = 2;
      y = 3;
      print();

      // Hochzählen der Klassenvariable anzahl.
      anzahl++;
      System.out.println ("\nKlassenvariable anzahl " +
                          "inkrementiert");
      System.out.println ("anzahl hat den Wert " + anzahl);
      System.out.println ("Ende des Konstruktors");
   }

   public void print()
   {
     System.out.println ("\nDie Koordinaten des Punktes sind:");
     System.out.println ("x = " + x + ", y = " + y);
   }
}

// Datei: TestPunkt9.java

public class TestPunkt9
{
   public static void main (String[] args)
   {
      System.out.println ("Anzahl der Punkte: " + Punkt9.anzahl);
      Punkt9 p1 = new Punkt9();   // Anlegen eines Punkt-Objektes
                                  // und Aufruf des Konstruktors
   }
}
```

Die Ausgabe des Programms ist:

```
Anzahl der Punkte: 0
Anfang des Konstruktors

Die Koordinaten des Punktes sind:
x = 0, y = 1
Klassenvariable anzahl noch unverändert
anzahl hat den Wert 0

Die Koordinaten des Punktes sind:
x = 2, y = 3

Klassenvariable anzahl inkrementiert
anzahl hat den Wert 1
Ende des Konstruktors
```

Mit

```
Punkt9 p1 = new Punkt9();    // Anlegen eines Punkt-Objektes
                             // und Aufruf des Konstruktors
```

wird die Referenzvariable `p1` angelegt und ein Objekt der Klasse `Punkt9` ohne Namen auf dem Heap durch den `new`-Operator. Im Anschluss an das Anlegen des Objektes auf dem Heap wird von der Java-Virtuellen Maschine automatisch der Konstruktor aufgerufen.

Das Anlegen eines Objektes und der Konstruktoraufruf sind untrennbar miteinander verknüpft.

Nach dem Aufruf des Konstruktors wird die Referenz auf das angelegte Objekt der Referenzvariablen `p1` zugewiesen.

Beachten Sie, dass in obigem Beispiel im Konstruktor zum einen die **Initialisierung des Punktes** erfolgt und zum anderen auch die **Anzahl der angelegten Punkte** durch `anzahl++` hochgezählt wird. Bei jedem Aufruf des Konstruktors wird die Klassenvariable `anzahl` automatisch inkrementiert.

Da ein Konstruktor automatisch nach der Allokierung des Speicherplatzes für ein Objekt aufgerufen wird, wird

- in der Regel die Initialisierung eines Objekts im Konstruktor durchgeführt
- und die Anzahl der erzeugten Objekte am besten auch im Konstruktor hochgezählt.

Konstruktoren werden zu Beginn der Lebensdauer eines Objektes automatisch aufgerufen. Im Einzelnen gilt:

- Konstruktoren werden unmittelbar nach der durchgeführten Reservierung des Speicherplatzes auf dem Heap durch den `new`-Operator aufgerufen.
- Konstruktoren von Basisklassen werden vor den Konstruktoren ihrer Nachkommen aufgerufen (siehe Kapitel 11.3.2).

Ein Konstruktor

- dient zum **Initialisieren eines Objektes**,
- unterscheidet sich von einer normalen Methode unter anderem dadurch, dass der Konstruktor **ohne Rückgabetyp** (auch nicht `void`) deklariert wird,
- unterscheidet sich weiterhin von einer normalen Methode auch dadurch, dass er **nicht** an eine abgeleitete Klasse **vererbt** wird,
- wird vom Compiler dadurch erkannt, dass er den **gleichen Namen** trägt wie die Klasse selbst und keinen Rückgabewert hat.

10.4.4.1 Beispiel zur Initialisierungsreihenfolge

Das folgende Beispiel zeigt, wie die Instanzvariablen eines Objektes zunächst mit manuell angegebenen Anfangswerten belegt werden. Mit Hilfe des Konstruktors können diese anschließend überschrieben werden:

```
// Datei: Person5.java

public class Person5
{
   private String vorname = "Rainer";
   private String name = "Brang";
   private int alter = 25;

   // Dies ist ein selbst geschriebener parameterloser Konstruktor
   public Person5()
   {
      System.out.print ("Felder beim Eintritt in den ");
      System.out.println ("Konstruktor:");
      print();
      vorname = "Franz";

      name = "Müller";
      alter = 35;
   }

   public void print()
   {
      System.out.println ("Vorname:  " + vorname);
      System.out.println ("Name:     " + name);
      System.out.println ("Alter:    " + alter);
   }
}

// Datei: TestPerson5.java

public class TestPerson5
{
   public static void main (String[] args)
   {
      Person5 p1 = new Person5();
      System.out.println ("Felder nach dem Konstruktoraufruf: ");
      p1.print();
   }
}
```

Die Ausgabe des Programms ist:

```
Felder beim Eintritt in den Konstruktor:
Vorname:  Rainer
Name:     Brang
Alter:    25
Felder nach dem Konstruktoraufruf:
Vorname:  Franz
Name:     Müller
Alter:    35
```

Die Default-Werte von Datenfeldern werden durch eine manuelle Initialisierung oder einen Initialisierungsblock überschrieben. Initialisierungen, die im Konstruktor durchgeführt werden, überschreiben sowohl Default-Werte als auch die Werte einer manuellen Initialisierung und die Werte der Initialisierungen eines Initialisierungsblocks.

10.4.4.2 Konstruktoren mit Parametern

Wie Methoden können auch Konstruktoren mit formalen Parametern versehen werden. Auch der Rumpf des Konstruktors ist wie bei einer normalen Methode aufgebaut. Bei der Erzeugung eines Objektes werden die aktuellen Parameter übergeben. Die folgende Klasse `Bruch3` hat einen Konstruktor mit zwei Parametern:

```
public class Bruch3
{
   private int zaehlerFeld;
   private int nennerFeld;

   public Bruch3 (int zaehler, int nenner)
   {
      zaehlerFeld = zaehler;
      nennerFeld  = nenner;
   }

   public void print()
   {
      . . . . .
   }
}
```

Beim Anlegen eines Objektes mit dem `new`-Operator müssen die aktuellen Parameter für den nachfolgenden Konstruktoraufruf übergeben werden. So bewirkt die Codezeile

```
Bruch3 refBruch = new Bruch3 (2, 4);
```

das Anlegen eines Objektes der Klasse `Bruch3` auf dem Heap, gefolgt vom Aufruf des Konstruktors mit 2 Parametern.

10.4.4.3 Default-Konstruktor

Der **Default-Konstruktor** wird vom Compiler zur Verfügung gestellt. Er ist für jede Klasse automatisch definiert, vorausgesetzt, es wird kein Konstruktor selbst definiert. Ein Default-Konstruktor einer Klasse hat keine Parameter. Er ist ein **parameterloser Konstruktor**.

.
Bevor weitere Erläuterungen folgen, sollen die Begriffe, die für Konstruktoren in den folgenden Kapiteln verwendet werden, hier nochmals zusammengefasst und gegeneinander abgegrenzt werden:

- Ein **parameterloser Konstruktor** ist ein Konstruktor ohne Parameter, der dem Benutzer in der Regel[145] keine frei vorgebbare individuelle Initialisierung von Datenfeldwerten ermöglicht, da keine Parameter übergeben werden können.
- **Konstruktoren mit Parametern** erlauben eine frei vorgebbare individuelle Initialisierung von Objekten.
- Der **Default-Konstruktor** ist der im Fall der Abwesenheit anderer Konstruktoren vom Compiler zur Verfügung gestellte parameterlose Konstruktor.

Wird überhaupt kein Konstruktor selbst geschrieben, so wird automatisch nach dem Anlegen eines jeden Objektes der **Default-Konstruktor** aufgerufen.

So war zum Beispiel im Programm `Bruch.java` in Kapitel 10.3 gar kein selbst geschriebener Konstruktor vorhanden. Mit

```
b = new Bruch();
```

wurde der Default-Konstruktor `Bruch()` des Compilers aufgerufen.

Konstruktoren können genauso wie normale Methoden überladen werden. Es können beliebig viele Konstruktoren selbst geschrieben werden, die sich in Typ und Anzahl der Übergabeparameter unterscheiden.

Es ist jedoch das Folgende zu beachten: Sobald nur ein einziger selbst geschriebener Konstruktor existiert – gleichgültig ob mit oder ohne Parameter –, ist der vom Compiler zur Verfügung gestellte Default-Konstruktor nicht mehr sichtbar. Deshalb kann von einer Klasse, die nur Konstruktoren mit Parametern zur Verfügung stellt, kein Objekt mehr mit der Anweisung

```
Klassenname refK = new Klassenname();
```

erzeugt werden. Der Compiler gibt in diesem Fall eine Fehlermeldung aus.

Sobald nur ein einziger selbst geschriebener Konstruktor existiert, ist der vom Compiler zur Verfügung gestellte Default-Konstruktor nicht mehr vorhanden.

10.4.5 Aufruf eines Konstruktors im Konstruktor

Ein **Konstruktor** einer Klasse kann **in seiner ersten Anweisung** einen **anderen Konstruktor derselben Klasse** aufrufen.

[145] Eine frei vorgebbare individuelle Initialisierung kann hier lediglich dadurch erreicht werden, dass im Konstruktur der Benutzer aufgefordert wird, von der Tastatur aus individuelle Werte einzugeben.

So kann beispielsweise ein selbst geschriebener parameterloser Konstruktor einer Klasse `Person4` einen Konstruktor mit Parametern derselben Klasse aufrufen und dabei die Default-Werte – bzw. die durch manuelle Initialisierungen und Initialisierungen in einem nicht statischen Initialisierungsblock erzeugten Werte – der Datenfelder überschreiben.

Mit Hilfe von

```
this (parameterliste)
```

kann ein solcher Aufruf eines anderen Konstruktors derselben Klasse erfolgen.

Der Aufruf eines anderen Konstruktors derselben Klasse wird in folgendem **Beispiel** gezeigt:

```
// Datei: Person4.java

public class Person4
{
   private String vorname;
   private String name;

   public Person4 (String v, String n)
   {
      System.out.println ("Im Konstruktor mit Parametern!");
      System.out.println ("   Name:    " + n);
      System.out.println ("   Vorname: " + v);
      vorname = v;
      name = n;
   }

   public Person4()
   {
      this ("Vorname unbekannt", "Nachname unbekannt");
      System.out.println ("Im parameterlosen Konstruktor!");
   }

   public void print()
   {
      System.out.println ("Ausgabe der print()-Methode");
      System.out.println ("   Name:    " + name);
      System.out.println ("   Vorname: " + vorname);
   }
}

// Datei: TestPerson4.java

public class TestPerson4
{
   public static void main (String[] args)
   {
      Person4 p = new Person4();
      p.print();
   }
}
```

Die Ausgabe des Programms ist:

```
Im Konstruktor mit Parametern!
   Name:     Nachname unbekannt
   Vorname: Vorname unbekannt
Im parameterlosen Konstruktor!
Ausgabe der print()-Methode
   Name:     Nachname unbekannt
   Vorname: Vorname unbekannt
```

Bei der Ausgabe des Programms ist zu beachten, dass im parameterlosen Konstruktor als allererstes der Konstruktor mit Parametern aufgerufen werden muss. Deshalb kann die Ausgabe `"Im parameterlosen Konstruktor!"` erst nach der Abarbeitung des Konstruktors mit Parametern erfolgen. Genauso kann ein Konstruktor mit Parametern mit `this()` als erste Anweisung den entsprechenden parameterlosen Konstruktor aufrufen.

Mit Hilfe von `this (parameterliste)` kann aus einem Konstruktor ein anderer Konstruktor derselben Klasse aufgerufen werden. Diese Anweisung muss allerdings die erste Anweisung im Rumpf des Konstruktors sein.

10.5 Instanziierung von Klassen

Objekte werden nach dem Bauplan einer Klasse erzeugt.

Das Erzeugen eines Objektes einer Klasse wird auch als **Instanziierung** oder **Instanziieren** einer Klasse bezeichnet. Damit soll zum Ausdruck gebracht werden, dass eine **Instanz** dieser Klasse geschaffen wird.

Zwei Begriffe, deren Bedeutung oft verwechselt wird, sind Instanziierung und Initialisierung:

- Bei der **Instanziierung** wird ein neues Objekt auf dem Heap angelegt. Die Instanziierung wird in der Regel mit dem `new`-Operator durchgeführt.
- Bei der **Initialisierung** eines Objekts werden die Datenfelder des erzeugten (instanziierten) Objektes mit Werten belegt. Die Initialisierung kann mit Default-Werten, mit Hilfe einer manuellen Initialisierung, mit einem nicht statischen Initialisierungsblock oder mit Hilfe des Konstruktors erfolgen.

Wird ein Objekt mit Hilfe des `new`-Operators geschaffen, so wird Speicher für dieses Objekt bereitgestellt. Durch Aufruf des Konstruktors kann das Objekt initialisiert werden.

10.5.1 Ablauf bei der Instanziierung

Anhand der folgenden Anweisung, in der `p1` ein Datenfeld einer Klasse sein soll und keine lokale Variable, wird betrachtet, welche Schritte in welcher Reihenfolge bei der Instanziierung

```
Person p1 = new Person();
```

ablaufen:

- In **Schritt 1** wird die **Referenzvariable** `p1` vom Typ `Person` **angelegt** und mit `null` initialisiert.
- In **Schritt 2** wird durch `new Person` der `new`-Operator aufgerufen und die **Klasse** `Person` **instanziiert**, mit anderen Worten, es wird ein Objekt der Klasse `Person` auf dem Heap erzeugt.
- Schließlich erfolgt in **Schritt 3** die **Initialisierung des Objektes**. Es werden Default-Initialisierungen der Instanzvariablen des Objekts durchgeführt (je nach Typ mit `0`, `0.0f`, `0.0d`, `'\u0000'`, `false` bzw. `null`) und dann eventuell angegebene manuelle Initialisierungen und Initialisierungen durch einen nicht statischen Initialisierungsblock. Anschließend wird der Konstruktor aufgerufen.
- In Schritt 4 gibt der `new`-Operator eine **Referenz** auf das neu im Heap erzeugte Objekt zurück, welche der **Referenzvariablen** `p1` **zugewiesen** wird.

Beachten Sie, dass der Konstruktor den gleichen Namen wie die Klasse tragen muss. Zum einen sagt in Schritt 2 der Klassenname `Person` dem `new`-Operator, dass ein Objekt der Klasse `Person` geschaffen werden soll. Weiterhin wird der `new`-Operator nach der Erzeugung des Objektes in Schritt 3 veranlasst, den Konstruktor `Person()` aufzurufen.

Diese Reihenfolge kann belegt werden durch die Aussage von Oracle [oraexe]: "Just before a reference to the newly created object is returned as the result the indicated constructor is processed to initialize the object …". Man kann sich diese Reihenfolge auch experimentell klar machen, da der `new`-Operator `null` zurückgibt, wenn im Konstruktor eine Exception geworfen wird.

10.5.2 Das Entwurfsmuster Singleton

Deklariert man alle selbst geschriebenen Konstruktoren als `private`, so ist es nicht möglich, in einer anderen Klasse Objekte dieser Klasse mit dem `new`-Operator zu erzeugen. Wenn man nur ein einziges Objekt erzeugen können soll, geht man in der Regel nach dem **Singleton-Entwurfsmuster** vor. Hierbei erfolgt die Erzeugung des einzigen Objektes über eine speziell zur Verfügung gestellte Klassenmethode.

Das folgende Programm zeigt eine Implementierung des Singleton-Entwurfsmusters und stellt sicher, dass nur ein einziges Objekt einer Klasse erzeugt wird:

```
// Datei: SingletonTest.java

class Singleton
{
   private static Singleton instance;

   private Singleton()
   {
      System.out.println ("Bin im Konstruktor");
   }

   public static Singleton getInstance()
   {
      if (instance == null)
      {
         instance = new Singleton();
      }
      return instance;
   }
}

public class SingletonTest
{
   public static void main (String[] args)
   {
      // Singleton s = new Singleton(); gibt Fehler
      Singleton s2 = Singleton.getInstance();  // new wird
                                               // aufgerufen
      Singleton s3 = Singleton.getInstance();  // new wird nicht
                                               // mehr aufgerufen
   }
}
```

Die Ausgabe des Programms ist:

```
Bin im Konstruktor
```

10.6 Freigabe von Speicher

In der Programmiersprache Java hat der Programmierer nicht die Möglichkeit – aber auch nicht die Pflicht – Speicherplatz auf dem Heap, der nicht länger benötigt wird, selbst freizugeben. Der **Garbage Collector** der Java-Virtuellen Maschine hat alleine die Verantwortung, Speicherplatz auf dem Heap, der nicht länger benötigt wird, aufzuspüren und freizugeben.

Der Programmierer kann bei der Verwendung der in diesem Buch eingeführten Referenzen – auch starke Referenzen genannt – die Freigabe eines Objektes nur dadurch beeinflussen, indem er alle Referenzen auf dieses Objekt auf `null` setzt[146].

[146] Sogenannte schwache Referenzen werden in diesem Einsteigerbuch nicht behandelt.

Denn wenn ein Objekt von niemanden mehr referenziert wird, kann es der Garbage Collector freigeben. Wann dies erfolgt, ist jedoch Sache der Java-Virtuellen Maschine.

In Java wird nicht garantiert, dass ein Objekt während der Laufzeit eines Programmes zerstört wird. Der Garbage Collector wird nur dann tätig, wenn er spürt, dass es eng im Heap wird. Wenn zum Anlegen eines neuen Objektes der vorhandene Platz im Heap nicht ausreicht[147], muss die Java-Virtuelle Maschine versuchen, durch eine Speicherbereinigung des Garbage Collectors Platz zu gewinnen. Schlägt dieser Versuch fehl, so wird eine **Exception**[148] vom Typ **`OutOfMemoryError`** ausgelöst.

Bei einer Speicherbereinigung werden die nicht referenzierten Objekte aus dem Heap entfernt. Mit anderen Worten, ihr Platz wird zum Überschreiben freigegeben.

Lässt man im Beispiel

```
Person p1 = new Person();
```

durch

```
p1 = null;
```

die Referenz `p1` nicht länger auf das mit `new` geschaffene Objekt, sondern auf `null` zeigen, so wird damit vom Programmierer explizit das Objekt im Heap zur Speicherbereinigung freigegeben – vorausgesetzt, es existiert keine weitere Referenz auf dieses Objekt.

Wann die Java-Virtuelle Maschine einen Lauf des Garbage Collectors durchführt, ist Sache der Java-Virtuellen Maschine.

Beeinflussen der Heap-Größe

Beim Start eines Java-Programms mit

```
java Klassenname
```

legt die Java-Virtuelle Maschine die maximale Heap-Größe in Abhängigkeit vom verwendeten Betriebssystem und der verwendeten Prozessor-Architektur fest. Die Ausführung des Java-Interpreters, z. B. mit `java.exe`, kann durch Hinzufügen verschiedenster Kommandozeilen-Optionen beeinflusst werden. So gibt es unter anderem die beiden Optionen `-Xms` bzw. `-Xmx`, mit deren Hilfe die minimale bzw. maximale Größe des Heap der gestarteten Java-Virtuellen Maschine individuell festgelegt werden kann. So startet der Aufruf

```
java -Xmx100m Klassenname
```

[147] Der Speicherbereich des Heap stößt dann an seine Grenzen, wenn mehr Objekte erzeugt werden, als der Heap aufnehmen kann.
[148] Exceptions werden in Kapitel 13 behandelt.

eine Java-Virtuelle Maschine mit einer maximalen Heap-Größe von 100 MB. Dabei ist zu beachten, dass die Heap-Größe direkt hinter den Optionen (ohne Leerzeichen) angegeben werden müssen und die Einheitspräfixe nicht vergessen werden dürfen. Eine Möglichkeit, wie man den Heap komplett belegen kann, wird im folgenden Beispielprogramm gezeigt[149]. Ein Objekt der generischen Klasse[150] `Vector<T>` wird verwendet, um beliebig viele Objekte vom Typ `StringBuffer` aufzunehmen. Das Füllen des `Vector<T>`-Objektes erfolgt innerhalb einer Endlosschleife. Es werden somit so lange Objekte vom Typ `StringBuffer` erzeugt und ihre Referenzen in das `Vector<T>`-Objekt eingefügt, bis der Heap keine Objekte mehr aufnehmen kann und der Speicher voll ist. Es wird dann eine Exception[151] vom Typ `OutOfMemoryError` geworfen. Dies bedeutet, dass die Endlosschleife verlassen wird und das Programm normalerweise abbricht. In dem Beispiel unten wird die geworfene Exception durch das `try/catch`-Konstrukt abgefangen und behandelt. Innerhalb des `catch`-Blocks findet die Fehlerbehandlung statt. Es wird zuerst eine Referenz auf das Objekt vom Typ `Runtime`, das die aktuelle Laufzeitumgebung darstellt, besorgt. Durch Aufruf der Methode `freeMemory()` für das Objekt vom Typ `Runtime` kann dann der noch freie Heap-Speicher abgefragt werden. Nachdem der freie Speicher auf dem Heap abgefragt und ausgegeben wurde, wird den Referenzen auf die erzeugten Objekte vom Typ `StringBuffer`, die das `Vector<T>`-Objekt hält, durch Aufruf der Methode `clear()` des `Vector<T>`-Objekts die `null`-Referenz zugewiesen. Damit werden die `StringBuffer`-Objekte nicht mehr referenziert und können jetzt durch den Garbage Collector freigegeben werden. Zwei weitere Abfragen des freien Speichers zeigen allerdings, dass der Garbage Collector nicht wirklich viel Speicher freigibt. Erst nachdem wieder erneut Speicher durch das Erzeugen eines `StringBuffer`-Objektes angefordert wird, gibt der Garbage Collector den Speicher frei.

Hier das bereits erwähnte **Beispielprogramm**:

```
// Datei: GarbageCollectorTest.java
import java.util.*;

public class GarbageCollectorTest
{
   // Bitte beachten Sie nicht das throws Exception-Konstrukt
   public static void main (String[] args) throws Exception
   {
      Vector<StringBuffer> v = new Vector<StringBuffer>();

      try // try-Block
      {
         for(;;) // Endlosschleife
         {
            // Viele String-Buffer-Objekte erzeugen
            v.add (new StringBuffer (2000));
         }
      }
```

[149] Es wird hierbei auf die Themen Ausnahmebehandlung (siehe Kapitel 13) und Collections (siehe Kapitel 19) vorgegriffen. Das Beispielprogramm wird besser verstanden, wenn die beiden genannten Kapitel zuvor behandelt wurden.

[150] Generische Klassen werden erst in Kapitel 17 behandelt.

[151] Exceptions werden in Kapitel 13 behandelt.

```
      catch (Throwable e) // catch-Block für die Fehlerbehandlung
      {
         // Hier wird die Exception vom Typ OutOfMemoryError
         // abgefangen. Es kann nun der Fehler behandelt werden.

         // Referenz auf die aktuelle Laufzeitumgebung.
         Runtime r = Runtime.getRuntime();

         // Der Aufruf von freeMemory() auf dem Runtime-Objekt frägt
         // von diesem den im Moment des Aufrufs zur Verfügung
         // stehenden freien Speicher des Heap ab.
         System.out.println (
            "Freier Speicher vor clear(): " + r.freeMemory());

         // Das Vector<T>-Objekt wurde mit Referenzen auf Objekte
         // vom Typ StringBuffer gefüllt. Das bedeutet, dass die
         // erzeugten StringBuffer-Objekte nur vom Vector<T>-Objekt
         // referenziert werden. Der Aufruf von clear() auf dem
         // Vector<T>-Objekt löscht nun alle im Vector gespeicherten
         // Referenzen. Somit werden die erzeugten StringBuffer-
         // Objekte nicht mehr referenziert und der Garbage
         // Collector kann die Objekte auf dem Heap löschen.
         v.clear();

         // Alle zuvor erzeugten Objekte vom Typ StringBuffer
         // werden nicht mehr referenziert und sind Datenmüll.
         // Sie können vom Garbage Collector beseitigt werden.
         System.out.println (
            "Freier Speicher nach clear(): " + r.freeMemory());

         // Fünf Sekunden warten. Vielleicht hat danach
         // die Java-Virtuelle Maschine aufgeräumt?
         Thread.sleep (5000);

         System.out.println (
            "Freier Speicher 5s nach clear(): " + r.freeMemory());

         System.out.println ("Wieder Speicher belegen!");
         v.add (new StringBuffer (2000));

         System.out.println ("Freier Speicher: " + r.freeMemory());
      }
   }
}
```

Die Ausgabe des Programms ist:

```
Freier Speicher vor clear(): 3912
Freier Speicher nach clear(): 3912
Freier Speicher 5s nach clear(): 3832
Wieder Speicher belegen!
Freier Speicher: 226531216
```

In Zusammenhang mit dem Garbage Collector wird auch oft die Methode `finalize()` erwähnt. Die Methode `finalize()` ist in der Klasse `Object` definiert.

Da jede Klasse in Java von der Klasse `Object` abgeleitet ist, besitzt auch jedes Objekt in Java eine geerbte Methode `finalize()`.

Die Methode `finalize()` der Klasse `Object` hat allerdings einen leeren Methodenrumpf. Das Besondere an der Methode `finalize()` ist nun, dass der Garbage Collector die Methode `finalize()` aufruft, bevor er ein Objekt aus dem Speicher entfernt. Damit könnte man durch Überschreiben der Methode `finalize()` (das Überschreiben von Methoden wird erst in Kapitel 11.2.2 behandelt) beispielsweise nicht mehr benötigte Ressourcen für ein Objekt freigeben.

Da allerdings weder definiert ist, wann der Garbage Collector die Objekte aus dem Speicher entfernt, noch sichergestellt ist, dass der Garbage Collector auf jeden Fall eine Speicherbereinigung vor der Beendigung eines Programms durchführt, ist generell von der Verwendung der Methode `finalize()` abzuraten.

10.7 Die Klasse `Object`

Jede Klasse und jeder Array-Typ wird implizit, d. h. ohne eine explizite Angabe des Programmierers, von der Klasse `Object` abgeleitet.

Damit beinhaltet jede Klasse und ein jedes Array automatisch alle Methoden der Klasse `Object`. Die Methoden der Klasse `Object` lassen sich in zwei Kategorien einteilen:

- in Methoden, die Threads (siehe Kapitel 16) unterstützen
- und in allgemeine Utility-Methoden.

Hier werden nur die Utility-Methoden aufgeführt:

- `public String toString()`

  Die Methode `toString()` ermöglicht die Ausgabe eines Strings, der für das Objekt charakteristisch ist. Der Rückgabewert der Methode `toString()` ist eine Zeichenkette, die das entsprechende Objekt charakterisiert.

- `public boolean equals (Object obj)`

  Diese Methode gibt bei einem Aufruf `x.equals (y)` das Ergebnis `true` zurück, wenn `x` und `y` Referenzen auf dasselbe Objekt sind.

- `protected Object clone() throws CloneNotSupportedException`[152]

  Die Methode `clone()` erlaubt es, eine Kopie eines Objektes zu erzeugen. Die Methode `clone()` wird in Kapitel 14.6 vorgestellt.

[152] Auf die Exception-Klasse `CloneNotSupportedException` soll hier nicht eingegangen werden.

- `protected void finalize() throws Throwable`[153]

  Die Methode `finalize()` erlaubt "Aufräumarbeiten" vor der Zerstörung eines Objektes. Von der Verwendung der Methode `finalize()` wird allerdings abgeraten.

10.8 Zusammenfassung

Kapitel **10.1** erläutert die Bedeutung des Information Hiding.

Ein Ziel der objektorientierten Programmierung ist es, die Daten und die Implementierung der Methoden eines Objekts zu verbergen. Das bedeutet, dass das Prinzip des Information Hiding angewandt werden soll. Es soll kein Unbefugter die Daten eines Objekts verändern können. Nur die eigenen Methoden eines Objektes sollen auf die Daten des entsprechenden Objektes Zugriff haben.

Java erfüllt dieses Ziel aber nur zum Teil. In Java ist der Zugriffsschutz nur klassenbezogen realisiert. Dies bedeutet, dass es nur Objekten fremder Klassen verwehrt werden kann, auf ein anderes Objekt zuzugreifen. Ein Objekt einer bestimmten Klasse kann in Java stets auf die Methoden und Datenfelder eines Objekts derselben Klasse zugreifen! Aus Gründen des Software Engineering sollte es in Java aber keinen direkten Zugriff auf die Daten eines Objektes durch andere Klassen bzw. Objekte anderer Klassen geben. Direkter Zugriff bedeutet, dass über eine Referenz direkt auf ein Datenfeld zugegriffen werden kann.

Kapitel **10.2** untersucht Klassenvariablen und Klassenmethoden.

Klassenvariablen, die für alle Objekte einer Klasse als globale Daten zur Verfügung stehen, werden mit Hilfe des Schlüsselwortes `static` deklariert. Klassenvariablen sind nicht Teil von Objekten und liegen deshalb nicht auf dem Heap. Klassenvariablen werden bei der Klasse in der sogenannten Method Area geführt und sind deshalb für alle Objekte einer Klasse nur ein einziges Mal vorhanden. Der Zugriff auf Klassenvariablen ist ohne die Existenz einer Instanz einer Klasse möglich. Der Zugriff erfolgt dann über den Klassennamen wie z. B. `Schueler.klassenStaerke`. Der Zugriff auf eine Klassenvariable kann auch über eine Referenz auf ein Objekt der entsprechenden Klasse erfolgen wie z. B. durch Aufruf von `schuelerInKlasse[0].klassenStaerke`, da ein Objekt stets weiß, zu welcher Klasse es gehört.

In Java ist es möglich, eine Klassenmethode über die Klasse selbst oder über eine Referenz auf ein Objekt anzusprechen. Instanzmethoden sind nur über eine Referenz auf ein Objekt aufrufbar. Es zeugt jedoch von einem guten Programmierstil, Klassenvariablen und Klassenmethoden nur über den Klassennamen anzusprechen wie beispielsweise bei

```
Klassenname.klassenvariable
```

Instanzmethoden haben stets Zugriff auf Instanzvariablen und auf Klassenvariablen. Eine Klassenmethode kann auch auf den Instanzvariablen und den Instanzmethoden

[153] Die Exception-Klasse `Throwable` soll hier nicht betrachtet werden.

eines bestimmten Objekts arbeiten, wenn ihr explizit eine Referenz auf das entsprechende Objekt übergeben wird. Auch wenn Instanzmethoden auf Klassenvariablen zugreifen können, wird dennoch empfohlen, dass das Schreiben von Klassenvariablen nur in Klassenmethoden, Konstruktoren oder sogenannten statischen Initialisierungsblöcken erfolgen soll. Aus Gründen des Software Engineering sollen Klassenmethoden übergebene Instanzvariablen nicht ändern.

Kapitel **10.3** führt die `this`-Referenz ein.

Ein Programmierer kann die `this`-Referenz in folgenden Fällen benutzen:

- Der Programmierer möchte explizit darauf aufmerksam machen, dass er auf eine Instanzvariable bzw. auf eine Instanzmethode des eigenen Objektes zugreift.
- Ein Datenfeld hat den gleichen Namen wie eine lokale Variable.
- Eine Referenz auf das aktuelle Objekt soll als Rückgabewert zurückgegeben werden.
- Eine Referenz auf das aktuelle Objekt soll als Übergabeparameter an eine Methode übergeben werden.

Klassenmethoden können nicht über die `this`-Referenz auf Instanzvariablen bzw. Instanzmethoden zugreifen.

Kapitel **10.4** befasst sich mit der Initialisierung von Datenfeldern und zwar mit der Default-Initialisierung, der manuellen Initialisierung, der Initialisierung durch statische und nicht statische Initialisierungsblöcke und durch die Anweisungen des Konstruktors. Bei Konstruktoren werden parameterlose Konstruktoren, der Default-Konstruktor und Konstruktoren mit Parametern betrachtet, ebenso wie der Aufruf eines anderen Konstruktors im Konstruktor.

In Java werden Klassenvariablen und Instanzvariablen, d. h. klassen- und objektbezogene Datenfelder, automatisch mit Default-Werten (Standard-Werten) initialisiert. Lokale Variablen werden nicht automatisch initialisiert.

In Java ist es möglich, bei einer manuellen Initialisierung nicht nur Konstanten zur Initialisierung zu verwenden, sondern beliebige Ausdrücke.

Klassenvariablen werden beim Laden der Klasse initialisiert. Instanzvariablen werden beim Erzeugen eines Objektes initialisiert. Initialisierungen erfolgen stets der Reihe nach. Deshalb ist es nicht erlaubt, eine Klassenvariable manuell mit dem Wert einer anderen Klassenvariablen zu initialisieren, welche erst später definiert wird. Dasselbe gilt sinngemäß für Instanzvariablen.

Ein Block, der mit dem Schlüsselwort `static` eingeleitet wird, ist ein statischer Initialisierungsblock. Ein statischer Initialisierungsblock kann beliebige Anweisungen enthalten. Er wird im Rahmen der Initialisierung von Klassenvariablen genau einmal ausgeführt.

Auch für die Initialisierung von Instanzvariablen gibt es einen Initialisierungsblock. Dieser ist allerdings nicht statisch. Der nicht statische Initialisierungsblock wird jedes Mal dann ausgeführt, wenn ein Objekt dieser Klasse angelegt wird. Für die Initiali-

sierung der Objekte anonymer Klassen wird der nicht statische Initialisierungsblock benötigt, da anonyme Klassen keinen Konstruktor besitzen.

Zuerst werden die Default-Initialisierungen durchgeführt. Manuelle Initialisierungen und Initialisierungen mit einem Initialisierungsblock werden der Reihe nach abgearbeitet und überschreiben die entsprechenden Default-Initialisierungen.

Da ein Konstruktor automatisch nach der Allokierung des Speicherplatzes für ein Objekt aufgerufen wird, wird

- in der Regel die Initialisierung eines Objekts im Konstruktor durchgeführt
- und die Anzahl der erzeugten Objekte am besten auch im Konstruktor hochgezählt.

Ein Konstruktor

- dient zum Initialisieren eines Objektes,
- unterscheidet sich von einer normalen Methode unter anderem dadurch, dass der Konstruktor ohne Rückgabetyp (auch nicht `void`) deklariert wird,
- unterscheidet sich weiterhin von einer normalen Methode auch dadurch, dass er nicht an eine abgeleitete Klasse vererbt wird,
- wird vom Compiler dadurch erkannt, dass er den gleichen Namen trägt wie die Klasse selbst und keinen Rückgabewert hat.

Die Default-Werte von Datenfeldern werden durch eine manuelle Initialisierung oder einen Initialisierungsblock überschrieben. Initialisierungen, die im Konstruktor durchgeführt werden, überschreiben sowohl Default-Werte als auch die Werte einer manuellen Initialisierung und die Werte der Initialisierungen eines Initialisierungsblocks.

Der Default-Konstruktor wird vom Compiler zur Verfügung gestellt. Er ist für jede Klasse automatisch definiert, vorausgesetzt, es wird kein Konstruktor selbst definiert. Ein Default-Konstruktor einer Klasse hat keine Parameter. Er ist ein parameterloser Konstruktor.

Ein parameterloser Konstruktor ist ein Konstruktor ohne Parameter, der dem Benutzer in der Regel[154] keine frei vorgebbare individuelle Initialisierung von Datenfeldwerten ermöglicht, da keine Parameter übergeben werden können.

Konstruktoren mit Parametern erlauben eine frei vorgebbare individuelle Initialisierung von Objekten.

Konstruktoren können genauso wie normale Methoden überladen werden. Es können beliebig viele Konstruktoren selbst geschrieben werden, die sich in Typ und Anzahl der Übergabeparameter unterscheiden.

Sobald nur ein einziger selbst geschriebener Konstruktor existiert – gleichgültig ob mit oder ohne Parameter –, ist der vom Compiler zur Verfügung gestellte Default-Konstruktor nicht mehr sichtbar.

---

[154] Eine frei vorgebbare individuelle Initialisierung kann hier lediglich dadurch erreicht werden, dass im Konstruktur der Benutzer aufgefordert wird, von der Tastatur aus individuelle Werte einzugeben.

Ein Konstruktor einer Klasse kann in seiner ersten Anweisung einen anderen Konstruktor derselben Klasse aufrufen. Mit Hilfe von `this (parameterliste)` kann aus einem Konstruktor ein anderer Konstruktor derselben Klasse aufgerufen werden. Diese Anweisung muss allerdings die erste Anweisung im Rumpf des Konstruktors sein.

Kapitel **10.5** beschreibt die Instanziierung von Klassen.

Das Erzeugen eines Objektes einer Klasse wird auch als Instanziierung oder Instanziieren einer Klasse bezeichnet. Damit soll zum Ausdruck gebracht werden, dass eine Instanz dieser Klasse geschaffen wird.

Zwei Begriffe, deren Bedeutung oft verwechselt wird, sind Instanziierung und Initialisierung:

- Bei der Instanziierung wird ein neues Objekt auf dem Heap angelegt. Die Instanziierung wird in der Regel mit dem `new`-Operator durchgeführt.
- Bei der Initialisierung eines Objekts werden die Datenfelder des erzeugten (instanziierten) Objektes mit Werten belegt. Die Initialisierung kann mit Default-Werten, mit Hilfe einer manuellen Initialisierung, mit einem nicht statischen Initialisierungsblock oder mit Hilfe des Konstruktors erfolgen

Kapitel **10.6** behandelt die Freigabe von Speicher.

Bei einer Speicherbereinigung werden die nicht referenzierten Objekte aus dem Heap entfernt. Mit anderen Worten, ihr Platz wird zum Überschreiben freigegeben. Wann die Java-Virtuelle Maschine einen Lauf des Garbage Collectors durchführt, ist Sache der Java-Virtuellen Maschine. Da weder definiert ist, wann der Garbage Collector die Objekte aus dem Speicher entfernt, noch sichergestellt ist, dass der Garbage Collector auf jeden Fall eine Speicherbereinigung vor der Beendigung eines Programms durchführt, ist generell von der Verwendung der Methode `finalize()` abzuraten.

Kapitel **10.7** stellt die Klasse `Object` vor.

Jede Klasse und jeder Array-Typ wird implizit, d. h. ohne eine explizite Angabe des Programmierers, von der Klasse `Object` abgeleitet. Damit beinhaltet jede Klasse und ein jedes Array automatisch alle Methoden der Klasse `Object`. Die Methoden der Klasse `Object` lassen sich in zwei Kategorien einteilen:

- in Methoden, die Threads unterstützen
- und in allgemeine Utility-Methoden.

10.9 Übungen

Aufgabe 10.1: Information Hiding

10.1.1 Kontovergleich

Es soll eine Klasse `Konto` entwickelt werden. Ein Konto beinhaltet den Kontostand und wird einer Person zugeordnet. Um eine Person zu beschreiben, wird die schon bekannte Klasse `Person` mit den Instanzvariablen `name` und `vorname` verwendet. Die Klasse `KontoTest` wird zum Testen der Klasse `Konto` verwendet. Sie beinhaltet neben der `main()`-Methode eine weitere Klassenmethode mit der Bezeichnung `kontoVergleich()`. Dieser Methode werden zwei Objekte der Klasse `Konto` übergeben. Die übergebenen Konten werden dann auf die Höhe des Kontostands hin verglichen und das Ergebnis auf dem Bildschirm ausgegeben. Ergänzen Sie das folgende Programmfragment. Fehlende Stellen sind durch `. . . . .` markiert.

```
// Datei: KontoTest.java
public class KontoTest
{
   public static void kontoVergleich ( . . . . . )
   {
      if ( . . . . . )
      {
         System.out.println (k1.getName()
                     + " hat mehr Geld auf dem Konto als "
                     + k2.getName());
      }
      else
      {
         System.out.println (k2.getName()
                     + " hat mehr Geld auf dem Konto als "
                     + k1.getName());
      }
   }

   public static void main (String[] args)
   {
      Konto konto1 = new Konto ("Müller", "Hans", 500);
      Konto konto2 = new Konto ("Krause", "Peter", 1500);
      Konto konto3 = new Konto ("Böhm", "Harald", 330);
      kontoVergleich (konto1, konto2);
      kontoVergleich (konto2, konto3);
      kontoVergleich (konto1, konto3);
   }
}

// Datei: Konto.java

public class Konto
{
   private Person person;
   private double kontostand;
   public Konto (String name,
                 String vorname,
                 float kontostand)
```

```
    {
        . . . . .
    }
    . . . . .
}
```

10.1.2 Die Klasse Buch

Entwickeln Sie eine Klasse `Buch`, die ein Buch durch folgende Attribute beschreibt: Titel, Autor, Verlag und Anzahl der Seiten. Alle Instanzvariablen sollen `private` sein und nur über entsprechende get- und set-Methoden zugänglich sein. Schreiben Sie zusätzlich einen Konstruktor, der es erlaubt, die Instanzvariablen benutzerdefiniert zu initialisieren.

Schreiben Sie eine Testklasse, die in ihrer `main()`-Methode mehrere Instanzen der Klasse `Buch` erzeugt und deren Inhalt auf dem Bildschirm ausgibt.

Aufgabe 10.2: Klassenvariablen und Klassenmethoden

10.2.1 Klassenvariable für die Kfz-Zulassung

a) Es soll eine Klasse für Kfz-Zulassungen erstellt werden. Schreiben Sie hierzu eine Klasse `KfzZulassung`. Die Informationen einer Kfz-Zulassung bestehen aus den beiden Datenfeldern `kennzeichen` und `fahrzeughalter`, die jeweils aus einem String bestehen und `private` sein sollen. Es soll eine Klassenvariable `anzahl` vom Typ `int` geben, welche die Anzahl der erzeugten Zulassungen zählt und `public` ist. Als Methoden sollen zur Verfügung stehen:

   1. eine Methode `print()` zur Ausgabe der beiden Datenfelder `kennzeichen` und `fahrzeughalter`,
   2. ein Konstruktor mit 2 Parametern zur Initialisierung der Datenfelder `kennzeichen` und `fahrzeughalter`
   3. und eine Methode `main()` zum Testen.

   Erzeugen Sie in der Methode `main()` zwei Objekte der Klasse `KfzZulassung` mit folgenden Werten:

   - "ES-FH 2003" und "Martin Mustermann"
   - "ES-FH 2004" und "Markus Müller"

   Die Referenz `z1` verweist auf das erste Objekt, die Referenz `z2` verweist auf das zweite Objekt. Bei jedem Erzeugen eines Objektes der Klasse `KfzZulassung` wird die Klassenvariable `anzahl` in der Methode `main()` um 1 hochgezählt. Vor und auch nach dem Erzeugen eines Objektes wird der Wert der Klassenvariablen `anzahl` am Bildschirm ausgegeben.

b) Nehmen Sie Ihre Lösung von a) und verlagern Sie die Methode `main()` in die Klasse `TestKfzZulassung`. Die Klasse `KfzZulassung` soll in die Klasse `KfzZulassung2` ohne eine Methode `main()` umgeschrieben werden. Versuchen Sie in der Methode `main()` der Klasse `TestKfzZulassung`, ob Sie über die Referenz `z1` das Kennzeichen ändern können durch

```
z1.kennzeichen = "N-EU 1111";
```

Ändern Sie den Zugriffsmodifikator des Datenfeldes `kennzeichen` von `private` auf `public` und versuchen Sie es erneut. Versuchen Sie dasselbe in der Methode `main()` der Klasse `KfzZulassung` aus Teilaufgabe a). Gibt es einen Unterschied?

c) Verbessern Sie Ihr Programm, indem Sie das Hochzählen der Anzahl der Zulassungen im Konstruktor durchführen.

10.2.2 Klassenvariablen und Klassenmethoden für Kinos

Ein Kinobesitzer möchte seine Kinosäle in einem Informationssystem halten können. Hierzu sind die Klassen `Kinosaal` und `TestKinosaal` zu entwickeln. Die Klasse `Kinosaal` besitzt folgende Eigenschaften:

- einen parameterlosen Konstruktor,
- die beiden Instanzvariablen `saalNummer` und `anzahlSitzplaetzeSaal`,
- die beiden Klassenvariablen `anzahlSitzplaetzeKino` und `anzahlKinosaele`,
- eine `get`- und `set`-Methode, um die Anzahl der Sitzplätze eines Saals auszulesen bzw. festzulegen
- und die beiden Klassenmethoden `getAnzahlSitzplaetzeKino`() sowie `getAnzahlKinosaele`().

Die Klasse `TestKinosaal` ist eine Wrapper-Klasse für die Methode `main()`. In dieser Methode soll die Klasse `Kinosaal` getestet werden. Hierzu sollen zwei Kinosäle mit 50 bzw. 100 Sitzplätzen angelegt werden. Alle Variablen sollen vom Typ `int` und `private` sein.

a) Schreiben Sie die Klasse `Kinosaal`. Bei jedem Erzeugen eines Kinosaals soll der Wert der Variablen `anzahlKinosaele` um 1 erhöht werden. Jeder Kinosaal soll beim Erzeugen eine eindeutige Nummer `saalNummer` erhalten, die direkt aus der Anzahl der Kinosäle abgeleitet wird. Mit der Methode

```
public void setAnzahlSitzplaetzeSaal (int
                                      anzahlSitzplaetzeSaal)
```

soll für einen neu erzeugten Kinosaal die Anzahl der Sitzplätze eines Saals, `anzahlSitzplaetzeSaal`, gesetzt werden. Dabei soll die Klassenvariable `anzahlSitzplaetzeKino` um den Wert der Instanzvariablen `anzahlSitzplaetzeSaal` erhöht werden.

b) Schreiben Sie die Methode `setAnzahlSitzplaetzeSaal()` aus Teilaufgabe a) so um, dass die Anzahl der Sitzplätze eines Kinosaals nachträglich geändert werden kann und die Anzahl der Sitzplätze des Kinos entsprechend angepasst wird.

Aufgabe 10.3: Selbstreferenzierung mit der `this`-Referenz

10.3.1 Verkettung von Methodenaufrufen

Wie bereits in Kapitel 10.3 erklärt wurde, können mit Hilfe der `this`-Referenz Methodenaufrufe verkettet werden. Erweitern Sie das folgende Programmfragment, sodass

die Methode zum Skalieren der Seitenlänge eines Würfels sowie die Methoden zum Berechnen der Grundfläche, der Oberfläche und des Volumens miteinander verkettet werden können. Fehlende Stellen sind durch . . . . . markiert.

```
// Datei: MethodenverkettungTest.java
public class MethodenverkettungTest
{
   static public void main (String[] args)
   {
      Wuerfel w = new Wuerfel (5.0f);

      w.berechneGrundflaeche()
       .berechneOberflaeche()
       .berechneVolumen()
       .print();
      w.skaliereSeitenlaenge (2.0f)
       .berechneGrundflaeche()
       .berechneOberflaeche()
       .berechneVolumen()
       .print();
   }
}

class Wuerfel
{
   private float seitenlaenge;
   private float grundflaeche;
   private float oberflaeche;
   private float volumen;

   public Wuerfel (float s)
   {
      this.seitenlaenge = s;
      this.grundflaeche = s * s;
      this.oberflaeche = this.grundflaeche * 6;
      this.volumen = this.grundflaeche * s;
   }

   public float getSeitenlaenge()
   {
      return this.seitenlaenge;
   }

   public float getGrundflaeche()
   {
      return this.grundflaeche;
   }

   public float getOberflaeche()
   {
      return this.oberflaeche;
   }

   public float getVolumen()
   {
      return this.volumen;
   }
```

```
   public Wuerfel setSeitenlaenge (float s)
   {
      this.seitenlaenge = s;
      . . . . .
   }

   public Wuerfel skaliereSeitenlaenge (float faktor)
   {
      this.seitenlaenge = this.seitenlaenge * faktor;
      . . . . .
   }

   public Wuerfel berechneGrundflaeche()
   {
      this.grundflaeche =
               this.seitenlaenge * this.seitenlaenge;
      . . . . .
   }

   public Wuerfel berechneOberflaeche()
   {
      this.oberflaeche =
               this.seitenlaenge * this.seitenlaenge * 6;
      . . . . .
   }

   public Wuerfel berechneVolumen()
   {
      this.volumen =
               this.seitenlaenge
               * this.seitenlaenge
               * this.seitenlaenge;
      . . . . .
   }

   public void print()
   {
      System.out.println ("Seitenlaenge: "
                          + this.seitenlaenge);
      System.out.println ("Grundflaeche: "
                          + this.grundflaeche);
      System.out.println ("Oberflaeche: "
                          + this.oberflaeche);
      System.out.println ("Volumen: "
                          + this.volumen);
   }
}
```

10.3.2 Hello World mit statischen Methoden

Im folgenden "Hello World"-Programm hat sich ein Fehler eingeschlichen. Finden und korrigieren Sie den Fehler.

Hier das Programm:

```
// Datei: HelloWorld.java

public class HelloWorld
{

   public static void main (String[] args)
   {
      this.print();
   }

   public static void print()
   {
      System.out.println ("Hello World!");
   }
}
```

Aufgabe 10.4: Flughafen-Projekt

In Kapitel 2.6 wurde das Flughafen-Projekt vorgestellt. Dieses Projekt soll nun als eine durchgehende Projektaufgabe in diesem und in den restlichen Kapiteln realisiert werden. Diese Aufgabe unterscheidet sich hierbei von den Übungen dahingehend, dass der durch die Problemlösung entstehende Quellcode einer Projektaufgabe immer als Basis für die darauf folgende Projektaufgabe dient. Hierbei wird versucht, das jeweils neu erworbene Wissen direkt in die Projektaufgaben der einzelnen Kapitel einfließen zu lassen. Auch soll bei diesen Projektaufgaben lediglich ein roter Faden vorgegeben werden. Die eigentliche Implementierung kann dabei von der Musterlösung erheblich abweichen. Die Musterlösung soll somit nur als eine mögliche Lösung dienen. Da die Realisierung des kompletten Flughafensystems den Rahmen dieses Buches sprengen würde, wird nur ein ausgesuchter Teil realisiert. Dieser Teil beschränkt sich auf die Rolle des Lotsen. Es werden dabei folgende Use Cases betrachtet:

- Zeitplan anfordern
- Landung durchführen
- Start durchführen

10.4.1 Erste Schritte mit der Klasse `Flugzeug`

Das wichtigste Objekt eines Flughafens ist das Flugzeug. Denn ohne Flugzeuge wird auch kein Flughafen benötigt. Aus diesem Grund beschäftigt sich diese Aufgabe damit, eine erste Version der Klasse `Flugzeug` zu erstellen. Diese Klasse soll dabei den Vorgang der Landung und den darauf folgenden Start eines Flugzeugs auf sehr einfache Weise ermöglichen.

Die Klasse `Flugzeug` soll folgende **Instanzvariablen** besitzen (wählen Sie jeweils einen Ihrer Meinung nach passenden Typ für die einzelnen Instanzvariablen):

- `fluggesellschaft` ist der Name der Fluggesellschaft, welcher das Flugzeug gehört,
- `flugnummer` kennzeichnet das Flugzeug durch eine eindeutige Flugnummer,
- `flugzeugtyp` gibt den Typ des Flugzeugs an,
- `istzeitLandung` ist die Istzeit, zu der das Flugzeug gelandet ist,
- `istzeitStart` ist die Istzeit, zu der das Flugzeug gestartet ist,

- `landebahn` ist die Bahn, welche dem Flugzeug zur Landung zugeteilt wird,
- `parkstelle` gibt an, wo das Flugzeug nach der Landung parken soll. Eine Parkstelle kann eine Parkposition, die Werft oder ein separates Parkfeld sein,
- `sollzeitLandung` ist die Sollzeit, zu der das Flugzeug wahrscheinlich landet,
- `sollzeitStart` ist die Sollzeit, zu der das Flugzeug wahrscheinlich startet,
- `startbahn` ist die Bahn, welche vom Flugzeug beim Start verwendet wird,
- `status` gibt den aktuellen Status des Flugzeugs an. Mögliche Werte des Status sind: Wartend, Landeanflug, Gelandet, Geparkt, Startvorbereitung und Gestartet.

Des Weiteren soll die Klasse `Flugzeug` eine **Klassenvariable** `anzahlFlugzeuge` besitzen. Diese Klassenvariable soll die Anzahl der erzeugten Flugzeuge zählen.

Die Landung lässt sich vorläufig in die folgenden vier Phasen einteilen:

- Neues Flugzeug melden

  Die Klasse soll hierzu einen Konstruktor mit drei Übergabeparametern besitzen. Diese drei Übergabeparameter beschreiben den Typ des Flugzeugs, die Fluggesellschaft, welcher das Flugzeug gehört, sowie die Sollzeit der Landung. Innerhalb des Konstruktors soll auch eine eindeutige Flugnummer generiert werden. Der Status des Flugzeugs soll auf "Wartend" gesetzt werden.

- Landebahn vergeben

  Hierfür soll eine Methode `vergebeLandebahn()` zuständig sein. Ihr soll die zu belegende Landebahn übergeben werden. Außerdem soll der Status des Flugzeugs auf "Landeanflug" gesetzt werden[155].

- Parkstelle vergeben

  Der Methode `vergebeParkstelle()` soll die zu belegende Parkstelle übergeben werden. Zusätzlich soll der Status auf "Gelandet" gesetzt werden.

- Erfolgreiche Landung melden

  Hierzu soll eine weitere Methode `meldeGelandet()` implementiert werden. Ihr wird die Istzeit der Landung übergeben. Des Weiteren soll der Status des Flugzeugs auf "Geparkt" gesetzt werden.

Für den Start können derzeit folgende zwei Phasen definiert werden:

- Startbahn vergeben

  Die Methode `vergebeStartbahn()` setzt die zu verwendende Startbahn, die erwartete Sollzeit für den Start und den Status auf "Startvorbereitung".

- Erfolgreichen Start melden

  Der Methode `meldeGestartet()` wird die Istzeit des Starts übergeben. Der Status wird auf "Gestartet" gesetzt.

[155] Bitte beachten Sie, dass die Projektaufgabe hier von den Kundenanforderungen abweicht. Ein Abbruch der Landung muss bis zum Aufsetzen des Flugzeugs in der Realität möglich sein.

Als letztes soll der Klasse Flugzeug eine Methode `print()` hinzugefügt werden. Diese Methode dient dazu, den aktuellen Zustand eines Flugzeuges auf dem Bildschirm auszugeben. Eine Darstellung einer möglichen Klasse ist im nächsten Bild zu sehen. Ein – in der Grafik bedeutet hierbei `private`, ein + bedeutet `public`. Das folgende Bild visualisiert die Klasse `Flugzeug`:

Schreiben Sie zum Testen der Klasse `Flugzeug` eine Klasse namens `Client`. Ein Programmablauf könnte wie folgend aussehen:

```
Flugzeug MI 101 befindet sich im Status 'wartend'.
Es ist vom Typ Bowling 474 und gehört der Fluggesellschaft Microair.

Flugzeug MI 101 befindet sich im Status 'Landeanflug'.
Es ist vom Typ Bowling 474 und gehört der Fluggesellschaft Microair.
Es wird verwendet/wurde verwendet:
- Landebahn 1

Flugzeug MI 101 befindet sich im Status 'Gelandet'.
Es ist vom Typ Bowling 474 und gehört der Fluggesellschaft Microair.
Es wird verwendet/wurde verwendet:
- Landebahn 1
- Parkposition 7

Flugzeug MI 101 befindet sich im Status 'Geparkt'.
Es ist vom Typ Bowling 474 und gehört der Fluggesellschaft Microair.
Es wird verwendet/wurde verwendet:
- Landebahn 1
- Parkposition 7

Flugzeug MI 101 befindet sich im Status 'Startvorbereitung'.
Es ist vom Typ Bowling 474 und gehört der Fluggesellschaft Microair.
Es wird verwendet/wurde verwendet:
```

```
- Landebahn 1
- Parkposition 7
- Startbahn 2

Flugzeug MI 101 befindet sich im Status 'Gestartet'.
Es ist vom Typ Bowling 474 und gehört der Fluggesellschaft Microair.
Es wird verwendet/wurde verwendet:

- Landebahn 1
- Parkposition 7
- Startbahn 2
```

10.4.2 Benutzereingabe

In Projektaufgabe 10.4.1 wurde ein Flugzeug mit fest vorgegebenen Werten initialisiert. Teil dieser Projektaufgabe soll es sein, die für die Landung und den Start notwendigen Informationen interaktiv vom Benutzer abzufragen. Ändern Sie hierzu die Klasse `Client` ab. Ein möglicher Dialog könnte wie folgend aussehen:

```
Geben Sie den Flugzeug Typ ein: Bowling 474
Geben Sie die Fluggesellschaft ein: Microwing
Geben Sie die Sollzeit der Landung ein: 7:49

Flugzeug MI 101 befindet sich im Status 'wartend'.
Es ist vom Typ Bowling 474 und gehört der Fluggesellschaft
Microwing.

Geben Sie die Landebahnnummer ein: 3

Flugzeug MI 101 befindet sich im Status 'Landeanflug'.
Es ist vom Typ Bowling 474 und gehört der Fluggesellschaft
Microwing.
Es wird verwendet/wurde verwendet:
- Landebahn 3

Soll das Flugzeug in die Werft verlegt werden? (j/n) n
Soll das Flugzeug auf ein separates Parkfeld verlegt werden? (j/n) n

Geben Sie die Parkposition ein: 7

Flugzeug MI 101 befindet sich im Status 'Gelandet'.
Es ist vom Typ Bowling 474 und gehört der Fluggesellschaft
Microwing.
Es wird verwendet/wurde verwendet:
- Landebahn 3
- Parkposition 7

Geben Sie die Istzeit der Landung ein: 7:53

Flugzeug MI 101 befindet sich im Status 'Geparkt'.
Es ist vom Typ Bowling 474 und gehört der Fluggesellschaft
Microwing.
Es wird verwendet/wurde verwendet:
- Landebahn 3
- Parkposition 7
```

```
Geben Sie die Startbahn ein: 1
Geben Sie die Sollzeit für den Start ein: 9:13

Flugzeug MI 101 befindet sich im Status 'Startvorbereitung'.
Es ist vom Typ Bowling 474 und gehört der Fluggesellschaft
Microwing.
Es wird verwendet/wurde verwendet:
- Landebahn 3
- Parkposition 7
- Startbahn 1

Geben Sie die Istzeit des Starts ein: 9:12

Flugzeug MI 101 befindet sich im Status 'Gestartet'.
Es ist vom Typ Bowling 474 und gehört der Fluggesellschaft
Microwing.
Es wird verwendet/wurde verwendet:
- Landebahn 3
- Parkposition 7
- Startbahn 1
```

Für die Eingabe von der Tastatur können Sie folgende Klasse `Abfrage` verwenden:

```
// Datei: Abfrage.java
public class Abfrage
{
   public static int abfrageInt (String frage)
   {
      try
      {
         int zahl = Integer.parseInt (abfrageString (frage));
         if (zahl < 0)
            throw new NumberFormatException ("");
         return zahl;
      }
      catch (NumberFormatException e)
      {
         System.out.println ("Bitte eine gültige Zahl eingeben");
         return abfrageInt (frage);
      }
   }

   public static String abfrageString (String frage)
   {
      try
      {
         System.out.print (frage + " ");
         java.util.Scanner eingabe =
                     new java.util.Scanner (System.in);
         return eingabe.nextLine();
      }
      catch (Exception e)
      {
         return "";
      }
   }
}
```

Kapitel 11

Vererbung und Polymorphie

11.1 Das Konzept der Vererbung
11.2 Erweitern und Überschreiben
11.3 Besonderheiten bei der Vererbung
11.4 Polymorphie und das liskovsche Substitutionsprinzip
11.5 Verträge
11.6 Identifikation der Klasse eines Objektes
11.7 Operatoren für Referenztypen
11.8 Konsistenzhaltung von Quell- und Bytecode
11.9 Kovarianz und Kontravarianz beim Überschreiben
11.10 Zusammenfassung
11.11 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_11

11 Vererbung und Polymorphie

Neben der Aggregation ist die Vererbung ein wesentliches Sprachmittel der objektorientierten Programmiersprachen für die Wiederverwendung von Programmcode:

- Bei der **Vererbung** wird in Java der Quellcode einer einzigen Superklasse in einer abgeleiteten Klasse wiederverwendet.
- Im Falle einer **Aggregation** werden vorhandene einzelne Klassen von den aggregierenden Klassen benutzt und damit wiederverwendet.

Besonders effizient ist die **Wiederverwendung**, wenn **ganze Programmsysteme polymorph verwendet werden können**. Dies ist der Fall, wenn ein Programm Referenzen auf Objekte von Basisklassen enthält und dieses Programm auch läuft, wenn Objekte abgeleiteter Klassen an die Stelle der Objekte der Basisklassen treten. Dies ist der Fall, wenn das Liskovsche Substitutionsprinzip (siehe Kapitel 11.4) gilt.

Vererbung und Aggregation erlauben nur die Wiederverwendung einzelner Klassen. **Polymorphie hingegen** erlaubt die **Wiederverwendung ganzer Programmsysteme** aus vielen Klassen.

In Kapitel 11.1 wird das Konzept der Vererbung vorgestellt. Anschließend wird in Kapitel 11.2 auf das Erweitern und Überschreiben von geerbten Methoden einer Basisklasse eingegangen. Kapitel 11.3 beschreibt Besonderheiten bei der Vererbung sowie Kapitel 11.4 die Polymorphie in Klassenhierarchien und das liskovsche Substitutionsprinzip. In Kapitel 11.5 werden Verträge von Klassen behandelt. Wie die Identifikation der Klasse eines Objekts funktioniert, wird in Kapitel 11.6 erklärt. Kapitel 11.7 fasst Operatoren für Referenztypen zusammen. Es folgen in Kapitel 11.8 Betrachtungen zur Konsistenzhaltung von Quell- und Bytecode. Im Anschluss daran werden in Kapitel 11.9 die Eigenschaften der Kovarianz und Kontravarianz beim Überschreiben von Methoden untersucht.

11.1 Das Konzept der Vererbung

Bei der Vererbung erbt eine abgeleitete Klasse alle **Eigenschaften** (Datenfelder und Methoden – also die Struktur und das Verhalten) ihrer Basisklasse und fügt ihre eigenen individuellen Eigenschaften hinzu oder überschreibt Methoden der Basisklasse. Die Eigenschaften der Basisklasse müssen nicht in der Spezifikation der abgeleiteten Klasse wiederholt werden. Man sagt, die **abgeleitete Klasse** wird **von** ihrer **Basisklasse abgeleitet**.

Am einfachsten soll dies anhand des Beispiels eines Studenten erläutert werden. Ein Student ist bekanntermaßen eine Person, die studiert. Wenn man studieren möchte, muss man immatrikuliert werden und erhält eine Matrikelnummer. Kurz, wer eine Matrikelnummer hat, ist eingeschrieben und ist somit ein Student. Also kann man einen Studenten beschreiben als eine Person, die eine Matrikelnummer hat. Im Folgenden erfolgt die Visualisierung dieser Vererbungsbeziehung:

Bild 11-1 Ableitung der Klasse `Student` von der Klasse `Person`

`matrikelnummer` ist ein zusätzliches Datenfeld, `setMatrikelnummer()` und `printMatrikelnummer()` sind zusätzliche Methoden der abgeleiteten Klasse.

Zu beachten ist, dass der Ableitungspfeil von der **Subklasse** (**abgeleiteten Klasse**, **Unterklasse**, **Sohnklasse**) zu der **Superklasse** (**Basisklasse**, **Oberklasse**, **Vaterklasse**) zeigt. Eine Superklasse merkt nicht, dass von ihr abgeleitet wird. Sie wird beim Vererben nicht aktiv.

Gibt es mehrere Hierarchieebenen der Vererbung, so wird als **Superklasse** oder **Basisklasse** bzw. **Oberklasse** einer Klasse eine an einer beliebigen höheren Stelle des Vererbungspfades stehende Klasse, als **Subklasse** oder **abgeleitete Klasse** bzw. **Unterklasse** einer Klasse eine an einer beliebigen tieferen Stelle des Vererbungspfades liegende Klasse bezeichnet.

Vater- und **Sohnklasse** heißen zwei Klassen, die in zwei direkt übereinander liegenden Ebenen eines Vererbungspfades angeordnet sind.

Die oberste Klasse eines Klassenbaumes wird **Wurzelklasse** oder **Rootklasse** genannt.

11.1.1 Die Klasse `Student` als abgeleitete Klasse

Ein Student **ist eine** Person (**"is a"-Beziehung**). Damit kann man den Studenten durch eine Vererbungsbeziehung von Person ableiten. Dies bedeutet, dass infolge der Vererbungsbeziehung jedes Objekt der Klasse `Student` automatisch alle Instanzvariablen besitzt, die auch ein Objekt der Klasse `Person` hat. Genauso verfügt

ein Objekt der Klasse `Student` auch über alle Methoden der Klasse `Person`. Dass ein Student eine spezielle Person ist, kommt dadurch zum Ausdruck, dass er zusätzliche Eigenschaften gegenüber einer normalen Person hat, nämlich die Matrikelnummer und die ihr zugeordneten Methoden. Das folgende Bild zeigt die Eigenschaften der Objekte der Klasse `Person` und `Student`:

Bild 11-2 Eigenschaften von Objekten der Klasse `Person` und der Klasse `Student`[156]

Bild 11-3 zeigt ein konkretes Objekt der Klasse `Person` und ein konkretes Objekt der Klasse `Student` nach UML mit den zugeordneten Datenfeldern:

Bild 11-3 Objekt `Müller` der Klasse `Person` und Objekt `Maier` der Klasse `Student`

Dieses **Beispiel** für eine Vererbung kann als Java-Programm – wie im Folgenden gezeigt – realisiert werden. Beachten Sie die fett gedruckten Teile. Hier das Programm:

```
// Datei: Person.java

public class Person
{
   private String nachname;
   private String vorname;

   public void setNachname (String nachname)
   {
      this.nachname = nachname;
   }

   public void setVorname (String vorname)
   {
      this.vorname = vorname;
   }
```

[156] nicht nach UML

```
   public void print()
   {
      System.out.println ("Nachname: " + nachname);
      System.out.println ("Vorname: " + vorname);
   }
}

// Datei: Student.java
public class Student extends Person  // die Klasse Student wird von
{                                    // der Klasse Person abgeleitet
   private int matrikelnummer;

   // Methoden der Klasse Student
   public void setMatrikelnummer (int matrikelnummer)
   {
      this.matrikelnummer = matrikelnummer;
   }

   public void printMatrikelnummer()
   {
      System.out.println ("Matrikelnummer: " + matrikelnummer);
   }
}

// Datei: TestVererbung.java
public class TestVererbung
{
   public static void main (String[] args)
   {
      System.out.println ("Student");
      Student studiosus = new Student();
      studiosus.setNachname ("Maier");
      studiosus.setVorname ("Fritz");
      studiosus.setMatrikelnummer (56123);
      studiosus.print();
      studiosus.printMatrikelnummer();

      System.out.println ("Person");
      Person pers = new Person();
      pers.setNachname ("Müller");
      pers.setVorname ("Peter");
      pers.print();
   }
}
```

Die Ausgabe des Programms ist:

```
Student
Nachname: Maier
Vorname: Fritz
Matrikelnummer: 56123
Person
Nachname: Müller
Vorname: Peter
```

Das folgende Bild visualisiert die Methodenaufrufe:

Bild 11-4 Visualisierung der Methodenaufrufe der Methode `main()`

In der Klassenmethode `main()` werden zuerst die Methoden `setNachname()`, `setVorname()`, `setMatrikelnummer()`, `print()` und `printMatrikelnummer()` eines Objekts der Klasse `Student` aufgerufen. Danach werden die Methoden `setNachname()`, `setVorname()` und `print()` eines Objekts der Klasse `Person` aufgerufen.

11.1.2 Hierarchien von Klassen

Das Konzept der Vererbung erlaubt es, dass eine Klasse, die von einer anderen Klasse abgeleitet wird, automatisch alle Eigenschaften (Datenfelder und Methoden) dieser anderen Klasse erhält, ohne diese explizit anschreiben zu müssen. Eine abgeleitete Klasse stellt eine **Spezialisierung** ihrer Vaterklasse dar. In der abgeleiteten Klasse sind nur die neuen spezifischen und zusätzlichen Eigenschaften festzulegen. Umgekehrt stellt natürlich eine Vaterklasse eine **Generalisierung** ihrer abgeleiteten Klassen dar. Dies zeigt Bild 11-5:

Bild 11-5 Vererbungshierarchie mit Generalisierung und Spezialisierung

Mit dem **Konzept der Vererbung** können **Wiederholungen im Entwurf vermieden** werden.

Gemeinsame Eigenschaften mehrerer Klassen werden in gemeinsame Oberklassen ausgelagert. Dies führt zu mehr Übersicht und zu weniger Wiederholung. Beim Entwurf werden also Datenfelder und Methoden, die mehreren Klassen gemeinsam sind, nach oben in der Klassenhierarchie geschoben, da sie dann automatisch durch Vererbung wieder zu den abgeleiteten Klassen weitergegeben werden. Bild 11-6 zeigt ein Beispiel für die **Generalisierung** und die **Spezialisierung** in einer Klassenhierarchie:

Bild 11-6 Beispiel für eine Klassenhierarchie

So ist ein Netzwerk-Rechner ein Rechner, der mit Hilfe einer Kommunikationsschnittstelle Daten übertragen kann. Ein Client-Rechner wiederum ist ein Rechnertyp, der von Server-Rechnern Dienstleistungen erbringen lassen kann, um deren Ergebnisse darzustellen. Das heißt, ein Client-Rechner verfügt gegenüber einem Netzwerk-Rechner über die zusätzliche Methode `ausgabenDarstellen()`. Ein Server-Rechner hat die zusätzliche Methode `dienstleistungenErbringen()`. Die abgeleiteten Klassen stellen hier also Erweiterungen ihrer Basisklassen dar.

11.2 Erweitern und Überschreiben

Neben der Erweiterung einer Basisklasse in einer abgeleiteten Klasse um zusätzliche Attribute und Methoden können auch Methoden einer Basisklasse in einer

abgeleiteten Klasse überschrieben werden, um eine Spezialisierung zu erreichen. Das Erweitern und Überschreiben wird in Kapitel 11.2.1 und 11.2.2 erläutert.

11.2.1 Erweitern

Wie in Kapitel 11.1 am Beispiel der Klassen `Person` und `Student` vorgestellt wurde, erweitert die Klasse `Student` die Klasse `Person`. So erhält die Klasse `Student` gegenüber der Klasse `Person` das zusätzliche Datenfeld `matrikelnummer` und die zusätzliche Aufrufschnittstelle aus den beiden Methoden `setMatrikelnummer()` und `printMatrikelnummer()`. Die Methoden und Datenfelder der Basisklasse werden im Rahmen der Vererbung von der abgeleiteten Klasse unverändert übernommen. Die Klassen `Person` und `Student` sind direkt benachbart und befinden sich in einem Vater-Sohn-Verhältnis.

Bei der reinen Erweiterung kommt in der **Sohnklasse** zur Aufrufschnittstelle der **Vaterklasse**, die geerbt wird, noch eine für die Sohnklasse spezifische Aufrufschnittstelle hinzu. Ein Objekt der Klasse `Sohn` besitzt als Aufrufschnittstelle

- sowohl die Aufrufschnittstelle der Vaterklasse
- als auch die zusätzliche Aufrufschnittstelle der Sohnklasse.

Das folgende Bild zeigt den Fall der reinen Erweiterung beim Ableiten von einer Basisklasse:

Bild 11-7 Zusätzliche Aufrufschnittstelle eines Objektes einer abgeleiteten Klasse

Es werden sowohl Instanzvariablen und Instanzmethoden, als auch Klassenvariablen und Klassenmethoden vererbt. Eine Sohnklasse erbt also grundsätzlich alles – auch private Datenfelder und Methoden. Dabei ist aber nicht alles, das geerbt wurde, in der Sohnklasse automatisch sichtbar und damit zugreifbar. Die Vaterklasse kann die Sichtbarkeit ihrer Datenfelder und Methoden über Zugriffsmodifikatoren steuern. Die in der Vaterklasse festgelegten Zugriffsmodifikatoren haben auch Konsequenzen bei der Vererbung (siehe Kapitel 12.7).

11.2.2 Überschreiben von Methoden

Enthält eine abgeleitete Klasse eine Methode mit gleicher Signatur und mit gleichem Rückgabetyp wie eine Methode einer Basisklasse, so sagt man, dass die Methode der Basisklasse durch die Methode der abgeleiteten Klasse überschrieben wurde.

Dies bedeutet, dass in der abgeleiteten Klasse nur die überschreibende Methode direkt sichtbar ist.

Beim Überschreiben von Methoden muss die Signatur der Methoden gleich sein, d. h. die formalen Parameter der überschreibenden Methode müssen die gleichen Typen, die gleiche Anzahl und Reihenfolge wie die der überschriebenen Methode haben. Zusätzlich muss auch der Typ des Rückgabewerts übereinstimmen.

Da Objekte abgeleiteter Klassen polymorph sind und ein Objekt einer abgeleiteten Klasse auch vom Typ der Basisklasse ist, kann es aufgrund der Polymorphie zur sogenannten **Kovarianz** und **Kontravarianz** kommen. Darauf wird in Kapitel 11.9 eingegangen.

Im folgenden Bild wird das Überschreiben der Methode `print()` gezeigt:

Bild 11-8 Überschreiben der Methode `print()`

Die Methode `print()` der Klasse `Person2` gibt die Datenfelder `nachname` und `vorname` aus. Die Methode `print()` der Klasse `Student2` hingegen sorgt dafür, dass die genannten Datenfelder und zusätzlich noch das Datenfeld `matrikelnummer` ausgegeben wird, wie das folgende Beispiel zeigt:

```
// Datei: Person2.java
public class Person2
{
   protected String nachname;
   protected String vorname;
```

```
   public Person2 (String vorname, String nachname)
   {
      this.nachname = nachname;
      this.vorname = vorname;
   }

   public void print()
   {
      System.out.println ("Nachname: " + nachname);
      System.out.println ("Vorname: " + vorname);
   }
}

// Datei: Student2.java
public class Student2 extends Person2
{
   private int matrikelnummer;

   public Student2 (String vorname, String nachname,
                    int matrikelnummer)
   {
      super (vorname, nachname);
      this.matrikelnummer = matrikelnummer;
   }

   public void print()
   {
      System.out.println ("Nachname: " + nachname);
      System.out.println ("Vorname: " + vorname);
      System.out.println ("Matr. Nr: " + matrikelnummer);
   }
}

// Datei: Test2.java
public class Test2
{
   public static void main (String[] args)
   {
      Person2 p1 = new Person2 ("Rainer", "Brang");
      System.out.println ("\nAusgabe der Person: ");
      p1.print();
      Student2 s1 = new Student2 ("Karl", "Klug", 123456);
      System.out.println ("\nAusgabe des Studenten: ");
      s1.print();
   }
}
```

Die Ausgabe des Programms ist:

```
Ausgabe der Person:
Nachname: Brang
Vorname: Rainer

Ausgabe des Studenten:
Nachname: Klug
Vorname: Karl
Matr. Nr: 123456
```

Hierbei ist zu beachten, dass der Ausdruck `super (vorname, nachname)` im Konstruktor der Klasse `Student2` den Konstruktor

```
public Person2 (String vorname, String nachname)
```

der Vaterklasse `Person2` aufruft.[157]

In der `main()`-Methode der Klasse `Test2` wird zuerst ein neues Objekt der Klasse `Person2` erzeugt und dann dessen `print()`-Methode aufgerufen. Wie nicht anders zu erwarten, gibt diese `print()`-Methode den Nachnamen und Vornamen des Objektes der Klasse `Person2` aus. Danach wird ein Objekt der Klasse `Student2` erzeugt und dessen `print()`-Methode aufgerufen. Betrachtet man die Ausgabe des Programms, so stellt man fest, dass nun die überschreibende, den Bedürfnissen der Klasse `Student2` angepasste Methode `print()` aufgerufen wird, die zusätzlich noch die Matrikelnummer ausgibt.

Private Methoden einer Basisklasse **können** – da sie in einer abgeleiteten Klasse gar nicht sichtbar sind – **nicht überschrieben werden**. Es kann deshalb in einer Klasse eine Methode geben, welche die gleiche Signatur und den gleichen Rückgabetyp wie eine private Methode einer Oberklasse hat, ohne dass ein Überschreiben dabei stattfindet. Es handelt sich um eine **Neudefinition**.

Gründe für das Überschreiben einer Methode können sein:

- Überschreiben zur **Verfeinerung**

  Dieser Fall wurde soeben besprochen. Beispielsweise verfeinert die Klasse `Student2` die Klasse `Person2`. Die Methode `print()` der Klasse `Person2` kann die im Rahmen der Verfeinerung hinzugefügten zusätzlichen Datenfelder der Klasse `Student2` nicht kennen. Daher muss diese Methode in der Klasse `Student2` überschrieben werden.

- Überschreiben zur **Optimierung**

  Es kann nützlich sein, in einer abgeleiteten Klasse interne Datenstrukturen oder die Implementierung eines Algorithmus zu optimieren. Das Außenverhalten der Klasse darf sich dabei jedoch nicht ändern.

Überschriebene Instanzmethoden können in der überschreibenden Instanzmethode aufgerufen werden.

Im vorigen Beispiel wurde die Methode `print()` in der Klasse `Student2` komplett neu geschrieben. Es wurden die Ausgaben von `name` und `vorname` wieder vollständig neu programmiert. Dies ist aber unnötig, da man zur Ausgabe von `name` und `vorname` die von der Klasse `Person2` ererbte `print()`-Methode verwenden kann. Der Zugriff auf ein Element einer Basisklasse erfolgt mit Hilfe des Schlüssel-

[157] In Kapitel 11.3.2 wird auf diesen Sachverhalt ausführlich eingegangen.

wortes `super`. Im folgenden Beispiel wurde die Klasse `Student2` so verändert, dass sie die Methode `print()` der Klasse `Person2` für die Ausgabe von Name und Vorname verwendet:

```
// Datei: Student3.java
public class Student3 extends Person2
{
   private int matrikelnummer;

   public Student3 (String vorname, String nachname,
                    int matrikelnummer)
   {
      super (vorname, nachname); // Aufruf des Konstruktors der
                                 // Basisklasse (siehe auch Kapitel
                                 // 11.3.2)
      this.matrikelnummer = matrikelnummer;
   }

   public void print()
   {
      super.print();
      System.out.println ("Matr. Nr: " + matrikelnummer);
   }
}

// Datei: Test3.java
public class Test3
{
   public static void main (String args[])
   {
      Person2 p1 = new Person2 ("Rainer", "Brang");
      System.out.println ("\nAusgabe der Person:");
      p1.print();

      Student3 s1 = new Student3 ("Karl", "Klug", 123456);
      System.out.println ("\nAusgabe des Studenten:");
      s1.print();
   }
}
```

Die Ausgabe des Programms ist:

```
Ausgabe der Person:
Nachname: Brang
Vorname: Rainer

Ausgabe des Studenten:
Nachname: Klug
Vorname: Karl
Matr. Nr: 123456
```

Überschriebene Instanzmethoden einer Basisklasse können mit Hilfe von **`super`** innerhalb von **Instanzmethoden** einer abgeleiteten Klasse angesprochen werden. Der Compiler setzt nämlich den Aufruf `super.methode()` in `((Vaterklasse) this).methode()` um.

`this` ist bekanntermaßen eine Referenz auf das aktuelle Objekt.

Das modifizierte Beispiel für die überschreibende Methode `print()` ist kleiner und verwendet die Methode `print()` der Klasse `Person2` in der Methode `print()` der Klasse `Student3`. Dass der Codeumfang hier reduziert wird, ist relativ belanglos. Wichtiger ist, dass der Code robuster wird, da Änderungen an der `print()`-Methode der Vaterklasse automatisch auch in der Sohnklasse wirksam werden.

Wird eine Methode mit Hilfe von `super.methode()` in einer Vererbungshierarchie angesprochen, so werden ausgehend von der aktuellen Klasse alle darüber liegenden Klassen des Vererbungsbaums der Reihe nach solange durchsucht, bis zum ersten Mal eine Instanzmethode `methode()` gefunden wird.

Ein Aufruf einer Instanzmethode einer Großvaterklasse ist deshalb mit `super.methodenname()` nur dann möglich, wenn diese Methode in der Vaterklasse nicht überschrieben wurde. Einen Aufruf `super.super`, um vom Sohn zum Großvater zu gelangen, gibt es nicht.

Überschriebene Klassenmethoden einer bekannten Basisklasse werden einfach durch Angabe des Klassennamens mit `Klassenname.methode()` aufgerufen.

11.2.3 Statische und dynamische Bindung von Methoden

Unter dem Begriff der **Bindung** versteht man die Zuordnung eines Methodenrumpfes zu einem aufgerufenen Methodenkopf.

Wird eine Methode aufgerufen, so ist der entsprechende Programmcode der Methode – das heißt der Methodenrumpf – auszuführen. Das folgende Bild visualisiert die Zuordnung eines Methodenrumpfes zum Methodenkopf:

Bild 11-9 Zuordnung des Methodenrumpfes zum Methodenkopf

In der Objektorientierung gibt es zwei prinzipiell verschiedene Arten des Bindens von Methoden in Programmen. Es gibt

- die **frühe Bindung**
- und die **späte Bindung**.

Statt des Begriffs "frühe Bindung" ist auch der Begriff "**statische Bindung"** üblich und genauso anstelle von "später Bindung" der Begriff "**dynamische Bindung"**.

Bei der **frühen Bindung** kann einem Methodenaufruf schon **zum Kompilierzeitpunkt** der entsprechende **Methodenrumpf zugeordnet** werden. Bei der **späten Bindung** wird dagegen einem Methodenaufruf **erst zur Laufzeit** der entsprechende **Methodenrumpf** zugeordnet.

Die folgende Tabelle fasst die Bindungsarten zusammen:

Bindung	
statisch == früh == zur Kompilierzeit	dynamisch == spät == zur Laufzeit

Tabelle 11-1 Die beiden Bindungsarten

In Java hat man keinen direkten Einfluss darauf, ob spät oder früh gebunden wird. Java verwendet in der Regel die späte Bindung, in wenigen spezifizierten Ausnahmefällen jedoch die frühe Bindung.

11.2.3.1 Dynamische Bindung bei Instanzmethoden

Besteht ein Programm aus Referenzen auf Objekte einer Basisklasse, so kann an jeder Stelle des Programms, bei der ein Objekt einer Basisklasse verlangt wird, auch ein Objekt einer Klasse stehen, die von dieser Basisklasse abgeleitet ist, wenn der Vertrag der Basisklasse nicht gebrochen wird (siehe Kapitel 11.4).[158].

Zeigt eine Referenz vom Typ einer Basisklasse auf ein Objekt vom Typ einer Subklasse, so wird im Falle des Überschreibens von Instanzmethoden, die weder `final` noch `private` sein dürfen, die überschreibende Methode der Subklasse aufgerufen.

Es ist folglich die Aufgabe des Interpreters, zur Laufzeit festzustellen, von welchem Typ ein Objekt ist, und daraufhin die entsprechende Methode aufzurufen. Man sagt auch, dass die Methode **dynamisch oder spät gebunden** wird.

11.2.3.2 Bindung bei Klassenmethoden

Ist eine Methode `static`, handelt es sich um eine **Klassenmethode**, also eine Methode, die exakt zu einer einzigen Klasse gehört. Da sie direkt zu einer Klasse gehört, macht eine dynamische Bindung wie beim Überschreiben einer Instanzmethode für eine Klassenmethode keinen Sinn, da es bei Klassenmethoden gar keine Auswahl gibt.

[158] Man spricht dann von Polymorphie. Siehe auch Kapitel 11.4

Es ist bei Klassenmethoden stets eindeutig, welche Methode aufgerufen werden muss.

Wird nun eine Klassenmethode über eine Referenz auf ein Objekt oder über den Klassennamen aufgerufen, so wird dieser Aufruf vom Compiler direkt an die Klasse, von deren Typ die Referenz ist bzw. deren Klassennamen angegeben wird, gebunden. Man sagt auch, sie wird **statisch gebunden**. Der Compiler weiß, welche Methode er aufrufen muss.

Klassenmethoden werden nicht dynamisch gebunden, sie werden **statisch gebunden**.

Im nächsten **Beispiel** ist die Methode `print()` der Klassen `Vater4` und `Sohn4` eine Klassenmethode und damit `static`. Wie oben schon beschrieben, werden Aufrufe von Klassenmethoden statisch gebunden. Dies bedeutet, dass in beiden Fällen die Klassenmethode direkt aufgerufen wird. Hier das erwähnte Programm:

```
// Datei: Vater4.java
public class Vater4
{
   public static void print()
   {
      System.out.println ("static print()-Methode des Vaters");
   }
}

// Datei: Sohn4.java
public class Sohn4 extends Vater4
{
   public static void print()
   {
      System.out.println ("static print()-Methode des Sohns");
   }
}

// Datei: Test9.java
public class Test9
{
   public static void main (String[] args)
   {
      Sohn4 s = new Sohn4();
      System.out.print ("s.print():      ");
      s.print();
      System.out.print ("Sohn4.print():  ");
      Sohn4.print();

      Vater4 v = s; // Impliziter Cast auf den Vater
      System.out.print ("v.print():      ");
      v.print();
      System.out.print ("Vater4.print(): ");
      Vater4.print();
```

```
    }
}
```

Die Ausgabe des Programms ist:

```
s.print():       static print()-Methode des Sohns
Sohn4.print():   static print()-Methode des Sohns
v.print():       static print()-Methode des Vaters
Vater4.print():  static print()-Methode des Vaters
```

Bei **Klassenmethoden** bestimmt der Typ der Referenz bzw. der Name der Klasse, welche Klassenmethode der Klassenhierarchie aufgerufen wird. Die Bindung erfolgt statisch d. h. zur Kompilierzeit.

11.2.3.3 Bindung bei privaten Methoden

Ist eine Methode `private`, handelt es sich um eine Methode, die nur innerhalb der Klasse sichtbar ist, in der sie definiert wird. Wird von einer Klasse abgeleitet, so werden **Methoden, die `private` sind, zwar** an die Sohnklasse **weitervererbt**, es kann aber nicht innerhalb des Sohnes darauf zugegriffen werden.

Da eine `private` Methode außerhalb der Klasse, in der sie definiert wurde, zu keiner Zeit sichtbar ist, kann ein Aufruf der Methode nur innerhalb der Klasse erfolgen, in der sie definiert wurde.

Für den Compiler ist es also bereits zur Zeit der Übersetzung des Quellcodes klar, dass er den Aufruf einer mit `private` gekennzeichneten Methode statisch zur aktuellen Klasse binden kann.

Methoden, **die `private` sind**, werden wie Klassenmethoden **früh gebunden**.

11.2.3.4 Bindung bei finalen Methoden

Ist eine **Methode** mit dem **Schlüsselwort `final` gekennzeichnet**, so kann sie niemals von einer abgeleiteten Klasse überschrieben werden.

Wird eine Methode, die mit `final` gekennzeichnet ist, aufgerufen, so kann ein Compiler feststellen, zu welcher Klasse die Methode tatsächlich gehört.

Der Aufruf einer `final`-Methode kann wie zuvor schon bei `private`- oder `static`-Methoden **früh gebunden** werden.

11.2.3.5 Zusammenfassung der Bindungsarten

Methoden, die `private`, `static` oder `final` sind, können vom Compiler **statisch** oder **früh gebunden** werden. Alle anderen Methoden werden **dynamisch** oder **spät gebunden**.

11.3 Besonderheiten bei der Vererbung

In diesem Kapitel werden wichtige Besonderheiten bei der Vererbung vorgestellt. Für den problemlosen Einsatz der Vererbung in der Praxis muss man Kenntnisse über die Typkonvertierung von Referenzen besitzen (siehe Kapitel 11.3.1), die Initialisierung mittels Konstruktoren in einer Vererbungshierarchie beherrschen (siehe Kapitel 11.3.2), wissen, dass Datenfelder beim Ableiten verdeckt werden können (siehe Kapitel 11.3.3) und dass man das Ableiten von Klassen und das Überschreiben von Methoden gezielt verhindern kann (siehe Kapitel 11.3.4). Abschließend werden in Kapitel 11.3.5 abstrakte Basisklassen vorgestellt.

11.3.1 Typkonvertierung von Referenzen

In Java ist es nicht unbedingt erforderlich, dass der Typ einer Referenzvariablen identisch mit dem Typ des Objektes ist, auf das die Referenzvariable zeigt. Genauso wenig muss bei einer Zuweisung der Typ der an der Zuweisung beteiligten Referenzen identisch sein. Dies gilt auch bei der Zuweisung von aktuellen Parametern an formale Parameter bei einem Methodenaufruf sowie bei Rückgabewerten.

Man braucht in diesen Fällen keine Identität der Typen, aber **Zuweisungskompatibilität**! So muss beispielsweise der Typ eines aktuellen Parameters nicht identisch mit dem Typ des formalen Parameters sein, er muss nur zuweisungskompatibel sein.

Es gibt wie bei der Typkonvertierung von einfachen Datentypen auch bei Referenztypen eine **implizite** (**automatische**) **Typkonvertierung** und eine **explizite Typkonvertierung** mit Hilfe des **cast-Operators** (siehe Kapitel 7.7.1).

11.3.1.1 Implizite Typkonvertierung von Referenzen

Eine implizite Typkonvertierung von Referenzen findet immer dann statt, wenn eine Referenzvariable vom Typ einer abgeleiteten Klasse einer Referenzvariablen vom Typ einer Basisklasse zugewiesen wird.

Als Beispiel für eine Basisklasse und eine abgeleitete Klasse sollen die Klassen `Vater` und `Sohn` dienen. Das folgende Bild zeigt die Vererbungshierarchie aus diesen beiden Klassen:

Bild 11-10 Vater-Sohn-Vererbungshierarchie

Ein Objekt einer abgeleiteten Klasse ist sowohl vom Typ der eigenen Klasse als auch vom Typ jeder zugehörigen Basisklasse.

Die implizite Typkonvertierung soll anhand des folgenden Codes diskutiert werden:

```
Sohn refSohn = new Sohn();
Vater refVater = refSohn;
```

Die Referenz `refVater` ist vom Referenztyp `Vater` und zeigt auf ein Objekt der Klasse `Sohn`. Dies ist deshalb zulässig und möglich, weil ein Sohnobjekt durch den Vererbungsmechanismus auf jeden Fall alle Eigenschaften besitzt, die auch ein Vaterobjekt besitzt. Ein Objekt der Klasse `Sohn` ist stets vom Typ `Sohn` und vom Typ `Vater`. Allerdings hat die Referenz `refVater` nur Zugriff auf die Vateranteile des Sohnobjektes – die Sohnanteile sind für die Referenz `refVater` unsichtbar und damit auch nicht zugreifbar.

Bei der Zuweisung `refVater = refSohn` wird auf die Referenz `refSohn` implizit der cast-Operator `(Vater)` angewandt, wenn der Typ der Referenz `refVater` ein Supertyp der Referenz `refSohn` ist, wobei die zusätzlichen Eigenschaften des Sohnes einfach wegprojiziert werden. Ist der Typ der Referenz `refVater` aber kein Supertyp der Referenz `refSohn`, so resultiert ein Kompilierfehler.

Da die cast-Operation für den Programmierer unsichtbar erfolgt, spricht man auch von **impliziter Typkonvertierung**. Der Rückgabewert dieser impliziten cast-Operation ist eine Referenz vom Typ `Vater` auf das gleiche Objekt, allerdings mit einer **eingeschränkten Sichtweise**. Nach der Zuweisung `refVater = refSohn` zeigen also zwei Referenzen auf dasselbe Objekt vom Typ `Sohn`.

Die **Referenz vom Typ** `Sohn` sieht das **gesamte Objekt** und die **Referenz vom Typ** `Vater` sieht nur die **Vateranteile des Sohnobjektes**.

Das folgende Bild zeigt diesen Zusammenhang:

Bild 11-11 Vom Typ der Referenz abhängige Sicht auf ein Objekt

Die für die Referenz `refVater` unsichtbaren Teile des Sohnobjektes sind in Bild 11-11 durch einen schwarzen Balken[159] symbolisiert.

Die implizite Typkonvertierung in eine Superklasse bezeichnet man auch als **Up-Cast**.

Für einen gültigen Up-Cast gilt die folgende Regel:

Zeigen eine Referenz vom Typ `Vater` und eine Referenz vom Typ `Sohn` auf ein Objekt vom Typ `Sohn` in einer Vater-Sohn-Hierarchie, so ist nach der Zuweisung der Referenz vom Typ `Sohn` an die Referenz vom Typ `Vater` **für die Referenz vom Typ `Vater` nur der Vater-Anteil des Objekts vom Typ `Sohn` erreichbar**.

11.3.1.2 Explizite Typkonvertierung von Referenzen

Eine explizite Typkonvertierung von Referenzen mit Hilfe des cast-Operators muss immer dann erfolgen, wenn bei einer Zuweisung eine Referenzvariable vom Typ `Vater`, die auf ein Objekt der Klasse `Sohn` zeigt, einer Referenzvariablen vom Typ `Sohn` zugewiesen wird. Eine solche explizite Typkonvertierung mit Hilfe des cast-Operators bezeichnet man auch als **Down-Cast**.

[159] Diese Darstellung gilt für den Fall, dass die Sohnklasse die Vaterklasse erweitert. Der Fall des Überschreibens wird erst später behandelt.

Es wird nun von der gezeigten Situation ausgegangen:

Bild 11-12 Verdeckte Eigenschaften durch Casten der Referenz wieder sichtbar machen

Im folgenden Codestück ist eine explizite cast-Operation bei der Zuweisung an die Referenzvariable vom Typ `Sohn` zu sehen:

```
Sohn refSohn = new Sohn();        // refSohn zeigt auf ein
                                  // Sohn-Objekt
Vater refVater = refSohn;         // die Referenz refVater zeigt
                                  // auch auf dieses Sohn-Objekt
Sohn refSohn2 = (Sohn) refVater;  // Die refSohn2 ist vom Typ
                                  // Sohn. Sie sieht das ganze
                                  // Sohn-Objekt der Referenz
                                  // refVater infolge des Casts
```

Die Referenz `refVater` zeigt auf ein Objekt der Klasse `Sohn`. Auf dieser Referenz wird nun eine explizite cast-Operation ausgeführt. Der Rückgabewert der cast-Operation ist eine Referenz vom Typ `Sohn`, die der Referenz `refSohn2` zugewiesen wird.

Durch diesen Cast wird wieder das ganze `Sohn`-Objekt sichtbar.

Für einen gültigen Down-Cast gilt die folgende Regel:

Bei einer Zuweisung `refSohn = (Sohn) refVater` ist die explizite cast-Operation nur dann zulässig, wenn die Referenz `refVater`

- auf ein Objekt vom Typ `Sohn` zeigt
- oder auf ein Objekt eines Subtyps der Klasse `Sohn` zeigt.

11.3.1.3 Gültige Up- und Down-Cast-Operationen

Ein impliziter **Up-Cast** bezeichnet einen Cast in einen Typ, der in der Vererbungshierarchie weiter oben liegt. Ein **Down-Cast** bezeichnet einen Cast in einen Typ, der in der Vererbungshierarchie weiter unten liegt.

Anhand der folgenden einfachen Vererbungshierarchie sollen nochmals die gültigen Up- und Down-Cast-Operationen erläutert werden:

Bild 11-13 Vererbungshierarchie zur Diskussion der zulässigen cast-Operationen

Ein **Down-Cast** erfordert immer die e**xplizite Angabe des cast-Operators**.

Durch den cast-Operator wird bei einem Down-Cast zum Ausdruck gebracht, in welchen Typ gewandelt werden soll. Die zu konvertierende Referenz kann dabei auf ein Objekt dieses Typs oder auf ein Objekt eines von diesem Typ abgeleiteten Typs zeigen.

Dagegen ist es bei einem gültigen **Up-Cast nie erforderlich**, **den cast-Operator anzugeben**, da der implizite Cast in die Superklasse eindeutig ist.

Ein Up-Cast erfolgt bei Bedarf automatisch durch den Compiler.

Bild 11-14 und Bild 11-15 zeigen beispielhaft, welche Referenz auf welches Objekt zeigen darf, und welche **impliziten Up-Cast-Operationen** bzw. welche **expliziten Down-Cast-Operationen** erlaubt sind.

Das folgende Bild zeigt **zulässige implizite Up-Cast-Operationen**:

Bild 11-14 Zulässige implizite Up-Cast-Operationen

Im Folgenden werden anhand eines Beispiels **zulässige explizite Down-Cast-Operationen** gezeigt:

Bild 11-15 Zulässige explizite Down-Cast-Operationen

Durch den impliziten Up-Cast-Mechanismus ist es auch möglich, dass einer Methode, die als formalen Parameter eine Referenz vom Typ einer Basisklasse hat, als aktueller Parameter eine Referenz vom Typ der Basisklasse oder eine Referenz vom Typ einer abgeleiteten Klasse übergeben werden kann. Ist der aktuelle Parameter vom Typ der abgeleiteten Klasse, so findet eine implizite Typumwandlung des aktuellen Parameters in den Typ der Basisklasse statt.

Ist ein formaler Parameter einer Methode eine Referenz vom Typ der Klasse `Object`, so kann jede beliebige Referenz an diese Methode übergeben werden, da bekanntlich jede Klasse von der Klasse `Object` abgeleitet ist.

Wird mit folgender Anweisung

```
B refB = new C();
```

eine Instanz der Klasse `C` angelegt, so ist das auf dem Heap erzeugte Objekt vom Typ `C`, die Referenz `refB` aber vom Typ `B`. Bei der Zuweisung findet implizit ein Cast auf die Klasse `B` statt.

Auf die Referenz `refB` kann nun – wie in Kapitel 11.3.1.2 gezeigt – der cast-Operator `(C)` wie in folgender Anweisung angewandt werden:

```
C refC = (C) refB;
```

Beim Down-Cast wird die Aufrufschnittstelle des Typs, auf den gecastet wird, sichtbar.

Mit dem cast-Operator kann natürlich auch eine Referenz einer abgeleiteten Klasse explizit auf die Basisklasse `A` und die Basisklasse `Object` gecastet werden. Allerdings ist dann natürlich nur die entsprechende Aufrufschnittstelle sichtbar, die zum Typ der Referenz gehört, in den gecastet wird.

Wird versucht, eine Referenz explizit auf einen Typ zu casten, der nicht zulässig ist, so wird eine Exception[160] vom Typ `ClassCastException` zur Laufzeit geworfen. Diese Exception wird nur bei expliziten cast-Operationen geworfen, da die Korrektheit einer impliziten cast-Operation immer schon zum Kompilierzeitpunkt überprüft werden kann.

Beispiel

Das nachfolgende Beispiel `Cast.java` zeigt gültige und ungültige cast-Operationen. Es wird die Vererbungshierarchie von Bild 11-16 zugrunde gelegt:

Bild 11-16 Klassenhierarchie

Man beachte, dass die Klasse `D` weder Subtyp noch Supertyp der Klassen `A`, `B` oder `C` ist. Cast-Operationen einer Referenz auf ein Objekt der Klasse `A`, `B` oder `C` in den Typ der Klasse `D` sind also unzulässig. Umgekehrt gilt natürlich das Gleiche: Eine Referenz auf ein Objekt der Klasse `D` kann nie in den Typ der Klasse `A`, `B` oder `C` gecastet werden.

[160] Exceptions werden in Kapitel 13 behandelt.

Nun das bereits erwähnte Beispiel:

```
// Datei: Cast.java
class A
{
   private int x = 1;
}

class B extends A
{
   private int y = 2;
}

class C extends B
{
   private int z = 3;
}

class D
{
   private float f = 2.0f;
}

public class Cast
{
   public static void main (String[] args)
   {
      B b = new C(); // Referenz vom Typ B auf ein neues Objekt der
                     // Klasse C. Impliziter Cast nach B.

      A a = b;       // Impliziter Cast zu einer Basisklasse.
      Object o = b;  // Impliziter Cast nach Object.

      C c = (C) b;   // Expliziter Cast zur ursprünglichen Klasse.
      C c2 = (C) o;  // Expliziter Cast von Object zur
                     // ursprünglichen Klasse.

      //D d = (D) b; // Fehler, da D nichts mit C zu tun hat.
   }
}
```

Casten auf Array-Typen

Da Array-Typen Subtypen der Klasse `Object` sind, kann auch auf Referenzen, die auf Arrays zeigen, der cast-Operator angewandt werden. Die folgenden Codeausschnitte zeigen das Casten auf Arrays:

```
String[] arr1 = {"Anna", "Katharina"};
Object ref1 = arr1;                // Impliziter Up-Cast
String[] arr2 = (String[]) ref1;   // Expliziter Down-Cast
```

Beachten Sie, dass wenn eine Referenz `ref`, auf der eine cast-Operation (`String[]`) erfolgt, nicht auf ein Array des Typs `String[]` zeigt, eine Exception[161] vom Typ `ClassCastException` zur Laufzeit geworfen wird.

[161] Exceptions werden in Kapitel 13 behandelt.

Der folgende Codeausschnitt verdeutlicht nochmals, dass Arrays von elementaren Datentypen genauso Objekte sind wie Arrays aus Referenztypen:

```
int[] arr3 = {1, 2};
Object ref2 = arr3;
int[] arr4 = (int[]) ref2;
```

11.3.2 Konstruktoren bei abgeleiteten Klassen

Die folgende Vererbungshierarchie dient als Hilfsmittel zur Diskussion:

Bild 11-17 Vererbungshierarchie zur Diskussion von Konstruktoraufrufen

Möchte man dem Benutzer einer Klasse die Möglichkeit bieten, zusammen mit der Objekterzeugung eine Initialisierung vorzunehmen, so stellt man ihm Konstruktoren zur Verfügung. Dieser Sachverhalt wurde bereits in Kapitel 10.4.4 vorgestellt. Um die ererbten Datenfelder einer Vaterklasse zu initialisieren, würde man sich wünschen, dass man einen Konstruktor einer Vaterklasse aus dem Konstruktor einer Sohnklasse aufrufen kann. Damit könnte man die Konstruktoren der Vaterklasse wiederverwenden. Ein Konstruktor der Vaterklasse könnte dann den von der Vaterklasse geerbten Anteil der Sohnklasse initialisieren, der Konstruktor der Sohnklasse müsste dann nur noch die neu hinzu gekommenen Datenfelder initialisieren.

Und so ist es auch:

Es ist möglich, innerhalb eines Konstruktors einer abgeleiteten Klasse einen Konstruktor der Vaterklasse aufzurufen. Dies wird durch den Aufruf von **`super()`** gemacht. Der Konstruktor der Vaterklasse initialisiert die von der Vaterklasse geerbten Anteile eines Objekts einer abgeleiteten Klasse.

Wie das Schlüsselwort `super` eingesetzt wird, kann man sich zum größten Teil vom Compiler abschauen. Der Compiler ergänzt nämlich den folgenden Programmcode, wie es in Fettschrift ausgedrückt ist:

```
public class Person
{
```

```
   private String name;
   private String vorname;
   public Person()
   {
      name = "Unbekannt";
      vorname = "Unbekannt";
   }
}

public class Student extends Person
{
   private int matrikelnummer
}
```

zu:

```
public class Person extends Object
{
   private String name;
   private String vorname;

   public Person()
   {
      super();
      name = "Unbekannt";
      vorname = "Unbekannt";
   }
}

public class Student extends Person
{
   private int matrikelnummer

   public Student()    // Default-Konstruktor wird
   {                   // vom Compiler eingefügt.
      super();
   }
}
```

Der Aufruf von `super()` muss immer in der ersten Zeile des Konstruktors stehen. Lässt man ihn weg, so fügt Java implizit einen Aufruf des parameterlosen Konstruktors der Superklasse ein.

Dass die **Anweisung `super()`** im selbst geschriebenen parameterlosen Konstruktor der Klasse `Person` in der ersten Codezeile steht, ist kein Zufall, es ist sogar ein Muss! Das bedeutet, dass der Konstruktor der obersten Klasse im Hierarchiebaum mit der Initialisierung beginnt und an letzter Stelle die Initialisierungsanweisungen des Konstruktors der Klasse, die dem `new`-Operator übergeben wurde, abgearbeitet werden. Man beachte, dass mit dem Aufruf `super()` im selbst geschriebenen para-

meterlosen Konstruktor der Klasse `Person` der Default-Konstruktor der Klasse `Object` aufgerufen wird.

11.3.2.1 Aufruf des parameterlosen Konstruktors

Der **Default-Konstruktor** der **Klasse** `Object` hat einen leeren Rumpf, d. h. er tut nichts. Ein **Default-Konstruktor** einer anderen **Klasse ruft automatisch den parameterlosen Konstruktor der Vaterklasse** auf.

Ein Konstruktor der Basisklasse wird immer vor den Initialisierungsanweisungen des Konstruktors der abgeleiteten Klasse ausgeführt.

Der Grund dafür ist, dass etwaige Initialisierungen des Konstruktors der Basisklasse übernommen werden können, aber auch bei Bedarf – sofern die Datenfelder der Basisklasse nicht `private` sind – im Konstruktor der abgeleiteten Klasse überschrieben werden können.

Falls kein selbst geschriebener Konstruktor in einer Klasse existiert, stellt der Compiler einen parameterlosen Konstruktor für diese Klasse zur Verfügung wie in der Klasse `Student` in Kapitel 11.3.2. Dies ist der sogenannte **Default-Konstruktor**. Um Initialisierungen der Instanzvariablen vorzunehmen, wird oft ein eigener Konstruktor geschrieben, wie in der Klasse `Person` in Kapitel 11.3.2 zu sehen ist.

Wird irgendein Konstruktor geschrieben, sei es mit oder ohne Parameter, steht der Default-Konstruktor des Compilers nicht zur Verfügung.

Ein **parameterloser Konstruktor** hat **keine Parameter**. Der Programmierer kann **selbst** einen **Konstruktor ohne Parameter schreiben**. Dann wird vom Compiler dieser selbstgeschriebene parameterlose Konstruktor und nicht der vom Compiler zur Verfügung gestellte **Default-Konstruktor** aufgerufen.

11.3.2.2 Aufruf eines Konstruktors mit Parametern

Die Probleme der Initialisierung der von der Vaterklasse geerbten Datenfelder sind durch den einfachen Aufruf von `super()` in der Regel noch nicht gelöst, da hiermit nur der parameterlose Konstruktor aufgerufen werden kann.[162]

Für den Fall, dass Parameter im Konstruktor der Vaterklasse benötigt werden, kann man diese Parameter mit Hilfe des Schlüsselwortes `super` an den Konstruktor der

[162] Es sei denn, im parameterlosen Konstruktor wird initialisiert z. B. durch eine interaktive Eingabe der Werte.

Vaterklasse weiterreichen. Bekanntlich besteht die Möglichkeit, für jede Klasse beliebig viele Konstruktoren zu schreiben. Die einzige Einschränkung hierfür ist: Die Parameterliste zweier Konstruktoren darf in Typ, Reihenfolge und Anzahl der Parameter nicht identisch sein. Wenn der Konstruktor der Basisklasse Parameter erwartet, so müssen diese vom Konstruktor der abgeleiteten Klasse bereitgestellt und mit Hilfe des Schlüsselwortes `super` an den entsprechenden Konstruktor der Basisklasse weitergereicht werden. Dabei gilt:

Ein formaler Parameter des Konstruktors der abgeleiteten Klasse kann als aktueller Parameter an den Konstruktor der Basisklasse übergeben werden.

Das folgende Beispiel zeigt die **Übergabe von Parametern an** den **Konstruktor** der **Vaterklasse**:

```
// Datei: Person4.java

public class Person4
{
   private String name;
   private String vorname;

   public Person4 (String name, String vorname)
   {
      System.out.println ("Konstruktoraufruf von Person4");
      this.name = name;
      this.vorname = vorname;
   }
}

// Datei: Student4.java

public class Student4 extends Person4
{
   private int matrikelnummer;

   public Student4 (String name, String vorname, int m)
   {
      super (name, vorname); // Aufruf des Konstruktors der
                             // Superklasse
      System.out.println ("Konstruktoraufruf von Student4");
      matrikelnummer = m;
   }
}

// Datei: Test4.java

public class Test4
{
   public static void main (String[] args)
   {
      Person4 p = new Person4 ("Müller", "Peter");
      Student4 s = new Student4 ("Brang", "Rainer", 666666);
   }
}
```

Die Ausgabe des Programms ist:

```
Konstruktoraufruf von Person4
Konstruktoraufruf von Person4
Konstruktoraufruf von Student4
```

Zum Schluss soll noch eine Kleinigkeit erwähnt werden, die des Öfteren zu unerwarteten Fehlern führt. Wird vom Programmierer ein Konstruktor mit Parametern geschrieben und kein zusätzlicher Konstruktor ohne Parameter, so steht für diese Klasse kein Konstruktor mit leerer Parameterliste mehr zur Verfügung. Wird diese Klasse dann als Vaterklasse für eine andere Klasse verwendet, so setzt der Compiler automatisch in jeden Konstruktor der Sohnklasse – in dem kein expliziter Aufruf von `super()` mit Parametern erfolgt – den Aufruf von `super()` ohne Parameter ein. Dieser Aufruf bezieht sich allerdings auf den Default-Konstruktor der Basisklasse, der ja gar nicht mehr existiert. Da die vom Compiler eingefügten Aufrufe von `super()` für den Programmierer nicht sichtbar sind, kann die Fehlersuche langwierig sein.

In den Konstruktor einer abgeleiteten Klasse wird automatisch vom Compiler der Aufruf des parameterlosen Konstruktors der Vaterklasse durch `super()` eingefügt, es sei denn, der Programmierer führt einen Aufruf von `super()` explizit selbst durch.

Dies hat die folgende Konsequenz:

Ein Programmierer sollte stets auch einen parameterlosen Konstruktor für eine Klasse schreiben, wenn er einen Konstruktor mit Parametern schreibt, weil durch das Schreiben des Konstruktors mit Parametern der Default-Konstruktor nicht mehr zur Verfügung steht.

11.3.2.3 Initialisierungsreihenfolge bei Aufruf von `super()`

Wie das folgende Beispiel zeigt, werden manuelle Initialisierungen und die Initialisierung durch einen nicht statischen Initialisierungsblock eines Objektes einer abgeleiteten Klasse direkt nach dem Aufruf von `super()` ausgeführt. Anschließend erfolgt die Abarbeitung der Anweisungen des Konstruktors der abgeleiteten Klasse.

Jetzt das bereits erwähnte Beispiel:

```
// Datei: Person5.java

public class Person5
{
   private String name;
   private String vorname;

   public Person5()
   {
```

```
      System.out.println ("Konstruktoraufruf von Person5");
      name = "Unbekannt";
      vorname = "Unbekannt";
   }
}

// Datei: Student5.java

public class Student5 extends Person5
{
   private int matrikelnummer = getMatrikelnummer();
   {
      System.out.println ("Nicht statischer Initialisierungsblock: "
                          + "matrNr = 2");
      matrikelnummer = 2;
   }

   public Student5()
   {
      super();     // expliziter Aufruf des Konstruktors von Person5
      System.out.println ("Konstruktoraufruf von Sohn5: matrNr = 3");
      matrikelnummer = 3;
   }

   private int getMatrikelnummer()
   {
      System.out.println ("Manuelle Initialisierung: matrNr = 1");
      return 1;
   }
}

// Datei: Test5.java

public class Test5
{
   public static void main (String[] args)
   {
      new Student5();
   }
}
```

Die Ausgabe des Programms ist:

```
Konstruktoraufruf von Person5
Manuelle Initialisierung: matrNr = 1
Nicht statischer Initialisierungsblock: matrNr = 2
Konstruktoraufruf von Sohn5: matrNr = 3
```

Im Folgenden wird der Ablauf beim Aufruf des Konstruktors einer abgeleiteten Klasse beschrieben. Dabei ist zu beachten, dass die Anweisungen des Konstruktorblocks der abgeleiteten Klasse erst am Schluss eine Reihenfolge von Konstruktoraufrufe steht.

Allgemein beschrieben sieht der Ablauf beim Aufruf eines Konstruktors einer abgeleiteten Klasse folgendermaßen aus:

1. Beginnt der Konstruktor der abgeleiteten Klasse in der ersten Zeile mit einem Aufruf eines anderen Konstruktors derselben Klasse (mittels `this()`), so wird der entsprechende Konstruktor aufgerufen.
2. Beginnt der Konstruktor der abgeleiteten Klasse in der ersten Zeile nicht mit einem Aufruf eines Konstruktors derselben Klasse (mittels `this()`), so folgt ein impliziter oder expliziter Aufruf eines Konstruktors der Vaterklasse mittels `super()`.
3. Nachdem der Konstruktor der Vaterklasse vollständig abgearbeitet wurde, werden die entsprechenden Instanzvariablen der abgeleiteten Klasse manuell und durch nicht statische Initialisierungsblöcke initialisiert. Die Reihenfolge entspricht dabei der Reihenfolge der Variablendefinition und der nicht statischen Initialisierungsblöcke innerhalb der Quelldatei.
4. Die eigentlichen Anweisungen des Konstruktorblocks der abgeleiteten Klasse werden ausgewertet.

Zu beachten ist, dass die Instanzvariablen durch die automatische Initialisierung bereits mit Default-Werten belegt sind, bevor der Block eines Konstruktors betreten wird.

11.3.2.4 Aufruf von `super()` und von `this()`

Bestimmt erinnern Sie sich an das Lernkästchen in Kapitel 10.4.5, in dem stand, dass mit Hilfe von `this (. . . . .)` aus einem Konstruktor ein anderer Konstruktor derselben Klasse aufgerufen werden kann, und dass dieser Aufruf in der ersten Codezeile des Konstruktors stehen muss. Das Gleiche gilt allerdings auch für den `super()`-Aufruf. Deshalb gilt:

In einem Konstruktor kann entweder ein anderer überladener Konstruktor derselben Klasse aufgerufen werden oder ein Konstruktor der Vaterklasse.

11.3.3 Verdecken

Es ist möglich, dass eine abgeleitete Klasse ein Datenfeld mit dem gleichen Namen einführt. Das entsprechende **Datenfeld** einer Vaterklasse wird zwar beim Sohn auch angelegt, ist aber infolge der Namensgleichheit **verdeckt**, d. h. unter dem Namen wird immer das entsprechende Datenfeld, das im Sohn neu eingeführt wurde, angesprochen.

11.3.3.1 Verdecken von Datenfeldern

Vom Verdecken eines Datenfeldes spricht man, wenn in der Sohnklasse ein Datenfeld angelegt wird, das den gleichen Namen trägt wie ein von der Vaterklasse geerbtes Datenfeld.

Das folgende Bild zeigt eine solche Vererbungshierarchie:

Bild 11-18 Sohnklasse definiert das gleiche Datenfeld wie die Vaterklasse

Da eine Instanz der Klasse `Sohn` einmal die eigenen Datenfelder besitzt und zum anderen die Datenfelder der Vaterklasse erbt, kann es also vorkommen, dass das Sohnobjekt zwei Datenfelder mit dem gleichen Namen besitzt.

Wird von der Klasse `Sohn` mit

```
Sohn s = new Sohn();
```

eine Instanz gebildet, so enthält die Instanz der Sohnklasse sowohl die Datenfelder der Vaterklasse, als auch die Datenfelder der Sohnklasse. Ein Objekt der Sohnklasse ist in Bild 11-19 zu sehen:

Bild 11-19 Doppeltes Datenfeld `x` in der Instanz der Sohnklasse

Dabei verdeckt aber das gleichnamige Datenfeld der Sohnklasse das von der Vaterklasse geerbte Datenfeld. Das bedeutet, dass man im Sohn über den Namen `x` stets auf das Datenfeld der Sohnklasse zugreift.

Das Verdecken von Datenfeldern erfolgt grundsätzlich bei Namensgleichheit. Deshalb wird ein `float`-Datenfeld `x` des Sohnes, das von der Vaterklasse geerbt wurde, auch durch ein `int`-Datenfeld `x` der Sohnklasse verdeckt.

Dies zeigt das folgende Bild:

Bild 11-20 Verdecken eines geerbten gleichnamigen Datenfeldes eines anderen Typs

11.3.3.2 Verwendung verdeckter Datenfelder

Zugriff auf verdeckte Instanzvariablen der Vaterklasse

Es soll nun folgendes Beispiel betrachtet werden: Die Klasse `Vater` hat ein Datenfeld `int x`. Die Klasse `Sohn`, die von der Klasse `Vater` abgeleitet ist, definiert ebenfalls ein Datenfeld `int x`. Beide Datenfelder können unabhängig voneinander in der Sohnklasse existieren, und es kann sogar auf beide Datenfelder von der Sohnklasse aus zugegriffen werden, sofern das Datenfeld des Vaters nicht den Zugriffsmodifikator `private` hat. Dies demonstriert das folgende Programm:

```
// Datei: VaterSohnTest.java

class Vater
{
   int x = 2;
}

class Sohn extends Vater
{
   int x = 1;
   public Sohn()
   {
      System.out.println ("x des Sohnes: " + x);
      System.out.println ("x des Sohnes: " + this.x);
      System.out.println ("vom Vater geerbtes x: " + super.x);
      System.out.println ("vom Vater geerbtes x: "+((Vater)this).x);
   }
}

public class VaterSohnTest
{
   public static void main (String[] args)
   {
      Sohn s = new Sohn();
   }
}
```

Die Ausgabe des Programms ist:

```
x des Sohnes: 1
x des Sohnes: 1
vom Vater geerbtes x: 2
vom Vater geerbtes x: 2
```

Für den Zugriff auf das eigene Datenfeld der Sohnklasse hat das Sohnobjekt zwei Möglichkeiten, die schon bekannt sind:

- einfach über den Namen der Variablen: `x`
- mit Hilfe der `this`-Referenz: `this.x`

Dabei wird die erste Variante vom Compiler automatisch in die zweite Variante umgesetzt.

Auf das von der Vaterklasse geerbte Element, das von der Sohnklasse durch ein gleichnamiges Element **verdeckt** wird, kann nicht mehr einfach über den Namen der Variablen zugegriffen werden.

Für den Zugriff auf ein von der Vaterklasse ererbtes und verdecktes Datenfeld bestehen die beiden folgenden Möglichkeiten:

- über das Schlüsselwort **`super`** mit `super.x`.
- über einen Cast der `this`-Referenz in das Vaterobjekt: `((Vater) this).x`.

Dabei wird die erste Möglichkeit vom Compiler automatisch in die zweite Möglichkeit umgesetzt. Das Schlüsselwort `super` wurde schon in Kapitel 11.3.2 vorgestellt. Dort wurde es in der Notation `super()` verwendet, um den Konstruktor der Basisklasse ohne Parameter aufzurufen. An dieser Stelle wird es dazu verwendet, um ein verdecktes Datenfeld der Vaterklasse aufzurufen. In der zweiten Möglichkeit wird von der `this`-Referenz Gebrauch gemacht.

Wie bekannt, ist die `this`-Referenz eine Referenz auf das eigene Objekt und kann genauso wie andere Referenzen auf den Typ der Vaterklasse gecastet werden. Wird der Typ einer Referenz auf ein Objekt auf die Klasse des Vaters gecastet, sind nur noch die Datenfelder des Vaters sichtbar und nicht mehr die des Sohnes.

Zugriff auf verdeckte Instanzvariablen einer "Großvaterklasse"

Hierzu soll folgendes Beispiel betrachtet werden: Die Klasse `Grossvater` hat ein Datenfeld `float x`. Die Klasse `Vater`, die von der Klasse `Grossvater` abgeleitet ist, hat ein Datenfeld `int x` und die Klasse `Sohn`, die von der Klasse `Vater` abgeleitet ist, ein Datenfeld `String x`. Es soll hier der Fall betrachtet werden, wie auf verdeckte Datenfelder zugegriffen werden kann, die im Klassenbaum weiter oben liegen und somit nicht von der Vaterklasse stammen. Weiterhin wird demonstriert, dass zum Verdecken von Datenfeldern nur Namensgleichheit, jedoch nicht die Typgleichheit erforderlich ist. Hier das Programm:

```
// Datei: Sohn2.java

class Grossvater2
{
   float x = 2.2F;
}

class Vater2 extends Grossvater2
{
   int x = 2;
}

public class Sohn2 extends Vater2
{
   String x = "Ich bin der Sohn";

   public Sohn2()
   {
      System.out.println ("x des Sohnes: " + x);
      System.out.println ("x des Sohnes: " + this.x);
      System.out.println ("ererbtes x vom Vater: " + super.x);
      System.out.println ("ererbtes x vom Vater: "
                          + ((Vater2) this).x);
      System.out.println ("ererbtes x vom Grossvater: "
                          + ((Grossvater2) this).x);
      // Das Folgende funktioniert nicht!
      // System.out.println ("ererbtes x vom Grossvater: "
      //                     + super.super.x);
   }

   public static void main (String[] args)
   {
      Sohn2 s = new Sohn2();
   }
}
```

Die Ausgabe des Programms ist:

```
x des Sohnes: Ich bin der Sohn
x des Sohnes: Ich bin der Sohn
ererbtes x vom Vater: 2
ererbtes x vom Vater: 2
ererbtes x vom Grossvater: 2.2
```

Nach wie vor gibt es jeweils 2 Möglichkeiten, um auf die Datenfelder der eigenen Klasse und die ererbten Datenfelder der Vaterklasse zuzugreifen. Es existiert aber in diesem Fall nur eine einzige Möglichkeit, um auf die ererbten Datenfelder der Klasse des Großvaters zuzugreifen. Nur der Cast der `this`-Referenz in die `Grossvater`-Klasse ermöglicht den Zugriff auf das verdeckte Datenfeld.

Eine Aneinanderreihung von `super.super` gestattet der Compiler nicht!

Wird in obigem Programm nur die folgende minimale Abänderung gemacht:

```
class Vater2 extends Grossvater2
{
  int y = 2;
}
```

dann erhält man folgende Ausgabe des Programms:

```
x des Sohnes: Ich bin der Sohn
x des Sohnes: Ich bin der Sohn
ererbtes x vom Vater: 2.2
ererbtes x vom Vater: 2.2
ererbtes x vom Grossvater: 2.2
```

Es gibt jetzt 3 Möglichkeiten, auf das ererbte `x` der Klasse `Grossvater` zuzugreifen:

- über das Schlüsselwort `super` durch `super.x`
- über den Cast der `this`-Referenz in den Vater: `((Vater2)this).x`
- über den Cast der `this`-Referenz in den Grossvater: `((Grossvater2)this).x`

Der Zugriff mit `super.x` auf das Datenfeld `x` des Großvaters ist deshalb möglich, weil in der Klasse `Vater2` kein Datenfeld vorhanden ist, welches das geerbte Datenfeld `x` der Klasse `Grossvater2` verdeckt.

Wird ein Datenfeld `x` mit Hilfe von `super` gesucht, so wird ausgehend von der aktuellen Klasse die gesamte Klassenhierarchie aufwärts der Reihe nach solange durchsucht, bis zum ersten Mal ein Datenfeld `x` gefunden wird. An dieser Stelle wird die Suche abgebrochen.

Zugriff auf verdeckte Klassenvariablen

Der Zugriff auf verdeckte Klassenvariablen kann über den Klassennamen erfolgen. Da ein Objekt seine Klasse kennt, kann auf eine verdeckte Klassenvariable auch über eine Referenz auf ein Objekt dieser Klasse zugegriffen werden. Das folgende Beispiel ist identisch mit dem vorhergehenden, mit dem einen Unterschied, dass alle Datenfelder `static` sind:

```
// Datei: Sohn3.java
class Grossvater3
{
   static float x = 2.2F;
}

class Vater3 extends Grossvater3
{
   static int x = 2;
}

public class Sohn3 extends Vater3
{
   static String x = "Ich bin der Sohn";
```

```
   Sohn3()
   {
      System.out.println ("x des Sohnes: " + x);
      System.out.println ("x des Sohnes: " + this.x);
      System.out.println ("x des Sohnes: " + Sohn3.x);

      System.out.println ("geerbtes x vom Vater: " + super.x);
      System.out.println ("geerbtes x vom Vater: "
                          + ((Vater3) this).x);
      System.out.println ("geerbtes x vom Vater: " + Vater3.x);

      System.out.println ("geerbtes x vom Grossvater: "
                          + ((Grossvater3) this).x);
      System.out.println ("geerbtes x vom Grossvater: "
                          + Grossvater3.x);

      // Das Folgende funktioniert nicht!
      // System.out.println ("geerbtes x vom Grossvater: "
      //                     + super.super.x);
   }

   public static void main (String[] args)
   {
       Sohn3 s = new Sohn3();
   }
}
```

Aus dem Programm ist ersichtlich, dass für Klassenvariablen immer die Möglichkeit besteht, dass auf sie über den Klassennamen zugegriffen wird. Somit kann auf das `x` aus der Klasse `Grossvater3` mit `Grossvater3.x` und auf das `x` aus der Klasse `Vater3` mit `Vater3.x` und natürlich auf das `x` aus der Klasse `Sohn3` mit `Sohn3.x` zugegriffen werden.

Die Ausgabe des Programms ist:

```
x des Sohnes: Ich bin der Sohn
x des Sohnes: Ich bin der Sohn
x des Sohnes: Ich bin der Sohn
geerbtes x vom Vater: 2
geerbtes x vom Vater: 2
geerbtes x vom Vater: 2
geerbtes x vom Grossvater: 2.2
geerbtes x vom Grossvater: 2.2
```

11.3.4 Finale Methoden und finale Klassen

Mit dem Schlüsselwort `final` gekennzeichnete Methoden lassen sich nicht überschreiben. Wird das aus Kapitel 11.2.2 bekannte Beispiel so modifiziert, dass man die Methode `print()` der Klasse `Person2` als `final` deklariert, so lässt sich die Klasse `Student2` nicht mehr kompilieren. Hier die finale Methode `print()`:

```
public final void print()
{
```

```
   System.out.println ("Nachname :" + nachname);
   System.out.println ("Vorname  :" + vorname);
}
```

Die Ausgabe des Compilers ist:

```
Student2.java:12: print() in Student2 cannot override
print() in Person2; overridden method is final
   public void print()
               ^
1 error
```

Finale Methoden können in einer Subklasse nicht überschrieben werden.

Finale Klassen sind Klassen, von denen man keine weiteren Klassen ableiten kann. Damit kann man nur die Benutzung von Klassen, aber nicht die Ableitung erlauben.

Hierfür sind konzeptionelle Gründe des Designs denkbar – das Problem ist gelöst –, aber auch Sicherheitsgründe. Da ein abgeleitetes Objekt, wenn die Verträge der Basisklasse eingehalten werden, überall dort stehen kann, wo ein Objekt seiner Basisklasse steht, kann auf diese Weise beispielsweise kein Trojanisches Pferd von einem Hacker eingeschleust werden. Finale Klassen werden mit dem Schlüsselwort `final` gekennzeichnet.

Datenfelder, die mit dem Schlüsselwort `final` gekennzeichnet werden, tragen einen konstanten Wert. Mit einer finalen Klasse lässt sich auch eine sogenannte Konstantenklasse aufbauen.

Da **Konstantenklassen nur Konstanten enthalten**, macht es keinen Sinn, solche Klassen zu instanziieren. Deshalb wird gerne der Konstruktor auf `private` gesetzt. Damit kann dann eine Instanziierung gezielt verhindert werden. Allerdings müssen dann auch alle Konstanten `static` sein, damit sie von anderen Klassen aus als Klassenvariablen angesprochen werden können.

Eine Konstantenklasse könnte zum Beispiel so aussehen:

```
// Datei: Konstanten.java
public final class Konstanten
{
   private Konstanten()        // Von der Klasse können keine
   {                           // Objekte erzeugt werden.
   }

   public static final float PI = 3.141f;
   public static final int MAX = 255;
}
```

Der Zugriff auf die Datenfelder erfolgt nun z. B. über

```
System.out.println (Konstanten.PI);
```

11.3.5 Abstrakte Basisklassen

Eine abstrakte Basisklasse kann nicht instanziiert werden.

So kann die Abstraktion "Getränk" nicht instanziiert werden, wohl aber die abgeleitete Klasse "Sprudel" oder "Cola".

Eine Klasse, in der **alle Methoden implementiert** sind, kann mit dem Schlüsselwort `abstract` zur einer **abstrakten Klasse** gemacht werden. Damit kann die Instanziierung einer solchen Klasse verhindert werden. Eine solche Klasse ist zwar vollständig, sie stellt aber eine Abstraktion dar, von der es in der Realität keine Objekte gibt. Eine solche abstrakte Basisklasse dient allein dem Ziel der **Generalisierung in der Klassenhierarchie**.

Ein anderer Grund, warum eine abstrakte Klasse nicht instanziierbar ist, ist, dass sie abstrakte Methoden enthält.

Wird in einer Basisklasse nur die Schnittstelle von Methoden festgelegt und die eigentliche Implementierung einer, mehrerer oder aller Methoden erst in den abgeleiteten Klassen vorgenommen, so liegt eine **abstrakte Basisklasse** vor.

Methoden, für die nur die Schnittstelle festgelegt werden soll, die in einer Klasse also keinen Methodenrumpf besitzen, müssen mit dem Schlüsselwort **`abstract`** deklariert werden.

Ist auch nur eine einzige abstrakte Methode in einer Klasse enthalten, so ist diese Klasse zwangsläufig abstrakt und ist damit mit dem Schlüsselwort `abstract` zu deklarieren. **Abstrakte Klassen** können **nicht instanziiert** werden.

Abstrakte Basisklassen können zwar nicht instanziiert werden, jedoch kann mit Referenzen, die vom Typ einer abstrakten Basisklasse sind, gearbeitet werden. Solche Referenzen können dann auf Objekte zeigen, deren Klassen von der abstrakten Basisklasse abgeleitet sind und alle abstrakten Methoden implementieren.

Klassen, die von einer abstrakten Klasse ableiten, müssen nicht unbedingt alle abstrakten Methoden implementieren. Implementiert eine abgeleitete Klasse nicht alle abstrakten Methoden, ist sie wiederum wie die Basisklasse abstrakt und muss deshalb mit dem Schlüsselwort `abstract` deklariert werden.

Beispiel:

Ein kleines Programm soll die abstrakten Klassen verdeutlichen:

```
// Datei: AbstractTest.java

abstract class X
{
   public X()
   {
      System.out.println ("Konstruktor X");
   }

   // ist in abgeleiteter Klasse zu implementieren
   public abstract void testPrint (int x);
}

class Y extends X
{
   public Y()
   {
      System.out.println ("Konstruktor Y");
   }


   // hier wird die Methode testprint () implementiert
   public void testPrint (int x)
   {
      System.out.println ("Übergabeparameter: " + x);
   }
}

public class AbstractTest
{
   public static void main (String[] args)
   {
      // X x = new X();       Fehler!!
      Y y = new Y();          // OK!
      X z = new Y();          // auch OK!
   }
}
```

Die Ausgabe des Programms ist:

```
Konstruktor X
Konstruktor Y
Konstruktor X
Konstruktor Y
```

11.4 Polymorphie und das liskovsche Substitutionsprinzip

Polymorphie ist ein zentrales Konzept der objektorientierten Programmierung und betrifft sowohl Methoden als auch Objekte. Nach der Erläuterung des Begriffs Polymorphie in Kapitel 11.4.1 wird in 11.4.2 die **Polymorphie von Objekten** und das **liskovsche Substitutionsprinzip** besprochen.

11.4.1 Der Begriff der Polymorphie

Polymorphie ist neben Identität und Vererbung ein weiterer wichtiger Aspekt des objektorientierten Ansatzes. Polymorphie bedeutet Vielgestaltigkeit. Das Wort Polymorphie gibt es bei Operationen[163] und Objekten.

Polymorphie von Operationen bedeutet, dass eine Operation mit einer bestimmten Signatur in nicht verwandten Klassen durch einen jeweils eigenen Methodenrumpf implementiert wird.

Gleiche Methodenköpfe in verschiedenen nicht verwandten Klassen stellen kein Problem dar, da jede Klasse einen eigenen Namensraum bildet. Polymorphie von Operationen wurde bereits in Kapitel 9.3 behandelt.

Eine **Polymorphie von Objekten** gibt es nur bei **Vererbungshierarchien**.

11.4.2 Polymorphie von Objekten

Polymorphie (**Vielgestaltigkeit**) **von Objekten** bedeutet, dass ein Objekt in einer Vererbungshierarchie von verschiedenem Typ sein kann. Es kann vom Typ einer Basisklasse sein, aber auch vom Typ einer Unterklasse.

Das ist genauso zu sehen wie im Alltag, wo beispielsweise ein Student einerseits als eine normale Person, andererseits aber auch als Student auftreten kann. Übertragen auf die Modellierung bedeutet dies, dass ein Student als Objekt der abgeleiteten Klasse `Student` und als Objekt der Basisklasse `Person` auftreten kann.

Die Kundenklassen[164] eines **polymorphen Programms** dürfen es hierbei gar nicht merken, dass plötzlich Objekte einer abgeleiteten Klasse an die Stelle von Objekten der Basisklasse treten.

---

[163] Den Begriff polymorphe Methoden gibt es auch bei polymorphen Objekten.

[164] Mit Kundenklassen sind hier Klassen gemeint, die außerhalb der Klassen eines Programmsystems stehen und Klassen des Programmsystems als Clients nutzen

Barbara Liskov hat in [Lis88] Bedingungen formuliert, unter denen ein Programm die Verwendung polymorpher Objekte gefahrlos unterstützt. Die Bedingungen sind seitdem als **liskovsches Substitutionsprinzip** (**LSP**) bekannt. Das liskovsche Substitutionsprinzip fordert:

Will man Polymorphie von Objekten für ein Programm haben, so muss im Programm eine Referenz auf eine Instanz einer abgeleiteten Klasse an die Stelle einer Referenz auf eine Instanz einer Basisklasse treten können.

Überschreibende Methoden in der abgeleiteten Klasse dürfen hierbei die Verträge einer Basisklasse mit Kundenklassen, d. h. die Vor- und Nachbedingungen der Methoden der Basisklasse und ihre Klasseninvarianten, nicht brechen.

Solange bei der Ableitung von einer Basisklasse der Vertrag der entsprechenden Basisklasse in der abgeleiteten Unterklasse nicht gebrochen wird, ist es also möglich, das für Basisklassen geschriebene Programm auch für Unterklassen zu verwenden, die eventuell erst später erfunden werden.

Wenn das liskovsche Substitutionsprinzip nicht gelten würde, müsste stets geprüft werden, von welchem Typ das aktuelle Objekt ist, auf welches die Referenz gerade verweist.

Der **Vertrag** einer Klasse umfasst die Vor- und Nachbedingungen der Methoden und die Invarianten einer Klasse. Dabei kann eine Klasse mit verschiedenen Kundenklassen verschiedene Verträge haben. Was es bedeutet, dass ein Vertrag durch eine abgeleitete Klasse nicht gebrochen werden darf, wird in Kapitel 11.5.2.2 erläutert.

Man muss das liskovsche Substitutionsprinzip also unter zwei Gesichtspunkten betrachten:

1. **Nur Erweitern**

   Erweitert eine Klasse eine andere Klasse nur und überschreibt nichts, sondern fügt nur Attribute und Methoden hinzu, dann ist ganz klar, dass eine Instanz einer abgeleiteten Klasse an die Stelle einer Instanz ihrer Basisklasse treten kann. Dabei wird nämlich einfach durch einen Cast der erweiternde Anteil ausgeblendet. Zusätzliche Methoden der abgeleiteten Klasse werden dabei nicht angesprochen. Bei einer reinen Erweiterung wird bei der Ableitung keine Methode überschrieben.

2. **Überschreiben**

   Dieses Casten erfolgt natürlich auch, wenn die abgeleitete Klasse Methoden der Basisklasse überschreibt. Im Fall einer späten Bindung von Instanzmethoden wie in Java wird dann die überschreibende Methode an Stelle der überschriebenen Methode aufgerufen. Das darf aber keinen schädlichen Einfluss auf den Ablauf des Programms haben! Dies erzwingt, dass der Programmierer beim Überschreiben dafür sorgen muss, dass die überschreibende Instanzmethode den Vertrag der entsprechenden überschriebenen Instanzmethode einhält.

Werden Instanzmethoden in einer abgeleiteten Klasse überschrieben, so tritt in der abgeleiteten Klasse die überschreibende Instanzmethode an die Stelle der ursprünglichen, überschriebenen Methode. Wird ein Objekt der abgeleiteten Klasse als Objekt der Basisklasse betrachtet, wird trotzdem die überschriebene Methode aufgerufen, also für jedes Objekt die Methode seines Typs.

11.4.2.1 Polymorphes Verhalten von Objekten bei der Erweiterung

Bei einer reinen Erweiterung einer Basisklasse zu einer abgeleiteten Klasse ist ein polymorphes Verhalten von Objekten der abgeleiteten Klasse problemlos möglich, da ein Objekt einer abgeleiteten Klasse alle Methoden und Datenfelder einer Basisklasse erbt, wie in Bild 11-21 zu sehen ist, und sie im Falle der Erweiterung nur um zusätzliche Datenfelder und Methoden ergänzt.

Ein Objekt einer Unterklasse kann auch Methodenaufrufe beantworten, die in der Basisklasse implementiert sind. Natürlich kann ein Objekt einer abgeleiteten Klasse zusätzliche Datenfelder oder Methoden besitzen. Aber an der Stelle im Programmcode, an der ein Objekt einer Basisklasse stehen kann, kann auch ein Objekt einer abgeleiteten Klasse die Aufgaben erfüllen. Es verhält sich an dieser Stelle als Objekt der Basisklasse. Die weiteren Eigenschaften des Objektes der abgeleiteten Klasse wie zusätzliche Datenfelder oder Methoden werden einfach ausgeblendet bzw. sind über die Referenzvariable vom Typ einer Basisklasse nicht sichtbar. Dies bedeutet, dass ein Objekt einer abgeleiteten Klasse in **verschiedenen Gestalten** auftreten kann. Bild 11-21 zeigt, dass sich ein `Sohn`-Objekt auch als `Vater`- bzw. als `Großvater`-Objekt und dass sich ein `Vater`-Objekt auch als `Großvater`-Objekt verhalten kann:

Bild 11-21 Polymorphes Verhalten von Objekten abgeleiteter Klassen

Polymorphie ist der Grund für die wunderbare Möglichkeit, dass sich ganze Klassenbibliotheken problemlos wiederverwenden lassen. **Polymorphie** von Objekten erlaubt es, einen wiederverwendbaren **Code** zu schreiben, **der** nur **Referenzen auf Objekte einer Basisklasse enthält**. Da ein Objekt einer abgeleiteten Klasse, welche eine Basisklasse nur erweitert, die Aufrufschnittstelle der Basisklasse und dieselben Datenfelder wie die Basisklasse hat, kann ein **Objekt einer abgeleiteten Klasse** sich ohne jegliches Problem **wie** ein **Objekt einer Basisklasse verhalten**.

Eine Zusatzaufrufschnittstelle aufgrund weiterer Methoden der abgeleiteten Klasse kommt nicht zum Tragen, da diese Methoden im Quellprogramm, das nur die Basisklasse kennt, überhaupt nicht angesprochen werden. Die Zusatzaufrufschnittstelle und die zusätzlichen Datenfelder einer abgeleiteten Klasse bleiben also unsichtbar.

Mit dem Ziel der Wiederverwendung eines Programmcodes, der für eine Basisklasse geschrieben wurde, lässt sich das liskovsche Substitutionsprinzip also formulieren zu:

Liskovsches Substitutionsprinzip **im Falle der Erweiterung**:

Im Falle einer reinen Erweiterung kann ein Objekt einer abgeleiteten Klasse problemlos an die Stelle eines Objektes einer Basisklasse treten.

11.4.2.2 Polymorphes Verhalten beim Überschreiben

Etwas diffiziler wird es, wenn das Überschreiben von Methoden ins Spiel kommt. Hier ist das liskovsche Substitutionsprinzip nicht mehr selbstverständlich gegeben. Der Programmierer muss hierfür selbst etwas tun! Er muss dafür sorgen, dass die Client-Programme, welche mit Referenzen auf Objekte einer Basisklasse arbeiten, keine Schwierigkeiten bekommen, wenn an die Stelle eines Objektes einer Basisklasse plötzlich ein Objekt einer abgeleiteten Klasse tritt.

Das liskovsche Substitutionsprinzip **im Falle des Überschreibens**:

Im Falle des Überschreibens muss der Programmierer selbst dafür sorgen, dass ein Objekt einer abgeleiteten Klasse an die Stelle eines Objektes einer Basisklasse treten darf. Er muss hierfür beim Überschreiben die Einhaltung der Verträge der Basisklasse gewährleisten.

Das hat den Sinn, dass ein Kundenprogramm nicht merkt, dass ein Objekt einer abgeleiteten Klasse an die Stelle eines Objektes der Basisklasse getreten ist.

Auf Verträge wird in Kapitel 11.5 eingegangen.

11.4.3 Beispiele

11.4.3.1 Einfaches Programmbeispiel `Utility` für den Fall der Erweiterung

Das folgende Beispielprogramm demonstriert die Wiederverwendung von Quellcode durch den polymorphen Einsatz von Objekten. Das Programm besteht aus den Klassen `Person5`, `Student5`, `Utility` und `Test5`. Die Klasse `Student5` erweitert die Klasse `Person5`. In der Klasse `Utility` existieren die Klassenmethoden `sortByName()`, `swap()` und `print()`. Die Methode `sortByName()` kann ein Array von Personen nach dem Namen sortieren. Hierbei kommt der aus Kapitel 9.2.7 bekannte Bubblesort-Algorithmus zum Einsatz. Alle drei Methoden besitzen als formalen Parameter ein Array von Personen. Das Besondere ist nun, dass sämtliche Methoden der Klasse `Utility` nicht nur für ein Array von Personen verwendet werden können, sondern auch für ein Array von Studenten.

Hier das bereits angekündigte Programm:

```
// Datei: Person5.java

public class Person5
{
   private String nachname;
   private String vorname;

   public Person5 (String nachname, String vorname)
   {
      this.nachname = nachname;
      this.vorname = vorname;
   }

   public String getNachname()
   {
      return nachname;
   }

   public void print()
   {
      System.out.println (nachname + ", " + vorname);
   }
}

// Datei: Student5.java

public class Student5 extends Person5
{
   private int matrikelnummer;

   public Student5 (String nachname, String vorname,
                    int matrikelnummer)
   {
      super (nachname, vorname);
      this.matrikelnummer = matrikelnummer;
   }
   // weitere studentenspezifische Methoden
}
```

```
// Datei: Utility.java

public class Utility
{
   public static void sortByName (Person5[] ref)
   {
      int obergrenze = ref.length;

      while (obergrenze > 1)
      {
         for (int i = 1; i < obergrenze; i++)
         {
            String a = ref [i].getNachname();
            String b = ref [i - 1].getNachname();
            if(a.compareTo(b) < 0)
               swap (ref, i, i - 1);
         }
         obergrenze--;
      }
   }

   public static void swap (Person5[] ref, int index1, int index2)
   {
      Person5 tmp = ref [index1];
      ref [index1] = ref [index2];
      ref [index2] = tmp;
   }

   public static void print (Person5[] ref)
   {
      for (int i = 0; i < ref.length; i++)
      {
         ref [i].print();
      }
   }
}

// Datei: Test5.java

public class Test5
{
   public static void main (String[] args)
   {
      // Sortieren von Personen
      Person5[] refPersonen = new Person5 [3];
      refPersonen [0] = new Person5 ("Müller", "Max");
      refPersonen [1] = new Person5 ("Auer", "Ulrike");
      refPersonen [2] = new Person5 ("Zink", "Mareike");
      Utility.sortByName (refPersonen);
      System.out.println ("Sortiertes Array mit Personen:");
      Utility.print (refPersonen);

      // Sortieren von Studenten
      Student5[] refStudenten = new Student5[3];
      refStudenten [0] = new Student5 ("Wunder", "Emanuel", 14567);
      refStudenten [1] = new Student5 ("Maier", "Sabrina", 14568);
      refStudenten [2] = new Student5 ("Binder", "Katharina",14569);
```

```
        Utility.sortByName (refStudenten);
        System.out.println ("\nSortiertes Array mit Studenten:");
        Utility.print (refStudenten);
    }
}
```

Die Ausgabe des Programms ist:

```
Sortiertes Array mit Personen:
Auer, Ulrike
Müller, Max
Zink, Mareike

Sortiertes Array mit Studenten:
Binder, Katharina
Maier, Sabrina
Wunder, Emanuel
```

In der Klasse `Utility` wird nur die Basisklasse `Person5` verwendet. Trotzdem kann der komplette Programmcode der Klasse `Utility` auch für Objekte der Klasse `Student5` verwendet werden, da sich ein Student auch wie eine Person verhalten kann. Deshalb kann ein Objekt der Klasse `Student5` auch überall dort stehen, wo ein Objekt der Klasse `Person5` verlangt wird.

Umgekehrt ist nicht jede Person ein Student. Daher kann ein Objekt der Klasse `Person5` im Programm nicht überall dort stehen, wo ein Objekt der Klasse `Student5` steht. Quellcode, der für eine Basisklasse geschrieben wurde, kann im Falle der Erweiterung also von jeder beliebigen abgeleiteten Klasse benutzt werden. Der Erweiterungsteil der Objekte einer abgeleiteten Klasse wird einfach weggecastet.

11.4.3.2 Großes Programmbeispiel Warenlager mit Überschreiben

Im Folgenden wird ein etwas umfangreicheres Beispiel vorgestellt, in dem gezeigt wird, wie eine abgeleitete Klasse den Code, der für eine Basisklasse geschrieben wurde, benutzen kann. Es soll eine kleine Bibliothek erstellt werden, die Klassen für ein Waren-Management-System enthält. Je nachdem, welche Waren verwaltet werden müssen (Lebensmittel, Drogeriewaren, etc.) können spezialisierte Unterklassen der Bibliotheksklasse `Ware` gebildet werden.

Als erstes wird die Klasse `Ware` vorgestellt. Die Klasse `Ware` hat für ihre Instanzen, die verschiedenen Produkte, die Instanzvariablen:

- `nummer` (eindeutige Nummer für ein bestimmtes Produkt),
- `name` (Bezeichnung für ein Produkt),
- `preis`
- und `anzahl` (womit die zur Verfügung stehende Stückzahl oder Menge des jeweiligen Produkts gemeint ist).

Zusätzlich ist noch eine Klassenvariable `aktuelleNummer` vorhanden, die zum Erzeugen einer eindeutigen Nummer für jedes Produkt benutzt werden soll. Die

Methode `print()` ist fett hervorgehoben, da diese später in der Sohnklasse `Milch` überschrieben wird. Hier das bereits erwähnte etwas umfangreiche Beispiel:

```
// Datei: Ware.java

public class Ware
{
   protected int nummer;
   protected String name;
   protected float preis;
   protected int anzahl;
   protected static int aktuelleNummer = 0;

   public Ware (String name, float preis)
   {
      nummer = aktuelleNummer++;
      this.name = name;
      this.preis = preis;
      anzahl = 0;
   }

   public int getNummer()
   {
      return nummer;
   }

   public void stueckzahlErhoehen (int anzahl)
   {
      this.anzahl += anzahl;
   }

   public int getAnzahl()
   {
     return anzahl;
   }

   public void print()
   {
      System.out.print ("ID: " + nummer + "  Bezeichnung: " + name +
                        "  Anzahl: " + anzahl);
   }
}
```

Die folgende Klasse `Warenlager` stellt zwei Methoden zur Verfügung:

- Methode `aufnehmen()`
- und die Methode `ausgeben()`.

Die Methode `aufnehmen()` erlaubt es, neue Waren ins Lager aufzunehmen oder bereits im Lager vorhandene Artikel nachzufüllen. Als Übergabeparameter erwartet diese Methode eine Referenz auf ein Objekt vom Typ `Ware`.

Die Methode `ausgeben()` ermöglicht die Ausgabe des gesamten Lagerinhalts auf dem Bildschirm. Fett hervorgehoben sind hier einige Programmzeilen, in denen der Typ `Ware` verwendet wird.

Jetzt die Klasse `Warenlager`:

```
// Datei: Warenlager.java
public class Warenlager
{
   protected Ware[] arr;
   public Warenlager (int max)
   {
      arr = new Ware [max];
   }

   // Die Methode aufnehmen() kann neue, noch nicht im Lager enthal-
   // tene Waren aufnehmen. Zu einer schon im Lager befindlichen Wa-
   // re wird die Anzahl der vorhandenen Exemplare erhöht. Das Array
   // wird beginnend vom Index 0 ab gefüllt. Die freien Array-Ele-
   // mente enthalten die null-Referenz. Wird die Ware erfolgreich
   // aufgenommen, wird der Wert 1, ansonsten -1 zurückgegeben.
   public int aufnehmen (Ware neueWare, int anzahl)
   {
      for (Ware ware : arr)  // Prüfen, ob Ware schon vorhanden.
      {
         if((ware!=null)&& (ware.getNummer()==neueWare.getNummer()))
         {
               ware.stueckzahlErhoehen (anzahl);
               return 1;
         }
      }

      if (arr [arr.length - 1] != null) // Warenlager voll!
         return -1;

      for (int i = 0; i < arr.length; i++)
      {
         if (arr [i] == null) // Erstes freies Feld gefunden - hier
         {                    // kann die Ware eingefügt werden.
            arr [i] = neueWare;
            arr [i].stueckzahlErhoehen (anzahl);
            break;
         }
      }
      return 1;
   }

   public void ausgeben()
   {
      for (Ware ware : arr)
      {
         if (ware == null) break;
         ware.print();
         System.out.println();
      }
   }
}
```

Die soeben gezeigten Klassen könnten jetzt in einer Bibliothek zur Verfügung gestellt werden und durch gezielte Ableitung an einen speziellen Problembereich angepasst werden. Für einen Milchlieferanten gibt es zum Beispiel eine Klasse `Milch` und eine

Klasse `Joghurt`, die von der Klasse `Ware` abgeleitet sind. Für einen Lieferanten von Drogeriewaren sind dagegen ganz andere Klassen von Bedeutung. Exemplarisch wird hier eine von der Klasse `Ware` abgeleitete Klasse `Milch` gezeigt:

```
// Datei: Milch.java
import java.util.GregorianCalendar;

public class Milch extends Ware
{
   private String typ;

   // Die Klasse GregorianCalendar aus dem Paket java.util
   // ermöglicht die Speicherung und Bearbeitung von Datumswerten!
   private GregorianCalendar verfallsDatum;
   private double maxLagerTemperatur;

   public Milch (String typ, float preis,
                 GregorianCalendar verfallsDatum, double maxTemp)
   {
      super ("Milch", preis);
      this.typ = typ;
      this.verfallsDatum = verfallsDatum;
      this.maxLagerTemperatur = maxTemp;
   }

   // Überschreiben der print()-Methode der Klasse Ware
   public void print()
   {
      super.print();
      System.out.print (" Typ: " + typ);
   }
   // weitere spezifische Methoden für die Klasse Milch!
}
```

Objekte der Klasse `Milch` können an die Methode `aufnehmen()` der Klasse `Warenlager` übergeben werden. Dies funktioniert aus dem Grund, weil die Klasse `Milch` eine Spezialisierung der Klasse `Ware` darstellt und die Klasse `Warenlager` mit Referenzen auf Objekte der Klasse `Ware` arbeitet. Das folgende Testprogramm zeigt, wie sich Objekte der Klasse `Milch` als Objekte der Klasse `Ware` verhalten und durch diese Eigenschaft den gesamten Code der Klasse `Warenlager` mitbenutzen können:

```
// Datei: Test8.java
import java.util.GregorianCalendar;

public class Test8
{
   public static void main (String[] args)
   {

      final int anzahl1 = 50;
      final int anzahl2 = 200;
      final int anzahl3 = 300;
      final int anzahl4 = 500;
      final int anzahl5 = 1000;
```

```
      // Erzeugen eines Warenlagers für 4 verschiedene Warengruppen.
      Warenlager lager = new Warenlager (4);

      // Die Klasse java.util.GregorianCalendar ermöglicht die
      // Speicherung und Bearbeitung von Datumswerten.
      // Der erste Parameter gibt das Jahr an, der zweite Parameter
      // den Monat und der dritte den Tag.
      GregorianCalendar date = new GregorianCalendar (1, 5, 5);

      System.out.println ("Mit dem Einlagern wird begonnen");
      Milch milch = new Milch ("Fettarme Milch", 0.6f, date, 7.0);

      if (lager.aufnehmen (milch, anzahl4) < 0)
         System.out.println ("Lager voll");
      else
         System.out.println(anzahl4 +" Fettarme Milch eingelagert");

      if(lager.aufnehmen (new Milch ("Frischmilch", 0.8f, date, 6.0)
                        , anzahl5) < 0)
         System.out.println ("Lager voll");
      else
         System.out.println (anzahl5 + " Frischmilch eingelagert");

      if (lager.aufnehmen (new Milch ("H-Milch", 0.5f, date, 7.5)
                        , anzahl4) < 0)
         System.out.println ("Lager voll");
      else
         System.out.println (anzahl4 + " H-Milch eingelagert");

      if (lager.aufnehmen (milch, anzahl3) < 0)
         System.out.println ("Lager voll");
      else
         System.out.println(anzahl3 +" Fettarme Milch eingelagert");

      if (lager.aufnehmen (new Milch ("Dosenmilch", 8.8f, date, 18)
                        , anzahl2) < 0)
         System.out.println ("Lager voll");
      else
         System.out.println (anzahl2 + " Dosenmilch eingelagert");

      if (lager.aufnehmen (new Milch ("Kakao", 9.9f, date, 18)
                        , anzahl1) < 0)
         System.out.println ("Lager voll");
      else
         System.out.println (anzahl1 + " Kakao eingelagert");

      System.out.println ("\nDer Gesamtbestand des Lagers ist");
      lager.ausgeben();
   }
}
```

Die Ausgabe des Programms ist:

```
Mit dem Einlagern wird begonnen
500 Fettarme Milch eingelagert
1000 Frischmilch eingelagert
500 H-Milch eingelagert
300 Fettarme Milch eingelagert
200 Dosenmilch eingelagert
Lager voll

Der Gesamtbestand des Lagers ist
ID: 0  Bezeichnung: Milch  Anzahl: 800 Typ: Fettarme Milch
ID: 1  Bezeichnung: Milch  Anzahl: 1000 Typ: Frischmilch
ID: 2  Bezeichnung: Milch  Anzahl: 500 Typ: H-Milch
ID: 3  Bezeichnung: Milch  Anzahl: 200 Typ: Dosenmilch
```

Das Besondere ist nun, dass innerhalb der Methode `ausgeben()` der Klasse `Warenlager` ein Array vom Typ `Ware` durchlaufen und für jedes Objekt in diesem Array, also für jedes Produkt, die `print()`-Methode aufgerufen wird. Hierbei wird im Falle einer Milch immer die überschriebene Methode der Klasse `Milch` aufgerufen, auch wenn die Referenzen, welche auf diese Objekte zeigen, vom Typ `Ware` sind. Dies hat seine Ursache in der dynamischen Bindung (siehe Kapitel 11.4.2.2).

11.4.3.3 Theoriebeispiel für eine Klassenhierarchie

Hier noch ein weiteres **Beispiel für** ein polymorphes Verhalten beim Überschreiben:

Bild 11-22 Vererbungshierarchie zur Veranschaulichung des liskovschen Substitutionsprinzips

Nach dem liskovschen Substitutionsprinzip kann eine Referenz auf ein Objekt einer Superklasse stets auch auf ein Objekt einer Subklasse zeigen, wenn der Programmierer die Verträge der Methoden beim Überschreiben einhält. Geht man von der in Bild 11-22 gezeigten Vererbungshierarchie aus, so können in einem Array aus

Referenzen auf Objekte der Klasse `Object` auch Referenzen auf Objekte der Klassen `X`, `A`, `B`, `C` und `D` gespeichert werden. Weiterhin können in einem Array aus Referenzen auf Objekte der Klasse `A` außer Referenzen auf Objekte der Klasse `A` auch Referenzen auf Objekte der Klasse `B`, `C` und `D` hinterlegt werden. Hier das bereits erwähnte Bild 11-22:

11.4.3.4 Kleines Programmbeispiel für ein Array mit Überschreiben

Das folgende **Beispiel** veranschaulicht das liskovsche Substitutionsprinzip für ein Array aus Referenzen auf Objekte der Klasse `Person6`. Da ein Student eine Person ist, können in diesem Array auch Referenzen auf Objekte der Klasse `Student6` gespeichert werden, weil beim Überschreiben der Methode `print()` keine Vertragsverletzung erfolgt. Die Methode `print()` gibt für das jeweilige Objekt die entsprechenden Daten aus. Handelt es sich um ein Objekt der Klasse `Person6`, wird die `print()`-Methode der Klasse `Person6` aufgerufen, handelt es sich um ein Objekt der Klasse `Student6`, wird die überschreibende `print()`-Methode der Klasse `Student6` aufgerufen. Hier das Programm:

```
// Datei: Person6.java
public class Person6                    // dies ist die Vaterklasse
{
   private String nachname;             // Datenfeld nachname
   private String vorname;              // Datenfeld vorname

   public Person6 (String nachname, String vorname)
   {
      this.nachname = nachname;
      this.vorname = vorname;
   }

   public void print()
   {
      System.out.println ("Nachname:  " + nachname);
      System.out.println ("Vorname:   " + vorname);
   }
}

// Datei: Student6.java
public class Student6 extends Person6   // dies ist die Sohnklasse
{
   private int matrikelnummer;

   public Student6 (String nachname, String vorname,
                    int matrikelnummer)
   {
      super (nachname, vorname);
      this.matrikelnummer = matrikelnummer;
   }

   public void print()
   {
      super.print();
      System.out.println ("Matrikelnummer: " + matrikelnummer);
   }
}
```

```
// Datei: Test6.java
public class Test6
{
   public static void main (String[] args)
   {
      Person6[] pa = new Person6 [3];
      pa [0] = new Person6 ("Brang", "Rainer");
      pa [1] = new Student6 ("Müller", "Peter", 123456);
      pa [2] = new Person6 ("Mayer", "Carl");
      for (Person6 person : pa)
      {
         person.print();
         System.out.println ("");
      }
   }
}
```

Die Ausgabe des Programms ist:

```
Nachname:  Brang
Vorname:   Rainer

Nachname:  Müller
Vorname:   Peter
Matrikelnummer: 123456

Nachname:  Mayer
Vorname:   Carl
```

Betrachtet man die Ausgabe des Programms näher, so stellt man fest, dass obwohl der zweite Aufruf der `print()`-Methode auf einer Referenzvariablen vom Typ `Person6` erfolgte, die `print()`-Methode der Klasse `Student6` aufgerufen wurde. Dieses Verhalten ist vielleicht auf den ersten Blick etwas überraschend. Mit etwas Überlegung erkennt man jedoch den Grund dafür. Wenn z. B. unterschiedliche Objekte, die eine gemeinsame Basisklasse haben, von einem Array aus Referenzen auf Objekte von Basisklassen aus referenziert werden sollen, so ist von außen nicht erkennbar, auf welche Objekte die im Array hinterlegten Referenzen im einzelnen zeigen. Wenn man jedoch über die in dem Array gespeicherten Referenzen eine Methode aufruft, die in den abgeleiteten Klassen und ihrer Basisklasse verschieden definiert ist, so muss gewährleistet sein, dass die für die Klasse des vorliegenden Objektes implementierte Methode aufgerufen wird, auch wenn man gar nicht weiß, von welchem Typ das Objekt eigentlich ist.

Damit beim Überschreiben von Methoden – wie erwartet – die überschreibende Methode eines Objektes der abgeleiteten Klasse aufgerufen werden kann, benötigt man den Mechanismus der **dynamischen Bindung**.

11.5 Verträge

Entwurf durch Verträge (engl. **Design by Contract**) wurde von Bertrand Meyer, dem Entwickler der Programmiersprache Eiffel, als Entwurfstechnik eingeführt. Diese Technik wurde im Falle von Eiffel in einer konkreten Programmiersprache umgesetzt,

stellt aber ein allgemeingültiges Prinzip dar, das beim objektorientierten Entwurf unabhängig von der jeweiligen objektorientierten Programmiersprache eingesetzt werden kann.

Eine Klasse besteht nicht nur aus Methoden und Datenfeldern – eine Klasse wird benutzt von anderen Klassen, hier **Kunden** genannt, und hat damit Beziehungen zu all ihren Kunden.

Das Konzept "Design by Contract" sieht diese Beziehungen als eine formale Übereinkunft zwischen den beteiligten Partnern an und definiert präzise, unter welchen Umständen ein korrekter Ablauf des Programms erfolgt.

Worum es bei Design by Contract vor allem geht, ist, dass sich beim **Aufruf einer Methode der Aufrufer und die aufgerufene Methode gegenseitig aufeinander verlassen können**.

Die Beziehung zwischen Aufrufer und aufgerufener Methode kann man formal als einen **Vertrag einer Methode** bezeichnen, der nicht gebrochen werden darf, da ansonsten eine Fehlersituation entsteht.

Bei einem Vertrag haben in der Regel **beide Seiten Rechte und Pflichten**. So wie im Alltag ein Vertrag die Beziehungen zwischen Vertragsparteien (Personen, Organisationen) regelt, beschreibt ein Vertrag die Beziehung eines Objekts einer Klasse mit seinem Kunden, dem Aufrufer.

Solange bei der Ableitung von einer Basisklasse der Vertrag der Basisklasse in einer Unterklasse nicht gebrochen wird, ist es möglich, den für die Basisklasse geschriebenen Code auch für die Unterklassen, die eventuell erst später erfunden werden, zu verwenden.

Das folgende Bild soll den für den Client unsichtbaren Objekttausch symbolisieren:

Bild 11-23 Liskovsches Substitutionsprinzip

Das Client-Objekt bemerkt den Tausch der Objekte nicht, solange der Vertrag nicht gebrochen wird.

11.5.1 Zusicherungen

Allgemein werden nach Bertrand Meyer **Verträge** durch sogenannte **Zusicherungen** spezifiziert. Zusicherungen bestehen aus Bedingungen über die Zustände von Objekten an einer bestimmten Programmstelle.

Entwurf durch Verträge verwendet drei verschiedene Arten von Zusicherungen:

- **Vorbedingungen** von **Methoden**,
- **Nachbedingungen** von **Methoden**
- und **Invarianten von Klassen**.

Der **Vertrag einer Methode** umfasst die Vor- und Nachbedingungen einer Methode.

Vor- und Nachbedingungen von Methoden werden in Kapitel 11.5.1.1 besprochen. Invarianten von Klassen werden in Kapitel 11.5.1.2 behandelt.

11.5.1.1 Vor- und Nachbedingungen einer Methode

Eine Vorbedingung stellt die Einschränkungen dar, unter denen eine Methode korrekt funktioniert. So darf beispielsweise eine Methode `push()`, die ein Element auf einem Stack ablegt, nicht aufgerufen werden, wenn der Stack voll ist, genauso wenig wie eine Methode `pop()`, die ein Element von einem Stack abholen soll, aufgerufen werden darf, wenn kein Element mehr auf dem Stack ist.

Eine Vorbedingung stellt eine Pflicht für einen Aufrufer dar, sei es, dass der Aufruf innerhalb der eigenen Klasse erfolgt oder von einem Kunden.

Ein korrekt arbeitendes System führt nie einen Aufruf in einem Zustand durch, der nicht die Vorbedingung der gerufenen Methode erfüllen kann. Eine Vorbedingung bindet also einen Aufrufer.

Eine **Vorbedingung** definiert die Bedingungen, unter denen ein Aufruf zulässig ist. Sie stellt eine **Pflicht für den Aufrufer** dar und einen **Nutzen für den Aufgerufenen**.

Ist die **Vorbedingung** verletzt, so ist der Aufgerufene nicht an die Nachbedingung gebunden und kann machen, was er will. Zum Beispiel kann die Verletzung der Vorbedingung einen Programmabsturz verursachen.

Eine **Nachbedingung** stellt den korrekten Zustand nach dem Aufruf einer Methode dar. So kann nach dem Aufruf von `push()` der Stack nicht leer sein und die Zahl der

Elemente auf dem Stack muss um 1 höher sein als vor dem Aufruf der Methode. Umgekehrt kann nach dem Aufruf von `pop()` der Stack leer sein, wobei die Zahl der Elemente auf dem Stack um 1 geringer sein muss als vor dem Aufruf.

Eine **Nachbedingung** bindet eine Methode einer Klasse. Eine Nachbedingung stellt die Bedingung dar, die von einer Methode eingehalten werden muss. Eine **Nachbedingung** ist eine **Pflicht für den Aufgerufenen** und ein **Nutzen für den Aufrufer**.

Mit der Nachbedingung wird garantiert, dass der Aufrufer nach Ausführung einer Methode einen Zustand mit korrekten Eigenschaften vorfindet, natürlich immer unter der Voraussetzung, dass beim Aufruf der Methode die Vorbedingung erfüllt war.

Wichtig ist, dass kein redundanter Code geschrieben wird. Das wäre zu fehlerträchtig und außerdem nicht performant. Es gilt somit das **single source-Prinzip**. Die Vorbedingung muss stets vom Aufrufer geprüft werden und keinesfalls vom Aufgerufenen. Umgekehrt muss die Einhaltung der Nachbedingung stets vom Aufgerufenen überwacht werden. Der Aufrufer darf die Prüfung der Nachbedingung nicht durchführen.

Wie bei einem guten Vertrag im täglichen Leben haben also **Aufrufer und Aufgerufener Pflichten und Vorteile**.

Der **Aufrufer** hat die Pflicht, den Aufgerufenen korrekt aufzurufen. Damit hat der Aufgerufene den Vorteil, dass er in einer korrekten Umgebung abläuft.

Der **Aufgerufene** wiederum hat die Pflicht, korrekte Werte zurückzugeben. Diese Pflicht des Aufgerufenen ist der Vorteil des Aufrufers, da er korrekte Werte erhält.

11.5.1.2 Invarianten von Klassen

Invarianten beziehen sich nicht auf eine einzelne Methode. Invarianten beziehen sich immer auf das gesamte Objekt. Eine Invariante muss von allen Methoden, die ein Kunde aufrufen kann, für jedes einzelne Objekt einer Klasse erfüllt werden, damit ein System korrekt arbeitet oder in einem korrekten Zustand ist.

Da die Invarianten von **allen Methoden einer Klasse**, die von einem Kunden aufgerufen werden können, eingehalten werden müssen, um die Korrektheit zu gewährleisten, spricht man auch von **Klasseninvarianten**.

Eine **Invariante** ist eine Zusicherung bezüglich einer Klasse. Es soll als Beispiel hierzu eine Klasse `Polygon` betrachtet werden. Ein Polygon besteht aus mindestens drei Eckpunkten, die mit geraden Linien verbunden sind. Somit besitzt die Klasse `Polygon` die Klasseninvariante, dass die Anzahl der aggregierten Punkte – die

Punkte können beispielsweise durch die Klasse `Punkt` repräsentiert werden – mindestens drei betragen muss, damit ein Polygon vorliegt. Diese Eigenschaft gilt für die gesamte Klasse und nicht individuell nur für eine einzelne Methode. Sie ist damit eine Klasseneigenschaft im Gegensatz zu Vor- und Nachbedingungen, die jeweils einzelne Methoden charakterisieren.

Eine Invariante muss gelten **vor Aufruf einer Methode und nach dem Aufruf einer Methode** durch einen Kunden. Eine Invariante kann temporär verletzt werden während der Ausführung einer Methode oder beim Aufruf von Service-Methoden, die nicht außerhalb der Klasse sichtbar sind – also nicht exportiert werden. Dies stellt kein Problem dar, da die Invariante dem Kunden erst nach Ausführung einer exportierten Methode wieder zur Verfügung steht. Nach Ausführung einer exportierten Methode muss die Klasseninvariante wieder eingehalten sein.

So hat zum Beispiel eine Klasse `Quadrat` – die Quadrate auf dem Bildschirm zeichnen, verschieben, drehen und skalieren kann – die Invariante, dass vor und nach dem Aufruf einer der Methoden `zeichne()`, `verschiebe()`, `drehe()` und `skaliere()` alle Seiten des Quadrats gleich lang sind und jeder Winkel ein rechter Winkel ist. Innerhalb der Methode `verschiebe()` kann aber temporär erst ein Teil der Eckpunkte verschoben sein, sodass temporär gar kein Quadrat vorliegt. Das sieht das "Kundenprogramm" aber nicht.

Eine **Klasseninvariante** muss vor und nach dem Aufruf einer nach außen sichtbaren Methode (Export-Methode) eingehalten sein.

Werden Methoden intern aufgerufen, wird eine Invariante nicht geprüft. Wenn Methoden von außen aufgerufen werden, sollte die Invariante überprüft werden, um sich der Korrektheit zu vergewissern, denn Verträge müssen eingehalten werden.

Der **Vertrag einer Klasse** umfasst die Verträge der Methoden und die Invarianten der Klasse. Werden verschiedenen Kunden einer Klasse jedoch verschiedene Leistungen der Klasse zur Verfügung gestellt, so ordnet man die Verträge der Methoden in verschiedene Verträge der Klasse jeweils mit dem entsprechenden Kunden ein.

Invarianten gelten bei exportierten Methoden vor und nach dem Aufruf, bei Aufruf von Konstruktoren nach dem Aufruf.

11.5.2 Einhalten der Verträge

Leitet – wie in Bild 11-24 gezeigt – eine Klasse `B` von einer Klasse `A` ab, so müssen beim Überschreiben der Methoden bestimmte Regeln für die Verträge der Methoden eingehalten werden. Ferner müssen auch die Klasseninvarianten beim Überschreiben von Export-Methoden eingehalten werden.

11.5.2.1 Regeln für das Einhalten der Verträge von Methoden

Beim Überschreiben von Methoden dürfen die Verträge nicht gebrochen werden, da sonst das liskovsche Substitutionsprinzip verletzt wird.

Überschreibende Methoden dürfen die **Vorbedingung** einer überschriebenen Methode **nur aufweichen** und nicht verschärfen, da sonst ein Aufrufer damit nicht fertig werden würde. Überschreibende Methoden dürfen **Nachbedingungen nur verschärfen**, da mit aufgeweichten Nachbedingungen ein Aufrufer nicht leben könnte.

Im Folgenden soll die Vererbungshierarchie aus Bild 11-24 betrachtet werden:

Bild 11-24 Überschreiben einer Methode

Die Klasse `B` sei von der Klasse `A` abgeleitet und soll die Methode `g()` aus `A` überschreiben. Aufrufer von `g()` sei eine Methode `f()` in einer Klasse `C`, der Client-Klasse. Die Methode `f()` soll die folgende Aufrufschnittstelle besitzen: `f (A a)`. Mit anderen Worten: an `f()` kann eine Referenz auf ein Objekt der Klasse `A` oder eine Referenz auf ein Objekt der abgeleiteten Klasse `B` übergeben werden.

Das folgende Bild soll symbolisieren, dass eine Methode `f()` Referenzen auf Objekte vom Typ `A` und vom Typ `B` akzeptiert:

Bild 11-25 Eine Methode `f()` akzeptiert Referenzen auf Objekte vom Typ `A` und Typ `B`

Der Kunde `f()` kann zur Laufzeit nicht wissen, ob ihm eine Referenz auf ein Objekt der Klasse `A` oder der Klasse `B` übergeben wird. Dem Kunden `f()` ist auf jeden Fall nur die Klasse `A` bekannt und daran richtet er sich aus! Also kann `f()` nur den Vertrag der Methode `g()` aus `A` beachten. `f()` stellt also die Vorbedingungen für `g()` aus `A` sicher und erwartet im Gegenzug, dass `g()` aus `A` seine Nachbedingungen erfüllt. Das folgende Bild visualisiert die Aufweichung einer Vorbedingung in einer abgeleiteten Klasse:

Bild 11-26 Aufweichen einer Vorbedingung in einer abgeleiteten Klasse

Wie im täglichen Leben auch, darf ein Vertrag übererfüllt, aber nicht verletzt werden! Dies hat zur Konsequenz, dass `g()` aus `B` die **Vorbedingungen nicht verschärfen** kann, denn darauf wäre der Kunde `f()` überhaupt nicht eingerichtet. `g()` aus `B` darf **aber** die **Vorbedingungen aufweichen**, wie in Bild 11-26 gezeigt wird. Dies stellt für `f()` kein Problem dar, denn aufgeweichte Vorbedingungen kann `f()` sowieso mühelos einhalten.

In entsprechender Weise liegt es auf der Hand, dass `g()` aus `B` die Nachbedingungen nicht aufweichen darf, denn der Kunde `f()` erwartet die Ergebnisse in einem bestimmten Bereich. Auf einen breiteren Bereich wäre der Kunde nicht eingerichtet, was im Bild 11-27 verdeutlicht wird:

Bild 11-27 Verschärfen einer Nachbedingung in einer abgeleiteten Klasse

Beispiele

Eine Methode einer abgeleiteten Klasse darf:

- eine **Nachbedingung nicht aufweichen**, d. h., wenn eine Methode z. B. einen Rückgabewert vom Typ `int` hat und garantiert, dass sie nur Werte zwischen 1 und 10 liefert, so darf die überschreibende Methode keine Werte außerhalb dieses Bereichs liefern.
- eine **Vorbedingung nicht verschärfen**, d. h., wenn eine Methode z. B. einen formalen Parameter vom Typ `int` spezifiziert und einen gültigen Wertebereich zwischen 1 und 10 hat, so darf die überschreibende Methode diesen Wertebereich nicht einschränken.

11.5.2.2 Regeln für das Einhalten der Gültigkeit von Klasseninvarianten

Beim Erweitern einer Klasse muss darauf geachtet werden, dass die von ihr ableitenden Klassen die Gültigkeit der Klasseninvarianten der Basisklasse nicht verletzen. Da eine Sohnklasse immer einen Vateranteil enthält, muss sichergestellt werden, dass der Vateranteil nach wie vor korrekt arbeitet. Aus diesem Grund gilt bei der Vererbung die Regel, dass sich die Invarianten einer abgeleiteten Klasse aus der booleschen UND-Verknüpfung der Invarianten der Basisklasse und der in der abgeleiteten Klasse definierten Invarianten ergeben. Ein Client, der ausschließlich mit Referenzen auf Objekte der Basisklasse arbeitet, muss sich auf die Invarianten der Vaterklasse verlassen können. Tritt an die Stelle eines Objektes einer Basisklasse ein Objekt einer abgeleiteten Klasse, so darf dieses Objekt die Invarianten der Basisklasse nicht verletzen, da der Client nicht damit zurecht kommen würde.

Hierzu wird wieder das in Kapitel 11.5.1 beschriebene Beispiel der Klasse `Polygon` betrachtet, dessen Invariante für die Anzahl an aggregierten Eckpunkten "mindestens drei Punkte" lautet. Von der Klasse `Polygon` leitet nun die Klasse `Viereck` ab. Das Viereck definiert nun eine Invariante für die Anzahl der aggregierten Punkte, welche lautet: "genau vier Punkte". Mit anderen Worten, ein Objekt der Klasse `Viereck` muss genau vier Objekte vom Typ `Punkt` aggregieren, damit es ein regelgerechtes Viereck darstellt. Die Invariante des Vaters ist dadurch nicht verletzt. Wenn dem Client nun eine Referenz auf ein Objekt der Klasse `Viereck` zugewiesen wird, so kann er mit diesem Viereck ohne Probleme arbeiten. Denn er weiß, dass die Klasseninvariante von Polygonen "mindestens drei Punkte" lautet. Ein Viereck hat genau vier Punkte, also auch mindestens drei. Die boolesche UND-Verknüpfung

"mindestens drei Punkte" && "genau vier Punkte"

hat für das Viereck den Wahrheitswert TRUE. Somit wurde die Invariante der Basisklasse `Polygon` von der abgeleiteten Klasse `Viereck` nicht verletzt.

Die Invarianten einer abgeleiteten Klasse ergeben sich aus der booleschen UND-Verknüpfung ihrer Invarianten und der Invarianten ihrer Basisklasse.

11.6 Identifikation der Klasse eines Objektes

Wie Sie in den letzten Kapiteln feststellen konnten, kann es vorkommen, dass man den Typ eines Objektes nicht kennt, auch wenn man eine Referenz hat, die auf dieses Objekt zeigt. Beispielsweise kann eine Referenz vom Typ `Object` auf jedes beliebige Objekt zeigen. Um den tatsächlichen Typ eines Objektes herausfinden zu können oder um testen zu können, ob ein Objekt von einem bestimmten Typ ist, gibt es Mechanismen, die in den folgenden beiden Kapiteln vorgestellt werden.

11.6.1 Der `instanceof`-Operator

Mit dem `instanceof`-Operator kann getestet werden, ob eine Referenz auf ein Objekt eines bestimmten Typs zeigt. Dies ist beispielsweise dann wichtig, wenn eine Referenz vom Typ einer Basisklasse ist. Eine solche Referenz kann ja auf Objekte aller abgeleiteten Klassen zeigen. Mit Hilfe des `instanceof`-Operators lässt sich nun nachprüfen, ob das referenzierte Objekt tatsächlich vom angenommenen Typ ist.

Mit dieser Erkenntnis kann dann die Referenz in den entsprechenden Typ gecastet werden. Das heißt, man kann überprüfen, ob ein expliziter Cast zulässig ist.

Die Syntax ist:

```
a instanceof Klassenname
```

Dieser Ausdruck gibt `true` zurück, wenn die Referenz `a` auf ein Objekt der Klasse `Klassenname` – bzw. auf ein Objekt, **dessen Klasse von der Klasse** `Klassenname` **abgeleitet ist** – zeigt. Betrachten Sie hierzu die Vererbungshierarchie aus Bild 11-21 mit den Klassen `Grossvater`, `Vater` und `Sohn`. Es zeigen nun Referenzen vom Typ `Object` auf Objekte aller drei Klassen:

```
Object refA = new Grossvater();
Object refB = new Vater();
Object refC = new Sohn();
```

Dann geben alle drei Ausdrücke

```
refA instanceof Grossvater
refB instanceof Grossvater
refC instanceof Grossvater
```

`true` zurück, da sowohl ein Objekt vom Typ `Vater` als auch ein Objekt vom Typ `Sohn` vom Typ `Grossvater` ist. Wird die Referenz `refB` getestet, ob sie auf ein Objekt vom Typ `Object`, `Grossvater`, `Vater` oder `Sohn` zeigt, so geben die Vergleiche

```
refB instanceof Object
refB instanceof Grossvater
refB instanceof Vater
```

`true` zurück. Dahingegen liefert der Vergleich

```
refB instanceof Sohn
```

den Wert `false`.

Die `null`-Referenz zeigt auf kein Objekt eines bestimmten Typs, deshalb ist

```
null instanceof Klassenname
```

immer `false`. Hier ein Beispiel für die Verwendung des `instanceof`-Operators:

```
// Test, ob ein Cast zulässig ist.
if (ref instanceof Sohn)
{
   Sohn refSohn = (Sohn) ref;
   . . . . .
}
```

11.6.2 Run Time Type Identification

Run Time Type Identification (RTTI) ist die **Erkennung** des **Typs** eines Objektes **zur Laufzeit**.

In Bild 11-1 wurde die Vererbungshierarchie für eine Person und einen Studenten gezeigt. Die Klasse `Student` ist dabei von der Klasse `Person` abgeleitet. Beide Klassen definieren eine Methode `print()`.

Tritt ein Objekt der Klasse `Student` als Person auf, so sind die zusätzlichen Datenfelder und Methoden der Klasse `Student` zwar nicht mehr ansprechbar, wird aber die Methode `print()` zu dem Studenten aufgerufen, der gerade in Gestalt einer Person auftritt, so wird die überschreibende `print()`-Methode des Studenten und nicht die überschriebene Methode der Person aufgerufen. Es wird also zur Laufzeit erkannt, dass die Person ja eigentlich ein Student ist.

Wie dies von der Java-Virtuellen Maschine erreicht wird, soll im Folgenden aufgezeigt werden. Als Diskussionsgrundlage sollen nicht die Klassen `Person` und `Student` dienen, sondern eine besonders einfache Klasse, die Klasse `Test11`. Die Klasse `Test11` soll nur die Instanzmethode `toString()` besitzen, eine `main()`-Methode sowie das Datenfeld `var`. Die `toString()`-Methode der Klasse `Object` wird dabei in der Klasse `Test11` überschrieben. Jetzt die Klasse `Test11`:

```
// Datei: Test11.java

public class Test11 extends Object
{
   private int var = 1;
```

```
    public String toString()
    {
       return Integer.toString (var);
    }

    public static void main (String[] args)
    {
       Object ref = new Test11();
       System.out.println (ref);
    }
}
```

Die Ausgabe des Programms ist:

```
1
```

Es wird – wie zu erwarten – die `toString()`-Methode der Klasse `Test11` aufgerufen und nicht die geerbte `toString()`-Methode der Klasse `Object`. Die Java-Virtuelle Maschine muss also so organisiert sein, dass dieses Verhalten möglich ist. Bekanntlich liegen die Instanzvariablen eines Objektes im Heap und die Klassenvariablen und der Bytecode für die Methoden in der Method-Area. Bis zu diesem Zeitpunkt wurde zwar schon erwähnt, dass ein jedes Objekt seine Klasse kennt, aber es wurde immer verschwiegen, wie dies realisiert ist – und dabei ist es ganz einfach.

Die erste Information, die zu einem Objekt im Heap abgelegt wird, ist ein Zeiger auf die in der Method-Area liegende Klasse des Objektes. Erst dann folgen die Instanzvariablen.

Bild 11-28 soll diesen Zusammenhang zeigen:

Bild 11-28 Objekte im Heap zeigen auf die zugehörige Klasse in der Method-Area

Das obige Bild ist eine vereinfachte Darstellung und soll im Folgenden vervollständigt werden.

Damit es möglich wird, jedes Mal die richtige Methode aufzurufen, benötigt jede Klasse noch zusätzlich eine **Methodentabelle**. In dieser Tabelle sind die Zeiger auf die Methodenimplementierungen aller Methoden eines Typs, die dynamisch gebunden werden können, zusammengestellt.

Eine mögliche Realisierung der dynamischen Bindung könnte so wie in Bild 11-29 aussehen:

Bild 11-29 Zeiger in der Methodentabelle zeigen auf den Bytecode einer Methode

Der Zeiger, der im Heap als erste Information vor den Instanzvariablen eines Objektes liegt, zeigt jetzt auf den ersten Eintrag in der Methodentabelle einer Klasse. Damit ist sichergestellt, dass ggfs. eine überschreibende Methode aufgerufen wird. Dort verweist wiederum der erste Eintrag auf den Bytecode der Klasse `Test11`. Wird eine Methode in der Klasse `Test11` überschrieben, so zeigt der Eintrag in der Methodentabelle auf die überschreibende Methode, hier also auf den Bytecode der Methode `toString()` der Klasse `Test11`.

In der Methodentabelle befinden sich nur Zeiger auf die Methoden, die für die dynamische Bindung in Frage kommen. Deshalb haben private, statische oder finale Methoden keinen Eintrag in der Methodentabelle. Damit ist sichergestellt, dass ggfs. eine überschreibende Methode aufgerufen wird.

Wird nun die Methode `toString()` aufgerufen, so kommt man über den Verweis im Heap zur Methodentabelle und von dort zum Bytecode der Methode. Hierbei wird nun auch klar, dass es gar keine Rolle spielt, ob die Referenz vom Typ einer Basisklasse ist – hier `Object` – oder ob die Referenz den genau gleichen Typ trägt wie das Objekt, auf das sie zeigt – es wird immer die gleiche Methodentabelle verwendet, egal von welchem Typ die Referenz ist. Das bedeutet, dass immer die **überschreibende** Methode des Objektes aufgerufen wird. Hierzu soll nochmals folgendes Beispiel diskutiert werden.

Die Klasse `Student` und die Klasse `Person` implementieren beide eine `print()`-Methode. Durch die folgenden beiden Aufrufe wird jeweils die `print()`-Methode der Klasse `Student` aufgerufen:

```
Student refStud = new Student();
refStud.print();
Person refPers = refStud;
refPers.print();
```

Der Aufruf der `print()`-Methode des Studenten ist auch nicht weiter verwunderlich, denn das Objekt und die zugehörige Methodentabelle in der Method-Area repräsentiert einen Studenten und keine Person, auch wenn die Referenz vom Typ `Person` ist.

Beim Casten verändert sich nur die Sichtweise auf die zur Verfügung stehenden Methodenköpfe. Das Objekt, auf das eine Referenz zeigt, und die zugehörige Methodentabelle bleiben beim Casten unverändert. Allerdings werden beim Cast auf eine Superklasse die erweiternden Methoden der abgeleiteten Klassen unsichtbar.

11.7 Operatoren für Referenztypen

Nachdem nun alle Operatoren, die auf Referenzen angewandt werden können, vorgestellt wurden, erfolgt hier nochmals eine übersichtliche Zusammenfassung:

- cast-Operator (siehe Kapitel 11.3.1).
- `instanceof`-Operator (siehe Kapitel 11.6.1).
- Der **Punkt-Operator `.`** wird auf eine Referenz angewandt, wenn ein **Datenfeld** eines Objektes **angesprochen** werden soll. Gleichermaßen findet der Punkt-Operator Anwendung, wenn über eine Referenz eine **Methode eines Objektes aufgerufen wird**.
- Der **Gleichheitsoperator** `==` und der **Ungleichheitsoperator** `!=` können ebenso für elementare Datentypen und für Referenztypen eingesetzt werden. Der Ausdruck `ref1 == ref2` liefert dabei den **Rückgabewert** `true`, **wenn** `ref1` **und** `ref2` **auf das gleiche Objekt zeigen** und `false`, wenn sie auf verschiedene Objekte zeigen. Der Ungleichheitsoperator liefert genau die entgegengesetzten Ergebnisse. Im Falle von Aufzählungstypen können Aufzählungskonstanten auf Gleichheit oder Ungleichheit geprüft werden (siehe Kapitel 6.6).
- Wird der Ausdruck `ref1 + ref2` in einem Programmstück geschrieben, so ist dies ein gültiger Ausdruck, sofern mindestens eine der Referenzen auf ein `String`-Objekt zeigt. Der Operator `+` wird dann als **Verkettungsoperator für `String`-Objekte** (**String-Concatenation-Operator**) bezeichnet. Der Rückgabewert eines solchen Ausdrucks ist eine Referenz auf ein `String`-Objekt, das die Aneinanderreihung der String-Repräsentationen der Objekte enthält, auf die `ref1` und `ref2` zeigen. Die String-Repräsentation eines Referenztyps wird erzeugt, indem die `toString()`-Methode des entsprechenden Objektes aufgerufen wird. Diese Methode ist bei jedem Objekt vorhanden, da sie in der Klasse `Object` implementiert ist. Jede Klasse hat die Möglichkeit, diese Methode zu überschreiben, um eine für die jeweiligen Objekte einer Klasse geeignete String-Repräsentation zur Verfügung zu stellen. Überschreibt eine Klasse die `toString()`-Methode nicht, so wird die `toString()`-Methode der Klasse `Object` aufgerufen. Der Rückgabewert dieser Methode ist der Name der Klasse, von deren Typ das Objekt ist, gefolgt von dem Zeichen `'@'` und einer Nummer, welche die Identität des Objektes in codierter Form widerspiegelt.
- Beim **Bedingungsoperator** `A ? B : C` können die Ausdrücke `B` und `C` Referenztypen sein. Der Bedingungsoperator wurde ausführlich in Kapitel 7.6.7

behandelt. Die Bedingung `A` kann auch eine Referenz auf ein Objekt vom Typ `Boolean` sein.
- Mit dem **Zuweisungsoperator** kann einer Referenzvariablen eine typkompatible Referenz (siehe Kapitel 11.3.1) oder die `null`-Referenz zugewiesen werden.

11.8 Konsistenzhaltung von Quell- und Bytecode

Der Ersteller eines Programms muss selbst darauf achten, dass der auszuführende Bytecode nicht älter als der Quellcode seiner Klassen ist. Darauf wird im Folgenden eingegangen. Es reicht aber überhaupt nicht aus, nur an seine eigenen Klassen zu denken. Wenn man effizient arbeitet, verwendet man des Öfteren Basisklassen als Ausgangspunkt für seine eigenen Klassen. Was aber, wenn die Basisklassen nach dem Kompilieren des Programmsystems geändert werden? Zu all diesen Problemen soll im Folgenden Stellung bezogen werden:

- Der einfachste Fall liegt vor, wenn man nur eine einzige Klasse hat. Hier ist der Programmierer natürlich jedes Mal selbst dafür verantwortlich, dass er seine Klasse neu kompiliert, wenn er Änderungen an ihr vorgenommen hat und die Ausführung des neuen Codes wünscht.
- Nicht wesentlich komplizierter wird es, wenn mehrere Klassen in einer Aggregationsbeziehung zueinander stehen. Es soll folgendes Beispiel betrachtet werden:

```
// Datei: A.java
public class A
{
   private B bRef;
   // . . . . .
}

// Datei: B.java
public class B
{
   // . . . . .
}
```

Zu beachten ist, dass im Folgenden davon ausgegangen wird, dass Klassen, die nichts miteinander zu tun haben, in jeweils unterschiedlichen Dateien liegen. Natürlich kann man die Konsistenzhaltungsprobleme auf unelegante Art und Weise auch so lösen, dass man alle Klassen in einer einzigen Quellcode-Datei unterbringt. Dies ist aber kein guter Programmierstil!

Innerhalb der Klasse `A` wird im gezeigten Beispiel ein privates Datenfeld der Klasse `B` verwendet. Wird nun die Klasse `B` verändert, so reicht es, die Klasse `A` neu zu kompilieren. Der Compiler sorgt automatisch dafür, dass alle anderen Klassen, die innerhalb von Klasse `A` – egal auf welche Weise – referenziert werden, neu kompiliert werden, wenn die Quellcode-Datei ein neueres Datum als die entsprechende Bytecode-Datei hat. Dieser Vorgang setzt sich rekursiv fort, bis alle verwendeten Klassen mit ihren aktualisierten `.class` Dateien vorliegen. Das klingt soweit wunderbar und äußerst praktisch, aber dieser ganze Mechanismus gerät außer Tritt, sobald entweder mehrere Klassen in einer gemeinsamen `.java`-Datei zusammengefasst werden oder wenn der Name der Quellcode-Datei

nicht dem Klassennamen entspricht. Denn dann hat der Compiler keine Möglichkeit mehr, aufgrund des Klassennamens auf die entsprechende Quellcode-Datei zu schließen, da es in diesen Fällen ja keine Namensgleichheit der Klasse mit der Quellcode-Datei mehr gibt. Hierzu wird nochmals das obige Beispiel betrachtet:

```
// Datei: A.java

public class A
{
   private B bRef;
   . . . . .
}

// Datei: MeineKlasseB.java

class B
{
   . . . . .
}
```

Es existieren demnach die beiden Quellcode-Dateien `A.java` und `MeineKlasseB.java` sowie die beiden Bytecode-Dateien `A.class` und `B.class`. Werden nun beide Quellcode-Dateien verändert und nur die Klasse `A` mit dem Aufruf `javac A.java` kompiliert, so funktioniert die rekursive Kompilierung der Klasse `B` nicht, da keine Quellcode-Datei mit dem Namen `B.java` existiert.

Man sollte sich am besten nie auf die rekursive Kompilierung verlassen und stets selbst eine Gesamtkompilierung durchführen.

- Im dritten Fall wird die Konsistenzhaltung von Quell- und Bytecode im Zusammenhang mit Vererbungshierarchien betrachtet. Hierzu soll das folgende Bild diskutiert werden:

Bild 11-30 Vererbungshierarchie zur Diskussion der Konsistenzprüfung

Die Klasse `Test` aggregiert als Datenfeld ein Objekt der Klasse `Sohn`. Die Klasse `Sohn` ist wiederum von der Klasse `Vater` abgeleitet. Der Programmcode sieht hierzu folgendermaßen aus:

```
// Datei: Vater.java

public class Vater
{
   // . . . . .
}

// Datei: Sohn.java

public class Sohn extends Vater
{
   // . . . . .
}

// Datei: Test.java

public class Test
{
   private Sohn refS;

   public Test()
   {
      refS = new Sohn();
      // . . . . .

   }
   // . . . . .
}
```

Wird nun die Klasse `Vater` entweder an den Schnittstellen oder in den Methodenrümpfen verändert, erfolgt ebenfalls eine Neukompilierung der Klasse `Vater`, wenn

```
javac Test.java
```

aufgerufen wird, obwohl der Quellcode der Klasse `Sohn` nicht verändert wurde. Der Programmierer kann in diesem Fall also stets davon ausgehen, dass immer alle Quellcode-Dateien neu übersetzt werden, an denen Veränderungen vorgenommen wurden, auch wenn die veränderten Dateien nicht direkt von der neu zu übersetzenden Klasse abhängen. Das Verhalten des Compilers kann man sich veranschaulichen, wenn `javac` mit der Option `verbose` aufgerufen wird. `verbose` veranlasst den Compiler dazu, Informationen über seine Tätigkeiten auszugeben. Angenommen, die Klasse `Vater` wird wie folgt verändert:

```
// Datei: Vater.java

public class Vater
{
   // . . . . .
   public void f()      //Diese Methode wurde hinzugefügt
   {
      // Mache etwas
   }
}
```

Dann gibt der Compiler beim Aufruf

```
javac -verbose Test.java
```

folgende Informationen aus[165]:

Die Ausgabe des Programms ist:

```
[parsing started Test.java]
[parsing completed 31ms]
. . .
[loading .\Sohn.class]
[checking Test]
[loading .\Vater.java]
[parsing started .\Vater.java]
[parsing completed 0ms]
[wrote Test.class]
[checking Vater]
[wrote .\Vater.class]
[total 234ms]
```

Es ist zu erkennen, dass der Compiler alle Klassen überprüft, von welchen die Klasse `Test` direkt oder indirekt abhängt, also die Klassen `Vater` und `Sohn`. Die Klasse `Vater` wird neu übersetzt, weil sich deren Quellcode seit der letzten Übersetzung geändert hat.

11.9 Kovarianz und Kontravarianz beim Überschreiben

Beim Überschreiben müssen die Rückgabetypen und Übergabeparameter einer überschreibenden Methode vom Typ der überschriebenen Methode sein.

Hierbei sind Objekte bzw. Referenztypen aufgrund ihrer polymorphen Eigenschaft speziell zu betrachten.

Ein Objekt einer abgeleiteten Klasse ist stets auch vom Typ der Basisklasse, wenn es die Verträge der Basisklasse einhält.

11.9.1 Kovariante Rückgabewerte

Bei Einhaltung der Verträge kann die überschreibende Methode ein Objekt einer abgeleiteten Klasse zurückgeben.

[165] Unwichtige Ausgaben sind durch die Punkte . . . . . ausgelassen worden.

Beim Rückgabewert kann man beim Überschreiben verschärfen, d. h. spezialisieren, denn das macht den Client-Programmen nichts aus. Verschärfen bedeutet die Bildung eines Subtyps durch Ableitung.

Wird in der überschreibenden Methode ein Subtyp des ursprünglichen Typs zurückgegeben, so macht das bei Einhalten der Verträge nichts aus, denn beim Aufrufer tritt dann an die Stelle eines Objektes der Basisklasse ein Objekt einer abgeleiteten Klasse. Und dies ist nach dem liskovschen Substitutionsprinzip möglich. Das Objekt der abgeleiteten Klasse ist ja auch vom Typ der Basisklasse.

Da die Änderung des Typ-Parameters mit der Vererbungsrichtung übereinstimmt, spricht man von einem **kovarianten Rückgabetyp**.

11.9.2 Kontravarianz von Übergabeparametern

Beim Überschreiben einer Methode darf in der überschreibenden Methode für den Typ eines Übergabeparameters der überschriebenen Methode eine Basisklasse gewählt werden. Damit hat ein Kunde bei Einhaltung der Verträge kein Problem, wenn er ein Objekt einer abgeleiteten Klasse übergibt. Denn nach dem liskovschen Substitutionsprinzip kann ein Objekt einer abgeleiteten Klasse stets an die Stelle eines Objektes einer Basisklasse treten, da ein Objekt einer abgeleiteten Klasse stets auch vom Typ der Basisklasse ist, wenn die Verträge eingehalten werden. Würde die überschreibende Methode an die Stelle der Klasse des Übergabeparameters eine von dieser abgeleitete Klasse (Spezialisierung) setzen, so hätten die Client-Programme Schwierigkeiten, da sie solche Objekte nicht liefern könnten.

Beim Überschreiben einer Methode darf also der Typ eines Übergabeparameters nur durch den Typ einer Klasse ersetzt werden, die im Vererbungsbaum des Typs des Übergabeparameters weiter oben steht.

Bei dem Typ eines **Übergabeparameters** ist beim Überschreiben nur eine Generalisierung erlaubt.

Da die Richtung nicht mit der Vererbungsrichtung übereinstimmt, redet man auch von **kontravarianten Übergabeparametern**.

11.9.3 Beispiel für einen kovarianten Rückgabetyp

In diesem Kapitel soll ein Beispiel dafür gegeben werden, dass eine **überschreibende Methode** einen **Subtyp des Rückgabetyps der Basisklasse** zurückgibt. Eine Methode einer abgeleiteten Klasse muss einen Wert zurückliefern, der mit der Basisklasse vereinbar ist. Ein Rückgabetyp darf nie von einem allgemeineren Typ als der Rückgabetyp der Basisklasse sein, sondern muss vom selben oder einem spe-

zielleren Typ sein, da ein spezieller Typ auch vom Typ der Basisklasse ist. Ein mit der Oberklasse kompatibler Rückgabetyp wird auch als **kovarianter Rückgabetyp** bezeichnet.

Das folgende Programm greift das Beispiel des Waren-Management-Systems aus Kapitel 11.4.2.2 auf. Zunächst soll eine Übersicht über die Struktur des beabsichtigten Beispiels gegeben werden:

- Die in Kapitel 11.4.2.2 vorhandene Klasse `Ware` wird in Klasse `Ware2` umbenannt und um die Methode `void stueckzahlVerringern (int anzahl)` zum Auslagern von Waren erweitert.
- Ferner wird eine Klasse `VerderbliceWare` von der Klasse `Ware2` abgeleitet.
- Die in Kapitel 11.4.2.2 vorhandene Klasse `Warenlager` wird durch die Klasse `Warenlager2` ersetzt.
- Weiterhin wird von der Klasse `Warenlager2` eine Klasse `Verderbliches-Warenlager` abgeleitet. Die Klasse `VerderblichesWarenlager` erweitert einerseits die Methoden der Klasse `Warenlager2`, andererseits überschreibt sie die Methode `entnehmen()`.
- Die Klasse `Test10` dient zum Testen des neuen Waren-Management-Systems.

Es wird nun die **allgemeine Situation** betrachtet, dass ein Client-Programm eine Methode `methode()` für ein Objekt einer Klasse `Klasse1` aufruft, die in der von der Klasse `Klasse1` abgeleiteten Klasse `Klasse2` überschrieben wird. Der Rückgabetyp der überschreibenden Methode `methode()` soll ein Subtyp des Rückgabetyps der überschriebenen Methode sein, da bei Einhaltung der Verträge ein Objekt eines abgeleiteten Typs stets auch vom Typ der Basisklasse ist.

Diese allgemeine Situation soll das folgende Bild visualisieren:

Bild 11-31 Subtyp als Rückgabetyp beim Überschreiben

Im folgenden Programm stellt nun die Testklasse `Test10` die Client-Klasse von Bild 11-31 dar. Die Klasse `Klasse1` wird repräsentiert durch die Klasse `Warenlager2`. Die Klasse `Klasse2` wird repräsentiert durch die Klasse `Verderbliches-Warenlager`, die von der Klasse `Warenlager2` abgeleitet wird.

Diese Situation zeigt das folgende Bild:

Bild 11-32 Klassenhierarchie für die Implementierung

Nun wieder zum **Programmbeispiel**:

Ein Warenlager kann ebenso wie ein verderbliches Warenlager prinzipiell sowohl verderbliche als auch nicht verderbliche Waren enthalten. Allerdings weist die Klasse `VerderblichesWarenlager` spezielle Methoden für verderbliche Waren auf. So wird in der Klasse `VerderblichesWarenlager` die Methode `entnehmen()` überschrieben.

In der Klasse `Warenlager2` lautet der **Methodenkopf** der Methode `entnehmen()`:

```
Ware2 entnehmen (String warenname);
```

In der Klasse `VerderblichesWarenlager` lautet der **überschreibende Methodenkopf** für die Methode `entnehmen()`:

```
VerderblicheWare entnehmen (String warenname);
```

Der **Rückgabetyp der überschreibenden Methode** ist also ein **Subtyp des Rückgabetyps der überschriebenen Methode** (kovarianter Rückgabetyp).

Hier nun das Programm:

```
// Datei: Ware2.java
public class Ware2
{
   protected int warenId;
   protected String name;
   protected float preis;
   protected int anzahl;
   protected static int aktuelleNummer = 0;

   public Ware2 (String name, float preis)
   {
      warenId = aktuelleNummer++;
      this.name = name;
      this.preis = preis;
      anzahl = 0;
   }

   public int getWarenId()
   {
      return warenId;
   }
```

```
   public String getName()
   {
      return name;
   }

   public void stueckzahlErhoehen (int anzahl)
   {
      this.anzahl += anzahl;
   }

   public void stueckzahlVerringern (int anzahl)
   {
      this.anzahl -= anzahl;
   }

   public int getAnzahl()
   {
     return anzahl;
   }

   public void setAnzahl(int zahl)
   {
     anzahl = zahl;
   }

   public void print()
   {
      System.out.println();
      System.out.println ("print()-Methode der Klasse Ware2");
      System.out.println ("ID: " + warenId + "  Bezeichnung: "
                          + name + "  Anzahl: " + anzahl);
   }
}
```

Die Klasse `VerderblicheWare` ist ein Subtyp vom Typ `Ware2` mit einer maximalen Lagertemperatur. Hier die Klassendefinition:

```
// Datei: VerderblicheWare.java
public class VerderblicheWare extends Ware2
{
   private double maxLagerTemperatur;

   public VerderblicheWare (String name, float preis,
                    double maxLagerTemperatur)
   {
      super (name, preis);
      this.maxLagerTemperatur = maxLagerTemperatur;
   }

   public void print()
   {
      super.print();
      System.out.println ("print-Erweiterung der Klasse " +
                          "VerderblicheWare");
      System.out.println ("maximale Lagertemperatur " +
                          maxLagerTemperatur);
   }
}
```

Wie Sie sehen, enthält die Klasse `VerderblicheWare` das zusätzliche Attribut `maxLagerTemperatur`, erhält einen eigenen Konstruktor und überschreibt die Methode `print()`.

Die Klasse `Warenlager2` enthält Waren. Natürlich können dort auch verderbliche Waren eingelagert werden, da bei Einhaltung der Verträge nach dem liskovschen Substitutionsprinzip stets ein Objekt einer abgeleiteten Klasse an die Stelle eines Objektes einer Basisklasse treten kann. Allerdings sind dort erweiternde Methoden der Klasse `VerderblichesWarenlager` nicht sichtbar, wohl aber überschreibende Methoden der Klasse `VerderblicheWare`.

Hier die Klasse `Warenlager2`:

```
// Datei: Warenlager2.java
public class Warenlager2
{
   protected Ware2[] arr;

   protected Warenlager2()
   {
   }

   protected Warenlager2 (int max)
   {
      arr = new Ware2 [max];  // Array kann Waren oder verderbliche
                              // Waren aufnehmen
   }

   // Die Methode aufnehmen() kann neue, noch nicht im Lager enthal-
   // tene Waren aufnehmen. Sie kann aber auch zu einer schon im
   // Lager vorhandenen Ware die Anzahl der vorhandenen Elemente
   // erhöhen. Das Array wird beginnend vom Index 0 ab gefüllt.
   // Die freien Array-Elemente enthalten die null-Referenz.

   public int aufnehmen (Ware2 ref, int anzahl)
   {
      // Prüfen, ob die Ware schon vorhanden ist.
      for (int i = 0; i < arr.length; i++)
      {
         if ((arr [i] != null) &&
             (ref.getWarenId() == arr [i].getWarenId()))
         {
            ref.stueckzahlErhoehen (anzahl);
            return 1;
         }
      }

      if (arr [arr.length - 1] != null)
      {
         return -1;           // Warenlager voll!
      }
      else
      {
         for (int i = 0; i < arr.length; i++)
         {
            if (arr [i] == null) // Erstes freies Feld gefunden,
            {                    // Ware noch nicht vorhanden.
               arr [i] = ref;
               arr [i].stueckzahlErhoehen (anzahl);
               break;
```

```
            }
         }
         return 1;
      }
   }

   // Die Methode entnehmen() entnimmt 1 Exemplar einer Ware
   public Ware2 entnehmen (String warenname)
   {
      Ware2 tmp = null;
      boolean gefunden = false;
      for (int i = 0; i < arr.length; i++)
      {
         if ((arr [i] != null) &&
             ((arr[i].getName()).equals (warenname)))
         {
            gefunden = true;

            if (arr[i].getAnzahl() >= 1)
            {
               arr[i].stueckzahlVerringern (1);
               tmp = arr[i];
            }
            else
            {
               System.out.print (
                  "Warenlager2 - entnehmen:");
               System.out.println (
                  "Ware nicht in ausreichender Anzahl am Lager");
            }
         }
      }

      if (gefunden == false)
      {
         System.out.println ("Gesuchte Ware " + warenname +
            " ist nicht im Lager");
      }
      return tmp;
   }

   public void print()
   {
      System.out.println ("Warenlager2 - print():");
      for (int i = 0; i < arr.length && arr [i] != null; i++)
      {
          arr [i].print();
      }
   }
}

// Datei: Test10.java
public class Test10
{
   public static void main (String[] args)
   {
      System.out.println ("Es wird ein verderbliches Warenlager" +
                          " angelegt");
      Warenlager2 lager = new VerderblichesWarenlager (4);
      // lager zeigt auf ein verderbliches Warenlager

      System.out.println ("Es wird das Produkt Milch angelegt");
      VerderblicheWare vRef =
                        new VerderblicheWare ("Milch", .99f, 6.0);
```

```
System.out.println ("Es wird das Produkt Seife angelegt");
Ware2 wareref = new Ware2 ("Seife", .79f);
// Seife ist nicht verderblich
System.out.println("+++++++++++++++++++++++++++++++++++++++++");
System.out.println("+++++++++++++++++++++++++++++++++++++++++");

System.out.println ("Es werden Exemplare von Milch und Seife" +
                    " eingelagert");
System.out.println();

lager.aufnehmen (vRef, 500); // 500 Milch aufgenommen
lager.aufnehmen (wareref, 300); // 300 Seifen aufgenommen
lager.aufnehmen (wareref, 300); // Lagerbestand erhöhen
// 300 Seifen zusätzlich

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

System.out.println("Test10 - Aufruf von lager.print()");
lager.print();
// gibt Attribute von Waren aus. Zeigt auf verderbliches
// Warenlager

VerderblichesWarenlager lager2 =
                              (VerderblichesWarenlager) lager;
// man beachte den Down-Cast. Damit ist das Verderbliche
// Warenlager sichtbar.

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

System.out.println
   ("Test10 - Aufruf von verderblicheWareAusgeben()");
lager2.verderblicheWareAusgeben();
// Aufruf einer speziellen Methode für ein verderbliches
// Warenlager

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

System.out.println ("\nTest10 - Aufruf von lager.print()");
lager.print();

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

System.out.println
   ("\nTest10 - Test der Rückgabewerte von entnehmen():");

System.out.println();
System.out.println ("Jetzt soll 1 Seife entnommen werden");
Ware2 ware = lager.entnehmen ("Seife");
if (ware != null)
{
   System.out.print
      ("Test10 - Die folgende Ware wurde entnommen:");
   ware.print();
}

System.out.println();
System.out.println ("Jetzt soll 1 Milch entnommen werden");
Ware2 ware2 = lager.entnehmen ("Milch");
```

```
      if (ware2 != null)
      {
         System.out.print
            ("Test10 - Die folgende Ware wurde entnommen:");
         ware2.print();
      }

      System.out.println();
      System.out.println ("Jetzt soll 1 Rasierschaum entnommen werden");
      Ware2 ware3 = lager.entnehmen ("Rasierschaum");
      if (ware3 != null)
      {
         System.out.print
            ("Test 10 - Die folgende Ware wurde entnommen:");
         ware3.print();
      }
   }
}
```

Die Ausgabe des Programms ist:

```
Es wird ein verderbliches Warenlager angelegt
Es wird das Produkt Milch angelegt
Es wird das Produkt Seife angelegt
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
Es werden Exemplare von Milch und Seife eingelagert

Test10 - Aufruf von lager.print()
Warenlager2 - print():

print()-Methode der Klasse Ware2
ID: 0  Bezeichnung: Milch  Anzahl: 500
print-Erweiterung der Klasse VerderblicheWare
maximale Lagertemperatur 6.0

print()-Methode der Klasse Ware2
ID: 1  Bezeichnung: Seife  Anzahl: 600
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
Test10 - Aufruf von verderblicheWareAusgeben()
bin in verderblicheWareAusgeben()

print()-Methode der Klasse Ware2
ID: 0  Bezeichnung: Milch  Anzahl: 500
print-Erweiterung der Klasse VerderblicheWare
maximale Lagertemperatur 6.0
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
Test10 - Aufruf von lager.print()
Warenlager2 - print():

print()-Methode der Klasse Ware2
ID: 0  Bezeichnung: Milch  Anzahl: 500
print-Erweiterung der Klasse VerderblicheWare
maximale Lagertemperatur 6.0

print()-Methode der Klasse Ware2
ID: 1  Bezeichnung: Seife  Anzahl: 600
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
Test10 - Test der Rückgabewerte von entnehmen():

Jetzt soll 1 Seife entnommen werden
Ware Seife ist nicht verderblich

Jetzt soll 1 Milch entnommen werden
Test10 - Die folgende Ware wurde entnommen:
print()-Methode der Klasse Ware2
ID: 0  Bezeichnung: Milch  Anzahl: 499
print-Erweiterung der Klasse VerderblicheWare
maximale Lagertemperatur 6.0

Jetzt soll 1 Rasierschaum entnommen werden
Gesuchte Ware Rasierschaum ist nicht im Lager
```

11.10 Zusammenfassung

Kapitel **11.1** stellt das Konzept der Vererbung vor.

Bei der Vererbung erbt eine abgeleitete Klasse die Struktur und das Verhalten ihrer Basisklasse. Eine abgeleitete Klasse kann die Basisklasse erweitern oder Methoden der Basisklasse überschreiben.

Gibt es mehrere Hierarchieebenen der Vererbung, so wird als Superklasse oder Basisklasse bzw. Oberklasse einer Klasse eine an einer beliebigen höheren Stelle des Vererbungspfades stehende Klasse, als Subklasse oder abgeleitete Klasse bzw. Unterklasse einer Klasse eine an einer beliebigen tieferen Stelle des Vererbungspfades liegende Klasse bezeichnet.

Kapitel **11.2** befasst sich mit dem Erweitern und Überschreiben von geerbten Methoden einer Basisklasse. Beim reinen Erweitern kommt in der abgeleiteten Klasse zur Aufrufschnittstelle der Basisklasse, die geerbt wird, noch eine für die abgeleitete Klasse spezifische Aufrufschnittstelle hinzu. Beim Überschreiben von Methoden muss die Signatur der Methoden gleich sein, d. h. der Methodenname muss identisch sein und die formalen Parameter der überschreibenden Methode müssen die gleichen Typen, die gleiche Anzahl und Reihenfolge wie die der überschriebenen Methode haben. Zusätzlich muss auch der Typ des Rückgabewerts übereinstimmen. Überschriebene Instanzmethoden einer Basisklasse können mit `super.methode()` in der überschreibenden Instanzmethode aufgerufen werden. Wird eine Methode mit Hilfe von `super.methode()` in einer Vererbungshierarchie angesprochen, so werden ausgehend von der aktuellen Klasse alle darüber liegenden Klassen des Vererbungsbaums der Reihe nach solange durchsucht, bis zum ersten Mal eine Instanzmethode `methode()` gefunden wird. Überschriebene Klassenmethoden einer bekannten Basisklasse werden einfach durch Angabe des Klassennamens mit `Klassenname.methode()` aufgerufen.

Bei der frühen Bindung kann einem Methodenaufruf schon zum Kompilierzeitpunkt der entsprechende Methodenrumpf zugeordnet werden. Bei der späten Bindung wird dagegen einem Methodenaufruf erst zur Laufzeit der entsprechende Methodenrumpf zugeordnet. In Java hat man keinen direkten Einfluss darauf, ob spät oder früh gebunden wird. Java verwendet in der Regel die späte Bindung, in wenigen spezifizierten Ausnahmefällen jedoch die frühe Bindung.

Kapitel **11.3** behandelt Besonderheiten bei der Vererbung, so zum Beispiel Up- und Down-Cast-Operationen, Konstruktoren bei abgeleiteten Klassen, das Verdecken von Datenfeldern, finale Methoden, finale Klassen und abstrakte Basisklassen.

Ein Sohnobjekt ist sowohl vom Typ der eigenen Klasse als auch vom Typ jeder zugehörigen Basisklasse. Zeigen eine Referenz vom Typ `Vater` und eine Referenz vom Typ `Sohn` auf ein Objekt der Klasse `Sohn`, so sieht die Referenz vom Typ `Sohn` das gesamte Objekt und die Referenz vom Typ `Vater` sieht nur die Vateranteile des Sohnobjektes. Die implizite Typkonvertierung in eine Superklasse bezeichnet man auch als Up-Cast.

Eine explizite Typkonvertierung von Referenzen mit Hilfe des cast-Operators muss immer dann erfolgen, wenn bei einer Zuweisung eine Referenzvariable vom Typ

`Vater`, die auf ein Objekt der Klasse `Sohn` zeigt, einer Referenzvariablen vom Typ `Sohn` zugewiesen wird. Eine solche explizite Typkonvertierung mit Hilfe des cast-Operators bezeichnet man auch als Down-Cast. Bei einer Zuweisung `refSohn = (Sohn) refVater` ist die explizite cast-Operation nur dann zulässig, wenn die Referenz `refVater`

- auf ein Objekt vom Typ `Sohn`
- oder auf ein Objekt eines Subtyps der Klasse `Sohn`

zeigt.

Es ist möglich, innerhalb eines Konstruktors einer abgeleiteten Klasse einen Konstruktor der Vaterklasse aufzurufen. Dies wird durch den Aufruf von `super()` gemacht. Der Konstruktor der Vaterklasse initialisiert die von der Vaterklasse geerbten Anteile eines Objekts einer abgeleiteten Klasse. Diese Initialisierung kann natürlich durch den Konstruktor der Sohnklasse wieder überschrieben werden.

Der Default-Konstruktor der Klasse `Object` hat einen leeren Rumpf, d. h. er tut nichts. Ein Default-Konstruktor einer anderen Klasse ruft automatisch den parameterlosen Konstruktor ihrer Vaterklasse auf. Ein Konstruktor der Basisklasse wird immer vor den Initialisierungsanweisungen des Konstruktors der abgeleiteten Klasse ausgeführt.

Ein parameterloser Konstruktor hat keine Parameter. Der Programmierer kann selbst einen Konstruktor ohne Parameter schreiben. Dann wird vom Compiler dieser selbstgeschriebene parameterlose Konstruktor und nicht der vom Compiler zur Verfügung gestellte parameterlose Konstruktor, der Default-Konstruktor, aufgerufen.

Ein formaler Parameter des Konstruktors der abgeleiteten Klasse kann als aktueller Parameter an den Konstruktor der Basisklasse übergeben werden.

In den Konstruktor einer abgeleiteten Klasse wird automatisch vom Compiler der Aufruf des parameterlosen Konstruktors der Vaterklasse durch `super()` eingefügt, es sei denn der Programmierer führt einen `super()`-Aufruf explizit selbst durch.

Ein Programmierer sollte stets auch einen parameterlosen Konstruktor für eine Klasse schreiben, wenn er einen Konstruktor mit Parametern schreibt, weil durch das Schreiben des Konstruktors mit Parametern der Default-Konstruktor nicht mehr zur Verfügung steht.

In einem Konstruktor kann entweder ein anderer überladener Konstruktor derselben Klasse aufgerufen werden oder ein Konstruktor der Vaterklasse.

Allgemein beschrieben sieht der Ablauf beim Aufruf eines Konstruktors einer abgeleiteten Klasse folgendermaßen aus:

1. Beginnt der Konstruktor der abgeleiteten Klasse in der ersten Zeile mit einem Aufruf eines anderen Konstruktors derselben Klasse (mittels `this()`), so wird der entsprechende Konstruktor aufgerufen.

2. Beginnt der Konstruktor der abgeleiteten Klasse in der ersten Zeile nicht mit einem Aufruf eines Konstruktors derselben Klasse (mittels `this()`), so folgt ein impliziter oder expliziter Aufruf eines Konstruktors der Vaterklasse mittels `super()`.
3. Nachdem der Konstruktor der Vaterklasse vollständig abgearbeitet wurde, werden die entsprechenden Instanzvariablen der abgeleiteten Klasse manuell initialisiert.
4. Der nicht statische Initialisierungsblock der Klasse wird abgearbeitet.
5. Die eigentlichen Anweisungen des Konstruktorblocks der abgeleiteten Klasse werden ausgewertet.

Zu beachten ist, dass die Instanzvariablen durch die automatische Initialisierung bereits mit Default-Werten belegt sind, bevor der Block des Konstruktors betreten wird.

Vom Verdecken eines Datenfeldes spricht man, wenn in der abgeleiteten Klasse ein Datenfeld angelegt wird, das den gleichen Namen trägt wie ein von der Basisklasse geerbtes Datenfeld. Das Verdecken von Datenfeldern erfolgt grundsätzlich bei Namensgleichheit. Deshalb wird ein `float`-Datenfeld `x` des Sohnes, das von der Vaterklasse geerbt wurde, auch durch ein `int`-Datenfeld `x` der Sohnklasse verdeckt. Für den Zugriff auf das eigene Datenfeld der Sohnklasse hat das Sohnobjekt zwei Möglichkeiten:

- einfach über den Namen der Variablen: `x`
- mit Hilfe der `this`-Referenz: `this.x`

Für den Zugriff auf ein von der Vaterklasse ererbtes und verdecktes Datenfeld bestehen die beiden folgenden Möglichkeiten:

- über das Schlüsselwort `super` mit `super.x`.
- über einen Cast der `this`-Referenz in das Vaterobjekt: `((Vater) this).x`.

Wie bekannt, ist die `this`-Referenz eine Referenz auf das eigene Objekt und kann genauso wie andere Referenzen auf den Typ der Vaterklasse gecastet werden. Wird der Typ einer Referenz auf ein Objekt auf die Klasse des Vaters gecastet, sind nur noch die Datenfelder des Vaters sichtbar und nicht mehr die des Sohnes.

Wird ein Datenfeld `x` mit Hilfe von `super` gesucht, so wird ausgehend von der aktuellen Klasse die gesamte Klassenhierarchie aufwärts der Reihe nach solange durchsucht, bis zum ersten Mal ein Datenfeld `x` gefunden wird. An dieser Stelle wird die Suche abgebrochen.

Finale Methoden können in einer Subklasse nicht überschrieben werden. Finale Klassen sind Klassen, von denen man keine weiteren Klassen ableiten kann. Damit kann man nur die Benutzung von Klassen, aber nicht die Ableitung erlauben.

Eine abstrakte Basisklasse kann nicht instanziiert werden.

Eine Klasse, in der alle Methoden implementiert sind, kann mit dem Schlüsselwort `abstract` zur einer abstrakten Klasse gemacht werden. Damit kann die Instanziierung einer solchen Klasse verhindert werden. Eine solche Klasse ist zwar vollständig, sie stellt aber eine Abstraktion dar, von der es in der Realität keine Objekte

gibt. Eine solche abstrakte Basisklasse dient allein dem Ziel der Generalisierung in der Klassenhierarchie.

Wird in einer Basisklasse nur die Schnittstelle von Methoden festgelegt und die eigentliche Implementierung einer, mehrerer oder aller Methoden erst in den abgeleiteten Klassen vorgenommen, so liegt ebenfalls eine abstrakte Basisklasse vor. Methoden, für die nur die Schnittstelle festgelegt werden soll, die in einer Klasse also keinen Methodenrumpf besitzen, müssen mit dem Schlüsselwort `abstract` deklariert werden. Ist auch nur eine einzige abstrakte Methode in einer Klasse enthalten, so ist diese Klasse zwangsläufig abstrakt und ist damit mit dem Schlüsselwort `abstract` zu deklarieren. Abstrakte Klassen können nicht instanziiert werden

Kapitel **11.4** behandelt Polymorphie von Objekten in Klassenhierarchien und das liskovsche Substitutionsprinzip.

Eine Polymorphie von Objekten gibt es nur bei Vererbungshierarchien. Polymorphie von Objekten setzt voraus, dass ein Programm geschrieben wird, das Referenzen auf Objekte einer Basisklasse enthält. Dieser Code kann dann später von Objekten beliebiger abgeleiteter Klassen benutzt werden. Polymorphie ist der Grund für die wunderbare Möglichkeit, dass sich ganze Klassenbibliotheken problemlos wiederverwenden lassen. Dabei ist natürlich zum Zeitpunkt des Schreibens des Codes der Basisklasse überhaupt nicht bekannt, welche Klassen zu späteren Zeitpunkten von der Basisklasse abgeleitet werden.

Da ein Objekt einer abgeleiteten Klasse, welche eine Basisklasse nur erweitert, die Aufrufschnittstelle der Basisklasse und dieselben Datenfelder wie die Basisklasse hat, kann ein Objekt einer abgeleiteten Klasse sich ohne jegliches Problem wie ein Objekt einer Basisklasse verhalten. Eine zusätzliche Aufrufschnittstelle aufgrund weiterer Methoden der abgeleiteten Klasse kommt nicht zum Tragen, da diese Methoden im Quellprogramm, das nur die Basisklasse kennt, überhaupt nicht angesprochen werden. Die zusätzliche Aufrufschnittstelle und die zusätzlichen Datenfelder einer abgeleiteten Klasse bleiben also unsichtbar.

Findet ein Überschreiben einer Operation in einer abgeleiteten Klasse statt, so muss darauf geachtet werden, dass in der abgeleiteten Klasse die Verträge der Basisklasse eingehalten werden.

Das liskovsche Substitutionsprinzip kann im Falle einer reinen Erweiterung folgendermaßen formuliert werden: Im Falle der Erweiterung kann ein Objekt einer abgeleiteten Klasse problemlos an die Stelle eines Objektes einer Basisklasse treten.

Das liskovsche Substitutionsprinzip im Falle des Überschreibens lautet: Im Falle des Überschreibens muss der Programmierer selbst dafür sorgen, dass ein Objekt einer abgeleiteten Klasse an die Stelle eines Objektes einer Basisklasse treten darf. Er muss hierfür beim Überschreiben die Einhaltung der Verträge der Basisklasse gewährleisten.

Verträge, deren Einhaltung bei der Vererbung für das liskovsche Substitutionsprinzip Voraussetzung sind, werden in Kapitel **11.5** betrachtet.

Worum es bei dem Prinzip Design by Contract von Bertrand Meyer vor allem geht, ist, dass sich beim Aufruf einer Methode der Aufrufer und die aufgerufene Methode gegenseitig aufeinander verlassen können. Solange bei der Ableitung von einer Basisklasse der Vertrag der Basisklasse in einer Unterklasse nicht gebrochen wird, ist es möglich, den für die Basisklasse geschriebenen Code auch für die Unterklassen, die eventuell erst später erfunden werden, zu verwenden.

Entwurf durch Verträge verwendet drei verschiedene Arten von Zusicherungen:

- Vorbedingungen von Methoden,
- Nachbedingungen von Methoden
- und Invarianten von Klassen.

Eine Vorbedingung stellt eine Pflicht für einen Aufrufer dar, sei es, dass der Aufruf innerhalb der eigenen Klasse erfolgt oder von einem Kunden. Die Vorbedingung definiert die Bedingungen, unter denen ein Aufruf zulässig ist. Sie stellt eine Pflicht für den Aufrufer dar und einen Nutzen für den Aufgerufenen.

Eine Nachbedingung bindet eine Methode einer Klasse. Eine Nachbedingung stellt die Bedingung dar, die von einer Methode eingehalten werden muss. Die Nachbedingung ist eine Pflicht für den Aufgerufenen und ein Nutzen für den Aufrufer. Der Aufgerufene hat also die Pflicht, korrekte Werte zurückzugeben. Diese Pflicht des Aufgerufenen ist der Vorteil des Aufrufers, da er korrekte Werte erhält.

Da die Invarianten von allen Methoden einer Klasse, die von einem Kunden aufgerufen werden können, eingehalten werden müssen, um die Korrektheit zu gewährleisten, spricht man auch von Klasseninvarianten. Eine Invariante muss gelten vor Aufruf einer Methode und nach dem Aufruf einer Methode durch einen Kunden. Eine Invariante kann temporär verletzt werden während der Ausführung einer Methode oder beim Aufruf von Service-Methoden, die nicht außerhalb der Klasse sichtbar sind – also nicht exportiert werden. Dies stellt kein Problem dar, da die Invariante dem Kunden erst nach Ausführung einer exportierten Methode wieder zur Verfügung steht. Eine Klasseninvariante muss also vor und nach dem Aufruf einer nach außen sichtbaren Methode (Export-Methode) eingehalten sein.

Der Vertrag einer Klasse umfasst die Verträge der Methoden und die Invarianten der Klasse. Werden verschiedenen Kunden einer Klasse jedoch verschiedene Leistungen der Klasse zur Verfügung gestellt, so ordnet man die Verträge der Methoden in verschiedene Verträge der Klasse jeweils mit dem entsprechenden Kunden ein.

Überschreibende Methoden dürfen die Vorbedingung einer überschriebenen Methode nur aufweichen und nicht verschärfen, da sonst ein Aufrufer damit nicht fertig werden würde. Überschreibende Methoden dürfen Nachbedingungen nur verschärfen, da mit aufgeweichten Nachbedingungen ein Aufrufer nicht leben könnte.

Die Invarianten einer abgeleiteten Klasse ergeben sich aus der booleschen UND-Verknüpfung ihrer Invarianten und der Invarianten ihrer Basisklasse.

In Kapitel **11.6** wird die Identifikation der Klasse eines Objekts erklärt.

Mit dem `instanceof`-Operator kann getestet werden, ob eine Referenz auf ein Objekt eines bestimmten Typs zeigt. Dies ist beispielsweise dann wichtig, wenn eine

Referenz vom Typ einer Basisklasse ist. Eine solche Referenz kann ja auf Objekte aller abgeleiteten Klassen zeigen. Mit Hilfe des `instanceof`-Operators lässt sich nun nachprüfen, ob das referenzierte Objekt tatsächlich vom angenommenen Typ ist.

Run Time Type Identification (RTTI) ist die Erkennung des Typs eines Objektes zur Laufzeit.

Die erste Information, die zu einem Objekt im Heap abgelegt wird, ist ein Zeiger auf die in der Method-Area liegende Klasse des Objektes. Erst dann folgen die Instanzvariablen.

Damit es möglich wird, jedes Mal die richtige Methode aufzurufen, benötigt jede Klasse in der Method Area noch zusätzlich eine Methodentabelle. In der Methodentabelle befinden sich nur Zeiger auf die Methoden, die für die dynamische Bindung in Frage kommen. Deshalb haben private, statische oder finale Methoden keinen Eintrag in der Methodentabelle. Damit ist sichergestellt, dass ggfs. eine überschreibende Methode aufgerufen wird. Beim Casten verändert sich nur die Sichtweise auf die zur Verfügung stehenden Methodenköpfe. Das Objekt, auf das eine Referenz zeigt, und die zugehörige Methodentabelle bleiben beim Casten unverändert. Allerdings werden beim Cast auf eine Superklasse die erweiternden Methoden der abgeleiteten Klassen unsichtbar.

Kapitel **11.7** fasst Operatoren für Referenztypen in einer Übersicht zusammen. Nach den Betrachtungen zur Konsistenzhaltung von Quell- und Bytecode in Kapitel **11.8** werden in Kapitel **11.9** die Eigenschaften der Kovarianz und Kontravarianz beim Überschreiben von Methoden untersucht.

11.11 Übungen

Aufgabe 11.1: Konzept der Vererbung

11.1.1 Vererbungshierarchie für Fahrzeuge

Die Klassen `Pkw` und `Motorrad` sollen von der Klasse `Fahrzeug` abgeleitet werden. In der Klasse `FahrzeugTest` sollen die Klassen `Pkw` und `Motorrad` getestet werden. Das folgende Java-Programm enthält die Klassen `Fahrzeug`, `Pkw`, `Motorrad` und `FahrzeugTest`. Die fehlenden und zu ergänzenden Teile des Programms sind durch . . . . . gekennzeichnet. Lesen Sie zuerst die Fragen nach dem Programm, bevor Sie das Programm vervollständigen!

```
// Datei: Fahrzeug.java
import java.util.*;

class Fahrzeug
{
   private float preis;
   private String herstellerName;
   protected static Scanner eingabe = new Scanner (System.in);

   public Fahrzeug()
   {
```

```
      System.out.println();
      System.out.print ("Geben Sie den "
                        + "Herstellernamen ein: ");
      herstellerName = eingabe.next();

      System.out.print ("Geben Sie den Preis ein: ");
      try
      {
         preis = eingabe.nextFloat();
      }
      catch (InputMismatchException e)
      {
         System.out.println ("Keine gültige Preisangabe!");
         System.exit (1);
      }
   }

   public void print()
   {
      System.out.println();
      System.out.println ("Herstellername: "
                          + herstellerName);
      System.out.println ("Preis         : "
                          + preis);
   }

   // Methode getPreis();
   . . . . .
}

// Datei: Pkw.java
class Pkw extends Fahrzeug
{
   private String fahrzeugtyp = "Pkw";
   private String modellBezeichnung;

   public Pkw()
   {
      . . . . .// Aufruf des Konstruktors
               // der Basisklasse

      System.out.print ("Geben Sie die "
                        + "Modellbezeichnung ein: ");
      modellBezeichnung = eingabe.next();
   }

   public void print()
   {
      . . . . . .
   }
}

// Datei: Motorrad.java
class Motorrad extends Fahrzeug
{
   private String fahrzeugtyp = "Motorrad";
```

```
   public void print()
   {
      . . . . . .
   }
}

// Datei: FahrzeugTest.java
public class FahrzeugTest
{
   public static void main (String args[])
   {
      System.out.println ("Start des Programms");

      // Anlegen eines Arrays aus 6 Fahrzeugen
      . . . . .

      // Die ersten 3 Elemente des Arrays sollen mit Pkws
      // gefüllt werden
      System.out.println();
      System.out.println ("3 Pkws");

      . . . . .

      // Die drei letzten Elemente mit Motorrädern füllen
      System.out.println();
      System.out.println ("3 Motorräder");

      . . . . .

      // Geben Sie in einer Schleife für alle Array-Elemente
      // die entsprechenden Datenfelder aus

      . . . . .

      // Ermittlung des Gesamtwerts aller Fahrzeuge

      . . . . .

      System.out.println();
      System.out.println ("Gesamtwert aller Fahrzeuge: "
                          + summe);
   }
}
```

a) Schreiben Sie die Methode `getPreis()` der Klasse `Fahrzeug`.

b) Vervollständigen Sie den Konstruktor der Klasse `Pkw`.

c) Überschreiben Sie in der Klasse `Pkw` die Methode `print()` der Klasse `Fahrzeug`. Die Methode `print()` der Klasse `Pkw` soll alle Datenfelder eines Objektes der Klasse `Pkw` unter Zuhilfenahme der Methode `print()` der Basisklasse ausgeben. Ergänzen Sie die Methode `print()` der Klasse `Pkw`. Ergänzen Sie in analoger Weise die Methode `print()` der Klasse `Motorrad`.

d) Ergänzen Sie die fehlenden Teile der Klasse `FahrzeugTest`.

11.1.2 Vererbungshierarchie für Fertigungsgüter

Ein produzierender Betrieb verwaltet seine hergestellten Produkte zurzeit mit folgenden drei Klassen:

```
public class Membranpumpe
{
   private String name;
   private int tiefe;
   private float maximalerBetriebsdruck;
   private int hoehe;
   private String membranmaterial;
   private int gewicht;
   private int maximaleFoerdermenge;
   private int breite;
}

public class Kreiselpumpe
{
   private int breite;
   private int hoehe;
   private int gewicht;
   private int anzahlSchaufeln;
   private int maximaleFoerdermenge;
   private int maximaleDrehzahl;
   private String name;
   private int tiefe;
   private float maximalerBetriebsdruck;
}

public class Auffangbecken
{
   private int tiefe;
   private int volumen;
   private int breite;
   private int gewicht;
   private String name;
   private int hoehe;
}
```

Entwickeln Sie eine passende Vererbungshierarchie, welche die gemeinsamen Attribute in Basisklassen zusammenfasst.

Aufgabe 11.2: Konstruktoren bei abgeleiteten Klassen

11.2.1 Reihenfolge von Konstruktoraufrufen

Die folgende Klasse `TestKonstruktoren` dient zum Testen der Aufrufreihenfolge von Konstruktoren bei abgeleiteten Klassen:

```
// Datei: TestKonstruktoren
public class TestKonstruktoren
{
   public static void main (String[] args)
   {
```

```
        System.out.println ("Exemplar von A wird angelegt");
        A aRef = new A();

        System.out.println();
        System.out.println ("Exemplar von B wird angelegt");
        B bRef = new B();

        System.out.println();
        System.out.println ("Exemplar von C wird angelegt");
        C cRef = new C();
        System.out.println();
    }
}
```

Schreiben Sie die drei Klassen `A`, `B` und `C`, welche jeweils nur einen Konstruktor ohne Parameter enthalten. Im Konstruktor der Klasse `A` soll folgender Text ausgegeben werden:

```
System.out.println ("Klasse A - Konstruktor ohne Parameter");
```

Schreiben Sie entsprechende Konstruktoren für die Klassen `B` und `C`. Beachten Sie, dass `B` von `A` und `C` von `B` abgeleitet ist. Der Konstruktor der Klasse `C` soll den parameterlosen Konstruktor der Basisklasse `B` durch `super()` explizit aufrufen. Beim Ausführen der Klasse `TestKonstruktoren` wird deutlich, dass es in diesem Fall keinen Unterschied macht, ob der Aufruf `super()` explizit durch den Programmierer eingefügt wird oder implizit durch den Compiler. Das folgende Bild gibt die Klassenhierarchie wieder:

11.2.2 Vererbungshierarchie für Fertigungsgüter, Teil 2

Ausgangspunkt ist das Programm aus Aufgabe 11.1.2. Erweitern Sie alle Klassen der Vererbungshierarchie mit Konstruktoren, um eine einfache Initialisierung der Klassen zu ermöglichen.

Aufgabe 11.3: Abstrakte Basisklasse

11.3.1 Flächen- und Umfangsberechnung

In dieser Übung sollen die beiden Klassen `Kreis` und `Quadrat` implementiert werden. Hierzu leiten beide Klassen von der abstrakten Basisklasse `Geometrische-Figur` ab und werden mit Hilfe der Klasse `TestBerechnung` getestet. Die beiden

Klassen haben die Aufgabe, die Fläche und den Umfang eines Kreises bzw. Quadrats zu berechnen.

Die folgende Tabelle charakterisiert die Eigenschaften von Kreis und Quadrat:

	Umfang	Fläche
Kreis	$2 \cdot \pi \cdot r$	$\pi \cdot r^2$
Quadrat	$4 \cdot a$	a^2

Eine Konstante für die Zahl π ist in der Klasse `java.lang.Math` definiert. Hier die Basisklasse `GeometrischeFigur`:

```
// Datei: GeometrischeFigur.java
public abstract class GeometrischeFigur
{
   protected abstract double berechneFlaeche();
   protected abstract double berechneUmfang();

   public void print()
   {
      System.out.println ("Die Fläche beträgt: " +
                          berechneFlaeche());
      System.out.println ("Der Umfang beträgt: " +
                          berechneUmfang());
      System.out.println();
   }
}
```

Und nun das Testprogramm:

```
// Datei: TestBerechnung.java
public class TestBerechnung
{
   public static void main (String[] args)
   {
      Kreis kreisRef = new Kreis (5);
      Quadrat quadratRef = new Quadrat (10);
      kreisRef.print();
      quadratRef.print();
   }
}
```

11.3.2 Hierarchie mehrerer abstrakter Klassen

Ein Getränkehändler verkauft eine Vielzahl unterschiedlicher Getränke. Es findet eine Unterscheidung zwischen alkoholfreien und alkoholhaltigen Getränken statt.

Dies zeigt die folgende Klassenhierarchie:

Die Klasse `Getraenk` hat die Instanzvariablen `hersteller`, `inhaltsmenge` und `preis`. Die Klasse `AlkoholischesGetraenk` hat zusätzlich die Instanzvariable `alkoholgehalt`.

Implementieren Sie die abstrakten Basisklassen `Getraenk`, `AlkoholischesGetraenk` und `AlkoholfreiesGetraenk` entsprechend der gezeigten Klassenhierarchie. Ergänzend sollen die Klassen `Bier`, `Cola`, `Mineralwasser` und `Wein` – die von den Klassen `AlkoholischesGetraenk` oder `AlkoholfreiesGetraenk` abgeleitet werden – mit eigenen Instanzvariablen erstellt werden. Alle Instanzvariablen sollen in den Konstruktoren der jeweiligen Klassen gesetzt werden, inklusive der geerbten Variablen aus den abstrakten Basisklassen. Mit der Klasse `GetraenkeTest` sollen Objekte der verschiedenen Getränke instanziiert werden.

Aufgabe 11.4: Polymorphie

11.4.1 Vererbungshierarchie für Fertigungsgüter, Teil 3

Ausgangspunkt ist das Programm aus Aufgabe 11.2.2. Erweitern Sie alle Klassen der Vererbungshierarchie um eine Methode `print()`, um den Inhalt einer Klasse auf dem Bildschirm auszugeben. Schreiben Sie eine Testklasse, die mehrere Produkte anlegt und deren Inhalt auf dem Bildschirm ausgibt.

11.4.2 Arzneimittel mit verschiedenen Maßeinheiten

Analysieren Sie das folgende Programm. Was erwarten Sie als Ausgabe?

```
public class Arzneimittel
{
   private String name;
   public Arzneimittel (String name)
   {
      this.name = name;
   }

   public void print()
   {
      System.out.println ("Das Arzneimittel heißt: " + name);
   }
}

public class Tablette extends Arzneimittel
{
   private int anzahl; // Anzahl der Tabletten

   public Tablette (String name, int anzahl)
   {

      super (name);
      this.anzahl = anzahl;
   }

   public void print()
   {
      super.print();
      System.out.println ("Es enthält "+anzahl+" Tabletten.");
```

```
   }
}

public class Salbe extends Arzneimittel
{
   private int menge;  // Menge der Salbe in ml

   public Salbe (String name, int menge)
   {
      super (name);
      this.menge = menge;
   }

   public void print()
   {
      super.print();
      System.out.println ("Es enthält " + menge+" ml Salbe.");
   }

   public int getMenge()
   {
      return this.menge;
   }
}

public class Augensalbe extends Salbe
{
   public Augensalbe (String name, int menge)
   {
      super (name, menge);
   }

   public void print()
   {
      System.out.println ("Die Augensalbe enthält "
                          + super.getMenge() + " ml.");
   }
}

public class Arzneimittellager
{

   public static void main (String[] args)
   {
      Arzneimittel mittel = Arzneimittel ("Schmerzmittel");
      Tablette tablette =
         new Tablette ("Schmerztablette", 12);
      Salbe salbe = new Salbe ("Wundsalbe", 200);
      Augensalbe augensalbe =
         new Augensalbe ("Augensalbe", 50);

      mittel.print();
      tablette.print();
      salbe.print();
      augensalbe.print();
   }
}
```

Aufgabe 11.5: Flughafen-Projekt – Einführung von Entity-Klassen

In der vorgehenden Projektaufgabe 10.4 wurden sämtliche Informationen in einer einzigen Klasse gehalten. Die Informationen wurden dabei sehr einfach mit Werten vom Typ `int` und vom Typ `String` gehalten. Dies ist im Falle des Flughafensystems natürlich nicht ausreichend. So besitzt zum Beispiel eine Fluggesellschaft nicht nur einen Namen, sondern auch eine Straße und einen Ort. Deshalb sollen in dieser Projektaufgabe neue Klassen – sogenannte Entity-Klassen – eingeführt werden. Ein Großteil der Arbeit wurde bereits durch das Finden dieser Klassen in der Systemanalyse in Kapitel 2.6.2 erledigt. Ein Teil dieser Entity-Klassen soll jetzt implementiert werden. Das folgende Klassendiagramm soll erst mal einen Überblick über eine mögliche Lösung geben.

Die Klassen `Fluggesellschaft` und `Flugzeugtyp` sollten jeweils einen Konstruktor besitzen, der es ermöglicht, die einzelnen Strings zu setzen. Die Klasse `Bahn` soll jeder erzeugten Instanz eine eindeutige Nummer vergeben. Der Aufzählungstyp `Status` definiert die unterschiedlichen Zustände, die ein Flugzeug annehmen kann. Hier zunächst ein Vorschlag für das Klassendiagramm:

Die Klasse `Parkstelle` stellt eine abstrakte Klasse dar, die keine Methoden und auch keine Instanzvariablen hält. Sie dient lediglich als gemeinsame Basisklasse für die drei Klassen `Parkposition`, `Werft` und `SeparateParkposition`. Jede Instanz der Klasse `Parkposition` soll eine eindeutige Nummer erhalten.

Für das Halten der Ist- und Sollzeiten soll die Klasse `java.util.GregorianCalendar` verwendet werden. Diese Klasse wurde bereits in Kapitel 11.4.2.2 verwen-

det. Zum Einlesen einer Uhrzeit von der Tastatur können Sie die Klasse `Abfrage` um folgende Methode erweitern:

```
public static java.util.Calendar abfrageUhrzeit (String frage)
{
   java.text.SimpleDateFormat formatter;
   formatter = new java.text.SimpleDateFormat ("HH:mm");
   try
   {
      java.util.Date date;
      date = formatter.parse (abfrageString (frage + " (HH:mm):"));
      java.util.Calendar calendar =
                                 new java.util.GregorianCalendar();
      calendar.setTime (date);
      return calendar;
   }
   catch (java.text.ParseException e)
   {
      System.out.println ("Bitte eine gültige Uhrzeit eingeben!");
      return abfrageUhrzeit (frage);
   }
}
```

Kapitel 12

Pakete

12.1 "Programmierung im Großen"
12.2 Pakete zur Gruppierung
12.3 Erstellung von Paketen
12.4 Benutzung von Paketen
12.5 Paketnamen
12.6 Gültigkeitsbereich von Klassennamen
12.7 Zugriffsmodifikatoren
12.8 Zusammenfassung
12.9 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_12

12 Pakete

Ehe auf die Gruppierung von Quellcode in Kapitel 12.2 eingegangen wird, stellt Kapitel 12.1 das Konzept der "Programmierung im Großen" vor. Kapitel 12.3 befasst sich mit der Erstellung von Paketen und Kapitel 12.4 mit der Benutzung von Paketen. Kapitel 12.5 erörtert Paketnamen und Kapitel 12.6 den Gültigkeitsbereich von Klassennamen. Kapitel 12.7 diskutiert die Bedeutung von Zugriffsmodifikatoren für Klassen bzw. Schnittstellen in Paketen sowie von Elementen in Klassen, d. h. von Methoden und Datenfeldern, für den Zugriff aus anderen Paketen.

12.1 "Programmierung im Großen"

Eine moderne Programmiersprache soll das Design (den Entwurf) eines Programms unterstützen. Hierzu sind Sprachmittel erforderlich, die es erlauben, ein Programm übersichtlich zu strukturieren. Man spricht bei solchen Sprachmitteln auch vom **"Programmieren im Großen"**.

In den prozeduralen Programmiersprachen stellen das Hauptprogramm und die dazugehörigen Unterprogramme in Form von Funktionen die einzigen Strukturierungsmöglichkeiten für ein Programm dar.

Ein Java-Programm kann mit Hilfe der folgenden Sprachmittel von Java grob strukturiert werden:

- Klasse bzw. Thread (ein Thread ist im Grunde eine spezielle Klasse),
- Schnittstelle (Interface)
- und Paket.

Im Folgenden werden diese Sprachmittel kurz beschrieben:

- Eine **Klasse** implementiert einen abstrakten Datentyp und definiert die Methoden, die für diesen abstrakten Datentyp zur Verfügung gestellt werden. Ferner implementiert sie die Daten der Klasse.

  Threads werden eingesetzt, um eine parallele (Mehrprozessorsystem) bzw. quasiparallele[166] (Einprozessorsystem) Bearbeitung zur Laufzeit zu ermöglichen. Zu beachten ist, dass Threads mit Hilfe von Klassen definiert werden (siehe Kapitel 16).

  Ein **Thread** definiert einen Bearbeitungsablauf oder Handlungsfaden, der nebenläufig zu anderen Threads durchgeführt werden kann.

[166] **Quasiparallel** bedeutet im Falle eines einzigen Prozessors, dass es für den Anwender nur so aussieht, als ob die Threads parallel laufen würden. Tatsächlich erhalten die verschiedenen Threads jeweils nur eine gewisse Menge Rechenzeit auf dem Prozessor. Ist die Wechselzeit für die Threads kurz wie z. B. 100 ms, so merkt ein Anwender nichts von dem Wechsel.

Mehrere **Threads** können quasiparallel auf ein und demselben Prozessor ablaufen oder parallel auf verschiedenen Prozessoren.

- Eine Schnittstelle spezifiziert eine Aufrufschnittstelle. Eine **Schnittstelle** ist in Java eine **Zusammenstellung von Methodenköpfen** und **eventuell** von **Konstanten**. Implementiert eine Klasse eine Schnittstelle, so stellt sie eine konkrete Implementierung der Methodenköpfe dieser Schnittstelle bereit.
- Ein **Paket** ist ein Mittel zur Gruppierung von Quellcode, trägt einen Namen und kann Klassen, Schnittstellen und Unterpakete als Komponenten, d. h. Bestandteile des Pakets, enthalten. Ein Paket kann aus einer oder aus mehreren Quellcode-Dateien bestehen. Dabei wird in jeder Quellcode-Datei deklariert, dass sie zu dem betreffenden Paket gehört.

Die **physisch greifbaren Bestandteile** eines Programms in Form von Quellcode sind die Dateien. Dateien, die Quellcode enthalten, sind **kompilierfähige Einheiten**. Sie können an den Compiler übergeben werden.

Kompilierfähige Einheiten werden oft auch als **Module** bezeichnet.

Eine Datei kann in Java Klassen bzw. Threads und Schnittstellen als Sprachmittel enthalten, sowie die Deklaration von Paketzugehörigkeiten.

Ein Paket kann physisch mehrere Dateien umfassen. Als Sprachmittel gehören zu einem Paket Klassen bzw. Threads, Schnittstellen und Unterpakete.

Das folgende Bild symbolisiert die Daten eines Pakets:

Bild 12-1 Bestandteile eines Java-Programms

Ein Paket dient zur **Gruppierung**. Der Zugriff auf die Komponenten eines Pakets erfolgt über den Paketnamen. Ein Paket ist auch ein Mittel zur **Strukturierung der Sichtbarkeit** von Klassen und Schnittstellen.

Klassen, Threads und Schnittstellen, die im selben Paket liegen, haben wechselseitig mehr Zugriffsrechte – wenn keine Zugriffsmodifikatoren angegeben sind – als ein außenstehender Benutzer, der die Komponenten des Pakets benutzen will.

Ein Nutzer eines Pakets kann prinzipiell nur diejenigen Teile eines Pakets nutzen, die der Ersteller des Pakets explizit zur externen Benutzung frei gegeben hat. Dies muss der Ersteller mit Hilfe des Schlüsselwortes `public` zum Ausdruck bringen.

Ein Paket stellt nicht nur ein Mittel zur Gruppierung und eine **Strukturierungseinheit für die Sichtbarkeit** dar, sondern auch einen eigenen **Namensraum**. Dies bedeutet, dass ein und derselbe Name einer Komponente eines Pakets auch in einem anderen Paket vorkommen darf. Nur innerhalb desselben Pakets darf der Name nicht ein zweites Mal auftreten.

Der Einsatz von Paketen bietet die folgenden Vorteile:

- Pakete bilden eigene Bereiche für den **Zugriffsschutz**. Mit Paketen kann man kapseln (Information Hiding).
- Jedes Paket bildet einen eigenen **Namensraum**. Damit können Namenskonflikte vermieden werden und identische Namen für Klassen bzw. Schnittstellen in verschiedenen Paketen vergeben werden.
- Pakete sind größere Einheiten für die **Strukturierung** von objektorientierten Systemen als die Klassen und dienen der **Gruppierung**.

Zwei Klassen bzw. zwei Schnittstellen mit identischem Namen können zwar nicht in einem gemeinsamen Paket liegen, aber sehr wohl in zwei unterschiedlichen Paketen. Hierzu stelle man sich eine Klasse `Printer` vor. Einmal kann diese Klasse in einer Ausprägung zum Drucken von Grafiken im Paket `grafiken` vorhanden sein, ein zweites Mal kann eine andere Klasse `Printer` zum Ausdrucken von Dokumenten dem Paket `dokumente` angehören.

Dabei ist zu beachten, dass `Printer` den sogenannten **einfachen Namen** (**simple name**) der beiden Klassen darstellt und nicht eindeutig ist. Zur Unterscheidung der beiden Klassen kann der **voll qualifizierte Name** (**fully qualified name**) benutzt werden, der sich aus dem jeweiligen Paketnamen und dem einfachen Klassennamen zusammensetzt. Im Beispiel des vorangegangenen Satzes sind die voll qualifizierten Namen demnach `grafiken.Printer` und `dokumente.Printer`.

12.2 Pakete zur Gruppierung

Pakete dienen dazu, die Software eines Projektes oder Systems in größere, inhaltlich zusammengehörige Bereiche einzuteilen. Dabei kann ein Paket mehrere Dateien und Unterpakete umfassen, wobei eine Datei genau zu einem Paket gehört.

Pakete stellen die **gröbsten Strukturierungseinheiten** der objektorientierten Technik dar. Sie werden im Rahmen des Entwurfs der Software als Strukturierungseinheit konzipiert.

Da Pakete eigene Bibliotheken darstellen und es der Übersichtlichkeit und Testbarkeit abträglich ist, wenn die Software eines Pakets die Software eines jeden anderen Pakets benutzen darf, versucht man in konkreten Projekten, eine gewisse Ordnung in die Beziehungen zwischen den Paketen zu bringen. Hierbei werden oft Schichtenmodelle derart aufgestellt, dass die in einem Paket enthaltenen Klassen nur die Klassen von Paketen in tieferen Schichten nutzen können. Das folgende Bild zeigt beispielhaft ein Schichtenmodell für drei Pakete:

Bild 12-2 Schichtenmodell für Pakete[167]

Eine rekursive Benutzung (Paket `A` nutzt Paket `B`, Paket `B` nutzt Paket `A`) sollte aus Gründen der Überschaubarkeit vermieden werden.

12.3 Erstellung von Paketen

Ein Paket wird definiert, indem **alle Dateien** des Pakets mit der **Deklaration** des **Paketnamens** versehen werden.

Die Deklaration eines Paketnamens erfolgt in Java mit Hilfe des Schlüsselworts **`package`** wie in folgendem **Beispiel**:

```
// Datei: Artikel.java
package lagerverwaltung;  // Deklaration des Paketnamens

public class Artikel      // Definition der Komponente Artikel des
{                         // Pakets lagerverwaltung
   private String name;
   private float preis;
```

[167] Der Pfeil bedeutet hier "benutzt".

```
   public Artikel (String name, float preis)
   {
      this.name = name;
      this.preis = preis;
   }
   // Es folgen die Methoden der Klasse
}
```

Dabei dürfen in einer Datei der Deklaration des Paketnamens allerhöchstens Kommentare vorausgehen. Die Klasse `Artikel` gehört also zum Paket `lagerverwaltung`.

Paketnamen werden konventionsgemäß klein geschrieben.

Pakete können aus verschiedenen Quellcode-Dateien bestehen. Eine jede Übersetzungseinheit (Quellcode-Datei), die zu einem Paket gehört, muss mit derselben Paketdeklaration beginnen. Alle Programmeinheiten einer Quellcode-Datei gehören auf jeden Fall zum gleichen Paket. Daraus resultiert, dass es **für eine bestimmte Datei nur eine einzige Paketdeklaration** geben darf.

Enthält eine **Datei** eine **Klasse** mit der Sichtbarkeit `public`, so muss der **Dateiname gleich** sein **wie der Name der** `public` **Klasse**. **Maximal eine Klasse einer Quellcode-Datei kann** `public` **sein**. Soll eine Klasse von einem anderen Paket aus nutzbar sein, so muss sie `public` sein. Ist sie es nicht, so ist sie nur innerhalb ihres eigenen Pakets als **Service-Klasse** (**Hilfsklasse**) verwendbar.

Das folgende **Beispiel** demonstriert das Sprachmittel der Pakete:

```
// Datei: Lager.java
// Der Dateiname muss nicht dem Namen einer Klasse entsprechen,
// sofern keine Klasse in der Datei public ist.
package lagerverwaltung;

class Kunde
{
   private String name;
   private String vorname;
   private int kundennummer;

   public Kunde (String n, String v, int knr)
   {
      name = n;
      vorname = v;
      kundennummer = knr;
   }
   // Es folgen die Methoden der Klasse
}
```

```
class Lieferant
{
   private String lieferantenName;
   private int lieferantenNummer;

   public Lieferant (String name, int nummer)
   {
      lieferantenName = name;
      lieferantenNummer = nummer;
   }
   // Es folgen die Methoden der Klasse
}
```

In diesem Beispiel gehören die Klassen `Kunde` und `Lieferant` zum Paket `lagerverwaltung`. Keine der beiden Klassen ist `public`. Dies bedeutet, dass beide Klassen nur interne Hilfsklassen im Paket `lagerverwaltung`[168] sind und von Klassen in anderen Paketen nicht genutzt werden können. Sie können nur von Klassen des Pakets `lagerverwaltung` verwendet werden.

Enthält eine Quellcode-Datei keine Klasse mit der Sichtbarkeit `public`, so kann der Dateiname beliebig sein, vorausgesetzt, der Dateiname ist syntaktisch zulässig.

Unterpakete

Ein Paket selbst kann wiederum **Unterpakete** enthalten. Zum Beispiel könnte es ein Paket `betriebsverwaltung` geben, das die Pakete `lagerverwaltung`, `personalverwaltung` und `finanzverwaltung` enthält. Die Deklaration des Paketes `lagerverwaltung`, die als erste Codezeile in jeder Datei stehen muss, die zu diesem Paket gehört, sieht dann folgendermaßen aus:

```
package betriebsverwaltung.lagerverwaltung;
```

Auf diese Art und Weise können beliebig tiefe Pakethierarchien aufgebaut werden.

12.4 Benutzung von Paketen

Sind die Klassen `A` und `B` einem Paket namens `paket` zugeordnet, so sind diese Klassen **Komponenten** bzw. **Elemente** des Pakets `paket`.

Genauso wie die Komponenten von Klassen – die Datenfelder und Methoden – mit Hilfe des Punktoperators angesprochen werden können, können auch die Komponenten von Paketen, also die Klassen – bzw. Schnittstellen oder Unterpakete – mit Hilfe des Punktoperators angesprochen werden.

[168] Das Paket `lagerverwaltung` kann verschiedene Dateien umfassen.

Soll aus einer Klasse `C` heraus, die nicht Bestandteil des Pakets `paket` ist, die Klasse `A` des Pakets `paket` angesprochen werden, so erfolgt dies mit `paket.A`. In folgendem Beispiel wird in einer Klasse des Pakets `kreiseckpaket` die Klasse `Eck` aus dem Paket `eckpaket` und die Klasse `Kreis` aus dem Paket `kreispaket` verwendet:

```
// Datei: Kreiseck.java

package kreiseckpaket;

public class Kreiseck
{
   eckpaket.Eck eckRef       = new eckpaket.Eck();
   kreispaket.Kreis kreisRef = new kreispaket.Kreis();
}
```

Die import-Vereinbarung

Stellt man alle Klassen zu Paketen zusammen, so findet man es bald lästig, die voll qualifizierten Klassennamen, d. h. Paketnamen gefolgt von Punktoperator und Klassennamen, niederzuschreiben. Um diese unliebsame Schreibarbeit einzusparen, wird die `import`-Vereinbarung benutzt. Die `import`-Vereinbarung ermöglicht es, dass auf eine Klasse oder eine Schnittstelle in einem anderen Paket, die den Zugriffsmodifikator `public` besitzt, direkt über ihren Namen zugegriffen werden kann, ohne dass diesem Namen die Paketstruktur getrennt durch einen Punkt vorangestellt werden muss.

Mit `public` deklarierte Klassen können mittels der `import`-Vereinbarung in anderen Paketen sichtbar gemacht werden.

Die `import`-Vereinbarung muss hinter der `package`-Deklaration, aber vor dem Rest des Programms stehen.

Es können beliebig viele `import`-Vereinbarungen aufeinanderfolgen. Das oben gezeigte Beispiel der Datei `Kreiseck.java` wird nun mit Hilfe der `import`-Vereinbarung realisiert:

```
// Datei: Kreiseck.java

package kreiseckpaket;

import kreispaket.*;
import eckpaket.*;

public class Kreiseck
{
   Eck eckRef       = new Eck();
   Kreis kreisRef   = new Kreis();
}
```

Mit

```
import kreispaket.*;
```
[169]

werden alle Klassen und Schnittstellen mit der Sichtbarkeit `public` des Pakets `kreispaket` importiert.

Unterpakete, die in einem Paket enthalten sind, **werden nicht zusammen mit den Klassen bzw. Schnittstellen dieses Pakets importiert**. Sie müssen explizit als Unterpaket importiert werden.

Wie dies zu erfolgen hat, wird noch gezeigt.

Import einzelner Klassen oder Schnittstellen

Soll nur eine einzige Klasse oder Schnittstelle eines Pakets importiert werden, so wird der entsprechende Name hinter dem Punkt angegeben, wie z. B.

```
import kreispaket.Kreis;
```

Werden jedoch zwei gleichnamige Klassen oder Schnittstellen importiert, so muss stets der **voll qualifizierte Name** angegeben werden, um eine Eindeutigkeit herzustellen.

Ein **voll qualifizierter Name** bezeichnet den Namen einer Klasse, der die Klasse durch Angabe der Paketstruktur gefolgt von einem Punkt und dem eigentlichen Klassennamen identifiziert.

Import von Unterpaketen

Natürlich muss es auch eine Möglichkeit geben, die Klassen von Unterpaketen zu importieren. Dies ist einfach durch die Anwendung des Punktoperators für das entsprechende Unterpaket möglich. Mit

```
import betriebsverwaltung.lagerverwaltung.*;
```

werden alle `public`-Klassen und `public`-Schnittstellen, die sich im **Unterpaket** `lagerverwaltung` befinden, importiert.

Richtet man eine Unterpaketstruktur ein, so spiegelt sich diese sowohl in der Paketdeklaration als auch in der `import`-Vereinbarung wider. Will eine Client-Klasse beispielsweise eine Klasse oder Schnittstelle aus dem Paket mit der Paketdeklaration `package betriebsverwaltung.lagerverwaltung` verwenden, so muss dafür die `import`-Vereinbarung `import betriebsverwaltung.lagerverwaltung.*` in der Client-Klasse angeschrieben werden.

[169] Das Sternchen `*` stellt eine sogenannte Wildcard dar. An die Stelle der Wildcard kann jeder beliebige Bezeichner treten.

Automatischer Import

Die Verwendung einer `import`-Vereinbarung ist für den Programmablauf nicht unbedingt nötig, sie kann dem Programmierer aber viel Schreibarbeit ersparen. Es gibt sogar einen Fall, bei dem der Compiler die Schreibarbeit für die `import`-Vereinbarung übernimmt:

Das **Paket `java.lang`** aus der Java-API wird automatisch in jede Quellcode-Datei importiert.

Static Imports

Seit Java 5 gibt es zusätzlich die sogenannten **Static Imports**. Vorher war es nur möglich, alle Klassen oder Schnittstellen der Sichtbarkeit `public` aus anderen Paketen zu importieren. Wurden Klassenmethoden oder -variablen aus anderen Klassen benötigt, so mussten diese Klassen insgesamt importiert oder die entsprechenden statischen Elemente über den Klassennamen qualifiziert werden.

Die **normale** `import`-Vereinbarung erlaubt es, direkt Klassen oder Schnittstellen zu verwenden, ohne dass ihr Name bei der Verwendung durch die Angabe des Paket-Pfades qualifiziert werden muss.

Oftmals werden Hilfsmethoden als Klassenmethoden von der Klassenbibliothek zur Verfügung gestellt. So enthält beispielsweise die Java-Klasse `java.lang.Math` eine ganze Reihe von Klassenmethoden für die Berechnung mathematischer Funktionen. In der Vergangenheit musste dabei stets der Klassenname mit angegeben werden, wenn man eine solche Klassenmethode einsetzte.

Hierfür ein **Beispiel**:

```
class A
{
   . . . . .
   double wert = 3.;
   // Berechnung der Quadratwurzel aus 3.
   double quadratWurzelAusWert = Math.sqrt (wert);
   . . . . .
}
```

Dieses Beispiel kann mit Hilfe der `static import`-Vereinbarung nun so geschrieben werden, dass `sqrt()` ohne den qualifizierenden Zugriff im Programm verwendet werden kann.

Dies zeigt das folgende **Beispiel**:

```
import static java.lang.Math.sqrt;

class A
{
   . . . . .
   double wert = 3.0;
```

```
    double quadratWurzelAusWert = sqrt (wert);
    . . . . .
}
```

Die static import-Vereinbarung erlaubt es, direkt Klassenvariablen oder Klassenmethoden zu verwenden, ohne sie durch die Angabe des Klassennamens qualifizieren zu müssen.

Das Importieren statischer Klassenelemente geschieht mit folgender Syntax:

```
import static paketname.Klassenname.statElement;
```

bzw.

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

Die erste Variante importiert nur ein einzelnes statisches Element der Klasse `paketname.Klassenname`. Die zweite Version hingegen importiert alle Klassenmethoden und Klassenvariablen der Klasse `paketname.Klassenname`. Anschließend können die statischen Elemente ohne weitere Qualifizierung durch den Klassennamen verwendet werden.

Allerdings sollte man mit dem Gebrauch von Static Imports vorsichtig und sparsam sein. Werden zu viele statische Elemente importiert, dann lässt sich nach einiger Zeit nicht mehr nachvollziehen, woher diese Elemente stammen, und darüberhinaus wird der Quellcode unleserlich.

12.5 Paketnamen

Kapitel 12.5.1 zeigt, dass die Struktur eines Paketnamens auf die Verzeichnisstruktur des Rechners abgebildet wird. In Kapitel 12.5.2 wird der Begriff des Klassenpfads erläutert und in Kapitel 12.5.3 wird das Setzen des Klassenpfads besprochen. Kapitel 12.5.4 und 12.5.5 befassen sich mit eindeutigen Paketnamen und anonymen Paketen (Default-Paketen).

12.5.1 Paketnamen und Verzeichnisstruktur

Ein voll qualifizierter Paketname wird in Java in die Verzeichnisstruktur des Rechners umgesetzt. Dabei müssen alle Dateien, die zu einem Paket gehören, in einem Verzeichnis liegen, dessen Name identisch zum Paketnamen ist.

Das folgende Beispiel zeigt exemplarisch den Aufbau einer Verzeichnisstruktur:

Bild 12-3 Verzeichnisstruktur mit Dateien im Dateisystem[170]

Jeder Verzeichnisname entspricht einem einfachen Paketnamen.

Es ist zu beachten, dass

- für die Bezeichner eines Verzeichnisses oder einer Datei dieselben Einschränkungen wie bei Variablennamen gelten,
- die Verzeichnisnamen den Paketnamen entsprechen,
- Paketnamen konventionsgemäß stets vollständig klein geschrieben werden, auch bei zusammengesetzten Namen,
- jedes Unterpaket ein Unterverzeichnis darstellt.

12.5.2 Klassenpfad oder Klassensuchpfad

Da ein einfacher Verzeichnisname einen Knoten in einem Pfad darstellt, muss zum Zugriff auf ein bestimmtes Paket der ganze Pfad bekannt sein.

Der **Klassenpfad** (engl. **class path**) oder besser **Klassensuchpfad** ist der Pfad, in dem das Java-Laufzeitsystem (Java runtime environment), also beispielsweise der Interpreter `java`, aber auch der Compiler `javac` nach Klassen und nach anderen erforderlichen Dateien sucht, die vom Anwender oder von Drittanbietern zur Verfügung gestellt werden und nicht Teil der Java-API sind.

Der Compiler `javac` nutzt zusätzlich beim Kompilieren den Klassenpfad, um nach Sourcen zu suchen und dadurch Abhängigkeiten zu anderen Klassen aufzulösen.

Es ist auch möglich, ohne den Klassenpfad zu arbeiten, wenn nur Klassen verwendet werden, die sich in einem Hierarchiebaum gerade im aktuell gesetzten Verzeichnis befinden.

[170] Verzeichnisse sind abgerundet, Dateien rechteckig gezeichnet.

Sollte jedoch mit Dateien wechselnder Knoten in der Verzeichnishierarchie gearbeitet werden, so ist es erforderlich, den jeweiligen Klassenpfad zu setzen.

Wird beispielsweise der Klassensuchpfad auf `C:\projekte\projekt1\classes\` gesetzt, so wird nach den Klassen in den Paketen unterhalb des Verzeichnisses `classes` gesucht. Dabei müssen diese Verzeichnisse bzw. Pakete natürlich mit ihrem Namen angegeben werden. Befindet man sich beispielsweise momentan im Verzeichnis `C:\test`, hat den Klassenpfad aber wie beschrieben gesetzt und hat die Klasse `paket1.Klasse1` in den Paketen unterhalb des Verzeichnisses `classes` eine `main()`-Methode, so wird diese durch den Aufruf

```
java paket1.Klasse1
```

aufgerufen.

Der Weg, den der Interpreter gehen muss, um zur Klasse `paket1.Klasse1` zu gelangen, wird dann durch zwei Teile bestimmt:

Fehlt einer der beiden Teile, kann der Interpreter die Klasse nicht finden. Beachten Sie bitte, dass der Name `paket1` im Betriebssystem ebenfalls ein Verzeichnis darstellt. Dieses Verzeichnis ist im Kontext der Klasse `Klasse1` mit der Semantik "Paket" belegt.

Um im obigen Beispiel eine Klasse, die zu dem Paket `paket1` gehört, zu kompilieren, wird der Compiler mit der Angabe von `javac paket1\Klasse1.java` aufgerufen. Dabei ist zu beachten, dass der Aufruf aus dem Verzeichnis erfolgt, welches das Paket bzw. das Verzeichnis `paket1` enthält, im Beispiel ist es das Verzeichnis `C:\projekte\projekt1\classes`.

Eine Klasse `Klasse1` innerhalb eines Paketes `paket1` kann kompiliert werden durch den Aufruf:

```
javac paket1\Klasse1.java
```

Diese Klasse `Klasse1` kann gestartet werden durch den Aufruf:

```
java paket1.Klasse1
```

12.5.3 Umgebungsvariable `CLASSPATH` und Option `-classpath`

Es gibt zwei Möglichkeiten, den Klassenpfad zu setzen:

- über die Option `-classpath` und
- über die Umgebungsvariable `CLASSPATH`.

Generell wird das Setzen der Option `-classpath` vorgezogen, da man diese Option für jede Anwendung getrennt setzen kann und damit die anderen Anwendungen nicht beeinflusst im Gegensatz zum zentralen Setzen der Umgebungsvariablen `CLASSPATH`.

Im Folgenden soll zunächst auf die Umgebungsvariable `CLASSPATH` eingegangen werden und anschließend auf die Option `-classpath`.

12.5.3.1 Die Umgebungsvariable `CLASSPATH`

Der Klassensuchpfad oder Klassenpfad `CLASSPATH` ist eine Umgebungsvariable, die den Pfad enthält, in dem Java-Tools und Anwendungen nach Klassen, Sourcen oder Ressourcen suchen, die nicht Teil der Java-API sind, also Klassen, Sourcen oder Ressourcen des Nutzers oder von dritten Parteien.

Der **vollständige Paketname** ist vom Klassenpfad aus anzugeben. Ein vollständiger Paketname setzt sich aus den einzelnen Paketnamen, die den Verzeichnisnamen entsprechen, zusammen. Für die Angabe eines vollständigen Paketnamens bei geschachtelten Pakethierarchien werden die **Unterpakete** von den **übergeordneten Paketen durch Punkte getrennt**. Der vollständige Paketname des Paketes `sub1` aus der in Bild 12-3 vorgestellten Paketstruktur ist somit `paket1.sub1`. Der Zugriff auf eine Klasse muss immer über den vollständigen Paketnamen erfolgen. Auf die Klasse `Klasse3` kann entsprechend mit `paket1.sub1.Klasse3` zugegriffen werden.

Es ist möglich, in der Umgebungsvariable `CLASSPATH` auch mehrere Suchpfade anzugeben. Die Suchpfade müssen durch ein Semikolon `;` (Windows) bzw. Doppelpunkt `:` (UNIX) voneinander getrennt werden. Bei einer `import`-Vereinbarung sucht der Compiler nach den Paketen in den Verzeichnissen der verschiedenen Alternativen der Umgebungsvariablen `CLASSPATH`.

So wird beispielsweise bei

```
CLASSPATH=C:\projekte\projekt1\classes;D:\weitereKlassen
```

nach Klassen sowohl in dem Verzeichnis `C:\projekte\projekt1\classes` als auch im Verzeichnis `D:\weitereKlassen` gesucht.

Das aktuelle Arbeitsverzeichnis wird durch einen Punkt `.` symbolisiert. Die Angabe von `..` würde für ein übergeordnetes Verzeichnis stehen, ausgehend von der Position im Verzeichnisbaum, wo der Compiler oder der Interpreter gestartet wird. Wenn der Klassenpfad nicht explizit gesetzt wird, werden Klassen, Sourcen und Ressourcen im aktuellen Verzeichnis gesucht. Die Verzeichnisse, die die Klassen der Java-API enthalten, werden in jedem Fall durchsucht.

12.5.3.2 Die Option -classpath

Es gibt noch eine weitere Möglichkeit, den Compiler `javac` bzw. den Interpreter `java` aufzurufen, ohne die Umgebungsvariable `CLASSPATH`, d. h. den Klassenpfad, explizit zu setzen. Und zwar ist für beide Programme die Option `classpath` definiert. Wird beispielsweise die obige Verzeichnisstruktur `C:\projekte\projekt1\classes` zugrunde gelegt, in der sich das Paket `paket1` mit der Datei `Klasse1.class` befindet, so kann die Klasse `Klasse1` vom Interpreter aus jedem beliebigen Verzeichnis heraus auch folgendermaßen gestartet werden:

```
java -classpath C:\projekte\projekt1\classes paket1.Klasse1
```

12.5.4 Eindeutige Paketnamen

Möchte man seine Pakete nicht nur selbst verwenden, sondern sie einem größeren Benutzerkreis zur Verfügung stellen, so sollte man sich um eindeutige Paketnamen bemühen. Um weltweit eindeutige Paketnamen zu erhalten, macht man sich die Internet-Domain-Namen, die eine weltweite Eindeutigkeit garantieren, zunutze. Dies bedeutet aber nicht, dass man über den Internet-Domain-Namen auf Klassen zugreifen kann, die auf dem entsprechenden Rechner im Internet liegen. Nur der Paketname soll eindeutig sein! Möchte man also für seine Programme eindeutige Paketnamen haben, so sollte man die folgende **Konvention** verwenden: Der Internet-Domain-Name ist in umgekehrter Reihenfolge vor den Rest des Namens zu stellen. Das heißt, dass beispielsweise aus dem Domain-Namen `oracle.com` der Paketname `com.oracle` wird. Somit erhält man einen eindeutigen Paketnamen.

12.5.5 Anonyme Pakete

Wird in einer Übersetzungseinheit – das heißt einer Quellcode-Datei – kein Paketname deklariert, so gehört diese Übersetzungseinheit zu einem **anonymen** oder **unbenannten Paket**. Ein anonymes Paket wird auch als **Default-Paket** bezeichnet.

Alle Klassen einer solchen Quellcode-Datei gehören also zu einem anonymen Paket. Alle Dateien, die sich innerhalb desselben Verzeichnisses befinden und die nicht explizit einem Paket zugeordnet wurden, gehören dann automatisch zum gleichen anonymen Paket.

Dies ist vor allem bei kleinen Testprogrammen sinnvoll, da man sich dann nicht um Pakete kümmern muss. Bei größeren Projekten sollte man sich jedoch auf jeden Fall über die Aufteilung der Anwendung in Pakete Gedanken machen.

12.6 Gültigkeitsbereich von Klassennamen

Das folgende **Beispiel** demonstriert, dass sich der **Gültigkeitsbereich** eines **Klassennamens auf das ganze Paket erstreckt**, also übergreifend über verschiedene Dateien ist.

Ein eingeführter Klassenname gilt also automatisch an jeder Stelle in sämtlichen Dateien, die zum selben Paket gehören. Die Klasse `Zensur`, die am Ende der Datei `Datei1.java` im folgenden Beispiel definiert wird, kann in dieser Datei bereits vor deren Definition verwendet werden. Ebenso kann sie in der Datei `Datei2.java`, die zum selben Paket gehört, problemlos genutzt werden.

Beachten Sie, dass keine der Klassen als `public` deklariert wird. Daher können die Dateinamen frei gewählt werden. Die Datei `Datei1.java` enthält zwei Klassen, die Klasse `personen.Student` und die Klasse `personen.Zensur`, die Datei `Datei2.java` enthält die Klasse `personen.Schueler`. Die Klassen `personen.Student` und `personen.Schueler` enthalten jeweils eine Methode `main()` zum Ausdrucken von Zeugnissen.

Hier die Datei `Datei1.java`:

```
// Datei: Datei1.java
package personen;
class Student
{
   public String name;
   public String vorname;
   public int matrikelnummer;
   public Zensur[] zensuren;

   public Student (String name, String vorname,
                   int matrikelnummer, Zensur[] zensuren)
   {
      this.name = name;
      this.vorname = vorname;
      this.matrikelnummer = matrikelnummer;
      this.zensuren = zensuren;
   }

   public void print()
   {
      System.out.println ("Name     : " + name);
      System.out.println ("Vorname  : " + vorname);
      System.out.println ("Matr. Nr : " + matrikelnummer);

      for (int i = 0; i < zensuren.length; i++)
      {
        System.out.println (zensuren [i].fach +
                            " : " + zensuren [i].note);
      }
   }

   public static void main (String[] args)
   {
      Zensur[] z = new Zensur [2];
      z [0] = new Zensur ("Mathe   ", 1.2f);
      z [1] = new Zensur ("Java    ", 1.0f);
      Student s = new Student ("Heinz", "Becker", 123456, z);
      s.print();
   }
}
```

```
class Zensur
{
   public String fach;
   public float note;

   public Zensur (String f, float n)
   {
      fach = f;
      note = n;
   }
}
```

Die Ausgabe des Programms ist:

```
Name     : Heinz
Vorname  : Becker
Matr. Nr : 123456
Mathe    : 1.2
Java     : 1.0
```

Und nun die Datei `Datei2.java`:

```
// Datei: Datei2.java

package personen;
class Schueler
{
   public String name;
   public String vorname;
   public Zensur[] zensuren;

   public Schueler (String name, String vorname, Zensur[] zensuren)
   {
      this.name = name;
      this.vorname = vorname;
      this.zensuren = zensuren;
   }

   public void print()
   {
      System.out.println ("Name     : " + name);
      System.out.println ("Vorname  : " + vorname);

      for (int i = 0; i < zensuren.length; i++)
      {
         System.out.println (zensuren [i].fach +
                             " : " + zensuren [i].note);
      }
   }

   public static void main (String[] args)
   {
      Zensur[] z = new Zensur [2];
      z[0] = new Zensur ("Mathe   ", 1.2f);
      z[1] = new Zensur ("Deutsch ", 2.0f);
```

```
        Schueler s = new Schueler ("Brang", "Rainer", z);
        s.print();
    }
}
```

Die Ausgabe des Programms ist:

```
Name     : Brang
Vorname  : Rainer
Mathe    : 1.2
Deutsch  : 2.0
```

Der Gültigkeitsbereich eines Klassennamens erstreckt sich über alle Dateien eines Pakets. Der Compiler geht in Java mehrfach über den Quellcode, bis er alle Klassendeklarationen gefunden hat.

12.7 Zugriffsmodifikatoren

Pakete entsprechen Verzeichnissen. Natürlich können Verzeichnisse – und damit die Pakete – für bestimmte Nutzergruppen durch Mittel des Betriebssystems gesperrt sein. Im Folgenden wird davon ausgegangen, dass keine Sperrung durch Mittel des Betriebssystems erfolgt.

Zur Regelung des **Zugriffsschutzes** in Java gibt es als Schlüsselwörter die **Zugriffsmodifikatoren** **`public`**, **`protected`** und **`private`**.

Ohne Zugriffsmodifikator ist der **Zugriffsschutz default** (**friendly**). Beachten Sie, dass **default** (bzw. **friendly**) **kein Schlüsselwort** von Java ist.

Während für Methoden, Datenfelder und Konstruktoren alle Zugriffsmodifikatoren – und auch das Weglassen eines Zugriffsmodifikators – erlaubt sind, kommt für Klassen[171] und Schnittstellen nur der Zugriffsmodifikator `public` oder aber kein Zugriffsmodifikator (default) in Frage. In den nächsten Kapiteln werden alle Fälle detailliert diskutiert.

12.7.1 Zugriffsschutz für Klassen und Schnittstellen

Zum Zugriff auf Klassen und Schnittstellen in einem Paket gibt es für den Zugriffsschutz nur die beiden Möglichkeiten:

- default (friendly)
- oder `public`.

[171] Anders sieht es bei Elementklassen, die geschachtelte Klassen darstellen, aus.

Eine Klasse oder Schnittstelle in einem Paket ist für Klassen bzw. Schnittstellen **aus anderen Paketen nur sichtbar** und kann damit beispielsweise durch `import` erreicht werden, wenn sie mit dem Zugriffsmodifikator `public` versehen ist. Ist der Zugriffsschutz einer Klasse oder Schnittstelle default, so ist sie nur für Klassen bzw. Schnittstellen desselben Paketes sichtbar.

Selbst in Unterpaketen ist eine Klasse oder Schnittstelle, die den Zugriffsschutz default hat, nicht sichtbar. Das folgende Beispiel demonstriert die Sichtbarkeit von Klassen in Paketen:

```
// Datei: Artikel.java
package lagerverwaltung;

public class Artikel
{
   . . . . .
}

// Diese Klasse hat den Zugriffsschutz default
class Lieferant
{
   . . . . .
}

// Datei: Materialabrechnung.java
package abrechnung;

import lagerverwaltung.Artikel;
//import lagerverwaltung.Lieferant;    // Fehler, da nicht public

public class Materialabrechnung
{
   . . . . .
}
```

12.7.2 Zugriffsschutz für Methoden und Datenfelder

Für ein Datenfeld und eine Methode einer Klasse gibt es den Zugriffsschutz:

- default (friendly),
- `public`,
- `protected`
- und `private`.

Alle Datenfelder und Methoden innerhalb einer **Schnittstelle** sind dagegen implizit `public`. Werden sie explizit auf `private` bzw. `protected` gesetzt, so resultiert ein Kompilierfehler.

Schnittstellen werden in Kapitel 14 behandelt.

Der Zugriffsschutz von Datenfeldern und Methoden wird anhand von Bild 12-4 erläutert. Dabei soll gezeigt werden, dass mit dem Zugriffsmodifikator `private` geschützte Datenfelder und Methoden einer Klasse den größten Zugriffsschutz besitzen, danach folgen default, `protected` und `public`. Für die Diskussion wird angenommen, dass die **Klasse `A` `public`** ist, sodass aus anderen Paketen auf sie zugegriffen werden kann. Im Folgenden das bereits erwähnte Szenario zur Diskussion des Zugriffsschutzes:

Bild 12-4 Anordnung der Klassen in Paketen

Die Klasse `A` im Paket `y` soll Datenfelder oder Methoden haben, die als Diskussionsgrundlage zuerst den Zugriffsschutz `private` haben sollen, dann default, danach `protected` und zum Schluss `public`.

Unabhängig davon, ob Instanzvariablen und Instanzmethoden oder Klassenvariablen und Klassenmethoden betrachtet werden, der Zugriffsschutz bleibt der gleiche, da der Zugriffsschutz in der Sprache Java **klassenbezogen** und nicht **objektbezogen** implementiert ist.

Deshalb wird im weiteren Verlauf nur noch von Datenfeldern und Methoden gesprochen.

Im Folgenden werden die vier verschiedenen Möglichkeiten für den Zugriffsschutz einzeln erläutert:

- **Zugriffsmodifikator `private`**

Auf Datenfelder oder Methoden, die mit dem Zugriffsmodifikator `private` geschützt sind, kann innerhalb der Klassendefinition, in der sie definiert sind, zugegriffen werden.

Das bedeutet, dass folgender Zugriff erlaubt ist[172]:

```
public class Punkt
{
   private int x;
   public void tausche (Punkt p)
   {
```

[172] Es gibt auch objektorientierte Programmiersprachen, bei denen der Zugriffsschutz objektbezogen ist. Dann würde das obige Beispiel nicht funktionieren, da dann jedes einzelne Objekt wirklich nur auf seine eigenen Datenfelder und Methoden mit der `this`-Referenz zugreifen kann.

```
        int help = p.x;
        p.x = x;
        x = help;
    }
}
```

Wird die Paketstruktur aus Bild 12-4 zugrunde gelegt, so kann aus keiner der Klassen `B`, `C`, `D` oder `E` auf die privaten Datenfelder und Methoden der Klasse `A` zugegriffen werden. Beachten Sie, dass in den folgenden Bildern ein **gestrichelter Pfeil mit** einem **Blitz** einen **verwehrten Zugriff** symbolisiert. Im Folgenden der Zugriff auf `private` Datenfelder und Methoden:

Bild 12-5 Zugriff auf `private` Datenfelder und Methoden

- **Zugriffsschutz `default`**

> Auf Datenfelder und Methoden, die den Zugriffsschutz default haben, kann aus Klassen heraus, die im gleichen Paket liegen, zugegriffen werden.

Der Zugriffsschutz gegenüber den mit `private` geschützten Datenfeldern und Methoden wird um die Zugriffsmöglichkeit von allen Klassen im gleichen Paket aufgeweicht. Hier die Visualisierung des Zugriffs auf Datenfelder und Methoden, bei denen der Zugriffsschutz default ist:

Bild 12-6 Zugriff auf default Datenfelder und Methoden

- **Zugriffsmodifikator `protected`**

Auf Datenfelder und Methoden, die den Zugriffsschutz `protected` haben, besteht ein erweiterter Zugriff gegenüber Datenfeldern und Methoden mit dem Zugriffsschutz default.

Auf Datenfelder und Methoden, die den Zugriffsschutz `protected` haben, kann aus allen Klassen im gleichen Paket zugegriffen werden, und zusätzlich können Subklassen in anderen Paketen auf die von der Vaterklasse ererbten Datenfelder und Methoden zugreifen. Bedingung ist allerdings, dass auf die eigen**en ererbten Datenfelder und Methoden zugegriffen wird** und nicht z. B. in der Subklasse `E` ein neues Objekt der Klasse `A` angelegt wird und dann versucht wird, auf die `protected` Datenfelder und Methoden des neu angelegten Objektes zuzugreifen.

Definiert die Klasse `A` z. B. eine `print()`-Methode mit dem Zugriffsmodifikator `protected`, so ist der Aufruf der Methode `print()` in den folgenden Anweisungen im Quellcode der Klasse `E` nicht zulässig:

```
A refA = new A();
refA.print();
```

Innerhalb der Klassendefinition von `E` kann aber auf die von der Vaterklasse `A` geerbte Methode `print()` zugegriffen werden. So kann an jeder Stelle im Programmcode der Klasse `E`, an der es erlaubt ist, eine Methode aufzurufen, die Anweisung `print();` stehen. Das folgende Bild visualisiert den Zugriff auf Datenfelder und Methoden, die den Zugriffsschutz `protected` aufweisen:

Bild 12-7 Zugriff auf `protected` Datenfelder und Methoden

- **Zugriffsmodifikator `public`**

Datenfelder und Methoden, die den Zugriffsmodifikator `public` besitzen, haben keinen Zugriffsschutz mehr.

Auf solche Datenfelder und Methoden kann von allen Klassen aus zugegriffen werden.

Dies symbolisiert das nächste Bild:

Bild 12-8 Zugriff auf `public` Datenfelder und Methoden

Das folgende Bild stellt den Zugriff auf Datenfelder und Methoden in einem Kreis dar:

Bild 12-9 Zugriff auf die Datenfelder und Methoden einer Klasse bzw. eines Objektes

Die folgende Tabelle fasst den Zugriffsschutz bei den unterschiedlichen Zugriffsmodifikatoren zusammen:

hat Zugriff auf	`private` Datenfelder und Methoden	default Datenfelder und Methoden	`protected` Datenfelder und Methoden	`public` Datenfelder und Methoden
Klasse `A` selbst	Ja	Ja	Ja	Ja
Klasse `B` gleiches Paket	Nein	Ja	Ja	Ja
Subklasse `C` gleiches Paket	Nein	Ja	Ja	Ja
Subklasse `E` anderes Paket	Nein	Nein	Ja/Nein	Ja
Klasse `D` anderes Paket	Nein	Nein	Nein	Ja

Tabelle 12-1 Zugriff auf Datenfelder und Methoden der Klasse `A`[173]

Dabei werden die Zugriffsmöglichkeiten der Klassen `A`, `B`, `C`, `D` und `E` aus Bild 12-4 auf Datenfelder und Methoden der Klasse `A` betrachtet.

Die Subklasse `E` hat nur Zugriff auf die geerbten Datenfelder und Methoden der Klasse `A`. Wird ein neues Objekt der Klasse `A` in `E` angelegt, so darf auf Datenfelder und Methoden dieses Objektes, die `protected` sind, nicht zugriffen werden. Man kann es auch einfach aus dem Gesichtspunkt betrachten, dass wenn `E` die Klasse `A` nicht im Sinne einer Vererbungsbeziehung benutzt – und das ist der Fall, wenn ein neues Objekt von `A` in `E` angelegt wird –, dass dann die Klasse `E` dieselben Zugriffsmöglichkeiten wie die Klasse `D` hat.

Bis auf `protected` ist der Zugriffsschutz gleich, egal ob auf geerbte Datenfelder und Methoden zugegriffen wird, oder ob in der entsprechenden Klasse ein neues Objekt der Klasse `A` angelegt wird und auf dessen Datenfelder und Methoden zugegriffen wird.

12.7.3 Zugriffsschutz für Konstruktoren

Stellt eine Klasse keinen Konstruktor mit dem Zugriffsmodifikator `public` bereit, sondern einen Konstruktor **ohne Zugriffsmodifikator**, so ist der Konstruktor nur von Klassen innerhalb des eigenen Pakets aufrufbar.

So kann im folgenden **Beispiel** der Konstruktor `Student (String n, String v, int nummer)` nur von Klassen im Paket `hochschule` aufgerufen werden:

```
// Datei: Student.java
```

[173] Die Klasse `A` hat natürlich den Zugriffsmodifikator `public` (`public class A{ . . .}`), damit der Zugriff auf die Klasse aus anderen Paketen möglich ist.

```
package hochschule;

public class Student
{
   private String name;
   private String vorname;
   private int matrikelnummer;
   Student (String n, String v, int nummer) // Konstruktor ohne
   {                                        // Zugriffsmodifikator
      name = n;
      vorname = v;
      matrikelnummer = nummer;
   }
}
```

Mit anderen Worten, hier ist es nur von Klassen innerhalb des Pakets `hochschule` aus möglich, Instanzen der Klasse `Student` zu schaffen.

Stellt eine Klasse Konstruktoren mit dem Zugriffsmodifikator `protected` zur Verfügung, so können von allen Klassen aus, die im selben Paket liegen, Objekte erzeugt werden. Abgeleitete Klassen in anderen Paketen können keine Objekte erzeugen, können aber den Konstruktor der Vaterklasse mit Hilfe von `super()` aufrufen.

Werden alle Konstruktoren einer Klasse für `private` erklärt, so kann von keiner anderen Klasse aus ein Objekt dieser Klasse erzeugt werden.

Nur innerhalb der Klasse selbst ist es noch möglich, Objekte dieser Klasse zu erzeugen. Diese Verhaltensweise wurde in Kapitel 10.5.2 dazu benutzt, um sicherzustellen, dass nur eine einzige Instanz einer Klasse erzeugt wird. Werden dagegen die Konstruktoren einer Klasse `public` gemacht, so kann von allen beliebigen Klassen aus ein Objekt dieser Klasse erzeugt werden.

Wird **überhaupt kein Konstruktor zur Verfügung gestellt**, so existiert der vom Compiler zur Verfügung gestellte Default-Konstruktor. Dieser Konstruktor hat den Zugriffsschutz der Klasse. Ist die **Klasse** `public`, so ist auch der **Default-Konstruktor** `public`. Ist die **Klasse default**, so ist auch der **Default-Konstruktor default**.

12.7.4 Zugriffsmodifikatoren beim Überschreiben von Methoden

Man darf die Zugriffsmodifikatoren einer überschriebenen Methode nicht einschränken, sondern nur erweitern, da sonst das liskovsche Substitutionsprinzip verletzt wird.

Man darf also zum Beispiel eine `protected`-Methode als `protected` oder `public` redefinieren, eine `public`-Methode aber nur als `public`. Die folgende Tabelle stellt die Zugriffsmodifikatoren beim Überschreiben von Methoden zusammen:

Zugriffsmodifikatoren in der Superklasse	**Zugriffsmodifikatoren in der Subklasse**
`private`	Kein Überschreiben möglich, aber neue Definition im Sohn.

default	default
	`protected`
	`public`

`protected`	`protected` `public`
`public`	`public`

Tabelle 12-2 Zugriffsmodifikatoren beim Überschreiben von Methoden

Der Grund für dieses Verhalten ist bereits in Kapitel 11.5.2 angesprochen worden.

Würde man die Zugriffsrechte beim Überschreiben einer Methode einschränken, so könnte nicht an jeder Stelle, an der ein Vater verlangt wird, ein Sohn stehen – der Vertrag der Klasse wäre verletzt, da die Vorbedingung verschärft wurde.

Bei Methoden, die als `private` deklariert sind, kann kein Überschreiben stattfinden, da sie zwar vererbt werden, aber im Code, der für den Sohn geschrieben wurde, nicht sichtbar sind.

12.8 Zusammenfassung

Kapitel 12 befasst sich mit den Paketen, der größten Einheit zur Strukturierung von Java-Programmen. Quellcode-Dateien werden Paketen zugeordnet. Diese Zuordnung entscheidet über die Sichtbarkeit der Programmelemente im Fall des Default-Zugriffs, bestimmt die Ablage im Dateisystem und bildet einen Namensraum, der es einem Programmierer ermöglicht, Klassen gleich zu benennen wie schon existente Klassen in anderen Paketen.

In Kapitel **12.1** wird die "Programmierung im Großen" vorgestellt und auf die Sprachmittel von Java eingegangen, die dabei Verwendung finden. Beim "Programmieren im Großen" geht es darum, ein Programm zu übersichtlich zu strukturieren. In Java gibt es zu diesem Zweck drei Sprachmittel: Klasse bzw. Thread (ein Thread ist eine spezielle Klasse), Schnittstelle (Interface) und Paket.

Eine Klasse implementiert einen abstrakten Datentyp und definiert die Methoden und die Daten der Klasse. Ein Thread ist eine spezielle Klasse, die einen Bearbeitungsablauf oder Handlungsfaden (engl. Thread) definiert, der parallel zu anderen Threads durchgeführt werden kann.

Eine Schnittstelle spezifiziert eine Aufrufschnittstelle. In einer Schnittstelle ist eine Zusammenstellung von Methodenköpfen enthalten, ggfs. kommen noch Konstanten hinzu. Eine Schnittstelle enthält keine Methodenrümpfe.

Pakete sind größere Einheiten für die Strukturierung von objektorientierten Systemen als die Klassen und dienen der Gruppierung. Zu einem Paket können mehrere Quellcode-Dateien gehören. In diesen Dateien können Klassen bzw. Threads und Schnittstellen enthalten sein. Pakete bilden eigene Bereiche für den Zugriffsschutz. Mit Paketen kann man kapseln (Information Hiding). Klassen, Threads und Schnittstellen, die im selben Paket liegen, haben wechselseitig mehr Zugriffsrechte – wenn keine Zugriffsmodifikatoren angegeben sind – als ein außenstehender Benutzer, der die Komponenten des Pakets benutzen will. Jedes Paket bildet einen eigenen Namensraum. Damit können Namenskonflikte vermieden werden und identische Namen für Klassen bzw. Schnittstellen in verschiedenen Paketen verwendet werden.

In Kapitel **12.2** wird die Gruppierungsfunktion von Paketen beleuchtet.

Pakete stellen die gröbsten Strukturierungseinheiten der objektorientierten Technik in Java dar. Pakete werden im Rahmen des Entwurfs der Software als Strukturierungseinheit konzipiert.

Pakethierarchien können zum Beispiel ein Schichtenmodell repräsentieren.

Kapitel **12.3** zeigt, wie Pakete erstellt werden.

Ein Paket wird definiert, indem alle Dateien des Pakets mit der Deklaration des Paketnamens versehen werden. Hierzu dient das Schlüsselwort `package`, vor dem in der Quellcode-Datei höchstens Kommentare auftauchen dürfen. Paketnamen werden konventionsgemäß klein geschrieben. Pakete können aus verschiedenen Quellcode-Dateien bestehen. Jede Übersetzungseinheit (Quellcode-Datei), die zu einem Paket gehört, muss mit der gleichen Paketdeklaration beginnen. Eine Quellcode-Datei kann nur maximal eine Paketdeklaration enthalten, d. h. einem Paket angehören.

Kapitel **12.4** geht auf die Benutzung von Paketen ein.

Genauso wie Komponenten von Klassen mit Hilfe des Punktoperators angesprochen werden können, können auch die Komponenten von Paketen, also die Klassen bzw. Schnittstellen oder Unterpakete, mit Hilfe des Punktoperators angesprochen werden.

Da die Verwendung der voll qualifizierten Klassennamen sehr umständlich ist, kann die `import`-Vereinbarung benutzt werden, um mit `public` deklarierte Klassen in anderen Paketen sichtbar zu machen. Von da an kann ohne die Angabe der Paketstruktur auf die Klassen zugegriffen werden. Die `import`-Vereinbarung muss hinter der `package`-Deklaration, aber vor dem Rest des Programms stehen. Unterpakete werden nicht automatisch mit importiert.

Zu den normalen Importen gibt es noch die Static Imports. Die `static import`-Vereinbarung erlaubt es, direkt Klassenvariablen oder Klassenmethoden zu verwenden, ohne sie durch die Angabe des Klassennamens qualifizieren zu müssen.

In Kapitel **12.5** werden die Paketnamen, deren Abbildung auf Verzeichnisse auf dem Rechner, ihre Eindeutigkeit und der Klassenpfad abgehandelt.

Bei Paketen ist zu beachten, dass

- für die Bezeichner eines Verzeichnisses oder einer Datei dieselben Einschränkungen wie bei Variablennamen gelten,
- die Verzeichnisnamen den Paketnamen entsprechen,
- Paketnamen konventionsgemäß stets vollständig klein geschrieben werden, auch bei zusammengesetzten Namen
- und dass jedes Unterpaket ein Unterverzeichnis darstellt.

Wird keine `package`-Deklaration vorgenommen, landen die Klassen und Interfaces der Quellcode-Datei im sogenannten Default-Paket (siehe Kapitel **12.5.5**). Das Default-Paket wird auch anonymes oder unbenanntes Paket genannt. Bei kleinen Testprogrammen ist die Verwendung des Default-Pakets problemlos. Bei größeren Programmen sollte immer ein eigener Namensraum nach den in Kapitel **12.5.4** beschriebenen Regeln angestrebt werden. Paketnamen werden per Konvention eindeutig gemacht, indem man den Internet-Domain-Namen in umgekehrter Reihenfolge als Paketstruktur vor den Rest des Namens stellt. Eine Firma mit der Domain `oracle.com` würde seine Paketstruktur mit `com.oracle` beginnen.

Der Klassenpfad (class path) oder besser Klassensuchpfad ist der Pfad, in dem das Java-Laufzeitsystem, also beispielsweise der Interpreter `java`, aber auch der Compiler `javac` nach Klassen und nach anderen erforderlichen Dateien sucht, die vom Anwender oder von Drittanbietern zur Verfügung gestellt werden und nicht Teil der Java-API sind.

Es gibt zwei Möglichkeiten, den Klassenpfad zu setzen:

- über die Option `-classpath` und
- über die Umgebungsvariable `CLASSPATH`.

Bevorzugt wird das Setzen der Option `-classpath`, da man diese Option für jede Anwendung getrennt setzen kann und damit keine andere Anwendung beeinflusst. Die Umgebungsvariable `CLASSPATH` wird zentral gesetzt. Es ist möglich, auch mehrere Suchpfade anzugeben. Die Suchpfade müssen durch ein Semikolon `;` (Windows) bzw. Doppelpunkt `:` (UNIX) voneinander getrennt werden. Bei einer import-Vereinbarung sucht der Compiler nach den Paketen in den Verzeichnissen der verschiedenen Alternativen der Umgebungsvariablen `CLASSPATH`.

Kapitel **12.6** handelt vom Gültigkeitsbereich von Klassennamen.

Der Gültigkeitsbereich eines Klassennamens erstreckt sich über alle Dateien eines Pakets. Der Compiler geht in Java mehrfach über den Quellcode, bis er alle Klassendeklarationen gefunden hat.

Kapitel **12.7** befasst sich mit den Zugriffsmodifikatoren, die auf die Sprachmittel von Java angewandt werden können.

Zur Regelung des Zugriffsschutzes in Java gibt es als Schlüsselwörter die Zugriffsmodifikatoren `public`, `protected` und `private`. Ohne Zugriffsmodifikator ist der Zugriffsschutz default. Beachten Sie, dass default kein Schlüsselwort von Java ist. Es wird beschrieben, welche Zugriffsmodifikatoren für Klassen und Schnittstellen, für Methoden und Datenfelder bzw. für Konstruktoren anwendbar sind. Es wird ferner gezeigt, wie sich die Zugriffsmodifikatoren beim Überschreiben von Methoden verhalten.

Zum Zugriff auf Klassen und Schnittstellen in einem Paket gibt es für den Zugriffsschutz nur die beiden Möglichkeiten:

- default (friendly)
- oder `public`.

Eine Klasse oder Schnittstelle in einem Paket ist für Klassen bzw. Schnittstellen aus anderen Paketen nur sichtbar und kann damit beispielsweise durch `import` erreicht werden, wenn sie mit dem Zugriffsmodifikator `public` versehen ist. Ist der Zugriffsschutz einer Klasse oder Schnittstelle default, so ist sie nur für Klassen bzw. Schnittstellen desselben Paketes sichtbar.

Für ein Datenfeld und eine Methode einer Klasse gibt es den Zugriffsschutz default (friendly), `public`, `protected` und `private`. Alle Datenfelder und Methoden innerhalb einer Schnittstelle sind dagegen implizit `public`. Werden sie explizit auf `private` bzw. `protected` gesetzt, so resultiert ein Kompilierfehler.

Auf Datenfelder oder Methoden, die mit dem Zugriffsmodifikator `private` geschützt sind, kann innerhalb der Klassendefinition, in der sie definiert sind, zugegriffen werden.

Auf Datenfelder und Methoden, die den Zugriffsschutz default haben, kann aus Klassen heraus, die im gleichen Paket liegen, zugegriffen werden.

Auf Datenfelder und Methoden, die den Zugriffsschutz `protected` haben, kann aus allen Klassen im gleichen Paket zugegriffen werden, und zusätzlich können Subklassen in anderen Paketen auf die von der Vaterklasse ererbten Datenfelder und Methoden zugreifen.

Datenfelder und Methoden, die den Zugriffsmodifikator `public` besitzen, haben keinen Zugriffsschutz mehr.

Bis auf `protected` ist der Zugriffsschutz gleich, egal ob auf geerbte Datenfelder und Methoden zugegriffen wird, oder ob in der entsprechenden Klasse ein neues Objekt der Klasse `A` angelegt wird und auf dessen Datenfelder und Methoden zugegriffen wird.

Stellt eine Klasse keinen Konstruktor mit dem Zugriffsmodifikator `public` bereit, sondern einen Konstruktor ohne Zugriffsmodifikator, so ist der Konstruktor nur von Klassen innerhalb des eigenen Pakets aufrufbar.

Stellt eine Klasse Konstruktoren mit dem Zugriffsmodifikator `protected` zur Verfügung, so können von allen Klassen aus, die im selben Paket liegen, Objekte erzeugt

werden. Abgeleitete Klassen in anderen Paketen können keine Objekte erzeugen, können aber den Konstruktor der Vaterklasse mit Hilfe von `super()` aufrufen.

Werden alle Konstruktoren einer Klasse für `private` erklärt, so kann von keiner anderen Klasse aus ein Objekt dieser Klasse erzeugt werden.

Wird überhaupt kein Konstruktor zur Verfügung gestellt, so existiert der vom Compiler zur Verfügung gestellte Default-Konstruktor. Dieser Konstruktor hat den Zugriffsschutz der Klasse. Ist die Klasse `public`, so ist auch der Default-Konstruktor `public`. Ist die Klasse default, so ist auch der Default-Konstruktor default.

Würde man die Zugriffsrechte beim Überschreiben einer Methode einschränken, so könnte nicht an jeder Stelle, an der ein Vater verlangt wird, ein Sohn stehen – der Vertrag der Klasse wäre verletzt, da die Vorbedingung verschärft wurde. Man darf die Zugriffsmodifikatoren einer überschriebenen Methode nicht einschränken, sondern nur erweitern, da sonst das liskovsche Substitutionsprinzip verletzt würde.

Bei Methoden, die als `private` deklariert sind, kann kein Überschreiben stattfinden, da sie zwar vererbt werden, aber im Code, der für den Sohn geschrieben wurde, nicht sichtbar sind.

12.9 Übungen

Aufgabe 12.1: Einfache Paketstruktur

Vervollständigen Sie die Klasse `Person`, die in einem Paket `pers` liegen soll und die Klasse `Student`, die im Paket `studi` liegen soll. Die Klasse `Student` soll von der Klasse `Person` abgeleitet sein. Vervollständigen Sie die Klasse `Test`, die je ein Objekt der Klassen `Person` und `Student` erzeugt und die Datenfelder dieser Objekte ausgibt. Die Klasse `Test` liegt im aktuellen Arbeitsverzeichnis. Welche Verzeichnisse müssen Sie einrichten? Wie lauten die Dateinamen Ihrer Programme in den Verzeichnissen?

Hier das zu ergänzende Programm:

```
// Datei: Person.java

. . . . .
import java.util.*;

public class Person
{
   private String name;
   private String vorname;

   public Person()
   {
      Scanner eingabe = new Scanner (System.in);
      System.out.print ("Geben Sie den Nachnamen ein: ");
      name = eingabe.next();

      System.out.print ("Geben Sie den Vornamen ein: ");
      vorname = eingabe.next();
   }
```

```
   public void print()
   {
      System.out.println ("Nachname: " + name);
      System.out.println ("Vorname: " + vorname);
   }
}

// Datei: Student.java
. . . . .
. . . . .
import java.util.*;

public class Student . . . . .
{
   private String matrikelnummer;

   public Student()
   {
      super();

      Scanner eingabe = new Scanner (System.in);
      System.out.print ("Geben Sie die Matrikelnummer ein: ");
      matrikelnummer = eingabe.next();
      System.out.println();
   }

   public void print()
   {
      . . . . .
      System.out.println ("Matrikelnummer: " + matrikelnummer);
   }
}

// Datei: Test.java
. . . . .
. . . . .
public class Test
{
   public static void main (String args[])
   {
      System.out.println ("Start des Programms");
      System.out.println();
      System.out.println ("Person erzeugen");
      . . . . .
      System.out.println();
      System.out.println ("Student erzeugen");
      . . . . .
      System.out.println();
      System.out.println ("Ausgabe Person");
      . . . . .
      System.out.println();
      System.out.println ("Ausgabe Student");
      . . . . .
   }
}
```

Aufgabe 12.2: Struktur mit Unterpaketen

Die einfache Paketstruktur aus Aufgabe 12.1 soll wie folgt erweitert werden: Erstellen Sie im Paket `pers` das Unterpaket `prof`. Dieses enthält die Klasse `Professor`, welche von der Klasse `Person` abgeleitet wird. Ein Objekt der Klasse `Professor` erzeugt dabei eine Instanz der Klasse `Student`. Erweitern Sie die Klasse `Test`, um Ihre neue Klasse `Professor` zu testen.

```
// Datei: Professor.java

package . . . . .;

import java.util.*;
import . . . . .;
import . . . . .;

public class Professor . . . . .
{
   private String fb;
   private Student stud;

   public Professor()
   {
      super();
      Scanner eingabe = new Scanner (System.in);
      System.out.print ("Geben Sie den Fachbereich ein: ");
      fb = eingabe.next();
      System.out.println();
      System.out.println ("Professor erstellt Student");
      stud = new Student();
   }

   public void print()
   {
      // Ausgabe der geerbten Instanzvariablen
      . . . . .
      System.out.println ("Fachbereich: " + fb);
      System.out.println ("Ausgabe des Studenten");
      . . . . .
   }
}
```

Aufgabe 12.3: Messwerte

Es sollen mehrere Klassen geschrieben werden, um Messwerte zu speichern und auszugeben. Entwickeln Sie die Klassen `Messwert`, `Messreihe` und `TemperaturMessreihe`.

Die Klasse `Messwert` soll folgende Kriterien erfüllen:

- Eine Klassenvariable `anzahlMesswerte` vom Typ `int` soll die Anzahl der Messwerte festhalten.
- Die Klasse soll die Datenfelder `wert` vom Typ `double`, `messDatum` vom Typ `GregorianCalendar` sowie `messwertID` vom Typ `int` enthalten.
- Die Klasse soll sich im Paket `messdaten` befinden.

- Es dürfen nur Klassen im selben Paket auf die Klasse `Messwert` zugreifen und sie verwenden.
- Der Konstruktor soll nur für Klassen im Paket `messdaten` aufrufbar sein. Der Konstruktor soll als Übergabeparameter `messwert` vom Typ `double` und `messDatum` vom Typ `GregorianCalendar` erwarten.

Folgende Methoden sollen implementiert werden:

- `double getWert()`
- `GregorianCalendar getMessDatum()`
- `int getMesswertID()`

Die Klasse `Messreihe` befindet sich ebenfalls im Paket `messdaten`, soll aber von Klassen in anderen Paketen verwendet werden können. Die Klasse erhält folgende Datenfelder und Methoden:

- `protected Messwert[] messwerte`

  Die Messwerte werden in diesem Array gespeichert.

- `public Messreihe (int messwertAnzahl)`

  Dem Konstruktor wird die Größe des Messwert-Arrays übergeben.

- ```
  public void addMesswert (double messwert,
                           GregorianCalendar datum)
  ```

  Fügt dem Array ein neues Messwert/Datum-Paar hinzu.

- `public double getMesswert (GregorianCalendar datum)`

  Ermittelt den Messwert, der zum übergebenen Datum gehört.

- `public void print()`

  Gibt alle gespeicherten Messwerte auf der Konsole aus.

Die Klasse `TemperaturMessreihe` wird von der Klasse `Messreihe` abgeleitet und befindet sich im Paket `temperaturmessung`. Sie soll folgende Datenfelder und Methoden erhalten:

- `private String temperaturEinheit`

  Gibt die verwendete Temperaturskala an, z. B. °C.

- ```
  public TemperaturMessreihe (int messwertAnzahl,
                              String temperaturEinheit)
  ```

  Der Konstruktor soll die Anzahl der zu speichernden Messwerte und die zu verwendende Temperaturskala entgegennehmen.

- `public void print()`

  Die Methode soll die verwendete Temperaturskala (z. B. °C) und alle gespeicherten Messwerte auf der Konsole ausgeben.

- `public static double celsiusToFahrenheit (double celsiusTemp)`

  Die Methode konvertiert eine Temperaturangabe von Celsius nach Fahrenheit.

Die entwickelten Klassen können mit folgender Testklasse, die sich im Default-Paket befindet, getestet werden:

```
// Datei: TestMesswerte.java

import temperaturmessung.TemperaturMessreihe;
import java.util.GregorianCalendar;

public class TestMesswerte
{

   public static void main (String[] args)
   {
      double fahrenheit;
      TemperaturMessreihe temperaturMessungen =
                                 new TemperaturMessreihe (5, "°C");

      GregorianCalendar datum1 = new GregorianCalendar (2000,5,10);
      temperaturMessungen.addMesswert (25.3, datum1);

      GregorianCalendar datum2 = new GregorianCalendar (2001,5,10);
      temperaturMessungen.addMesswert (23.0, datum2);

      GregorianCalendar datum3 = new GregorianCalendar (2002,5,10);
      temperaturMessungen.addMesswert (18.4, datum3);

      GregorianCalendar datum4 = new GregorianCalendar (2003,5,10);
      temperaturMessungen.addMesswert (26.9, datum4);

      GregorianCalendar datum5 = new GregorianCalendar (2004,5,10);
      temperaturMessungen.addMesswert (28.0, datum5);

      fahrenheit = TemperaturMessreihe.celsiusToFahrenheit (25.0);
      System.out.println("25.0 °C entsprechen " +
                         fahrenheit + "° F.");
      System.out.println();
      temperaturMessungen.print();
   }
}
```

Aufgabe 12.4: Maßeinheiten umrechnen

Entwickeln Sie die drei Klassen `Umrechner`, `Laenge` und `Temperatur` zum Umrechnen von Maßeinheiten.

Die Klassen `Laenge` und `Temperatur` befinden sich im Unterpaket `konstanten` des Pakets `umrechnung` und beinhalten die für die Umrechnung notwendigen Konstanten.

Die Klasse `Laenge` enthält folgende Datenfelder:

- `public static final float FAKTOR_MEILEN_NACH_KM`

  Diese Klassenkonstante enthält den Faktor 1,60934 für die Umrechnung von Meilen in Kilometer.

- `public static final float FAKTOR_KM_NACH_MEILEN`

  Diese Klassenkonstante enthält den Faktor 1/1,60934 für die Umrechnung von Kilometern in Meilen.

Die Klasse `Temperatur` enthält folgende Datenfelder:

- `public static final float FAKTOR_FAHRENHEIT_NACH_CELSIUS`

  Diese Klassenkonstante enthält den Faktor 5/9, der für die Umrechnung von °F in °C benötigt wird.

- `public static final float SUMMAND_FAHRENHEIT_NACH_CELSIUS`

  Diese Klassenkonstante enthält den Summanden 32, der für die Umrechnung von °F in °C benötigt wird.

- `public static final float FAKTOR_CELSIUS_NACH_FAHRENHEIT`

  Diese Klassenkonstante enthält den Faktor 9/5, der für die Umrechnung von °C in °F benötigt wird.

- `public static final float SUMMAND_CELSIUS_NACH_FAHRENHEIT`

  Diese Klassenkonstante enthält den Summanden 32, der für die Umrechnung von °C in °F benötigt wird.

Die Klasse `Umrechner` befindet sich im Unterpaket `tools` des Pakets `umrechnung` und enthält folgende Klassenmethoden:

- `public static float kmNachMeilen (float km)`

  Die Methode rechnet Längenangaben von Kilometern in Meilen um. Die Umrechnungsformel lautet: Meilen = km * (1/1,60934)

- `public static float meilenNachKm (float meile)`

  Die Methode rechnet Längenangaben von Meilen in Kilometer um. Die Umrechnungsformel lautet: km = Meilen * 1,60934

- `public static float celsiusNachFahrenheit (float celsius)`

  Die Methode rechnet Temperaturangaben von Celsius in Fahrenheit um. Die Umrechnungsformel lautet: °F = (°C * 9/5) + 32

- `public static float fahrenheitNachCelsius (float fahrenheit)`

  Die Methode rechnet Temperaturangaben von Fahrenheit in Celsius um. Die Umrechnungsformel lautet: °C = (°F – 32) * (5/9)

Die entwickelten Klassen können mit folgender Testklasse, die sich im Default-Paket befindet, getestet werden:

```
// Datei: TestEinheitenUmrechner.java

import umrechnung.tools.Umrechner;

public class TestEinheitenUmrechner
{
   public static void main (String[] args)
   {
      float km = 100f;
      float kmInMeilen;
      float meilen = 250.35f;
      float meilenInKm;
      float celsius = 0f;
      float celsiusInFahrenheit;
      float fahrenheit = 85f;
      float fahrenheitInCelsius;

      kmInMeilen = Umrechner.kmNachMeilen (km);
      meilenInKm = Umrechner.meilenNachKm (meilen);
      celsiusInFahrenheit =
                      Umrechner.celsiusNachFahrenheit (celsius);
      fahrenheitInCelsius =
                  Umrechner.fahrenheitNachCelsius (fahrenheit);

      System.out.println (km + " km entsprechen " +
                          kmInMeilen + " Meilen");
      System.out.println (meilen + " Meilen entsprechen " +
                          meilenInKm + " km");
      System.out.println (celsius + " °C entsprechen " +
                          celsiusInFahrenheit + " °F");
      System.out.println (fahrenheit + " °F entsprechen " +
                          fahrenheitInCelsius + " °C");
   }
}
```

Aufgabe 12.5: Flughafen-Projekt – Integration von Paketen

Das Programm von Projektaufgabe 11.5 beinhaltet inzwischen 11 Klassen. Um die Übersichtlichkeit zu verbessern, sollen die Klassen nun in eine sinnvolle Paketstruktur eingeordnet werden. Ein Vorschlag hierfür wäre:

Neben der Erstellung einer Paketstruktur soll eine Auswahl von Bahnen und Parkpositionen ermöglicht werden. Der bisherige Programmablauf hat einem Flugzeug eine beliebige Parkposition und eine beliebige Bahn zugewiesen. Dabei besitzt der Flughafen 4 Start-/Landebahnen und 10 Parkpositionen. Dem Lotsen soll nun ermöglicht werden, die Parkposition und auch die Start-/Landebahn auszuwählen. Hierzu sollten zwei Arrays `parkpositionen` und `bahnen` mit den entsprechenden Größen angelegt und mit Instanzen der Klasse `Parkposition` bzw. Instanzen der Klasse Bahn gefüllt werden. Fügen Sie folgende Methode der Hilfsklasse `Abfrage` hinzu. Diese Methode wird Ihnen bei der Eingabe eines Wertebereiches hilfreich sein:

```
public static int abfrageInt (String frage, int min, int max)
{
   int zahl = abfrageInt (frage);
   if (zahl < min || zahl > max)
   {
      System.out.println ("Bitte eine Zahl im Bereich von " + min +
                          " und " + max + " eingeben.");
      zahl = abfrageInt (frage, min, max);
   }
   return zahl;
```

Kapitel 13

Ausnahmebehandlung

13.1 Das Konzept des Exception Handling
13.2 Implementierung von Exception Handlern in Java
13.3 Die Klasse `Throwable`
13.4 Die Exception-Hierarchie
13.5 Behandlung von Ausnahmen
13.6 Vorteile des Exception-Konzeptes
13.7 Assertions
13.8 Zusammenfassung
13.9 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_13

13 Ausnahmebehandlung

Abnormale Ereignisse, die während des Programmablaufs auftreten, brechen die normale Ausführung eines Programms ab und bringen es zum Absturz. Diese abnormalen Ereignisse werden **Ausnahmen bzw. Exceptions** genannt.

Das Konzept des Exception Handling (dt. der Ausnahmebehandlung), das normalen Code von fehlerbehandelndem Code trennt, wird programmiersprachenunabhängig in Kapitel 13.1 vorgestellt. Kapitel 13.2 erläutert, wie das Konzept des Exception Handling in Java umgesetzt wird. Die gemeinsame Basisklasse aller Exceptions, die Klasse `Throwable`, wird in Kapitel 13.3 vorgestellt. Kapitel 13.4 zeigt den Klassenbaum der Exception-Klassen mit der Klasse `Throwable` als Wurzelklasse. Die wichtigsten Klassen dieses Klassenbaums werden vorgestellt.

In Java werden Ausnahmen, die vom Programmierer zwingend behandelt werden müssen (**Checked Exceptions**) von solchen unterschieden, die nicht zwingend behandelt werden müssen (**Unchecked Exceptions**).

Bei Unchecked Exceptions wird der Programmierer vom Compiler nicht darauf hingewiesen, dass er sich um die Exception kümmern muss. Bei Checked Exceptions wird zur Kompilierzeit überprüft, ob diese vom Programmierer behandelt werden.

In Kapitel 13.5 wird aufgezeigt, wie Ausnahmen in Java behandelt, d. h. durch sogenannte Exception Handler gefangen werden. Dabei spielt die Reihenfolge der Handler eine wichtige Rolle. Weiter können Ausnahmen an den Aufrufer weitergereicht werden. Dann übernimmt die aufrufende Methode die Ausnahmebehandlung.

Die Vorteile des Exceptions-Konzepts sind u. a. die saubere Trennung zwischen "normalem" Code und fehlerbehandelndem Code sowie die Möglichkeit, Programmfehler bereits zur Kompilierzeit durch Checked Exceptions aufzudecken.

Die Vorteile des Exception-Konzeptes werden in Kapitel 13.6 dargestellt.

Assertions sind in Java zwar mittels Exceptions realisiert, sie sollen aber im Gegensatz zu Exceptions nur während der Entwicklung eines Programms eingesetzt werden. Assertions stellen somit eine Möglichkeit zum Testen dar (Kapitel 13.7).

13.1 Das Konzept des Exception Handling

Während der normalen Abarbeitung einer Methode kann zur Laufzeit ein abnormales Ereignis auftreten, das die normale Ausführung der Methode unterbricht. Ein solches abnormales Ereignis ist eine **Exception bzw. Ausnahme**.

Eine **Exception** kann man als ein **durch eine Datenstruktur repräsentiertes Ereignis** auffassen. Tritt in einem Programm der Ausnahmezustand ein, so wird dieser Ausnahmezustand mit Hilfe der Datenstruktur einer Exception an das Programm gemeldet.

Dabei gilt jedoch, dass Exceptions nur **synchron** als Resultat von Anweisungen in einem Programm auftreten. Sie sind also nicht mit Interrupts oder anderen asynchronen Ereignissen zu verwechseln!

Eine häufige Quelle für Exceptions sind beispielsweise

- der Zugriff auf eine Referenz, die auf `null` zeigt,
- ungenügende Systemressourcen wie ein Mangel an Speicherplatz,
- eine Verletzung der Array-Grenzen,
- Ein- und Ausgabeoperationen
- oder Fehlfunktionen externer Services.

Eine Exception stellt damit ein **Laufzeit-Ereignis** dar, das zum Versagen einer Methode und damit zu einem Laufzeit-Fehler des Programms führen kann. In vielen Fällen führt das Auslösen einer Exception tatsächlich zum Versagen einer Methode und stellt dann auch eine Ausnahme dar. Mit anderen Worten: Eine solche Methode wirft eine Exception. Tatsächlich stellt in der Praxis das Versagen einer gerufenen Methode eine der Hauptquellen für Exceptions dar.

Es gibt aber auch die Möglichkeit, einen Exception Handler zu schreiben, in welchem auf Exceptions, die man vorausgesehen hat, so reagiert wird, dass sich das Programm von der Exception "erholt" und fehlerfrei weiterarbeitet.

Formal betrachtet tritt in einer Methode eine Exception auf, wenn trotz erfüllter Vorbedingung die Nachbedingung dieser Methode verletzt wird.

Ein defensiver Programmierstil gebietet es, auf Ausnahmen vorbereitet zu sein und zu verhindern, dass die Ausnahmen fehlerhafte Ergebnisse oder Ausfälle nach sich ziehen. Ein erstelltes Programm sollte stabil sein. Daraus folgt, dass man **Programmcode zur Erkennung** und **Behandlung von Exceptions** vorsehen muss.

Eine **traditionelle Methode** zur Behandlung von Fehlern ist die **Rückgabe eines Fehlercodes** durch Funktionen, entweder als direkter Rückgabewert oder über eine globale Variable. Oftmals wird auch beides gemacht wie z. B. bei UNIX-Systemaufrufen. Der direkte Rückgabewert (-1 bei UNIX) zeigt an, dass etwas schief gelaufen ist, und eine globale Variable (UNIX: `errno`) enthält den genauen Fehlercode. Eine globale Variable ist dann notwendig, wenn der Rückgabewert einer Funktion keine Lücken aufweist, die man zur Signalisierung von Fehlern nutzen kann. Dies gilt z. B. für viele mathematische Funktionen. Der zurückgelieferte Fehlercode muss nach jedem Aufruf geprüft werden. Diese Maßnahmen sind sehr aufwendig und resultieren nicht selten in Code, der nicht mehr sonderlich leicht zu lesen ist.

Ziel des Exception Handling ist es, **normalen und fehlerbehandelnden Code** übersichtlich **zu trennen** und **Ausnahmesituationen sicher zu behandeln.**

Das folgende Bild visualisiert die Technik der klassischen Fehlerbehandlung und das Konzept des Exception Handling:

Bild 13-1Klassische Fehlerbehandlung und Exception Handling

Ein weiteres Ziel des Exception Handling ist, **bei gewissen Ausnahmen eine Auseinandersetzung des Programms mit dem Fehler zu erzwingen.**

Es darf nicht sein, dass Folgeprobleme auftreten, weil eine Checked Exception nicht behandelt wurde.

Auch im Zusammenhang mit **Bibliotheken**, die ja eine immer größere Rolle in der Programmierung spielen, lassen sich Ausnahmen elegant einsetzen: Der **Ersteller einer Bibliothek** weiß sehr genau, wie er **Ausnahmen entdecken** kann. Er kann jedoch schwerlich eine optimale Lösung für die Behandlung dieser Ausnahmen in allen Anwendungen, die auf der Bibliothek aufsetzen, implementieren. Der **Anwendungsprogrammierer** steht vor dem umgekehrten Problem. Er weiß zwar, wie er **mit den Ausnahmen umzugehen** hat, aber da er die Implementierung der Bibliothek in der Regel nicht kennt und nach dem Prinzip des Information Hiding auch gar nicht kennen soll, kann er sie – wenn überhaupt – nur unter Mühen entdecken. Auch hier bietet das Konzept des Exception Handling eine leistungsfähige Lösung. Es ist möglich, Exceptions zu "werfen". So kann ein Bibliotheksprogramm Exceptions dem Anwendungsprogramm, das die Bibliothek benutzt, "zuwerfen" und dieses kann dann gezielt reagieren.

Exceptions ermöglichen einer Bibliothek, Ausnahmezustände in einfacher Weise an das aufrufende Programm zu melden und gegebenenfalls Daten über die näheren Begleitumstände zu liefern.

13.2 Implementierung von Exception Handlern in Java

Die Sprache Java hat als Strategie, Checked und Unchecked Exceptions einzuführen.

Wenn der Programmierer nicht alle Checked Exceptions abfängt, ist das Programm nicht lauffähig.

Damit wird das **Erkennen von Fehlern** von der Laufzeit auf die **Kompilierzeit** vorverlegt. Ehe auf Checked und Unchecked Exceptions eingegangen werden kann, soll zunächst erklärt werden, wie das Exception Handling in Java realisiert wird.

13.2.1 Die `try`-Anweisung

Das Exception Handling wird in Java durch eine **`try`-Anweisung** realisiert. Eine **`try`-Anweisung** umfasst in ihrer ursprünglichen Form einen **`try`-Block** und einen oder mehrere **`catch`-Blöcke** bzw. einen **`finally`-Block**. Ist mindestens ein `catch`-Block vorhanden, so kann der `finally`-Block entfallen. Ist kein `catch`-Block vorhanden, so ist der `finally`-Block erforderlich.

Auf den seit Java 7 möglichen multi-catch-Block wird in Kapitel 13.2.2 eingegangen.

Mit Hilfe des Schlüsselwortes `try` wird ein Block aus beliebigen Anweisungen des normalen Programms gekennzeichnet, dessen Ausführung "versucht" werden soll (**`try`-Block**), wobei aber Exceptions auftreten können, die eine normale Ausführung verhindern.

Eventuell auftretende Exceptions können durch einen oder mehrere sogenannte **Exception Handler** mit Hilfe eines `catch`-Blocks "gefangen" und behandelt werden. In anderen Worten, in Java wird ein Exception Handler durch einen `catch`-Block realisiert.

Eine **`try`-Anweisung** hat die folgende Struktur:

```
try
{                           // try-Block. Das ist der
   . . . . .                // normale Code, in dem
}                           // Fehler auftreten können
catch (Exceptiontyp1 name1)
{                           // catch-Block 1.
   . . . . .                // Fängt Fehler der
}                           // Klasse Exceptiontyp1 ab
catch (Exceptiontyp2 name2)
{                           // catch-Block 2.
   . . . . .                // Fängt Fehler der
}                           // Klasse Exceptiontyp2 ab
```

try-Block

catch-Block

```
. . .                          // weitere catch-Blöcke
                               // als Exception Handler
finally                        // finally-Block ist
{                              // optional, wenn mindestens
   . . . . .                   // ein catch-Block da ist.
}
```

Der `finally`-Block wird in jedem Fall durchlaufen, egal ob ein Fehler aufgetreten ist oder nicht.

13.2.2 Der multi-catch-Block

Die Struktur einer `try`-Anweisung kann seit Java 7 verkürzt werden, indem mehrere `catch`-Blöcke zu einem einzigen `catch`-Block, dem sogenannten **multi-catch-Block,** zusammengefasst werden. Dieser `catch`-Block beinhaltet dann mehrere Ausnahmetypen, die von ihm gefangen werden können. Realisiert wird das Fangen mehrerer Ausnahmetypen mit der binären Oder-Verknüpfung (`|`) in der Parameterliste des `catch`-Blocks. Dadurch wird die Zahl der `catch`-Blöcke reduziert und somit eine Duplikation von Code vermieden, da die Fehlerbehandlung nun in einem einzigen `catch`-Block erfolgen kann und nicht in mehreren `catch`-Blöcken bis auf die Übergabeparameter identisch implementiert werden muss. Das folgende **Beispiel** zeigt einen multi-catch-Block:

```
try
{
    // Quelltext, in dem Fehler auftreten können
}
catch (Exceptiontyp1 | Exceptiontyp2 ex)
{
    // Dieser catch-Block fängt sowohl Fehler der Klasse
    // Exceptiontyp1 als auch der Klasse Exceptiontyp2 ab
}
finally
{
    // Dinge, die stets gemacht werden müssen
}
```

Zu beachten ist, dass die Klassen `Exceptiontyp1` und `Exceptiontyp2` in unterschiedlichen Zweigen der Vererbungshierarchie der Klasse `Exception` (siehe Kapitel 13.4) liegen müssen und dass der Parameter des `catch`-Blockes vom Compiler implizit als `final` deklariert wird. Die Reihenfolge der Exception-Klassen in den runden Klammern ist hierbei irrelevant.

Liegen bei einem multi-catch-Block zwei Klassen im selben Vererbungszweig (z. B. weil die eine Klasse von der anderen Klasse abgeleitet ist), greift das Polymorphie-Konzept der Objektorientierung. Beim Kompilieren wird ein Fehler ausgegeben, da ein Objekt der verfeinerten Klasse vom Typ seiner Basisklasse ist und infolgedessen der Handler für Objekte der Basisklasse stets auch die Objekte der abgeleiteten Klassen abfangen würde.

Die richtige Reihenfolge der Handler im Falle mehrerer Handler bei einer Vererbungshierarchie der Exception-Typen wird in Kapitel 13.5.2 näher erläutert, da diese Reihenfolge bereits bei der konventionellen Ausnahmebehandlung (vor Java 7) relevant ist.

13.2.3 Werfen und Fangen von Exceptions

Wird während der Ausführung eines Programms im `try`-Block ein Ausnahmezustand erkannt, kann mit Hilfe von `throw` eine Exception "geworfen", also eine Ausnahme ausgelöst, und der gerade ausgeführte `try`-Block verlassen werden.

Das Auslösen einer Ausnahme bricht die Anweisungsfolge ab, die gerade ausgeführt wurde. Die Kontrolle wird an das Laufzeitsystem der Java-Virtuellen Maschine übergeben und das Laufzeitsystem sucht einen Handler für die Ausnahme in der Umgebung des `try`-Blocks. Im einfachsten Fall steht der **Exception Handler** direkt **in Form eines `catch`-Blockes** hinter dem `try`-Block.

Falls ein Handler gefunden wird, werden die Anweisungen des Handlers als nächstes ausgeführt und das Programm wird nach den Handlern fortgesetzt.

Es wird also nicht an die Stelle des Auslösens der Exception zurückgekehrt.

Falls überhaupt kein Handler vorhanden ist, wird das Programm von der Java-Virtuellen Maschine abgebrochen.

Ein Exception Handler hat das Ziel, eine geworfene Exception zu fangen und zu "entschärfen", d. h. eine Methode vom Ausnahmezustand in den Normalzustand zu überführen. Dies visualisiert das folgende Bild:

Bild 13-2 Entschärfen einer Exception durch einen Exception Handler

Der Einsatz eines Exception Handlers hat auch den Vorteil, dass man im Exception Handler noch Fehlerhinweise ausgeben kann.

Die Generierung eines Exception-Objektes und die Übergabe mit `throw` an die Java-Virtuelle Maschine werden als das **Auslösen** (**Werfen**) **einer Exception** bezeichnet. Das Exception-Objekt enthält Informationen über den aufgetretenen Fehler.

Der `finally`-Block ist – wie bereits erwähnt – optional, falls mindestens ein `catch`-Block existiert. Anweisungen in diesem Block werden auf jeden Fall ausgeführt, egal, ob eine Exception geworfen wurde oder nicht. Der Block kann also dazu verwendet werden, Aktionen auszuführen, die immer beim Verlassen des aktuellen `try`-Blockes erledigt werden müssen, ungeachtet dessen, ob eine Ausnahme aufgetreten ist oder nicht. So können z. B. Dateien geschlossen oder Ressourcen freigegeben werden.[174]

Prinzipiell gibt es für eine Methode, die nach dem Willen ihres Entwicklers von ihr ausgelöste Exceptions selbst abfangen soll, nur eine einzige Möglichkeit: Sie muss die Exception mit einem `try-catch`-Block abfangen.

Das folgende Bild zeigt den Ablauf einer Fehlerbehandlung unter Einschluss eines `finally`-Blockes:

Bild 13-3 Ablauf einer Fehlerbehandlung unter Einschluss eines `finally`-Blockes

13.2.4 Propagieren von Exceptions

Es ist nicht zwingend erforderlich, dass Exceptions direkt nach einem `try`-Block in einem `catch`-Block abgefangen werden. Sie können auch an den Aufrufer weitergeleitet werden. Dann braucht man natürlich keinen `try`-Block.

[174] Seit Java 7 gibt es das Konzept try-with-resources, welches in Kapitel 13.2.4 behandelt wird.

Bild 13-4 zeigt, wie eine nicht abgefangene Exception an den Aufrufer – bis hin zur Java-Virtuellen Maschine – propagiert wird:

Bild 13-4 Propagieren einer nicht abgefangenen Exception

Die aufgerufene Methode kann die in ihr aufgetretene Exception an den Aufrufer weiterleiten, wenn sie diese nicht erfolgreich behandeln kann. Man sagt dann, Exceptions werden **propagiert**. Die aufgerufene Methode kann aber auch im Rahmen der Behandlung der aufgetretenen Exception eine andere Exception erzeugen und diese an den Aufrufer weiterleiten.

Das Weiterleiten von Exceptions wird in Kapitel 13.5.5 behandelt.

Fängt eine Methode Checked Exceptions nicht selbst ab, sondern leitet sie an ihren Aufrufer weiter, so muss die geworfene **Exception** in der **Signatur der Methode** durch das Schlüsselwort **`throws`** **angegeben** werden. Ansonsten resultiert ein Kompilierfehler.

Das Schlüsselwort `throws` wird in Kapitel 13.5.5 behandelt.

13.2.5 Try-with-resources

Zusätzlich zu den bereits vorgestellten `try`-Anweisungen existiert seit Java 7 ein neues **try-with-resources**-Konzept.

Das try-with-resources-Konzept ermöglicht es, Objekte von Ressourcen, die für die Ein- und Ausgabe von Daten aller Art verantwortlich sind, einer automatisierten Überwachung und Kontrolle zu unterstellen.

Hierfür werden eine oder mehrere Ressourcen nach dem `try`-Statement in runden Klammern angelegt und initialisiert. Werden mehrere Ressourcen verwendet, so werden diese mit einem Strichpunkt (`;`) getrennt angegeben. Eine Ressource muss die seit Java 7 vorhandene Schnittstelle `AutoCloseable` implementieren. Beispiele für solche Ressourcen sind die Klassen[175] `FileReader`, `DatagramSocket`, `SSLSocket`, `PrintStream`, `URLClassLoader` und `XMLEncoder`. In den Versionen vor Java 7 musste stets darauf geachtet werden, dass Ressourcen dieser Art nach ihrer Verwendung oder im Falle eines ungewünschten Verhaltens in selbst kontrollierter Weise geschlossen wurden. Damit früher die Freigabe einer Ressource nach ihrer Verwendung sowie im Falle eines Fehlers bzw. ungewünschten Verhaltens sichergestellt werden konnte, musste in den früheren Versionen von Java hierfür der bereits vorgestellte `finally`-Block verwendet werden.

Der `finally`-Block kann bei der Verwendung des try-with-resources-Konzeptes entfallen, da das Schließen der Ressourcen implizit durch die Java-Virtuelle Maschine erledigt wird.

Das neue Konzept hat die folgende allgemeine Struktur:

```
try (Ressource1 r1 = new Ressource1();
     Ressource2 r2 = new Ressource2())
{
    // Programmabschnitt, der die angelegten Ressourcen nutzt
}

catch (Exceptiontyp1 | Exceptiontyp2 ex)
{
    // Behandlung der Ausnahmen, die im try-Block auftreten
    // können
}
```

Je nachdem, ob das try-with-resources-Konzept verwendet wird, oder ob die Ressource explizit im `finally`-Block geschlossen wird, können sich auftretende Exceptions auf unterschiedliche Art überlagern und sich als Folge davon gegenseitig unterdrücken. Abhängig von der Exception, die gefangen werden soll, haben beide Konzepte ihre Existenzberichtigung.

Die Überlagerung und Unterdrückung von Ausnahmen wird in Kapitel 13.5.5 behandelt.

Konkrete Beispiele zum try-with-resources-Konzept werden in Kapitel 19 gegeben.

13.3 Die Klasse `Throwable`

Bei der Ausnahmebehandlung kann der objektorientierte Ansatz konsequent eingesetzt werden. Eine Ausnahme wird in Java wie in C++ durch ein Objekt repräsentiert. Tritt eine Ausnahme ein, so wird das entsprechende Objekt erzeugt.

[175] Diese Klassen sind hier noch nicht bekannt.

In Java haben alle Exceptions eine gemeinsame Basisklasse. Dies ist die **Klasse `Throwable`** aus dem Paket `java.lang`.

Ausnahmen können mit Hilfe der Anweisung `throw` an beliebiger Stelle in einem `try`-Block ausgelöst oder "geworfen" werden. Die **`throw`-Anweisung** akzeptiert jede Referenz, die auf ein Objekt vom Typ `Throwable` zeigt. Damit können natürlich alle Objekte eines Subtyps der Klasse `Throwable` mit `throw` geworfen werden[176].

Die `throw`-Anweisung kann auch ohne `try`-Block in einer Methode stehen, wenn die Methode eine `throws`-Klausel deklariert. Dann muss natürlich der Aufruf der Methode in einem `try`-Block stehen.

Die `throws`-Klausel wird in Kapitel 13.5.5 behandelt.

Eine Ausnahme-Klasse unterscheidet sich nicht von einer "normalen" Klasse, außer dass sie von der Klasse `Throwable` abgeleitet ist. Die besondere Bedeutung erhält sie durch die Verwendung in `throw`-Anweisungen und in `catch`-Blöcken.

Haben Ausnahmen ganz bestimmte spezifische Eigenschaften, die im **Klassenbaum der Exceptions** noch nicht vertreten sind, so wird man eine spezielle Klasse vereinbaren, die von der Klasse `Exception` abgeleitet ist.

Da Exceptions nichts anderes als Klassen[177] sind, leitet man sich für seine Bedürfnisse einfach eine Klasse von der Klasse `Exception` oder einer ihrer Subklassen ab. Dadurch können die Exceptions unterschieden und modifizierte Fehlermeldungen angegeben werden. Bei Bedarf kann man auch Datenfelder und Methoden hinzufügen.

Das folgende Beispiel zeigt, wie man eine selbst definierte Exception generieren, auswerfen und wieder fangen kann:

```
// Datei: MyClass.java

class MyException extends Exception
{
   public MyException()
   {
      // Aufruf des Konstruktors der Klasse Exception.
      // Ihm wird ein String mit dem Fehlertext übergeben, der in
      // einem Datenfeld der Klasse Throwable gespeichert wird.
      super ("Fehler ist aufgetreten!");
```

[176] Ein Anwendungsprogrammierer wirft in der Regel ein Objekt vom Typ `Exception` oder eines Subtyps von `Exception`. Die Klasse `Exception` ist von der Klasse `Throwable` abgeleitet.

[177] Siehe Kapitel 13.4.

```
   }
}

public class MyClass
{
   public static void main (String[] args)
   {
      // Dieser try-Block ist untypisch, da in ihm nur eine
      // Exception zu Demonstrationszwecken geworfen wird.
      try
      {
         MyException ex = new MyException();
         throw ex;
         // Anweisungen unterhalb einer throw-Anweisung
         // werden nie abgearbeitet.
      }
      catch (MyException e)
      {
         System.out.println (e.getMessage());
      }
   }
}
```

Die Ausgabe des Programms ist:

```
Fehler ist aufgetreten!
```

In Programmen muss eine Fehlermeldung natürlich aussagekräftig sein. Eine Fehlermeldung sollte immer die Fehlerursache enthalten. Jede Exception ist eine Variable eines bestimmten Typs.

Zur Erstellung einer eigenen Exception wird im Beispiel die Klasse `MyException` von der Klasse `Exception` abgeleitet. Im parameterlosen Konstruktor der Klasse `MyException` wird mit `super()` der Konstruktor der Klasse `Exception` aufgerufen. An den Konstruktor der Klasse `Exception` wird ein String übergeben, der den Fehlertext enthält. Der Fehlertext beschreibt die Exception genauer und kann aus einer Exception mit der Methode `getMessage()`, die in der Klasse `Throwable` definiert ist, ausgelesen werden. Dieser Fehlertext kann dann im Fehlerfall ausgegeben werden.

Die Klasse `Throwable` – Vaterklasse der Klasse `Exception` – besitzt ein privates Datenfeld vom Typ `String`, um einen Fehlertext zu hinterlegen. Dieser Fehlertext kann eine Beschreibung des aufgetretenen Fehlers sein. Der Fehlertext kann gesetzt werden, indem er an den Konstruktor der Klasse `Exception` übergeben wird, der den Fehlertext an den Konstruktor der Klasse `Throwable` weitergibt, und damit den von der Klasse `Throwable` geerbten Anteil einer Exception initialisiert.

Der Empfänger einer geworfenen Exception, ein `catch`-Block, kann sich den Fehlertext mit Hilfe der Methode **`getMessage()`**, die von der Klasse `Throwable` geerbt wird, beschaffen.

Im `try`-Block in der `main()`-Methode der obigen Klasse `MyClass` wird durch `new` ein neues Objekt vom Typ `MyException` erzeugt. Dieses wird anschließend mit der Anweisung `throw` geworfen. Mit der `throw`-Anweisung wird der `try`-Block verlassen. Eine darauf folgende Codezeile wird nie erreicht. Die an die Java-Virtuelle Maschine übergebene Exception wird dann im `catch`-Block der `main()`-Methode gefangen und die übergebene Nachricht – also der Fehlertext – ausgegeben. Der Aufruf des `catch`-Blockes ist dabei durchaus mit dem Aufruf einer Methode zu vergleichen.

Aufgerufen wird ein `catch`-Block nicht vom Programm, sondern von der Java-Virtuellen Maschine. Ein Exception-Objekt wird an die Java-Virtuelle Maschine übergeben. Diese übernimmt die Kontrolle, sucht den passenden `catch`-Block und übergibt ihm die Exception.

13.4 Die Exception-Hierarchie

Wie bereits erwähnt, haben alle Exceptions eine gemeinsame Basisklasse, die Klasse `Throwable`. Diese selbst ist von der Klasse `Object` abgeleitet. Das folgende Bild zeigt die Exception-Hierarchie:

Bild 13-5 Ausschnitt der Klassenhierarchie von Throwable

`Throwable` ist die Basisklasse der beiden Klassenhierarchien `java.lang.Error` und `java.lang.Exception`. Spricht man von einer Exception, sind oft beide Hierarchien gemeint.

Die Klasse `Error`

Ausnahmen der Klasse `Error` sollten zur Laufzeit eines Java-Programms eigentlich gar nicht auftreten. Ein Programm sollte in der Regel nicht versuchen, einen solchen Fehler abzufangen.

Wenn eine solche Ausnahme auftritt, ist ein schwerwiegender Fehler in der Java-Virtuellen Maschine aufgetreten, der eigentlich gar nicht hätte auftreten sollen und der in der Regel auch nicht während der Laufzeit des Programms behandelbar ist, wie z. B. ein Fehler beim dynamischen Binden. Hier soll die Java-Virtuelle Maschine das Programm abbrechen. Es kann allerdings auch Fälle geben, wo es sinnvoll ist, selbst den Fehler zu behandeln. Hat beispielsweise ein Server-Rechner Probleme mit dem verfügbaren Speicher und generiert eine Exception vom Typ `OutOf-MemoryError`, so kann man in einem Exception Handler beispielsweise die Clients des Servers davon verständigen oder eventuell selbst genügend Speicher freigeben, damit die Ausnahme nicht mehr auftritt.

Die Klasse `Exception`

Normalerweise lösen Java-Programme Ausnahmen aus, die von der Klasse `Exception` abstammen.

Es handelt sich um Ausnahmen, die der Programmierer zur Laufzeit behandeln kann.

Die Klasse `RuntimeException`

Ausnahmen der Klasse `RuntimeException` oder eines Subtyps treten zur Laufzeit in der Java-Virtuellen Maschine auf. Dies sind aber keine "harten" Fehler der Java-Virtuellen Maschine, sondern Fehler im Programm wie z. B. die Anwendung des Punkt-Operators auf eine `null`-Referenz.

Dies kann passieren, wenn bei einem Methodenaufruf `a.f()` die Referenz `a` noch kein Objekt referenziert, sondern eine `null`-Referenz darstellt. In diesem Fall wird eine Exception vom Typ `NullPointerException` geworfen. Eine Exception vom Typ `NullPointerException` ist vom Typ der Klasse `RuntimeException` und kann im Prinzip bei jedem Zugriff auf ein Datenfeld oder eine Methode eines Objektes auftreten.

Es wäre überhaupt nicht praktikabel, die fehlerhafte Anwendung des Punkt-Operators auf die `null`-Referenz in der Anwendung zu behandeln, da es einfach zu viele Stellen im Programm gibt, an denen ein solcher Fehler auftreten kann.

Daher wurde bei der Definition von Java entschieden, dass Ausnahmen vom Typ `RuntimeException` von der Java-Virtuellen Maschine behandelt werden müssen. Der Programmierer hat aber die Möglichkeit, solche Ausnahmen zu behandeln. Da

es sich um **Unchecked Exceptions** (siehe Kapitel 13.4.1) handelt, wird eine eventuell fehlende Ausnahmebehandlung vom Compiler aber nicht überprüft.

13.4.1 Checked und Unchecked Exceptions

Man spricht von **Checked Exceptions**, falls eine Exception vom Programmierer behandelt werden muss, und dies auch vom Compiler überprüft (checked) wird.

Wird eine Checked Exception vom Programmierer nicht abgefangen, ist das Programm nicht kompilierbar.

Man spricht von **Unchecked Exceptions**, falls eine geworfene Exception vom Programmierer weder abgefangen, noch in der **`throws`-Klausel** der Schnittstelle der werfenden Methode angegeben werden muss. Auf Unchecked Exceptions wird ein Programm **vom Compiler nicht überprüft**.

Alle Exceptions bis auf diejenigen der Unterbäume `RuntimeException` und `Error` sind Checked Exceptions d. h. zu berücksichtigende Ausnahmen. Dies visualisiert die folgende Tabelle:

Unchecked Exceptions	Checked Exceptions
Klassen `RuntimeException` und `Error` sowie davon abgeleitete Klassen	alle anderen

Tabelle 13-1 Checked und Unchecked Exceptions

Checked Exceptions müssen vom Programmierer entweder in einem Exception Handler einer Methode behandelt werden oder aber in der `throws`-Klausel der Methode, welche die Exception wirft, angegeben werden, um anzuzeigen, dass sie die entsprechende Exception nach außen weitergibt.

Das Konzept in Java, zwischen sogenannten **Checked Exceptions** und **Unchecked Exceptions** zu unterscheiden, hat das Ziel, dass vom Compiler geprüft wird, ob der Programmierer alle zu berücksichtigenden Ausnahmen (Checked Exceptions) tatsächlich behandelt hat. Ist dies nicht der Fall, so wird das Programm vom Compiler nicht übersetzt und der Programmierer muss nachbessern, indem er die nicht behandelten Checked Exceptions auch einer Fehlerbehandlung unterzieht. Es wird also schon zur **Übersetzungszeit** und nicht erst zur Laufzeit durch den Compiler überprüft, ob das Programm für jede Checked Exception die verlangte Fehlerbehandlung durchführt.

13.4.2 Beispiele für Exceptions

Im Folgenden einige Exceptions, die von der Klasse `Error` abgeleitet sind:

Exception	Ursache
`VirtualMachineError`	Die virtuelle Maschine ist defekt oder hat keine Ressourcen
`AnnotationFormatError`	Annotationen im Bytecode sind fehlerhaft
`AssertionError`	Eine Aussage wird als falsch bewertet
`IOError`	Eingabe- oder Ausgabemechanismen weisen ernsthafte Fehler auf

Tabelle 13-2 Beispiele für Exceptions vom Typ `Error`

Einige Exceptions, die von der Klasse `Exception` abgeleitet sind:

Exception	Ursache
`CertificateException`	Probleme mit Zertifikaten sind aufgetreten
`CloneNotSupportedException`	Ein Objekt sollte kopiert werden, welches das Cloning aber nicht unterstützt
`DataFormatException`	Ein Datenformat ist fehlerhaft

Tabelle 13-3 Beispiele für Exceptions vom Typ `Exception`

Einige Exceptions, die von der Klasse `RuntimeException` abgeleitet sind:

Exception	Ursache
`ArithmeticException`	Ein Integer-Wert wurde durch Null dividiert
`IndexOutOfBoundsException`	Auf ein Feld mit ungültigem Index wurde zugegriffen
`ClassCastException`	Cast wegen fehlender Typverträglichkeit nicht möglich
`NullPointerException`	Versuchter Zugriff auf ein Datenfeld oder eine Methode über die `null`-Referenz

Tabelle 13-4 Beispiele für Exceptions vom Typ `RuntimeException`

Werden zusätzliche Pakete benutzt, so können weitere Exceptions hinzukommen. Im Paket `java.io` werden z. B. Objekte vom Typ `IOException` benutzt, um Fehler bei der Ein- und Ausgabe anzuzeigen.

13.5 Behandlung von Ausnahmen

Ein `try`-Block kennzeichnet eine Anweisungsfolge, innerhalb derer Exceptions ausgelöst werden können. Vorgänge, die Exceptions auslösen können und behandelt werden sollen, müssen grundsätzlich in einem `try`-Block stehen.

Wird eine Exception nicht unmittelbar in einer Methode behandelt, in der sie auftreten kann, so wird sie an den Aufrufer der Methode weitergereicht. Der Aufruf dieser Methode muss dann in einem `try`-Block stehen.

Der `try`-Block bedeutet:

Es wird versucht, den Code in den geschweiften Klammern auszuführen. Wenn Exceptions geworfen werden, hat sich der Programmierer im Falle von Checked Exceptions um die Behandlung zu kümmern.

Eine Exception kann in einem `catch`-Block, der dem `try`-Block folgt, behandelt oder an die aufrufende Methode zur Behandlung weitergereicht werden.

Unmittelbar hinter dem `try`-Block können ein oder mehrere Exception Handler in Form von `catch`-Blöcken folgen. Ein `catch`-Block besteht aus dem Schlüsselwort `catch`, gefolgt von mindestens einer Typspezifizierung der zu behandelnden Ausnahme und einem formalen Parameter in runden Klammern. Mehrere Typspezifizierungen in einem multi-catch-Block werden mit einer binären Oder-Verknüpfung (`|`) getrennt (Kapitel 13.2.2 zeigt die genaue Syntax). Anschließend folgt der Codeblock zur Behandlung der aufgetretenen Ausnahme bzw. Ausnahmen. Behandelnde Maßnahmen können sein:

- Eine Fehlermeldung ausgeben und für den Fehlerfall vorgesehene Default-Werte setzen, damit weitergearbeitet werden kann.
- Eine Fehlermeldung ausgeben gefolgt von einem Programmabbruch durch Aufruf der Methode `System.exit(1)`[178].

Existieren mehrere Handler, dann müssen diese unmittelbar aufeinanderfolgen.

Normaler Code zwischen den Handlern ist nicht erlaubt!

Existiert kein Exception Handler in einer Methode, kann die weitergereichte Exception in der aufrufenden Methode oder deren Aufrufer usw. gefangen werden.

Hat der Programmierer jedoch **keinen Exception Handler für eine weitergereichte Checked Exception** geschrieben, dann meldet sich bereits der **Compiler** mit einer **Fehlermeldung**. Damit erzwingt der Compiler, dass eine Checked Exception vom Programmierer behandelt wird.

[178] Die Klasse `System` ist im Anhang C beschrieben.

Wird eine **weitergereichte Unchecked Exception** vom Programmierer **nicht abgefangen**, meldet sich das **Laufzeitsystem** beim Auftreten der Unchecked Exception mit einer **Fehlermeldung** und bricht das Programm ab.

Die Syntax des Exception Handling erinnert zum einen an die `switch`-Anweisung, zum anderen an Methodenaufrufe. Beide Vergleiche haben ihre Berechtigung:

- Wie bei einem `switch` werden verschiedene Codestücke, hier einer der Exception Handler (oder auch keiner), angesprungen. Im Unterschied zu `switch` sind jedoch keine `break`-Anweisungen zwischen den Handlern nötig, und wenn im `try`-Block keine Exception auftritt, werden alle dem `try`-Block folgenden Handler übersprungen!
- Die Schnittstelle eines Handlers und die Deklaration einer einfachen **Methode** haben strukturelle Ähnlichkeiten. Beide Konstrukte besitzen eine Parameterliste und einen Rumpf. Die Schnittstelle des Handlers besitzt jedoch keine Modifikatoren und keinen Rückgabewert und ermöglicht es, innerhalb der Parameterliste einen formalen Parameter mit einer oder mehreren Spezifizierungen des Exceptiontyps zu definieren.

13.5.1 Beispiel für das Fangen einer Exception

Am folgenden Beispiel wird das Fangen einer Exception der Klasse **`ArrayIndexOutOfBoundsException`**, einer Subklasse der Klasse `RuntimeException`, demonstriert. Eine Exception vom Typ `ArrayIndexOutOfBoundsException` wird geworfen, wenn die Bereichsgrenzen eines Arrays überschritten werden. Eine Exception der Klasse `RuntimeException` oder eines Subtyps gehört zu den **Unchecked Exceptions und muss nicht – aber kann – vom Programmierer abgefangen werden**. Hier das bereits erwähnte Beispiel:

```
// Datei: Test.java

public class Test
{
   public static void main (String[] args)
   {
      int[] intarr = new int [4];
      for (int lv = 0; lv < 8; lv++)
      {
         try
         {
            intarr [lv] = lv;
            System.out.println (intarr [lv]);
         }
         catch (ArrayIndexOutOfBoundsException e)
         {
            System.out.println ("Array-Index " + lv +
                                " ist zu gross!");
         }
      }
   }
}
```

Die Ausgabe des Programms ist:

```
0
1
2
3
Array-Index 4 ist zu gross!
Array-Index 5 ist zu gross!
Array-Index 6 ist zu gross!
Array-Index 7 ist zu gross!
```

Dabei ist zu beachten, dass das Programm nach dem Behandeln jeder Exception ganz normal mit der Abarbeitung der `for`-Schleife fortfährt.

13.5.2 Reihenfolge der Handler

Die Suche nach dem passenden Handler erfolgt von oben nach unten, d. h. **die Reihenfolge der Handler ist relevant**. Der Handler für eine Exception, die im Klassenbaum der Exceptions am weitesten oben steht, muss an letzter Stelle stehen. Dies ist darauf zurückzuführen, dass überall da, wo ein Objekt einer Basisklasse erwartet wird, stets auch ein Objekt einer Unterklasse verwendet werden kann.

Ein Handler für Exceptions einer Klasse `A` passt infolge des Polymorphie-Konzeptes der Objektorientierung auch auf Exceptions aller von `A` abgeleiteten Klassen.

Würde ein Handler mit einem Parameter vom Typ der Basisklasse also ganz vorne in der Liste der Handler stehen, so würde er jede Exception des entsprechenden Unterbaums des Typs des Parameters abfangen und die für die Unterklassen spezialisierten Handler würden überhaupt nie aufgerufen werden. Also ist eine umgekehrte Anordnung der Handler erforderlich.

Zuerst müssen die **Handler für** die **spezialisiertesten Klassen** der **Exception-Hierarchie** aufgelistet werden und **dann in** der **Reihenfolge der zunehmenden Generalisierung** die **entsprechenden allgemeinen Handler**.

Hat man also eine Klassenhierarchie für Exceptions definiert, dann muss sich diese Hierarchie in den Handlern widerspiegeln – allerdings in umgekehrter Reihenfolge.

Fügt man am Ende der Folge der Handler noch einen Handler für die Basisklasse ein, ist man auch in Zukunft sicher, dass alle Exceptions behandelt werden, auch wenn jemand neue Exceptions ableitet.

Die richtige Anordnung der Handler wird vom Compiler überprüft. Der Compiler prüft, ob alle Handler erreichbar sind. Im folgenden Beispielprogramm wird ein Kompilierfehler durch einen nicht erreichbaren `catch`-Block demonstriert:

```
// Datei: Catchtest.java
class MyException extends Exception
{
   public MyException()
   {
      super ("Fehler ist aufgetreten!");
   }
}

public class Catchtest
{
   public void testMethode()
   {
      try
      {
         throw new MyException();
      }
      catch (Exception e)
      {
         System.out.println (e.getMessage());
      }
      catch (MyException e)
      {
         System.out.println (e.getMessage());
      }
   }

   public static void main (String[] args)
   {
      Catchtest x = new Catchtest();
      x.testMethode();
   }
}
```

Die Ausgabe des Compilers ist:

```
Catchtest.java:25: catch not reached.
      catch (MyException e)
      ^
1 error
```

13.5.3 Ausnahmen weiterreichen

Eine Exception wird bearbeitet, sobald ein Handler zu ihrer Bearbeitung gefunden und aufgerufen wurde.

Stellt sich innerhalb des Handlers (z. B. anhand der in der Exception übergebenen Informationen oder weil Korrekturmaßnahmen fehlschlagen) heraus, dass dieser Handler die ihm übergebene Exception nicht behandeln kann, so kann **dieselbe Exception erneut** im `catch`-Block mit `throw` **ausgeworfen** werden. Der Handler kann aber gegebenenfalls auch **andere Exceptions auswerfen**.

Im Folgenden ein Ausschnitt eines Programms, der das erneute Auswerfen einer Exception zeigt:

```
try
{
   AException aEx = new AException ("schwerer Fehler");
   throw aEx;
}
catch (AException e)
{
   String message = e.getMessage();
   if (message.equals ("schwerer Fehler"))
      throw e;
}
```

Im gezeigten Beispiel wird nur eine Exception des Typs `AException` aus dem `try`-Block gefangen und durch den `catch`-Block weitergereicht. Ist im `try`-Block das Werfen mehrerer Typen von Exceptions möglich, so können diese Typen von Exceptions alternativ durch mehrere `catch`-Blöcke oder durch einen multi-catch-Block (siehe Kapitel 13.2.2) behandelt werden. Das folgende Beispiel zeigt ein Beispiel für die Variante mit mehreren `catch`-Blöcken:

```
try
{
   if(bedingung)
   {
      AException aEx = new AException ("schwerer Fehler");
      throw aEx;
   }
   else
   {
      throw new BException()
   }
}
catch (AException e)     // catch-Block
{
   throw e;
}
catch (BException e)     // weiterer catch-Block
{
   throw e;
}
```

Das erneute Werfen einer Exception, die aber verschiedene Typen annehmen kann, kann auch mit einem einzigen Handler erfolgen. Da sowohl die Klasse `AException` als auch die Klasse `BException` von der Klasse `Exception` abgeleitet wird, kann auch ein einziger Handler geschrieben werden, der alle Exceptions vom Typ `Exception` oder eines Subtyps behandelt. Dies zeigt das folgende Beispiel, das eine Alternative zur obigen Lösung darstellt:

```
try
{
    // wirft je nach Bedingung eine AException oder BException
}
```

```
catch(Exception e)
{
    // In diesem Block werden alle Ausnahmetypen, die von der Klasse
    // Exception abgeleitet sind, protokolliert und weitergeworfen
    System.out.println("Es ist eine Ausnahme aufgetreten!");
    // weiterreichen
    throw e;
}
```

Bis zu der Version Java 6 konnte in diesem Beispiel erst zur Laufzeit festgestellt werden, ob eine Ausnahme vom Typ `AException` oder `BException` ist. Zur Kompilierzeit konnte der Typ der Exception nur durch den aktuellen Parameter des `catch`-Blocks bestimmt werden, im Beispiel der Typ `Exception`. Es musste somit im weiteren Verlauf mit einer Ausnahme vom Typ `Exception` gearbeitet werden, obwohl vielleicht eine Exception vom Typ `AException` geworfen wurde, wie im folgenden Beispiel der Ankündigung der Ausnahme zu sehen ist. Eine präzise Unterscheidung der Typen war zur Kompilierzeit nicht möglich. Das folgende Beispiel zeigt, wie bis zu der Version Java 6 ein Fangen und Weiterreichen verschiedener Typen von Ausnahmen über eine gemeinsame Basisklasse der Exceptions in einem einzigen `catch`-Block realisiert werden musste:

```
public werfeAusnahme(boolean bedingung) throws Exception
{
   try
   {
      if(bedingung)
      {
         AException aEx = new AException ("schwerer Fehler");
         throw aEx;
      }
      else
      {
         throw new BException()
      }
   }
   catch(Exception e)
   {
         System.out.println("Es ist eine Ausnahme aufgetreten!");
         throw e;
   }
}
```

Die `throws`-Klausel[179] kündigt einem Aufrufer der Methode `werfeAusnahme()` an, dass eine Exception vom Typ `Exception` geworfen wird. Der Aufrufer wird also aufgefordert, diese Exception zu behandeln oder sie erneut weiterzureichen. Wie aber das Beispiel zeigt, werden genau genommen Spezialisierungen des Typs `Exception` – nämlich `AException` und `BException` – geworfen. Dem Aufrufer bleibt bis einschließlich der Version Java 6 diese Information zur Kompilierzeit verborgen, der tatsächliche Typ kann in solchen Fällen erst zur Laufzeit ermittelt werden.

[179] Auf die `throws`-Klausel wird an dieser Stelle vorgegriffen. Eine genaue Erörterung dieses Konstrukts folgt im Kapitel 13.5.5.

Verbesserte Analyse seit Java 7

Seit der Version Java 7 wird bereits zur Kompilierzeit der Typ der Ausnahme durch eine Betrachtung des `try`-Blocks festgestellt.

Im vorangegangenen Beispiel sind nur Ausnahmen vom Typ `AException` oder `BException` möglich. Es kann seit der Version Java 7 im weiteren Programmverlauf mit einer der beiden Ausnahmen gerechnet und gearbeitet werden, wenn man die jeweils geworfene Ausnahme weiterreicht. Die Information über die Typen der Spezialisierungen ist seit der Version Java 7 also bereits dem Compiler bekannt, obwohl die im `catch`-Block weitergereichte Ausnahme vom Typ `Exception` ist. Das obige Beispiel kann seit Java 7 folgendermaßen formuliert werden:

```
public werfeAusnahme(boolean bedingung) throws AException,
BException
{
   try
   {
      if(bedingung)
      {
         AException aEx = new AException ("schwerer Fehler");
         throw aEx;
      }
      else
      {
         throw new BException()
      }
   }
   catch(Exception e)
   {
         System.out.println("Es ist eine Ausnahme aufgetreten!");
         throw e;
   }
}
```

Durch die verbesserte Analyse eines `try`-Blocks durch den Compiler können seit Java 7 die möglichen Typen der weitergereichten Ausnahmen eines `catch`-Blocks bereits zur Kompilierzeit ermittelt werden. Dies gilt allerdings nur dann, wenn die dem `catch`-Block übergebene Referenz auf das Objekt der Ausnahme innerhalb des `catch`-Blocks nicht verändert wird.

13.5.4 Schichtenstruktur für das Exception Handling

Jede Gruppe von Handlern ist nur für die Behandlung von Exceptions aus ihrem zugeordneten `try`-Block verantwortlich.

Alle innerhalb von Handlern ausgeworfenen Exceptions werden nach außen an die nächste umschließende `try`-Anweisung weitergereicht. Die **`try`-Anweisungen** können also **geschachtelt** werden.

Dieser Mechanismus gestattet die Implementierung von mehreren Schichten zur Fehlerbehandlung. Ebenfalls nach außen weitergereicht werden Exceptions, für die kein Handler existiert. Das folgende Beispiel zeigt geschachtelte `try`-Anweisungen:

```
// Datei: Versuch.java

class MyException2 extends Exception
{
   public MyException2()
   {
      super ("Fehler ist aufgetreten!");
   }
}

public class Versuch
{
   public static void main (String[] args)
   {
      try
      {
         try
         {
            throw new Exception();
         }
         catch (MyException2 e)
         {
            System.out.println ("MyException2 gefangen");
         }
      }
      catch (Exception e2)
      {
         System.out.println ("Exception gefangen");
      }
   }
}
```

Die Ausgabe des Programms ist:

```
Exception gefangen
```

Das folgende Bild zeigt die Anordnung der `try`-Anweisungen aus dem Beispielprogramm der Klasse `Versuch`:

Bild 13-6 Geschachtelte `try`-Anweisungen

13.5.5 Ausnahmen ankündigen – die `throws`-Klausel

In Java wird zwingend verlangt, bestimmte Exceptions, die von einer Methode ausgelöst werden können, in die Deklaration der Methode mit Hilfe der `throws`-Klausel aufzunehmen. Dabei müssen **Checked Exceptions unbedingt angegeben** werden, während das bei Unchecked Exceptions nicht erforderlich ist. Dadurch wird dem Aufrufer signalisiert, welche Ausnahmen von einer Methode ausgelöst bzw. weitergereicht werden. Dies spielt auch eine Rolle bei Bibliotheken. Ein Programmierer, der Bibliotheken nutzt, muss wissen, welche Exceptions die Bibliotheksmethoden werfen können. Seine Aufgabe ist es, die geworfenen Exceptions sinnvoll zu behandeln.

Eine Methode kann nur die Checked Exceptions auslösen, die sie in der `throws`-Klausel angegeben hat. Unchecked Exceptions hingegen kann sie immer werfen.

Soll also eine Exception erst außerhalb einer Methode verarbeitet werden, muss die Methodendeklaration wie folgt erweitert werden:

```
[Zugriffsmodifikatoren] Rückgabewert Methodenname
([Parameter]) throws Exceptionname1 [,Exceptionname2, . . . .]
```

Beachten Sie, dass `throws Exceptionname1 [,Exceptionname2, . . . .]` die sogenannte `throws`-Klausel darstellt. Die Methode gibt also eine oder mehrere Exceptions nach außen weiter.

Durch die `throws`-Klausel informiert eine Methode den Aufrufer (und den Compiler) über eine mögliche abnormale Rückkehr aus der Methode.

Diese zusätzliche Information bei der Deklaration dient nicht der Unterscheidung von Methoden im Sinne einer Überladung!

Eine Exception kann also in der aufrufenden Methode, eventuell sogar erst in der `main()`-Methode oder überhaupt nicht vom Anwendungsprogramm, gefangen werden. Die Methode `pruefeDatum()` im nächsten Beispiel behandelt die Exception `ParseException` nicht selbst und besitzt deshalb eine `throws`-Klausel. Die Exception wird in diesem Beispiel in der aufrufenden Methode – hier in der `main()`-Methode – behandelt.

Jetzt das bereits angekündigte **Beispiel**:

```
// Datei: DatumEingabe.java
import java.util.Date;
import java.text.*;

public class DatumEingabe
{
   public Date pruefeDatum (String datum) throws ParseException
   {
      // Eine auf die Rechnerlokation abgestimmte Instanz der Klasse
      // DateFormat wird erzeugt.
      DateFormat df = DateFormat.getDateInstance();

      // strenge Datumsprüfung einschalten
      df.setLenient (false);

      // Datum überprüfen und in ein Date-Objekt wandeln.
      // Die Methode parse() wirft eine ParseException, wenn in
      // datum kein gültiges Datum steht.
      Date d = df.parse (datum);
      return d;
   }

   public static void main (String[] args)
   {
      DatumEingabe v = new DatumEingabe();
      String[] testdaten = {"13.06.2013", "13.13.2013"};

      Date datum = null;
      for (int i = 0; i < testdaten.length; i++)
      {
         try
         {
            datum = v.pruefeDatum (testdaten [i]);
            System.out.println ("Eingegebenes Datum ist ok:\n"
                                + datum);
         }
         catch (ParseException e)
         {
            System.out.println ("Eingegebenes Datum ist nicht ok:\n"
                                 + testdaten [i]);
         }
      }
   }
}
```

Die Ausgabe des Programms ist:

```
Eingegebenes Datum ist ok:
Thu Jun 13 00:00:00 CEST 2013
Eingegebenes Datum ist nicht ok:
13.13.2013
```

Für das Überschreiben von Methoden gibt es die folgende Einschränkung: Wird eine Methode einer Basisklasse, die keine Exceptions mit `throws` weiterreicht, bei einer Ableitung überschrieben, so kann die überschreibende Methode auch keine Exceptions weiterreichen, da sonst der Vertrag der Basisklasse gebrochen würde. Die Fehlerbehandlung muss dann in der überschreibenden Methode selbst erfolgen. Verstöße gegen diese Vorschrift verursachen eine ganze Reihe von Fehlern beim Kompilieren.

13.5.6 Überlagerung und Unterdrückung von Ausnahmen

Bei der **Verwendung der `finally`-Klausel** kann es vorkommen, dass sich Ausnahmen überlagern. Hierzu soll die folgende Klasse betrachtet werden, welche die Schnittstelle `AutoCloseable` implementiert:

```
// Datei: MeineKlasse.java

public class MeineKlasse implements AutoCloseable
{
   // meineMethode() soll eine Ausnahme vom Typ Exception auslösen
   // und diese direkt an den Aufrufer reichen. Deshalb wird hier
   // kein try verwendet.
   public void meineMethode() throws Exception
   {
      throw new Exception ("Ausnahme in meineMethode");
   }

   // Die Schnittstelle AutoCloseable schreibt die Implementierung
   // der Methode public void close() throws Exception vor.
   public void close() throws Exception
   {
      throw new Exception ("Ausnahme in close");
   }
}
```

Bei der Verwendung von Objekten dieser Klasse werden Ausnahmen sowohl in der Methode `meineMethode()` als auch in der Methode `close()` geworfen. Werden die Methoden eines Objekts der Klasse `MeineKlasse`, das durch die Referenz `meineRef` referenziert wird, wie folgt aufgerufen:

```
try
{
   meineRef.meineMethode();
}
```

```
finally
{
   meineRef.close();
}
```

so wird in diesem Beispiel innerhalb des `try`-Blockes eine Ausnahme geworfen werden. Der `finally`-Block wird in jedem Fall – also auch wenn eine Ausnahme im `try`-Block auftritt – ausgeführt. Bei diesem Aufruf der Methode `close()` soll in diesem Beispiel eine weitere Ausnahme geworfen werden. Die beiden Ausnahmen überlagern sich also. **Dabei wird die zuerst geworfene Ausnahme innerhalb des `try`-Blockes durch die Ausnahme innerhalb des `finally`-Blockes unterdrückt.**

Eine ähnliche gelagerte Situation kann auch bei der Verwendung des **try-with-resources-Konzeptes**[180] auftreten. Der Aufruf

```
try (MeineKlasse meineRef = new MeineKlasse())
{
   meineRef.meineMethode();
}
```

soll in diesem Beispiel eine Ausnahme innerhalb des `try`-Blockes erzeugen. Beim Verlassen des `try`-Blockes wird implizit durch die try-with-resources-Anweisung die Methode `close()` aufgerufen und es soll eine weitere Ausnahme erzeugt werden. Auch hier überlagern sich die aufgetretenen Ausnahmen. **Allerdings wird in diesem Beispiel die Ausnahme der Methode `close()`, die implizit durch die try-with-resources-Anweisung aufgerufen wird, unterdrückt.** Der Aufrufer muss die Ausnahme, die innerhalb des `try`-Blocks geworfen wurde, behandeln. Im gezeigten Beispiel muss also die Ausnahme der Methode `meineMethode()` behandelt werden.

Während bei der Verwendung eines `finally`-Blockes die Ausnahme, die im `finally`-Block aufgetreten ist, eine Ausnahme des `try`-Blockes unterdrückt, unterdrückt bei der Verwendung von try-with-resources die Ausnahme, die innerhalb des `try`-Blockes auftritt, eine Ausnahme, die beim impliziten Aufruf der Methode `close()` auftritt.

Will man die unterdrückten Ausnahmen eines try-with-resources-Blockes behandeln, bietet die Klasse `Throwable` die Methode `getSuppressed()` an. Diese Methode liefert ein Array vom Typ `Throwable`, das alle unterdrückten Exceptions beinhaltet, zurück.

13.5.7 Nachverfolgung von Ausnahmen

Im Fehlerfall ist das Finden einer Fehlerursache und das Lokalisieren eines Fehlers unvermeidlich. In der Regel will man wissen, wie die Aufrufreihenfolge der Methoden in einer virtuellen Maschine war, die letztendlich zum Auftreten eines Fehlers in der zuletzt aufgerufenen Methode geführt hat. Java bietet hierfür innerhalb der Klasse

[180] Siehe Kapitel 13.2.5

`Throwable` – und damit in allen davon abgeleiteten Unterklassen – die Methode `printStackTrace()` an. Diese Methode ermöglicht einen Einblick in die Fehlerursache, indem sie die Klasse der Ausnahme samt Fehlertext, sowie den sogenannten Stack-Trace, der die Aufrufreihenfolge der Methoden zeigt, mitsamt deren Klassen auflistet.

Das folgende Beispiel soll dies zeigen. Dabei wird dem Konstruktor der Klasse `FileInputStream` als Dateiname absichtlich `null` übergeben, was bei der Instanziierung zu einer `NullPointerException` führt:

```
// Datei: StackTraceExample.java

import java.io.FileInputStream;

public class StackTraceExample
{
   public static void main(String[] args)
   {
      try
      {
         String fileName = null;
         // Daten einlesen
         FileInputStream inStream = new FileInputStream (fileName);
      }
      catch (Exception ex)
      {
         ex.printStackTrace();
      }
   }
}
```

Die Ausgabe des Programms ist:

```
java.lang.NullPointerException
   at java.io.FileInputStream.<init>(
      FileInputStream.java:134)
   at java.io.FileInputStream.<init>(
      FileInputStream.java:97)
   at StackTraceExample.main(
      StackTraceExample.java:8)
```

Aus dem Stack-Trace kann man nun die Aufrufhierarchie erkennen. Dabei ist der Stack-Trace von unten nach oben zu lesen. Zuerst wurde in Zeile 8 durch die Instanziierung mit `new` der Konstruktor der Klasse `FileInputStream` aufgerufen. In Zeile 97 des Konstruktors (`<init>`) wurde nun über `this` ein überladener Konstruktor (`<init>`) der Klasse `FileInputStream` aufgerufen. Dieser Konstruktor konnte dann bis Zeile 134 abgearbeitet werden, bis, wie in diesem Fall, explizit eine Exception des Typs `NullPointerException` ausgelöst wurde.

Werden nun beim Kompilieren der Datei `StackTraceExample.java` die Debug-Informationen mit `-g:none` ausgeschaltet, so würde die Ausgabe wie folgt aussehen:

Die Ausgabe des Programms ist:

```
java.lang.NullPointerException
   at java.io.FileInputStream.<init>(
      FileInputStream.java:134)
   at java.io.FileInputStream.<init>(
      FileInputStream.java:97)
   at StackTraceExample.main(Unknown Source)
```

Da nur die Datei `StackTraceExample.java` ohne Debug-Informationen kompiliert wurde, werden die ersten Zeilen des Stack-Trace mit Dateiname und Zeilennummer ausgegeben und erst in der letzten Zeile ist nur noch die Information vorhanden, dass der Aufruf des Konstruktors der Klasse `FileInputStream` in der Methode `main()` der Klasse `StackTraceExample` stattgefunden hat. Dateiname und Zeilennummer fehlen.

13.6 Vorteile des Exception-Konzeptes

Vorteile des Exception Handling sind:

- Eine saubere Trennung des Codes in "normalen" Code und in Fehlerbehandlungscode.
- Der Compiler prüft, ob Checked Exceptions vom Programmierer abgefangen werden. Damit werden **Nachlässigkeiten** beim Programmieren **bereits zur Kompilierzeit** und nicht erst zur Laufzeit **entdeckt**.
- Das Propagieren einer Exception erlaubt, diese auch in einem umfassenden Block oder einer aufrufenden Methode zu behandeln.
- Da Exception-Klassen in einem Klassenbaum angeordnet sind, können – **je nach Bedarf – spezialisierte Handler oder generalisierte Handler geschrieben** werden.

13.7 Assertions

Exceptions werden von einem operationell eingesetzten Programm zur **Laufzeit** geworfen. **Assertions** hingegen werden zum **Testen** von Programmen verwendet. Sie überprüfen beim Testen das Vorliegen bestimmter **Bedingungen**.

Das englische Wort Assertion steht für Aussage oder Behauptung.

Assertions sind beim Testen eingeschaltet. Sie müssen im operationellen Betrieb eines Programms abgeschaltet werden.

Mit Assertions kann beispielsweise überprüft werden, ob

- ein berechneter Wert innerhalb eines bestimmten Wertebereichs liegt,
- eine Variable nur bestimmte Werte annimmt oder
- ein bestimmter Zweig im Kontrollfluss nie durchlaufen wird.

Dafür werden in Kapitel 13.7.3 noch Programmbeispiele angegeben.

Mit dem Konzept der Assertions können auch die Ursachen von aufgetretenen Exceptions untersucht und beseitigt werden.

Nach erfolgreichem Abschluss der Entwicklung verbleiben Assertions im Programmcode, werden aber nicht mehr aktiviert und erfordern deshalb keine zusätzlichen Ressourcen.

Assertions sind also **kein Bestandteil der Programmfunktionalität** und sollten auch nicht mit dieser verwoben werden.

Vor- und Nachbedingungen von exportierten Methoden und Invarianten von Klassen sollten nicht mit Assertions realisiert werden, da diese Bestandteile der Verträge von Klassen sind und zur Laufzeit nicht abgeschaltet werden können sollten.

Assertions dürfen auch keine Seiteneffekte haben, da diese nicht mehr eintreten, wenn die Assertions abgeschaltet sind.

13.7.1 Notation von Assertions

Assertions werden umgesetzt, indem zur Laufzeit des Programms **boolesche Ausdrücke** ausgewertet werden. Die Syntax in Java für Assertions kennt zwei Ausprägungen:

```
assert Ausdruck1;              /* erste Variante  */
```

oder

```
assert Ausdruck1 : Ausdruck2; /* zweite Variante */
```

In der **ersten Variante** der **`assert`-Anweisung** wird der boolesche Ausdruck `Ausdruck1` ausgewertet. Ergibt der Ausdruck `Ausdruck1` den Wert `true`, d. h. die zu überprüfende Bedingung ist richtig, so wird die Ausführung eines Programms mit der nächsten Anweisung nach der Assertion normal fortgesetzt. Ergibt der boolesche Ausdruck `Ausdruck1` jedoch den Wert `false`, dann wirft die `assert`-Anweisung eine Exception vom Typ `AssertionError`.

Bei der **zweiten Variante der `assert`-Anweisung** wird ebenfalls der boolesche Ausdruck `Ausdruck1` ausgewertet. Ergibt die Auswertung `true`, so wird die Ausführung wiederum nach der Assertion fortgesetzt. Ist der Ausdruck `Ausdruck1` allerdings `false`, dann wird der Ausdruck `Ausdruck2` ausgewertet. Der Rückgabewert des Ausdrucks `Ausdruck2` wird dem Konstruktor des Objektes der

Klasse `AssertionError` übergeben, um die fehlgeschlagene Assertion genauer zu beschreiben und den Entwickler bei der Fehlersuche zu unterstützen.

Die Klasse `AssertionError` ist von der Klasse `Error` abgeleitet. Da Objekte der Klasse `Error` oder einer ihrer Subklassen – wie zuvor erwähnt – in der Regel nicht vom Programmierer behandelt werden können, wird die Ausführung des Programms abgebrochen.

13.7.2 Aktivierung und Deaktivierung

Assertions können aktiviert oder deaktiviert werden. Hierfür gibt es die Kommandozeilenschalter `-ea` (enable assertion) und `-da` (disable assertion) der Java-Virtuellen Maschine.

Standardmäßig sind Assertions deaktiviert.

Die Kommandozeilenschalter können mit verschiedenen Parametern aufgerufen werden:

- **ohne Parameter**

  Es werden die Assertions aller Klassen außer den Systemklassen[181] deaktiviert bzw. aktiviert.

- **Paketname...**

  Die drei Punkte sind Teil der Syntax und müssen mit angegeben werden. Im angegebenen Paket und allen darin enthaltenen Unterpaketen werden die Assertions aktiviert bzw. deaktiviert.

- **...**

  Die drei Punkte sind Teil der Syntax und müssen angegeben werden. Assertions, die im anonymen Paket (aktuelles Arbeitsverzeichnis) enthalten sind, werden aktiviert bzw. deaktiviert.

- **Klassenname**

  Nur in der spezifizierten Klasse werden die Assertions aktiviert bzw. deaktiviert.

Um Assertions in mehreren Paketen oder Klassen gezielt ein- oder auszuschalten, können die Parameter auch mehrfach mit verschiedenen Optionen angegeben werden.

Soll beispielsweise das Programm `MyAssertionTest` ausgeführt werden und während der Ausführung alle Assertions in den Klassen des Pakets `java-`

[181] Systemklassen sind Klassen, die bereits vor dem Aufruf der Methode `main()` einer individuellen Anwendung innerhalb der Java-Virtuellen Maschine geladen sind. Sie werden implizit beim Starten der Java-Virtuellen Maschine geladen. Die Klasse `java.lang.Integer` ist beispielsweise eine solche Systemklasse.

`buch.uebungen` sowie in dessen Unterpaketen aktiviert werden, so lautet die Kommandozeile:

```
java –ea:javabuch.uebungen... MyAssertionTest
```

Sollen allerdings alle Assertions im Paket `javabuch.uebungen` einschließlich der Unterpakete aktiviert werden mit Ausnahme der Assertions in der Klasse `Test`, so lautet der Kommandozeilenaufruf:

```
java –ea:javabuch.uebungen...
     -da:javabuch.uebungen.Test
     MyAssertionTest
```

Um das Laufzeitverhalten zu verbessern, sollte ein Programm nach der Entwicklung – also im produktiven Einsatz – mit deaktivierten Assertions ausgeführt werden.

13.7.3 Anwendungsbeispiele

Es gibt verschiedene Situationen, in denen Assertions verwendet werden können, beispielsweise:

- Plausibilitätsprüfungen, d. h., ob ein Ergebnis in einem bestimmten Bereich liegt,
- Überprüfung des Kontrollflusses oder
- Überprüfung auf das Einhalten definierter diskreter Werte.

Beispiel 1: **Plausibilitätsprüfungen**, d. h. ob ein Ergebnis in einem bestimmten Bereich liegt

Werden in einer Methode Berechnungen durchgeführt, so ist es unter Umständen sinnvoll, das Ergebnis auf bestimmte Anforderungen zu testen: Das Ergebnis muss beispielsweise positiv sein, ein berechneter Stundenwert muss beispielsweise größer oder gleich 0 und kleiner oder gleich 23 sein, etc. Solche Überprüfungen können entweder direkt in der `assert`-Anweisung mit Hilfe eines Ausdrucks erfolgen, oder in einer Methode, die einen geeigneten Rückgabewert besitzt und von der Assertion aufgerufen wird.

Im Folgenden werden Plausibilitätsprüfungen demonstriert:

```
// Datei: AssertionTest.java
public class AssertionTest
{
   public static void main (String[] args)
   {
      int ergebnis = berechne(); // Der Rückgabewert von berechne()
                                 // soll überprüft werden.
      System.out.println("Das Ergebnis muss größer gleich 0 und " +
                         "kleiner als 24 sein.");

      assert testeErgebnis (ergebnis): "Ergebnis = " + ergebnis;
      System.out.println ("Das Ergebnis lautet: " + ergebnis);
   }
```

```
   private static int berechne()
   {
      //beliebige Berechnung
      return -1;
   }

   private static boolean testeErgebnis (int erg)
   {
      //Das Ergebnis muss größer gleich 0
      //und kleiner gleich 23 sein
      if ((erg >= 0) && (erg <= 23))
         return true;
      return false;
   }
}
```

Die Ausgabe des Programms ist:

```
Das Ergebnis muss größer gleich 0 und kleiner als 24 sein.
Exception in thread "main" java.lang.AssertionError: Ergebnis = -1
       at AssertionTest.main(AssertionTest.java:12)
```

Beachten Sie bitte, dass das Programm für die obige Ausgabe folgendermaßen aufgerufen wurde:

```
java -ea AssertionTest
```

Im vorangegangenen Beispiel wird im Fehlerfall das Ergebnis an den Konstruktor des Objekts der Klasse `AssertionError` übergeben. Dadurch erfährt der Entwickler nicht nur, dass die Berechnung fehlschlug, sondern auch, welches Ergebnis den Fehler verursacht hat.

Tritt eine Exception vom Typ `AssertionError` auf, so kann sie mit Hilfe eines `try-catch`-Konstrukts gefangen werden. Wie zuvor schon erwähnt, tritt eine Exception vom Typ `Error` – und damit auch vom Typ `AssertionError` – nur in schweren Ausnahmesituationen auf, von denen sich ein Programm normalerweise nicht erholt. Da die Ursache des Fehlers im Exception Handler nicht behoben werden kann, liegt ein schwerer Fehler vor und das Programm wird im `catch`-Block durch `System.exit(1)` abgebrochen. Man kann auf das Abfangen der Exception und das Anfordern des Programmabbruchs verzichten, da das Programm abgebrochen wird, wenn die Java-Virtuelle Maschine keinen Exception Handler für die aufgetretene Ausnahme finden kann.

Beispiel 2: Überprüfung des Kontrollflusses

Ebenso lässt sich überprüfen, ob eine Stelle im Kontrollfluss erreicht wird, die nie zur Ausführung kommen sollte. Dies zeigt das folgende Beispiel:

```
// finde eine Person, die zuvor über den Namen ausgewählt wurde

public Person findePerson (String name)
{
   for (Person p : allePersonen)
```

```
   {
      if (p.getName().equals(name))
      {
         return p;
      }
   }
   assert false;  // wird nur erreicht, wenn Person nicht
                  // auffindbar ist
}
```

In diesem Beispiel löst die Assertion einen Fehler aus, wenn die Bedingung in der `if`-Abfrage nicht erfüllt und die `return`-Anweisung somit nicht ausgeführt wird.

Beispiel 3: **Überprüfung auf das Einhalten definierter diskreter Werte**

Wird vorausgesetzt, dass eine Variable nur einige bestimmte Werte annehmen kann, so kann dies mit einer Assertion überprüft werden, wie in diesem Beispiel zu sehen ist:

```
int zahl;
. . . . .
switch (zahl)
{
   case 1:  \\ weitere Anweisungen
             break;
   case 2:  \\ weitere Anweisungen
             break;
   case 3:  \\ weitere Anweisungen
             break;
   default:  assert false;
}
. . . . .
```

Auch wenn keine `switch`-Anweisung verwendet wird, kann die Überprüfung von diskreten Werten notwendig sein. Eine Realisierungsmöglichkeit hierfür zeigt das folgende Beispiel:

```
assert (zahl==1 || zahl==2 || zahl==3); // nur 1,2,3 sind
                                        // zulässig
```

Es wird hier also erwartet, dass die Variable `zahl` keinen anderen Wert als `1`, `2` oder `3` annimmt. Sollte dies wider Erwarten dennoch geschehen, so wird die Assertion fehlschlagen und eine Exception vom Typ `AssertionError` auslösen. Auf diese Weise wird erkannt, dass die erwartete Eigenschaft nicht erfüllt wird und dass der Quellcode überarbeitet werden muss.

13.7.4 Unterschied zwischen Exceptions und Assertions

Ist eine Bedingung verletzt, so soll das Programm abbrechen, da es falsche Werte berechnet. In Java werden Assertions mit dem Schlüsselwort `assert` spezifiziert und auf Exceptions abgebildet. Die im Falle von Assertions geworfenen Exceptions

dürfen aber nur zu Zwecken der Fehlerausgabe verwendet werden. Anschließend muss das Programm beendet werden, da es schlichtweg inkorrekt ist.

Assertions kann man ein- und ausschalten. Im Gegensatz zu typisierten Exceptions (z. B. `OutOfMemoryError`), die erwartete Fehlersituationen darstellen, handelt es sich bei Assertions um Fehler, deren Ursache unbekannt ist und die nur durch das Verletzen von Bedingungen erkannt werden können. Ist beispielsweise eine Plausibilität verletzt, so erkennt man mit Assertions nur das Symptom, nicht aber die Ursache. Natürlich kann man beim Debuggen noch weitere Assertions einbauen, um einen erkannten Fehler rückwärts im Programm weiter zu verfolgen.

Beim Exception-Konzept besteht die Möglichkeit, eigene Exception-Typen zu definieren. Dies dient einerseits der differenzierten Behandlung von Exceptions und andererseits der Übergabe von Zusatzinformationen zur weiteren Verarbeitung. Beides ist bei Assertions nicht notwendig da diese stets zum Programmabbruch führen. Daher verwendet das Assertion-Konzept ausschließlich Objekte vom Typ `AssertionError`.

Im Gegensatz zu Exceptions können Assertions aktiviert und deaktiviert werden. Diese Möglichkeit besteht bei Exceptions nicht, da diese ein fester Bestandteil der Funktionalität eines Programms sind.

Während **Exceptions** ein **zentraler Baustein für die Behandlung von Ausnahmen** in einem produktiv eingesetzten Softwaresystem sind (mächtige Syntax der Exceptions und eine individuelle Anpassung), sind **Assertions Hilfskonstrukte**, die dem Programmierer während der Entwicklung dabei helfen sollen, **Programmfehler** schneller und effektiver zu erkennen und somit die Korrektheit des spezifizierten Umfanges während der Entwicklung im Auge behalten zu können (einfache Syntax und bei Bedarf abschaltbar).

13.8 Zusammenfassung

Abnormale Ereignisse, die während des Programmablaufs auftreten, brechen die normale Ausführung eines Programms ab und bringen es zum Absturz. Diese abnormalen Ereignisse werden Ausnahmen bzw. Exceptions genannt. Das Konzept des Exception Handling, das normalen Code von fehlerbehandelndem Code trennt, wird programmiersprachenunabhängig in Kapitel **13.1** vorgestellt.

Kapitel **13.2** zeigt, wie das Konzept des Exception Handling in Java umgesetzt wird. Kritische Ereignisse, die zu einem Programmabbruch führen könnten, werden innerhalb eines `try`-Blocks ausgeführt. Treten innerhalb eines `try`-Blocks Ausnahmen auf, so können diese in darauffolgenden `catch`-Blöcken gefangen werden. Gefangene Exceptions können entweder direkt in einem `catch`-Block behandelt werden, oder aber an den Aufrufer der Methode weitergereicht werden. Das Weiterreichen von Exceptions wird auch Propagieren von Exceptions genannt.

Die gemeinsame Basisklasse aller Exceptions, die Klasse `Throwable`, wird in Kapitel **13.3** vorgestellt. Die Klasse `Throwable` besitzt ein privates Datenfeld vom Typ `String`, um einen Fehlertext zu hinterlegen. Dieser Fehlertext kann mit der Methode `getMessage()` ausgelesen werden, die von allen Exceptions geerbt wird.

Kapitel **13.4** zeigt die Exception-Hierarchie anhand des Klassenbaums mit der Klasse `Throwable` als Wurzelklasse. Weiter werden die wichtigsten Klassen dieses Klassenbaums vorgestellt. In Java werden Ausnahmen, die vom Programmierer zwingend behandelt werden müssen (Checked Exceptions) von solchen unterschieden, die nicht zwingend behandelt werden müssen (Unchecked Exceptions). Unchecked Exceptions sind dabei alle Exceptions, die von der Klasse `Error` oder der Klasse `RuntimeException` ableiten. Checked Exceptions sind alle restlichen Exceptions in der Klassen-Hierarchie mit der Klasse `Throwable` als Wurzelklasse.

In Kapitel **13.5** wird aufgezeigt, wie Ausnahmen behandelt, d. h. durch sogenannte Exception Handler gefangen werden. Dabei spielt die Reihenfolge der Handler eine wichtige Rolle. Weiter können Ausnahmen an den Aufrufer weitergereicht werden. Dann übernimmt die aufrufende Methode die Ausnahmebehandlung. Werden Checked Exceptions an den Aufrufer weitergereicht, so müssen diese in der Methodensignatur angekündigt werden. Das Ankündigen von Exceptions erfolgt mit dem Schlüsselwort `throws`.

Die Vorteile des Exceptions-Konzepts sind u. a. die saubere Trennung zwischen "normalem" Code und fehlerbehandelndem Code sowie die Möglichkeit, Programmfehler bereits zur Kompilierzeit durch Checked Exceptions aufzudecken (siehe Kapitel **13.6**).

Assertions sind zwar mittels Exceptions realisiert, sie sollen aber im Gegensatz zu Exceptions nur während der Entwicklung eines Programms eingesetzt werden und stellen somit eine Möglichkeit zum Testen dar (Kapitel **13.7**). Assertions können beim Programmablauf aktiviert oder deaktiviert werden. Im produktiven Einsatz sollten sie immer deaktiviert sein.

13.9 Übungen

Aufgabe 13.1: Division durch Null

Entwickeln Sie eine Klasse `Teilen`. Initialisieren Sie in der `main()`-Methode die zwei Variablen `zaehler` und `nenner` vom Typ `int`. Der Variablen `zaehler` wird eine beliebige ganze Zahl und der Variablen `nenner` die Zahl `0` zugewiesen. Danach soll das Ergebnis der Berechnung `zaehler/nenner` in der Konsole ausgegeben werden. Fangen Sie die bei der Berechnung entstehende Ausnahme vom Typ `ArithmeticException` in einem `try-catch`-Block ab. Analysieren Sie die Ausnahme, indem Sie sich Informationen über die Exception mit Hilfe der Methode `printStackTrace()` in der Konsole ausgeben lassen.

Aufgabe 13.2: Exceptions

Erstellen Sie eine Klasse `Bankkonto`. Eine Kontoführung soll durch Einzahlungen und Auszahlungen simuliert werden. Die Klasse `Bankkonto` besitzt die Methoden:

- `public void einzahlen (double betrag)`
- `public void auszahlen (double betrag)`
- `public double getKontostand()`

Die Methoden `einzahlen()` und `auszahlen()` werfen eine Exception vom Typ `TransaktionsException` beim Auftreten eines Transaktionsfehlers. Leiten Sie hierzu die Klasse `TransaktionsException` von der Klasse `Exception` ab. Ein Transaktionsfehler wird durch einen negativen Ein- oder Auszahlungsbetrag oder ein nicht ausreichend großes Guthaben für einen Auszahlungsbetrag verursacht. Die Methode `getKontostand()` liefert den aktuellen Kontostand, der durch ein privates Datenfeld vom Typ `double` realisiert wird. Die Klasse `Bankkonto` soll mit folgender Klasse getestet werden:

```
// Datei: TestBankkonto.java
public class TestBankkonto
{
   public static void main (String[] args)
   {
      Bankkonto konto = new Bankkonto();
      double betrag;
      System.out.println ("Kontostand: " + konto.getKontostand());

      try
      {
         betrag = 123.45;
         System.out.println();
         System.out.println ("Einzahlung: " + betrag);
         konto.einzahlen (betrag);
         System.out.println ("Kontostand: " +
                              konto.getKontostand());
      }
      catch (TransaktionsException ex)
      {
         System.out.println (ex.getMessage());
      }

      try
      {
         //Negative Einzahlung
         betrag = -12.45;
         System.out.println();
         System.out.println ("Einzahlung: " + betrag);
         konto.einzahlen (betrag);
         System.out.println ("Kontostand: " +
                              konto.getKontostand());
      }
      catch (TransaktionsException ex)
      {
         System.out.println (ex.getMessage());
      }

      try
      {
         //Negative Auszahlung
         betrag = -12.45;
         System.out.println();
         System.out.println ("Auszahlung: " + betrag);
         konto.auszahlen (betrag);
         System.out.println ("Kontostand: " +
                              konto.getKontostand());
      }
```

```
      catch (TransaktionsException ex)
      {
         System.out.println (ex.getMessage());
      }

      try
      {
         betrag = 12;
         System.out.println();
         System.out.println ("Auszahlung: " + betrag);
         konto.auszahlen (betrag);
         System.out.println ("Kontostand: " +
                              konto.getKontostand());
      }
      catch (TransaktionsException ex)
      {
         System.out.println (ex.getMessage());
      }

      try
      {
         //Konto überziehen
         betrag = 130;
         System.out.println();
         System.out.println ("Auszahlung: " + betrag);
         konto.auszahlen (betrag);
         System.out.println ("Kontostand: " +
                              konto.getKontostand());
      }
      catch (TransaktionsException ex)
      {
         System.out.println (ex.getMessage());
      }
   }
}
```

Aufgabe 13.3: Exceptions

Es soll ein Login-Szenario entwickelt werden. Die Klasse `Login` besitzt folgende Instanzvariablen und Methoden:

- `private boolean angemeldet;`
- `public void anmelden (String benutzer, String passwort)`
- `public void abmelden()`
- `public void bearbeiten()`

Die Methode `anmelden()` setzt bei erfolgreicher Anmeldung die Instanzvariable `angemeldet` auf `true` und wirft bei fehlschlagender Authentifizierung ein Objekt der Klasse `ZugriffUngueltigException`, die von der Klasse `Exception` abgeleitet wird. Ebenfalls soll, wenn ein nicht angemeldeter Benutzer auf die Methode `bearbeiten()` zugreifen möchte, eine Ausnahme vom Typ `KeineBerechtigungException` geworfen werden. Die Methode `abmelden()` setzt die Instanzvariable `angemeldet` auf `false`. Die Methode `bearbeiten()` gibt eine Meldung auf der Konsole aus, um einen Arbeitsvorgang zu simulieren. Entwickeln Sie die Klassen

`Login`, `ZugriffUngueltigException`, `KeineBerechtigungException`. Die entwickelten Klassen sollen mit folgender Testklasse getestet werden:

```
// Datei: TestLogin.java
import java.util.Scanner;

public class TestLogin
{
   public static void main (String[] args)
   {
      Login login = new Login();
      Scanner input = new Scanner (System.in);
      String benutzer = "";
      String passwort = "";

      System.out.print ("Bitte geben Sie den " +
                           "Benutzernamen ein:");
      benutzer = input.next();
      System.out.println ("Bitte geben Sie das Passwort ein:");
      passwort = input.next();

      try
      {
         System.out.println ("Sie werden angemeldet ...");
         login.anmelden (benutzer, passwort);
         System.out.println ("Anmeldung erfolgreich!");
      }
      catch (ZugriffUngueltigException ex)
      {
         System.out.println (ex.getMessage());
      }

      try
      {
         System.out.println ("Methode bearbeiten() " +
                              "wird aufgerufen ...");
         login.bearbeiten();
      }
      catch (KeineBerechtigungException ex)
      {
         System.out.println (ex.getMessage());
      }

      System.out.println ("Sie werden abgemeldet ...");
      login.abmelden();

      try
      {
         System.out.println ("Methode bearbeiten() " +
                              "wird aufgerufen ...");
         login.bearbeiten();
      }
      catch (KeineBerechtigungException ex)
      {
         System.out.println (ex.getMessage());
      }
   }
}
```

Aufgabe 13.4: Exception Handling durch Assertion ersetzen

Aufbauend auf der Übungsaufgabe 13.1 entfernen Sie das Exception Handling und ersetzen es durch eine Assertion. Es soll eine Unchecked Exception verhindert werden, indem der Wert des Nenners auf `0` geprüft wird. Geben sie eine Meldung auf der Konsole aus, sobald der Nenner `0` ist.

Vergleichen Sie die Ausgabe des Programms aus dieser Aufgabe mit der Ausgabe des Programms aus Aufgabe 13.1.

Aufgabe 13.5: Aufgabe zu Assertions

Erweitern sie das folgende Beispielprogramm `AssertionTest` um einen Exception Handler. Im Exception Handler soll ausgegeben werden, mit welchem falschen aktuellen Parameter die Methode `berechne()` aufgerufen wurde. Vergessen Sie aber nicht, mit `System.exit(1)` das Programm zu verlassen! Hier nochmals das Beispielprogramm:

```
// Datei: AssertionTest.java

class AssertionTest
{
   public static void main (String[] args)
   {
      int y = 11;

      assert (y > 0 && y < 10) :
         "Falscher Übergabeparameter von berechne()";
      System.out.println (berechne (y));
   }

   // Die Vorbedingung von berechne() ist: 0 < x < 10
   static int berechne (int x)
   {
      return x*x*x*x;
   }
}
```

Aufgabe 13.6: Flughafen-Projekt – Integration von Exceptions

Bevor Exceptions in das Flughafen-Projekt integriert werden, soll zuerst noch eine kleine Weiterentwicklung gemacht werden: Bisher konnte eine Parkstelle und eine Start-/Landebahn nicht als frei/belegt gekennzeichnet werden. Dies soll nun geändert werden.

Fügen Sie die folgenden beiden abstrakten Methoden der Klasse `Parkstelle` hinzu:

```
public abstract void belegen (Flugzeug flugzeug);
public abstract void freigeben (Flugzeug flugzeug);
```

Diese Methoden sollen dann in den abgeleiteten Klassen implementiert werden. Da weder eine Werft noch ein separates Parkfeld als belegt zu kennzeichnen ist, sollen die Methodenrümpfe der beiden Klassen `SeparatesParkfeld` und `Werft` ohne Funktion – d. h. mit einem leeren Methodenrumpf – implementiert werden. Die

Klasse `Parkposition` hingegen soll die Referenz auf das Flugzeug beim Aufruf von `belegen()` intern speichern und bei `freigeben()` wieder auf `null` setzen. Die gleiche Funktionalität soll in der Klasse `Bahn` implementiert werden. Ändern Sie dabei die Methoden für die Phasen "Landebahn vergeben", "Parkstelle vergeben" und "Startbahn vergeben" so ab, dass die Bahn beziehungsweise Parkstelle durch das Flugzeug belegt wird.

Die bisherigen Phasen für die Landung und den Start sollen um drei Phasen erweitert werden. Der Status eines Flugzeugs soll sich dadurch nicht ändern. Die drei Phasen sind:

- Landebahn freigeben,
- Parkstelle freigeben
- und Startbahn freigeben.

Schreiben Sie die dafür notwendigen Methoden und passen Sie zusätzlich die Klasse `Client` so an, dass diese Methoden während des Lande- bzw. Startvorgangs aufgerufen werden.

Bislang kann eine Landebahn von zwei verschiedenen Flugzeugen gleichzeitig belegt werden. Ein Beispiel hierzu wäre:

```
Bahn bahn = new Bahn();
bahn.belegen (flugzeug1);
bahn.belegen (flugzeug2);
```

Dieser und weitere Fehler sollen nun abgefangen werden. Es sollen dabei folgende zwei Exception-Klassen geschrieben werden:

- `BelegungsException`

  Die Exception `BelegungsException` soll in den Methoden `belegen()` der Klassen `Bahn` und `Parkposition` geworfen werden, wenn diese bereits von einem anderen Flugzeug belegt sind. Beachten Sie dabei, dass die Klasse `Parkposition` die abstrakte Klasse `Parkstelle` erweitert, womit diese Klasse auch angepasst werden muss. Des Weiteren soll diese Exception beim Aufruf der Methoden für die Phasen "Landebahn vergeben", "Parkstelle vergeben" und "Startbahn vergeben" geworfen werden, falls dem Flugzeug bereits eine Lande-/Startbahn bzw. Parkstelle zugewiesen wurde.

- `FreigabeException`

  Die Exception `FreigabeException` soll in den Methoden `freigeben()` der Klassen `Bahn` und `Parkposition` geworfen werden, wenn die Parkposition von einem anderen Flugzeug belegt ist. Diese Exception soll auch beim Aufruf der Methoden für die Phasen "Landebahn freigeben", "Parkstelle freigeben" und "Startbahn freigeben" geworfen werden, falls dem Flugzeug noch keine Start-/Landebahn bzw. Parkstelle zugewiesen wurde.

Ändern Sie auch den Client so ab, dass die eventuell geworfenen Exceptions gefangen und verarbeitet werden. Die Verarbeitung könnte hierbei als ein erneuter Versuch oder auch als Programmabbruch implementiert sein.

Kapitel 14

Schnittstellen

14.1 Trennung von Spezifikation und Implementierung
14.2 Ein weiterführendes Beispiel
14.3 Aufbau einer Schnittstelle
14.4 Verwenden von Schnittstellen
14.5 Vergleich Schnittstelle und abstrakte Basisklasse
14.6 Die Schnittstelle `Cloneable`
14.7 Marker Interfaces
14.8 Zusammenfassung
14.9 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_14

14 Schnittstellen

Eine Klasse enthält Methoden und Datenfelder. Methoden bestehen aus Methodenköpfen und Methodenrümpfen. **Methodenköpfe**[182] der Exportmethoden stellen die **Schnittstellen** (engl. **Interfaces**) **eines Objektes zu** seiner **Außenwelt** dar. Dies zeigt das folgende Bild:

Bild 14-1 Methodenköpfe als Schnittstellen verbergen Methodenrümpfe und Datenfelder

In einer guten Implementierung sind die Daten eines Objekts im Inneren des Objektes verborgen. Nach außen sind nur die Schnittstellen der Exportmethoden – also die Methodenköpfe – sichtbar.

Entwirft man komplexe Systeme, so werden sie im ersten Schritt in einfachere Teile, die **Teilsysteme** bzw. **Subsysteme**, zerlegt. Die Identifikation eines Subsystems ist nicht trivial. Als Qualitätsmaß für die Güte des Entwurfs werden hierbei das **Coupling**, d. h. die Stärke der Wechselwirkungen zwischen den Subsystemen, und die **Cohesion**, d. h. die Stärke der Abhängigkeiten innerhalb eines einzelnen Subsystems, betrachtet.

Ein Entwurf gilt dann als gut, wenn innerhalb eines Subsystems eine **Strong Cohesion** und zwischen den Subsystemen ein **Loose Coupling** besteht.

Als nächstes müssen dann die Wechselwirkungen zwischen den Subsystemen "festgezurrt", in anderen Worten in Form von Schnittstellen definiert werden.

Die Implementierung der Subsysteme interessiert beim Grobentwurf nicht und wird verborgen (**Information Hiding**), d. h. die Schnittstellen stellen eine **Abstraktion** der Subsysteme dar.

[182] Hierbei wird vorausgesetzt, dass die Methoden nicht den Zugriffsmodifikator `private` tragen.

Sind die Schnittstellen zwischen den Subsystemen stabil, so können sich nun verschiedene Arbeitsgruppen parallel mit dem Entwurf der jeweiligen Subsysteme befassen. Diese Arbeitsgruppen können vollkommen unabhängig voneinander arbeiten, solange sie die Schnittstellen nicht antasten.

Kapitel 14.1 behandelt die Trennung der Spezifikation und Implementierung von Schnittstellen. Anhand eines weiterführenden Beispiels wird in Kapitel 14.2 die Implementierung von Schnittstellen verdeutlicht. Kapitel 14.3 beschreibt den Aufbau einer Schnittstelle und Kapitel 14.4 die Verwendung von Schnittstellen. Anschließend wird in Kapitel 14.5 das Konzept einer Schnittstelle mit dem Konzept einer abstrakten Basisklasse verglichen. Kapitel 14.6 stellt die Schnittstelle `Cloneable` und ihre Verwendung für das Klonen von Objekten vor. Zum Schluss werden in Kapitel 14.7 Marker Interfaces analysiert.

14.1 Trennung von Spezifikation und Implementierung

Eine gute Programmiersprache sollte das **Programmieren im Großen** – sprich den Entwurf – unterstützen. Java bietet mit dem Sprachmittel **`interface`** die Möglichkeit, den Entwurf zu unterstützen und die Schnittstellen einer Klasse in der Sprache Java zu formulieren.

Die **Implementierung** einer Schnittstelle stellt eine **Verfeinerung des Entwurfs** dar. Bei stabilen Schnittstellen einer Klasse kann dann deren Implementierung geändert werden, ohne dass ein Kundenprogramm, das diese Klasse benutzt, verändert werden muss.

Damit die Sache greifbar wird, sofort ein **Beispiel**:

```
// Datei: Punkt.java
interface PunktSchnittstellen
{
   public int getX();        // Eine Methode, um den x-Wert abzuholen
   public void setX (int i);// Eine Methode, um den x-Wert zu setzen
}

public class Punkt implements PunktSchnittstellen
{
   private int x;            //x-Koordinate vom Typ int

   public int getX()         // Alle Methoden der Schnittstelle
   {                         // Punktschnittstellen müssen in der
      return x;              // Klasse implementiert werden, wenn die
   }                         // Klasse instanziierbar sein soll.

   public void setX (int i)
   {
      x = i;
   }

   public static void main (String[] args)
   {
      Punkt p = new Punkt(); // Hiermit wird ein Punkt erzeugt
      p.setX (3);
```

```
        System.out.println ("Die Koordinate des Punktes p ist: ");
        System.out.println (p.getX());
    }
}
```

Die Ausgabe des Programms ist:

```
Die Koordinate des Punktes p ist:
3
```

Visualisiert werden kann die Verwendung der hier verwendeten Schnittstelle `PunktSchnittstellen` durch die folgende grafische Notation nach UML:

Bild 14-2 Implementierung der Schnittstelle `PunktSchnittstellen`

Hierbei symbolisiert der gestrichelte Pfeil von der Klasse `Punkt` zur Schnittstelle `PunktSchnittstellen`, dass die Klasse Punkt die **Schnittstelle** `PunktSchnittstellen` **implementiert**. Der gestrichelte Pfeil bedeutet eine **Verfeinerung**. Mit anderen Worten, die Schnittstelle `PunktSchnittstellen` enthält nur die Spezifikation der Methodenköpfe, die Verfeinerung der Methoden – sprich die Implementierung der Rümpfe – erfolgt in der Klasse `Punkt`. Das folgende Bild zeigt die "Lollipop"-Notation für Schnittstellen:

Bild 14-3 "Lollipop"-Notation

Anstelle der Notation mit einem Rechteckrahmen wie in Bild 14-2 kann eine implementierte Schnittstelle auch als "Lollipop" – als ein Kreis mit einer zugehörigen Linie – notiert werden (siehe Bild 14-3).

Der "Lollipop" dient nur zur Darstellung der Nahtstellen und kann nur dann verwendet werden, wenn die Schnittstellendefinition an anderer Stelle bereits ersichtlich ist.

Die "Lollipop"-Darstellung hat jedoch den Vorteil, dass sie sehr kompakt ist.

Eine **Schnittstelle** (ein **Interface**) ist ein **Sprachmittel für** den **Entwurf**. Eine **Klasse beinhaltet** dagegen – sofern sie nicht abstrakt ist – den **Entwurf und** die **Implementierung**, d. h. auch die Methodenrümpfe.

Es ist auch möglich, dass eine Klasse mehrere Schnittstellen implementiert.

Damit hat man die Möglichkeit, Schnittstellen aufzuteilen und auch "Bibliotheks-Schnittstellen" zu identifizieren, die in mehreren Klassen – ggf. mit verschiedenen Methodenrümpfen – implementiert werden können.

Alle Schnittstellen, die von einer Klasse implementiert werden, bilden zusammen die **Aufrufschnittstelle** der Klasse.

14.2 Ein weiterführendes Beispiel

Es soll folgendes Szenario betrachtet werden: Eine Person ist immer an wichtigen Ereignissen interessiert. Deshalb implementiert sie eine **Schnittstelle** `NachrichtenEmpfaenger`. Nachrichten wiederum können von verschiedenen Quellen erzeugt werden, z. B. könnten Objekte wie Radio, Fernseher, Zeitung usw. Informationen erzeugen und sie an alle interessierten Benutzer senden. Die Fähigkeit, Nachrichten zu versenden, lässt sich somit auch in eine **Schnittstelle** `NachrichtenQuelle` abstrahieren. Alle Klassen, deren Objekte die Fähigkeit erhalten sollen, Nachrichten zu versenden, implementieren also die Schnittstelle `NachrichtenQuelle`. Jede Person kann sich nun nach Interesse bei den verschiedenen Nachrichtenquellen anmelden. Erzeugt eine Nachrichtenquelle eine Nachricht, so werden alle angemeldeten Interessenten benachrichtigt.

Aus dieser Beschreibung ergeben sich die folgenden Schnittstellen:

```
// Datei: Nachrichten.java

interface NachrichtenQuelle
{
   public boolean anmelden (NachrichtenEmpfaenger empf);
   public void sendeNachricht (String nachricht);
}

interface NachrichtenEmpfaenger
{
   public void empfangeNachricht (String nachricht);
}
```

Eine Klasse `Radio`, `Zeitung` oder `Fernseher` könnte z. B. die Schnittstelle `NachrichtenQuelle` implementieren, wie das folgende Bild zeigt:

Bild 14-4 Klassen, welche die Schnittstelle NachrichtenQuelle implementieren

Genauso wie man, um Post zu empfangen, dem Sender seine Adresse mitteilen muss, müssen auch Objekte, die Nachrichten empfangen wollen, ihre Adresse dem Sender bekannt geben. Dies geschieht in der Methode `anmelden()`. Diese Methode einer Nachrichtenquelle hat die Aufgabe, die Adresse (programmtechnisch die Referenz) eines Objektes, welches das Interface `NachrichtenEmpfaenger` implementiert, entgegenzunehmen, damit bei einer auftretenden Nachricht der Interessent informiert werden kann. Der Code für eine Nachrichtenquelle `Zeitung` könnte wie in folgendem **Beispiel** aussehen:

```
// Datei: Zeitung.java

public class Zeitung implements NachrichtenQuelle
{
   private String name;
   private NachrichtenEmpfaenger[] arr;
   private int anzahlEmpfaenger = 0;

   public Zeitung (String name, int maxAnzahlEmpfaenger)
   {
      this.name = name;
      arr = new NachrichtenEmpfaenger [maxAnzahlEmpfaenger];
   }

   public boolean anmelden (NachrichtenEmpfaenger empf)
   {
      if (anzahlEmpfaenger < arr.length)
      {
          arr [anzahlEmpfaenger++] = empf;
          return true;
      }
      return false;
   }

   public void sendeNachricht (String nachricht)
   {
      // Alle angemeldeten Nachrichtenempfänger
      // werden benachrichtigt
      for (int i = 0; i < anzahlEmpfaenger; i++)
      {
         arr [i].empfangeNachricht (nachricht);
      }
   }
}
```

Wie in Bild 14-5 gezeigt, soll die Klasse `Person` die Schnittstelle `Nachrichten-Empfaenger` implementieren:

Bild 14-5 Klasse `Person` implementiert die Schnittstelle `NachrichtenEmpfaenger`

Hier die Klasse `Person`:

```
// Datei: Person.java

public class Person implements NachrichtenEmpfaenger
{
   private String name;
   private String vorname;

   public Person (String name, String vorname)
   {
      this.name = name;
      this.vorname = vorname;
   }

   public void empfangeNachricht (String nachricht)
   {
      System.out.println ("an " + name + " " + vorname
                          + ":  " + nachricht);
   }
}
```

Zum Testen der Klassen `Person` und `Zeitung` wird folgendes Programm benutzt:

```
// Datei: Test.java

public class Test
{
   public static void main (String[] args)
   {
      Person p1 = new Person ("Fischer", "Fritz");
      Person p2 = new Person ("Maier", "Hans");
      Person p3 = new Person ("Kunter", "Max");

      Zeitung z1 = new Zeitung ("-Frankfurter Allgemeine-", 10);
      z1.anmelden (p1);
      z1.anmelden (p2);
```

```
        Zeitung z2 = new Zeitung ("-Südkurier-", 10);
        z2.anmelden (p1);
        z2.anmelden (p3);

        System.out.println ("Frankfurter Allgemeine Schlagzeile:");
        z1.sendeNachricht ("Neues Haushaltsloch von 30 Mrd. EURO");
        System.out.println();
        System.out.println ("Südkurier Schlagzeile:");
        z2.sendeNachricht ("Bayern München Deutscher Meister");
    }
}
```

Die Ausgabe des Programms ist:

```
Frankfurter Allgemeine Schlagzeile:
an Fischer Fritz:  Neues Haushaltsloch von 30 Mrd. EURO
an Maier Hans:  Neues Haushaltsloch von 30 Mrd. EURO

Südkurier Schlagzeile:
an Fischer Fritz:  Bayern München Deutscher Meister
an Kunter Max:  Bayern München Deutscher Meister
```

Mit dem `Aufruf z1.sendeNachricht ("Neues Haushaltsloch von 30 Mrd. Euro")` werden zwei Personen benachrichtigt. Das sind genau die Personen `p1` und `p2`, die sich mit `z1.anmelden (p1)` und mit `z1.anmelden (p2)` als Nachrichtenempfänger angemeldet haben.

Das folgende Bild veranschaulicht den Benachrichtigungsablauf:

Bild 14-6 Nachrichtenquelle Zeitung benachrichtigt die registrierten Nachrichtenempfänger

Zusätzlich ist zu beachten, dass in der Deklaration der Methode `anmelden()` als **formaler Übergabeparameter** ein **Schnittstellentyp** angegeben wird. Als **aktueller Übergabeparameter** wird allerdings eine Referenz auf ein Objekt der Klasse `Person` übergeben. Dies funktioniert, da die Klasse `Person` die Schnittstelle `NachrichtenEmpfaenger` **implementiert** und bei der Parameterübergabe ein Up-Cast in den Schnittstellentyp erfolgt.

Wird als **formaler Übergabeparameter** ein **Schnittstellentyp** angegeben, so kann eine **Referenz auf ein Objekt**, **dessen Klasse diese Schnittstelle implementiert**, als **aktueller Parameter** übergeben werden. Referenzen auf Objekte eines anderen Typs werden vom Compiler abgelehnt.

14.3 Aufbau einer Schnittstelle

Eine **Schnittstellendefinition** besteht ähnlich wie eine Klassendefinition aus zwei Teilen:

- der **Schnittstellendeklaration**
- und dem **Schnittstellenkörper** mit Konstantendefinitionen und Methodendeklarationen.

Das folgende Beispiel demonstriert die **Definition einer Schnittstelle**:

```
public interface NachrichtenQuelle2      } Schnittstellendeklaration
{
   public static final int SPORT = 0;
   public static final int POLITIK = 1;
   public static final int KULTUR = 2;
   public static final int ANZEIGEN = 3;          Schnittstellen-
   public static final int GESAMT = 4;            körper

   public boolean anmelden
            (NachrichtenEmpfaenger empf, int typ);
   public void sendeNachricht (String nachricht);
}
```

Die Schnittstellendeklaration

Die Schnittstellendeklaration setzt sich aus drei Elementen zusammen:

- dem optionalen **Zugriffsmodifikator** `public`[183],
- dem Schlüsselwort **interface** mit dem Schnittstellennamen
- und optional dem Schlüsselwort **extends** mit durch Kommata getrennten Schnittstellennamen, von denen abgeleitet wird.

Wird `public` nicht angegeben, so wird der Zugriffsschutz default verwendet, was bedeutet, dass die Sichtbarkeit auf das eigene Paket beschränkt ist.

Im Folgenden sind diese Elemente tabellarisch zusammengefasst:

[183] Bei der Programmierung mit geschachtelten Klassen (siehe Kapitel 15) sind auch die Zugriffsmodifikatoren `private` und `protected` für Schnittstellen möglich.

`public`	optional
`interface Schnittstellenname`	zwingend erforderlich
`extends S1, S2, . . . , Sn`	optional ableiten von anderen Schnittstellen `S1` bis `Sn`

Tabelle 14-1 Elemente einer Schnittstellendeklaration

Der Zugriffsmodifikator `public` sorgt dafür, dass die Schnittstelle nicht nur im eigenen Paket, sondern in allen Paketen sichtbar ist.

Mit `public` deklarierte Schnittstellen können – genauso wie Klassen – mittels der `import`-Vereinbarung in anderen Paketen sichtbar gemacht werden. Schnittstellen, die nicht mit `public` deklariert sind, sind default und damit nur im eigenen Paket sichtbar.

Ist eine Schnittstelle mit `public` deklariert, so darf in derselben Quellcode-Datei keine weitere Klasse oder Schnittstelle stehen, die auch `public` ist. Hier gelten die gleichen Konventionen wie bei Klassen.

Schnittstellen können mit **`extends`** **von** anderen **Schnittstellen abgeleitet werden**. Mit anderen Worten, es ist möglich, eigene **Schnittstellenhierarchien** aufzubauen.

Klassen können dagegen mit `extends` **nicht von Schnittstellen abgeleitet** werden. Sie können jedoch beliebig viele **Schnittstellen** mit `implements` **implementieren**.

Der Schnittstellenkörper

Der Schnittstellenkörper enthält:

- Konstantendefinitionen
- und Methodendeklarationen.

Alle in einer Schnittstelle aufgeführten Methoden sind automatisch `public` und `abstract`. Somit enthält eine Schnittstelle auch keine Methodenimplementierung, da abstrakte Methoden keinen Methodenrumpf besitzen können.

Bei der Methodendeklaration ist die explizite Angabe von `public` und **`abstract`** optional. Fehlen diese Schlüsselwörter, so werden sie automatisch vom Compiler eingefügt. Methoden in Schnittstellen besitzen – da sie `abstract` sind – keinen Methodenrumpf.

Versucht man, den Zugriffsmodifikator einer Schnittstellenmethode z. B. auf `private` zu setzen[184], bringt der Compiler eine Fehlermeldung.

Es macht ebenso keinen Sinn, eine Schnittstellenmethode als `final` zu deklarieren, da als `final` deklarierte Methoden bekanntlich nicht mehr überschrieben und damit auch nicht implementiert werden können. Dies wird ebenfalls vom Compiler überprüft.

Zur Anschauung einige korrekte und falsche Methodendeklarationen:

```
public interface NachrichtenQuelle3
{
   // Explizit public abstract
   public abstract boolean anmelden (NachrichtenEmpfaenger empf);

   // Explizit public, implizit abstract
   public void sendeNachricht (String nachricht);

   // Auch möglich: Implizit public abstract
   // void sendeNachricht (String nachricht);

   // Nicht möglich
   // private sendeNachricht (String nachricht);
}
```

Konstanten in Schnittstellen können als Übergabeparameter für eine Schnittstellenmethode verwendet werden.

Im oben angeführten Beispiel der Schnittstelle `NachrichtenQuelle2` sind die Konstanten `SPORT`, `POLITIK`, `KULTUR`, `ANZEIGEN` und `GESAMT` definiert, die als Übergabeparameter für die Methode `anmelden()` verwendet werden. Damit kann ein Objekt vom Typ `NachrichtenEmpfaenger` beim Anmelden angeben, welchen Nachrichtentyp es empfangen möchte.

Bezüglich der Angabe der **Modifikatoren** `public`, `static` und `final` bei den **Konstantendefinitionen** besteht vollkommene Freiheit. Es können alle Modifikatoren angegeben werden, es können aber auch alle weggelassen werden. Die nicht angegebenen Modifikatoren werden dann durch den Compiler hinzugefügt.

Wird jedoch versucht, explizit den Zugriffsmodifikator `private` oder `protected` zu setzen, so bringt der Compiler eine Fehlermeldung. Ob nun die Angabe `public static final` gemacht wird oder nicht, alle Konstanten einer Schnittstelle müssen initialisiert werden. Das folgende Beispiel zeigt verschiedene zulässige und nicht zulässige Varianten bei der Initialisierung:

```
public interface NachrichtenQuelle4
{
```

[184] Vorbedingungen einer Methode dürfen nicht verschärft werden.

```
   public static final int SPORT = 0;
   int POLITIK = 1;  // ist public static final
   public int KULTUR = 2;
   public int ANZEIGEN = 3;
   public int GESAMT = 4;
   public int ZUFALL = (int) (Math.random() * 5);
   // private int REGIONALES = 5; Fehler, da kein Zugriff möglich
   // int SONSTIGES; Fehler, da Konstante initialisiert werden muss

   public boolean anmelden (NachrichtenEmpfaenger empf, int typ);
   public void sendeNachricht (String nachricht);
}
```

Jede Konstantendefinition in einer Schnittstelle muss einen Initialisierungsausdruck besitzen.

Der **Initialisierungsausdruck muss** dabei **nicht konstant sein**, sondern kann – wie im obigen Beispiel zu sehen ist – sogar einen Funktionsaufruf wie z. B. `Math.random()` enthalten.

14.4 Verwenden von Schnittstellen

Bei der Verwendung von Schnittstellen gibt es einige Besonderheiten zu beachten, welche in diesem Kapitel betrachtet werden sollen.

14.4.1 Implementieren einer Schnittstelle

Eine Schnittstelle kann durch Angabe des Schlüsselwortes `implements` und des Schnittstellennamens von einer Klasse implementiert werden. Durch

```
class B extends A implements I1, I2
```

deklariert eine Klasse `B`, dass sie ein Subtyp der Klasse `A` ist und zusätzlich die Schnittstellen `I1` und `I2` implementiert.

Eine Klasse gibt mit dem Schlüsselwort `implements` an, welche Schnittstellen sie implementiert.

Implementiert eine Klasse eine Schnittstelle, so muss sie **alle Methoden** dieser Schnittstelle **implementieren**, **wenn sie instanziiert werden soll** – d. h. wenn von ihr Objekte geschaffen werden sollen. **Ansonsten** muss die **Klasse** als **abstrakt deklariert** werden und ist damit **nicht instanziierbar**.

Implementiert die Klasse `Zeitung` aus Kapitel 14.2 nur die abstrakte Methode `anmelden()` der Schnittstelle `NachrichtenQuelle` und die Methode `sendeNachricht()` nicht, so ist die Klasse `Zeitung` mit dem Schlüsselwort `abstract` zu kennzeichnen. Abstrakte Klassen können nicht instanziiert werden.

Eine Klasse, die eine Schnittstelle implementiert, erbt die in der Schnittstelle enthaltenen Konstanten und abstrakten Methoden.

Es kann durch Schnittstellen keine Funktionalität geerbt werden, da Schnittstellen keine Methodenimplementierung beinhalten.

Ein Programmierer hat bei der Implementierung einer Schnittstellenmethode darauf zu achten, dass er den Vertrag der Methode erfüllt.

14.4.2 Schnittstellen als Datentyp

Einer Referenz vom Typ einer Schnittstelle kann als Wert eine Referenz auf ein Objekt zugewiesen werden, dessen Klasse die Schnittstelle implementiert. Hierzu soll das Beispiel aus Kapitel 14.2 nochmals betrachtet werden. Die Klasse `Person` implementiert die Schnittstelle `NachrichtenEmpfaenger`. Es kann also beim Anlegen von Objekten der Klasse `Person` anstatt

```
Person p1 = new Person ("Fischer", "Fritz");
```

auch

```
NachrichtenEmpfaenger p1 = new Person ("Fischer", "Fritz");
```

geschrieben werden.

Eine **Schnittstelle** ist ein **Referenztyp**. Von ihm können **Referenzvariablen** gebildet werden, die auf **Objekte zeigen**, **deren Klassen die Schnittstelle implementieren**. Es ist damit auch möglich, **Arrays von Schnittstellentypen** anzulegen und diese Arrays mit Referenzen auf Objekte zu füllen, deren Klassen die Schnittstelle implementieren.

Das folgende **Beispiel** zeigt erneut die Testklasse aus Kapitel 14.2 mit der gleichen Funktionalität wie dort, hier jedoch in der Ausprägung, dass Arrays von Schnittstellen verwendet werden.

```
// Datei: Test2.java

public class Test2
{
   public static void main (String[] args)
   {
      NachrichtenEmpfaenger[] senke = new NachrichtenEmpfaenger [3];
      senke [0] = new Person ("Fischer", "Fritz");
      senke [1] = new Person ("Maier", "Hans");
      senke [2] = new Person ("Kunter", "Max");
```

```
        NachrichtenQuelle[] quelle = new NachrichtenQuelle[2];
        quelle [0] = new Zeitung ("-Frankfurter Allgemeine-", 10);
        quelle [0].anmelden (senke [0]);
        quelle [0].anmelden (senke [1]);
        quelle [1] = new Zeitung ("-Südkurier-", 10);
        quelle [1].anmelden (senke [0]);
        quelle [1].anmelden (senke [2]);

        System.out.println ("Frankfurter Allgemeine Schlagzeile:");
        quelle [0].sendeNachricht ("Neues Haushaltsloch " +
                                   "von 30 Mrd. EURO");
        System.out.println();
        System.out.println ("Südkurier Schlagzeile:");
        quelle [1].sendeNachricht("Bayern München Deutscher Meister");
    }
}
```

Die Ausgabe des Programms ist:

```
Frankfurter Allgemeine Schlagzeile:
an Fischer Fritz:  Neues Haushaltsloch von 30 Mrd. EURO
an Maier Hans:  Neues Haushaltsloch von 30 Mrd. EURO

Südkurier Schlagzeile:
an Fischer Fritz:  Bayern München Deutscher Meister
an Kunter Max:  Bayern München Deutscher Meister
```

Man beachte, dass mit der Programmzeile

```
NachrichtenEmpfaenger[] senke = new NachrichtenEmpfaenger [3];
```

ein Array von Referenzen vom Typ einer Schnittstelle angelegt wird (siehe Bild 14-7), wobei diese Referenzen auf Instanzen zeigen können, deren Klassen die Schnittstelle `NachrichtenEmpfaenger` implementieren. Hier Bild 14-7:

Bild 14-7 Array von Referenzen des Schnittstellentyps `NachrichtenEmpfaenger`

Da die Klasse `Person` die Schnittstelle `NachrichtenEmpfaenger` implementiert, können Referenzen auf Instanzen der Klasse `Person` den Komponenten des Schnittstellen-Arrays `senke` als Elemente zugewiesen werden. Nach den folgenden Anweisungen ist das Array gefüllt:

```
senke [0] = new Person ("Fischer", "Fritz");
senke [1] = new Person ("Maier", "Hans");
senke [2] = new Person ("Kunter", "Max");
```

Das folgende Bild demonstriert das Speicherabbild:

Bild 14-8 Referenzen im Array zeigen auf konkrete Objekte

14.4.3 Typsicherheit von Schnittstellen

Bisher wurde es immer als großer Vorteil angesehen, dass einer Methode, die als formalen Übergabeparameter eine Referenz vom Typ `Object` hat, ein Objekt einer beliebigen Klasse übergeben werden kann. Dies funktioniert deshalb, weil die gemeinsame Basisklasse aller Klassen in Java die Klasse `Object` ist. Genau diese Vorgehensweise kann unter Umständen zu Laufzeitfehlern führen. Betrachtet werden soll hierzu die bekannte Methode `anmelden()` der Schnittstelle `NachrichtenQuelle`. Diese Methode hat einen Übergabeparameter des Schnittstellentyps `NachrichtenEmpfaenger`:

```
interface NachrichtenQuelle
{
   public boolean anmelden (NachrichtenEmpfaenger empf);
   public void sendeNachricht (String nachricht);
}
```

An dieser Stelle könnte man auch einen Übergabeparameter vom Typ `Object` verwenden, wie es das folgende Beispiel zeigt:

```
interface NachrichtenQuelle
{
   public boolean anmelden (Object empf);
   public void sendeNachricht (String nachricht);
}
```

Die Implementierung der Methode `anmelden()` könnte dann wie folgt aussehen:

```
public boolean anmelden (Object empf)
{
   if (anzahlEmpfaenger < arr.length)
   {
      // Jetzt muss beim Zuweisen der übergebenen Referenz an das
      // Array arr vom Typ NachrichtenEmpfaenger ein expliziter Cast
      // durchgeführt werden.
```

```
        arr [anzahlEmpfaenger ++] = (NachrichtenEmpfaenger) empf;
        return true;
    }
    return false;
}
```

Von der Funktionalität her betrachtet, ist es egal, welche Variante verwendet wird – beide funktionieren gleich gut. Aber man sollte auch daran denken, dass man der jetzigen Methode `anmelden()` nicht mehr ansieht, dass es für den Übergabeparameter zwingend erforderlich ist, die Schnittstelle `NachrichtenEmpfaenger` zu implementieren. Wird eine Referenz auf ein anderes beliebiges Objekt übergeben, dessen Klasse diese Schnittstelle nicht implementiert, so kann dies erst zur Laufzeit festgestellt werden, wenn die Typumwandlung von `Object` nach `NachrichtenEmpfaenger` fehlschlägt und eine Exception vom Typ `ClassCastException` geworfen wird. Dies ist sehr nachteilig, da der Compiler keine Möglichkeit hat, diesen Fehler aufzudecken.

Wird ein Schnittstellentyp als Übergabeparameter angegeben, so können nur Referenzen auf Objekte übergeben werden, deren Klassen auch tatsächlich diese Schnittstelle implementieren. Werden andere Parameter übergeben, so meldet schon der Compiler einen Fehler.

Deshalb gilt stets: Wenn bei einem Referenztyp als Übergabeparameter nicht jede beliebige Referenz übergeben werden können soll, so ist davon abzusehen, den Referenztyp `Object` als Übergabeparameter zu verwenden.

Schnittstellen bieten ein elegantes Mittel zur Prüfung durch den Compiler, ob der Anwender den richtigen Typ übergeben hat. Deshalb sollte bei Übergabeparametern stets geprüft werden, ob durch Einsatz eines Schnittstellentyps eine höhere Typsicherheit erreicht werden kann.

14.4.4 Implementieren von mehreren Schnittstellen

Eine Klasse kann nicht nur eine einzige, sondern kann **beliebig viele Schnittstellen implementieren**.

Syntaktisch gibt die Klasse dies mit dem Schlüsselwort `implements` an, gefolgt von einer Liste von gültigen Schnittstellennamen, die durch Kommata getrennt sind. Im folgenden Beispiel ist eine Klasse `Vermittler` aufgeführt, die sowohl die Schnittstelle `NachrichtenQuelle` als auch die Schnittstelle `NachrichtenEmpfaenger` implementiert. Bild 14-9 zeigt dies grafisch:

Bild 14-9 Die Klasse `Vermittler` implementiert zwei Schnittstellen

```
// Datei: Vermittler.java

public class Vermittler implements NachrichtenEmpfaenger,
                                   NachrichtenQuelle
{
   private NachrichtenEmpfaenger[] arr;
   private int anzahlEmpfaenger = 0;

   public Vermittler (int maxAnzahlEmpfaenger)
   {
      arr = new NachrichtenEmpfaenger [maxAnzahlEmpfaenger];
   }

   public boolean anmelden (NachrichtenEmpfaenger empf)
   {
      if (anzahlEmpfaenger < arr.length)
      {
          arr [anzahlEmpfaenger++] = empf;
          return true;
      }
      return false;
   }

   public void sendeNachricht (String nachricht)
   {
      // Alle angemeldeten Nachrichtenempfänger
      // werden benachrichtigt
      for (int i = 0; i < anzahlEmpfaenger; i++)
      {
         arr [i].empfangeNachricht (nachricht);
      }
   }

   public void empfangeNachricht (String nachricht)
   {
      sendeNachricht (nachricht);
   }
}
```

Ein Objekt der Klasse `Vermittler` kann sich nun bei einem Objekt der Klasse `Zeitung` als Nachrichtenempfänger über dessen Methode `anmelden()` registrieren lassen. Objekte der Klasse `Person` können sich wiederum bei einem Objekt der

Klasse `Vermittler` – mit Hilfe der Instanzmethode `anmelden()` der Klasse `Vermittler` – registrieren und erhalten somit automatisch alle Nachrichten von allen Zeitungen. Damit muss sich eine Person nicht mehr bei allen Zeitungen einzeln anmelden, sondern gibt die Adresse allein dem Vermittler bekannt, der alle Nachrichten von jeder Zeitung weiterleitet. Sicherlich ist dies nicht eine allzu realistische Variante, denn da nun alle Personen alle Zeitungsnachrichten erhalten, werden diese bald merken, dass sie zwar hervorragend informiert werden, aber dass Zeitungen eben auch Geld kosten. Es ist hierzu folgende Variante denkbar: Das Objekt der Klasse `Vermittler` bietet eine Anmeldeschnittstelle, die es ermöglicht, den Typ der Zeitung, die man abonnieren möchte, mit anzugeben. Damit hat jede Person die Möglichkeit, sich über das Objekt der Klasse `Vermittler` gezielt bei einer oder mehreren Zeitungen anzumelden. In dem vorliegenden Beispiel wird aber aus Aufwandsgründen nur die vereinfachte Variante betrachtet, in der eine Person, die sich über das Objekt vom Typ `Vermittler` anmeldet, alle Nachrichten aller Zeitungen erhält.

Die folgende Testklasse veranschaulicht diese Variante:

```
// Datei: VermittlerTest.java
public class VermittlerTest
{
   public static void main (String[] args)
   {
      NachrichtenQuelle z1 =
                     new Zeitung ("-Frankfurter Allgemeine-", 3);
      NachrichtenQuelle z2 = new Zeitung ("-Südkurier-", 3);
      Vermittler mittler = new Vermittler (3);

      // Vermittler tritt in Gestalt des Nachrichtenempfängers auf
      z1.anmelden (mittler);
      z2.anmelden (mittler);
      // Vermittler tritt in der Gestalt der NachrichtenQuelle auf
      mittler.anmelden (new Person ("Fischer", "Fritz"));
      mittler.anmelden (new Person ("Maier", "Hans"));
      mittler.anmelden (new Person ("Kunter", "Max"));

      System.out.println ("Frankfurter Allgemeine Schlagzeile:");
      z1.sendeNachricht ("Neues Haushaltsloch von 30 Mrd. EURO");
      System.out.println ();
      System.out.println ("Südkurier Schlagzeile:");
      z2.sendeNachricht ("Bayern München Deutscher Meister");
   }
}
```

Die Ausgabe des Programms ist:

```
Frankfurter Allgemeine Schlagzeile:
an Fischer Fritz:  Neues Haushaltsloch von 30 Mrd. EURO
an Maier Hans:  Neues Haushaltsloch von 30 Mrd. EURO
an Kunter Max:  Neues Haushaltsloch von 30 Mrd. EURO

Südkurier Schlagzeile:
an Fischer Fritz:  Bayern München Deutscher Meister
an Maier Hans:  Bayern München Deutscher Meister
an Kunter Max:  Bayern München Deutscher Meister
```

Falls es noch nicht aufgefallen ist, **unsere Objekte haben das Reden untereinander gelernt**. Dies zeigt Bild 14-10:

Bild 14-10 Nachrichtenempfänger und Nachrichtenquellen reden miteinander

Einer Zeitung wird eine neue Nachricht zum Versenden gegeben, und diese schickt die Nachricht weiter an die angemeldeten Vermittler. Dabei weiß die Zeitung nichts davon, wie ein Vermittler mit der Nachricht weiter umgeht. Ein Vermittler benachrichtigt daraufhin alle ihm bekannten Nachrichtenempfänger. Überlässt man den Personen das Anmelden selbst, indem man z. B. im Konstruktor der Klasse `Person` die Anmeldung an einen übergebenen Vermittler vornimmt, so reden die Objekte in beiden Richtungen miteinander.

Ein Objekt der Klasse `Vermittler` tritt in zweierlei Gestalt auf:

- als Objekt des Typs `NachrichtenEmpfaenger`
- und als Objekt des Typs `NachrichtenQuelle`.

Durch die Implementierung einer Schnittstelle erhält ein Objekt die Möglichkeit, sich zusätzlich wie ein spezieller Schnittstellentyp zu verhalten. Es wird also ein zusätzliches Verhalten bzw. eine **zusätzliche Aufrufschnittstelle** implementiert. Jedes Objekt, dessen Klasse eine Schnittstelle implementiert, kann sich auch wie ein Typ der implementierten Schnittstelle verhalten.

Mit dem Schlüsselwort **`implements`** können **mehrere Schnittstellen** in einer Klasse **implementiert** werden. Dabei werden die Namen der zu implementierenden Schnittstellen durch Kommata getrennt hinter `implements` aufgeführt. Damit erhalten Objekte einer Klasse, die mehrere Schnittstellen implementiert, die Fähigkeit, in der Gestalt von mehreren Typen aufzutreten. Eine **Instanz kann als Referenztyp der Klasse oder als Referenztyp jeder implementierten Schnittstelle auftreten**.

Die folgenden Probleme können beim gleichzeitigen Implementieren mehrerer Schnittstellen auftreten.

Die zu implementierenden Schnittstellen:

- beinhalten Methoden mit gleicher Signatur und gleichem Rückgabewert, d. h. mit demselben Methodenkopf,
- beinhalten Konstanten mit demselben Namen,
- enthalten Methoden, die sich nur darin unterscheiden, dass sie unterschiedliche Exceptions werfen,
- beinhalten Methoden, die bis auf den Rückgabewert gleich sind.

Die soeben genannten Problemfälle werden im Folgenden diskutiert:

- **Zwei zu implementierende Schnittstellen haben denselben Methodenkopf**

  In diesem Fall wird diese Methode nur ein einziges Mal in der Klasse implementiert. Sie kann nicht für jede Schnittstelle getrennt implementiert werden. Auch die Verträge der beiden Methoden müssen übereinstimmen.

- **Zwei zu implementierende Schnittstellen haben Konstanten mit exakt demselben Namen**

  Das folgende **Beispiel** zeigt einen solchen Fall. Die Konstante `VAR1` ist sowohl in der Schnittstelle `Schnitt1` als auch in der Schnittstelle `Schnitt2` und zusätzlich noch in der Klasse `KonstantenTest` vorhanden:

```
// Datei: KonstantenTest.java

interface Schnitt1
{
   public static final int VAR1 = 1;
   public static final int VAR2 = 2;
}

interface Schnitt2
{
   public static final int VAR1 = 3;
   public static final int VAR3 = 4;
}

public class KonstantenTest implements Schnitt1, Schnitt2
{
   private static final int VAR1 = 9;

   public static void main (String[] args)
   {
      System.out.println (VAR1); // VAR1 der Klasse KonstantenTest
      System.out.println (VAR2);
      System.out.println (VAR3);
      System.out.println (Schnitt1.VAR1);
      System.out.println (Schnitt2.VAR1);
   }
}
```

Die Ausgabe des Programms ist:

```
9
2
4
1
3
```

Auf die doppelt vorhandenen Schnittstellenkonstanten kann nur über die Angabe des Schnittstellennamens, z. B. `Schnitt1.VAR1`, zugegriffen werden.

Existieren Konstanten mit demselben Namen in verschiedenen Schnittstellen oder sind sie von Datenfeldern der Klasse verdeckt, so müssen diese Konstanten über den qualifizierten Namen mit Angabe des Schnittstellennamens angesprochen werden.

- **Zwei zu implementierende Schnittstellen haben zwei Methoden, die bis auf die Exceptions in der `throws`-Klausel identisch sind.**

Im folgenden **Beispiel** werden zwei Client-Programme `Client1` und `Client2` gezeigt, wobei das Programm `Client1` mit Referenzen auf Objekte vom Typ `Eins` und das Programm `Client2` mit Referenzen auf Objekte vom Typ `Zwei` arbeitet. In den Schnittstellen `Eins` und `Zwei` soll jeweils eine Methode deklariert sein, die sich nur durch den Typ der Exception in der `throws`-Klausel unterscheiden. Das Client-Programm `Client1` erwartet Ausnahmen vom Typ `Exception` und das Client-Programm `Client2` erwartet Ausnahmen vom Typ `MyException`. Solange die beiden Schnittstellen in getrennten Klassen implementiert werden, gibt es kein Problem. Das folgende Bild visualisiert die Benutzung der Schnittstelle durch die Client-Programme:

Bild 14-11 Client-Programme `Client1` und `Client2`, die Schnittstellen benutzen

Sollen beide Schnittstellen in einer gemeinsamen Klasse implementiert werden (siehe Bild 14-13), so ist dies nur dann möglich, wenn die beiden Ausnahmen zueinander in einer Vererbungshierarchie stehen (siehe Bild 14-12) und wenn die implementierte Methode nur Ausnahmen vom Typ der Klasse wirft, die in der Vererbungshierarchie weiter unten steht. Das folgende Bild zeigt die Vererbungshierarchie der betrachteten Exceptions:

Bild 14-12 Vererbungshierarchie der Ausnahmen `Exception` und `MyException`

Das folgende Bild visualisiert die Implementierung der Schnittstellen `Eins` und `Zwei` in der Klasse `Server`:

Bild 14-13 Implementieren der Schnittstellen `Eins` und `Zwei` in der Klasse `Server`

Das Client-Programm `Client1` erwartet Ausnahmen vom Typ `Exception` und das Client-Programm `Client2` erwartet Ausnahmen vom Typ `MyException`. Bekommt das Client-Programm `Client1` eine Ausnahme vom Typ `MyException`, so ist dies auch in Ordnung, da ein Sohnobjekt immer an die Stelle des Vaters treten kann. Bekommt aber das Client-Programm `Client2`, das ja Ausnahmen vom Typ `MyException` erwartet, nur eine Ausnahme vom Typ `Exception`, so kommt es zu einer Fehlersituation, da das Client-Programm `Client2` mehr erwartet. Wirft also die Klasse `Server` eine Ausnahme vom Typ `MyException`, so sind beide Client-Programme `Client1` und `Client2` zufrieden.

Enthalten zwei Schnittstellen Methoden, die sich nur durch den Typ der Exception in der `throws`-Klausel unterscheiden, so können diese Schnittstellen nur dann von einer Klasse gemeinsam implementiert werden, wenn die beiden Ausnahmen zueinander in einer Vererbungshierarchie stehen und von der implementierten Methode nur die Exception geworfen wird, die in der Vererbungshierarchie weiter unten steht.

Hier nun das entsprechende **Programmbeispiel**:

```
// Datei: MyException.java
class MyException extends Exception
{
   MyException()
```

```
   {
      super ("MyException-Fehler!!");
   }
}
```

Hier die beiden Schnittstellen:

```
// Datei: Eins.java
public interface Eins
{
   public void methode() throws Exception;
}

// Datei: Zwei.java
public interface Zwei
{
   public void methode() throws MyException;
}
```

Hier die Klasse `Server`:

```
// Datei: Server.java
public class Server implements Eins, Zwei
{
   // Wirft die Methode methode() eine Exception vom Typ
   // MyException, so sind Client-Programme, welche die
   // Schnittstelle Eins verwenden, als auch Client-Programme,
   // welche die Schnittstelle Zwei verwenden, zufrieden.
   public void methode() throws MyException
   {
      throw new MyException();
   }
}
```

Im Folgenden werden die beiden Client-Programme `Client1` und `Client2` vorgestellt:

```
// Datei: Client1.java
class Client1
{
   public static void main (String[] args)
   {
      // Client1 arbeitet mit einer Referenzvariablen vom Typ
      // Eins. Aus deren Sicht ist die Methode bekannt,
      // die Ausnahmen vom Typ Exception wirft.
      Eins x = new Server();

      try
      {
         x.methode();
      }
      // Client1 ist auch mit Exceptions vom Typ MyException
      // zufrieden.
      catch (Exception e)
      {
         System.out.println (e.getMessage());
      }
```

```
   }
}

// Datei: Client2.java
class Client2
{
   public static void main (String[] args)
   {
      // Client2 arbeitet mit einer Referenzvariablen vom Typ
      // Zwei. Aus deren Sicht wirft die Methode methode()
      // eine Exception vom Typ MyException.
      Zwei x = new Server();

      try
      {
         x.methode();
      }
      // Client2 arbeitet sowieso mit Exceptions vom Typ
      // MyException. Hier gibt es also auch keine Probleme.
      catch (MyException e)
      {
         System.out.println (e.getMessage());
      }
   }
}
```

Die Ausgabe des Programms `Client1` ist:

```
MyException-Fehler!!
```

Die Ausgabe des Programms `Client2` ist:

```
MyException-Fehler!!
```

- **Zwei zu implementierende Schnittstellen besitzen Methoden, die sich nur in ihrem Rückgabewert unterscheiden**

  In diesem Fall können die Schnittstellen nicht gemeinsam implementiert werden, da die Methoden anhand des Rückgabewertes nicht unterschieden werden können (siehe Kapitel 9.2.3). Hier ein **Beispiel**:

```
interface ReturnObject
{
   public Object gebeWert();
}

interface ReturnInteger
{
   public Integer gebeWert();
}

class Implementierung implements ReturnObject//bzw. ReturnInteger
{
   // Beide Methoden in der gleichen Klasse zu implementieren
   // funktioniert nicht, da sich die Methoden nur anhand ihres
   // Rückgabetyps unterscheiden und somit dem Compiler keine
```

```
      // Möglichkeit zur Differenzierung ermöglichen. Denn in Java
      // ist es nicht erforderlich, den Rückgabewert eines Methode-
      // aufrufs abzuholen.

      public Object gebeWert()
      {
         return new Object();
      }

      // public Integer gebeWert()
      // {
      //    return new Integer();
      // }
   }
```

14.4.5 Vererbung von Schnittstellen

Einfachvererbung bei Schnittstellen

Schnittstellen besitzen – genauso wie Klassen – die Möglichkeit, mit dem Schlüsselwort `extends` eine schon vorhandene Schnittstelle zu erweitern.

Hierzu ein **Beispiel**:

```
// Datei: Einfach.java

interface NachrichtenQuelle
{
   public int SPORT    = 0;
   public int POLITIK  = 1;
   public int KULTUR   = 2;
   public int ANZEIGEN = 3;
   public int GESAMT   = 4;
   public boolean anmelden (NachrichtenEmpfaenger empf, int typ);
   public void sendeNachricht (String nachricht);
}

interface Vermittler extends NachrichtenQuelle
{
   public void empfangeNachricht (String nachricht);
}
```

Die Schnittstelle `Vermittler` erweitert die Schnittstelle `NachrichtenQuelle` um die Methode `empfangeNachricht()`. Wie bei der Vererbung von Klassen besitzt die Schnittstelle `Vermittler` neben den eigenen Elementen auch die von der Schnittstelle `NachrichtenQuelle` ererbten Elemente.

Mehrfachvererbung bei Schnittstellen

Im Gegensatz zur Einfachvererbung von Klassen ist in Java bei Schnittstellen eine Mehrfachvererbung erlaubt. Damit kann ein Schnittstelle nicht nur eine einzige Schnittstelle erweitern, sondern mehrere gleichzeitig.

Schnittstellen lassen im Gegensatz zu Klassen eine **Mehrfachvererbung** zu.

Hierzu ein **Beispiel**:

```
// Datei: Mehrfach.java

interface NachrichtenQuelle
{
   public int SPORT    = 0;
   public int POLITIK  = 1;
   public int KULTUR   = 2;
   public int ANZEIGEN = 3;
   public int GESAMT   = 4;

   public boolean anmelden (NachrichtenEmpfaenger empf, int typ);
   public void sendeNachricht (String nachricht);
}

interface NachrichtenEmpfaenger
{
   public void empfangeNachricht (String nachricht);
}

interface Vermittler extends NachrichtenQuelle, NachrichtenEmpfaenger
{

}
```

Dennoch ist die Mehrfachvererbung bei Schnittstellen von nicht allzu großer Bedeutung – **viel wichtiger ist die Möglichkeit, mehrere Schnittstellen gemeinsam in einer Klasse implementieren zu können**.

Damit können Instanzen dieser Klassen sich zusätzlich wie Typen aller implementierten Schnittstellen verhalten. Dies wurde bereits in Kapitel 14.4.4 gezeigt.

14.5 Vergleich Schnittstelle und abstrakte Basisklasse

Abstrakte Basisklassen und Schnittstellen sind einander ähnlich. Beide sind ein Mittel zur Abstraktion. Im Folgenden sollen die Übereinstimmungen und Gegensätze aufgezeigt werden.

Abstrakte Basisklassen können Variablen, Konstanten, implementierte und abstrakte Methoden enthalten. Schnittstellen können nur Konstanten und abstrakte Methoden enthalten.

Für Klassen stellt Java nur den Mechanismus der Einfachvererbung bereit. Es ist nicht möglich, von mehreren Klassen zu erben. Dies zeigt das folgende Bild:

Bild 14-14 Bei Klassen ist keine Mehrfachvererbung erlaubt

Eine Klasse kann aber mehrere Schnittstellen implementieren, wie Bild 14-15 zeigt:

Bild 14-15 Eine Klasse kann mehrere Schnittstellen implementieren

Eine Klasse kann auch eine vorhandene abstrakte Basisklasse erweitern bzw. deren leere Methodenrümpfe ausprogrammieren und gleichzeitig eine oder mehrere Schnittstellen implementieren, wie folgendes Bild zeigt:

Bild 14-16 Ableiten von einer Klasse und gleichzeitig mehrere Schnittstellen implementieren

Mit dem Mechanismus der Schnittstelle ist es damit möglich, mehrere Schnittstellen, die nur abstrakte Methoden und Konstanten enthalten, zu implementieren. Aber es ist keine Vererbung, sondern eine **Verfeinerung** im Sinne einer schrittweisen Verfeinerung, in deren Rahmen erst die Schnittstelle festgelegt wird und im zweiten Schritt dann die Implementierung erfolgt.

Sowohl eine **Unterklassenbildung** aus einer abstrakten Basisklasse im Rahmen der Vererbung als auch eine **Implementierung** einer Schnittstelle stellt **die Bildung eines Untertypen** dar. Ein Objekt einer Klasse – die eine Schnittstelle implementiert – ist vom Typ seiner Klasse und vom Typ der Schnittstelle.

Zwischen Klassen und Schnittstellen gibt es aber einen wichtigen Unterschied: Arbeitet man mit Klassen und dem Prinzip der Vererbung, so muss man die zu vererbende Information in die Wurzel des Klassenbaums bringen, wenn sie über alle Zweige nach unten vererbt werden soll. Die erbende Klasse und die abstrakte Klasse müssen also verwandt sein.

Das folgende Bild zeigt eine Vererbungshierarchie:

Bild 14-17 Vererbungsbaum mit einer abstrakten Basisklasse als Wurzel

Eine Schnittstelle kann von jeder beliebigen Klasse implementiert werden, ohne dass die Schnittstelle in den Klassenbaum eingeordnet werden muss. Die zu **implementierende Schnittstelle erfordert keine Verwandtschaft**. Sie kann implementiert werden, wo sie gebraucht wird.

Das folgende Bild zeigt, dass Schnittstellen überall in der Hierarchie der Klassen eingefügt werden können:

Bild 14-18 Gemischte Hierarchie mit Klassen und Schnittstellen

Eine in einer Vaterklasse implementierte Schnittstelle wird an abgeleitete Sohnklassen weitervererbt. Somit kann sich ein Objekt der Sohnklasse wie ein Objekt der Vaterklasse verhalten und zusätzlich wie ein Objekt aller in der darüberliegenden

Hierarchie implementierten Schnittstellen. Mit Hilfe der folgenden Vererbungshierarchie soll der **Typbegriff** von Objekten erläutert werden:

Bild 14-19 Klassenhierarchie zur Diskussion des Typbegriffs

Ein Objekt einer Klasse kann in der Gestalt unterschiedlicher Typen auftreten. In der Tabelle 14-2 ist aufgelistet, von welchem Typ ein Objekt der Klasse `Klasse1`, `Klasse2` und der `Klasse3` ist.

Objekt der Klasse	ist vom Typ
`Klasse3`	`Klasse3`, `Klasse2`, `Klasse1`, `Schnittstelle3`, `Schnittstelle2`, `Schnittstelle1`
`Klasse2`	`Klasse2`, `Klasse1`, `Schnittstelle3`, `Schnitt-stelle2`, `Schnittstelle1`
`Klasse1`	`Klasse1`

Tabelle 14-2 Ein Objekt einer Klasse kann in Gestalt mehrerer Typen auftreten

14.6 Die Schnittstelle `Cloneable`

Klonen bedeutet nichts anderes, als eine **exakte Kopie** von etwas schon Existentem zu erstellen. Wenn ein Objekt geklont wird, erwartet man, dass man eine Referenz auf ein neues Objekt bekommt, dessen Datenfelder exakt die gleichen Werte haben wie die des Objektes, das als Vorlage zum Klonen benutzt wurde.

Im Folgenden soll der Unterschied zwischen den beiden Fällen:

- zwei Referenzen zeigen auf das gleiche Objekt,
- die zweite Referenz zeigt auf ein geklontes Objekt des ersten Objektes

erläutert werden. Betrachtet werden soll hierzu zunächst das folgende **Programmbeispiel**:

```
// Datei: KopieTest.java
class Kopie
{
   public int x;

   public Kopie (int x)
   {
      this.x = x;
   }
   public void print()
   {
      System.out.println ("x = " + x);
   }
}

public class KopieTest
{
   public static void main (String[] args)
   {
      Kopie ref1 = new Kopie (1);
      Kopie ref2 = ref1;

      System.out.print ("Wert über ref1: ");
      ref1.print();
      System.out.print ("Wert über ref2: ");
      ref2.print();
      ref1.x = 5;
      System.out.print ("Wert über ref1: ");
      ref1.print();
      System.out.print ("Wert über ref2: ");
      ref2.print();
   }
}
```

Die Ausgabe des Programms ist:

```
Wert über ref1: x = 1
Wert über ref2: x = 1
Wert über ref1: x = 5
Wert über ref2: x = 5
```

Das Ergebnis dürfte nicht verwundern. Da die Referenz `ref2` genau auf das gleiche Objekt zeigt wie die Referenz `ref1`, wird eine Datenänderung, egal ob sie über die Referenz `ref1` oder `ref2` erfolgt, immer am gleichen Objekt vorgenommen. Im folgenden Bild 14-20 ist dies grafisch zu sehen:

Bild 14-20 Zwei Referenzen, die auf das gleiche Objekt zeigen

Wenn ein Objekt geklont bzw. kopiert wird, erhält man zwei Objekte, deren Werte unabhängig voneinander verändert werden können. Bild 14-21 zeigt diese Situation:

Bild 14-21 Zwei Referenzen, die auf zwei verschiedene Objekte mit gleichem Inhalt zeigen

Das folgende Programm – es wird unterhalb des Programmcodes erläutert – erzeugt eine exakte Kopie und dient zur Erläuterung dieses **Beispiels**:

```
// Datei: CloneTest.java
class Kopie2 implements Cloneable
{
   public int x;

   public Kopie2 (int x)
   {
      this.x = x;
   }
   public void print()
   {
      System.out.println ("x = " + x);
   }

   // Überschreiben der Methode clone() der Klasse Object
   public Object clone() throws CloneNotSupportedException
   {
      // Mit super.clone() wird die überschriebene Methode clone()
      // der Klasse Object aufgerufen
      return super.clone();
   }
}

public class CloneTest
{
   public static void main (String[] args)
                           throws CloneNotSupportedException
   {
      Kopie2 ref1 = new Kopie2 (1);
      Kopie2 ref2 = (Kopie2) ref1.clone();
      System.out.print ("Wert über ref1: ");
      ref1.print();
      System.out.print ("Wert über ref2: ");
      ref2.print();
      ref1.x = 5;
      System.out.print ("Wert über ref1: ");
      ref1.print();
      System.out.print ("Wert über ref2: ");
      ref2.print();
   }
}
```

Die Ausgabe des Programms ist:

```
Wert über ref1: x = 1
Wert über ref2: x = 1
Wert über ref1: x = 5
Wert über ref2: x = 1
```

Das Ergebnis ist im Gegensatz zu dem vorherigen bemerkenswert. Die einzigen Änderungen, die in dem Programm vorgenommen wurden, sind fett hervorgehoben. Die Klasse `Kopie2` implementiert die Schnittstelle `Cloneable` des Pakets `java.lang` und überschreibt die Methode **`clone()`** der Klasse `Object` mit dem Zugriffsmodifikator `public`.

Man könnte zunächst vermuten, dass die Deklaration der Methode `clone()` in der Schnittstelle `Cloneable` enthalten ist. Dies ist aber nicht der Fall – die Schnittstelle `Cloneable` hat einen leeren Schnittstellenrumpf:

```
package java.lang;

public interface Cloneable
{
}
```

Was gewinnt aber eine Klasse hinzu, wenn sie eine solche Schnittstelle implementiert? Die Klasse gibt damit an, dass ihre Objekte kopierbar sein sollen! Ein **Überschreiben der Methode `clone()`** der Klasse `Object` mit der Sichtbarkeit `public` ist hierbei **zwingend erforderlich**.

Da die Methode `clone()` der Klasse `Object` den Zugriffsmodifikator `protected` hat, kann diese nur in abgeleiteten Klassen oder in Klassen, die sich im gleichen Paket befinden, aufgerufen werden.

Die Klasse `Kopie2` ist implizit von der Klasse `Object` abgeleitet. Daher kann in ihr selbst die von der Klasse `Object` geerbte Methode `clone()` aufgerufen werden. Würde in der Klasse `Kopie2` die Methode `clone()` der Klasse `Object` nicht mit der Sichtbarkeit `public` überschrieben, so würde der Aufruf `ref1.clone()` in der `main()`-Methode der Klasse `CloneTest` beim Kompilieren den folgenden Fehler erzeugen:

```
CloneTest.java:23: clone() has protected access in java.lang.Object
      Kopie2 ref2 = (Kopie2) ref1.clone();
```

Würde hingegen die Klasse `Kopie2` das Interface `Cloneable` nicht implementieren, so ließe sich das Programm zwar ohne Fehler kompilieren, beim Aufruf der Methode `Object.clone()` würde zur Laufzeit aber eine Exception vom Typ `CloneNotSupportedException` geworfen.

Liefert der Ausdruck

```
ref instanceof Cloneable
```

`true` zurück, so muss es auch möglich sein, `ref.clone()` aufzurufen. Dies ist aber nur dann möglich, wenn die Methode `clone()` in der Klasse des Objektes, auf das `ref` zeigt, mit dem Zugriffsmodifikator `public` überschrieben wird.

Ob ein Objekt kopierbar ist oder nicht, kann folgendermaßen überprüft werden:

```
if (ref instanceof Cloneable)
{
   // Kopie möglich
}
```

Dadurch, dass explizit bei einer Klasse angegeben werden muss, dass diese kopierbar ist, kann verhindert werden, dass Objekte von Klassen kopiert werden können, für die das gar nicht vorgesehen war, und für die die Kopierfunktionalität deshalb auch nicht richtig implementiert worden ist.

Man muss also

- die Methode `clone()` erstens durch Überschreiben mit der Sichtbarkeit `public` zugänglich machen und
- zweitens das Interface `Cloneable` implementieren, da die Methode `Object.clone()` überprüft, ob das klonende Objekt vom Typ `Cloneable` ist. [langer]

Schnittstellen, die gar keine Methoden enthalten, werden auch **Marker-Interfaces** genannt. Siehe hierzu Kapitel 14.7.

Die Methode `clone()` der Klasse `Object` sieht folgendermaßen aus:

```
protected Object clone() throws CloneNotSupportedException
{
   . . . . // Die Implementierung soll hier nicht betrachtet werden
}
```

Die Aufgabe der Methode `clone()` der Klasse `Object` besteht darin, eine Eins-zu-Eins-Kopie des Objektes zu erstellen, für das sie aufgerufen wird.

Mit anderen Worten:

Die Methode **`clone()`** erzeugt ein neues Objekt und belegt die Datenfelder mit den exakt gleichen Werten wie das Objekt, für das die Methode aufgerufen wird. Es wird eine Referenz vom Typ `Object` auf das neue Objekt zurückgegeben. Diese muss nur noch in den richtigen Typ gecastet werden.

Alle Objekte besitzen also schon eine Kopierfähigkeit. Ihre Klassen müssen jedoch das Interface `Cloneable` implementieren, damit die Methode `clone()` der Klasse `Object` verwendet werden kann.

Anzumerken ist, dass die Methode `Object.clone()` nicht in Java implementiert ist, sondern als native Funktion.

Flache Kopie

Im vorliegenden Beispiel war es ausreichend, in der Methode `clone()` der Klasse `Kopie2` einfach die Methode `clone()` der Klasse `Object` aufzurufen.

Die Methode `clone()` der Klasse `Object` erzeugt eine Eins-zu-Eins-Kopie aller Datenfeldwerte.

Sobald die Datenfelder eines Objektes nicht mehr nur aus primitiven Datentypen bestehen, muss in der Methode `clone()` mehr erfolgen als nur der Aufruf der Methode `clone()` der Basisklasse `Object`.

Denn wenn ein Datenfeld eine Referenz ist, so wird von der Methode `clone()` der Klasse `Object` nur die Referenz kopiert und kein neues Objekt angelegt. Es handelt sich um eine sogenannte **"flache" Kopie**.

Das folgende Bild zeigt diese Problematik:

Bild 14-22 "flache" Kopie

Wird von einem `Kreiseck`-Objekt eine flache Kopie erzeugt, so werden die Werte der Referenzvariablen `refKreis` und `refEck` einfach kopiert. Das heißt, es werden keine neuen Objekte vom Typ `Kreis` und `Eck` erzeugt.

Tiefe Kopie

Bei der **"tiefen" Kopie** hingegen entstehen neue Objekte vom Typ `Kreis` und `Eck`, auf welche dann die Referenzvariablen `refKreis` und `refEck` des neuen Objektes vom Typ `Kreiseck` zeigen. Die Instanzvariablen der neuen Objekte vom Typ `Kreis` und vom Typ `Eck` werden mit den Werten der kopierten Objekte initialisiert.

Bild 14-23 zeigt den Sachverhalt bei einer "tiefen" Kopie:

Bild 14-23 "tiefe" Kopie

Beispielprogramm

Das unten stehende Beispiel demonstriert die Realisierung einer **"tiefen" Kopie**. Die Klasse `MyClass` enthält eine Referenz auf die Klasse `Mini` (siehe auch Bild 14-24 und Bild 14-25). Beim Anlegen eines Objektes vom Typ `MyClass` wird auch die Klasse `Mini` instanziiert. Beim Kopieren über die überschriebene Methode `clone()` wird auch das Objekt der Klasse `Mini` mit kopiert.

Die Datenfelder der Objekte, auf welche die Referenzen `orig` und `kopie` zeigen, können daher völlig unabhängig voneinander verändert werden.

Hier das bereits erwähnte Beispiel:

```
// Datei: Clone2.java

class Mini implements Cloneable
{
   public int x = 1;
   public int y = 1;

   public Object clone() throws CloneNotSupportedException
   {
      return super.clone();
   }
}
```

```
class MyClass implements Cloneable
{
   public int var = 1;
   public Mini ref = new Mini();

   public Object clone() throws CloneNotSupportedException
   {
      MyClass tmp = (MyClass) super.clone(); // Flache Kopie
      tmp.ref = (Mini) ref.clone(); // Kopieren des Objektes, auf
      return tmp;                   // das die Referenz zeigt
   }
}

public class Clone2
{
   public static void main (String[] args) throws
                                      CloneNotSupportedException
   {
      MyClass orig = new MyClass();
      MyClass kopie = (MyClass) orig.clone(); // Kopie erstellen
      kopie.var = 2;      // Datenfeld der Kopie ändern
      kopie.ref.x = 2;    // Datenfeld der Kopie ändern
      System.out.println ("Original:");
      System.out.println ("var = " + orig.var);
      System.out.println ("Mini.x = " + orig.ref.x +
                          "    Mini.y = " + orig.ref.y);
      System.out.println();
      System.out.println ("Kopie:");
      System.out.println ("var = " + kopie.var);
      System.out.println ("Mini.x = " + kopie.ref.x +
                          "    Mini.y = " + kopie.ref.y);
   }
}
```

Die Ausgabe des Programms ist:

```
Original:
var = 1
Mini.x = 1    Mini.y = 1

Kopie:
var = 2
Mini.x = 2    Mini.y = 1
```

Die folgenden Bilder zeigen nochmals den Vorgang des Klonens für das obige Programm.

Das erste Bild zeigt den Zustand der Objekte nach der Programmzeile

`MyClass tmp = (MyClass) super.clone();  // oben fett`

in der Methode `clone()` der Klasse `MyClass`:

Bild 14-24 Objektzustand nach dem Aufruf `super.clone()`

Nach der Ausführung der folgenden Codezeile

```
tmp.ref = (Mini) ref.clone();  //Seite vorher, auch fett
```

in der Methode `clone()` sehen die Verhältnisse folgendermaßen aus:

Bild 14-25 Objektzustände nach Aufruf der Methode `clone()` *der Klasse* `Mini`

14.7 Marker Interfaces

Marker Interfaces müssen nicht zwingend eine Methode enthalten.

Marker Interfaces dienen dazu, dass Klassen in zwei Mengen aufgeteilt werden, in diejenigen Klassen, welche eine bestimmte Schnittstelle implementieren, und in diejenigen Klassen, welche diese Schnittstelle nicht implementieren.

Mit dem `instanceof`-Operator kann dann festgestellt werden, zu welcher der beiden Mengen ein Objekt gehört bzw. ob eine Klasse ein bestimmtes Verhalten besitzt oder nicht. Beispielsweise implementiert eine Klasse das `Cloneable`-Interface, um der Methode `Object.clone()` zu signalisieren, dass Instanzen dieser Klasse feldweise kopiert werden können.

Wichtige Marker Interfaces sind z. B. `java.lang.Cloneable` (siehe Kapitel 14.6) und `java.io.Serializable` (siehe Kapitel 19.7.1).

14.8 Zusammenfassung

Kapitel 14 befasst sich mit den Schnittstellen (Interfaces) in Java. Eine Schnittstelle in Java ist ein Sprachmittel für den Entwurf und enthält eine Reihe von Methodenköpfen. Darüber hinaus können auch Konstanten in Schnittstellen definiert werden. Eine Klasse beinhaltet dagegen – sofern sie nicht abstrakt ist – den Entwurf und die Implementierung, d. h. auch die Methodenrümpfe. Eine Schnittstelle beinhaltet keinen Methodenrumpf.

Kapitel **14.1** geht auf die Trennung von Spezifikation und Implementierung ein.

Mit dem Sprachmittel `interface` können Schnittstellen in Java formuliert werden. Dies wird am Anfang des Kapitels **14.1** gezeigt.

In Kapitel **14.2** wird ein weiterführendes Beispiel diskutiert.

Im Beispiel wird gezeigt, dass wenn als formaler Übergabeparameter ein Schnittstellentyp angegeben wird, eine Referenz auf ein Objekt, dessen Klasse diese Schnittstelle implementiert, als aktueller Parameter übergeben werden kann. Referenzen auf Objekte eines anderen Typs werden vom Compiler abgelehnt.

Wie eine Schnittstelle aufgebaut ist, wird in Kapitel **14.3** gezeigt.

Ähnlich wie bei einer Klassendefinition besteht auch die Schnittstellendefinition aus zwei Teilen:

- der Schnittstellendeklaration
- und dem Schnittstellenkörper mit Konstantendefinitionen und Methodendeklarationen.

Die Schnittstellendeklaration besteht aus dem Zugriffsmodifikator, dem Schlüsselwort `interface` und optional dem Schlüsselwort `extends`, falls von weiteren Schnittstellen abgeleitet werden soll.

Mit `public` deklarierte Schnittstellen können – genauso wie Klassen – mittels der `import`-Vereinbarung in anderen Paketen sichtbar gemacht werden. Schnittstellen, die nicht mit `public` deklariert sind, sind default und damit nur im eigenen Paket sichtbar.

Schnittstellen können mit `extends` von anderen Schnittstellen abgeleitet werden. Mit anderen Worten, es ist möglich, eigene Schnittstellenhierarchien aufzubauen. Klassen können dagegen mit `extends` nicht von Schnittstellen abgeleitet werden. Sie können jedoch beliebig viele Schnittstellen mit `implements` implementieren.

Der Schnittstellenkörper enthält:

- Konstantendefinitionen
- und Methodendeklarationen.

Bei der Methodendeklaration ist die explizite Angabe von `public` und `abstract` optional. Fehlen diese Schlüsselwörter, so werden sie automatisch vom Compiler

eingefügt. Methoden in Schnittstellen besitzen – da sie `abstract` sind – keinen Methodenrumpf.

Jede Konstantendefinition in einer Schnittstelle muss einen Initialisierungsausdruck besitzen, da es ja keine Methodenrümpfe gibt, in denen eine Initialisierung stattfinden könnte.

Kapitel **14.4** zeigt die Verwendung von Schnittstellen und die Besonderheiten, auf die man achten muss.

Eine Klasse gibt mit dem Schlüsselwort `implements` an, welche Schnittstellen sie implementiert. Implementiert eine Klasse eine Schnittstelle, so muss sie alle Methoden der Schnittstelle implementieren, wenn sie instanziiert werden soll – d. h. wenn von ihr Objekte geschaffen werden sollen. Ansonsten muss die Klasse als abstrakt deklariert werden und ist damit nicht instanziierbar.

Es kann durch Schnittstellen keine Funktionalität geerbt werden, da Schnittstellen keine Methodenimplementierung beinhalten.

Ein Programmierer hat bei der Implementierung einer Schnittstellenmethode darauf zu achten, dass er den Vertrag der deklarierten Methode erfüllt.

Eine Schnittstelle kann als Datentyp verwendet werden. Eine Schnittstelle ist ein Referenztyp. Von ihm können Referenzvariablen gebildet werden, die auf Objekte zeigen, deren Klassen die Schnittstelle implementieren. Es ist damit auch möglich, Arrays von Schnittstellentypen anzulegen und diese Arrays mit Referenzen auf Objekte zu füllen, deren Klassen die Schnittstelle implementieren. Schnittstellen sind typsicher.

Um Objekte beliebiger Klassen als aktuelle Übergabeparameter zu akzeptieren, kann man in Java die Klasse `Object` als formalen Parameter verwenden, die gemeinsame Basisklasse aller Objekte in Java. Das ist aber anfällig für Laufzeitfehler, da der Programmierer hier vom Compiler keine Hilfe zur Überprüfung des Typs eines aktuellen Parameters erhält. Der Compiler prüft nur, ob die übergebene Referenz vom Typ `Object` ist (was in Java immer der Fall ist). Durch Verwendung von Schnittstellentypen als formale Parameter kann man die aktuellen Parameter vom Compiler prüfen lassen und kann sich sicher sein, dass das übergebene Objekt typgerecht ist. Schnittstellen bieten damit ein elegantes Mittel zur Prüfung durch den Compiler, ob der Anwender den richtigen Typ übergeben hat. Deshalb sollte bei Übergabeparametern stets geprüft werden, ob durch Einsatz eines Schnittstellentyps eine höhere Typsicherheit erreicht werden kann.

Mit dem Schlüsselwort `implements` können mehrere Schnittstellen in einer Klasse implementiert werden. Dabei werden die Namen der zu implementierenden Schnittstellen durch Kommata getrennt hinter `implements` aufgeführt. Damit erhalten Objekte einer Klasse, die mehrere Schnittstellen implementiert, die Fähigkeit, in der Gestalt von mehreren Typen aufzutreten. Eine Instanz kann als Referenztyp der Klasse oder als Referenztyp jeder implementierten Schnittstelle auftreten.

Existieren Konstanten mit demselben Namen in verschiedenen Schnittstellen oder sind sie von Datenfeldern der Klasse verdeckt, so müssen diese Konstanten über

den qualifizierten Namen mit Angabe des Schnittstellennamens angesprochen werden.

Enthalten zwei Schnittstellen Methoden, die sich nur durch den Typ der Exception in der `throws`-Klausel unterscheiden, so können diese Schnittstellen nur dann von einer Klasse gemeinsam implementiert werden, wenn die beiden geworfenen Ausnahmen zueinander in einer Vererbungshierarchie stehen und von der implementierten Methode nur diejenigen Exception geworfen wird, die in der Vererbungshierarchie weiter unten steht.

Bei Schnittstellen ist sowohl die Einfachvererbung als auch die Mehrfachvererbung möglich. Im Gegensatz zu Klassen erlaubt Java bei Schnittstellen die Mehrfachvererbung. Das bedeutet, dass eine Klasse mehrere Schnittstellen implementieren kann und dass eine Schnittstelle von mehreren anderen Schnittstellen ableiten kann.

In Kapitel **14.5** wird ein Vergleich zwischen Schnittstelle und abstrakter Basisklasse gezogen.

Abstrakte Basisklassen können Variablen, Konstanten, implementierte und abstrakte Methoden enthalten. Schnittstellen können nur Konstanten und abstrakte Methoden enthalten.

Eine Schnittstelle kann von jeder beliebigen Klasse implementiert werden, ohne dass die Schnittstelle in den Klassenbaum eingeordnet werden muss. Die zu implementierende Schnittstelle erfordert keine Verwandtschaft. Sie kann implementiert werden, wo sie gebraucht wird. Leitet man von einer abstrakten Basisklasse ab, wird die abgeleitete Klasse unterhalb der Basisklasse in die Vererbungshierarchie eingeordnet.

Kapitel **14.6** befasst sich mit der Schnittstelle `Cloneable`.

Wird eine Referenzvariable einer anderen Referenzvariable gleichen Typs zugewiesen, zeigen beide Variablen am Ende auf dasselbe Objekt im Speicher. Oft will man aber eine Kopie eines Objekts erzeugen, um unabhängig Änderungen auf der Kopie durchführen zu können, ohne dass das Original davon betroffen ist.

Die Schnittstelle `Cloneable` wird in Java dazu verwendet, zu markieren, ob ein Objekt klonbar ist oder nicht. Sie ist ein sogenanntes Marker Interface.

Dadurch, dass explizit bei einer Klasse angegeben werden muss, dass diese kopierbar ist, kann verhindert werden, dass Objekte von Klassen kopiert werden können, für die das gar nicht vorgesehen war, und für die die Kopierfunktionalität deshalb auch nicht richtig implementiert worden ist.

In der Klasse `Object` wird eine `clone()`-Methode zur Verfügung gestellt, die von allen Klassen geerbt wird.

Um eine Klasse klonbar zu machen, muss man

- die Methode `clone()` erstens durch Überschreiben mit der Sichtbarkeit `public` zugänglich machen und

- zweitens das Interface `Cloneable` implementieren, da die Methode `Object.clone()` überprüft, ob das klonende Objekt vom Typ `Cloneable` ist.

Würde die kopierende Klasse das Interface `Cloneable` nicht implementieren, so würde eine Exception geworfen.

Die Methode `clone()` erzeugt ein neues Objekt und belegt die Datenfelder mit den exakt gleichen Werten wie das Objekt, für das die Methode aufgerufen wird. Es wird eine Referenz vom Typ `Object` auf das neu erzeugte Objekt zurückgegeben. Diese muss nur noch in den richtigen Typ gecastet werden.

Kapitel **14.7** befasst sich mit Marker Interfaces wie die Schnittstelle `Cloneable`.

Marker Interfaces dienen dazu, dass Klassen in zwei Mengen aufgeteilt werden, in diejenigen Klassen, welche eine bestimmte Schnittstelle implementieren und in diejenigen Klassen, die diese Schnittstelle nicht implementieren.

14.9 Übungen

Aufgabe 14.1: Testschnittstelle für die Klasse Person

Implementieren Sie in der Klasse `Person` das Interface `Testschnittstelle`:

```
// Datei: Testschnittstelle.java

public interface Testschnittstelle
{
   public void print();
}
```

Die Methode `print()` soll die Werte aller Datenfelder eines Objektes ausgeben.

```
// Datei: Person.java
import java.util.Scanner;

public class Person . . . . .
{
   private String name;
   private String vorname;
   public Person()
   {
      Scanner eingabe = new Scanner (System.in);

      try
      {
         System.out.print ("\nGeben Sie den Nachnamen ein: ");
         name = eingabe.nextLine();

         System.out.print ("\nGeben Sie den Vornamen ein: ");
         vorname = eingabe.nextLine();
      }
      catch (Exception e)
      {
```

```
            System.out.println ("Eingabefehler");
            System.exit (1);
         }
      }
      . . . . .
}
```

Verwenden Sie zum Testen die Klasse `TestPerson`:

```
// Datei: TestPerson.java

public class TestPerson
{
   public static void main (String[] args)
   {
      Person refPerson = new Person();
      refPerson.print();
   }
}
```

Aufgabe 14.2: Implementieren von mehreren Schnittstellen

Die Klasse `Laserdrucker` soll die Schnittstelle `Drucker`, die Klasse `Faxgeraet` die Schnittstelle `Fax` und die Klasse `Kombigeraet` soll die beiden Schnittstellen `Fax` und `Drucker` implementieren. Die Schnittstellen `Fax` und `Drucker` sind gegeben durch:

```
// Datei: Fax.java

public interface Fax
{
   String faxsimulation = . . . . . ;
   public void senden (String sendeRef);
}


// Datei: Drucker.java

public interface Drucker
{
   String drucksimulation = . . . . . ;
   public void drucken (String druckRef);
}
```

Das folgende Bild zeigt die Implementierung der Schnittstellen:

Schreiben Sie die Klassen `Laserdrucker`, `Faxgeraet` und `Kombigeraet` so, dass die Klasse `TestGeraete`

```
// Datei: TestGeraete.java

public class TestGeraete
{
   public static void main (String[] args)
   {
      Laserdrucker l1 = new Laserdrucker();
      Laserdrucker l2 = new Laserdrucker();
      Faxgeraet f1 = new Faxgeraet();
      Faxgeraet f2 = new Faxgeraet();
      Kombigeraet k1 = new Kombigeraet();
      Kombigeraet k2 = new Kombigeraet();
      f1.senden ("Dies ist ein Test");
      f2.senden ("Dies ist ein Test");
      l1.drucken ("Dies ist ein Test");
      l2.drucken ("Dies ist ein Test");
      k1.senden ("Dies ist ein Test");
      k2.senden ("Dies ist ein Test");
      k1.drucken ("Dies ist ein Test");
      k2.drucken ("Dies ist ein Test");
   }
}
```

die folgende Ausgabe erzeugt:

```
Absender ist: Fax1

Das Senden wird simuliert
Dies ist ein Test

Absender ist: Fax2

Das Senden wird simuliert
Dies ist ein Test

Drucker Laser1 meldet sich

Das Drucken wird simuliert
Dies ist ein Test

Drucker Laser2 meldet sich

Das Drucken wird simuliert
Dies ist ein Test

Absender ist: Kombigerät1

Das Senden wird simuliert
Dies ist ein Test

Absender ist: Kombigerät2

Das Senden wird simuliert
Dies ist ein Test
```

```
Kombigerät Kombigerät1 meldet sich

Das Drucken wird simuliert
Dies ist ein Test

Kombigerät Kombigerät2 meldet sich

Das Drucken wird simuliert
Dies ist ein Test
```

Aufgabe 14.3: Vererbung und Schnittstellen

Der folgende Java-Code ist zu analysieren. Anschließend sind die folgenden beiden Aufgaben zu lösen. Hier das Programm:

```
// Datei: Adressierbar.java

public interface Adressierbar
{
   public void setEmpfaenger (String[] adresse);
   public String[] getEmpfaenger();
}

// Datei: Versendbar.java
public interface Versendbar extends Adressierbar
{
   public void setAbsender (String[] absender);
   public String[] getAbsender();
   public int getGewicht();
}

// Datei: Postamt.java

public class Postamt
{
   . . . . .
}

// Datei: Sendung.java

public class Sendung . . . . .
{
   . . . . .
}

// Datei: Start.java

public class Start
{
   public static void main (String[] args)
   {
      int gewicht = 80;
      String[] an  = {"Thomas Vollmer",
                      "Flandernstrasse 101", "73730 Esslingen"};
      String[] von = {"Bernhard Hirschmann",
                      "Hölderlinweg 161", "73728 Esslingen"};
```

```
        Sendung brief = new Sendung (an, von, gewicht);
        Postamt post  = new Postamt();
        post.versende (brief);
    }
}
```

a) Implementieren Sie die `versende()`-Methode der Klasse `Postamt`. Es soll ein Übergabeparameter vom Typ `Versendbar` entgegengenommen werden. Außerdem soll folgende Bildschirmausgabe erfolgen:

   Sendung wurde entgegengenommen und wird jetzt versandt.
   Absender: Bernhard Hirschmann Hölderlinweg 161 73728 Esslingen
   Empfänger: Thomas Vollmer Flandernstrasse 101 73730 Esslingen

b) Schreiben Sie eine gültige Implementierung der Klasse `Sendung`, welche die Schnittstelle `Versendbar` implementiert, sodass ein Objekt der Klasse `Sendung` von der Methode `versenden()` der Klasse `Postamt` verarbeitet werden kann.

Aufgabe 14.4: Schnittstelle Musikinstrument

Schreiben Sie die Schnittstelle `Musikinstrument` mit der Methode `spieleInstrument()`. Implementieren Sie diese Schnittstelle in den beiden Klassen `Trommel` und `Trompete`. Das Musikinstrument soll hierbei beim Spielen eine entsprechende Ausgabe auf dem Bildschirm machen. So soll z. B. eine Trommel am Bildschirm `"Trommel, Trommel"` ausgeben. Zum Testen der Klassen soll die Methode `main()` in der Klasse `Musikantenstadl` mehrere Musikinstrumente erzeugen und abspielen.

Aufgabe 14.5: Bildschirmschoner

Die unten abgedruckte Klasse `BildschirmschonerTest` simuliert einen einfachen Bildschirmschoner, indem sie mehrere Objekte geometrischer Figuren erzeugt und deren Größe und Position verändert. In diesem Beispiel verwendet die Klasse `BildschirmschonerTest` die zwei Klassen `Kreis` und `Quadrat`. Damit auch andere geometrische Figuren in den Bildschirmschoner integriert werden können, werden zwei Schnittstellen verwendet, die von den verschiedenen Klassen zu implementieren sind.

Die Schnittstelle `Position` enthält die Methode

```
verschiebe (float x, float y),
```

um die Position einer Figur zu ändern. Sie verändert die Position eines Körpers dadurch, dass sie die übergebenen Parameter zu den aktuellen Koordinaten hinzuaddiert.

Die Schnittstelle `Groesse` enthält die Methode

```
aendereGroesse (float faktor),
```

um die Größe einer Figur ändern zu können. Die Methode `aendereGroesse()` soll eine Exception bei negativem Parameter werfen. Sie kann die Größe verändern, in-

dem sie den Radius des Kreises mit einem Faktor multipliziert, der als Parameter übergeben wird.

Die Klasse `Kreis` implementiert beide Schnittstellen. Zusätzlich enthält die Klasse `Kreis` ein Datenfeld `radius` vom Typ `float` sowie zwei `float`-Datenfelder, um den Mittelpunkt des Kreises zu definieren (alternativ können Sie für den Mittelpunkt auch die Klasse `Punkt` aus früheren Übungen verwenden). Die Klasse `Quadrat` implementiert nur die Schnittstelle `Position`. Außerdem enthält die Klasse `Quadrat` ein Datenfeld `seitenlaenge` vom Typ `float` und zwei `float`-Datenfelder, um die linke obere Ecke des Quadrats zu bestimmen. Auch hier können Sie alternativ die Klasse `Punkt` wiederverwenden. Bei jeder Änderung der Größe oder der Position einer geometrischen Figur soll ein entsprechender Text auf der Konsole ausgegeben werden. Eine grafische Ausgabe der geometrischen Figuren ist in dieser Aufgabe nicht beabsichtigt.

Verwenden Sie bitte folgende Testklasse:

```
// Datei: BildschirmschonerTest.java

import java.util.Random;

public class BildschirmschonerTest
{
   public static void main (String[] args)
   {
      Random random = new Random();

      for (int i = 0; i <= 10; i++)
      {
         Object koerper;

         if (random.nextBoolean())
         {
            koerper = new Kreis (2.0f);
         }
         else
         {
            koerper = new Quadrat (3.0f);
         }

         if (koerper instanceof Position)
         {
            Position position = (Position) koerper;
            position.verschiebe (random.nextFloat(),
                                 random.nextFloat());
         }

         if (koerper instanceof Groesse)
         {
            Groesse groesse = (Groesse) koerper;

            try
            {
               groesse.aendereGroesse (random.nextFloat()-0.5f);
            }
```

```
            catch (Exception e)
            {
               e.printStackTrace();
            }
         }
      }
   }
}
```

Aufgabe 14.6: Schnittstelle Cloneable

Die Übungsaufgabe 14.2 soll erweitert werden, damit Objekte der Klasse `Faxgeraet`, `Kombigeraet` und `Laserdrucker` geklont werden können. Dazu müssen diese Klassen die Schnittstelle `Cloneable` implementieren und um die Methode `clone()` erweitert werden. Innerhalb der Methode `clone()` muss die Instanzvariable `geraeteId` neu gesetzt werden und die Anzahl der Geräte muss erhöht werden.

Die Geräte der Klasse `Kombigeraet` und `Laserdrucker` sollen um das Datenfeld `anzahlDruckpapier` des Typs `int` erweitert werden. Die Anzahl der verfügbaren Druckpapiere sollte beim Konstruktoraufruf mitgegeben werden. Bei jedem Druckvorgang muss das Datenfeld `anzahlDruckpapier` um eine Seite dekrementiert werden. Beim Druck muss nicht geprüft werden, ob noch Papier vorhanden ist.

Prüfen Sie anhand folgender Testklasse die Funktion Ihrer Anpassungen:

```
// Datei: TestGeraete.java

public class TestGeraete
{
   public static void main (String[] args)
      throws CloneNotSupportedException
   {
      Laserdrucker l1 = new Laserdrucker(120);
      Laserdrucker l2 = l1.clone();

      l1.drucken ("Dies ist ein Test");
      l1.drucken ("Dies ist ein Test");

      // Das geklonte Objekt besitzt vor dem folgenden
      // Druckvorgang weiterhin 120 Druckpapiere und ist somit
      // keine Referenz auf den ersten Laserdrucker.
      l2.drucken ("Dies ist ein Test");
   }
}
```

Aufgabe 14.7: Flughafen-Projekt – Simulator und Schnittstellen

Innerhalb dieser Projektaufgabe soll die bisherige Anwendung um einen Flugzeugsimulator erweitert werden. Dieser hat die Aufgabe, den Landeanflug und den Startvorgang eines Flugzeugs zu simulieren, indem er den Statuswechsel des Flugzeugs anfordert.

Der Status eines Flugzeugs soll nun nicht mehr in den Methoden für die einzelnen Lande-/Startphasen gesetzt werden, sondern zentral über eine neue Methode. Diese Methode hat den folgenden Methodenkopf:

```
void aktualisiereStatus() throws StatusUngueltigException
```

Diese Methode ermittelt als Erstes bei jedem Aufruf den nächsten Status, überprüft dann, ob dieser gesetzt werden kann, und setzt – wenn möglich – den neuen Status. Ist der neue Status nicht möglich, soll eine Exception vom Typ `StatusUngueltigException` geworfen werden. So muss z. B. verhindert werden, dass das Flugzeug den Landeanflug einleitet, bevor diesem eine Landebahn zugewiesen wurde.

Schreiben Sie als nächstes die Klasse `FlugzeugSimulator`. Bei dieser Klasse soll das zu simulierende Flugzeug über eine Methode angemeldet werden. Zusätzlich erhält die Klasse `FlugzeugSimulator` eine Methode `aktualisiereStatus()`, welche bei jedem Aufruf die gleichnamige Methode `aktualisiereStatus()` des zu simulierenden Flugzeugs aufruft. Beachten Sie dabei, dass der Flugzeugsimulator derzeit nur ein einziges Flugzeug simulieren soll. Schreiben Sie auch die Klasse `StatusUngueltigException` und testen Sie Ihre Änderungen, indem Sie die Klasse `Client` anpassen.

Der Flugzeugsimulator soll nun zusätzlich bei Statusänderung des simulierten Flugzeuges angemeldete Interessenten über die Statusänderung benachrichtigen. Ein solches Konzept wurde schon in Kapitel 14.2 vorgestellt und soll nun in ähnlicher Form implementiert werden. Ein Interessent – also Nachrichtenempfänger – muss folgendes Interface implementieren:

```
public interface FlugzeugListener
{
   public void meldeStatusAenderung (Flugzeug flugzeug);
}
```

Um Benachrichtigungen zu empfangen, muss sich der Nachrichtenempfänger bei dem Flugzeugsimulator anmelden. Das Anmelden geschieht über die folgende Methode:

```
public void setFlugzeugListener (FlugzeugListener flugzeugListener)
{
   this.flugzeugListener = flugzeugListener;
}
```

Schreiben Sie eine Klasse, welche die Schnittstelle `FlugzeugListener` implementiert und bei jeder Statusänderung die Methode `print()` der Klasse `Flugzeug` aufruft. Melden Sie ein Objekt dieser Klasse beim Flugzeugsimulator an und testen Sie Ihre Anwendung.

Kapitel 15

Geschachtelte Klassen

15.1 Elementklassen
15.2 Lokale Klassen
15.3 Statisch geschachtelte Klassen und Schnittstellen
15.4 Realisierung von geschachtelten Klassen
15.5 Zusammenfassung
15.6 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_15

15 Geschachtelte Klassen

Geschachtelte Klassen werden in Java meist dazu eingesetzt, um Klassen oder Schnittstellen in einer sie umfassenden Klasse zu verbergen und sie anderen Klassen außerhalb der umfassenden Klasse nicht zugänglich zu machen.

Mit geschachtelten Klassen können **Typen** gebildet werden, die nur für die Implementierung **innerhalb einer Klasse benötigt** werden. In den Aufrufschnittstellen der umfassenden Klasse sind diese Typen nicht nach außen sichtbar und können damit bei Bedarf problemlos geändert werden.

Für geschachtelte Klassen sind auch die Begriffe **innere Klasse**, **eingebettete Klasse** und **nested class** bekannt.

Es gibt **verschiedene Arten von geschachtelten Klassen**:

- die **Elementklasse** (gekapselt als ein Element in einer äußeren Klasse),
- die **lokale Klasse** (gekapselt in Methoden bzw. in einem Block)
- und die **statisch geschachtelte Klasse**.

Bei den lokalen Klassen gibt es

- lokale Klassen mit Klassennamen
- und lokale Klassen ohne Klassennamen, die sogenannten anonymen Klassen

Die Bilder 15-1 bis 15-3 zeigen verschiedene Arten von geschachtelten Klassen:

- Bild 15-1 zeigt ein Objekt einer **Elementklasse**, das innerhalb eines Objektes als Element der umschließenden Klasse lebt:

Objekt einer Elementklasse in einem umschließenden Objekt.

Bild 15-1 Objekt einer Elementklasse

- In Bild 15-2 ist links ein Objekt einer **lokalen Klasse** dargestellt. Diese lokale Klasse ist nur innerhalb einer Instanzmethode sichtbar. Rechts ist ein Objekt einer lokalen Klasse zu sehen, dessen Klasse in einer Klassenmethode sichtbar ist:

Bild 15-2 Objekte einer lokalen Klasse

- Eine **statisch geschachtelte Klasse**[185] wird ebenfalls innerhalb einer umschließenden Klasse definiert, wie Bild 15-3 zeigt:

Bild 15-3 Umsetzung einer statisch geschachtelten Klasse durch den Compiler

Statisch geschachtelte Klassen werden vom Compiler in neben der entsprechenden äußeren Klasse stehende Klassen umgesetzt. Diese Klassen sind voneinander **unabhängig bis auf den gemeinsamen Namensraum**.

Elementklassen, lokale Klassen – speziell anonyme Klassen – und statisch geschachtelte Klassen sind grundlegend verschieden. Für alle gemeinsam gilt jedoch:

Eine **geschachtelte Klasse wird innerhalb** einer **umschließenden Klasse definiert**.

In den folgenden Unterkapiteln werden die unterschiedlichen Arten von geschachtelten Klassen ausführlich vorgestellt.

[185] Java kennt nicht nur statisch geschachtelte Klassen, sondern auch statisch **geschachtelte Schnittstellen** (siehe Kapitel 15.3).

Kapitel 15.1 erläutert Elementklassen, Kapitel 15.2 lokale Klassen und Kapitel 15.3 statisch geschachtelte Klassen bzw. Schnittstellen als Ausprägungen geschachtelter Klassen. Um das Verständnis für geschachtelte Klassen zu vertiefen, wird in Kapitel 15.4 die Realisierung von geschachtelten Klassen anhand von Beispielen gezeigt.

15.1 Elementklassen

Eine Elementklasse ist – wie der Name schon sagt – ein **Element** einer Klasse, das einen Namen trägt.

Das folgende Bild zeigt eine geschachtelte Klasse (innere Klasse) als Elementklasse bzw. Element einer äußeren Klasse:

Bild 15-4 Geschachtelte Klasse als Element einer äußeren Klasse

Objekte von Elementklassen können nur existieren, wenn auch ein Objekt der umschließenden Klasse existiert.

Die Möglichkeit einer Instanziierung einer Elementklasse ist also an die Existenz eines Objektes der umschließenden Klasse gebunden. Daraus folgt, dass nur über ein Objekt einer umschließenden Klasse ein Objekt einer Elementklasse erzeugt werden kann.

Eine Methode eines **Elementobjektes** kann auf alle Datenfelder und Methoden – selbstverständlich auch auf private – des **umfassenden Objektes** zugreifen, zu dem sie gehört. Genauso kann eine Methode eines **Objektes einer äußeren Klasse** auf jede Methode und jedes Datenfeld eines erzeugten **Objektes einer Elementklasse zugreifen**, d. h. es besteht **kein Zugriffsschutz**.

Eine innere Klasse und die sie einschließende äußere Klasse sind wechselseitig also vollkommen gleichberechtigt. Es gibt keine speziellen Privilegien bezüglich des gegenseitigen Zugriffs. Die Zugriffe auf bzw. von einer Elementklasse symbolisiert das folgende Bild:

Bild 15-5 Wechselseitiger Zugriff zwischen Objekten der inneren und äußeren Klasse

15.1.1 Zugriffsschutz für eine Elementklasse

Da eine Elementklasse ein Element einer Klasse ist, das einen Namen trägt, ist der Zugriffsschutz gleich wie bei den schon bekannten Bestandteilen einer Klasse, den Datenfeldern und Methoden. Genauso wie man auf Instanzmethoden und Instanzvariablen nur über die Referenz auf ein Objekt zugreifen kann, kann man auch auf Objekte von Elementklassen nicht direkt zugreifen. Objekte von Elementklassen werden immer über das umschließende Objekt angesprochen.

Als Zugriffsschutz für eine Elementklasse existieren dieselben Möglichkeiten wie für Methoden und Datenfelder:

- `private`,
- `protected`,
- `public`
- und default, d. h. der Zugriff kann nur von Klassen erfolgen, die im selben Paket liegen.

Bei äußeren Klassen gibt es jedoch nur den Zugriffsschutz `public` und default.

15.1.2 Syntax einer Elementklasse

Hier ein Beispiel für die **Syntax einer Elementklasse**:

```
public class AeussereKlasse
{
   . . . . .

   . . . . .
   class ElementKlasse
   {
      . . . . .
   }
}
```

Welcher Zugriffsmodifikator für eine Elementklasse verwendet wird, hängt davon ab, ob diese nur innerhalb der äußeren Klasse sichtbar sein soll (`private`), innerhalb eines Paketes (default bzw. `protected`), innerhalb einer Sohnklasse eines anderen Paketes (`protected`) oder ob sie auch für Klassen anderer Pakete sichtbar sein soll (`public`). Wird eine Schnittstelle als Element einer Klasse definiert, so können entsprechend wie bei Elementklassen die Zugriffsmodifikatoren `private`, default, `protected` und `public` verwendet werden.

Die Sichtbarkeit ändert allerdings nichts an der Tatsache, dass ein Objekt einer inneren Klasse nur mit Hilfe eines Objektes einer äußeren Klasse erzeugt werden kann. Ist die Elementklasse und die äußere Klasse z. B. `public`, so kann von jeder beliebigen Stelle eines Programms ein Objekt einer Elementklasse folgendermaßen erzeugt werden:

```
AuessereKlasse ref = new AuessereKlasse();
AuessereKlasse.ElementKlasse elem = ref.new ElementKlasse();
```

Die Notation erscheint zuerst ein bisschen seltsam – was sie wohl auch ist – doch mit zunehmendem Verständnis gewöhnt man sich schnell daran. Dass die Elementklasse nur über den Namensraum der umschließenden Klasse – also mit der Punktnotation `AuessereKlasse.ElementKlasse` – angesprochen werden kann, klingt logisch. Dagegen tut man sich wesentlich schwerer damit, dass mit `ref.new ElementKlasse()` ein Objekt der Elementklasse erzeugt wird.

Der Compiler macht daraus jedoch Folgendes, was wiederum verständlicher wirkt:

```
new AuessereKlasse.ElementKlasse (ref);
```

Wie aus der soeben gezeigten internen Darstellung des Compilers ersichtlich ist, muss jede Elementklasse automatisch einen Konstruktor besitzen, der für den Programmierer unsichtbar und implizit als ersten Parameter – und im Fall eines parameterlosen Konstruktors als einzigen Parameter – die Referenz `ref` auf ein Objekt der umschließenden Klasse entgegennimmt.

Diese Referenz wird in einem für den Programmierer ebenfalls unsichtbaren, privaten Datenfeld des erzeugten Objektes der Elementklasse abgespeichert. Diese Referenz **gewährleistet** damit den **Zugriff auf** die **Datenfelder und Methoden des umschließenden Objektes vom Objekt der Elementklasse aus**. Das folgende Bild zeigt den Zugriff eines Objekts einer Elementklasse auf ein Objekt seiner äußeren Klasse:

Bild 15-6 Zugriff eines Objektes einer inneren Klasse auf sein zugehöriges äußeres Objekt

15.1.3 Einschränkungen für Elementklassen

Es bestehen die folgenden Einschränkungen:

- Elementklassen dürfen keine Klassenvariablen, Klassenmethoden oder statische Klassen beinhalten. Klassenvariablen und Klassenmethoden sind nur bei einer statisch geschachtelten Klasse erlaubt.
- Elementklassen dürfen nicht den gleichen Namen wie eine umschließende Klasse besitzen.

15.1.4 Beispiel für eine Elementklasse

Im folgenden Beispiel wird die Klasse `Viewer1` dazu benutzt, um Bilder anzuzeigen. Da nur bestimmte Formate unterstützt werden, benutzt die Klasse `Viewer1` eine Elementklasse `Typen`, um abzuprüfen, welche Bildformate erlaubt sind. Dabei wird im Konstruktor des Viewers mit Hilfe eines Objektes der Klasse `Typen` abgeprüft, ob das Bild angezeigt werden kann. Es ist sinnvoll, die Klasse `Typen` als eine innere Klasse zu entwerfen, da sie als eigene Klasse außerhalb der Klasse `Viewer1` keinen Sinn macht. Hier das bereits erwähnte Programm:

```
// Datei: Viewer1.java
public class Viewer1                        // äussere Klasse
{
   private String typ;
   private String dateiname;

   // Konstruktor der äußeren Klasse
   public Viewer1 (String dateiname, String typ)
   {
      Typen refTyp = new Typen();           // Objekt der inneren
                                            // Klasse erzeugen
      this.typ = typ;
      this.dateiname = dateiname;

      if (refTyp.testTyp (typ))             // Methode für Objekt der
      {                                     // Elementklasse
         System.out.println ("Bild " + dateiname
                             + " kann angezeigt werden!");
         show();                            // Bild anzeigen
      }
      else
         System.out.println ("Es werden nur die Formate: "
                             + refTyp + " unterstützt.");
   }

   public void show()                       // Methode der äußeren
   {                                        // Klasse
      //Bild am Bildschirm anzeigen
   }

   private class Typen                       // innere Klasse
   {
      private String typ1 = "gif";
      private String typ2 = "jpg";
```

```
      private String typ3 = "bmp";

      public boolean testTyp (String typ)
      {
         if (typ1.equals (typ) || typ2.equals (typ)
            || typ3.equals (typ))
            return true;
         return false;
      }

      public String toString()
      {
         return typ1 + " " + typ2 + " " + typ3;
      }
   }
}

// Datei: TestViewer.java
public class TestViewer
{
   public static void main (String[] args)
   {
      Viewer1 bild1 = new Viewer1 ("C:\\verz\\Bild.jpg", "jpg");
      Viewer1 bild2 = new Viewer1 ("C:\\verz\\Bild.cpg", "cpg");
   }
}
```

Die Ausgabe des Programmes `TestViewer.java` ist:

```
Bild C:\verz\Bild.jpg kann angezeigt werden!
Es werden nur die Formate: gif jpg bmp unterstützt.
```

Zu beachten ist bei dem obigen Programm:

- In der Elementklasse wird die Methode `toString()`, die von der Klasse `Object` geerbt ist, überschrieben. Die Methode `toString()` wird automatisch aufgerufen, wenn die String-Repräsentation für ein Objekt angefordert wird.
- Die Codezeile `Typen refTyp = new Typen()` ist eine Kurzschreibweise für die Codezeile `Viewer1.Typen refTyp = this.new Typen()`[186]. Diese Kurzschreibweise ist selbstverständlich aber nur innerhalb der äußeren Klasse erlaubt, weil nur dort der Elementname `Typen` bekannt ist. Im Konstruktor der äußeren Klasse wird also ein Objekt der Elementklasse erzeugt.
- Der Compiler generiert bei der Kompilierung der Klasse `Viewer1` zwei `.class`-Dateien, nämlich die Dateien `Viewer1.class` und `Viewer1$Typen.class`.

15.2 Lokale Klassen

Bei lokalen Klassen wird zwischen **lokalen Klassen mit Klassennamen** und anonymen lokalen Klassen unterschieden.

[186] Hierbei ist `this` eine Referenz auf das umschließende Objekt der Elementklasse.

Man beachte, dass für lokale Klassen kein Zugriffsmodifikator vergeben werden kann. Dies würde auch keinen Sinn ergeben, da eine solche Klasse sowieso nur innerhalb des Blockes gültig ist, in dem sie definiert wurde.

Ein Objekt einer lokalen Klasse kann auf alle Datenfelder und Methoden des umschließenden Objektes zugreifen. Zusätzlich kann das Objekt einer lokalen Klasse auch auf alle als `final` deklarierten lokalen Variablen und Übergabeparameter der Methode, in der sich das Objekt befindet, zugreifen. Für alle mit `final` deklarierten lokalen Variablen, die von der lokalen Klasse benutzt werden, erstellt der Compiler für die lokale Klasse eine **lokale Kopie** in Form einer **privaten Instanzvariablen**.

Lokale Klassen können – unabhängig davon, ob sie nun innerhalb einer Instanzmethode oder einer Klassenmethode definiert werden – auf alle lokalen finalen Variablen innerhalb der umschließenden Methode zugreifen. Einzige Voraussetzung ist, dass eine lokale finale Variable vor der lokalen Klasse definiert wird.

15.2.1 Einsatzgebiete von lokalen Klassen

Lokale Klassen trifft man oft bei der Oberflächenprogrammierung an, wo diese entweder von einer **Adapterklasse** ableiten oder eine Schnittstelle implementieren.

Diese Adapterklassen und Schnittstellen, von denen die lokalen Klassen ableiten bzw. diese implementieren, sind häufig Klassen bzw. Schnittstellen aus der Java-API. Diese Klassen und Schnittstellen werden zur Ereignisbehandlung von Oberflächenkomponenten verwendet.

Da für jede Oberflächenkomponente in der Regel eine eigenständige Ereignisbehandlung erfolgen muss – diese Behandlung aber nur für die einzelne Komponente verwendet werden kann – ist es sinnvoll, diese Ereignisbehandlung in einer lokalen Klasse zu kapseln.

15.2.2 Einschränkungen für lokale Klassen

Es existieren die folgenden Einschränkungen:

- Lokale Klassen dürfen wie die Elementklassen keine als `static` deklarierten Datenfelder, Methoden oder Klassen definieren.
- Lokale Klassen dürfen nicht den gleichen Namen wie die umschließende Klasse haben.
- Eine lokale Klasse darf in der Klassendeklaration keinen der Modifikatoren `private`, `protected`, `public` oder `static` verwenden. All diese Modifikatoren sind gebräuchlich für Elemente einer Klasse – eine lokale Klasse ist aber kein Element einer Klasse.

15.2.3 Lokale Klassen mit Klassennamen

Innerhalb eines jeden Blockes (siehe Kapitel 9.1.1) können Deklarationsanweisungen zwischen normalen Anweisungen stehen. Eine Deklarationsanweisung kann eine Definition einer statischen Variablen, aber auch die Definition eines neuen Datentyps – sprich einer neuen Klasse – darstellen.

Lokale Klassen[187] werden im Rahmen einer Deklarationsanweisung[188] definiert. Ihr **Gültigkeitsbereich** und ihre **Sichtbarkeit** erstreckt sich auf den **umfassenden Block** (typischerweise eine Methode).

Das folgende Bild visualisiert eine lokale Klasse innerhalb einer Methode:

Bild 15-7 Eine lokale Klasse ist nur in ihrem umfassenden Block sichtbar.

Für **lokale Klassen mit Klassennamen in Instanzmethoden** gilt:

- Da jede Instanzmethode auf die Datenfelder und Methoden ihres Objektes zugreifen kann, besteht auch für j**edes Objekt einer lokalen Klasse mit Klassennamen**, das sich **innerhalb einer Instanzmethode** befindet, die Möglichkeit, auf die Instanzvariablen und Instanzmethoden ihres umschließenden Objektes zuzugreifen. Da jede Instanzmethode auch auf die Klassenvariablen und Klassenmethoden der zugehörigen Klasse zugreifen kann, hat jedes Objekt einer lokalen Klasse – das sich innerhalb einer Instanzmethode befindet – auch die Möglichkeit, auf die Klassenvariablen und Klassenmethoden der umschließenden Klasse zuzugreifen.

Für **lokale Klassen mit Klassennamen in Klassenmethoden** gilt:

- Da jede Klassenmethode auf die Klassenvariablen und Klassenmethoden der eigenen Klasse zugreifen kann, besteht auch für **jedes Objekt einer lokalen Klasse** – das sich **innerhalb einer Klassenmethode** befindet – die Möglichkeit, auf

[187] Lokale Schnittstellen gibt es nicht.
[188] Eine Deklaration gibt dem Compiler einen Namen bekannt.

die **Klassenvariablen und Klassenmethoden** der umschließenden Klasse zuzugreifen.

Lokale Klassen können innerhalb eines Blockes definiert werden. Es gibt lokale Klassen somit in Instanzmethoden und in Klassenmethoden. Lokale Klassen sind nur in dem umschließenden Block sichtbar.

Objekt einer lokalen Klasse in einer Instanzmethode des umschließenden Objektes.

Objekt einer lokalen Klasse in einer Klassenmethode der umschließenden Klasse.

Bild 15-8 Lokale Klassen in Instanzmethoden und Klassenmethoden

15.2.3.1 Beispiel für eine innere Klasse mit Namen in einer Instanzmethode

Das folgende Beispiel zeigt eine lokale Klasse `Inner`, die in ihrem Konstruktor auf ein Datenfeld der umschließenden Klasse zugreift:

```
// Datei: Outer.java
public class Outer
{
   private int x;

   public void methode()
   {
      class Inner
      {
         Inner()
         {
            // Zugriff auf Datenfeld der umschliessenden Klasse
            System.out.println ("Wert des Datenfeldes x: " + x);
         }
      }
      // Erzeugung eines Objektes der lokalen Klasse
      new Inner();
   }

   public static void main (String[] args)
   {
      Outer ref = new Outer();
      ref.methode();
   }
}
```

Die Ausgabe des Programmes ist:

```
Wert des Datenfeldes x: 0
```

15.2.3.2 Beispiel für den Zugriff einer lokalen Klasse mit Namen auf lokale Variablen der umschließenden Klasse

Das folgende Beispielprogramm zeigt, wie ein Objekt einer lokalen Klasse auf lokale Variablen der umschließenden Methode zugreift.

```
// Datei: Outer1.java

public class Outer1
{
   public void methode (final int y)
   {
      final int x = 1;
      class Inner
      {
         Inner()
         {
            // Zugriff auf lokale finale Variable x
            System.out.println ("Wert der lokalen finalen"
                                + " Variablen x: " + x);
            // Zugriff auf einen finalen Übergabeparameter
            System.out.println ("Wert des Übergabeparameters y: "
                                + y);
         }
      }
      // Erzeugung eines Objektes der lokalen Klasse
      new Inner();
   }

   public static void main (String[] args)
   {
      Outer1 ref = new Outer1();
      ref.methode (7);
   }
}
```

Die Ausgabe des Programmes ist:

```
Wert der lokalen finalen Variablen x: 1
Wert des Übergabeparameters y: 7
```

Da der Compiler eine Kopie für jede benutzte lokale Variable anlegt, ist es zwingend erforderlich, dass diese Variablen `final` sind. Denn die Kopien der benutzten Variablen werden beim Konstruktoraufruf angelegt und mit den entsprechenden Werten initialisiert. Da aber gewährleistet sein muss, dass die Kopie sowie die originale lokale Variable immer die gleichen Werte tragen, müssen sie folglich `final` sein, um zu verhindern, dass ihre Werte verändert werden können.

15.2.4 Anonyme Klasse

Anonyme Klassen sind **lokale Klassen ohne Namen**. Präziser müsste man formulieren: Eine anonyme Klasse ist eine lokale Klasse, deren Klassennamen für den Programmierer nicht sichtbar und damit auch nicht verwendbar ist. Mit dem versteckten Klassennamen von anonymen Klassen wird sich später das Kapitel 15.4.4 beschäftigen. Vorerst ist relevant, dass eine anonyme Klasse keinen für den Programmierer zugänglichen Klassennamen besitzt. Aufgrund des fehlenden Klassennamens wird von einer anonymen Klasse sofort bei der Klassendefinition ein Objekt erzeugt. Da die Klasse keinen Namen trägt, kann man nur ein einziges Exemplar der Klasse erzeugen.

Wie bei lokalen Klassen mit Klassennamen können anonyme Klassen innerhalb von Instanzmethoden und Klassenmethoden definiert werden.

Anonyme Klassen können als Deklarationsanweisung oder innerhalb einer Ausdrucksanweisung definiert und instanziiert werden.

Eine Deklarationsanweisung könnte zum Beispiel wie folgt aussehen:

```
ActionListener l = new ActionListener()
{
   void actionPerformed (ActionEvent e)
   {
      ...
   }
};

JButton button = ...
button.addActionListener (l);
```

In diesem Beispiel wird eine anonyme Klasse definiert und gleichzeitig instanziiert. Dabei implementiert die anonyme Klasse die Schnittstelle `ActionListener`. Die Instanziierung erfolgt durch den `new`-Operator, gefolgt vom Namen der zu implementierenden Schnittstelle. Die runden Klammern stellen dabei den Konstruktoraufruf dar. Soll eine Klasse anstelle einer Schnittstelle verwendet werden, so muss der Name des Konstruktors der zu erweiternden Klasse angegeben werden. Dabei können auch Konstruktoren mit Parametern verwendet werden. Die darauf folgenden geschweiften Klammern stellen den Definitionsblock der anonymen Klasse dar. Im Definitionsblock müssen nun alle Methoden der Schnittstelle bzw. bei abstrakten Klassen alle abstrakte Methoden implementiert werden. Es dürfen auch zusätzliche Attribute angegeben werden, die bei der Definition sofort initialisiert werden müssen, da bei anonymen Klassen kein individueller Konstruktor zur Verfügung gestellt werden kann. Das durch den `new`-Operator erzeugte Objekt wird der lokalen Variablen `l` zugewiesen, damit diese später der Methode `addActionListener()` übergeben werden kann.

Eine Ausdrucksanweisung würde diese beiden Schritte in einem einzigen Ausdruck vereinen und könnte zum Beispiel wie folgt aussehen:

```
JButton button = ...
button.addActionListener (new ActionListener()
{
   void actionPerformed (ActionEvent e)
   {
     ...
   }
});
```

Hierbei wird das durch den `new`-Operator erzeugte Objekt direkt der Methode `addActionListener()` übergeben, ohne dass es einer Referenz zugewiesen wird.

Da eine anonyme Klasse keinen Namen hat, können auch keine Konstruktoren geschrieben werden.

Um Objekte anonymer Klassen initialisieren zu können, wurde in Java 1.1 der **nicht statische Initialisierungsblock** eingeführt (siehe Kapitel 10.4.3).

Anonyme Klassen werden genau wie lokale Klassen mit Namen oft von Adapterklassen abgeleitet oder implementieren eine Schnittstelle zur Ereignisbehandlung.

15.2.4.1 Einschränkungen für anonyme Klassen

Einschränkungen für anonyme Klassen sind:

- Anonyme Klassen dürfen keine als `static` deklarierten Datenfelder, Methoden oder Klassen definieren.
- Anonyme Klassen können keinen Konstruktor haben, da sie auch keinen Namen besitzen.
- Von anonymen Klassen kann nur eine einzige Instanz erzeugt werden.[189] Deshalb sollte man lokale Klassen mit Klassennamen oder Elementklassen den anonymen Klassen dort vorziehen, wo man mehrere Instanzen einer inneren Klasse benötigt.

15.2.4.2 Beispiele

Beispiel für eine anonyme Klasse

Das folgende Beispiel zeigt das schon bekannte Anwendungsbeispiel eines Viewers, der Bilder anzeigen soll. Hierbei wird eine anonyme Klasse definiert, die die Schnittstelle `ITypen` implementiert:

[189] Genau genommen müsste diese Aussage präzisiert werden: Über die Instanz einer anonymen Klasse erhält man durch den Aufruf von `getClass().newInstance()` eine neue bzw. weitere Instanz.

```
// Datei: Viewer2.java

interface ITypen
{
   public boolean testTyp();
}

public class Viewer2
{
   private String bildTyp;
   private String dateiname;

   // Lokale Variablen einer umschließenden Methode können nur dann
   // in anonymen oder lokalen Klassen verwendet werden, wenn diese
   // final sind.
   public Viewer2 (String dateiname, final String typ)
   {
      bildTyp = typ;
      this.dateiname = dateiname;

      // Erzeugung und Definition der anonymen Klasse, welche die
      // Schnittstelle ITypen implementiert. Dabei wird das
      // Schlüsselwort implements nicht verwendet. Die Erläuterung
      // folgt nach dem Beispiel.
      ITypen refTyp = new ITypen()
      {
         private String typ1 = "gif";
         private String typ2 = "jpg";
         private String typ3 = "bmp";

         public boolean testTyp()
         {
            if(typ1.equals (typ) || typ2.equals (typ) ||
               typ3.equals (typ))
            {
               return true;
            }
            return false;
         }

         public String toString()
         {
            return typ1 + " " + typ2 + " " + typ3;
         }
      }; // Ende der anonymen Klasse

      // Zugriff auf die Methode der anonymen Klasse
      if (refTyp.testTyp())
      {
         System.out.println ("Bild " + dateiname +
                             " kann angezeigt werden!");
      }
      else
      {
         System.out.println ("Es werden nur die Formate: "
                             + refTyp + " unterstützt.");
      }
   }
```

```
   public void show()
   {
      //Bild am Bildschirm anzeigen
   }
}
```

Objekte anonymer Klassen werden durch eine spezielle **Variante des `new`-Operators** erzeugt. Dabei hat die folgende Anweisung[190] eine Bedeutung, die nicht ganz einfach ersichtlich ist:

```
ITypen refTyp = new ITypen()
{
   //Datenfelder und Methoden der anonymen Klasse
};
```

Zum Verständnis soll vergleichsweise die Ersatzdarstellung des Compilers für diese Deklarationsanweisung betrachtet werden:

```
class Viewer2$1 implements ITypen
{
   //Datenfelder und Methoden der Klasse
}
ITypen refTyp = new Viewer2$1();
```

Von der Bedeutung her ist diese Ersatzdarstellung gleichwertig mit der obigen Deklarationsanweisung. Der Name der anonymen Klasse `Viewer2$1` kann allerdings vom Programmierer nicht verwendet werden. Der Compiler erzeugt für die anonyme Klasse in der Klasse `Viewer2` eine Datei mit dem Namen `Viewer2$1.class`. Für jede weitere anonyme Klasse innerhalb der Klasse `Viewer2` würde der Compiler einfach die `.class`-Dateien weiter durchnummerieren.

Mit dem Ausdruck `new ITypen()` und den nachfolgenden geschweiften Klammern wird ausgesagt, dass die in den geschweiften Klammern definierte anonyme Klasse die Schnittstelle `ITypen` implementiert. Gleichzeitig wird mit dieser Anweisung ein Objekt dieser anonymen Klasse angelegt und die zurückgegebene Referenz auf die Schnittstelle `ITypen` gecastet.

Eine anonyme Klasse kann auch von einer anderen Klasse ableiten. Dies wird in dem folgenden Codestück der Klasse `Viewer2` gezeigt:

```
Object ref = new Object()
{
   public String toString()
   {
      return super.toString().toUpperCase();
   }
};
```

[190] Eine Deklaration gibt dem Compiler stets einen Namen bekannt. Dass hier eine Deklaration vorliegt, kann man erst erkennen, wenn man die Ersatzdarstellung mit dem Klassennamen `Viewer2$1` betrachtet.

Zum Verständnis soll auch hier die entsprechende Ersatzdarstellung betrachtet werden. Es soll angenommen werden, dass diese anonyme Klasse in der Klasse `Viewer2` unterhalb der anonymen Klasse vom Typ `ITypen` definiert wird. In diesem Fall würde der Compiler eine Klasse mit dem Namen `Viewer2$2` generieren:

```
class Viewer2$2 extends Object
{
   // Überschreiben der toString()-Methode der Klasse Object
   public String toString()
   {
      return super.toString().toUpperCase();
   }
}
Object ref = new Viewer2$2();
```

Es können von der anonymen Klasse nur die Methoden der implementierten Schnittstelle bzw. der Basisklasse aufgerufen werden.

Da immer nur die Methoden der Schnittstelle bzw. die Methoden der Basisklasse einer anonymen Klasse angesprochen werden können, macht es keinen Sinn, zusätzliche Methoden, die `public` sind, in einer anonymen Klasse zu definieren. Diese zusätzlichen Methoden könnten von außen gar nicht angesprochen werden.

Anonyme Klassen bei Aufzählungskonstanten

Das individuelle Überschreiben von Methoden für eine Aufzählungskonstante[191] bei Aufzählungstypen wird ebenfalls mit Hilfe einer anonymen Klasse realisiert. Dies soll an folgendem Beispiel gezeigt werden:

```
// Datei: AmpelTest.java
enum Ampel
{
   ROT,
   GELB,
   GRUEN
   {
      // Überschreiben der toString()-Methode der Klasse Object
      // für die Aufzählungskonstante GRUEN.
      public String toString()
      {
         return super.toString().toLowerCase();
      }
   };
}

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

[191] Das sieht man erst in der Ersatzdarstellung des Compilers und noch nicht an dieser Stelle.

```
      Ampel amp1 = Ampel.ROT;
      Ampel amp2 = Ampel.GELB;
      Ampel amp3 = Ampel.GRUEN;

      System.out.println (amp1);
      System.out.println (amp2);
      System.out.println (amp3);
   }
}
```

Die Ausgabe des Programmes ist:

```
Die Koordinaten des Punktes sind:
3
2
```

Für die Aufzählungskonstante `GRUEN` wird die `toString()`-Methode überschrieben. Beim Aufruf der `toString()`-Methode für die Aufzählungskonstante `GRUEN` werden damit Kleinbuchstaben anstatt Großbuchstaben ausgegeben. Da der Compiler für jede Aufzählungskonstante ein Objekt des Aufzählungstyps anlegt, wird im obigen Programmcode mit

```
GRUEN
{
   public String toString()
   {
      return super.toString().toLowerCase();
   }
};
```

eine anonyme Klasse angelegt, die von der Klasse `Ampel` ableitet und die Methode `toString()` überschreibt. Der Compiler generiert auch eine entsprechende Datei `Ampel$1.class`. Die Ersatzdarstellung sieht damit folgendermaßen aus:

```
class Ampel$1 extends Ampel
{
   public String toString()
   {
      return super.toString().toLowerCase();
   }
}
public static final Ampel GRUEN = new Ampel$1();
```

Initialisierung des von der Basisklasse geerbten Anteils in einem Objekt einer abgeleiteten Klasse

Da eine anonyme Klasse keinen Konstruktor besitzt, kann auch nicht mit Hilfe des Schlüsselwortes `super` ein Konstruktor mit Parametern in der Basisklasse aufgerufen werden. Doch auch hierfür gibt es eine Lösung, die in folgendem Beispiel vorgestellt werden soll:

```
// Datei: Outer2.java

class Basisklasse
{
   private int x;
```

```
   public Basisklasse (int x)
   {
      System.out.println ("Wert der Variablen x: " + x);
   }
}

public class Outer2
{
   public Outer2 (int x)
   {
      // Es wird ein Objekt der anonymen Klasse, die von Basisklasse
      // abgeleitet ist, angelegt. Nach der Erzeugung erfolgt ein
      // Cast auf die Basisklasse. x ist der Parameter für den Kon-
      // struktor der Basisklasse.
      Basisklasse refB = new Basisklasse (x)
      {
         // Überschriebene Methoden
      };
   }
   public static void main (String[] args)
   {
      Outer2 out = new Outer2 (10);
   }
}
```

Die Ausgabe des Programmes ist:

```
Wert der Variablen x: 10
```

Die Parameter für die Basisklasse werden einfach in die runden Klammern geschrieben. Diese Parameter werden dann an den entsprechenden Konstruktor der Basisklasse weitergegeben.

15.3 Statisch geschachtelte Klassen und Schnittstellen

Statisch geschachtelte Klassen und **statisch geschachtelte Schnittstellen** werden auch als sogenannte **statische Top-Level-Klassen** bzw. **statische Top-Level-Schnittstellen** bezeichnet.

Auch wenn eine statisch geschachtelte Klasse bis auf das Schlüsselwort `static` identisch definiert wird wie eine Elementklasse, so ist doch gerade dies der entscheidende Unterschied. Bei Elementklassen kennt ein eingeschlossenes Objekt sein umgebendes Objekt und umgekehrt. Bei statischen Top-Level-Klassen gibt es diesen Bezug nicht.

Bei statischen Top-Level-Elementen braucht man kein Objekt einer äußeren Klasse, um ein Objekt einer inneren Klasse zu erzeugen.

Eine geschachtelte Top-Level-Klasse oder eine geschachtelte Top-Level-Schnittstelle wird definiert als ein Element einer anderen Top-Level-Klasse oder einer anderen Top-Level-Schnittstelle, welches den Modifikator `static` aufweist.

Das Schlüsselwort `static` hat zur Konsequenz, dass diese Klasse bzw. diese Schnittstelle sich vollkommen gleich wie jede andere normale Top-Level-Klasse bzw. jede andere Top-Level-Schnittstelle verhält, mit dem Unterschied, dass die geschachtelte Klasse über den Namen der umschließenden Klasse angesprochen wird.

So wird eine geschachtelte Top-Level-Klasse z. B. über `AeussereKlasse.InnereKlasse` aufgerufen. Damit bieten geschachtelte Top-Level-Klassen die Besonderheit, dass man zusammengehörige Klassen im Namensraum der umfassenden Klasse gruppieren kann und durch den gemeinsamen Namensraum die Zusammengehörigkeit demonstriert wird. Das folgende Beispiel zeigt geschachtelte Top-Level-Klassen und deren Instanziierung:

```
// Datei: TopLSchicht1.java

public class TopLSchicht1
{
   // Datenfelder

   public TopLSchicht1()
   {
      System.out.println ("TopLSchicht1-Konstruktor");
   }

   public static class TopLSchicht2
   {
      // Datenfelder
      public TopLSchicht2()
      {
         System.out.println ("TopLSchicht2-Konstruktor");
      }

      public static class TopLSchicht3
      {
         // Datenfelder
         TopLSchicht3()
         {
            System.out.println ("TopLSchicht3-Konstruktor");
         }
      }
   }
}

// Datei: TestTopLevel.java

public class TestTopLevel
{
   public static void main (String[] args)
   {
      TopLSchicht1 refSchicht1 = new TopLSchicht1();
```

```
      TopLSchicht1.TopLSchicht2 refSchicht2
                     = new TopLSchicht1.TopLSchicht2();

      TopLSchicht1.TopLSchicht2.TopLSchicht3 refSchicht3
                     = new TopLSchicht1.TopLSchicht2.TopLSchicht3();
   }
}
```

Die Ausgabe des Programmes ist:

```
TopLSchicht1-Konstruktor
TopLSchicht2-Konstruktor
TopLSchicht3-Konstruktor
```

Bei in Klassen geschachtelten Top-Level-Klassen ist das Schlüsselwort `static` immer explizit anzugeben. Geschachtelte Schnittstellen innerhalb von Klassen sind **implizit** `static`.

Bei geschachtelten Schnittstellen oder Klassen, die in Schnittstellen geschachtelt werden, kann man das Schlüsselwort `static` aber auch weglassen, da in Schnittstellen geschachtelte Klassen oder Schnittstellen **implizit** als `static` betrachtet werden.

Folgendes Beispiel zeigt das Verschachteln von Klassen und Schnittstellen:

```
class AeussereKlasse
{
   static class InnereKlasse
   {
      . . . . .
   }

   interface InnereSchnittstelle
   {
      . . . . .
   }
}

interface AeussereSchnittstelle
{
   interface InnereSchnittstelle
   {
      . . . . .
   }

   class InnereKlasse
   {
      . . . . .
   }
}
```

Normalerweise kann innerhalb einer Schnittstelle keine Methodenimplementierung erfolgen. Jedoch kann eine statische geschachtelte Klasse ein Teil einer Schnittstelle sein.

Die Regelungen von Schnittstellen werden nicht verletzt, da eine statisch geschachtelte Klasse in einer Schnittstelle selbst eine Top-Level-Klasse darstellt und nur über den Namensraum mit der Schnittstelle gekoppelt ist.

Zu beachten ist, dass geschachtelte Top-Level-Klassen bzw. -Schnittstellen nur innerhalb von Top-Level-Klassen bzw. -Schnittstellen und nicht in einer sonstigen geschachtelten Klasse (Elementklasse, lokale Klasse, anonyme Klasse) geschachtelt werden können.

Das folgende Bild visualisiert **Top-Level-Klassen** und **Top-Level-Schnittstellen**:

```
class A
{
   static class B
   {
      . . . . .
   }
}
```

```
interface A
{
   interface B
   {
      . . . . .
   }
}
```

```
class A
{
   interface B
   {
      . . . . .
   }
}
```

```
interface A
{
   class B
   {
      . . . . .
   }
}
```

Bild 15-9 Geschachtelte Top-Level-Klassen und Top-Level-Schnittstellen

15.4 Realisierung von geschachtelten Klassen

Um das Verständnis für geschachtelte Klassen abzurunden, ist es hilfreich, sich anzusehen, wie der Compiler geschachtelte Klassen umsetzt. Ein Werkzeug namens `javap` wird mit dem JDK mitgeliefert.

Mit Hilfe von `javap` ist es möglich, eine `.class`-Datei zu disassemblieren, das heißt, aus Bytecode den Coderahmen des ursprünglichen Quellcodes herzustellen. Damit hat man die Möglichkeit, zu sehen, wie der Java-Compiler geschachtelte Klassen umsetzt. Der Coderahmen umfasst die Definition der Klassen mit Datenfeldern und Methodenköpfen. Dabei können die zusätzlichen Datenfelder und Methoden, die vom Compiler zur Realisierung einer geschachtelten Klasse hinzugefügt werden, sichtbar gemacht werden.

15.4.1 Elementklassen mit Zugriff auf Datenfelder der umschließenden Klasse

Es soll betrachtet werden, wie es einer Elementklasse ermöglicht wird, auf die Datenfelder eines äußeren Objektes zuzugreifen:

```
// Datei: Outer3.java

public class Outer3
{
   private int x = 1;

   public class Inner
   {
      public Inner()
      {
         x = x + 10; // Zugriff auf Datenfeld der äußeren Klasse
      }
   }
}
```

Wird die Bytecode-Datei `Outer3.class` disassembliert, so erhält man folgende Ausgabe[192]:

```
// Ausgabe des Disassemblierers bei Eingabe javap -private Outer3
public class Outer3 extends java.lang.Object
{
   private int x;
   public Outer3();  //parameterloser Konstruktor

   // zusätzliche Methode, um Datenfeld x zu schreiben
   static int access$002 (Outer3, int);[193]
   // zusätzliche Methode, um Datenfeld x zu lesen
   static int access$000 (Outer3);
}

// Ausgabe des Disassemblierers bei Eingabe
// javap -private Outer3$Inner
public class Outer3$Inner extends java.lang.Object
{
   // Datenfeld, um auf das umschliessende Objekt zuzugreifen
   final Outer3 this$0;
   // Konstruktor, dem im ersten Parameter eine Referenz auf das
   // umschliessende Objekt übergeben wird
   public Outer3$Inner (Outer3);
}
```

Die `access()`-Methode zum Lesen eines Datenfeldes `x` der äußeren Klasse wird nur dann angelegt, wenn innerhalb der Elementklasse lesend auf das Datenfeld `x` zugegriffen wird.

[192] Die Kommentare wurden von Hand hinzugefügt.
[193] Beachten Sie, dass nur der Typ und nicht der Name des formalen Parameters angegeben wird.

Die `access()`-Methode zum Schreiben eines Datenfeldes `x` der äußeren Klasse wird nur dann angelegt, wenn innerhalb der Elementklasse schreibend auf das Datenfeld `x` zugegriffen wird.

Dabei hat jedes Datenfeld, das von einer Elementklasse benutzt wird, seine eigenen `access()`-Methoden. Die private Referenz `this$0` in der Elementklasse und der Typ des formalen Parameters `Outer3` im Konstruktor werden immer vom Compiler ergänzt, unabhängig davon, ob nun auf Datenfelder der umschließenden Klasse zugegriffen wird oder nicht. Damit wird sichergestellt, dass ein Objekt der Elementklasse nur dann erzeugt werden kann, wenn auch tatsächlich ein umschließendes Objekt existiert. Denn ein Objekt der Elementklasse `Outer3.Inner` kann nur mit Hilfe eines Konstruktors initialisiert werden, dem eine Referenz auf ein Objekt der umschließenden Klasse übergeben wird.

15.4.2 Zugriff von lokalen Klassen mit Namen auf Datenfelder

Eine lokale Klasse kann nur innerhalb eines Blockes instanziiert werden. Ihre Sichtbarkeit beschränkt sich damit auf den umschließenden Block. Die lokale Klasse hat wie eine Elementklasse Zugriff auf die Datenfelder der umschließenden Klasse. Da eine lokale Klasse kein Element einer Klasse mehr ist, kann für sie auch kein Zugriffsmodifikator mehr vergeben werden. Im Folgenden wird eine lokale Klasse innerhalb eines Konstruktors diskutiert:

```
// Datei: Outer4.java

public class Outer4
{
   private int x;

   public Outer4()
   {
      class Inner
      {
         public Inner()
         {
            // Zugriff auf Datenfeld der äußeren Klasse
            x = 10;
         }
      }
   }
}
```

Zu beachten ist, dass der `.class`-Dateiname der lokalen Klasse `Inner` der Name `Outer4$1Inner` ist.

Da innerhalb verschiedener Methoden einer Klasse lokale Klassen mit gleichem Namen definiert werden können, ist eine Durchnummerierung für lokale Klassen erforderlich.

Hat beispielsweise eine Klasse `Outer` die Methoden `methode1()` und `methode2()`, so kann in jeder dieser Methoden eine lokale Klasse mit dem Namen `Inner` definiert werden. Der Compiler würde daraus dann die `.class`-Dateien `Outer$1Inner` und `Outer$2Inner` erzeugen. Das folgende Codestück wird vom Disassemblierer `javap` generiert, wenn auf der Kommandozeile `javap -private Outer4$1Inner` eingegeben wird:

```
class Outer4$1Inner extends java.lang.Object
{
   final Outer4 this$0;
   Outer4$1Inner (Outer4);
}
```

Das Ergebnis der Eingabe von `javap -private Outer4` ist:

```
class Outer4 extends java.lang.Object
{
   private int x;
   public Outer4();
   static int access$002(Outer4, int);
}
```

Da in der lokalen Klasse nur schreibend auf das Datenfeld `x` der umschließenden Klasse zugegriffen wird, existiert nur die `access()`-Methode zum Schreiben des Datenfeldes `x`. Das folgende Beispiel zeigt die Zusammenhänge, wenn eine lokale Klasse innerhalb einer Klassenmethode liegt:

```
// Datei: Outer5.java

class Outer5
{
   private static int x;

   public static void methode()
   {
      class Inner
      {
         Inner()
         {
            // Zugriff auf Datenfeld der äußeren Klasse
            x = 10;
         }
      }
   }
}
```

Werden die Bytecode-Dateien `Outer5$1Inner.class` und `Outer5.class` disassembliert, so erhält man folgende Ausgaben:

```
// Eingabe von: javap -private Outer5$1Inner

class Outer5$1Inner extends java.lang.Object
{
   Outer5$1Inner();
}
```

```
// Eingabe von: javap -private Outer5
public class Outer5 extends java.lang.Object
{
   private static int x;
   Outer5();
   public static void methode();
   static int access$002 (int);
}
```

Die Referenzen auf ein umschließendes Objekt fallen sowohl bei den `access()`-Methoden, als auch beim Konstruktor weg. Dies ist auch logisch, da der Aufruf einer Klassenmethode ja nicht an die Existenz eines Objektes gekettet ist.

15.4.3 Zugriff von lokalen Klassen mit Namen auf lokale Variablen

Auf lokale Variablen eines umschließenden Blockes kann eine lokale Klasse mit Namen zugreifen, sofern diese `final` sind. Der Compiler legt für jede finale lokale Variable, die in einer lokalen Klasse mit Klassennamen verwendet wird und deren Wert zum Kompilierzeitpunkt **nicht** bekannt ist, ein privates Datenfeld in der lokalen Klasse an. Das Datenfeld wird beim Konstruktoraufruf mit dem Wert der entsprechenden lokalen Variablen initialisiert. Der Wert wird somit kopiert. Für eine lokale finale Variable, deren Wert zum Kompilierzeitpunkt bekannt ist, wird einfach der Name der Variablen in der lokalen Klasse durch deren Wert ersetzt. Und nun ein **Programm** mit finalen Variablen:

```
// Datei: Outer6.java

public class Outer6
{
   private int x;

   public Outer6()
   {
      final int y = Math.abs (-10);
      final int w = 11;
      class Inner
      {
         Inner()
         {
            int lokal = w;
         }
         public void print()
         {
            System.out.println (y);
         }
      }
   }
}
```

Wird die Bytecode-Datei `Outer6$1Inner.class` disassembliert, so erhält man folgende Ausgaben:

```
// Eingabe von: javap -private Outer6$1Inner

class Outer6$1Inner extends java.lang.Object
{
   // Kopie der benutzten lokalen Variablen speichern, da zum
   // Kompilierzeitpunkt der Wert noch nicht bekannt ist.
   final int val$y;

   final Outer6 this$0;

   // Übergabe der lokalen Variablen im Konstruktor und Initiali-
   // sierung von val$Y
   Outer6$1Inner (Outer6, int);

   public void print();
}
```

Der Code der Klasse `Outer6` bleibt unverändert, da auf keine Datenfelder zugegriffen wird.

15.4.4 Realisierung anonymer Klassen

Anonyme Klassen behandelt der Compiler identisch wie lokale Klassen mit Klassennamen. Für den Programmierer ergibt sich lediglich der Unterschied, dass er keinen Konstruktor für eine anonyme Klasse anlegen kann.

Der Compiler jedoch erzeugt bei anonymen Klassen einen Konstruktor, um eine eventuell benötigte Referenz auf die umschließende Klasse bzw. die benötigten lokalen Variablen entgegenzunehmen.

Hier ein **Beispiel**:

```
// Datei: Outer7.java

public class Outer7
{
   private int x;

   public Outer7()
   {
      final int y = 10;

      Object obj = new Object()
      {
         // Überschreiben der toString()-Methode der Klasse Object
         public String toString()
         {
            return Integer.toString (x) + Integer.toString (y);
         }
      };
   }
}
```

Die Eingabe von `javap -private Outer7$1` ergibt zusammengefasst:

```
final class Outer7$1 extends java.lang.Object
{
   // Referenz auf umschliessendes Objekt aufnehmen
   final Outer7 this$0;

   // Konstruktor mit einer Referenz auf die umschließende Klasse
   Outer7$1 (Outer7);

   public java.lang.String toString();
}
```

15.5 Zusammenfassung

In Java ist es möglich, Klassen zu verschachteln, d. h. eine geschachtelte Klasse wird innerhalb einer umschließenden Klasse definiert. Dadurch können Sichtbarkeiten und Zugriffsrechte nochmals reduziert werden. Mit geschachtelten Klassen können Typen gebildet werden, die nur innerhalb einer umschließenden Klasse benötigt werden.

Es gibt drei Arten von geschachtelten Klassen: Elementklassen, lokale Klassen und statisch geschachtelte Klassen. Die lokalen Klassen werden wiederum unterschieden in lokale Klassen mit Klassennamen und lokale Klassen ohne Klassennamen (anonyme Klassen).

Kapitel **15.1** gibt einen Überblick über Elementklassen. Eine Elementklasse ist ein Element einer umschließenden Klasse, deshalb kann ein Objekt einer Elementklasse nur existieren, wenn auch ein Objekt der umschließenden Klasse existiert. Elementklassen können eigene Datenfelder und Methoden besitzen. Innerhalb einer Elementklasse kann auf die Datenfelder und Methoden der umschließenden Klasse zugegriffen werden und umgekehrt. Für Elementklassen gilt derselbe Zugriffsschutz wie auch bei anderen Bestandteilen einer Klasse. Objekte von Elementklassen können nur über das umschließende Objekt angesprochen werden. Wie für andere Elemente einer Klasse auch existieren für Elementklassen die Zugriffsmodifikatoren `private`, `protected`, `public` und default.

In Kapitel **15.2** werden lokale Klassen beschrieben, die sich in lokale Klassen mit Namen und lokale Klassen ohne Namen aufteilen. Lokale Klassen werden innerhalb von Methoden – sowohl Instanz- als auch Klassenmethoden – der umschließenden Klasse definiert. Ein Objekt einer lokalen Klasse kann auf alle Datenfelder und Methoden des umschließenden Objektes zugreifen. Eine lokale Klasse ist nur in dem umschließenden Block der Methode, in der sie definiert wurde, sichtbar. Lokale Klassen ohne Namen – sogenannte anonyme Klassen – werden innerhalb einer Ausdrucksanweisung definiert und instanziiert. Somit kann von einer lokalen Klasse ohne Namen nur ein einziges Objekt erzeugt werden.

Statisch geschachtelte Klassen und statisch geschachtelte Schnittstellen werden in Kapitel **15.3** beschrieben. Statisch geschachtelte Klassen werden wie Elementklassen definiert. Im Gegensatz zu den Elementklassen wird bei statisch geschachtelten Klassen kein Objekt der umschließenden Klasse benötigt, um ein Objekt einer inneren Klasse zu erzeugen. Statisch geschachtelte Klassen werden auch als sta-

tische Top-Level-Klassen bezeichnet. Einziger Unterschied zu anderen Top-Level-Klassen ist, dass sie über den Namen der umschließenden Klasse angesprochen werden.

Kapitel **15.4** erklärt, wie der Compiler geschachtelte Klassen umsetzt. Anhand von Beispielen wird gezeigt, wie mit Hilfe des Werkzeugs `javap` eine Klasse disassembliert werden kann und somit die vom Compiler zur Realisierung einer geschachtelten Klasse hinzugefügten Datenfelder und Methoden sichtbar gemacht werden können.

15.6 Übungen

Aufgabe 15.1: Lokale Klasse mit Klassennamen

Implementieren Sie eine Klasse `Email`, die folgende Aufgaben erfüllt:

- Die Klasse soll die Instanzvariablen `betreff`, `text`, `empfaenger` und `absender` vom Typ `String` enthalten.
- Der Konstruktor soll folgende Übergabeparameter besitzen:
  - `String empfaenger`
  - `String absender`
  - `String betreff`
  - `String text`

  Der Konstruktor soll die Attribute mit den entsprechenden Übergabeparametern initialisieren.
- Schreiben Sie eine Methode `senden()`, die bei gültigen Mail-Adressen den Empfänger, den Sender, den Betreff und den Text auf dem Bildschirm ausgibt. Ist entweder die Mail-Adresse des Absenders oder die Mail-Adresse des Empfängers ungültig, so soll eine entsprechende Meldung auf dem Bildschirm ausgegeben werden. Um eine Mail-Adresse auf Gültigkeit zu überprüfen, verwendet die Methode `senden()` die Methode `isValid()` der lokalen Klasse `InternetMailAddress`.

Die Klasse `InternetMailAddress` soll als lokale Klasse in der Methode `senden()` der Klasse `Email` realisiert werden. Die lokale Klasse besitzt folgende Methode:

```
private boolean isValid (String address);
```

Diese Methode überprüft, ob eine übergebene Mail-Adresse gültig ist oder nicht. Gültige Adressen

- besitzen genau ein `@`-Zeichen, das den Namen des Empfängers vom Server trennt (`empfaengername@servername`)
- erlauben folgende Zeichen für Empfängername und Server: a-z, A-Z, 0-9, _, .
- erfordern mindestens ein Zeichen für Empfängername bzw. Server
- beginnen mit einem Zeichen, das nicht `@` oder `*` ist
- erlauben das Weglassen von `@servername`, falls das letzte Zeichen ein `'*'` ist

Trifft die letzte Regel zu, so wird von der Methode `isValid()` als Servername für die Mail-Adresse `"it-designers.de"` eingetragen. **Beispiele**:

- Hans.Muster@gmz.de gültig
- Hans.Muster* gültig (wird zu Hans.Muster@it-designers.de)
- @gmz.de ungültig
- Hans.Mustergmz.de ungültig

Zum Überprüfen der Gültigkeit einer Email-Adresse können Sie reguläre Ausdrücke in Kombination mit der Methode `matches()` der Klasse `String` verwenden. Konsultieren Sie hierzu die Java-Dokumentation. Testen Sie Ihre Klasse `Email` mit folgendem Programm:

```
// Datei: TestEmail.java
public class TestEmail
{
   public static void main (String[] args)
   {
      Email e1 = new Email ("Klaus.Gross*","Lotte.Klein@gmz.de",
                            "Hallo","Hallo Welt");
      e1.senden();
      Email e2 = new Email ("Klaus.Gross*","Lotte.Kleingmz.de",
                            "Hallo","Hallo Welt");
      e2.senden();
   }
}
```

Hier ein Auszug der Programmausgabe:

```
-----------------------------------------------
Mail von: Lotte.Klein@gmz.de
An: Klaus.Gross@it-designers.de
Betreff: Hallo
Text:
Hallo Welt
-----------------------------------------------

Die Email konnte nicht verschickt werden!
Die Email-Adresse des Absenders ist ungültig.
```

Aufgabe 15.2: Anonyme Klasse

Es wird ein Betankungsvorgang simuliert. Dazu werden die Klassen `Tank`, `Tanksaeule` und die Schnittstelle `FuellstandSensor` benötigt. Die Schnittstelle `FuellstandSensor` soll in einer anonymen Klasse implementiert werden.

Die Klasse `Tank` besitzt folgende Methoden und Instanzvariablen:

- `public Tank (int volumen)`
- `public void anmeldenFuellstandSensor (FuellstandSensor fuellstandSensor)`
- `public int fuellen()`
- `private int maxVolumen`

- `private int tankinhalt`
- `private FuellstandSensor fuellstandSensor`

Die Klasse `Tanksaeule` enthält folgende Methoden und Instanzvariablen:

- `public Tanksaeule()`
- `public void tankstutzenEntnehmen (Tank tank)`
- `public void startTanken()`
- `private Tank tank`
- `private boolean stopFuellen`

Die Schnittstelle `FuellstandSensor` soll folgende Methode enthalten:

```
public void meldeFuellstand (int fuellstand, int maxVolumen)
```

Der Tankvorgang wird mit dem Entnehmen des Tankstutzens aus der Tanksäule mit der Methode `tankstutzenEntnehmen()` eingeleitet. Anschließend beginnt durch Aufruf der Methode `startTanken()` das Befüllen. In `startTanken()` muss zuerst ein `FuellstandSensor` mit Hilfe einer anonymen Klasse erzeugt und der Methode `anmeldenFuellstandSensor()` übergeben werden. Nach Aufruf der Methode `anmeldenFuellstandSensor()` kann der eigentliche Füllvorgang beginnen. Durch Aufruf der Methode `fuellen()` wird der Tank jedes Mal um einen Liter gefüllt. Durch wiederholten Aufruf der Methode `fuellen()` wird der Tank aufgefüllt, bis das Maximalvolumen des Tanks erreicht ist. Nach jedem Auffüllen des Tanks um einen weiteren Liter wird die Methode `meldeFuellstand()` der anonymen Klasse – welche die Schnittstelle `FuellstandSensor` implementiert – aufgerufen. Innerhalb dieser Methode wird überprüft, ob der Tank vollständig gefüllt wurde. Ist dies der Fall, so wird der Tankvorgang beendet. Zusätzlich gibt die Methode `meldeFuellstand()` den aktuellen Füllstand auf der Konsole aus.

Testen Sie die entwickelten Klassen mit folgender Klasse:

```
// Datei: TestTanken.java
public class TestTanken
{
   public static void main (String[] args)
   {
      Tanksaeule tanksaeule = new Tanksaeule();
      Tank tank = new Tank (50);
      tanksaeule.tankstutzenEntnehmen (tank);
      tanksaeule.startTanken();
   }
}
```

Aufgabe 15.3: Elementklasse

Es soll eine Liste zur Archivierung einer CD-Sammlung entwickelt werden. Hierzu wird die Klasse `CdListe` entwickelt. Diese Klasse besitzt folgende Methoden:

- `public CdListe (String archivTitel, int maxAnzahl)`
- `public void cdHinzufuegen (String cdTitel, String kuenstler, int jahr)`
- `public void listeAnzeigen()`

CDs sollen zum Archiv nur hinzugefügt werden können. Des Weiteren besteht die Möglichkeit, die gesamte Liste der im Archiv aufgenommenen CDs auf der Konsole auszugeben.

Die Klasse `CdListe` enthält noch eine innere Klasse `Cd`. Diese innere Klasse `Cd` ist als Elementklasse zu realisieren und besitzt die Methoden:

- `Cd (String cdTitel, String kuenstler, int jahr)`
- `public String toString()`

Entwickeln Sie die Klasse `CdListe`. Beim Hinzufügen einer CD soll ein neues Objekt der inneren Klasse `Cd` erzeugt und zu der Liste hinzugefügt werden. Die Ausgabe der CD-Liste erfolgt unsortiert auf der Konsole durch den Aufruf der Methode `listeAnzeigen()`.

Die Klasse `CdListe` könnte mit folgender Klasse getestet werden:

```
// Datei: TestCdListe.java
public class TestCdListe
{
   public static void main (String[] args)
   {
      CdListe liste = new CdListe ("Klassik", 3);
      liste.listeAnzeigen();

      liste.cdHinzufuegen ("Zauberflöte", "Mozart", 2003);
      liste.cdHinzufuegen ("Nussknacker", "Tschaikowsky", 2001);
      liste.listeAnzeigen();

      liste.cdHinzufuegen ("Für Elise", "Beethoven", 1990);
      liste.listeAnzeigen();
   }
}
```

Aufgabe 15.4: Statische innere Klasse

Schreiben Sie eine Schnittstelle `Obst`, welche die Methoden `getObstname()` und `getAnzahl()` deklariert. Weiter soll die Schnittstelle noch eine statische geschachtelte Klasse `Obstmengenausgabe` enthalten, welche die statische Methode `print (Obst obst)` implementiert. Die Methode `print (Obst obst)` soll die Art des Obstes und die Anzahl der entsprechenden Früchte auf der Konsole ausgeben. Implementieren Sie zusätzlich die Klassen `Apfel` und `Birne`, welche jeweils die Schnittstelle `Obst` implementieren. Im Konstruktor soll die Anzahl der Birnen oder Äpfel mit übergeben werden. Testen Sie ihre Klassen mit folgender Testklasse:

```
// Datei: TestObst.java
public class TestObst
{
   public static void main (String[] args)
   {
      Obst obst1 = new Apfel (9);
      Obst.Obstmengenausgabe.print (obst1);
      Obst obst2 = new Birne (7);
      Obst.Obstmengenausgabe.print (obst2);
```

```
    }
}
```

Aufgabe 15.5: Flughafen-Projekt – Aufräumen

Ziel dieser Aufgabe ist, eine neue Klasse `Flughafen` einzuführen. Diese Klasse `Flughafen` soll das Erzeugen und Verwalten aller für den Flughafen notwendigen Objekte übernehmen. Es folgt die grafische Repräsentation dieser Klasse:

Flughafen
- bahnen : Bahn[] - parkpositionen : Parkposition[] - simulator : FlugzeugSimulator
+ Flughafen() + aktualisiereStatus() + erstelleFluggesellschaft() + erstelleFlugzeug() + erstelleFlugzeugtyp() + getAnzahlBahnen() + getAnzahlParkpositionen() + getBahn() + getParkposition() + getSeparateParkposition() + getWerft() + setFlugzeugListener()

Die Klasse `Flughafen` besitzt die folgenden Datenfelder und Methoden:

- ein Array für die Start- und Landebahnen (`Bahn[]`), ein Array für die Parkpositionen (`Parkposition[]`) und eine Referenz auf das Objekt vom Typ `FlugSimulator`,
- einen Konstruktor, um die drei Referenzen `bahnen`, `parkpositionen` und `simulator` zu initialisieren,
- die Methode `aktualisiereStatus()`, welche den Aufruf an den Flugzeugsimulator durchreicht,
- die Methoden `erstelleFluggesellschaft()`, `erstelleFlugzeug()` und `erstelleFlugzeugtyp()` zur Erstellung der entsprechenden Objekte,
- die zwei Methoden `getAnzahlBahnen()` und `getAnzahlParkpositionen()`, welche die jeweilige Anzahl zurückgeben,
- die Methoden `getBahn()`, `getParkposition()`, `getSeparateParkposition()` und `getWerft()`, um eine Referenz auf das angeforderte Objekt zu erhalten[194] und
- die Methode `setFlugzeugListener()`, um eine Klasse, die eine Schnittstelle `FlugzeugListener` implementiert, beim Flugzeugsimulator anzumelden.

Ändern sie die Klasse `Client` so ab, dass diese mit der neuen Klasse `Flughafen` arbeitet. Bitte beachten Sie, dass diese Projektaufgabe **keine geschachtelten Klassen** enthält.

[194] Dabei muss den Methoden `getBahn()` und `getParkposition()` ein Index übergeben werden, da es mehrere Bahnen und Parkpositionen gibt.

Kapitel 16

Threads

16.1 Prozesskonzept für Betriebssystem-Prozesse
16.2 Zugriffe auf Betriebsmittel und Synchronisation
16.3 Schwergewichtige und leichtgewichtige Prozesse
16.4 Zustände und Zustandsübergänge von Betriebssystem-Prozessen
16.5 Zustände und Zustandsübergänge von Threads
16.6 Programmierung von Threads
16.7 Scheduling von Threads
16.8 Zugriff auf gemeinsame Ressourcen
16.9 Dämon-Threads
16.10 Zusammenfassung
16.11 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_16

16 Threads

Bei vielen Anwendungen ist es wünschenswert, dass ein Benutzer an seinem Computer parallel verschiedene Aufgaben durchführen kann. So möchte z. B. ein Nutzer eine Datei aus dem Internet laden, während er zugleich einen Text in einem anderen Fenster des Bildschirms schreibt. Er wäre – wenn er nur einen einzigen Prozessor hat – überhaupt nicht zufrieden, wenn er während des Ladevorgangs seine sonstigen Aktivitäten am Rechner einstellen und untätig auf den Abschluss des Ladens warten müsste.

Hat man mehrere physische Prozessoren zur Verfügung, so kann man Programme, die nicht voneinander abhängig sind, tatsächlich unabhängig auf verschiedenen Prozessoren ablaufen lassen. Da das Laden einer beliebigen Datei und das Schreiben eines Textes nichts miteinander zu tun hat – es sei denn, der Inhalt der geladenen Datei soll in den Text übernommen werden –, ist in obigem Beispiel eine parallele Abarbeitung auf einem Mehrprozessorsystem tatsächlich hilfreich.

Mehrprozessorsysteme sind auf jeden Fall bei allen Anwendungen nützlich, die **nebenläufig** (**concurrent**) sind, d. h., die unabhängig voneinander ausgeführt werden können. Aber auch **Einprozessorsysteme** unterstützen **nebenläufige Anwendungen**. Darauf soll in diesem Kapitel eingegangen werden.

Kapitel 16.1 erläutert das Prozesskonzept für Betriebssystem-Prozesse als virtuelle Prozessoren, durch welches es möglich wurde, mehrere Nutzer zur gleichen Zeit parallel auf einem einzigen Prozessor arbeiten zu lassen. In Kapitel 16.2 werden die Zugriffe auf Betriebsmittel erörtert und die Kategorien "exklusiv benutzbare" und "aufteilbare" Betriebsmittel betrachtet. Anschließend werden in Kapitel 16.3 Prozesse in leichtgewichtige und schwergewichtige Prozesse eingeteilt, wobei schwergewichtige Prozesse den Betriebssystem-Prozessen entsprechen und leichtgewichtige den Threads. In Kapitel 16.4 werden Zustände und Zustandsübergänge von Betriebssystem-Prozessen und in Kapitel 16.5 von Threads betrachtet. Wie Threads in Java programmiert werden, wird in Kapitel 16.6 gezeigt. Kapitel 16.7 betrachtet das Scheduling von Threads und den Einfluss von Prioritäten. Kapitel 16.8 analysiert die Synchronisation des Zugriffs auf gemeinsame Ressourcen und hierbei insbesondere das Semaphorkonzept und das Konzept eines Monitors. Kapitel 16.9 untersucht die Rolle von Dämon-Threads.

16.1 Prozesskonzept für Betriebssystem-Prozesse

Hat man nur einen einzigen Prozessor, so kann tatsächlich zu einem bestimmten Zeitpunkt nur ein einziges Programm den Prozessor besitzen, d. h. verschiedene Programme können nur abwechselnd auf dem Prozessor ablaufen.

Bis in die sechziger Jahre des zwanzigsten Jahrhunderts waren die Betriebssysteme von Rechnern sogenannte **batch-Betriebssysteme**, bei denen ein Programm, das den Prozessor besaß, komplett ablaufen musste, ehe das nächste Programm den Prozessor erhalten konnte. An ein paralleles Arbeiten mehrerer interaktiver Anwen-

der auf einem einzigen Rechner war damals überhaupt nicht zu denken, da ein paralleles Bedienen mehrerer Anwender durch einen einzigen Prozessor technisch noch gar nicht möglich war. Der Programmablauf war tatsächlich streng sequenziell. Dies symbolisiert das folgende Bild:

Bild 16-1 Abarbeitung von Programmen bei einem batch-Betriebssystem

Das **Konzept eines Betriebssystem-Prozesses** brachte für das nebenläufige Arbeiten den Durchbruch und ermöglichte es, dass mehrere Nutzer gleichzeitig arbeiten konnten.

Ein Betriebssystem-Prozess virtualisiert den physischen Prozessor und multiplexed ihn. Ein **Betriebssystem-Prozess** ist hierbei definiert als "**ein Programm in Ausführung**" **oder** als "**ein Programm, das laufen möchte**".

Dabei ist bei dieser Definition vorausgesetzt, dass ein solches Programm selbst nur sequenzielle Abläufe enthält.

Die einfache Idee war, ein Programm unterbrechbar zu machen, das heißt, es sollte möglich sein, zur Laufzeit eines Betriebssystem-Prozesses diesem die Ressource (das physische Betriebsmittel) Prozessor zu entziehen, für eine kurze Zeit dann einem anderen Betriebssystem-Prozess den Prozessor zu geben und so abwechselnd nach einer gewissen Strategie verschiedene Betriebssystem-Prozesse zu bedienen. Findet der Wechsel zwischen den Betriebssystem-Prozessen nur schnell genug statt, so merkt ein Beobachter eines Betriebssystem-Prozesses gar nicht, dass diesem momentan der Prozessor gar nicht gehört. Für einen Beobachter sieht es so aus, als würden alle Betriebssystem-Prozesse **quasi parallel** ablaufen. So gut sich diese Idee anhört, so aufwendig ist sie in der Praxis umzusetzen, denn:

Ein Betriebssystem-Prozess darf gar nicht merken, dass er unterbrochen worden ist. Wird ihm der Prozessor wieder zugeteilt, so muss der Betriebssystem-Prozess in genau derselben Weise weiterarbeiten, als hätte er die ganze Zeit den Prozessor besessen.

Betriebssystem-Prozesse sind also in erster Linie ein Mittel zur Strukturierung von nebenläufigen Programmsystemen. Dabei kann man sich einen jeden Betriebssystem-Prozess als einen eigenen **virtuellen Prozessor** vorstellen.

Betriebssystem-Prozesse können also

- **parallel** von mehreren Prozessoren
- oder abwechselnd von einem einzigen Prozessor (**quasi-parallel**)

ausgeführt werden.

Letztendlich ermöglicht ein Betriebssystem, welches ein Betriebssystem-Prozesskonzept unterstützt und **Betriebssystem-Prozesse** als sogenannte **virtuelle Betriebsmittel**[195] zur Verfügung stellt, ein **Multiplexen** des Prozessors.

Nach einer gewissen vorgegebenen Strategie erhalten die Betriebssystem-Prozesse abwechselnd den Prozessor, wobei sie, wenn sie den Prozessor wieder erhalten, nahtlos so weiterlaufen, als hätten sie den Prozessor nie abgegeben.

16.1.1 Time Sharing-Betriebssysteme

Die ersten Betriebssysteme, die ein Betriebssystem-Prozesskonzept unterstützten, waren die sogenannten **Zeitscheiben-Betriebssysteme** (**Time Sharing-Betriebssysteme**), bei denen jeder Betriebssystem-Prozess periodisch vom **Scheduler**[196] eine bestimmte **Zeitscheibe** (**Time Slice**) lang den Prozessor zur Verfügung gestellt bekam. Das folgende Bild symbolisiert die Arbeitsweise eines Time Sharing-Betriebssystems:

Bild 16-2 Time Sharing-Betriebssystem

Erhält ein Betriebssystem-Prozess zum ersten Mal eine Zeitscheibe, so beginnt er zu laufen. Ist das Ende der Zeitscheibe erreicht, so wird ihm der Prozessor entzogen (**preemptive scheduling**[197]). Erhält er die nächste Zeitscheibe, so arbeitet er exakt an derjenigen Stelle weiter, an der er unterbrochen worden ist. Dies muss das Betriebssystem bewerkstelligen.

16.1.2 Der Prozesskontext

Ein jeder Betriebssystem-Prozess hat seinen eigenen **Prozesskontext**. Zu einem Prozesskontext eines in einer klassischen Programmiersprache wie C geschriebenen Programms gehört:

- der **eigentliche Programmtext** (Programmcode)
- mit **Daten**,
- **Stack**
- und **Heap**.

---

[195] Virtuell im Gegensatz zu dem **physischen Betriebsmittel Prozessor**.

[196] Der Scheduler ist eine Komponente des Betriebssystems, die den Prozessor nach einer vorgegebenen Strategie wie z. B. dem Zeitscheibenverfahren vergibt.

[197] **Preemptive Scheduling** bedeutet Scheduling durch Entzug des Prozessors. Ein Betriebssystem-Prozess kann nicht mitbestimmen, wann ihm der Prozessor entzogen wird, sondern das Betriebssystem ist in der Lage, den Prozessor gezielt nach einer gewissen Strategie dem Betriebssystem-Prozess zu entziehen.

Dies ist in folgendem Bild skizziert:

Bild 16-3 Segmente eines ablauffähigen C-Programms

Ferner gehört zum Prozesskontext die **Prozessumgebung** beispielsweise mit:

- **Registerinhalten** wie dem Stackpointer[198], dem Befehlszeiger[199] oder temporären Daten,
- geöffneten Dateien
- sowie **weiteren Informationen**, die zur Ausführung des Programms benötigt werden.

Will das Betriebssystem einen Betriebssystem-Prozess vom Prozessor nehmen und dem Prozessor einen anderen Betriebssystem-Prozess zuweisen, so findet ein sogenannter **Kontextwechsel** statt.

Hierbei muss der Kontext des alten Betriebssystem-Prozesses komplett gerettet werden, damit bei einer erneuten Vergabe des Prozessors an den alten Betriebssystem-Prozess dessen gesamter Kontext in identischer Form wiederhergestellt werden kann und zwar so, als hätte der Betriebssystem-Prozess nie den Prozessor abgeben müssen. Wegen des hohen Aufwands für den Kontextwechsel wird ein Betriebssystem-Prozess auch als **schwergewichtiger Prozess** bezeichnet.

16.2 Zugriffe auf Betriebsmittel und Synchronisation

All das, **was** ein **Betriebssystem-Prozess zum Laufen braucht**, wird als **Betriebsmittel** bezeichnet. Betriebsmittel können z. B. der Prozessor oder ein I/O-Kanal eines Programms zu einer Datei auf der Festplatte sein.

Hierbei sind I/O-Kanäle[200] Eingabe- bzw. Ausgabeströme. Eingabeströme können beispielsweise von der Tastatur oder der Platte kommen, Ausgabeströme können auf den Bildschirm oder die Platte gehen.

[198] Der Stackpointer zeigt auf die Spitze des Stacks.
[199] Der Befehlszeiger zeigt auf die als nächste abzuarbeitende Anweisung.
[200] I/O ist die Abkürzung für Input/Output.

16.2.1 Exklusiv benutzbare und aufteilbare Betriebsmittel

Betriebsmittel können **exklusiv benutzbar** sein, aber dennoch **zeitlich aufteilbar** wie der Prozessor.

Betriebsmittel können **räumlich aufteilbar** sein wie z. B. die Festplatte oder der Arbeitsspeicher.

Betriebsmittel können auch nur **exklusiv benutzbar** und **nicht aufteilbar** sein wie z. B. ein Drucker.

Bei einem Drucker macht es keinen Sinn, dass er von mehreren Prozessen (Betriebssystem-Prozessen bzw. sogenannten Threads[201]) parallel benutzt wird. Würde von verschiedenen Prozessen abwechselnd das Papier des Druckers beschrieben, so würde keine sinnvolle Ausgabe entstehen. Ein Drucker muss **exklusiv benutzt** werden. Der Zugriff auf den Drucker muss also synchronisiert werden.

Das gleiche Problem eines **exklusiven Zugriffs** gibt es auch bei **globalen Daten**, auf die von mehreren Prozessen zugegriffen werden kann, aber auch bei Zugriffen auf **Funktionen**[202].

16.2.2 Synchronisation

Synchronisation umfasst Konkurrenz und Kooperation.

16.2.2.1 Synchronisation von Prozessen

Eine weitere Problemstellung, die zur einer Synchronisation führt, kann außer einem **exklusiven Zugriff bei Konkurrenz** sein, dass Prozesse eine Aufgabe gemeinsam bearbeiten und dass dabei eine **definierte Reihenfolge der Prozesse** zwingend notwendig ist (**Kooperation**), wie beispielsweise Einlese-Prozess, Verarbeitungs-Prozess, Ausgabe-Prozess.

In all diesen Fällen eines exklusiven Zugriffs oder einer definierten Reihenfolge müssen Prozesse synchronisiert werden.

16.2.2.2 Synchronisation von Teilfolgen von Anweisungen als kritische Abschnitte

Ein Prozess selbst besteht aus einem zeitlich geordneten Ablauf von **Teilfolgen von Anweisungen**.

Probleme zwischen verschiedenen Prozessen kann es nur geben, wenn Teilfolgen verschiedener Prozesse auf exklusiv genutzte Betriebsmittel wie globale Variablen

[201] Parallelen leichtgewichtigen Einheiten innerhalb eines Betriebssystem-Prozesses

[202] Es sei denn, diese werden wiedereintrittsfähig (engl. reentrant) geschrieben und legen die Zwischenergebnisse eines jeden Prozesses in einen eigenen Speicherbereich ab.

zugreifen wollen oder wenn Teilfolgen verschiedener Prozesse in bestimmten zeitlichen Reihenfolgen ausgeführt werden müssen.

Eine **Synchronisation** dient zur Sicherstellung von zeitlichen Beziehungen zwischen Teilfolgen verschiedener Prozesse.

Bei der **Kooperation von Teilfolgen** wird eine definierte Reihenfolge von Teilfolgen verschiedener Prozesse erzwungen.

Beim **wechselseitigen Ausschluss** kann die Reihenfolge von Teilfolgen verschiedener Prozesse **beliebig** sein, nur dürfen sie nicht gleichzeitig vorkommen, d. h. sie schließen sich wechselseitig aus.

Teilfolgen, die sich **wechselseitig ausschließen**, heißen **kritische Bereiche** (**kritische Abschnitte**, **critical sections**). Kritische Abschnitte sind kritisch in dem Sinn, dass gleichzeitig nur ein einziger Prozess einen kritischen Abschnitt bearbeiten kann.

Dabei muss ein Prozess, der einen kritischen Abschnitt betritt, diesen auch vollkommen abarbeiten. Erst dann darf ein anderer Prozess einen kritischen Abschnitt betreten. Die einfachste Möglichkeit, den wechselseitigen Ausschluss von kritischen Abschnitten auf einem Einprozessor-Rechner zu realisieren, ist es, den kritischen Abschnitt ununterbrechbar zu machen. Dies hätte natürlich zur Konsequenz, dass während der Abarbeitung des kritischen Abschnitts alle anderen Prozesse warten müssten. Dieses Mittel ist nur für kurze und erprobte Betriebssystemroutinen denkbar, ist aber ansonsten nicht brauchbar. So könnte ein unwichtiger Prozess einen wichtigeren Prozess oder eine fehlerhafte Routine in einer Endlos-Schleife das ganze System blockieren.

16.3 Schwergewichtige und leichtgewichtige Prozesse

Bei Prozessen unterscheidet man schwer- und leichtgewichtige Prozesse. Darauf wird in Kapitel 16.3.1 und Kapitel 16.3.2 eingegangen.

16.3.1 Schwergewichtige Prozesse

Ein **klassischer Betriebssystem-Prozess** stellt eine **Einheit** sowohl für das **Memory Management** als auch für das **Scheduling** dar. Er wird als **schwergewichtiger Prozess** (engl. **heavyweight process**) bezeichnet.

Da das Memory Management aufwendig ist, werden Betriebssystem-Prozesse als schwergewichtige Prozesse bezeichnet.

Einem Betriebssystem-Prozess wird vom Memory Management zur gegebenen Zeit ein Platz im Arbeitsspeicher zugeordnet. Der Scheduler gewährt einem Betriebssystem-Prozess Rechenzeit.

16.3.2 Leichtgewichtige Prozesse

Bei modernen Betriebssystemen gibt es außer Betriebssystem-Prozessen auch Threads.

Ein **Thread** ist **nur** eine **Einheit für das Scheduling**, nicht jedoch für das Memory Management.

Das Wort **Thread** steht im Englischen für Faden. Hierbei ist der Ablauffaden des Programmcodes gemeint, sprich der Kontrollfluss.

Innerhalb eines Betriebssystem-Prozesses können mehrere Threads laufen. Während einem Betriebssystem-Prozess Speicher zugeordnet ist und ein Kontextwechsel – ein anderer Betriebssystem-Prozess erhält die CPU – mit Aufwand beim Memory Management verbunden ist, ist ein Wechsel eines Threads auf der CPU nicht mit der Verwaltung des Speichers gekoppelt.

Ein Thread wird als leichtgewichtiger Prozess (lightweight process) bezeichnet.

Das Konzept war es also, **innerhalb eines Betriebssystem-Prozesses** diese neuartigen Prozesse – **Threads** genannt – einzuführen, die quasi parallel ablaufen können. So können solche Threads beispielsweise in einem Server-Betriebssystem-Prozess verschiedene Nutzeranfragen quasi parallel abarbeiten (engl. **Multithreading**).

Damit diese Threads unabhängig voneinander arbeiten können, braucht man für **jeden Thread**:

- einen Befehlszeiger,
- einen eigenen Stack
  - für die Speicherung der lokalen Variablen,
  - der Übergabeparameter
  - und des Befehlszeigers zum Rücksprung bei Funktionsaufrufen, um Funktionen unabhängig von anderen Threads aufrufen zu können
- sowie einen Satz von Prozessorregistern.

Alle anderen Informationen werden geteilt, insbesondere **Programmcode**, **Programmdaten** und **Dateiinformationen**. Diese stellen **gemeinsame Daten für alle Threads** dar.

Threads in Java

Da Threads ein Sprachmittel von Java sind, muss es möglich sein, Threads zu unterstützen, ganz unabhängig davon, ob das jeweilige Betriebssystem nur ein Prozesskonzept für Betriebssystem-Prozesse oder auch für Threads unterstützt. Wie die Java-Virtuelle Maschine die Threads in Zusammenarbeit mit dem jeweiligen Betriebssystem verwaltet, bleibt dem Anwender verborgen[203].

---

[203] Unterstützt das Betriebssystem kein Prozesskonzept für Threads, so erfolgt die Threadverwaltung allein durch die Java-Virtuelle Maschine. Man spricht dann von "green threads" [Oak98]. Hat das Betriebssystem die Fähigkeit der Threadverwaltung, so spricht man bei den Java-Threads von "na-

Eine **Java-Virtuelle Maschine** selbst läuft **in einem Betriebssystem-Prozess** ab, d. h. sollen mehrere Java-Programme in getrennten Betriebssystem-Prozessen ablaufen, so hat jeder Betriebssystem-Prozess seine eigene Java-Virtuelle Maschine.

Es ist nicht möglich, eine gemeinsame Java-Virtuelle Maschine für getrennte Betriebssystem-Prozesse ablaufen zu lassen.

Das folgende Bild symbolisiert Betriebssystem-Prozesse und Threads:

Bild 16-4 Threads und Prozesse

Threads in Java **teilen sich**, da sie im selben Betriebssystem-Prozess ablaufen:

- den **Heap** für die Ablage von Objekten,
- **Code** und **Klassenvariablen** in der Method-Area
- sowie **I/O-Kanäle**.

Ein **Thread selbst hat**:

- einen **eigenen Befehlszähler**,
- einen **eigenen Registersatz**
- und einen **eigenen Stack** zur Ablage der lokalen Daten, der Übergabeparameter und des Befehlszeigers zum Rücksprung bei Methodenaufrufen.

tive threads". In der Zwischenzeit unterstützen alle gängigen Betriebssysteme Threads, so dass bei den gängigen Betriebssystemen Threads nicht mehr von der virtuellen Maschine nachgebildet werden müssen.

16.4 Zustände und Zustandsübergänge von Betriebssystem-Prozessen

Betriebssystem-Prozesse haben Zustände. Der Zustand eines Betriebssystem-Prozesses hängt davon ab, welche Betriebsmittel er momentan besitzt.

In Bild 16-5 wird ein **vereinfachtes Zustandsübergangsdiagramm** für Betriebssystem-Prozesse vorgestellt. Jeder Kreis stellt einen Zustand eines Betriebssystem-Prozesses dar. Die Pfeile kennzeichnen die Übergänge zwischen den Zuständen. Es gibt folgende Zustände und Zustandsübergänge in Bild 16-5:

- Hat ein Betriebssystem-Prozess alle Betriebsmittel, die er braucht, um laufen zu können, bis auf den Prozessor, so ist er im Zustand "**ready-to-run**".
- Erhält ein Betriebssystem-Prozess im Zustand "**ready-to-run**" vom Scheduler den Prozessor zugeteilt, so geht er in den Zustand "**running**" über.
- Wird einem Betriebssystem-Prozess im Zustand "**running**" der Prozessor entzogen, so geht er in den Zustand "**ready-to-run**" über.
- Macht ein laufender Betriebssystem-Prozess eine I/O-Operation, so verliert er den Prozessor und geht in den Zustand "**blocked**" über.
- Ist die I/O-Operation beendet, so geht der Betriebssystem-Prozess in den Zustand "**ready-to-run**" über.

Das folgende Bild zeigt ein vereinfachtes Zustandsdiagramm für Betriebssystem-Prozesse:

Bild 16-5 Vereinfachtes Zustandsübergangsdiagramm

Nur ein Betriebssystem-Prozess, der alle Betriebsmittel bis auf den Prozessor hat, das heißt im Zustand "ready-to-run" ist, kann am Wettbewerb um den Prozessor teilnehmen.

16.5 Zustände und Zustandsübergänge von Threads

Threads können ähnlich wie Betriebssystem-Prozesse verschiedene Zustände haben. Zustandsübergänge können erfolgen:

- als Konsequenz von Methodenaufrufen wie z. B. `sleep()`,
- aber auch durch Aktionen des Betriebssystems wie z. B. die Zuteilung des Prozessors durch den Scheduler.

Bei Threads müssen die 5 Zustände "new", "ready-to-run", "blocked", "running" und "dead" betrachtet werden.

Die Zustände "ready-to-run", "blocked" und "running" wurden bereits im vorangehenden Kapitel erklärt. Der Zustand "**new**" bedeutet, dass ein Thread durch den `new`-Operator generiert wurde und sich in seinem Anfangszustand befindet. Er ist noch nicht ablauffähig. Seine Datenfelder und Methoden können jedoch angesprochen werden. In den Zustand "**dead**" gelangt ein Thread nach Abarbeitung seines Programmcodes.

Im Zustand "dead" können dann weiterhin die Datenfelder und eigene Methoden – bis auf die Methode `run()` – des Threads angesprochen werden. Ein Thread, der einmal den Zustand "dead" erreicht hat, kann jedoch nicht wieder gestartet werden.

Der Übersichtlichkeit halber werden Zustandsübergänge, die aus Methodenaufrufen resultieren, und Zustandsübergänge, die durch die Java-Virtuelle Maschine verursacht werden, im Folgenden in getrennten Grafiken dargestellt:

- **Zustandsübergänge als Folge von Methodenaufrufen**

  In Bild 16-6 ist zu sehen, welche Zustandsübergänge von Threads explizit durch Methodenaufrufe hervorgerufen werden können:

Bild 16-6 Zustandsübergänge[204] von Threads als Folge von Methodenaufrufen

Es fällt dabei auf, dass es keinen Pfeil zum Zustand "running" gibt. Dies liegt daran, dass nur die Java-Virtuelle Maschine (genauer gesagt der Scheduler) einen Thread in den Zustand "running" versetzen kann. Ein Thread kann nicht per

[204] Die Methoden `notify()`, `notifyAll()` und `wait()` sind Methoden der Klasse `Object` und dürfen nur in Codeblöcken aufgerufen werden, die als `synchronized` gekennzeichnet sind. Siehe hierzu Kapitel 16.8.7.

Methodenaufruf in den Zustand "running" versetzt werden. Der Zustand "dead" wird in Bild 16-6 gar nicht aufgeführt, da man einen Thread durch Methodenaufrufe weder in den Zustand "dead" überführen kann, noch den Zustand "dead" verlassen kann.

Nachfolgend werden die Methoden der Klasse `Thread`, die einen Zustandsübergang eines Threads bewirken können, genauer erläutert:

- `public void start()`

  Der Aufruf der Methode `start()` überführt einen Thread vom Zustand "new" in den Zustand "ready-to-run". Wurde der Thread schon einmal gestartet, so wird eine Exception vom Typ **`IllegalThreadStateException`** geworfen.

- `public static void sleep (long millis)`
  `public static void sleep (long millis, int nanos)`

  Der Aufruf der Methode `sleep()` versetzt den gerade laufenden Thread für mindestens `millis` msec (bzw. `millis` msec und `nanos` nsec) in den Zustand "blocked". Die präzise Auflösung in Schritte von 1 ms (bzw. 1 ns) ist dabei jedoch nicht gewährleistet, sondern hängt vom Betriebssystem ab. Oft wird der Wert entsprechend auf- oder abgerundet. Beide `sleep()`-Methoden können eine Exception vom Typ **`InterruptedException`** werfen. Dies tritt genau dann ein, wenn ein Thread, der sich durch den Aufruf der Methode `sleep()` im Zustand "blocked" befindet, durch den Aufruf der Instanzmethode **`interrupt()`** in den Zustand "ready-to-run" überführt wird. Die `interrupt()`-Methode muss dabei natürlich von einem anderen gerade laufenden Thread aufgerufen werden.

- `public static void yield()`

  Der Aufruf von `yield()` bricht die Verarbeitung des gerade laufenden Threads ab und überführt diesen wieder in den Zustand "ready-to-run", wo er erneut auf die Zuteilung von Rechenzeit warten muss. Der Aufruf dieser Methode für einen Thread im Zustand "running" gibt anderen Threads die Möglichkeit zum Ablauf.

- `public final void join()`
  `public final void join (long millis)`
  `public final void join (long millis, int nanos)`

  Ein Thread kann die Methode `join()` eines anderen Threads aufrufen. Hierbei wird der Thread, der die Methode aufruft, in den Zustand "blocked" versetzt, bis der Thread, dessen `join()`-Methode aufgerufen wird, beendet ist. Somit kann gezielt auf das Ende eines Threads gewartet werden. Muss Thread1 z. B. auf die Beendigung von Thread2 warten, so ruft Thread1 die `join()`-Methode von Thread2 auf. Dadurch wird Thread1 solange in den Zustand "blocked" versetzt, bis Thread2 beendet ist. Wird die Methode `join()` eines bereits beendeten Threads aufgerufen, so wird der aufrufende Thread nicht in den Zustand "blocked" versetzt. Werden die `join()`-Methoden, welche die Angabe einer Wartezeit erlauben, verwendet, so wartet ein Thread entweder auf das Ablaufen der Wartezeit oder auf das tatsächliche Beenden des Threads, dessen `join()`-Methode aufgerufen wurde. Die Methode `join()` kehrt auf jeden Fall

nach dem Ablauf der Wartezeit zurück, auch wenn der Thread noch nicht beendet ist. Alle drei `join()`-Methoden können ebenso wie die `sleep()`-Methoden eine Exception vom Typ **`InterruptedException`** werfen. Dies tritt genau dann ein, wenn der durch den Aufruf der `join()`-Methode wartende Thread durch einen Aufruf der `interrupt()`-Methode unterbrochen wird.

- `public void interrupt()`

  Die Methode `interrupt()`, die zu einem gerade blockierten Thread aufgerufen wird, überführt diesen Thread in den Zustand "ready-to-run". Versetzt sich also ein Thread freiwillig – zum Beispiel durch den Aufruf der Methode `sleep()` oder der Methode `join()` – in den Zustand "blocked", so kann man diesen Thread wieder vorzeitig aufwecken – das heißt, in den Zustand "ready-to-run" versetzen – indem für diesen Thread die Methode `interrupt()` aufgerufen wird. Damit wird der blockierende Methodenaufruf wie `join()` oder `sleep()` beendet und die Methode `run()` arbeitet mit dem `catch`-Block hinter der blockierenden Methode weiter.

- **Zustandsübergänge durch die Java-Virtuelle Maschine**

  Die Zustandsübergänge in Bild 16-7 werden automatisch von der Java-Virtuellen Maschine aufgrund von bestimmten Ereignissen vollzogen. Ein Programmierer hat nur indirekt Einfluss auf die Zustandsübergänge, z. B. durch Dateizugriff, durch Setzen von Prioritäten oder durch Beenden der Methode `run()`. Das Verlassen des Zustandes "blocked", der durch den Aufruf der Methoden `sleep()` oder `join()` betreten wurde, kann durch den Aufruf der Methode `interrupt()` beschleunigt werden.

  Hier die von der JVM verursachten Zustandsübergänge von Threads:

Bild 16-7 Zustandsübergänge[205] von Threads verursacht durch die Java-Virtuelle Maschine

[205] Die Synchronisation von Codeblöcken wird in Kapitel 16.8.4 erläutert.

Befindet sich ein Thread in einem der Zustände "new", "ready-to-run", "blocked" oder "running", so sagt man, dass der Thread "alive" ist.

Durch den Aufruf der Instanzmethode `isAlive()` der Klasse `Thread` kann geprüft werden, ob ein Thread gerade "alive" ist. Ist ein Thread, für den diese Methode aufgerufen wird, "alive", so wird `true` zurückgegeben, andernfalls `false`. Dabei ist ein Thread "alive" vom Zeitpunkt seiner Generierung durch `new` bis zum endgültigen Erreichen des Zustandes "dead".

16.6 Programmierung von Threads

Threads lassen sich in Java, da sie bereits im Sprachumfang zur Verfügung gestellt werden, sehr einfach programmieren, erzeugen und starten.
:

Es gibt zwei Möglichkeiten, einen Thread zu programmieren

- durch eine direkte Ableitung der Klasse dieses Threads von der Klasse **`Thread`**
- oder durch die Übergabe eines Objektes, dessen Klasse die Schnittstelle **`Runnable`** implementiert, an ein Objekt der Klasse `Thread`.

Diese beiden Möglichkeiten werden in Kapitel 16.6.1 und 16.6.2 vorgestellt. Auf das Beenden von Threads wird in Kapitel 16.6.3 eingegangen.

16.6.1 Ableiten von der Klasse `Thread`

Durch Ableiten von der Klasse `java.lang.Thread` kann eine eigene Thread-Klasse geschrieben werden. Dabei ist die Methode `run()` der Klasse `java.lang.Thread` zu überschreiben. Der in der Methode `run()` enthaltene Code wird während des "running"-Zustandes des entsprechenden Thread-Objekts ausgeführt.

Hier die Visualisierung der Ableitung:

Bild 16-8 Implementieren von Threads durch Ableiten von der Klasse `Thread`

Erzeugt wird ein Thread-Objekt mit Hilfe des `new`-Operators und zum Starten dieses Thread-Objekts wird seine Methode `start()` aufgerufen.

Die Methode `start()` **reserviert** die **Systemressourcen**, welche notwendig sind, um einen Thread zu starten. Außerdem **ruft** sie die **Methode `run()`** auf.

Im Folgenden wird anhand eines Beispiels gezeigt, wie die Klasse eines Threads durch Ableiten von der Klasse `Thread` definiert werden kann:

```
// Datei: Time.java

import java.util.*;

public class Time extends Thread
{
   public void run()
   {
      while (true) // Endlosschleife
      {
         GregorianCalendar d = new GregorianCalendar();

         // Calendar.HOUR_OF_DAY, Calendar.MINUTE und
         // Calendar.SECOND sind Konstanten der Klasse Calendar
         System.out.println (d.get (Calendar.HOUR_OF_DAY) + ":"
                           + d.get (Calendar.MINUTE) + ":"
                           + d.get (Calendar.SECOND));
         try
         {
            Thread.sleep (100);
         }
         catch (InterruptedException e)
         {
         }
      }
   }
}

// Datei: Uhr.java

public class Uhr
{
   public static void main (String[] args)
   {
      Time t = new Time();
      t.start();
      // Möglichkeit zum Erzeugen und Starten weiterer Threads.
   }
}
```

Die Ausgabe des Programms ist:

```
16:28:27
16:28:27
16:28:27
16:28:27
16:28:27
16:28:28
16:28:28
16:28:28
16:28:28
```

Im obigen Beispiel soll die Klasse `Time` eine Thread-Klasse sein, von der Threads erzeugt werden können. Ein solcher Thread wird durch Instanziieren der Klasse `Time` erzeugt. Die Klasse `Time` wird definiert, indem man direkt von der Klasse `Thread` ableitet und die `run()`-Methode überschreibt. In der `run()`-Methode der Klasse `Time` wird ein Objekt der Klasse `java.util.GregorianCalendar` verwendet. Sie enthält die aktuellen Datums- und Uhrzeitangaben. Datum und Uhrzeit werden jedoch nicht fortlaufend aktualisiert. Um die aktuelle Uhrzeit zu erhalten, muss jedes Mal ein neues Objekt der Klasse `GregorianCalendar` geschaffen werden.

Besonders einfach lassen sich der Ablauf eines Threads und die Auswirkungen der einzelnen Methoden anhand von Bild 16-6 verfolgen. In der Klasse `Uhr` des vorherigen Beispiels wird mit `new` ein Objekt der Klasse `Time` erzeugt und somit ein neuer Thread generiert. Dieser Thread befindet sich im Zustand "new". Durch den Aufruf der von der Klasse `Thread` geerbten Methode `start()` des Threads wird dieser Thread gestartet und befindet sich dann im Zustand "ready-to-run". Die Methode `start()` ruft die Methode `run()` auf. Nach der Zuteilung von Rechenzeit durch den Scheduler und dem Überführen des Threads in den Zustand "running" kommt der Thread und damit die Methode `run()` zur Ausführung. In der Methode `run()` wird zuerst die aktuelle Uhrzeit auf der Standardausgabe ausgegeben. Danach versetzt sich der Thread mit der Methode `sleep (100)` für mindestens 100 ms in den Zustand "blocked". Nach Ablauf der 100 ms geht der Thread wieder in den Zustand "ready-to-run" über und muss erneut auf die Zuteilung von Prozessorzeit warten. Da sich in der Methode `run()` eine Endlos-Schleife befindet, wird die Ausgabe fortgeführt, d. h. der Thread wird nie beendet.

Da die Methode `sleep()` eine Exception vom Typ `InterruptedException` werfen kann, benötigt man einen `try`-Block und einen `catch`-Block, um die Ausnahme abzufangen. Innerhalb des `catch`-Blocks stehen aber keine Anweisungen. Dies ist in dem vorliegenden Beispielprogramm auch vollkommen korrekt, da hier nie der Fall eintreten kann, dass eine solche Exception geworfen wird. Damit die Methode `sleep()` eine Exception vom Typ `InterruptedException` werfen würde, müsste die Methode `interrupt()` für ein Objekt der Klasse `Time` aufgerufen werden, welches zuvor durch Aufruf der Methode `sleep()` in den Zustand "blocked" versetzt wurde.

16.6.2 Implementieren der Schnittstelle `Runnable`

Im Kapitel 16.6.1 wurde eine Thread-Klasse durch direktes Ableiten von der Klasse `Thread` geschrieben. Wenn man aber zwingend von einer weiteren Klasse ableiten muss, ist dieses Vorgehen nicht möglich, da Java keine Mehrfachvererbung unterstützt.

Implementiert man die Schnittstelle `Runnable` in einer Klasse, die zum Thread werden soll, so schafft man dadurch die Möglichkeit, dass diese Klasse von einer anderen Klasse abgeleitet werden kann. **Die Schnittstelle `Runnable` deklariert nur eine einzige Methode**, nämlich die Methode **`run()`**.

Ein Thread wird erzeugt, indem man mit dem `new`-Operator eine Instanz der Klasse `java.lang.Thread` generiert und dabei als Übergabeparameter beim Konstruktoraufruf eine Referenz auf ein Objekt mitgibt, dessen Klasse die Schnittstelle `Runnable` implementiert. Das folgende Bild zeigt die Implementierung des Interface `Runnable`:

Bild 16-9 Die Klasse `Time` implementiert das Interface `Runnable`

Innerhalb der Klasse `Thread` wird die übergebene Referenz in einem privaten Datenfeld vom Typ `Runnable` abgelegt. Das folgende Codestück zeigt einen Ausschnitt aus der Implementierung der Klasse `Thread`:

```
public Thread
{
   private Runnable target;
   . . . . .
   public Thread (Runnable target)
   {
      . . . . .
      this.target = target;
      . . . . .
   }
}
```

Dadurch, dass im Konstruktoraufruf der Klasse `Thread` der formale Parameter vom Schnittstellentyp `Runnable` ist, kann der Compiler sicherstellen, dass das Objekt, auf das die übergebene Referenz zeigt, die `run()`-Methode implementiert.

Das folgende **Beispiel** zeigt, wie ein Thread mit Hilfe eines Objektes der Klasse `Time1` erzeugt wird. Die Klasse `Time1` implementiert dabei das Interface `Runnable`. Hier zunächst die Klasse `Time1`:

```
// Datei: Time1.java

import java.util.*;

public class Time1 implements Runnable
{
   public void run()
   {
      while (true) // Endlosschleife
      {
         GregorianCalendar d = new GregorianCalendar();
         System.out.println (d.get (Calendar.HOUR_OF_DAY) + ":"
                            + d.get (Calendar.MINUTE) + ":"
                            + d.get (Calendar.SECOND));
         try
         {
            // Die Methode sleep() ist eine Klassenmethode der
            // Klasse Thread.
            Thread.sleep (100);
         }
         catch (InterruptedException e)
         {
         }
      }
   }
}
```

Und jetzt die Klasse `Uhr1`:

```
// Datei: Uhr1.java

public class Uhr1
{
   public static void main (String[] args)
   {
      // Die Klasse Time1 implementiert die Schnittstelle Runnable.
      // Eine Referenz auf ein Objekt dieser Klasse kann also als
      // Konstruktorparameter bei der Erzeugung eines Objektes der
      // Klasse Thread verwendet werden.
      Thread timeThread = new Thread (new Time1());
      timeThread.start();
      // Möglichkeit zum Erzeugen und Starten weiterer Threads
   }
}
```

Mit

```
new Thread (new Time1());
```

wird ein Objekt der Klasse `Thread` erzeugt, wobei die Referenz auf das ebenfalls neu erzeugte Objekt der Klasse `Time1` an die oben erwähnte Instanzvariable `target` vom Typ `Runnable` zugewiesen wird. Wird die `start()`-Methode des erzeug-

ten Thread-Objektes mit Hilfe der Referenz `timeThread` aufgerufen, so wird von der `start()`-Methode die ausprogrammierte `run()`-Methode der Klasse `Time1` aufgerufen.

16.6.3 Beenden von Threads

Die Beispiele aus den Kapiteln 16.6.1 und 16.6.2 beinhalten noch keine Möglichkeit, einen gestarteten Thread auch wieder zu beenden. Ein Thread wird bekanntlich beendet, wenn die Abarbeitung der `run()`-Methode beendet ist. Da aber die Endlos-Schleife in der `run()`-Methode der Klasse `Time` nie aufhört, benötigt man eine Möglichkeit, die Abarbeitung der `run()`-Methode abzubrechen. Im Folgenden werden zwei Möglichkeiten vorgestellt, einen einmal gestarteten Thread auch wieder zu beenden. Hier die **erste Möglichkeit**:

```
// Datei: Time2.java
import java.util.*;

public class Time2 extends Thread
{
   public void run()
   {
      while (true)
      {
         GregorianCalendar d = new GregorianCalendar();
         System.out.println (d.get (Calendar.HOUR_OF_DAY) + ":"
                            + d.get (Calendar.MINUTE) + ":"
                            + d.get (Calendar.SECOND));
         try
         {
            Thread.sleep (100);
         }
         catch (InterruptedException e)
         {
            System.out.println ("Interrupted!");
            return;
         }
      }
   }
}

// Datei: Uhr2.java
import java.io.*;

public class Uhr2
{
   public static void main (String[] args) throws IOException
   {
      Time2 t = new Time2();
      t.start();

      // Warten, bis der Benutzer "exit" gefolgt von RETURN eingibt
      while (true)
      {
         BufferedReader reader =
             new BufferedReader (new InputStreamReader (System.in));
```

```
            String kommando = reader.readLine();
            if (kommando.equals ("exit"))
               break;
         }
         t.interrupt();
      }
}
```

An der Implementierung des Threads in der `run()`-Methode von `Time2` hat sich nicht viel geändert – lediglich im `catch`-Block wurde eine Ausgabe und die `return`-Anweisung eingefügt. Mit `return` wird aus der Methode, die den `catch`-Block enthält, zur aufrufenden Methode zurückgesprungen.

Wird die `interrupt()`-Methode für das Objekt der Klasse `Time2` aufgerufen, während seine `run()`-Methode abgearbeitet wird, so wird beim nächsten Abarbeiten der `sleep()`-Methode dieses Threads eine Exception vom Typ `InterruptedException` geworfen und die `run()`-Methode wird mit `return` beendet. Genau genommen muss man zwei Fälle unterscheiden:

- Befindet sich der Thread durch den Aufruf der Methode `sleep()` gerade im Zustand "blocked", während seine `interrupt()`-Methode aufgerufen wird, so wird der Thread in den Zustand "ready-to-run" gebracht und die Methode `sleep()` kehrt mit dem Auswerfen einer Exception vom Typ `InterruptedException` zurück.
- Bearbeitet der Thread gerade die Anweisungen vor dem Aufruf der Methode `sleep()`, während die `interrupt()`-Methode für dieses Objekt aufgerufen wird, so werden alle Anweisungen einschließlich des Aufrufs der `sleep()`-Methode abgearbeitet, wobei die `sleep()`-Methode sofort wieder durch den Auswurf einer Exception vom Typ `InterruptedException` zurückkehrt.

In der **zweiten Möglichkeit**, einen Thread zu beenden, wird ein privates Datenfeld vom Typ `boolean` benutzt. Dieses Datenfeld – im unteren Beispiel `running` genannt – wird in der Methode `beenden()` zurückgesetzt, um die `while()`-Schleife in der `run()`-Methode zu beenden. Dies zeigt das folgende Beispiel:

```
// Datei: Time3.java

import java.util.*;

public class Time3 extends Thread
{
   private boolean running = true;

   public void run()
   {
      while (running)
      {
         GregorianCalendar d = new GregorianCalendar();
         System.out.println (d.get (Calendar.HOUR_OF_DAY) + ":"
                           + d.get (Calendar.MINUTE) + ":"
                           + d.get (Calendar.SECOND));
```

```
         try
         {
            Thread.sleep (100);
         }

         catch (InterruptedException e)
         {
         }
      }
   }

   public void beenden()
   {
      running = false;
   }
}

// Datei: Uhr3.java

import java.io.*;

public class Uhr3
{
   public static void main (String[] args) throws IOException
   {
      Time3 t = new Time3();
      t.start();
      // Warten, bis der Benutzer "exit" gefolgt von RETURN eingibt
      while (true)
      {
        BufferedReader reader =
             new BufferedReader (new InputStreamReader (System.in));
        String kommando = reader.readLine();
        if (kommando.equals ("exit"))
           break;
      }
      t.beenden();
   }
}
```

Beim Testen der Beispiele wird zyklisch ungefähr alle 100 ms eine Ausgabe in das Ausgabefenster geschrieben. Die Eingabe der Buchstaben `'e'`, `'x'`, `'i'` und `'t'` gefolgt von einem `<RETURN>` ist aber trotzdem problemlos möglich, auch wenn die Buchstaben nicht zusammenhängend im Ausgabefenster gesehen werden können. Die Ursache dafür ist, dass die Methode `main()` der Klasse `Uhr3` in einem eigenen Thread parallel zu dem Thread der Klasse `Time3` läuft.

Nachdem nun bekannt ist, wie man Threads programmieren, starten und beenden kann, sei hier noch kurz darauf verwiesen, dass es in der Klassenbibliothek von Java schon einige typische Thread-Klassen – wie zum Beispiel die Klassen `Timer` und `TimerTask` – im Paket `java.util` gibt. Es lohnt sich auf jeden Fall, die Einsatzmöglichkeiten dieser Klassen mit Hilfe der Java-Dokumentation etwas genauer zu studieren. Nichtsdestotrotz muss man sich auch in den Themen der folgenden Kapitel auskennen, um selbst nebenläufige Anwendungen in Java schreiben zu können.

16.7 Scheduling von Threads

Als **Scheduling** bezeichnet man die Zuteilung von Rechenzeit auf einem Prozessor durch ein Betriebssystem oder durch eine Laufzeitumgebung.

Die Implementierung der Java-Virtuellen Maschine beim Scheduling von Threads wird von Oracle nicht genau spezifiziert.

Der Java-Run-Time-Scheduler ist **prioritätengesteuert**. Er weist Threads im Zustand "ready-to-run" Prozessorzeit zu und entzieht ihnen wieder den Prozessor.

In der Spezifikation der Java-Virtuellen Maschine wird nur verlangt, dass Threads mit höherer Priorität im Schnitt mehr Rechenzeit erhalten sollen als Threads mit niedriger Priorität.

Dies ermöglicht Freiräume für die Implementierung der Java-Virtuellen Maschine, was wiederum zu Problemen bei der Portierung eines Programms auf andere Plattformen führen kann.

Besondere Probleme gibt es bei Threads gleicher Priorität. Die Java-Spezifikation macht keine Aussage darüber, ob bei Threads gleicher Priorität ein preemptive Scheduling mit Hilfe von Round-Robin[206] erfolgen soll oder nicht.

Daher findet man Implementierungen mit und ohne Round-Robin.

In der Praxis spielt es allerdings oft keine Rolle, ob ein Round-Robin implementiert ist oder nicht. Ist ein Round-Robin nicht implementiert, so kommen bei gleicher Priorität manche Threads immer dann nicht an die Reihe, wenn andere Threads den Prozessor über Gebühr benutzen. In der Praxis werden jedoch Threads häufig für Ein- und Ausgaben eingesetzt. Dies bedeutet, dass sie zwischen den Zuständen "running", "blocked" und "ready-to-run" abwechseln. Das Blockieren bei Ein- und Ausgabeoperationen ermöglicht es dann letztendlich anderen Threads, auch den Prozessor zu erhalten.

Der Programmierer kann selbst darauf achten, dass rechenintensive Threads nicht zu lange den Prozessor benötigen. Hierzu wird **empfohlen**, **bei rechenintensiven Threads gezielt die `yield()`-Methode aufzurufen**.

[206] Round-Robin ist ein Zeitscheibenverfahren, bei dem alle Teilnehmer die gleiche Priorität haben und abwechselnd der Reihe nach drankommen. Das Wort Round-Robin kommt aus dem amerikanischen Englisch und bedeutet einen Wettbewerb, bei dem jeder Teilnehmer gegen jeden anderen Teilnehmer spielt.

Durch den Aufruf von `yield()` geht der aufrufende Thread selbst in den Zustand "ready-to-run" über und erlaubt es, dass ein Thread gleicher oder geringerer Priorität den Prozessor erhält.

Eine andere Möglichkeit ist, rechenintensive Threads mit einer geringeren Priorität als I/O-intensive Threads zu versehen, da Threads jedes Mal in den Zustand "blocked" versetzt werden, wenn Ein- und Ausgaben durchgeführt werden. Die I/O-intensiven Threads bieten damit niederprioren Threads die Möglichkeit zum Laufen.

Prioritäten

Für die Priorität eines Threads gibt es 3 Konstanten in der Klasse `Thread`:

```
MAX_PRIORITY  = 10
NORM_PRIORITY = 5
MIN_PRIORITY  = 1
```

Die Zahlenwerte der Konstanten entsprechen der Gewichtung der Prioritäten. Diese Konstanten müssen aber für die Angabe der Priorität eines Threads nicht verwendet werden – es ist durchaus problemlos jede andere ganze Zahl zwischen eins und zehn möglich. Zu beachten ist, dass bei Applets die höchste Priorität 6 beträgt. Wird für einen Thread keine Priorität gesetzt, so ist sie in der Regel `NORM_PRIORITY`.

Die **Änderung der Priorität eines Threads** und d**ie Abfrage einer gesetzten Priorität erfolgt** mit den Methoden:

- `public final void setPriority (int newPriority)`
  Mit dieser Methode kann die Priorität eines Threads auf den als aktueller Parameter übergebenen Wert gesetzt werden.
- `public final int getPriority()`
  Diese Methode gibt die aktuelle Priorität eines Threads zurück.

16.8 Zugriff auf gemeinsame Ressourcen

Threads innerhalb eines Betriebssystem-Prozesses können wechselseitig auf ihre Variablen zugreifen.

Variablen, die von mehreren Threads gemeinsam als Shared Memory benutzt werden, werden auch als **kritische Variablen** bezeichnet, weil diese Variablen durch die Verwendung in mehreren Threads inkonsistent werden können.

So kann beispielsweise ein Thread, der den Wert einer kritischen Variablen ändert, den Prozessor entzogen bekommen, ehe er mit dem Schreiben der Daten fertig ist. Ein Lese-Thread kann dann schon zu lesen beginnen und bekommt dann inkonsistente Daten (**Reader/Writer-Problem**). Dies ist ein Beispiel für eine sogenannte **Race Condition**.

Bei einer Race Condition hängt das Ergebnis von der Reihenfolge, in der die Threads ausgeführt werden, ab. Um deterministische Ergebnisse zu erzielen, ist daher eine **Synchronisation** im Sinne einer definierten Abarbeitungsreihenfolge der Threads zwingend erforderlich.

Beispiel für eine Race Condition beim Reader/Writer-Problem

Die beiden Threads 1 und 2 greifen auf eine gemeinsam genutzte Variable (ein Array) zu. Das Array ist damit eine **kritische Variable** im System. Das folgende Bild gibt ein Beispiel:

Bild 16-10 Array mit einem Zähler für die benutzten Elemente

Thread 1 schreibt Messdaten in das Array beginnend ab Array-Index 1 und die Anzahl der Werte in die Variable `Zaehler`, die im Array an der Position mit Array-Index 0 steht. **Thread 2** liest die Daten aus und quittiert das Auslesen, indem er die Zählervariable auf 0 setzt. Es kann aber sein, dass schneller geschrieben als gelesen wird. Dabei kann Thread 1 neue Werte an die nächsten Positionen in das Array eintragen und muss dann den Zähler entsprechend erhöhen. Dies ist im folgenden Bild dargestellt:

Bild 16-11 Sequenzdiagramm zur Darstellung einer Race Condition. Die Zeitachse t gibt den zeitlichen Verlauf an.

Es kann nun der Fall eintreten, dass Thread 2 gerade, als er das Array gelesen, aber den Zähler noch nicht auf 0 gesetzt hat, vom Scheduler den Prozessor entzogen bekommt. Thread 1 schreibt nun die neuen Daten hinter die bereits gelesenen Daten und erhöht den Zähler. Wenn nun Thread 2 die Arbeit wieder aufnimmt, setzt er den Zähler auf 0, und damit sind die soeben geschriebenen Daten verloren.

16.8.1 Prinzip des wechselseitigen Ausschlusses

Zur Vermeidung von Race Conditions wendet man das **Prinzip des wechselseitigen Ausschlusses** (engl. **mutual exclusion**) an. Dazu führt man **kritische Abschnitte** ein.

Ein **kritischer Abschnitt** ist eine **Folge von Befehlen, die ein Thread nacheinander vollständig abarbeiten muss**, auch wenn er vorübergehend die CPU an einen anderen Thread abgibt. Kein anderer Thread darf einen kritischen Abschnitt betreten, der auf die gleiche kritische Variable zugreift, solange der erstgenannte Thread mit der Abarbeitung der Befehlsfolge noch nicht fertig ist.

In den nächsten beiden Kapiteln wird das Semaphorkonzept und das Monitorkonzept zuerst in allgemeiner Form vorgestellt, das heißt unabhängig von der Sprache Java. Beide Konzepte ermöglichen einen wechselseitigen Ausschluss. In Kapitel 16.8.4 wird dann das in Java realisierte Monitorkonzept zur Realisierung eines wechselseitigen Ausschlusses analysiert.

16.8.2 Das Semaphorkonzept

Ein wechselseitiger Ausschluss kann mit Semaphoren[207] realisiert werden.

Ein Semaphor hat die folgenden Eigenschaften:

- Ein **Semaphor** wird repräsentiert durch eine **ganzzahlige, nicht negative Variable** verbunden mit einer **Warteschlange für Prozesse**, die einen der kritischen Abschnitte, denen dieselbe Semaphorvariable zugeordnet ist, bearbeiten wollen.
- Auf einem **Semaphor** kann man **nur** mit den **Befehlen** **`wait()`** und **`signal()`**[208]arbeiten.

Alle kritischen Abschnitte, die auf die gleiche kritische Variable zugreifen, verwenden eine gemeinsame **Semaphorvariable**.

Beim **Eintritt** in einen kritischen Abschnitt ruft ein Prozess zuerst den **Befehl** `wait()` für die entsprechende Semaphorvariable auf. Beim Verlassen eines kri-

[207] Java kennt keine Semaphoren, sondern das Monitorkonzept.
[208] Die Befehle `wait()` und `signal()` sind hier Befehle in einem Pseudocode.

tischen Abschnitts ruft ein Prozess den **Befehl** `signal()` für diese Semaphorvariable auf.

Die Funktionsweise der Befehle `wait()` und `signal()`[209] wird im Folgenden beschrieben:

- **Der Befehl `wait()`**

  Der Befehl `wait()` wird beim Eintritt in einen kritischen Abschnitt aufgerufen. Wird z. B. für eine Semaphorvariable mit dem Namen `sem` der Befehl `wait (sem)` aufgerufen, so wird überprüft, ob die Variable `sem` gleich 0 ist. Ist die Variable `sem` gleich 0, so wird der Prozess, der den Befehl `wait (sem)` aufgerufen hat, in die Warteschlange der Semaphorvariablen `sem` gestellt. Ist die Variable `sem` größer als 0, so wird die Semaphorvariable um eins erniedrigt und der `wait()`-Befehl ist erfolgreich beendet. Der Prozess darf dann den folgenden kritischen Abschnitt bearbeiten.

- **Der Befehl `signal()`**

  Der Befehl `signal()` wird beim Verlassen eines kritischen Abschnitts aufgerufen. Wird z. B. für die Semaphorvariable `sem` der Befehl `signal (sem)` aufgerufen, so wird die Variable um eins erhöht. Zusätzlich wird noch die Warteschlange von `sem` überprüft. Warten dort Prozesse, so wird einer dieser Prozesse befreit. Der befreite Prozess darf dann den kritischen Abschnitt bearbeiten.

Stellen Sie sich als Beispiel eine Datenbank vor, die zur Speicherung von Personaldaten dient. Jedes Mal, wenn eine neue Person erfasst wird, ermittelt der zuständige Schreibprozess mit Hilfe der schon bestehenden Personaldaten die nächste freie Personalnummer. Ist diese ermittelt, trägt der Prozess die neue Personalnummer mit den restlichen Personendaten als einen neuen Datensatz in der Datenbank ein. Läuft dieser ganze Vorgang – Ermittlung der Personalnummer und Eintrag des neuen Datensatzes – ohne Unterbrechung des entsprechenden Prozesses ab, so hat man keine Inkonsistenzen zu befürchten. Wird der Prozess allerdings nach der Ermittlung der nächsten freien Personalnummer durch einen anderen Schreibprozess unterbrochen, der auch einen neuen Datensatz einfügen möchte, kann es zum Datenverlust kommen.

Folgendes Szenario zeigt einen solchen Datenverlust:

- Prozess A ermittelt die Nummer 10 als nächste freie Nummer.
- Prozess A wird durch einen Prozess B unterbrochen, der auch einen neuen Datensatz schreiben möchte.
- Prozess B ermittelt ebenfalls die Nummer 10 als nächste freie Nummer.
- Prozess B trägt unter der Nummer 10 seine neuen Daten ein.
- Prozess A wird fortgeführt und schreibt ebenfalls unter der Nummer 10 seine neuen Daten. Somit sind die Daten, die Prozess B geschrieben hat, verloren.

Ein solches Problem kann man umgehen, wenn man den gesamten Schreibvorgang als einen kritischen Abschnitt implementiert. Diesem kritischen Abschnitt wird eine

[209] Die Befehle `wait()` und `signal()` sind selbst unteilbar. Dies wird üblicherweise mittels Hardware realisiert.

Semaphorvariable mit dem Namen `sem` zugeordnet, die den kritischen Abschnitt überwachen soll. Die Semaphorvariable hat den Anfangswert 1. Der kritische Abschnitt der Schreiboperation kann nun folgendermaßen durch die Semaphorvariable `sem` geschützt werden:

```
wait (sem)
// kritischer Abschnitt beginnt
// ermittle nächste freie Personalnummer
// schreibe alle Datenfelder
// kritischer Abschnitt zu Ende
signal (sem)
```

Der erste Prozess, der diesen Codeabschnitt abarbeitet, ruft den Befehl `wait (sem)` auf. Dieser prüft, ob die Semaphorvariable `sem` gleich 0 ist. Da `sem` als Anfangswert den Wert 1 hat, wird die Variable um eins erniedrigt – also auf 0 gesetzt – und der Prozess kann den **kritischen Abschnitt** bearbeiten. Kommt nun ein zweiter Prozess und möchte Daten ebenfalls schreiben, so ruft auch er den `wait (sem)`-Befehl auf, die Semaphorvariable `sem` ist jedoch gleich 0, und deshalb wird dieser Prozess in die Warteschlange eingereiht. Ist der erste Prozess mit der **Abarbeitung des kritischen Abschnitts fertig**, so ruft er den Befehl `signal (sem)` auf. Dieser erhöht `sem` um 1 und befreit den wartenden Prozess aus der Warteschlange.

Dieser setzt nun die Abarbeitung des Befehls `wait (sem)` an derjenigen Stelle fort, wo er zuvor unterbrochen wurde, und erniedrigt die Variable `sem` um 1, womit er den kritischen Abschnitt für sich reserviert. Nach der Abarbeitung des kritischen Abschnitts wird wiederum `signal (sem)` aufgerufen. Dies hat zur Folge, dass `sem` wieder auf den Wert 1 gesetzt wird. Da in diesem Beispiel keine Prozesse in der Warteschlange warten, kann auch kein Prozess aufgeweckt werden.

16.8.3 Das Monitorkonzept

Eine Lösung mit Semaphoren kann für den Programmierer leicht unübersichtlich werden. Von Hoare wurden 1974 **Monitore** als ein Synchronisationsmittel, das auf Semaphoren aufsetzt, diese aber gegenüber dem Programmierer kapselt, vorgeschlagen.

Die Grundidee eines Monitors ist, die **Daten**, auf denen die kritischen Abschnitte arbeiten, und die **kritischen Abschnitte selbst in einem zentralen Konstrukt zusammenzufassen**.

In objektorientierten Programmiersprachen lässt sich ein Monitor als ein Objekt mit speziellen Eigenschaften für die Methoden realisieren.

Das folgende Bild zeigt einen Monitor als Struktur aus Daten und Methoden und der Warteschlange der wartenden Prozesse:

Bild 16-12 Immer nur ein Prozess kann den Monitor benutzen

Die Funktionalität von Monitoren ist äquivalent zu derjenigen von Semaphoren.

Monitore sind jedoch vom Programmierer einfacher zu überschauen als Semaphoren, da gemeinsam benutzte Daten und Zugriffsfunktionen zentral gebündelt an einer einzigen Stelle lokalisiert sind und nicht wie im Falle von Semaphoren getrennt und über mehrere Prozesse verteilt im Programmcode stehen.

Die grundlegenden Eigenschaften eines Monitors sind:

- Kritische Abschnitte, die auf denselben Daten arbeiten, sind Methoden eines Monitors.
- Ein Prozess betritt einen Monitor durch den Aufruf einer **Methode** des Monitors.
- **Nur ein einziger Prozess kann zur selben Zeit den Monitor benutzen**. Jeder andere Prozess, der den Monitor aufruft, wird suspendiert und muss warten, bis der Monitor verfügbar wird.

Einfache kritische Abschnitte reichen aus für einen wechselseitigen Ausschluss. Oftmals jedoch ist das Betreten eines kritischen Abschnitts abhängig vom Vorliegen einer bestimmten Bedingung. So kann etwa ein Erzeuger-Prozess nur dann in einen Puffer, der eine globale Variable darstellt, schreiben, wenn der Puffer nicht voll ist. Ist der Puffer voll, so muss der Erzeuger-Prozess mit dem Schreiben warten.

Um einen Monitor praktikabel zu machen, muss also ein Monitor die Möglichkeit **bedingter kritischer Abschnitte** (engl. conditional critical sections) bieten, damit ein Prozess **beim Betreten eines kritischen Abschnitts** prüfen kann, ob er diesen kritischen Abschnitt ausführen soll oder nicht. Aufgrund einer vorliegenden Bedingung kann ein Prozess freiwillig die Abarbeitung einer Methode unterbrechen, z. B. wenn die Bedingung "Daten vorhanden" nicht erfüllt ist. Ein Prozess kann seine Arbeit unterbrechen, indem er einen `wait()`-Befehl an den Monitor gibt. Damit wird dieser Prozess blockiert und der Monitor für einen anderen Prozess freigegeben. Ein anderer Prozess kann dann den Monitor betreten – z. B. ein Schreibprozess –, die Bedingung ändern und vor dem Verlassen des kritischen Abschnitts ein Signal mit dem `signal()`-Befehl an die Warteschlange der Prozesse senden, die auf die Erfüllung der Bedingung "Daten vorhanden" warten. Durch das Senden eines `sig-`

`nal()`-Befehls wird ein Prozess aus der Warteschlange aufgeweckt und kann die Bearbeitung fortsetzen.

16.8.4 Mutual exclusion in Java mit dem Monitorkonzept

Ein wechselseitiger Ausschluss wird in Java mit dem Monitorkonzept und nicht mit Semaphoren realisiert.

In Java wird das Monitorkonzept mit Hilfe des Schlüsselwortes `synchronized` umgesetzt. Das Schlüsselwort **`synchronized`** kann als Schlüsselwort für Methoden verwendet werden oder einen zu synchronisierenden Block kennzeichnen.

Im Folgenden werden die Möglichkeiten zur Realisierung eines Monitors vorgestellt.

16.8.4.1 Monitor für den gegenseitigen Ausschluss von synchronisierten Klassenmethoden einer Klasse.

Werden eine oder mehrere **Klassenmethoden** mit dem Schlüsselwort `synchronized` versehen, so wird ein Monitor um diese Methoden herumgebaut. Dadurch kann nur ein einziger Thread zu einer bestimmten Zeit eine der synchronisierten Methoden bearbeiten.

Der folgende Codeausschnitt zeigt zwei synchronisierte Klassenmethoden:

```
public class Syn1
{
   . . . . .
   public static synchronized void methode1()
   {
      // kritischer Abschnitt
   }
   public static synchronized void methode2()
   {
      // kritischer Abschnitt
   }
}
```

Es wird für **alle synchronisierten Klassenmethoden einer Klasse ein Monitor** angelegt, der den Zugriff auf alle synchronisierten Klassenmethoden dieser Klasse überwacht.

Das folgende Bild zeigt den Sachverhalt:

Bild 16-13 Gemeinsamer Monitor für synchronisierte Klassenmethoden einer Klasse `Syn1`

Zu beachten ist, dass, wenn andere nicht synchronisierte Klassenmethoden zu der Klasse `Syn1` noch vorhanden sind, diese dann nicht durch den Monitor geschützt sind.

16.8.4.2 Monitor für den gegenseitigen Ausschluss von synchronisierten Instanzmethoden zu einem speziellen Objekt.

Werden mehrere **Instanzmethoden** einer Klasse mit dem Schlüsselwort `synchronized` versehen, so hat **jedes Objekt dieser Klasse** einen **eigenen Monitor**, der den Zugriff auf diese Instanzmethoden überwacht.

Der folgende Codeausschnitt zeigt zwei synchronisierte Instanzmethoden:

```
public class Syn2
{
   . . . . .
   public synchronized void methode1()
   {
      // kritischer Abschnitt
   }

   public synchronized void methode2()
   {
      // kritischer Abschnitt
   }
}
```

Im folgenden Bild sind zwei Instanzen der Klasse `Syn2` zu sehen. Jede dieser Instanzen hat ihren eigenen Monitor für alle synchronisierten Instanzmethoden:

Bild 16-14 Bei synchronisierten Instanzmethoden existiert jeweils ein Monitor pro Objekt

Genauso wie bei synchronisierten Klassenmethoden gilt bei synchronisierten Instanzmethoden, dass wenn nicht synchronisierte Instanzmethoden existieren, diese nicht durch den Monitor geschützt sind.

16.8.4.3 Monitor für den gegenseitigen Ausschluss von einzelnen synchronisierten Codeblöcken.

Es ist möglich, Blöcke in unterschiedlichen Methoden und sogar in unterschiedlichen Klassen gemeinsam zu synchronisieren. Ein Monitor für einen Codeblock wird in Java durch Aufruf der **`synchronized`-Anweisung** mit einer Referenz auf ein Objekt, das einen Schlüssel darstellt, als Übergabeparameter realisiert. Der folgende Codeausschnitt zeigt einen synchronisierten Codeblock:

```
public class Syn2
{
   public void methode1()
   {
      Object schluessel = Schluessel.getSchluessel();

      synchronized (schluessel)
      {
         // kritischer Abschnitt
      }
   }
}
```

Beispiel für eine `synchronized`-Anweisung

Hierbei ist `schluessel` eine Referenzvariable auf ein Objekt, das beispielsweise mit Hilfe der Klassenmethode `getSchluessel()` der Klasse `Schluessel` zur Verfügung gestellt wird.

Die `synchronized`-Anweisung, um einen Block zu synchronisieren, ist grundverschieden von der Verwendung des Schlüsselwortes `synchronized`, um Instanz- und Klassenmethoden zu synchronisieren. Dies liegt aber in erster Linie daran, dass dem Programmierer bei der Synchronisation von Methoden einige Details vorenthalten bleiben. Bevor jedoch darauf eingegangen wird, soll zuerst eine Erklärung für die Synchronisation eines Blockes geliefert werden.

Eine **`synchronized`-Anweisung** wird mit dem Schlüsselwort `synchronized` eingeleitet. In den runden Klammern erwartet die `synchronized`-Anweisung eine Referenz auf ein Objekt, das als Schlüssel dient. Die nachfolgenden geschweiften Klammern schließen die Anweisungen eines kritischen Abschnitts ein.

Um einen Block zu synchronisieren, benötigt man also einen Schlüssel (auch Lock genannt). **Als Schlüssel wird in Java ein Objekt verwendet**. Dieser Schlüssel kann zu einer bestimmten Zeit nur von einer einzigen `synchronized`-Anweisung verwendet werden. Stellen Sie sich hierzu mehrere synchronisierte Blöcke vor, die alle den gleichen Schlüssel benutzen – also alle das gleiche Schlüsselobjekt –, um den Zutritt zu den kritischen Abschnitten zu erlangen. Der Thread, der als erstes den Schlüssel vom Schlüsselbrett abholt, kann den kritischen Abschnitt somit betreten. Alle anderen Threads, die den gleichen Schlüssel benutzen, stellen fest, dass der Schlüssel gerade nicht am Schlüsselbrett hängt, und deshalb können sie einen kritischen Abschnitt, der den gleichen Schlüssel benötigt, nicht betreten.

Die Aufgabe des Schlüsselbrettes wird nun von einem Monitor wahrgenommen. Der Monitor gibt dem ersten Thread, der den Monitor betreten möchte, den Schlüssel und nimmt dem Thread beim Verlassen des Monitors den Schlüssel wieder ab. Damit kann der Monitor dem nächsten wartenden Thread, der den Schlüssel und damit den Zugang zu einem kritischen Abschnitt möchte, den Schlüssel aushändigen.

Es kann jedes beliebige Objekt als Schlüsselobjekt verwendet werden. Es ist nur auf eines zu achten:

Alle Blöcke, die das gleiche Schlüsselobjekt verwenden, haben einen gemeinsamen Monitor, der den Zutritt zu allen kritischen Abschnitten überwacht.

Innerhalb eines Programms können auch synchronisierte Blöcke existieren, die unterschiedliche Schlüsselobjekte verwenden. Dann existieren insgesamt so viele Monitore, wie unterschiedliche Schlüsselobjekte verwendet werden.

Alle synchronisierten Abschnitte, die das **gleiche Schlüsselobjekt** als Schlüssel verwenden, **schließen sich gegenseitig** aus, da sie einen gemeinsamen Monitor verwenden.

Werden zwei Codeblöcke einer Klasse mit **unterschiedlichen Schlüsselobjekten** synchronisiert, so **schließen sich** diese Codeblöcke **gegenseitig nicht** aus, da jeder Codeblock seinen eigenen Monitor hat.

Wird eine **`synchronized`-Anweisung** betreten, so wird nachgeschaut, **ob das Schlüsselobjekt**, auf das die übergebene Referenz zeigt, **schon bereits von einem anderen Thread als Schlüssel benutzt wird**. Ist dies der Fall, so muss der gerade angefragte Thread warten, bis das Schlüsselobjekt freigegeben wird.

Zusammenfassend kann gesagt werden: Beim **Eintritt** in eine mit `synchronized` gekennzeichnete Stelle **wird geprüft, ob das verwendete Schlüsselobjekt gerade von einem anderen Thread als Schlüssel benutzt wird**. Ist dies nicht der Fall, so kann ein Thread nach Erhalt des Schlüsselobjekts den kritischen Abschnitt betreten. Solange dieses dann von diesem Thread benutzt wird – das heißt, so lange sich dieser Thread im kritischen Abschnitt befindet – kann kein anderer Thread einen kritischen Abschnitt betreten, der das gleiche Schlüsselobjekt verwendet. Verlässt der Thread den synchronisierten Block, kann das Schlüsselobjekt von dem nächsten Thread benutzt werden, um den Monitor zu betreten.

16.8.5 Der versteckte Schlüssel für synchronisierte Methoden

Auch Methoden, die mit dem Schlüsselwort `synchronized` versehen sind, verwenden einen Schlüssel, mit dessen Hilfe ein Monitor einen wechselseitigen Ausschluss realisiert.

Im Folgenden soll auf die **Synchronisation** von **Instanzmethoden** und **Klassenmethoden** eingegangen werden:

- **Instanzmethoden**

  Für synchronisierte Instanzmethoden eines Objekts wird ein eigener Monitor für jedes Objekt angelegt, d. h. pro Objekt wird ein Monitor angelegt. Über die Verwendung der `this`-Referenz wird als Schlüsselobjekt das Objekt verwendet, zu dem die Instanzmethode aufgerufen wurde. Der gesamte Rumpf einer synchronisierten Instanzmethode wird in eine **`synchronized`-Anweisung** umgesetzt. Als Parameter wird der `synchronized`-Anweisung die `this`-Referenz übergeben. Die synchronisierte Instanzmethode

```
public synchronized void methode()
{
   . . . . .
}
```

  wird damit umgesetzt in:

```
public void methode()
{
   synchronized (this)
   {
      . . . . .
   }
}
```

Synchronisierte Instanzmethoden verwenden als Schlüsselobjekt das Objekt, zu dem die Instanzmethode aufgerufen wurde.

- **Klassenmethoden**

  Wie in Kapitel 17.5 gezeigt wird, wird zu jeder Klasse, die in die Java-Virtuelle Maschine geladen wird, ein spezielles Objekt der Klasse `Class<T>` angelegt. Für jede Klasse existiert also ein spezielles Objekt der Klasse `Class<T>`.

Werden Klassenmethoden synchronisiert, so wird als Schlüsselobjekt das Objekt der speziellen Klasse `Class<T>` verwendet.

Zusammenfassung

Bei synchronisierten

- **Instanzmethoden** wird als Schlüsselobjekt **das eigene Objekt** verwendet.
- **Klassenmethoden** wird als Schlüsselobjekt das entsprechende **Objekt der Klasse** `Class<T>` verwendet.
- **Blöcken** wird als Schlüsselobjekt das **Objekt** verwendet, auf das die **übergebene Referenz** zeigt.

16.8.6 Synchronisation von Instanzmethoden und von Blöcken

Im Folgenden soll die Synchronisation von Blöcken und Instanzmethoden kurz zusammengefasst werden.

Am einfachsten ist eine Synchronisation der kritischen Abschnitte eines Objekts zu erreichen, wenn man die **kritischen Methoden** eines von mehreren Threads besuchten **Objektes** mit `synchronized` markiert. Dazu hier ein Ausschnitt aus einem einfachen **Beispiel**:

```
public class Stack
{
   private int[] array;
   private int index = 0;
   public synchronized void push (int wert)
   {
      . . . . .
   }

   public synchronized void pop()
   {
      . . . . .
   }
}
```

Der **Monitor** legt sich um die beiden Methoden, die mit `synchronized` gekennzeichnet sind, wie in folgendem Bild zu sehen ist:

Bild 16-15 Synchronisierte Methoden werden von einem Monitor geschützt

Da immer nur ein einziger Thread den Monitor betreten darf, ist sichergestellt, dass ein kritischer Abschnitt vollständig abgearbeitet wird, bevor der nächste kritische Abschnitt betreten wird. Wird also die Methode `push()` von einem Thread aufgerufen, so sind die Methoden `push()` und `pop()` für den Zugriff eines anderen Threads gesperrt. Hätte die Klasse `Stack` jedoch noch andere Methoden, die nicht mit `synchronized` gekennzeichnet sind, so könnten diese Methoden durchaus zeitgleich von anderen Threads ausgeführt werden, auch wenn sich bereits ein Thread in einer der synchronisierten Methoden befindet.

Ein **Monitor** legt sich **nur um die synchronisierten Methoden** eines Objektes. Methoden, die nicht `synchronized` sind, werden von dem Monitor nicht geschützt.

Bei der **Synchronisation von Blöcken** als feinere Einheiten wird dem Block eine Referenz auf ein Schlüsselobjekt übergeben. Der `synchronized`-Block mit formalem Parameter hat damit die folgende Struktur:

```
synchronized (Object ref)
{
   // kritische Operationen
}
```

Die Synchronisation mit Blöcken bietet zwei Vorteile:

- eine feinere Granularität
- und eine klassenübergreifende Synchronisation von kritischen Abschnitten.

Durch einen Block erhält man eine feinere Granularität beim Synchronisieren. Da hierdurch nicht eine ganze Methode für den Zugriff durch mehrere Threads gesperrt ist, kann ein anderer Thread rascher den Monitor betreten.

Das Schlüsselobjekt, auf das eine Referenz übergeben wird, kann in Codeblöcken verschiedener Klassen benutzt werden. Somit kann eine **klassenübergreifende Synchronisation von kritischen Abschnitten** erfolgen.

16.8.7 Die Methoden `wait()`, `notify()` und `notifyAll()`

Im folgenden Beispiel soll eine **Pipe** zum Austausch von Zeichen zwischen Threads entwickelt werden. Pipes sind Puffer im Arbeitsspeicher, die nach dem "First In – First Out"-Prinzip (FIFO-Prinzip) funktionieren.

Das heißt, es kann nur in der Reihenfolge aus der Pipe gelesen werden, in der auch hineingeschrieben wurde.

Zum Lesen wird eine Methode `read()`, zum Schreiben eine Methode `write()` implementiert. Wenn die Pipe voll ist, dann soll der schreibende Thread solange angehalten werden, bis wieder Platz in der Pipe vorhanden ist. Umgekehrt soll ein Thread, der aus einer leeren Pipe zu lesen versucht, solange in den Wartezustand versetzt werden, bis wieder Daten vorhanden sind. Um diese Steuerung zu ermöglichen, werden die von der Klasse `Object` geerbten Methoden **`wait()`** und **`notify()`** verwendet. Bevor auf die Wirkungsweise der Methoden `wait()` und `notify()` eingegangen wird, soll zuerst Schritt für Schritt der Quellcode für das beschriebene Beispiel entwickelt werden. Der folgende Programmcode kann noch nicht kompiliert werden:

```
public class Pipe
{
   private int[] array = new int [10];
   private int index = 0;

   public synchronized void write (int wert)
   {
      if (index == array.length) // Array ist voll, es muss zuerst
         wait();                 // wieder ein Element gelesen
                                 // werden
      // Schreiboperation durchführen
      // Index erhöhen
      if (index == 1)  // Einen eventuell wartenden Leser aufwecken,
         notify();     // da ein Element gelesen werden kann.
   }

   public synchronized int read()
   {
      if (index == 0)  // Wenn es keine Elemente zu lesen gibt
         wait();
      // Leseoperation durchführen
      // Index erniedrigen
      // Array-Elemente um eine Position nach vorne schieben.
      if (index == array.length - 1) // Einen eventuell wartenden
         notify();                   // Schreiber aufwecken
      // Gelesenen Wert zurückgeben
   }
}
```

Innerhalb der Methode `write()` wird geprüft, ob noch ein Datum in das Array geschrieben werden kann. Ist das Array schon voll, so versetzt sich der entsprechende Thread durch den Aufruf der Methode `wait()` in den Wartezustand. Dieser Thread verharrt solange im Wartezustand, bis er durch den Methodenaufruf `notify()` in der Lesemethode wieder aufgeweckt wird.

Neben der Methode `notify()` stellt die Klasse Object auch die Methode `notifyAll()` zur Verfügung. Die Methoden `notify()` und `notifyAll()` unterscheiden sich folgendermaßen: Sind mehrere Threads im Wartezustand, so wird durch die Methode `notify()` nur ein einziger Thread wieder aufgeweckt. Nach welcher Reihenfolge dies erfolgt, ist nicht spezifiziert. Möchte man alle Threads, die im Wartezustand sind, aufwecken, kann man die Methode `notifyAll()` benutzen. Welcher Thread dann allerdings zum Zuge kommt, entscheidet der Zufall. Alle anderen Threads werden allerdings auch aufgeweckt, um gleich wieder festzustellen, dass die Bedingung immer noch nicht zutrifft. Diese Vorgehensweise mag vielleicht ineffizient erscheinen, stellt aber auf der anderen Seite sicher, dass ein Thread nicht für immer im Wartezustand verharren muss.

Die Methode `read()` verhält sich ähnlich wie die Methode `write()`:

Es wird geprüft, ob das Array leer ist. Ist das Array leer, so versetzt sich der Thread durch den Aufruf der Methode `wait()` in den Wartezustand. Dieser Thread verharrt solange im Wartezustand, bis er durch den Methodenaufruf `notify()` in der Schreibmethode wieder aufgeweckt wird.

Durch den Aufruf von `wait()` versetzt sich ein Thread, der eine synchronisierte Methode abarbeitet, selbst in den Zustand "blocked". Dadurch wird der Monitor für einen anderen Thread freigegeben.

Durch `notify()` wird ein Thread, der sich zuvor durch den Aufruf von `wait()` in den Zustand "blocked" brachte, wieder aufgeweckt. Der Thread wird "ready-to-run" und hat nun die Chance auf die Zuteilung des Monitors.

Die Methoden `wait()` und `notify()` sind eng mit den Befehlen `wait()` und `signal()` verwandt, die auf eine Semaphorvariable angewandt werden:

- Genauso wie der `wait()`-Befehl den aktuellen Prozess in eine Warteschlange einreiht, die der Semaphorvariablen zugehörig ist – reiht die Methode `wait()` den aktuellen Thread in eine dem Schlüsselobjekt zugehörige Warteschlange ein.
- Genauso wie der `signal()`-Befehl einen Prozess aus der Warteschlange befreit, die der Semaphorvariablen zugeordnet ist, befreit die `notify()`-Methode einen Thread aus der Warteschlange, die dem Schlüsselobjekt zugeordnet ist.

Ein Thread, der durch die `notify()`-Methode aufgeweckt wird, konkurriert genauso wie alle anderen Threads, die gerade in den Monitor eintreten wollen, um den Zugriff.

Beachten Sie, dass die Instanzmethoden `wait()` und `notify()` der Klasse `Object` nur für ein Schlüsselobjekt aufgerufen werden dürfen. Wird eine dieser Methoden zu einem Objekt aufgerufen, das gerade nicht als Schlüsselobjekt für einen synchronisierten Abschnitt benutzt wird, so wird eine Exception vom Typ **`IllegalMonitorStateException`** geworfen.

Wird die Blocksynchronisation verwendet, so muss der Aufruf der Methoden `wait()` und `notify()` explizit für ein Schlüsselobjekt erfolgen. Das vorhergehende Beispiel muss dann folgendermaßen aussehen:

```
public class Pipe
{
   private int[] array = new int [10];
   private int index = 0;
   private Object key = new Object();

   public void write (int wert)
   {
      synchronized (key)
      {
         if (index == array.length) // Array ist voll, es muss
            key.wait();             // zuerst ein Element gelesen
                                    // werden
         // Schreiboperation durchführen
         // Index erhöhen
         if (index == 1)            // Einen eventuell wartenden
            key.notify();           // Leser aufwecken
      }
   }

   public int read()
   {
      synchronized (key)
      {
         if (index == 0)  // Wenn es keine Elemente zu lesen gibt
            key.wait();
         // Leseoperation durchführen
         // Index erniedrigen
         // Array-Elemente um eine Position nach vorne schieben.
         if (index == array.length - 1) // Einen eventuell wartenden
            key.notify();               // Schreiber aufwecken
      }
   }
}
```

Lauffähiges Beispiel

Das folgende Beispiel zeigt die ausprogrammierte Pipe und zwei Threads, die diese benutzen. Dabei schreibt der Thread `Writer` die Werte in die Pipe hinein und der Thread `Reader` liest die Werte aus der Pipe heraus. Jetzt das Beispiel:

```
// Datei: Pipe.java

public class Pipe
{
   private int[] array = new int [3];
   private int index = 0;

   public synchronized void write (int i)
   {
      if (index == array.length) // Falls Array Grenze erreicht,
      {                          // Thread anhalten
         System.out.println ("Schreibender Thread muss warten");

         try
         {
            this.wait();
         }
         catch (InterruptedException e)
         {
         }
      }

      array [index] = i; // Wert in Array speichern
      index++;

      if (index == 1)    // Einen eventuell wartenden
         this.notify();  // Leser aufwecken

      System.out.println ("Geschrieben: " + i);
   }

   public synchronized int read()
   {
      int value;

      if (index == 0) // Falls kein Wert vorhanden,
      {               // Thread anhalten
         System.out.println ("Lesender Thread muss warten");

         try
         {
            this.wait();
         }
         catch (InterruptedException e)
         {
         }
      }

      value = array [0]; // Wert auslesen
      index--;

      for (int i = 0; i < index; i++)
         array [i] = array [i + 1];

      if (index == array.length - 1) // Einen eventuell wartenden
         this.notify();              // Schreiber aufwecken
```

```
      System.out.println ("Empfangen: " + value);
      return value;
   }
}

// Datei: Writer.java
// Thread, der int-Werte in eine Pipe schreibt
public class Writer extends Thread
{
   private Pipe pipe;
   // Eine gesendete 0 soll das Ende kennzeichnen.
   private int[] sendeArray = {1, 2, 3, 4, 0};

   public Writer (Pipe p)
   {
      pipe = p;
   }

   public void run()
   {
      for (int i = 0; i < sendeArray.length; i++)
         pipe.write (sendeArray [i]);
   }
}

// Datei: Reader.java
// Thread, der int-Werte aus einer Pipe liest
public class Reader extends Thread
{
   private Pipe pipe;

   public Reader (Pipe p)
   {
      pipe = p;
   }

   public void run()
   {
      int empfang;

      while ((empfang = pipe.read()) != 0) // Eine gesendete 0 kenn-
         ;                                 // zeichnet das Ende
   }
}

// Datei: Test1.java
public class Test1
{
   public static void main (String args[])
   {
      Pipe pipe = new Pipe();

      Reader readerThread = new Reader (pipe);
      Writer writerThread = new Writer (pipe);

      readerThread.start();
      writerThread.start();
```

```
    }
}
```

Die Ausgabe des Programms ist:

```
Lesender Thread muss warten
Geschrieben: 1
Geschrieben: 2
Geschrieben: 3
Schreibender Thread muss warten
Empfangen: 1
Empfangen: 2
Empfangen: 3
Lesender Thread muss warten
Geschrieben: 4
Empfangen: 4
Geschrieben: 0
Empfangen: 0
```

Die Methode `wait()`, die einen Thread in den Zustand "blocked" überführen kann, kann eine Exception vom Typ `InterruptedException` werfen, die entsprechend abgefangen werden muss. Diese Exception wird allerdings nur geworfen, wenn ein Thread, der sich aufgrund der Methode `wait()` im Zustand "blocked" befindet, durch den Aufruf der Methode `interrupt()` unterbrochen wird. Da dies hier nicht der Fall ist, braucht die Exception nicht weiter behandelt zu werden.

16.8.8 Gefahr durch Deadlocks

Das Prinzip des wechselseitigen Ausschlusses (realisiert durch die Einführung von Monitoren) löst zwar einerseits das Problem der Race Conditions, eröffnet jedoch andererseits die Gefahr von möglichen Deadlocks. Zur Visualisierung ein Bild:

Bild 16-16 Deadlocksituation im Straßenverkehr

Das Problem eines Deadlocks gibt es nicht nur in der Softwaretechnik. Diese Problemstellung kann ebenso in anderen Bereichen auftreten. Das oben stehende Beispiel aus dem Straßenverkehr soll dies verdeutlichen. Das Bild zeigt eine blockierte Straßenkreuzung. Die Fahrzeuge aus beiden Fahrtrichtungen blockieren sich gegenseitig. Die Situation lässt sich nicht mehr auflösen, ohne dass ein Fahrzeug zurücksetzt und einen Teil der Straße freigibt.

Deadlocks treten meistens an unverhofften Stellen auf! Man muss sehr wachsam sein, um überhaupt alle möglichen Deadlocksituationen ausfindig zu machen. Ver-

sucht man, sauber mit Threads zu programmieren, das heißt, achtet man darauf, dass ein Thread auf jeden Fall ordentlich beendet wird und seine allokierten Ressourcen wieder freigibt, stolpert man automatisch über jede Menge Deadlock-Situationen!

Im folgenden Beispiel wird das bereits bekannte Beispiel mit den Klassen `Pipe`, `Reader` und `Writer` erweitert, sodass der Lese- und der Schreib-Thread jeweils Endlos-Schleifen sind. Die umgeschriebenen Klassen `Reader1` und `Writer1` bekommen eine zusätzliche Methode `beenden()`, die das Beenden des jeweiligen Threads ermöglicht. Durch die Eingabe der Buchstaben `'e'`, `'x'`, `'i'` und `'t'` gefolgt von einem `<RETURN>` kann ein Benutzer die beiden Threads beenden. Um sicher zu gehen, dass die Threads auch wirklich beendet sind, wird die `join()`-Methode verwendet, um auf das Ende der Threads zu warten. Die Klasse `Pipe` hat sich nicht verändert und wird deshalb nicht nochmals aufgeführt. Hier das **Beispiel**:

```
// Datei: Reader1.java
public class Reader1 extends Thread
{
   private Pipe pipe;
   private boolean running = true;

   public Reader1 (Pipe p)
   {
      pipe = p;
   }

   public void run()
   {
      int empfang;
      while (running)
         empfang = pipe.read();
   }

   public void beenden()
   {
      running = false;
   }
}

// Datei: Writer1.java
public class Writer1 extends Thread
{
   private Pipe pipe;
   private boolean running = true;

   public Writer1 (Pipe p)
   {
      pipe = p;
   }

   public void run()
   {
      int i = 0;
      while (running)
```

```
         pipe.write (i = ++i % 1000);
   }

   public void beenden()
   {
      running = false;
   }
}

// Datei: Test2.java
import java.io.*;

public class Test2
{
   public static void main (String args[]) throws IOException,
                                           InterruptedException
   {
      Pipe pipe = new Pipe();

      Writer1 writerThread = new Writer1 (pipe);
      Reader1 readerThread = new Reader1 (pipe);

      readerThread.start();
      writerThread.start();

      // Warten, bis "exit" gefolgt von RETURN eingegeben wird.
      while (true)
      {
         BufferedReader reader =
             new BufferedReader (new InputStreamReader (System.in));
         String kommando = reader.readLine();
         if (kommando.equals ("exit"))
            break;
      }

      writerThread.beenden();
      writerThread.join();
      System.out.println ("Schreibender Thread ist beendet!");
      readerThread.beenden();
      readerThread.join();
      System.out.println ("Lesender Thread ist beendet!");
   }
}
```

Hier eine mögliche Ausgabe des Programms:

```
. . . . .
Empfangen: 278
Geschrieben: 279
Empfangen: 279
Schreibender Thread ist beendet!
Lesender Thread muss warten
```

Nanu, wo ist denn die Ausgabe: `Lesender Thread ist beendet!`? Diese Ausgabe wird nie erscheinen und das Programm wird auch nie beendet werden! Es liegt

hier nämlich ein **Deadlock**[210] vor. Der Grund dafür lautet wie folgt: Der schreibende Thread wird beendet. Danach versucht der lesende Thread nochmals, ein Datum zu lesen. Da aber nichts mehr in der Pipe ist, wird der lesende Thread durch den Aufruf der Methode `wait()` in den Wartezustand versetzt. Da es keinen schreibenden Thread mehr gibt, der den lesenden Thread durch den Aufruf der Methode `notify()` aufwecken könnte, wird dieser Wartezustand nie mehr aufgehoben. Die Methode `join()`, die zum lesenden Thread aufgerufen wurde, kehrt nie zurück, da der lesende Thread auch nie beendet wird. Somit verharrt der Thread, der von der Java-Virtuellen Maschine zum Abarbeiten der `main()`-Methode gestartet wurde und der lesende Thread für immer im Zustand "blocked".

Was kann nun getan werden, um diesen Deadlock zu vermeiden? Hierzu soll eine einfache Möglichkeit gezeigt werden! Wenn der schreibende Thread beendet wird, schickt er noch die Zahl `-1`. Diese interpretiert der lesende Thread als das Endezeichen. Aus der Klasse `Test2` muss nur der Aufruf `readerThread.beenden()` entfernt werden – sie ist deshalb nicht nochmals aufgeführt. Hier die geänderten Klassen `Reader2` und `Writer2`:

```
// Datei: Reader2.java
public class Reader2 extends Thread
{
   private Pipe pipe;
   public Reader2 (Pipe p)
   {
      pipe = p;
   }

   public void run()
   {
      while (pipe.read() != -1)
         ;
   }
}

// Datei: Writer2.java
public class Writer2 extends Thread
{
   private Pipe pipe;
   private boolean running = true;

   public Writer2 (Pipe p)
   {
      pipe = p;
   }

   public void run()
   {
      int i = 0;
      while (running)
         pipe.write (i = ++i % 1000);
      pipe.write (-1);
   }
```

[210] Eventuell muss man das Programm mehrmals laufen lassen, um den Deadlock zu erzeugen.

```
   public void beenden()
   {
      running = false;
   }
}
```

Hier eine mögliche Ausgabe des Programms:

```
. . . . .
Empfangen: 80
Geschrieben: 81
Empfangen: 81
Geschrieben: -1
Schreibender Thread ist beendet!
Empfangen: -1
Lesender Thread ist beendet!
```

16.8.9 Sinnvoller Einsatz der Synchronisation

Kandidaten für eine Synchronisation sind grundsätzlich Methoden oder Codeblöcke, die auf gemeinsamen Daten arbeiten und von unterschiedlichen Threads besucht werden.

Alle Methoden oder Codeblöcke, die mit den gleichen Instanz- oder Klassenvariablen arbeiten, können bei gleichzeitigem Aufruf durch mehrere Threads Probleme verursachen. Deshalb ist hier eine Synchronisation notwendig.

Lokale Variablen sind, da jeder Thread einen eigenen Stack besitzt, nicht gefährdet.

Man sollte sich für jeden einzelnen Fall überlegen, ob eine Synchronisation gerechtfertigt ist, denn die Synchronisation von Codeblöcken benötigt Rechenzeit. Werden wahllos alle Methoden vorsichtshalber synchronisiert, kann es zu Performanceproblemen und eventuell zu Deadlocksituationen kommen.

16.8.10 Änderungen seit Java 5 bei der Synchronisation

Seit Java 5 gibt es noch **feinere Möglichkeiten der Synchronisation**. Die bereits vorhandenen und hier dargestellten Mechanismen bilden aber weiterhin die Grundfunktionalität. Außerdem werden nun auch Klassen zur Verfügung gestellt, die man in der Vergangenheit selbst schreiben musste, wie z. B. Queues für den Austausch von Produkten bei einer Producer-Consumer-Anwendung oder Klassen für ein Thread-Pooling.

Beispiele für die Erweiterungen seit Java 5 sind:

- explizite Sperren,
- Semaphoren,

- ein Framework zur Erzeugung und Kontrolle von Threads,
- die zeitgesteuerte Ausführung von Threads,
- Thread-Pooling,
- Warteschlangen und andere performante, nebenläufig benutzbare Collections,
- Klassen für atomare Operationen mit einfachen Datentypen,
- Klassen zur einfachen Synchronisierung mehrerer Threads.

Für das Studium dieser Erweiterungen wird die API-Dokumentation oder das Buch von Johannes Nowak [Now04] empfohlen.

16.9 Dämon-Threads

In Java gibt es neben den seither behandelten normalen Threads noch sogenannte Dämon-Threads. Der einzige Unterschied zwischen beiden Threads ist das Verhalten der Virtuellen Maschine. Die Virtuelle Maschine wird beendet, sobald der letzte normale Thread beendet wird und es keinen normalen Thread mehr gibt. Auf die Dämon-Threads wird dabei keine Rücksicht genommen.

Der Java-Interpreter wird erst beendet, wenn keine Threads mehr abgearbeitet werden. Dies gilt nicht für Dämon-Threads. Sind nur noch Dämon-Threads in einer Java-Virtuellen Maschine vorhanden, gibt es für diese Dämon-Threads, die Dienstleistungen für andere Threads erbringen sollen, nichts mehr zu tun. Die Java-Virtuelle Maschine wird trotz aktiver Dämon-Threads beendet.

Jedes Java-Programm hat Dämon-Threads, die automatisch von der Laufzeitumgebung gestartet werden. Das bekannteste Beispiel ist der Garbage Collector, der im Hintergrund läuft. Er bleibt solange aktiv, bis der letzte nicht Dämon-Thread beendet wurde.

Dämon-Threads sind Threads, die für eine Applikation oder für andere Threads Dienstleistungen erbringen. Sie haben dabei oft nicht – wie allgemein bei Threads üblich – eine Abbruchbedingung, sondern laufen meist in einer Endlosschleife ab.

Dämon-Threads können auch selbst gestartet werden.

Jeder Java-Thread kann zum Dämon-Thread werden, indem seine Instanzmethode **`setDaemon (true)`** der Klasse `Thread` aufgerufen wird. Ein neu erzeugter Thread ist nur dann ein Dämon-Thread, wenn der erzeugende Thread ein Dämon-Thread ist. Ob ein Thread ein Dämon-Thread ist, kann man mit der Instanzmethode **`isDaemon()`** der Klasse `Thread` überprüfen.

Eine Änderung eines Threads in einen Dämon-Thread ist nur im Zustand "new" zulässig, d. h. bevor die Methode `start()` der Klasse `Thread` aufgerufen wurde. Wird die Methode `setDaemon (true)` aufgerufen, während der Thread sich in einem anderen Zustand befindet, wird eine Exception vom Type `IllegalThreadStateException` geworfen.

16.10 Zusammenfassung

Kapitel **16.1** führt das Konzept eines Betriebssystemprozesses ein.

Ein Betriebssystem-Prozess virtualisiert den physischen Prozessor und multiplexed ihn. Ein Betriebssystem-Prozess ist hierbei definiert als "ein Programm in Ausführung" oder als "ein Programm, das laufen möchte". Ein Betriebssystem-Prozess darf gar nicht merken, dass er unterbrochen worden ist. Wird ihm der Prozessor wieder zugeteilt, so muss der Betriebssystem-Prozess in genau derselben Weise weiterarbeiten, als hätte er die ganze Zeit den Prozessor besessen.

Betriebssystem-Prozesse können

- parallel von mehreren Prozessoren
- oder abwechselnd von einem einzigen Prozessor (quasi-parallel)

ausgeführt werden.

Letztendlich ermöglicht ein Betriebssystem, welches ein Betriebssystem-Prozesskonzept unterstützt und Betriebssystem-Prozesse als sogenannte virtuelle Betriebsmittel zur Verfügung stellt, ein Multiplexen des Prozessors.

Kapitel **16.2** befasst sich mit Betriebsmitteln.

All das, was ein Betriebssystem-Prozess zum Laufen braucht, wird als Betriebsmittel bezeichnet. Betriebsmittel können z. B. der Prozessor oder ein I/O-Kanal eines Programms zu einer Datei auf der Festplatte sein. Betriebsmittel können exklusiv benutzbar sein, aber dennoch zeitlich aufteilbar wie der Prozessor. Betriebsmittel können räumlich aufteilbar sein wie z. B. die Festplatte oder der Arbeitsspeicher. Betriebsmittel können auch nur exklusiv benutzbar und nicht aufteilbar sein wie z. B. ein Drucker.

Eine Synchronisation dient zur Sicherstellung von zeitlichen Beziehungen zwischen Teilfolgen verschiedener Prozesse. Teilfolgen, die sich wechselseitig ausschließen, heißen kritische Bereiche (kritische Abschnitte, critical sections). Kritische Abschnitte sind kritisch in dem Sinn, dass gleichzeitig nur ein einziger Prozess einen kritischen Abschnitt bearbeiten kann.

Kapitel **16.3** untersucht schwer- und leichtgewichtige Prozesse.

Ein klassischer Betriebssystem-Prozess stellt eine Einheit sowohl für das Memory Management als auch für das Scheduling dar. Er wird als schwergewichtiger Prozess (heavyweight process) bezeichnet. Ein Thread ist nur eine Einheit für das Scheduling, nicht jedoch für das Memory Management. Ein Thread wird als leichtgewichtiger Prozess (lightweight process) bezeichnet.

Eine Java-Virtuelle Maschine selbst läuft in einem Betriebssystem-Prozess ab, d. h. sollen mehrere Java-Programme in getrennten Betriebssystem-Prozessen ablaufen, so hat jeder Betriebssystem-Prozess seine eigene Java-Virtuelle Maschine.

Kapitel **16.4** und Kapitel **16.5** befassen sich mit den Zuständen und Zustandsübergängen zum einen von Betriebssystem-Prozessen und zum anderen von Threads.

Kapitel **16.6** behandelt das Programmieren von Threads. Es gibt zwei Möglichkeiten, einen Thread zu programmieren:

- durch eine direkte Ableitung der Klasse dieses Threads von der Klasse `Thread`
- oder durch die Übergabe eines Objektes, dessen Klasse die Schnittstelle `Runnable` implementiert, an ein Objekt der Klasse `Thread`.

Implementiert man die Schnittstelle `Runnable` in einer Klasse, die zum Thread werden soll, so schafft man dadurch die Möglichkeit, dass diese Klasse von einer anderen Klasse abgeleitet werden kann. Die Schnittstelle `Runnable` deklariert nur eine einzige Methode, nämlich die Methode `run()`.

In Kapitel **16.7** wird das Scheduling von Threads analysiert.

Als Scheduling bezeichnet man die Zuteilung von Rechenzeit auf einem Prozessor durch ein Betriebssystem oder durch eine Laufzeitumgebung. In der Spezifikation der Java-Virtuellen Maschine wird nur verlangt, dass Threads mit höherer Priorität im Schnitt mehr Rechenzeit erhalten sollen als Threads mit niedriger Priorität.

Kapitel **16.8** befasst sich mit dem Zugriff auf gemeinsame Ressourcen. Variablen, die von mehreren Threads gemeinsam als Shared Memory benutzt werden, werden auch als kritische Variablen bezeichnet, weil diese Variablen durch die Verwendung in mehreren Threads inkonsistent werden können.

Ein kritischer Abschnitt eines Threads ist eine Folge von Befehlen, die ein Thread nacheinander vollständig abarbeiten muss, auch wenn er vorübergehend die CPU an einen anderen Thread abgibt. Kein anderer Thread darf einen kritischen Abschnitt betreten, der auf die gleiche kritische Variable zugreift, solange der erstgenannte Thread mit der Abarbeitung der Befehlsfolge noch nicht fertig ist.

Das Konzept eines Monitors lokalisiert gemeinsam benutzte Daten und Zugriffsfunktionen zentral an einer einzigen Stelle und nicht wie im Falle von Semaphoren getrennt und über mehrere Prozesse verteilt im Programmcode. Die grundlegenden Eigenschaften eines Monitors sind:

- Kritische Abschnitte, die auf denselben Daten arbeiten, sind Methoden eines Monitors.
- Ein Prozess betritt einen Monitor durch den Aufruf einer Methode des Monitors.
- Nur ein einziger Prozess kann zur selben Zeit den Monitor benutzen. Jeder andere Prozess, der den Monitor aufruft, wird suspendiert und muss warten, bis der Monitor verfügbar wird.

Werden eine oder mehrere Klassenmethoden mit dem Schlüsselwort `synchronized` versehen, so wird ein Monitor um diese Methoden herumgebaut. Dadurch kann nur ein einziger Thread zu einer bestimmten Zeit eine der synchronisierten Methoden bearbeiten. Ein Monitor legt sich nur um die synchronisierten Methoden eines Objektes. Methoden, die nicht `synchronized` sind, werden von dem Monitor nicht geschützt.

Werden mehrere Instanzmethoden einer Klasse mit dem Schlüsselwort `synchronized` versehen, so hat jedes Objekt dieser Klasse einen eigenen Monitor, der den Zugriff auf diese Instanzmethoden überwacht.

Eine `synchronized`-Anweisung, um einen Block zu synchronisieren, wird mit dem Schlüsselwort `synchronized` eingeleitet. In den runden Klammern erwartet die `synchronized`-Anweisung eine Referenz auf ein Objekt, das als Schlüssel dient. Die nachfolgenden geschweiften Klammern schließen die Anweisungen eines kritischen Abschnitts ein. Alle Blöcke, die das gleiche Schlüsselobjekt verwenden, haben einen gemeinsamen Monitor, der den Zutritt zu allen kritischen Abschnitten überwacht. Alle synchronisierten Abschnitte, die das gleiche Schlüsselobjekt als Schlüssel verwenden, schließen sich gegenseitig aus, da sie einen gemeinsamen Monitor verwenden. Wird eine `synchronized`-Anweisung betreten, so wird nachgeschaut, ob das Schlüsselobjekt, auf das die übergebene Referenz zeigt, schon bereits von einem anderen Thread als Schlüssel benutzt wird. Ist dies der Fall, so muss der gerade angefragte Thread warten, bis das Schlüsselobjekt freigegeben wird.

Bei synchronisierten

- Instanzmethoden wird als Schlüsselobjekt das eigene Objekt verwendet,
- Klassenmethoden wird als Schlüsselobjekt das entsprechende Objekt der Klasse `Class<T>` verwendet
- und Blöcken wird als Schlüsselobjekt das Objekt verwendet, auf das die übergebene Referenz zeigt.

Kapitel **16.9** behandelt Dämon-Threads.

Dämon-Threads sind Threads, die für eine Applikation oder für andere Threads Dienstleistungen erbringen. Sie haben dabei oft nicht – wie allgemein bei Threads üblich – eine Abbruchbedingung, sondern laufen meist in einer Endlosschleife ab.

Der Java-Interpreter wird erst beendet, wenn keine Threads mehr abgearbeitet werden. Dies gilt nicht für Dämon-Threads. Sind nur noch Dämon-Threads in einer Java-Virtuellen Maschine vorhanden, gibt es für diese Dämon-Threads, die Dienstleistungen für andere Threads erbringen sollen, nichts mehr zu tun. Die Java-Virtuelle Maschine wird trotz aktiver Dämon-Threads beendet.

16.11 Übungen

Aufgabe 16.1: Erzeugen eines Threads

a) Erzeugen eines Threads durch Ableiten von der Klasse `Thread`

Das nachfolgende Programm ist ein Beispiel dafür, wie man einen eigenen Thread durch Ableiten von der Klasse `Thread` erzeugen kann.

Kompilieren und starten Sie das folgende Programm:

```
// Datei: EigenerThread.java

public class EigenerThread extends Thread
{
```

```
    public void run()
    {
       for (int a = 0; a <= 5; a++)
          System.out.println ("Hier EigenerThread > " + a);
    }
}

// Datei: ThreadTest1.java
public class ThreadTest1
{
    public static void main (String[] args)
    {
       EigenerThread meinThread = new EigenerThread();
       System.out.println ("Hier ist main(), ich starte Thread.");
       meinThread.start();
       System.out.println ("Hier ist main(), ich bin fertig.");
    }
}
```

b) Erzeugen eines Threads mit Hilfe der Schnittstelle `Runnable`

Ändern Sie die Klasse `EigenerThread` und `ThreadTest1` so ab, dass die Klasse `EigenerThread` nicht von der Klasse `Thread` abgeleitet wird, sondern die Schnittstelle `Runnable` implementiert.

Aufgabe 16.2: Helfen Sie Cowboy Jim

Cowboy Jim hat neuerdings Schwierigkeiten, beim Tabakkauen möglichst cool zu gehen. Entwerfen Sie die Klassen `LaufThread` und `KauThread` als separate Threads. In der `run()`-Methode der Klasse `LaufThread` soll Jim abwechselnd sein linkes und rechtes Bein benutzen. Geben Sie hierzu abwechselnd "links" bzw. "rechts" auf dem Bildschirm aus. In der `run()`-Methode der Klasse `KauThread` soll die wiederholte Ausgabe eines einfachen "Schmatz" auf dem Bildschirm erfolgen.

Schreiben Sie eine weitere Klasse `CowboyJim`, welche die beiden Threads startet. Starten Sie die Klasse `CowboyJim` mehrmals und erklären Sie die Ausgabe.

Auf schnellen Rechnern sollten Sie die Schleifen in den `run()`-Methoden der Klassen `LaufThread` und `KauThread` mindestens 1000 mal durchlaufen, damit sich die Threads gegenseitig unterbrechen. Selbstverständlich kann diese Zahl je nach Hardware-Ausstattung nach oben und unten variieren. Wird die Anzahl der Schleifendurchläufe in den Threads zu niedrig gewählt, dann werden die einzelnen Threads komplett abgearbeitet, bevor zum anderen Thread umgeschaltet wird. Ein "cooles Gehen beim Tabakkauen" ist damit für Cowboy Jim nicht möglich. Sollte die Textausgabe so groß sein, dass sie trotz Gebrauch der Scrollbalken nicht komplett im Konsolenfenster angezeigt werden kann, dann geben Sie das Ergebnis seitenweise aus oder leiten es in eine Textdatei um. Zum Beispiel in Windows:

```
java CowboyJim | more
```

oder

```
java CowboyJim > cowboyJim.txt
```

Aufgabe 16.3: Synchronisation

Die Problematik der Synchronisation soll im Folgenden betrachtet werden:

a) Schreiben Sie eine Klasse `Zwischenspeicher`, die eine Instanzvariable `wert` vom Typ `int` besitzt und diese Instanzvariable über get- und set-Methoden öffentlich zur Verfügung stellt.
b) Schreiben Sie eine Klasse `ZahlThread`, die einen Thread darstellt. Beim Erzeugen eines Objektes der Klasse `ZahlThread` wird eine Referenz auf ein Objekt der Klasse `Zwischenspeicher` übergeben, die in einem privaten Datenfeld abgespeichert werden soll. Die Methode `run()` soll einen zufälligen Wert in den Zwischenspeicher schreiben, 2 Sekunden warten, diesen Wert erneut aus dem Zwischenspeicher auslesen und überprüfen, ob der gelesene Wert mit dem geschriebenen übereinstimmt. Weichen die beiden Werte voneinander ab, so soll eine Meldung ausgegeben werden.
c) Schreiben Sie eine Klasse `ZwischenspeicherTest`, die in der `main()`-Methode eine Instanz der Klasse `Zwischenspeicher` erzeugt. Erzeugen und starten Sie dann einen Thread der Klasse `ZahlThread`. Überprüfen Sie, welcher Wert in den Zwischenspeicher geschrieben und welcher Wert aus dem Zwischenspeicher gelesen wird. Fügen Sie hierzu Kontrollausgaben in der Methode `run()` ein.
d) Erzeugen Sie nun in der Methode `main()` mindestens einen weiteren Thread der Klasse `ZahlThread`. Es wird zu fehlerhaften Werten kommen. Wie kann dies unterbunden werden? Erweitern Sie Ihr Programm so, dass keine fehlerhaften Werte mehr auftreten.

Aufgabe 16.4: Stack-Zugriff mit mehreren Threads

a) Es soll ein anonymes Paket aus den Dateien

- `Test.java`,
- `Reader.java`,
- `Writer.java`
- und `Stack.java`

erstellt werden.

Die Aufgabe dieser Klassen ist:

- `Writer.java`

  Die Datei `Writer.java` enthält eine Klasse `Writer`, die von der Klasse `Thread` ableitet. Der `Writer`-Thread schreibt insgesamt 100 `int`-Zahlen auf den Stack.

- `Reader.java`

  Die Datei `Reader.java` enthält eine Klasse `Reader`, die ebenfalls von der Klasse `Thread` ableitet. Der `Reader`-Thread liest die `int`-Zahlen vom Stack ein und gibt sie auf der Konsole aus.

- `Test.java`

  Die Datei `Test.java` enthält eine Klasse `Test`, die nur eine Methode `main()` zum Testen der anderen Klassen einhüllt. In der Methode `main()` soll ein

Objekt der Klasse `Stack` erzeugt werden. Eine Referenz auf dieses Objekt soll an die zu erzeugenden Instanzen der Klassen `Writer` und `Reader` übergeben werden. Starten Sie daraufhin die beiden Threads `Writer` und `Reader`. Warten Sie, bis die beiden Threads ihre Aufgabe erledigt haben und beenden Sie danach die Anwendung. Die Threads sollen willkürlich auf den Stack zugreifen. Ist der Stack beim Schreiben voll, so soll der `Writer`-Thread warten. Der `Reader`-Thread liest die Zahlen einzeln vom Stack. Ist der Stack leer, so soll er warten, bis er erneut Werte lesen kann. Dieser Ablauf soll solange fortgesetzt werden, bis alle 100 Zahlen vom `Writer`-Thread auf den Stack geschrieben wurden und vom `Reader`-Thread ausgelesen wurden. Machen Sie den Stack nicht zu groß, damit die Zahlen vom `Writer`-Thread nicht auf einmal in den Stack geschrieben werden können.

- `Stack.java`

  Zum Schreiben auf den Stack dient die Methode `push()` und zum Lesen vom Stack dient die Methode `pop()` der Klasse `Stack`. Synchronisieren Sie die Methoden des Stacks, sodass es zu keinen Synchronisationsproblemen kommt.

b) Ersetzen Sie Ihre Implementierung des Stacks mit der von Java mitgelieferten Klasse `LinkedBlockingDeque` aus dem Paket `java.util.concurrent`. Passen Sie Ihre Klassen so an, dass diese Klasse anstelle der Klasse `Stack` benutzt wird. Nutzen Sie dabei die Möglichkeiten der generischen Typisierung (Generics) und programmieren Sie – wo möglich – gegen die Schnittstelle `BlockingDeque`.

Um einen blockierenden Effekt zu erzielen, darf man die Methoden `push()` und `pop()` der Schnittstelle `BlockingQueue` nicht verwenden. Verwenden Sie stattdessen die Methoden `putFirst()` statt `push()` und `takeFirst()` statt `pop()`.

Aufgabe 16.5: Das Philosophenproblem

Zu den klassischen Synchronisationsproblemen zählt das Problem der speisenden Philosophen ("Dining Philosophers").

Der Tagesablauf eines Philosophen besteht abwechselnd aus Nachdenken und Essen. Fünf Philosophen sitzen an einem runden Tisch.

Jeder Philosoph hat seinen festen Platz am Tisch, vor ihm einen Teller mit Spaghetti und zu seiner Linken und Rechten liegt jeweils eine Gabel, die er mit seinen direkten Nachbarn teilt.

Das Problem der Philosophen besteht nun darin, dass ein Philosoph nur essen kann, wenn er zwei Gabeln in den Händen hält. Hierbei darf ein Philosoph jedoch nur die Gabel zu seiner linken sowie zu seiner rechten Seite verwenden. Hat sich ein Philosoph zum Essen beide Gabeln seitlich seines Tellers genommen, können seine direkten Tischnachbarn selbst nichts von ihren Tellern essen, da ihnen mindestens eine Gabel fehlt. Dies bedeutet also, dass es nicht möglich ist, dass zwei nebeneinander sitzende Philosophen zur gleichen Zeit essen.

Ist ein Philosoph vorläufig fertig mit Essen, legt er seine beiden Gabeln wieder an die Stelle zurück, von dieser er sie genommen hat und gibt diese damit sozusagen seinen Tischnachbarn frei. Anschließend verfällt er in einen jeweils unterschiedlich langen Zustand des Philosophierens und Nachdenkens, bis er wieder Hunger bekommt.

Lösen Sie die Aufgabe, dass keiner der Philosophen unendlich lange auf die zweite Gabel warten muss, d. h. nicht verhungert. Zudem sollen möglichst viele Philosophen gleichzeitig essen können, wenn sie Hunger haben und das Nachdenken beendet haben. Unterscheiden Sie hierbei die folgenden beiden Fälle:

a) Schreiben Sie ein Java-Programm für den Fall, dass ein Philosoph gleichzeitig beide Gabeln in die Hand nimmt.
b) Schreiben Sie ein weiteres Java-Programm für den Fall, dass ein Philosoph immer zuerst die linke Gabel und danach erst die rechte Gabel in die Hand nimmt. Beachten Sie, dass es hierbei zu sogenannten Deadlocks kommen kann. Zum Beispiel dann, wenn alle die linke Gabel in der Hand halten und auf die rechte Gabel warten.

Aufgabe 16.6: Flughafen-Projekt – Threads

In der aktuellen Version können Flugzeuge immer nur nacheinander landen und starten. Dies soll nun in einer verbesserten Version parallelisiert werden. Erst dadurch kann ein Lotse einen Flughafen wirklich effizient auslasten. Um dies zu erreichen, soll eine Thread-Klasse geschrieben werden, welche zyklisch die Methode `aktualisiereStatus()` der Klasse `Flughafen` aufruft.

Kapitel 17

Generizität

17.1 Generische Klassen
17.2 Eigenständig generische Methoden
17.3 Bounded Typ-Parameter und Wildcards
17.4 Generische Schnittstellen
17.5 Die Klasse `Class<T>`
17.6 Generizität bei variablen Parameterlisten
17.7 Generizität und Polymorphie
17.8 Zusammenfassung
17.9 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_17

17 Generizität

Seit Java 5 unterstützt Java die Definition

- generischer Klassen,
- generischer Schnittstellen und
- generischer Methoden.

Diese werden in Java als **Generics** bezeichnet.

Generisch bedeutet in diesem Zusammenhang, dass Klassen, Schnittstellen und Methoden Parameter verwenden, welche einen Typ darstellen.

Dieses Konzept dient der Typsicherheit, wie im Folgenden noch gezeigt wird.

Die Generizität erlaubt es, die Definition von Klassen, Methoden und Schnittstellen mit Hilfe von einem oder mehreren formalen Typ-Parametern durchzuführen. Dadurch werden parametrisierbare Elemente geschaffen, die im Programm mit konkreten Datentypen als aktuellen Parametern aufgerufen werden können. Dies sieht man am besten an einem Beispiel. Definiert man eine Klasse `Punkt` mit einem **formalen Typ-Parameter `T`**:

```
public class Punkt <T>
{
   private T x;
   private T y;

   public Punkt (T x, T y)
   {
      this.x = x;
      this.y = y;
   }
   // weitere Elemente der Klasse Punkt
}
```

so kann man beispielsweise Punkte mit Koordinaten vom Typ `Integer` oder `Double` anlegen durch:

```
// Aktueller Typ-Parameter Integer tritt an die Stelle von T
Punkt<Integer> integerPunkt = new Punkt<Integer> (1, 2);
```

oder

```
Punkt<Integer> integerPunkt = new Punkt<>(1, 2);
               //<> ist der Diamant-Operator (neu seit Java 7)
```

bzw.

```
// Aktueller Typ-Parameter Double tritt an die Stelle von T
Punkt<Double> doublePunkt = new Punkt<Double> (1.0, 2.0);
```

Seit Java 7 gibt es auch die Schreibweise <>:

```
Punkt<Double> doublePunkt = new Punkt<>(1.0, 2.0);
                                          // seit Java 7
```

Der Operator <> wird auch als **Diamant-Operator** bezeichnet.

Eine Klasse mit formalem **Typ-Parameter** stellt eine **generische Klasse** dar. Der formale Typ-Parameter ist ein symbolischer Name, der wie ein normaler Bezeichner in Java aufgebaut ist und in spitzen Klammern eingeschlossen nach dem Klassennamen angegeben werden muss.

Die Klasse `Punkt<T>` stellt eine generische Klasse mit dem formalen Typ-Parameter `T` dar. Die Entscheidung über den konkreten zu verwendenden Typ – hier `Integer` bzw. `Double` – findet erst bei der Nutzung der generischen Klasse statt.

Generische Klassen bringen einen großen Vorteil bei den sogenannten Collections oder Container-Klassen (siehe Kapitel 18), da es nicht mehr erforderlich ist, dass die Collections aus Referenzen auf Objekte der Klasse `Object` aufgebaut sind. In eine Collection, die Elemente vom Typ `Object` enthält, können bekanntermaßen Objekte jeder beliebigen Klasse eingefügt werden, da jede Klasse von der Klasse `Object` abgeleitet ist. Werden die Collections als generische Datentypen definiert, so kann man Collections für bestimmte Typen erzeugen und man merkt bereits beim Kompilieren, ob versucht wird, eine Referenz auf ein Objekt eines falschen Typs in der Collection abzulegen. Dadurch werden Laufzeitfehler vermieden. Während bis einschließlich der Version Java 1.4 alle Collection-Klassen auf der Klasse `Object` basierten, sind seit Java 5 alle Collection-Klassen generisch.

Mit der **Generizität** von Klassen, Schnittstellen und Methoden werden die folgenden Ziele verfolgt:

- **Höhere Typsicherheit**: Erkennen von Typ-Umwandlungsfehlern zur Kompilierzeit statt zur Laufzeit.
- **Wiederverwendbarkeit von Quellcode**.
- **Vermeiden des expliziten Casts**, der beim Auslesen aus einer Collection aus Elementen vom Typ `Object` notwendig ist.

Kapitel 17.1 zeigt, wie man generische Klassen in Java definiert und wo die entscheidenden Unterschiede zu herkömmlichen Klassen liegen. Kapitel 17.2 befasst sich mit der Definition eigenständig generischer Methoden und Kapitel 17.3 zeigt die Verwendung sogenannter Wildcards. Nach der Definition generischer Schnittstellen in Kapitel 17.4 wird in Kapitel 17.5 die generische Klasse `Class<T>` vorgestellt. Kapitel 17.6 geht auf die Generizität bei variablen Parameterlisten ein. Zum Schluss wird in Kapitel 17.7 über den Zusammenhang von Generizität und Polymorphie reflektiert.

Dieses "Theorie"-Kapitel sollte zunächst übersprungen und erst bei Bedarf oder Interesse gelesen werden.

17.1 Generische Klassen

Am Beispiel der Klasse `Punkt` wird gezeigt, wie generische Klassen implementiert werden können. Angenommen, die Klasse `Punkt` sei nicht generisch, dann könnte ihre Definition wie folgt aussehen:

```
public class Punkt
{
   private Integer x;
   private Integer y;

   public Punkt (Integer x, Integer y)
   {
      this.x = x;
      this.y = y;
   }
   // weitere Elemente der Klasse Punkt
}
```

Nachteilig an dieser Definition ist, dass hiermit die Koordinaten eines Punktes immer vom Typ `Integer` sein müssen. Will man vielleicht die Genauigkeit der Koordinaten eines Punktes erhöhen – etwa durch die Verwendung von Gleitkommawerten – so müsste dafür eine eigene Klasse – beispielsweise die Klasse `PunktDouble` – definiert werden. Der erst genannten Klasse würde man dann vermutlich zur Differenzierung den Namen `PunktInteger` geben.

Eleganter lässt sich dieses Problem durch die Verwendung von **generischen Klassen** lösen.

Generische Klassen werden mit einem sogenannten **formalen Typ-Parameter** deklariert, der in spitzen Klammern <> direkt hinter dem Klassennamen angegeben wird.

Dies zeigt das folgende Beispiel:

```
public class Punkt<T>
```

Der Typ-Parameter `T` wird nun als Stellvertreter innerhalb der Klassendefinition verwendet. Dort, wo bei der herkömmlichen Definition der Klasse `Punkt` der Datentyp `Integer` verwendet wurde – beispielsweise bei der Definition der Instanzvariablen `x` und `y` – wird nun der formale Typ-Parameter `T` eingesetzt:

```
public class Punkt<T>
{
   private T x:
   private T y;

   public Punkt (T x, T y)
   {
      this.x = x;
      this.y = y;
```

```
    }
    // weitere Elemente der Klasse Punkt<T>
}
```

Der **formale Typ-Parameter** `T` in der Klassendeklaration wird auch zur Definition der Datenfelder der generischen Klasse verwendet. Genauso tritt er als Typ für formale Parameter und Rückgabewerte von Methoden auf. Damit können Klassen unabhängig von einem speziellen Typ generisch definiert werden. Der formale Typ-Parameter wird bei der **Verwendung** der Klasse dann durch den gewünschten **konkreten Datentyp** ersetzt.

Bei der Verwendung einer generischen Klasse wird der formale Typ-Parameter durch einen **aktuellen Typ-Parameter** ersetzt.

Aktuelle Typ-Parameter für die generische Klasse `Punkt<T>` sind beispielsweise die Klassen `Integer`, `Float` oder `Double`. So wird ein Punkt mit den Koordinaten (1, 2), dessen Koordinaten vom Typ `Integer` sein sollen, folgendermaßen erzeugt:

```
Punkt<Integer> integerPunkt = new Punkt<Integer> (1, 2);
```

oder in einer verkürzenden Schreibweise:

```
Punkt<Integer> integerPunkt = new Punkt<> (1, 2);
                                        // mit Diamant-Operator
```

Zu beachten ist, dass das Anlegen der Referenzvariablen `integerPunkt` für die aktuell parametrisierte Klasse `Punkt<Integer>` erfolgt.

Werden die formalen Typ-Parameter durch aktuelle Typ-Parameter ersetzt, so handelt es sich um eine konkrete Ausprägung einer generischen Klasse, in anderen Worten um eine **aktuell parametrisierte Klasse**.

Es existiert eine **Kurzschreibweise** bei der Erzeugung von generischen Typen. Die Angabe des **aktuellen Typ-Parameters** kann für den Aufruf des `new`-Operators seit Java 7 **weggelassen werden**, wie in folgendem Beispiel gezeigt wird:

```
Punkt<Integer> integerPunkt = new Punkt<> (1, 2);
```

Die Schreibweise `<>` wird als **Diamant-Operator** bezeichnet.

Somit ist das folgende Beispiel ebenfalls korrekt: Einer Instanzvariablen vom Typ `Punkt<Integer>` wird im Konstruktor mit `new Punkt<> (x, y)` eine Referenz auf ein Objekt vom Typ `Punkt<Integer>` zugewiesen, ohne den aktuellen Typ-Parameter `Integer` für den Aufruf des `new`-Operators und des Konstruktors anzugeben. Hier das bereits erwähnte **Beispiel**:

```
class PunktBehaelter
{
   private Punkt<Integer> integerPunkt;

   public PunktBehaelter (int x, int y)
   {
      this.integerPunkt = new Punkt<> (x, y);
   }

   public void setPunkt (Punkt<Integer> integerPunkt)
   {
      this.integerPunkt = integerPunkt;
   }

   public Punkt getPunkt()
   {
      return integerPunkt;
   }
   // weitere Elemente der Klasse PunktBehaelter
}
```

Entsprechend kann die vereinfachte Schreibweise des **Diamant-Operators** auch beim Aufruf der Methode `setPunkt()` der Klasse `PunktBehaelter` wie folgt eingesetzt werden:

```
PunktBehaelter behaelter = new PunktBehaelter (1, 2);
behaelter.setPunkt (new Punkt<> (3, 4));
```

Bei einem **expliziten Cast** muss jedoch stets der vollständige Typ inklusive eines aktuellen Typ-Parameters angegeben werden, wie in folgendem Beispiel:

```
Punkt<Integer> punkt = (Punkt<Integer>) behaelter.getPunkt();
```

In den bisherigen Beispielen wurde als aktueller Typ-Parameter für die generische Klasse `Punkt<T>` der Typ `Integer` eingesetzt. Benötigt man Punkte mit einer höheren Präzision, deren Punkt-Koordinaten Gleitkommazahlen darstellen, so verwendet man die aktuell parametrisierte Klasse `Punkt<Double>`, wie in folgendem Beispiel gezeigt wird:

```
Punkt<Double> doublePunkt = new Punkt<> (1.0, 2.0);
```

Der Compiler erzeugt bei der Übersetzung einer generischen Klasse nur **eine einzige Bytecode-Datei**, unabhängig davon, wie viele aktuell parametrisierte Klassen der generischen Klasse verwendet werden. Für die generische Klasse `Punkt<T>` erzeugt der Compiler die Datei `Punkt.class`.

Der Bytecode einer generischen Klasse wird von allen aktuell parametrisierten Klassen gemeinsam benutzt. Dies wird auch als **Code Sharing** bezeichnet.

Wird eine generische Klasse `Punkt<T>` mit dem Typ-Parameter `T` verwendet, so wird dieser durch einen aktuellen Typ-Parameter – z. B. `Integer` bei Punkt `<Integer>` – ersetzt.

Weiterhin tritt der **formale Typ-Parameter** bei der **Deklaration der Instanzmethoden** einer generischen Klasse in Erscheinung. Beispielsweise wird innerhalb der generischen Klasse `Punkt<T>` die Methode zum Abholen der `x`-Koordinate wie folgt definiert:

```
public T getX()
{
   return x; // x ist vom Typ T
}
```

Der formale Typ-Parameter ersetzt immer dort einen konkreten Datentyp innerhalb einer Klassendefinition, wo man Generizität erreichen möchte.

17.1.1 Die generische Klasse `Punkt<T>`

Im folgenden **Beispiel** nun die generische Klasse `Punkt<T>` als Ganzes:

```
// Datei: Punkt.java
public class Punkt<T>
{
   // Ein Punkt hat 2 Koordinaten vom Typ T
   private T x;
   private T y;

   // Der Konstruktor erwartet 2 Parameter vom Typ T
   public Punkt (T x, T y)
   {
      this.x = x;
      this.y = y;
   }

   // get- und set-Methoden für die Koordinaten
   public T getX()
   {
      return x;
   }

   public T getY()
   {
       return y;
   }

   public void setX (T x)
   {
      this.x = x;
   }

   public void setY (T y)
   {
      this.y = y;
   }
```

```
   public String toString()
   {
      return ("x = " + x + ", y = " + y);
   }
}
```

Zum Testen dient die Klasse `PunktTest`. Sie erzeugt je ein Objekt der aktuell parametrisierten Klassen `Punkt<Integer>` und `Punkt<Double>` und gibt anschließend die Koordinaten der beiden Punkte aus:

```
// Datei: PunktTest.java
public class PunktTest
{
   public static void main (String[] args)
   {
      Punkt<Integer> integerPunkt = new Punkt<> (1, 2);
      Punkt<Double> doublePunkt = new Punkt<> (1.0, 2.0);
      System.out.print ("Punkt<T> mit aktuellem Parameter");
      System.out.println (" Integer: " + integerPunkt);
      System.out.print ("Punkt<T> mit aktuellem Parameter");
      System.out.println (" Double: " + doublePunkt);
   }
}
```

Die Ausgabe des Programms ist:

```
Punkt<T> mit aktuellem Parameter Integer: x = 1, y = 2
Punkt<T> mit aktuellem Parameter Double : x = 1.0, y = 2.0
```

17.1.2 Repräsentation aktuell parametrisierter Klassen

Wie bereits erwähnt, wird vom Compiler aus der generischen Klasse `Punkt<T>` der **Bytecode** für nur **eine einzige Klasse** erstellt. Diese Klasse enthält anstelle des formalen Typ-Parameters `T` den Typ `Object`. Aus diesem Grund können auch als aktuelle Typ-Parameter nur Datentypen verwendet werden, die von der Basisklasse `Object` ableiten, d. h. einfache Datentypen dürfen nicht verwendet werden. Die generische Klasse `Punkt<T>` kann dann mit Datentypen wie `Integer`, `Float` oder `Number` arbeiten.

Der aktuelle Typ-Parameter muss ein **Referenztyp** sein. Damit ist es in Java nicht möglich, generische Klassen zu schreiben, die als aktuellen Typ-Parameter einen einfachen Typ wie `int`, `float` oder `double` akzeptieren.

Bild 17-1 zeigt, wie durch Verwendung der generischen Klasse `Punkt<T>` mit verschiedenen aktuellen Typ-Parametern die aktuell parametrisierten Klassen `Punkt<Integer>`, `Punkt<Float>` und `Punkt<Number>` entstehen. **Der Compiler ersetzt** beim Übersetzen dieses Beispiels den **formalen Typ-Parameter `T` durch** den **Typ** `Object` und **lässt im Klassennamen den Typ-Parameter weg**.

Die generische Klasse `Punkt<T>` nimmt je nach aktuellem Typ-Parameter eine unterschiedliche Gestalt an. Sie ist also polymorph. Bitte beachten Sie, dass die **aktuell parametrisierten Klassen auf derselben Ebene** stehen und dass zwischen ihnen **keine Vererbungsbeziehung** besteht.

In Bild 17-1 ist zu sehen, dass der Compiler mit der sogenannten **Type Erasure**[211]**-Technik** beim Kompilieren quasi den formalen Typ-Parameter `T` "ausradiert". Anstelle des formalen Typ-Parameters `T` – als Typ für Datenfelder, formale Parameter und Rückgabewerte – wird die Klasse `Object` eingesetzt. Mit der Type Erasure-Technik wird also genau eine einzige `class`-Datei erzeugt, ungeachtet dessen, wie viele konkrete Ausprägungen durch eine Substitution mit aktuellen Typ-Parametern davon erzeugt werden. Der Compiler-generierte sharable Bytecode wird auch **Raw Type** genannt.

Das folgende Bild zeigt Objekte einiger aktuell parametrisierten Klassen:

Bild 17-1 Objekte der aktuell parametrisierten Klassen `Punkt<Integer>`, `Punkt<Float>` und `Punkt<Number>`

Besonderes Augenmerk soll in Bild 17-1 auf die Instanz der aktuell parametrisierten Klasse `Punkt<Number>` gelegt werden. Bekanntermaßen ist die Klasse `Number` abstrakt. Es können von ihr also keine Instanzen mit Hilfe des `new`-Operators erzeugt werden. Trotzdem ist die Codezeile

[211] Engl. für Typ-Auslöschung (siehe [Mon06])

```
Punkt<Number> punkt = new Punkt<Number> (1, 2);
```

vollkommen korrekt, da eine Instanz der aktuell parametrisierten Klasse `Punkt<Number>` und keine Instanz der Klasse `Number` angelegt wird. Es tritt damit lediglich der aktuelle Typ-Parameter `Number` an die Stelle des formalen Typ-Parameters `T`. Beim Konstruktoraufruf für die aktuell parametrisierte Klasse `Punkt<Number>` werden Referenzen auf Objekte vom Typ `Number` erwartet. Der Compiler übersetzt

```
new Punkt<Number> (1, 2);
```

mit Hilfe des Auto-Boxing zu

```
new Punkt<Number> (Integer.valueOf (1), Integer.valueOf (2));
```

Da Objekte vom Typ `Integer` auch automatisch vom Typ `Number` sind, ist der Aufruf vollkommen korrekt. Das Beispiel stellt jedoch zugleich auch einen Ausnahmefall dar. Es ist nicht möglich, auf dieses Beispiel die verkürzte Schreibweise anzuwenden.

Durch die Type Erasure werden beim Übersetzen einer generischen Klasse alle Vorkommen des formalen Typ-Parameters `T` im Rumpf der Klasse durch den Typ `Object` ersetzt. So wird für die Klasse

```
class GenKlasse<T>
{
   T ref;
}
```

folgender Bytecode erzeugt:

```
class GenKlasse[212]
{
   Object ref;
}
```

Es soll hier angemerkt werden, dass **Type Erasure bei überladenen Methoden zu Problemen führt**. So werden beispielsweise die überladenen Methoden

```
void methode (List<Integer> list);
void methode (List<Double> list);
```

durch Type Erasure beide in

```
void methode (List list);
```

überführt.

Beide Methoden haben also die gleiche Signatur und führen deshalb zu einem Kompilierfehler.

[212] `<Object>` wird nicht notiert.

Das folgende Bild visualisiert das **Speicherabbild** für ein Objekt vom Typ `Punkt<Number>`, das einmal mit Werten vom Typ `int` und einmal mit Werten vom Typ `double` gemäß den folgenden Anweisungen initialisiert wird:

```
Punkt<Number> integerPunkt = new Punkt<Number> (1, 2);
Punkt<Number> doublePunkt = new Punkt<Number> (1.0, 2.0);
```

Hier das bereits erwähnte Speicherabbild:

Bild 17-2 Repräsentation einer Instanz vom Typ `Punkt<Number>` im Falle von Initialisierungsparametern vom Typ `int` und vom Typ `double`

Im Folgenden soll nicht eine Instanz vom Typ `Punkt<Number>`, sondern eine Instanz vom Typ `Punkt<Integer>` betrachtet werden. Der Codeausschnitt

```
Punkt<Integer> ref = new Punkt<> (1, 2);
```

zeigt, wie ein Objekt der aktuell parametrisierten Klasse `Punkt<Integer>` angelegt und die Referenz darauf der Referenzvariablen `ref` zugewiesen wird. Über `ref` können nun wie gewohnt Methodenaufrufe für die aktuell parametrisierte Klasse `Punkt<Integer>` erfolgen. Das Besondere daran ist nun, dass beim Abholen des Rückgabewertes kein expliziter Cast durchgeführt werden muss. Dies zeigt das folgende Beispiel:

```
Integer i = ref.getX(); // OK!
```

Auf den ersten Blick erscheint dieser Umstand nicht intuitiv, weil beim Übersetzen der generischen Klasse `Punkt<T>` alle Vorkommen des formalen Typ-Parameters `T` durch `Object` ersetzt wurden. Somit erzeugt der Compiler für die in der generischen Klasse `Punkt<T>` definierte Methode

```
public T getX() { . . . . . }
```

Bytecode für

```
public Object getX() { . . . . . }
```

Also müsste der Programmierer beim Aufruf der Methode `getX()` eigentlich einen expliziten Cast durchführen – hier auf `Integer`. Dies ist allerdings nicht erforderlich, da der Compiler bei der Übersetzung der Zeile

```
Integer i = ref.getX();
```

den expliziten Cast selbst einfügt:

```
Integer i = (Integer) ref.getX();
```

Die Methode `getX()` gibt also nach wie vor eine Referenz vom Typ `Object` zurück. Diese Referenz wird aber – unsichtbar für den Programmierer – vom Compiler auf den entsprechenden Typ – hier also `Integer` – gecastet.

17.1.3 Klassen mit mehreren formalen Typ-Parametern

Die bisher gezeigten Beispiele weisen alle nur einen einzigen formalen Typ-Parameter auf.

Eine **generische Klasse** kann aber **mehrere formale Typ-Parameter** besitzen. Die **allgemeine Notation generischer Klassen** ist:

```
modifier_opt class GenerischeKlasse<T1, . . . , TN>
   (extends BasisKlasse)_opt
   (implements I1, . . . , IN)_opt
```

Das tiefgestellte $_{opt}$ gibt an, welche Elemente bei der Klassendefinition optional sind. Beachten Sie, dass `modifier` einen Stellvertreter für die möglichen Zugriffsmodifikatoren `public`, `protected` und `private` und die weiteren zulässigen Modifikatoren `abstract` und `final` darstellt.

Die Liste mit den formalen Typ-Parametern in den spitzen Klammern, welche direkt nach dem Klassennamen angeschrieben wird, ist auch unter dem Namen **Typ-Parameter-Sektion** bekannt. Es ist somit möglich, eine generische Klasse mit beliebig vielen formalen Typ-Parametern zu versehen, beispielsweise:

```
class GenerischeKlasse<A, B, C>
{
   A a;
   B b;
   C c;
   // Weitere Verwendung der formalen Typ-Parameter
}
```

Beim Übersetzen der generischen Klasse `GenerischeKlasse<A, B, C>` werden dann durch die Type Erasure – wie bei einer generischen Klasse mit nur einem einzigen formalen Typ-Parameter auch – alle Vorkommen der formalen Typ-Parameter – hier `A`, `B` und `C` – durch den Typ `Object` ersetzt. Wird nun eine generische Klasse mit drei formalen Typ-Parametern verwendet wie im folgenden Beispiel:

```
GenerischeKlasse<Integer, Double, Float> ref =
   new GenerischeKlasse<>();
```

so muss jeder formale Typ-Parameter durch einen aktuellen Typ-Parameter ersetzt werden.

17.1.4 Subtyping und generische Klassen

Aktuell parametrisierte Klassen desselben generischen Typs stehen in keiner Vererbungsbeziehung.

Dies soll an den aktuell parametrisierten Klassen `Punkt<Number>` und `Punkt<Integer>` diskutiert werden. Obwohl die Klassen `Integer` und `Float` von der Klasse `Number` abgeleitet sind, wie in Bild 17-3 gezeigt, kann eine Referenz vom Typ `Punkt<Number>` nicht auf ein Objekt der aktuell parametrisierten `Klasse Punkt<Integer>` zeigen, da beide Klassen sich auf derselben Hierarchiestufe befinden.

Das folgende Bild zeigt die Klassen `Integer` und `Float` als Subtypen der Klasse `Number`:

Bild 17-3 `Integer` und `Float` als Subtypen von `Number`

Sowohl die Klasse `Punkt<Number>` als auch die Klasse `Punkt<Integer>` stellen **aktuell parametrisierte Klassen** des generischen Typs `Punkt<T>` dar.

Das folgende Bild visualisiert, dass Objekte aktuell parametrisierter Klassen nicht in einer Vererbungshierarchie stehen, obwohl die aktuellen Parameter sich in einer Vererbungshierarchie befinden:

Bild 17-4 Keine Vererbungshierarchie bei aktuell parametrisierten Klassen trotz Vererbungshierarchie der Parameter

Die Objekte der verschiedenen aktuell parametrisierten Klassen werden erzeugt, indem der aktuelle Typ-Parameter – hier `Integer` oder `Number` – an die Stelle des formalen Typ-Parameters – hier `T` – tritt. Dass die aktuell parametrisierten Klassen `Punkt<Number>` und `Punkt<Integer>` zueinander keine Vererbungsbeziehungen haben und dadurch eine Referenz vom Typ `Punkt<Number>` nicht auf ein Objekt vom Typ `Punkt<Integer>` zeigen kann (siehe auch Bild 17-4), wird im folgenden Beispiel demonstriert, wobei zu Beginn des Beispiels absichtlich von dem Trugschluss ausgegangen wird, die aktuell parametrisierte Klasse `Punkt<Integer>` stünde mit der aktuell parametrisierten Klasse `Punkt<Number>` in einer Vererbungsbeziehung. Basierend auf dieser falschen Annahme wird dann anhand des Beispiels gezeigt, dass diese Annahme unhaltbar ist.

Und nun als Gedankenexperiment ein Beispiel auf Basis des soeben erwähnten Trugschlusses, dass der Typ `Punkt<Integer>` mit dem Typ `Punkt<Number>` in einer Vererbungsbeziehung stehen würde. Die dazu verwendete generische Klasse `Punkt<T>` ist in Kapitel 17.1.1 dokumentiert. Verwendet ein Programm Referenzen auf Objekte, so kann infolge des liskovschen Substitutionsprinzips ein Objekt einer abgeleiteten Klasse stets an die Stelle eines Objektes einer Basisklasse treten, solange die Verträge nicht verletzt werden. Die Basisklasse von `Integer` und `Double` ist die Klasse `Number`. Somit ist die Zuweisung

```
Number number = new Double (1);
```

gültig und der Compiler übersetzt diese Anweisung ohne Probleme. Auch die folgende Anweisung ist gültig:

```
Punkt<Integer> integerPunkt = new Punkt<>(1, 2);
```

Die nächste Anweisung

```
Punkt<Number> numberPunkt = integerPunkt;
```

ist jedoch falsch. Der Typ `Punkt<Integer>` ist nicht von `Punkt<Number>` abgeleitet. Daher kann die Referenz `numberPunkt` nicht auf ein Objekt vom Typ `Punkt<Integer>` zeigen.

Der Compiler weiß das auch und lehnt die Übersetzung ab. Im Folgenden soll nun als **Gedankenexperiment** angenommen werden, der Compiler würde die letzte Zeile doch übersetzen. Es sollen nun die beiden folgenden Anweisungen abgearbeitet werden, die zweifellos korrekt sind:

```
Number number = new Double (10);
numberPunkt.setX (number);
```

Es wird hier eine Referenzvariable vom Typ `Number` angelegt, die auf ein Objekt vom Typ `Double` zeigt, was nach dem liskovschen Substitutionsprinzip erlaubt ist. Über die Referenzvariable `numberPunkt` wird dann die Methode `setX()` der aktuell parametrisierten Klasse `Punkt<Number>` aufgerufen und ihr die Referenz auf ein Objekt vom Typ `Double` übergeben.

`numberPunkt` zeigt aber in dem durchgeführten Gedankenexperiment auf ein Objekt vom Typ `Punkt<Integer>`. Dies hätte nun zur Folge, dass über den Methodenaufruf `setX()` der Instanzvariablen `x` vom Typ `Integer` eine Referenz auf ein Objekt vom Typ `Double` zugewiesen würde, was natürlich nicht möglich ist, weil `Integer` und `Double` auf derselben Stufe der Vererbungshierarchie stehen und insbesondere `Double` kein Subtyp von `Integer` ist. Aber soweit kommt es nicht, da der Compiler die Regeln kennt. Diese lauten:

Angenommen, die Klasse `K1` ist die Basisklasse für eine Klasse `K2` und `GenKlasse` stellt die Definition einer generischen Klasse dar. Dann gilt **nicht**, dass `GenKlasse<K2>` einen Subtyp von `GenKlasse<K1>` darstellt!

17.1.5 Einschränkungen bei generischen Klassen

Es gibt einige wichtige **Einschränkungen** bei der Verwendung eines formalen Typ-Parameters:

- Ein formaler Typ-Parameter einer Klasse darf nicht bei der Definition von Klassenvariablen und -methoden verwendet werden.
- Von einem **formalen Typ-Parameter** kann **kein Objekt** – beispielsweise mit `new T` – angelegt werden.
- Ein **formaler Typ-Parameter** darf **nicht** in Verbindung **mit** dem **`instanceof`-Operator** verwendet werden. Eine Typ-Prüfung `ref instanceof T` ist generell nicht zulässig.
- Eine **Referenzvariable auf ein Array**, das Referenzen auf Objekte aktuell parametrisierter Klassen enthalten soll, darf **nur unter Verwendung der Unbounded Wildcard ?**[213] angelegt werden.
- Auch das **Anlegen eines Array-Objektes** unter Verwendung eines aktuellen oder formalen Typ-Parameters ist unzulässig und es muss wiederum die Unbounded **Wildcard ?** eingesetzt werden. Ein korrektes Array ist beispielsweise: `Punkt<?>[] arr = new Punkt<?>[10];`

Der Compiler wird Verstöße gegen diese Einschränkungen mit einem Übersetzungsfehler bemängeln und der Programmierer muss entsprechend nachbessern.

Die Einschränkung, dass der formale Typ-Parameter beispielsweise nicht bei der Deklaration von Klassenvariablen verwendet werden darf, kann einfach nachvollzogen werden: Der Bytecode einer generischen Klasse existiert in einer einzigen Ausprägung. Es können aber beliebig viele aktuell parametrisierte Klassen desselben generischen Typs erzeugt werden.

Eine Klassenvariable kann nicht gleichzeitig mehrere Typen repräsentieren, deshalb kann ein formaler Typ-Parameter nicht zur Definition von Klassenvariablen verwendet werden.

[213] Siehe Kapitel 17.3.2.

17.1.6 Generische Klassen und Vererbungsbeziehungen

Generische Klassen können – wie herkömmliche Klassen auch – Teil einer Vererbungshierarchie sein. Dies bedeutet, dass eine generische Klasse

- eine andere generische Klasse oder eine herkömmliche Klasse erweitern kann
- oder aber eine generische Klasse die Vaterklasse einer herkömmlichen Klasse ist.

Diese drei Fälle werden in Kapitel 17.1.6.1 bis 17.1.6.3 näher betrachtet.

17.1.6.1 Generische Klasse leitet von generischer Klasse ab

Leitet eine generische Klasse von einer anderen generischen Klasse ab, so muss der formale Typ-Parameter der Klasse, von der abgeleitet wird, entweder

- durch den formalen Typ-Parameter der abgeleiteten Klasse oder
- durch einen aktuellen Typ-Parameter ersetzt werden.

Hier zunächst die **erste Möglichkeit**, dass der formale Typ-Parameter der Klasse, von der abgeleitet wird, durch den formalen Typ-Parameter der abgeleiteten Klasse ersetzt wird:

```
public class B<T> extends A<T>
```

In diesem Fall ist der formale Typ-Parameter `T` der Klasse `B` der aktuelle Typ-Parameter der Klasse `A`. Wird nun die Klasse `B` mit einem aktuellen Parameter `AktuellerParameter` instanziiert, so werden auch alle Vorkommen des formalen Typ-Parameters innerhalb der Klassendefinition von Klasse `A` durch `AktuellerParameter` ersetzt.

Und nun die **zweite Möglichkeit** in einem Beispiel:

```
public class B<T> extends A<KonkreteKlasse>
```

Eine Definition in dieser Form hat zur Konsequenz, dass alle Vorkommen eines für die Klasse `A` definierten formalen Typ-Parameters beim Ableiten durch den Typ `KonkreteKlasse` ersetzt werden. Wird die Klasse `B` mit einem aktuellen Typ-Parameter `AktuellerParameter` instanziiert:

```
B<AktuellerParameter> b = new B<>();
```

so wird der formale Typ-Parameter `T` nur in der Klassendefinition der Klasse `B` durch `AktuellerParameter` ersetzt, nicht aber in der Klassendefinition der Klasse `A`, weil dort der formale Typ-Parameter schon durch `KonkreteKlasse` zum Zeitpunkt des Ableitens – genauer gesagt zum Zeitpunkt des Übersetzens der Klasse `B` – ersetzt wurde.

Das folgende **Beispielprogramm** veranschaulicht beide Fälle:

```
// Datei: AbleitenTest1.java

class GenKlasseA<T>
{
```

```
   public void methodeA (T t)
   {
      System.out.println ("methodeA() gerufen mit " + t);
   }
}

// Zweite Möglichkeit:
// GenKlasseB leitet ab von der aktuell parametrisierten
// Klasse GenKlasse<Integer>. Die Methode methodeA() der
// Klasse GenKlasseA kann somit nur mit Referenzen auf Objekte
// vom Typ Integer aufgerufen werden.
class GenKlasseB<T> extends GenKlasseA<Integer>
{
   public void methodeB (T t)
   {
      System.out.println ("methodeB() gerufen mit " + t);
   }
}
// Erste Möglichkeit:
// GenKlasseC leitet ebenfalls von der generischen Klasse
// GenKlasseA ab, aber ersetzt deren formalen Typ-Parameter T
// durch ihren formalen Typ-Parameter S.
class GenKlasseC<S> extends GenKlasseA<S>
{
   public void methodeC (S s)
   {
      System.out.println ("methodeC() gerufen mit " + s);
   }
}

public class AbleitenTest1
{
   public static void main (String[] args)
   {
      GenKlasseB<Double> ref1 = new GenKlasseB<>();

      // Die Methode methodeB() kann somit nur mit Referenzen auf
      // Double-Objekte aufgerufen werden.
      ref1.methodeB (4.9);

      // Die Methode methodeA() kann nur mit Referenzen auf Objekte
      // vom Typ Integer aufgerufen werden, weil beim Ableiten von
      // der Klasse GenKlasseA deren formaler Typ-Parameter durch
      // Integer ersetzt wurde.
      ref1.methodeA (3);

      // Die Referenz ref1 vom Typ GenKlasseB<Double> kann einer
      // Referenzvariablen ref2 vom Typ GenKlasseA<Integer>
      // zugewiesen werden. GenKlasseA<Integer> ist dabei der
      // Basistyp von GenKlasseB<Double>.
      GenKlasseA<Integer> ref2 = (GenKlasseA<Integer>) ref1;
      ref2.methodeA (5);

      // Instanziierung der Klasse GenKlasseC mit aktuellem Typ-
      // Parameter vom Typ String. Die zurückgelieferte Referenz
      // wird einer Referenzvariablen vom Typ GenKlasseA<String>
      // zugewiesen.
```

```
      GenKlasseA<String> ref3 = new GenKlasseC<>();
      ref3.methodeA ("String");

      GenKlasseC<String> ref4 = (GenKlasseC<String>) ref3;
      ref4.methodeC ("String");
   }
}
```

Die Ausgabe des Programms ist:

```
methodeB() gerufen mit 4.9
methodeA() gerufen mit 3
methodeA() gerufen mit 5
methodeA() gerufen mit String
methodeC() gerufen mit String
```

17.1.6.2 Generische Klasse erweitert herkömmliche Klasse

Im Gegensatz zur vorherigen Möglichkeit, bei der eine generische Klasse von einer anderen generischen Klasse abgeleitet wurde, wird nun der Fall betrachtet, bei dem eine generische Klasse von einer nicht generischen Klasse abgeleitet wird:

```
class GenKlasse<T> extends EinfacheKlasse
```

Der Umstand, dass eine generische Klasse von einer nicht generischen Klasse abgeleitet wird, hat keinen Einfluss auf den Code der nicht generischen Klasse. Die generische Klasse `GenKlasse<T>` erbt hier nur die nicht generischen Eigenschaften wie Datenfelder und Methoden der herkömmlichen Klasse `EinfacheKlasse`. Hierzu ein **Beispiel**:

```
// Datei: AbleitenTest2.java
class EinfacheKlasse
{
   private String s = "Ich bin ein String";

   public void methodeA()
   {
      System.out.println ("Wert des Datenfeldes" +
                          " in EinfacheKlasse: " + s);
   }
}

class GenKlasse<T> extends EinfacheKlasse
{
   private T t;

   public GenKlasse(T t)
   {
      this.t = t;
   }

   public void methodeB()
   {
      System.out.println ("Wert des Datenfeldes" +
                          " in GenKlasse: " + t);
   }
}
```

```
public class AbleitenTest2
{
   public static void main (String[] args)
   {
      GenKlasse<Integer> ref = new GenKlasse<> (4);
      ref.methodeB();
      ref.methodeA();
   }
}
```

Die Ausgabe des Programms ist:

```
Wert des Datenfeldes in GenKlasse: 4
Wert des Datenfeldes in EinfacheKlasse: Ich bin ein String
```

17.1.6.3 Herkömmliche Klasse leitet ab von generischer Klasse

Der dritte Fall beschreibt das Ableiten einer nicht generischen von einer generischen Klasse. Dies kann nur geschehen, wenn **beim Ableiten der formale Typ-Parameter der generischen Klasse durch einen aktuellen Typ-Parameter ersetzt** wird:

```
class EinfacheKlasse2 extends GenKlasse<Integer>
```

Denn aus Sicht der herkömmlichen – also der abgeleiteten – Klasse ist der formale Typ-Parameter der generischen Klasse nicht bekannt und kann somit zur Instanziierungszeit nicht durch einen aktuellen Typ-Parameter ersetzt werden. Diese Ersetzung muss also wiederum bei der Ableitung stattfinden. Das folgende Beispiel zeigt den Zusammenhang. Bitte beachten Sie, dass in der Methode `main()` der Klasse `AbleitenTest3` die Referenzvariable `refGenUniversal` vom Typ `GenKlasse<T>` mit `?` als aktuellen Typ-Parameter angelegt wird. `?` ist hierbei eine sogenannte Wildcard und verleiht der Referenzvariablen die Eigenschaft, dass in ihr Referenzen auf Objekte beliebiger aktuell parametrisierter Klassen des generischen Typs `GenKlasse<T>` abgespeichert werden können. Diese und andere Wildcards werden in Kapitel 17.3 ausführlich besprochen. Nun aber zum angekündigten **Beispiel**:

```
// Datei: AbleitenTest3.java

class GenKlasse<T>
{
   private T t;

   public GenKlasse(T t)
   {
      this.t = t;
   }

   public void methodeA()
   {
      System.out.println ("Wert des Datenfeldes" +
                          " in GenKlasse: " + t);
   }
}
```

```
class EinfacheKlasse2 extends GenKlasse<Integer>
{
   private String s = "Ich bin ein String";

   public EinfacheKlasse2()
   {
      super (2);
   }

   public void methodeB()
   {
      System.out.println ("Wert des Datenfeldes" +
                          " in EinfacheKlasse2: " + s);
   }
}

public class AbleitenTest3
{
   public static void main (String[] args)
   {
      // Die Klasse EinfacheKlasse2 kann nun ganz normal verwendet
      // und instanziiert werden. Es sind keine generischen
      // Eigenschaften mehr vorhanden.
      EinfacheKlasse2 refEinfach = new EinfacheKlasse2();
      refEinfach.methodeA();
      refEinfach.methodeB();

      // Wird die Referenz auf den Typ der Basisklasse gecastet, so
      // muss die Referenzvariable mit dem aktuellen Parameter
      // angelegt werden, mit dem die generische Klasse aktuell
      // parametrisiert wurde - in diesem Fall also Integer.
      GenKlasse<Integer> refGenInteger = refEinfach;
      refGenInteger.methodeA();

      // Oder es muss die Wildcard ? eingesetzt werden.
      GenKlasse<?> refGenUniversal = refEinfach;
      refGenUniversal.methodeA();
   }
}
```

Die Ausgabe des Programms ist:

```
Wert des Datenfeldes in GenKlasse: 2
Wert des Datenfeldes in EinfacheKlasse2: Ich bin
ein String
Wert des Datenfeldes in GenKlasse: 2
Wert des Datenfeldes in GenKlasse: 2
```

17.2 Eigenständig generische Methoden

Eigenständig generische Methoden sind Methoden mit formalen Typ-Parametern, wobei die Klasse, welche die Methode enthält, nicht generisch ist.

Klassenmethoden, **Instanzmethoden** und **Konstruktoren** können als **eigenständige generische Methoden** in einer Klasse existieren, ohne dass die Klasse selbst generisch ist.

Eigenständig generische Methoden können auch innerhalb eines Aufzählungstyps definiert werden. Ein Aufzählungstyp selbst kann jedoch nicht generisch sein.

"Normale" generische Methoden wurden bereits bei der Klasse `Punkt<T>` in Kapitel 17.1.1 behandelt. Bei **eigenständig generischen Methoden** erfolgt die Deklaration einer Klasse **ohne einen formalen Typ-Parameter**. Somit ist innerhalb des Klassenrumpfes kein formaler Typ-Parameter bekannt.

Bei der Deklaration **eigenständig generischer Methoden** steht die **Typ-Parameter-Sektion** direkt vor dem Rückgabewert der **Methode** bzw. direkt vor dem Klassennamen des **Konstruktors**, womit die dort aufgeführten formalen Typ-Parameter für diese Methode bzw. den Konstruktor bekannt gemacht werden.

Die allgemeine Notation einer generischen Instanz- oder Klassenmethode lautet:

```
modifier_opt <T1, . . . , TN>
   returnTyp methodenName (Parameterliste_opt) throws-Klausel_opt
```

bzw. für die Deklaration von Konstruktoren:

```
modifier_opt <T1, . . . , TN>
   Klassenname (Parameterliste_opt) throws-Klausel_opt
```

Das tiefgestellte $_{opt}$ gibt wiederum an, welche Elemente bei der Deklaration einer generischen Methode optional sind. `modifier` steht stellvertretend für die gültigen Zugriffsmodifikatoren `public`, `protected` oder `private`. Bei der Deklaration generischer Instanz- oder Klassenmethoden steht `modifier` auch zusätzlich für die möglichen Modifikatoren `static`, `final` und `abstract`.

Die Gültigkeit der durch die Typ-Parameter-Sektion bekannt gemachten formalen Typ-Parameter bezieht sich nicht wie bei einer generischen Klasse auf die gesamte Klasse, sondern nur auf die entsprechende Methode.

Das folgende Beispiel definiert die gewöhnliche Klasse `EinfacheKlasse`. In ihr ist eine gewöhnliche Methode `einfacheMethode()` und eine eigenständig generische Methode `generischeMethode()` definiert. Die eigenständig generische Methode kann als Übergabeparameter eine Referenz auf ein Objekt eines beliebigen Typs haben. Weiterhin besitzt diese Klasse einen eigenständig generischen Konstruktor, der ebenfalls als Übergabeparameter eine Referenz auf ein Objekt eines beliebigen Typs hat.

Jetzt das **Beispiel**:

```
// Datei: EinfacheKlasse.java

public class EinfacheKlasse
{
   private Integer datenfeld;

   // Eigenständig generischer Konstruktor.
   public <T> EinfacheKlasse (T parameter)
   {
      System.out.println ("Konstruktor:        " + parameter);
   }

   // Herkömmliche Methode
   public void einfacheMethode (String s)
   {
      System.out.println ("Einfache Methode:   " + s);
   }

   // Eigenständig generische Methode
   public <T> void generischeMethode (T para)
   {
      System.out.println ("Generische Methode: " + para);
   }

   public static void main (String[] args)
   {
      String stringRef = new String ("Ich bin ein String.");
      EinfacheKlasse ref = new EinfacheKlasse (stringRef);

      // Die herkömmliche Methode kann als aktuellen Parameter
      // nur Referenzen auf Objekte vom Typ String haben
      ref.einfacheMethode (stringRef);

      // Die generische Methode kann beispielsweise mit Referenzen
      // auf Objekte vom Typ String aufgerufen werden
      ref.generischeMethode (stringRef);

      // Oder die generische Methode wird mit Referenzen auf Objekte
      // von beliebigem Typ - hier z. B. Double - aufgerufen
      Double doubleRef = new Double (10.0);
      ref.generischeMethode (doubleRef);

      // Der Konstruktor kann auch mit einer Referenz auf ein
      // Objekt vom Typ java.util.Date aufgerufen werden.
      new EinfacheKlasse (new java.util.Date());
   }
}
```

Die Ausgabe des Programms ist:

```
Konstruktor:        Ich bin ein String.
Einfache Methode:   Ich bin ein String.
Generische Methode: Ich bin ein String.
Generische Methode: 10.0
Konstruktor:        Mon Sep 18 16:30:48 CEST 2006
```

Üblicherweise setzt man jedoch eigenständig generische Methoden bei Hilfsklassen ein, um ein und dieselbe Methode für verschiedene Typ-Parameter zu verwenden. Mit anderen Worten, ein Algorithmus soll unabhängig vom Datentyp der Objekte sein, auf denen er ausgeführt wird. Es soll hierzu nun die Klasse `PunktUtils` betrachtet werden. Diese Klasse definiert eine generische Klassenmethode `tausche()`, mit deren Hilfe die Koordinaten zweier Punkte vertauscht werden können. Es wird die Definition der Klasse `Punkt<T>` aus Kapitel 17.1.1 zugrunde gelegt:

```
// Datei: PunktUtils.java
class PunktUtils
{
   public static <T> void tausche (Punkt<T> p1,  Punkt<T> p2)
   {
      T x = p1.getX();
      T y = p1.getY();
      p1.setX (p2.getX());
      p1.setY (p2.getY());
      p2.setX (x);
      p2.setY (y);
   }
}
```

Die Klasse `TestPunktUtils` legt zwei Objekte vom Typ `Punkt<Float>` an und übergibt die Referenzen auf diese Objekte an die Methode `tausche()` der Klasse `PunktUtils`. Danach werden zwei Objekte vom Typ `Punkt<Integer>` angelegt, deren Referenzen ebenfalls an die Methode `tausche()` übergeben werden:

```
// Datei: TestPunktUtils.java
public class TestPunktUtils
{
   public static void main (String[] args)
   {
      Punkt<Float> floatPunkt1 = new Punkt<> (7.0f, 3.2f);
      Punkt<Float> floatPunkt2 = new Punkt<> (4.0f, 2.0f);

      System.out.println ("Erzeugte Float-Punkte:");
      System.out.println (floatPunkt1);
      System.out.println (floatPunkt2);

      // Vertauschen der Punkt-Koordinaten
      PunktUtils.tausche (floatPunkt1, floatPunkt2);
      System.out.println ("Nach dem Vertauschen:");
      System.out.println (floatPunkt1);
      System.out.println (floatPunkt2);
      Punkt<Integer> integerPunkt1 = new Punkt<> (4,2);
      Punkt<Integer> integerPunkt2 = new Punkt<> (1,9);

      System.out.println ("\nErzeugte Integer-Punkte:");
      System.out.println (integerPunkt1);
      System.out.println (integerPunkt2);

      // Vertauschen der Punkt-Koordinaten
      PunktUtils.tausche (integerPunkt1, integerPunkt2);
      System.out.println ("Nach dem Vertauschen:");
      System.out.println (integerPunkt1);
      System.out.println (integerPunkt2);
```

```
    }
}
```

Die Ausgabe des Programms ist:

```
Erzeugte Float-Punkte:
x = 7.0, y = 3.2
x = 4.0, y = 2.0
Nach dem Vertauschen:
x = 4.0, y = 2.0
x = 7.0, y = 3.2

Erzeugte Integer-Punkte:
x = 4, y = 2
x = 1, y = 9
Nach dem Vertauschen:
x = 1, y = 9
x = 4, y = 2
```

Der Versuch, die Methode `tausche()` mit Referenzen auf Objekte von verschiedenem Typ aufzurufen, wird vom Compiler mit einer Fehlermeldung abgelehnt. Das hat einen einfachen Grund:

Ein Typ-Parameter `T` steht für genau einen Typ. Genau ein Typ bedeutet, dass `T` nicht gleichzeitig zwei oder mehr Typen repräsentieren kann.

Daher ist der Aufruf von

```
PunktUtils.tausche (integerPunkt1, floatPunkt1);
```

nicht zulässig, weil der formale Typ-Parameter `T` der Klasse `Punkt<T>` hier gleichzeitig den Typ `Integer` und den Typ `Float` repräsentieren soll, was nicht geht!

Eigenständig generische Methoden mit einem formalen Typ-Parameter `T` können mit unterschiedlichen Datentypen aktuell parametrisiert werden, wodurch eine mehrfache Implementierung der Methode entfällt.

17.3 Bounded Typ-Parameter und Wildcards

Ein Typ-Parameter kann eingeschränkt werden durch den Einsatz eines sogenannten Bounded Typ-Parameters `T extends UpperBound` (siehe Kapitel 17.3.1) oder durch den Einsatz einer Wildcard. Es werden drei Arten von Wildcards unterschieden:

- die Unbounded Wildcard `?` (siehe Kapitel 17.3.2),
- die Upper Bound Wildcard `? extends UpperBound` (siehe Kapitel 17.3.3)
- und die Lower Bound Wildcard `? super LowerBound` (siehe Kapitel 17.3.4).

17.3.1 Der Bounded Typ-Parameter

Die Klasse `Punkt<T>` kann mit jedem beliebigen Typ-Parameter aktuell parametrisiert werden, also auch mit den Datentypen `Object` oder `String`. Das Anlegen eines Objektes wie im folgenden Beispiel:

```
Punkt<String> quatsch = new Punkt<>("eins", "zweidrei");
```

ist syntaktisch völlig korrekt, aber verfehlt den Sinn der Klasse `Punkt<T>`. Die Koordinaten eines Punktes sollen stets numerisch sein. Das obige Beispiel könnte dadurch verbessert werden, indem man erzwingt, dass der formale Typ-Parameter nur durch Referenztypen ersetzt werden kann, die Zahlen repräsentieren. Dies wird erreicht, indem als formaler Typ-Parameter ein **Bounded Typ-Parameter** `T extends UpperBound` verwendet wird:

```
public class Punkt<T extends Number>
{
   . . . . . // hier ändert sich nichts
}
```

Damit können nur noch Punkte angelegt werden, deren Koordinaten durch Objekte der Klasse `Number` oder deren Subklassen repräsentiert werden. Das Anlegen eines Objektes vom Typ `Punkt<String>` ist damit nicht mehr möglich und wird durch den Compiler unterbunden.

Bei der Angabe eines Bounded Typ-Parameters für einen formalen Typ-Parameter ersetzt der Compiler mit Hilfe der Type Erasure-Technik den formalen Typ-Parameter `T` durch den Typ, der mittels des Bounded Typ-Parameters angegeben wird. So werden bei der Übersetzung der Klasse

```
class Punkt<T extends Number>
```

alle Vorkommen von `T` durch `Number` ersetzt.

Die generische Klasse `Punkt<T extends Number>` darf also nur mit dem Typ `Number` oder einem von `Number` abgeleiteten Typ aktuell parametrisiert werden. Das Bild 17-5 zeigt einen Ausschnitt aus der Klassenhierarchie der Klasse `Number`:

Bild 17-5 Ausschnitt aus der Klassenhierarchie der Klasse Number

Mit einem Bounded Typ-Parameter kann der zulässige Wertebereich von Typ-Parametern auf einen Teilbaum einer Klassenhierarchie eingeschränkt werden.

Das folgende Beispiel zeigt die Verwendung des Bounded Typ-Parameters. Die Klasse `Punkt2` kann nun nur noch mit Parametern vom Typ `Number` oder dessen Subtypen aktuell parametrisiert werden. Dies zeigt das folgende **Programmbeispiel**:

```
// Datei: Punkt2.java

public class Punkt2<T extends Number>
{
   // Ein Punkt hat 2 Koordinaten vom Typ T
   private T x;
   private T y;

   // Der Konstruktor erwartet 2 Parameter vom Typ T
   public Punkt2 (T x, T y)
   {
      this.x = x;
      this.y = y;
   }

   // get- und set-Methoden für die Koordinaten
   public T getX()
   {
      return x;
   }

   public T getY()
   {
       return y;
   }

   public void setX (T x)
   {
      this.x = x;
   }


   public void setY (T y)
   {
      this.y = y;
   }

   public String toString()
   {
      return ("x = " + x + ", y = " + y);
   }
}
```

Die Klasse `TestPunkt4` legt nun Objekte der aktuell parametrisierten Klassen `Punkt2<Integer>`, `Punkt2<Double>` und `Punkt2<Number>` an:

```
// Datei: TestPunkt4.java

public class TestPunkt4
{
   public static void main (String[] args)
   {
      Punkt2<Integer> ip = new Punkt2<> (4, 2);
      System.out.println (ip);

      Punkt2<Double> dp = new Punkt2<> (1.0, 2.0);
      System.out.println (dp);

      Punkt2<Number> np = new Punkt2<Number> (5, 6);
      System.out.println (np);
   }
}
```

Die Ausgabe des Programms ist:

```
x = 4, y = 2
x = 1.0, y = 2.0
x = 5, y = 6
```

Obwohl die Klasse `Number` abstrakt ist, kann ein Objekt vom Typ `Punkt2<Number>` angelegt werden. Dies funktioniert aber nur deshalb, weil der Compiler die übergebenen `int`-Werte beim Konstruktoraufruf automatisch in ein Objekt vom Typ `Integer` verpackt. Der Versuch, ein Objekt vom Typ `Punkt2<Object>` anzulegen, wird vom Compiler allerdings abgelehnt, da die aktuellen Typ-Parameter vom Typ `Number` sein müssen.

Anbei noch ein Hinweis zu den Grenzen generischer Klassen: In der Klasse `Punkt2<T extends Number>` ist es nach wie vor nicht möglich, eine Methode `verschiebe()` zu implementieren:

```
public T verschiebe (T deltaX, T deltaY)
{
   x = x + deltaX; // geht nicht!
   y = y + deltaY; // geht nicht!
}
```

Durch die Type Erasure-Technik wird bei Einsatz des Bounded Typ-Parameters `T extends Number` zwar der formale Typ-Parameter `T` durch den Typ `Number` ersetzt, aber auch für den Typ `Number` ist der `+`-Operator nicht definiert.

17.3.2 Die Unbounded Wildcard ?

In den Kapiteln 17.1 und 17.1.4 wurde gezeigt, dass es für aktuell parametrisierte Klassen eines generischen Typs keine Vererbungshierarchie gibt, auch wenn zwischen den aktuellen Parametern eine Vererbungsbeziehung besteht. Als Beispiel

hierfür sei nochmals die Vererbungsbeziehung zwischen `Number` und `Integer` genannt. Obwohl `Integer` von `Number` ableitet, ist der Typ `Punkt<Number>` nicht der Basistyp von `Punkt<Integer>`. Damit kann eine Referenzvariable vom Typ `Punkt<Number>` nicht auf ein Objekt vom Typ `Punkt<Integer>` zeigen.

Um eine Referenzvariable definieren zu können, die auf Objekte beliebiger aktuell parametrisierter Klassen eines generischen Typs zeigen kann, wurde die Wildcard `?` eingeführt.

Die Referenzvariable

```
Punkt<?> ref;
```

kann auf Objekte aktuell parametrisierter Klassen des generischen Typs `Punkt<T>` zeigen. Das `?` wird auch als **Unbounded**[214] **Wildcard** bezeichnet, weil es keine Einschränkungen gibt, durch welchen konkreten Typ die Wildcard `?` ersetzt werden kann.

Die Wildcard `?` steht im Gegensatz zu einem formalen Typ-Parameter `T` nicht stellvertretend für genau einen Typ, sondern für alle möglichen Typen. **So kann eine Referenzvariable vom Typ `Punkt<?>` auf Objekte aller aktuell parametrisierten Klassen des generischen Typs `Punkt<T>` zeigen.**

Hier ein **Beispiel**:

```
Punkt<?> ref = new Punkt<Integer> (1, 2);
ref = new Punkt<Double> (2.0, 7.0);
```

Es ist nicht erlaubt, die Wildcard `?` in der Typ-Parameter-Sektion einer generischen Klasse, Schnittstelle oder Methode anzugeben. **Die Wildcard `?` darf nur beim Anlegen einer Referenzvariablen oder beim Anlegen eines Arrays aus Referenzen auf Objekte generischer Typen verwendet werden.**

Im folgenden **Beispielprogramm** `TestPunkt3` wird die Verwendung der Unbounded Wildcard `?` gezeigt. Es wird dort ein Array-Objekt vom Typ `Punkt<?>` angelegt, in dem Referenzen auf Objekte aktuell parametrisierter Klassen des generischen Typs `Punkt<T>` hinterlegt werden können, wobei der formale Typ-Parameter `T` bei der Erzeugung der Objekte durch beliebige aktuelle Typ-Parameter ersetzt wird. Danach werden die im Array enthaltenen Referenzen in einer `for`-Schleife ausgelesen und einer Referenzvariablen vom Typ `Punkt<?>` zugewiesen. Es wird wiederum die Definition der Klasse `Punkt<T>` aus Kapitel 17.1.1 zugrunde gelegt.

[214] Engl. für unbegrenzt.

Hier das Programm:

```
// Datei: TestPunkt3.java

public class TestPunkt3
{
   public static void main (String[] args)
   {
      // Anlegen dreier Punkt<T>-Objekte, je eines mit dem
      // aktuellen Parameter Integer, Double und Float.
      Punkt<Integer> ip = new Punkt<> (4, 2);
      Punkt<Double> dp = new Punkt<> (1.0, 2.0);
      Punkt<Float> fp = new Punkt<> (7.0f, 3.2f);

      // Anlegen eines Arrays aus Referenzen auf Objekte vom Typ
      // Punkt<?>. Die Verwendung der Wildcard ist dabei zwingend
      // vorgeschrieben.
      Punkt<?>[] arr = new Punkt<?> [3];

      // Füllen des Arrays
      arr [0] = ip;
      arr [1] = dp;
      arr [2] = fp;

      System.out.println ("\nInhalt des Arrays ist:");
      for (Punkt<?> element : arr)
      {
         System.out.println (element);
      }
   }
}
```

Die Ausgabe des Programms ist:

```
Inhalt des Arrays ist:
x = 4, y = 2
x = 1.0, y = 2.0
x = 7.0, y = 3.2
```

Das Beispiel verdeutlicht zudem die in Kapitel 17.1.5 aufgeführte Restriktion für das Anlegen von Arrays generischer Typen. Diese Einschränkung besagt, dass beim Anlegen eines Arrays weder ein formaler noch ein aktueller Typ-Parameter vorkommen darf. So ist z. B. die folgende Anweisung falsch:

```
Punkt<Integer>[] fehler = new Punkt<Integer>[3]; // Fehler!
```

Beim Erzeugen eines Arrays, das Referenzen auf Objekte aktuell parametrisierter Klassen eines generischen Typs enthalten soll, muss der Typ-Parameter durch die Unbounded Wildcard `?` ersetzt werden:

```
Punkt<?>[] richtig = new Punkt<?> [3]; // OK!
```

17.3.3 Die Upper Bound Wildcard

Die Upper Bound Wildcard `? extends UpperBound` stellt sicher, dass nur Klassen einer Klassenhierarchie als Typ-Parameter verwendet werden können, die von einer bestimmten Klasse abgeleitet sind. Auch die Basisklasse selbst kann verwendet werden.

Die Verwendung einer Upper Bound Wildcard soll mit Hilfe der Klasse `Hilfsklasse` und einer Klassenmethode `methode()` erklärt werden. Die Klassenmethode `methode()` hat einen formalen Übergabeparameter vom Typ der generischen Klasse `GenKlasse<T>` aus den vorhergehenden Kapiteln. Ferner soll die Klassenhierarchie aus Bild 17-6 als Betrachtungsgrundlage dienen.

Der Typ-Parameter `T` kann mit Hilfe der Upper Bound Wildcard auf einen Typ `A` selbst und auf Subtypen der Klasse `A` einschränkt werden.

```
public class Hilfsklasse
{
   public static void methode (GenKlasse<? extends A> ref)
   {
      // Nicht im Fokus.
   }
}
```

Die Klassenmethode `methode()` akzeptiert damit Referenzen auf Objekte vom Typ `GenKlasse<T>`, wobei für den formalen Typ-Parameter `T` entweder der Typ `A` oder Subtypen von `A` erlaubt sind. Es werden also Referenzen auf Objekte vom Typ der grau hinterlegten Klassen aus Bild 17-6 akzeptiert. Hier das Bild 17-6:

Bild 17-6 Klassenhierarchie der erlaubten Typen

Die Upper Bound Wildcard `<? extends UpperBound>` legt eine obere Schranke für einen formalen Typ-Parameter fest.

Auf die Referenz `ref` des Übergabeparameters kann innerhalb der Methode `methode()` nur lesend zugegriffen werden!

Die Upper Bound Wildcard darf nicht in der Typ-Parameter-Sektion einer generischen Klasse verwendet werden.

17.3.4 Die Lower Bound Wildcard

Die Lower Bound[215] Wildcard `? super LowerBound` kann nur in Zusammenhang mit der Unbounded Wildcard `?` eingesetzt werden.

Die Lower Bound Wildcard dient dazu, den Wertebereich von zulässigen aktuellen Typ-Parametern in der Klassenhierarchie nach unten einzuschränken.

Das folgende Beispiel verdeutlicht den Zusammenhang.

Es soll die Definition einer generischen Klasse `GenKlasse<T>` betrachtet werden:

```
public class GenKlasse<T>
{
   // Nicht von Interesse
}
```

Weiterhin seien die gewöhnlichen Klassen `A`, `B`, `C` und `D` definiert, wobei die Vererbungshierarchie aus Bild 17-7 zugrunde gelegt wird:

Bild 17-7 Zugrunde gelegte Klassenhierarchie

[215] Engl. für untere Grenze.

In einer weiteren Klasse `Hilfsklasse` sei nun eine Klassenmethode `methode()` definiert, die einen formalen Übergabeparameter vom Typ der generischen Klasse `GenKlasse<T>` hat:

```
public class Hilfsklasse
{
   public static void methode (GenKlasse<? super C> ref)
   {
      // Nicht von Interesse
   }
}
```

Als aktuelle Parameter der Methode `methode()` kommen nur Referenzen vom Typ `GenKlasse<Parameter>` in Frage, wobei der aktuelle Typ-Parameter `Parameter` vom Typ `C` oder einem Supertyp von `C` ist. Somit können nur die in Bild 17-7 grau eingefärbten Typen – also `C`, `A` und `Object` – als aktueller Typ-Parameter verwendet werden. Bei allen anderen Typen gibt der Compiler eine Fehlermeldung aus.

Die Lower Bound Wildcard `<? super C>` deckt die Klasse `C` und alle Klassen ab, von denen `C` direkt oder indirekt abgeleitet ist. Es wird damit eine untere Schranke für den Wertebereich eines formalen Typ-Parameters definiert.

Anzumerken ist, dass auf Lower Bound Wildcards auch schreibend zugegriffen werden kann.

Im Folgenden einige zulässige und unzulässige Aufrufe von `methode()`:

```
Hilfsklasse.methode (new GenKlasse<C>());       // OK!
Hilfsklasse.methode (new GenKlasse<A>());       // OK!
Hilfsklasse.methode (new GenKlasse<D>());       // Fehler!
Hilfsklasse.methode (new GenKlasse<Object>()); // OK!
Hilfsklasse.methode (new GenKlasse<B>());       // Fehler!
```

Die Lower Bound Wildcard darf nicht in der Typ-Parameter-Sektion einer generischen Klasse verwendet werden.

Es ist deshalb unzulässig, eine generische Klasse wie folgt zu definieren:

```
public class GenKlasse<? super C>
{
   . . . . .
}
```

Die Lower Bound Wildcard darf deshalb wie die Unbounded Wildcard nur bei der Definition von Referenzvariablen verwendet werden.

17.4 Generische Schnittstellen

Genauso wie Klassen können auch Schnittstellen generisch sein. Eine allgemeingültige Notation für die Definition generischer Schnittstellen lautet:

```
interface GenerischeSchnittstelle<T1, . . . , TN>
   (extends I1, . . . , IN)opt
```

Die optionalen Parameter `I1, . . . , IN` sind Schnittstellen.

Beispielsweise ist die Schnittstelle `GenSchnittstelle` generisch:

```
// Datei: GenSchnittstelle.java

public interface GenSchnittstelle<T>
{
   public void methode1 (T param);
   public void methode2();
}
```

Wie zu erkennen ist, wird der formale Typ-Parameter `T` innerhalb der generischen Schnittstelle genauso verwendet wie innerhalb einer Klassendefinition.

Implementiert nun eine Klasse eine generische Schnittstelle, so gibt es dafür zwei Möglichkeiten:

- Die Klasse implementiert die Schnittstelle, wobei die in der Typ-Parameter-Sektion der Schnittstelle definierten **formalen Typ-Parameter durch aktuelle Typ-Parameter ersetzt** werden müssen. Damit werden auch die formalen Typ-Parameter in den Methodenköpfen der Schnittstelle durch die aktuellen Typ-Parameter ersetzt.
- Die zweite Möglichkeit besteht darin, die Schnittstelle zu implementieren, aber **ohne die formalen Typ-Parameter durch aktuelle Typ-Parameter** zu ersetzen. Damit behalten die Methodenköpfe der Schnittstelle ihre formalen Typ-Parameter bei und die Klasse, welche die Schnittstelle implementiert, wird automatisch generisch. Daher muss die Klasse die formalen Typ-Parameter in ihrer Typ-Parameter-Sektion dem Compiler bekannt machen.

Beide Möglichkeiten werden im Folgenden genauer betrachtet.

17.4.1 Generische Schnittstellen mit Ersetzung der formalen Typ-Parameter

In diesem Fall müssen alle formalen Typ-Parameter, welche für die Schnittstelle definiert sind, durch aktuelle Typ-Parameter ersetzt werden. Hierfür wird das **Beispiel** der oben deklarierten Schnittstelle `GenSchnittstelle<T>` mit einem formalen Typ-Parameter `T` betrachtet. Eine Klasse `Test` implementiert nun diese Schnittstelle mit dem aktuellen Typ-Parameter `Integer`, indem in der Klassendeklaration der Zusatz

```
implements GenSchnittstelle<Integer>
```

angefügt wird. Die Klasse muss nun alle Methodenköpfe, welche die Schnittstelle enthält – hier also den Methodenkopf von `methode1()` – mit dem aktuellen Typ-Parameter `Integer` implementieren:

```
// Datei: Test.java

public class Test implements GenSchnittstelle<Integer>
{
   private Integer ref;

   public Test (Integer i)
   {
      ref = i;
   }

   public void methode1 (Integer param)
   {
      System.out.println ("Aufruf von methode1() mit: " + param);
   }

   public void methode2()
   {
      System.out.println ("Inhalt von ref: " + ref);
   }

   public static void main (String[] args)
   {
      // Die Klasse Test wird wie eine
      // herkömmliche Java-Klasse instanziiert.
      Test test = new Test (5);

      test.methode1 (new Integer (1));
      test.methode2();

      // Es besteht auch die Möglichkeit, eine Referenz-
      // variable vom Typ GenSchnittstelle<Integer> anzulegen
      // und dieser eine Referenz vom Typ Test zuzuweisen.
      GenSchnittstelle<Integer> ref = new Test (1);
      ref.methode1 (new Integer (5));
   }
}
```

Die Ausgabe des Programms ist:

```
Aufruf von methode1() mit: 1
Inhalt von ref: 5
Aufruf von methode1() mit: 5
```

Wie dem obigen Beispiel zu entnehmen ist, wird die **Klasse `Test` wie eine herkömmliche Klasse behandelt**. Das liegt daran, dass sie keine generische, sondern eine mit dem Typ-Parameter `Integer` aktuell parametrisierte Schnittstelle implementiert.

Implementiert eine Klasse eine generische Schnittstelle und ersetzt bei der Implementierung alle in der Typ-Parameter-Sektion der Schnittstelle auftretenden formalen Typ-Parameter durch aktuelle Typ-Parameter, so besitzt die **Klasse keine generischen Eigenschaften mehr**.

17.4.2 Ohne Ersetzung der formalen Typ-Parameter

Implementiert eine Klasse eine generische Schnittstelle, ohne dass die formalen Typ-Parameter durch aktuelle Typ-Parameter ersetzt werden, so wird die Klasse selbst generisch. Mit anderen Worten, die Klasse implementiert alle Methoden, welche die generische Schnittstelle deklariert, und verwendet bei der Implementierung den formalen Typ-Parameter.

Der Compiler erzwingt, dass die Klasse, welche die generische Schnittstelle ohne die Ersetzung der formalen Typ-Parameter implementiert, selbst generisch wird und in ihrer Typ-Parameter-Sektion die verwendeten formalen Typ-Parameter bekannt macht.

Der Methodenkopf der zu implementierenden Methode muss mit der Deklaration in der Schnittstelle übereinstimmen. Innerhalb des Klassenrumpfes können außerdem die in der Typ-Parameter-Sektion deklarierten formalen Typ-Parameter nach Belieben verwendet werden.

Im Folgenden wird das obige **Beispiel** modifiziert. Die Klasse `Test2` implementiert nun die generische Schnittstelle `GenSchnittstelle<T>`, ohne jedoch den formalen Typ-Parameter `T` durch einen aktuellen Typ-Parameter zu ersetzen. Dadurch wird die Klasse `Test2` selbst generisch und innerhalb ihres Klassenrumpfes kann der formale Typ-Parameter `T` – dieser wird ja durch ihre Typ-Parameter-Sektion bekannt gemacht – beliebig verwendet werden:

```
// Datei: Test2.java
// Die Klasse Test2 macht durch ihre Typ-Parameter-Sektion den
// formalen Typ-Parameter T dem Compiler bekannt.

public class Test2<T> implements GenSchnittstelle<T>
{
   // Definition einer generischen Referenzvariablen
   private T formalTypRef;

   public Test2 (T t)
   {
      formalTypRef = t;
   }

   public void methode1 (T param)
   {
      System.out.println ("Aufruf von methode1() mit: " + param);
   }

   public void methode2()
   {
```

```
      System.out.println ("Inhalt von formalTypRef: "
                          + formalTypRef);
   }

   public static void main (String[] args)
   {
      // Instanziierung mit aktuellem Typ-Parameter Integer
      Test2<Integer> testRef1 = new Test2<> (5);
      testRef1.methode1 (new Integer (1));
      testRef1.methode2();
      // Instanziierung mit aktuellem Typ-Parameter String
      Test2<String> testRef2 =
         new Test2<> ("String-Objekt");
      testRef2.methode1 ("Ich bin ein String");
      testRef2.methode2();
   }
}
```

Die Ausgabe des Programms ist:

```
Aufruf von methode1() mit: 1
Inhalt von formalTypRef: 5
Aufruf von methode1() mit: Ich bin ein String
Inhalt von formalTypRef: String-Objekt
```

17.4.3 Schnittstellen und Bounds

Im Kapitel 17.3.1 wurde der Bounded Typ-Parameter `T extends UpperBound` eingeführt. Dabei wurde der Platzhalter `UpperBound` durch eine konkrete Klasse ersetzt, um den Gültigkeitsbereich eines formalen Typ-Parameters `T` nach oben zu beschränken. Wird eine generische Klasse beispielsweise mit folgender Typ-Parameter-Sektion deklariert:

```
public class GenKlasse<T extends Number>
```

so kann die Klasse `GenKlasse` nur mit dem aktuellen Typ-Parameter `Number` oder dessen Subklassen aktuell parametrisiert werden. Die Klasse `Number` stellt damit die obere Schranke des Gültigkeitsbereiches des formalen Typ-Parameters `T` dar. Die Klasse `Number` ist damit die Upper Bound des Typ-Parameters `T`.

Die Definition einer **Bound für einen formalen Typ-Parameter** kann nun um die Verwendung von **generischen Schnittstellen** erweitert werden. In den bisherigen Beispielen wurde dabei die Upper Bound nur durch eine Klasse repräsentiert, so z. B.

```
T extends Number    // Number ist hier die Upper Bound
```

Man sagt auch, dass der formale Typ-Parameter `T` an die Klasse `Number` gebunden ist. Dies bedeutet, dass als aktueller Typ-Parameter nur die Klasse `Number` selbst oder eine von `Number` abgeleitete Klasse eingesetzt werden kann. Ein formaler Typ-Parameter kann zusätzlich an beliebig viele Schnittstellen-Typen gebunden werden.

Eine **Bound** kann **aus einer einzigen Klasse `Klasse` und beliebig vielen Schnittstellen** `I1` bis `IN` bestehen. Die Verknüpfung der einzelnen Bounds findet über den logischen UND-Operator `&` statt.

Die Deklaration einer solchen Bound aus einer einzigen Klasse `Klasse` und beliebig vielen Schnittstellen `I1` bis `IN` für einen formalen Typ-Parameter `T` lautet:

```
T extends Klasse & I1 & I2 & . . . & IN
```

Diese Notation hat zur Konsequenz, dass der aktuelle Typ-Parameter, der den formalen Typ-Parameter `T` ersetzt,

- von der Klasse `Klasse` ableiten muss
- und ferner alle Schnittstellen `I1` bis `IN` implementieren muss,

damit der Compiler die Ersetzung des formalen Typ-Parameters durch den aktuellen Typ-Parameter zulässt.

Es darf nur eine einzige Klasse bei der Upper Bound angegeben werden. Dies erscheint auch logisch, denn sonst müsste eine Klasse, welche an die Stelle des formalen Typ-Parameters treten soll, von mehr als einer Klasse ableiten. Und bekanntermaßen wird Mehrfachvererbung von Klassen – also das Ableiten von mehr als einer Klasse – in Java nicht unterstützt. Dahingegen ist die Anzahl der Schnittstellen, welche die Bound bilden, nicht begrenzt, weil eine Klasse eine beliebige Anzahl von Schnittstellen implementieren kann.

Es ist nicht zwingend notwendig, dass die Bound durch eine Klasse und eine beliebige Anzahl von Schnittstellen gebildet wird. Sie kann auch nur aus Schnittstellen bestehen. Dabei muss bei einer **Bound** stets das Schlüsselwort **`extends` verwendet** werden. Angenommen, `I1` und `I2` sind zwei Schnittstellen und `Klasse` eine Klasse, dann sind folgende Bounds gültig:

- `T extends Klasse`
- `T extends I1` bzw. `T extends I2`
- `T extends I1 & I2`
- `T extends Klasse & I1` bzw. `T extends Klasse & I2`
- `T extends Klasse & I1 & I2`

Das folgende **Beispiel** soll den Zusammenhang nochmals verdeutlichen. Es sind dort die beiden Schnittstellen `I1` und `I2` und die vier Klassen `A`, `B`, `C` und `D` definiert. Die zugrunde gelegte Hierarchie wird im folgenden Bild gezeigt:

Bild 17-8 Zugrunde gelegte Vererbungs- und Realisierungsbeziehungen

Weiterhin existiert eine generische Klasse `UpperBound`, die einen formalen Typ-Parameter `T` mit der Bound

```
T extends A & I1 & I2
```

besitzt. In der Testklasse `UpperBoundTest` werden dann die möglichen Instanziierungsversuche der Klasse `UpperBound` durchgespielt, um zu erläutern, wie sich die für den Typ-Parameter `T` definierte Upper Bound auf die Instanziierung auswirkt:

```
// Datei: UpperBoundTest.java

interface I1 {/* hier nicht interessant */}

interface I2 {/* hier nicht interessant */}

class A {/* hier nicht interessant */}

class B extends A {/* hier nicht interessant */}

class C implements I1, I2 {/* hier nicht interessant */}

class D extends B implements I1, I2 {/* hier nicht interessant */}

class UpperBound<T extends A & I1 & I2>
{
   public UpperBound (T param)
   {
      // Die Methode getClass() ist in der Klasse Object definiert
      // und gibt eine Referenz auf ein Objekt vom Typ Class<T>
      // zurück. Über dieses Objekt kann mit Hilfe der Methode
      // getName() der Klassenname des Objektes erfragt werden.
      String name = param.getClass().getName();
      System.out.println ("Mit aktuellem Typ-Parameter " + name +
                          " instanziiert!");
   }
}

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

```
        // Klasse A implementiert nicht die Schnittstellen I1 und I2
        // new UpperBound<A> (new A());

        // Klasse B leitet von A ab, implementiert
        // aber nicht die geforderten Schnittstellen
        // new UpperBound<B> (new B());

        // Klasse C implementiert die Schnittstellen
        // leitet aber nicht von A ab.
        // new UpperBound<C> (new C());

        // Die Klasse D leitet direkt von der Klasse B und damit
        // indirekt von der Klasse A ab. Sie implementiert zusätzlich
        // die geforderten Schnittstellen I1 und I1.
        new UpperBound<D> (new D());
    }
}
```

Die Ausgabe des Programms ist:

```
Mit aktuellem Typ-Parameter D instanziiert!
```

Durch die Type Erasure werden beim Übersetzen einer generischen Klasse die formalen Typ-Parameter, die eine Upper Bound besitzen, durch den **ersten** Typ der Upper Bound ersetzt. So werden bei

```
class GenKlasse<T extends Klasse & I1>
```

alle Vorkommen von `T` durch `Klasse` und bei

```
class GenKlasse<T extends I1 & I2>
```

alle Vorkommen von `T` durch `I1` ersetzt.

17.4.4 Vererbungsbeziehung zwischen generischen Schnittstellen

Generische Schnittstellen können – wie herkömmliche Schnittstellen auch – Vererbungshierarchien aufbauen. Dabei gelten die gleichen Regeln, wie sie schon bei der Beschreibung von Vererbungsbeziehungen bei generischen Klassen aufgestellt wurden (siehe Kapitel 17.1.6). Es werden somit nur die Kernpunkte aufgelistet, die beim Aufbau von Vererbungshierarchien bei generischen Schnittstellen zu beachten sind:

- Wird eine **generische Schnittstelle** von einer anderen **generischen Schnittstelle** abgeleitet, so muss der formale Typ-Parameter der Schnittstelle, von der abgeleitet wird, entweder durch einen aktuellen Typ-Parameter

  ```
  GenInterface2<T> extends GenInterface1<AktuellerTyp>
  ```

  oder durch den formalen Typ-Parameter der abgeleiteten Schnittstelle

  ```
  GenInterface2<T> extends GenInterface1<T>
  ```

  ersetzt werden.

- Im Folgenden wird der Fall betrachtet, dass eine **generische Schnittstelle** von einer **nicht generischen Schnittstelle** abgeleitet wird:

```
interface I1
{
   public void methode1();
}

interface I2<T> extends I1
{
   public void methode2 (T t);
}
```

  Eine Klasse, welche die Schnittstelle `I2` implementiert, muss dann eine Implementierung der Methoden `methode1()` und `methode2()` bereitstellen. Der formale Typ-Parameter der generischen Schnittstelle `I2` kann bei der Implementierung durch einen aktuellen (siehe Kapitel 17.4.1) oder einen formalen Typ-Parameter (siehe Kapitel 17.4.2) ersetzt werden.

- Leitet eine **herkömmliche Schnittstelle** von einer **generischen Schnittstelle** ab, so muss bei der Ableitung der formale Typ-Parameter durch einen aktuellen Typ-Parameter ersetzt werden:

```
interface I2 extends I1<KonkreteKlasse>
{
   // Hier nicht interessant
}
```

  Die generischen Eigenschaften der Schnittstelle `I1` gehen hier verloren.

Wie auch bei der Ableitung von generischen Klassen muss bei der Ersetzung der formalen Typ-Parameter auf die Bounds – das heißt auf die für die aktuellen Typ-Parameter definierten Schranken – geachtet werden. Die Ersetzung der formalen Typ-Parameter durch aktuelle Typ-Parameter muss somit natürlich innerhalb der zulässigen Schranken erfolgen.

17.4.5 Die generische Schnittstelle `Comparable`

Als Beispiel für eine generische Schnittstelle aus der Java-Klassenbibliothek soll die Schnittstelle `Comparable<T>` aus dem Paket `java.lang` betrachtet werden. Diese Schnittstelle kann von allen Klassen implementiert werden und ermöglicht dann, dass Arrays, bestehend aus Referenzen auf Objekte dieser Klassen, durch die Klassenmethode

```
public static void sort (Object[] a)
```

der Klasse `Arrays` aus dem Paket `java.util` sortiert werden können. Die Implementierung der generischen Schnittstelle `Comparable<T>` stellt sicher, dass alle Objekte eine bestimmte Vergleichsoperation unterstützen.

Die generische Schnittstelle `Comparable<T>` hat den folgenden Aufbau:

```
public interface Comparable<T>
{
   public int compareTo (T o);
}
```

Sortieren setzt voraus, dass Elemente, die sortiert werden sollen, ein Merkmal haben, für das eine **Ordnung** besteht. Eine Ordnung liegt dann vor, wenn für ein Merkmal die Vergleichsoperationen ==, !=, <, <=, >= und > existieren. Bei **Objekten** geht es letztendlich immer darum, wie Objekte miteinander verglichen werden und welche Datenfelder für eine größer-, kleiner- und gleich-Entscheidung herangezogen werden müssen. Werden **Strings** miteinander verglichen, so wird zumeist ein **lexikalischer Vergleich** herangezogen, sollen jedoch beliebige Objekte miteinander verglichen werden, so stellt sich die Frage, **nach welchen Datenfeldern** die Objekte zu **vergleichen** sind.

Die Schnittstelle `Comparable<T>` wird bereits von allen Wrapper-Klassen wie `Byte`, `Character`, `Double`, `Integer` usw. implementiert. Damit können also Arrays aus Referenzen auf Objekte dieser Klassen schon automatisch mit der oben genannten `sort()`-Methode sortiert werden.

Hierzu ein **Beispielprogramm** zur Sortierung eines `Integer`-Arrays:

```
// Datei: IntegerSort.java

import java.util.Arrays;

public class IntegerSort
{
   public static void main (String[] args)
   {
      Integer[] arr = new Integer [4];
      arr [0] = new Integer (7);
      arr [1] = new Integer (3);
      arr [2] = new Integer (5);
      arr [3] = new Integer (1);

      System.out.println ("Vor Sortierung:");
      System.out.println (arr [0] + "\n" + arr [1] + "\n" + arr [2]
                          + "\n" + arr [3]);

      Arrays.sort (arr);

      System.out.println ("Nach Sortierung:");
      System.out.println (arr [0] + "\n" + arr [1] + "\n" + arr [2]
                          + "\n" + arr [3]);
   }
}
```

Die Ausgabe des Programms ist:

```
Vor Sortierung:
7
3
5
1
Nach Sortierung:
1
3
5
7
```

Damit die Methode `sort()` der Klasse `Arrays` ein übergebenes Array sortieren kann, muss das Array als Elemente Referenzen auf Objekte enthalten, deren Klassen die Schnittstelle `Comparable<T>` implementieren. Wie schon erwähnt, implementiert die Klasse `Integer` die Schnittstelle `Comparable<T>`. Die Deklaration der Klasse `Integer` hat folgendes Aussehen:

```
public final class Integer extends Number
                           implements Comparable<Integer>
```

Dadurch, dass die Schnittstelle `Comparable` mit dem aktuellen Typ-Parameter `Integer` implementiert wird, kann zur Übersetzungszeit vom Compiler sichergestellt werden, dass die Implementierung der Methode `compareTo()` für die Klasse `Integer` auch nur Objekte vom Typ `Integer` vergleichen kann. Der folgende Codeausschnitt zeigt die Implementierung der Methode `compareTo()` für die Klasse `Integer`:

```
public int compareTo (Integer anotherInteger)
{
   int thisVal = this.value;
   // this bezieht sich auf das Objekt für
   // das die Methode compareTo() aufgerufen wird
   int anotherVal = anotherInteger.value;
   return (thisVal < anotherVal ? -1 :
             (thisVal == anotherVal ? 0 : 1));
}
```

Die Methode `public int compareTo (Integer o)` hat die Aufgabe, das aktuelle Objekt, für das die Methode aufgerufen wird, mit dem Objekt, dessen Referenz übergeben wurde, zu vergleichen. Dabei sind folgende Rückgabewerte zu erwarten:

- Ist das Objekt, dessen `compareTo()`-Methode aufgerufen wird, kleiner als das übergebene, so wird `-1` zurückgegeben.
- Ist das Objekt, dessen `compareTo()`-Methode aufgerufen wird, gleich mit dem übergebenen, so wird `0` zurückgegeben.
- Ist das Objekt, dessen `compareTo()`-Methode aufgerufen wird, größer als das übergebene, so wird `1` zurückgegeben.

Sollen Objekte selbst geschriebener Klassen von der `sort()`-Methode der Klasse `Arrays` sortiert werden, so müssen sie lediglich die Schnittstelle `Comparable<T>` implementieren und in der Methode `compareTo()` die entsprechenden Rückgabewerte liefern.

Dies soll an nachfolgendem Beispiel gezeigt werden. Die Klasse `Artikel` hat die Datenfelder `artNr` und `name`. Die Instanzen der Klasse `Artikel` sollen nach dem Datenfeld `name` in lexikographischer Ordnung sortiert werden, wobei Groß- und Kleinschreibung zu ignorieren ist. Damit ein Array von Artikeln sortiert werden kann, muss lediglich die Klasse `Artikel` die Schnittstelle `Comparable<Artikel>` implementieren, wie im folgenden Programm zu sehen ist.

```
// Datei: Vergleich.java

import java.util.*;

class Artikel implements Comparable<Artikel>
{
   private String artNr;
   private String name;

   public Artikel (String artNr, String name)
   {
      this.artNr = artNr;
      this.name = name;
   }


   public int compareTo (Artikel artikel)
   {
      // Die Methode compareToIgnoreCase() ist in der Klasse
      // String implementiert und liefert den entsprechenden
      // Wert -1, 0, 1 bei Vergleich beider Strings
      return name.compareToIgnoreCase (artikel.name);
   }

   // Überschreiben der toString()-Methode, um die String-
   // repräsentation des Objektes ausgeben zu können.
   public String toString()
   {
      return (artNr + "  " + name);
   }
}

public class Vergleich
{
   public static void main (String[] args)
   {
      Artikel[] arr = new Artikel [4];
      arr [0] = new Artikel ("1000", "Mutter 8x10");
      arr [1] = new Artikel ("1001", "Dichtungsring 20x100");
      arr [2] = new Artikel ("1002", "Abstreifring 25x125");
      arr [3] = new Artikel ("1003", "montierbarer Zackenring");
```

```
        Arrays.sort (arr);
        for (Artikel artikel : arr)
           System.out.println (artikel);
    }
}
```

Die Ausgabe des Programms ist:

```
1002  Abstreifring 25x125
1001  Dichtungsring 20x100
1003  montierbarer Zackenring
1000  Mutter 8x10
```

Sollen Objekte der Klasse `Artikel` nach der Artikelnummer sortiert werden, so ist die Methode `compareTo()` entsprechend umzuschreiben. Dabei muss natürlich beachtet werden, dass die Artikelnummern numerisch zu sortieren sind.

Zum Abschluss dieses Kapitels soll nochmals ein Hinweis zur Typsicherheit gemacht werden. Würde in der Methode `public static void sort (Object[] a)` der Klasse `Arrays` kein `Object[]`-Array als Parameter verlangt, sondern ein Array des Schnittstellentyps `Comparable<T>`, so könnten Fehler bezüglich der Übergabe eines Typs, der die Schnittstelle `Comparable<T>` gar nicht implementiert, schon zur Kompilierzeit erkannt werden.

17.5 Die Klasse `Class<T>`

Im Paket `java.lang` gibt es eine Klasse mit dem Namen `Class<T>`. Diese generische Klasse ist eine besondere Klasse, denn es existieren zu jedem Java-Programm eine Vielzahl von Instanzen dieser Klasse, ohne dass vom Programmierer davon auch nur eine einzige mit dem `new`-Operator erzeugt wird. Jedes Mal, wenn von der Java-Virtuellen Maschine eine Klasse geladen wird, – eine Klasse wird immer dann geladen, wenn sie zum ersten Mal benutzt wird – wird ein Objekt der Klasse `Class<T>` angelegt. Ein Objekt dieser Klasse repräsentiert alle Eigenschaften der entsprechenden Klasse. Mit Hilfe der Klasse `Class<T>` werden in Java Mechanismen wie Reflection[216] oder Objektserialisierung (siehe Kapitel 19) umgesetzt.

Der Programmierer selbst kann kein Objekt der Klasse `Class<T>` mit dem `new`-Operator erzeugen, da die Klasse `Class<T>` keinen öffentlichen Konstruktor zur Verfügung stellt.

Jeder **Typ** in einem Java-Programm wird in der Java-Virtuellen Maschine **durch** ein **Objekt** der Klasse `Class<T>` **repräsentiert**. Dieses Objekt repräsentiert alle Eigenschaften des entsprechenden Typs.

[216] Reflection wird in diesem Einsteigerbuch nicht behandelt.

Damit existiert für jede benutzte **Schnittstelle**, für jede benutzte **Klasse**, für jedes benutzte **Array**, für jeden Aufzählungstyp und sogar für jeden benutzten **elementaren Datentyp genau ein Objekt der Klasse** `Class<T>` in der Java-Virtuellen Maschine.

Durch Aufruf der Instanzmethode `getClass()` der Klasse `Object` erhält man eine Referenz auf das zugehörige Objekt der Klasse `Class<T>`. Das folgende **Beispielprogramm** zeigt, wie eine Referenz auf das `Class<T>`-Objekt der Klasse `Object` mit Hilfe der Instanzmethode `getClass()` erhalten werden kann:

```
// Datei: Typen.java
public class Typen
{
   public static void main (String[] args)
   {
      Object refObj = new Object();

      // Es muss die Unbounded Wildcard verwendet werden, weil
      // erst zur Laufzeit bekannt ist, von welchem Typ die
      // zurückgelieferte Referenz ist.
      Class<?> refClass = refObj.getClass();

      // Den Namen der Klasse ausgeben
      System.out.println ("Name: " + refClass.getName());
   }
}
```

Die Ausgabe des Programms ist:

```
Name: java.lang.Object
```

17.5.1 Erzeugen eines Objektes mit `newInstance()`

Über ein `Class<T>`-Objekt kann auch eine Instanz des Typs erzeugt werden, den das `Class<T>`-Objekt repräsentiert. Voraussetzung ist, dass es sich um einen instanziierbaren Typ[217] handelt. Das folgende **Programmbeispiel** besorgt sich zuerst eine Referenz auf das `Class<T>`-Objekt der Klasse `Typen2` mit Hilfe der Klassenmethode **`forName()`** der Klasse `Class<T>` und erzeugt dann durch Aufruf der Methode `newInstance()` ein Objekt der Klasse `Typen2`, ohne den `new`-Operator zu verwenden.

Hier das Programm:

```
// Datei: Typen2.java
interface Schnittstelle
{/* hier nicht interessant */}

public class Typen2
{
```

[217] Das heißt, es darf sich weder um eine Schnittstelle noch um eine abstrakte Klasse handeln.

```
   private int x;

   public Typen2()
   {
     x = 3;
   }

   public static void main (String[] args) throws Exception
   {
      Class<?> typen2ClassRef = Class.forName ("Typen2");

      // Da die Methode newInstance() eine Referenz vom Typ Object
      // zurückgibt, muss auf den Typ Typen2 gecastet werden.
      Typen2 typen2 = (Typen2) typen2ClassRef.newInstance();
      System.out.println ("Wert der Variablen x: " + typen2.x);

      // Eine Referenz auf das Class<T>-Objekt
      // von Schnittstelle wird besorgt.
      Class<?> schnittstelleClassRef =
         Class.forName ("Schnittstelle");

      // Der Versuch, über das referenzierte Class<T>-Objekt
      // eine Instanz vom Schnittstellen-Typ zu erzeugen, wird das
      // Werfen einer Exception vom Typ InstantiationException
      // hervorrufen.
      // Schnittstelle schnittstelle =
      //    (Schnittstelle) schnittstelleClassRef.newInstance();
   }
}
```

Die Ausgabe des Programms ist:

```
Wert der Variablen x: 3
```

Wie man an der Ausgabe erkennen kann, wird der selbst geschriebene parameterlose Konstruktor auch hier aufgerufen! Es wird allerdings immer vorausgesetzt, dass ein parameterloser Konstruktor vorhanden ist, ansonsten wird zur Laufzeit eine Fehlermeldung ausgegeben. Dieser Fall kann eintreten, wenn in einer Klasse nur ein Konstruktor mit Parametern definiert wird, da dann der Default-Konstruktor nicht mehr vorhanden ist.

Mit den folgenden Anweisungen kann ein Objekt der Klasse `Klassenname` auch ohne den `new`-Operator geschaffen werden:

```
Class<?> c = Class.forName ("Klassenname");
Klassenname p2 = (Klassenname) c.newInstance();
```

17.5.2 Einsatz des Klassenliterals

Die Verwendung des sogenannten Klassenliterals stellt eine weitere Möglichkeit dar, sich die Referenz auf ein bestimmtes `Class<T>`-Objekt zu beschaffen.

Das Klassenliteral ist ein Ausdruck, der sich aus dem Klassennamen eines Referenztyps oder dem Namen eines einfachen Datentyps, gefolgt von einem Punkt `.` und dem Schlüsselwort `class` zusammensetzt.

Gültige Klassenliterale sind beispielsweise:

- `Object.class`
- `int.class`
- `double[].class`

Das Klassenliteral `TypBezeichnung.class` liefert eine Referenz auf ein `Class<T>`-Objekt zurück, wobei folgende Regeln gelten:

- Stellt `TypBezeichnung` einen Referenztyp – also eine Klasse, einen Aufzählungstyp, eine Schnittstelle oder ein Array – dar, so ist die zurückgelieferte Referenz vom Typ `Class<TypBezeichnung>`.
- Wird mit `TypBezeichnung` ein einfacher Datentyp – also `int`, `double`, `boolean`, etc. – bezeichnet, so besitzt die zurückgelieferte Referenz den Typ `Class<Wrapperklassentyp>` – also `Class<Integer>`, `Class<Double>`, `Class<Boolean>` etc.

Der Compiler wird Klassenliterale nicht akzeptieren und einen Übersetzungsfehler melden, wenn:

- im Klassenliteral statt des Klassennamens eine Referenzvariable verwendet wird, beispielsweise `primitivRef.class` oder `klassenRef.class`.
- das Klassenliteral einen Bezeichner beinhaltet, der einen formalen Typ-Parameter eines generischen Typs darstellt, beispielsweise `T.class`.

Das folgende **Beispielprogramm** veranschaulicht die Verwendung des Klassenliterals:

```
// Datei: Klassenliteral.java
public class Klassenliteral<T>
{
   public Klassenliteral (T t)
   {
      Class<?> thisClassRef = this.getClass();

      Class<?> paramClassRef;

      // Das geht nicht
      // paramClassRef = T.class;

      // Aber das ist hingegen möglich
      paramClassRef = t.getClass();

      System.out.println ("Klasse " + thisClassRef.getName() +
         " ist mit \"" + paramClassRef.getName() +
         "\" aktuell parametrisiert worden.");
```

```
      System.out.println (
         "Referenzvariable t zeigt auf \"" + t + "\"");
   }

   public static void main (String[] args)
   {
      Class<String> stringRef = String.class;
      System.out.println ("stringRef zeigt auf " + stringRef);

      Class<Integer> intRef = int.class;
      System.out.println ("intRef zeigt auf " + intRef);

      Class<int[]> intArrayRef = int[].class;
      System.out.println ("intArrayRef zeigt auf " + intArrayRef);

      Klassenliteral<String> genObjektRef =
         new Klassenliteral<> ("Literaltest");

      // Eine gültige Referenzvariable
      Class<Klassenliteral<String>> genClassRef;

      // Die Zusammensetzung eines Klassenliterals
      // aus der Bezeichnung einer Referenzvariablen
      // lässt der Compiler nicht zu
      // genClassRef = genObjektRef.class;
   }
}
```

Die Ausgabe des Programms ist:

```
stringRef zeigt auf class java.lang.String
intRef zeigt auf int
intArrayRef zeigt auf class [I
Klasse Klassenliteral ist mit "java.lang.String"
aktuell parametrisiert worden.
Referenzvariable t zeigt auf "Literaltest"
```

Bitte beachten Sie bei der Ausgabe der Referenzvariablen `intArrayRef`, dass für ein Array aus `int`-Werten ein Objekt vom Typ `[I` angelegt wird. Dieser außergewöhnliche Klassenname bedeutet, dass durch das Objekt ein Array aus `int`-Werten repräsentiert wird.

17.6 Generizität bei variablen Parameterlisten

Variable Parameterlisten lassen sich auch mit Generizität kombinieren. Es ist also möglich, Methoden zu spezifizieren, bei denen der Elementtyp der Parameterliste variabler Länge ein generischer Typ ist. So ist beispielsweise die folgende Methode möglich:

```
public <T> void methode(T...t)
{
    // generischer Quelltext
}
```

Verwendet man als Elementtyp für die Liste variabler Länge einen generischen Typ, so wird im Hintergrund ein Array erzeugt. Die Programmiersprache Java erlaubt aber keine Definition von Arrays, die Objekte eines generischen Typs als Arrayelemente enthalten. Während der automatischen Generierung des Arrays bei einer Liste variabler Länge mit einem generischen Parameter wird statt eines Arrays von generischen Elementen ein **Array vom Typ `Object[]`**, d. h. aus Elementen der Klasse `Object`, erstellt. Das kann zu Problemen bei der Typsicherheit führen, die durch den Compiler als Warnungen angezeigt werden. Diese Warnungen deuten auf eine mögliche Heap Pollution hin.

Heap Pollution ist ein Prozess, in welchem einer Referenz eines parametrisierten Typs ein Objekt zugewiesen wird, das nicht von diesem parametrisierten Typ ist. Dies ist durch die nachträgliche Implementierung der Generizität in Java möglich geworden.

Der Grund hierfür ist die Sicherstellung der Kompatibilität zu älteren Bibliotheken und Anwendungen, die Generizität nicht unterstützen. Der Java-Compiler entfernt hierfür zur Kompilierzeit einen Teil der Typinformation (**Type Erasure**) aus den Klassen mit generischen Parametern (siehe Kapitel 17.1.2). Diese Klassen werden in Java als zur Laufzeit nicht eindeutig bestimmbare Typen (engl. non-reifiable types) bezeichnet, da ihre Typen zur Laufzeit nicht vollständig verfügbar sind. Ein Objekt der Klasse `KlasseA<T>` resultiert zur Laufzeit in einem Objekt der Klasse `KlasseA<Object>`.

Heap Pollution kann zur Laufzeit zum Werfen einer Exception[218] vom Typ `ClassCastException` führen, da infolge der Type Erasure zu `Object` eine Referenz auf eine generische Klasse nicht auf eine generische Klasse des richtigen Typs zeigt.

Das folgende **Beispiel** verdeutlicht, wie man Heap Pollution und eine daraus resultierende Exception vom Typ `ClassCastException` hervorrufen kann.

Hier zunächst die im Folgenden benötigte Klasse `KlasseA`:

```
public class KlasseA<T>
{
   private T wert;

   public void hinzufuegen (T wert)
   {
      this.wert = wert;
   }


   public T holen()
   {
      return wert;
   }
}
```

[218] Siehe Kapitel 13

Und nun das relevante Codestück:

```
KlasseA a = new KlasseA<Integer>(); // Die Klasse KlasseA
                                    // links des Zuweisungs-
                                    // operators ist nicht
                                    // parametrisiert.
a.hinzufuegen(new Integer(42));

KlasseA<String> s = new KlasseA<String>();
s = a;                      // Heap Pollution geht problemlos
                            // wegen der Type Erasure

String str = s.holen()      // ClassCastException, da der
Rückgabetyp
                            // nicht vom Typ String sondern
                            // vom Typ Integer ist
```

Ist gesichert, dass Heap Pollution nicht auftritt – also keine Exception vom Typ `ClassCastException` auftreten kann –, kann dies mit Annotationen (siehe Kapitel 6.9) kenntlich gemacht werden. Durch das Annotieren kann eine Warnung bezüglich der Typsicherheit unterdrückt werden. Es werden zwei Möglichkeiten angeboten:

- Methoden, die mit den Modifikatoren `final` oder `static` gekennzeichnet sind, können mit der Annotation `@SuppressWarnings ("unchecked")` bzw. `@SafeVarargs`[219] ausgezeichnet werden.
- Die Methoden, die nicht mit den Modifikatoren `final` oder `static` gekennzeichnet sind, können mit der Annotation `@SuppressWarnings ("unchecked")` versehen werden.

Wird an vielen Stellen im Quellcode eine Abschaltung der Warnungen notwendig, kann die Compileroption `-Xlint:varargs` verwendet werden.

17.7 Generizität und Polymorphie

Um die Einordnung der generischen Klassen (siehe Parametrische Polymorphie) in die verschiedenen Formen der Polymorphie zu verdeutlichen, soll das Bild 17-9 betrachtet werden:

[219] Mit `@SafeVarargs` können Warnungen des Compilers unterdrückt werden, die bei Methoden und Konstruktoren mit generischen Parameterlisten variabler Länge eine mögliche Typunsicherheit signalisieren.

Bild 17-9 Formen der Polymorphie

Dieses Bild zeigt die verschiedenen Formen der Polymorphie in einer hierarchischen Anordnung.

17.7.1 Parametrische Polymorphie

Bei **generischen Klassen** handelt es sich um eine sogenannte **Parametrische Polymorphie**. Dies bedeutet, dass ein und derselbe Algorithmus auf verschiedene Datentypen angewandt werden kann. Ein Beispiel ist ein Algorithmus zur Ermittlung der Anzahl der Elemente einer Liste. Dieser Algorithmus ist unabhängig davon, welchen Typ die Listenelemente haben. Parametrische Polymorphie wird auch **Generizität** genannt.

Während man bei der Vererbung durch das **liskovsche Substitutionsprinzip** die Möglichkeit hat, einen **dynamischen Typ** zu variieren, hat man durch die **Generizität** die Möglichkeit, **statische Typen** zu variieren.

Bei dynamischen Typen kann einer Referenzvariablen einer Basisklasse ein Objekt einer abgeleiteten Klasse zugewiesen werden. Von welchem Typ dieses Objekt ist, steht erst zur Laufzeit fest. Daher spricht man hier von einem **dynamischen Typ**.

Bei **statischen Datentypen** ist der **Typ zur Kompilierzeit bekannt** und wird einer Variablen beim Kompilieren zugewiesen. Bei **dynamischen Typen** ist der **Typ erst zur Laufzeit bekannt** und wird einer Variablen zur Laufzeit des Programms durch das Laufzeitsystem zugewiesen.

Während ein "normaler" statischer Datentyp fest vorgegeben ist wie z. B. der Typ `int` oder die Klasse `Punkt`, wird bei einem **generischen Datentyp**, der auch ein statischer Datentyp ist, mit einem sogenannten formalen Typ-Parameter gearbeitet, der einen Stellvertreter für einen konkreten Datentyp darstellt und es erlaubt, einen statischen Datentyp zu variieren.

17.7.2 Polymorphie von Objekten bei Vererbung

Polymorphie von Objekten bei Vererbung – in der wissenschaftlichen Literatur oft als **Inklusions-Polymorphie** bezeichnet – bedeutet, dass eine Referenz einer Basisklasse auf ein Objekt einer Basisklasse oder einer abgeleiteten Klasse zeigen kann. Nach dem liskovschen Substitutionsprinzip kann ein Objekt einer abgeleiteten Klasse an die Stelle eines Objektes der Basisklasse treten, wenn es im Falle des Überschreibens deren Verträge einhält. Liegt kein Überschreiben vor, sondern nur ein Erweitern, so ist die Ersetzung problemlos, da die Methoden der abgeleiteten Klasse durch das Casten unsichtbar werden. Wird überschrieben, so werden infolge der dynamischen Bindung die überschreibenden Instanzmethoden der abgeleiteten Klasse aufgerufen.

Es gibt einen prinzipiellen Unterschied zwischen Parametrischer Polymorphie und InklusionsPolymorphie:

- Bei der Inklusions-Polymorphie arbeitet der Programmierer in seiner Anwendung mit Basisklassen und abgeleiteten Klassen.
- Bei der Parametrischen Polymorphie geht es darum, dass ein und derselbe Algorithmus für verschiedene Datentypen funktioniert, die nicht unbedingt voneinander abgeleitet sein müssen.

17.7.3 Coercion

Coercion ist eine automatische Typanpassung, in anderen Worten eine implizite Typkonvertierung. Sie findet beispielsweise statt, wenn zwei Werte von verschiedenem Typ Operanden desselben Operators sind und eine Typkompatibilität besteht. Andere Beispiele sind implizite Typkonvertierungen beim Zuweisungsoperator, bei Übergabeparametern und Rückgabewerten von Methoden. Ein einfaches Beispiel ist die Multiplikation 3 * 2.0, wobei vor der Durchführung der Multiplikation die ganze Zahl 3 in den Typ `double` gewandelt wird.

17.7.4 Überladen

Überladen von Methoden bedeutet, dass die überladenen Methoden denselben Methodennamen, aber eine unterschiedliche Signatur besitzen. Stimmen die Typen der aktuellen Parameter nicht genau mit den Typen der formalen Parameter beim Methodenaufruf überein, so muss entsprechend den Regeln der impliziten Typkonvertierung die "ähnlichste" überladene Methode gefunden werden.

Die Möglichkeiten der Überladung werden durch den Einsatz von Generics eingeschränkt, da durch die Type Erasure verschiedene generische Methoden dieselbe Signatur haben können (siehe Kapitel 17.1.2 "Type-Erasure-Technik").

Beispiel:

```
// Der Compiler wandelt beide Methoden in
// void methode(List list); um.
```

```
void methode(List<Integer> list);
void methode(List<Double> list);
```

Der Compiler weist mit einer Fehlermeldung wie "`name clash: method (List<Integer>) and methode(List<Double>) have the same erasure`".

17.8 Zusammenfassung

In Kapitel 17 wird die Generizität behandelt.

Seit Java 5 unterstützt Java die Definition:

- generischer Klassen,
- generischer Schnittstellen
- und generischer Methoden.

Diese werden in Java als Generics bezeichnet. Generisch bedeutet in diesem Zusammenhang, dass Klassen, Schnittstellen und Methoden Parameter verwenden, die einen Typ darstellen.

Eine Klasse mit formalem Typ-Parameter stellt eine generische Klasse dar. Der formale Typ-Parameter ist ein symbolischer Name, der wie ein normaler Bezeichner in Java aufgebaut ist und in spitzen Klammern eingeschlossen nach dem Klassennamen angegeben werden muss.

Mit der Generizität werden die folgenden Ziele verfolgt:

- Höhere Typsicherheit: Erkennen von Typ-Umwandlungsfehlern zur Kompilierzeit statt zur Laufzeit.
- Wiederverwendbarkeit von Quellcode.
- Vermeiden des expliziten Casts, der beim Auslesen aus einer Collection aus Elementen vom Typ `Object` notwendig ist.

Kapitel **17.1** beschreibt die Definition von generischen Klassen anhand von Codebeispielen und zeigt auf, wie die Klassen definiert werden und wie Objekte davon instanziiert werden können. Die Idee hinter generischen Klassen ist, dass dieselbe Klasse für unterschiedliche Typ-Parameter verwendet werden kann, ohne dass die verwendeten Typen in derselben Vererbungshierarchie sind oder Casts nötig werden. Generische Klassen werden mit einem sogenannten formalen Typ-Parameter deklariert, der in spitzen Klammern `<>` direkt hinter dem Klassennamen angegeben wird. Die Klasse `Punkt<T>` stellt eine generische Klasse mit dem formalen Typ-Parameter `T` dar. Die Entscheidung über den konkret zu verwendenden Typ – hier `Integer` bzw. `Double` – findet erst bei der Nutzung der generischen Klasse statt.

Der formale Typ-Parameter `T` in der Klassendeklaration wird auch zur Definition der Datenfelder der generischen Klasse verwendet. Genauso tritt er als Typ für formale Parameter und Rückgabewerte von Methoden auf. Damit können Klassen unabhängig von einem speziellen Typ generisch definiert werden. Der formale Typ-Parameter wird bei der Verwendung der Klasse dann durch den gewünschten konkreten Daten-

typ ersetzt. Dadurch entsteht eine konkrete Ausprägung der generischen Klassen, in anderen Worten eine aktuell parametrisierte Klasse.

Es existiert eine Kurzschreibweise bei der Erzeugung von generischen Typen. Die Angabe des aktuellen Typ-Parameters kann für den Aufruf des `new`-Operators plus Konstruktor seit Java 7 weggelassen werden, wie in folgendem Beispiel gezeigt wird:

```
Punkt<Integer> integerPunkt = new Punkt<> (1, 2);
```

Diese Schreibweise wird als Diamant-Operator bezeichnet.

Der Compiler erzeugt bei der Übersetzung einer generischen Klasse nur eine einzige Bytecode-Datei, unabhängig davon, wie viele aktuell parametrisierte Klassen der generischen Klasse verwendet werden. Für die generische Klasse `Punkt<T>` erzeugt der Compiler die Datei `Punkt.class`.

Der aktuelle Typ-Parameter muss ein Referenztyp sein. Damit ist es in Java nicht möglich, generische Klassen zu schreiben, die als aktuellen Typ-Parameter einen einfachen Typ wie `int`, `float` oder `double` akzeptieren.

Die generische Klasse `Punkt<T>` nimmt je nach aktuellem Typ-Parameter eine unterschiedliche Gestalt an. Sie ist also polymorph. Bitte beachten Sie, dass die aktuell parametrisierten Klassen auf derselben Ebene stehen und dass zwischen ihnen keine Vererbungsbeziehung besteht.

Der Compiler entfernt beim Kompilieren den formalen Typ-Parameter `T` und ersetzt ihn durch die Klasse `Object`. Dadurch hat man vor dem Kompilieren die Typprüfung durch den Compiler und nach dem Kompilieren hat man nur eine einzige `class`-Datei. Diese Technik nennt man Type Erasure, da der Typ durch die Ersetzung durch die Klasse `Object` quasi "ausradiert" wird.

Type Erasure kann bei überladenen Methoden zu Problemen führen, wenn die Signatur mehrerer Methoden nach der Ersetzung des Typs durch `Object` nicht mehr eindeutig ist.

Es wird aufgezeigt, wie man Klassen mit mehreren formalen Typ-Parametern definieren kann. Typ-Parameter werden in spitzen Klammern kommasepariert aufgelistet. Die Liste mit den formalen Typ-Parametern in den spitzen Klammern, welche direkt nach dem Klassennamen angeschrieben wird, ist auch unter dem Namen Typ-Parameter-Sektion bekannt. Es ist somit möglich, eine generische Klasse mit beliebig vielen formalen Typ-Parametern zu versehen. Beim Übersetzen einer generischen Klasse `GenerischeKlasse<A, B, C>` werden dann durch die Type Erasure – wie bei einer generischen Klasse mit nur einem einzigen formalen Typ-Parameter auch – alle Vorkommen der formalen Typ-Parameter – hier `A`, `B` und `C` – durch den Typ `Object` ersetzt.

Es wird erklärt, dass aktuell parametrisierte Klassen desselben generischen Typs in keiner Verwandtschaftsbeziehung zueinander stehen. Angenommen, die Klasse `K1` ist die Basisklasse für eine Klasse `K2` und `GenKlasse` stellt die Definition einer

generischen Klasse dar. Dann gilt nicht, dass `GenKlasse<K2>` einen Subtyp von `GenKlasse<K1>` darstellt.

Eine generische Klasse kann:

- eine andere generische Klasse erweitern oder
- eine herkömmliche Klasse erweitern oder
- aber die Vaterklasse einer herkömmlichen Klasse sein.

Leitet eine generische Klasse von einer anderen generischen Klasse ab, so muss der formale Typ-Parameter der Klasse, von der abgeleitet wird, entweder durch den formalen Typ-Parameter der abgeleiteten Klasse oder durch einen aktuellen Typ-Parameter ersetzt werden.

Der Umstand, dass eine generische Klasse von einer nicht-generischen Klasse abgeleitet wird, hat keinen Einfluss auf den Code der einfachen Klasse. Eine generische Klasse erbt hier nur die nicht generischen Eigenschaften wie Datenfelder und Methoden einer herkömmlichen Klasse.

Der dritte Fall beschreibt das Ableiten einer nicht generischen von einer generischen Klasse. Dies kann nur geschehen, wenn beim Ableiten der formale Typ-Parameter der generischen Klasse durch einen aktuellen Typ-Parameter ersetzt wird.

Kapitel **17.2** beschreibt die Definition eigenständig generischer Methoden. Eigenständig generische Methoden sind Methoden mit formalen Typ-Parametern, wobei die Klasse, welche die Methode enthält, nicht generisch ist. Bei der Deklaration eigenständig generischer Methoden steht die Typ-Parameter-Sektion direkt vor dem Rückgabewert der Methode bzw. direkt vor dem Klassennamen des Konstruktors, womit die dort aufgeführten formalen Typ-Parameter für diese Methode bzw. den Konstruktor bekannt gemacht werden. Die Gültigkeit der durch die Typ-Parameter-Sektion bekannt gemachten formalen Typ-Parameter bezieht sich nicht wie bei einer generischen Klasse auf die gesamte Klasse, sondern nur auf die entsprechende Methode.

Eine Erklärung der Bounded Typ-Parameter und Wildcards findet sich in Kapitel **17.3**.

Bei der Angabe eines Bounded Typ-Parameters für einen formalen Typ-Parameter ersetzt der Compiler mit Hilfe der Type Erasure-Technik den formalen Typ-Parameter `T` durch den Typ, der mittels des Bounded Typ-Parameters angegeben wird. So werden bei der Übersetzung der Klasse `class Punkt<T extends Number>` alle Vorkommen von `T` durch `Number` ersetzt. Mit einem Bounded Typ-Parameter kann der zulässige Wertebereich von Typ-Parametern auf einen Teilbaum einer Klassenhierarchie eingeschränkt werden.

Die Unbounded Wildcard löst das Problem, dass aktuell parametrisierte Klassen eines generischen Typs in keiner Vererbungsbeziehung zueinander stehen. Um eine Referenzvariable definieren zu können, die auf Objekte beliebiger aktuell parametrisierter Klassen eines generischen Typs zeigen kann, wurde die Wildcard `?` eingeführt.

Die Wildcard `?` steht im Gegensatz zu einem formalen Typ-Parameter `T` nicht stellvertretend für genau einen Typ, sondern für alle möglichen Typen. So kann eine Referenzvariable vom Typ `Punkt<?>` auf Objekte aller aktuell parametrisierten Klassen des generischen Typs `Punkt<T>` zeigen. Es ist nicht erlaubt, die Wildcard `?` in der Typ-Parameter-Sektion einer generischen Klasse, Schnittstelle oder Methode anzugeben. Die Wildcard `?` darf nur beim Anlegen einer Referenzvariablen oder beim Anlegen eines Arrays aus Referenzen auf Objekte generischer Typen verwendet werden.

Der Typ-Parameter `T` kann mit Hilfe der Upper Bound Wildcard auf einen Typ `A` selbst und auf Subtypen der Klasse `A` einschränkt werden. Die Upper Bound Wildcard `<? extends UpperBound>` legt eine obere Schranke für einen formalen Typ-Parameter fest. Die Upper Bound Wildcard darf nicht in der Typ-Parameter-Sektion einer generischen Klasse verwendet werden.

Die Lower Bound Wildcard dient dazu, den Wertebereich von zulässigen aktuellen Typ-Parametern in der Klassenhierarchie nach unten einzuschränken. Die Lower Bound Wildcard `<? super C>` deckt die Klasse `C` und alle Klassen ab, von denen `C` direkt oder indirekt abgeleitet ist. Es wird damit eine untere Schranke für den Wertebereich eines formalen Typ-Parameters definiert. Auch die Lower Bound Wildcard darf nicht in der Typ-Parameter-Sektion einer generischen Klasse verwendet werden.

Kapitel **17.4** geht auf generische Schnittstellen ein. Bei generischen Schnittstellen kann die implementierende Klasse wählen, ob sie den Typ-Parameter durch eine konkrete Klasse parametrisiert oder generisch lässt. Ersetzt die Klasse den generischen Typ, implementiert sie eine konkrete Ausprägung der generischen Schnittstelle. Behält die Klasse den generischen Typ bei, implementiert sie alle Ausprägungen, die die generische Schnittstelle vorgibt. Die Techniken der vorherigen Kapitel lassen sich mit generischen Schnittstellen kombinieren, um z. B. Einschränkungen bezüglich der möglichen Typ-Parameter vorzugeben. Die Bounds können auch auf Schnittstellen angewendet werden können.

Generische Schnittstellen können – wie herkömmliche Schnittstellen auch – Vererbungshierarchien aufbauen. Dabei gelten die gleichen Regeln, wie sie schon bei der Beschreibung von Vererbungsbeziehungen bei generischen Klassen aufgestellt wurden.

Weiter wird die Schnittstelle `Comparable` vorgestellt und gezeigt, wie die Methode `compareTo()` der generischen Schnittstelle `Comparable<T>` implementiert werden kann, um Objekte eigener Klassen mittels `Arrays.sort()` sortierbar zu machen.

Kapitel **17.5** stellt die Klasse `Class<T>` vor. Für jede verwendete Klasse existiert eine Instanz der Klasse `Class<T>` in der virtuellen Maschine. Dieses Objekt enthält alle Eigenschaften der verwendeten Klasse und wird für Java-Mechanismen wie Reflection und Serialisierung herangezogen. Zu jedem existenten Objekt kann durch Aufruf der Instanzmethode `getClass()` der Klasse `Object` eine Referenz auf das zugehörige `Class<T>`-Objekt erhalten werden. Ein Programmierer kann von der Klasse `Class<T>` selbst keine Objekte mit dem `new`-Operator anlegen, da der Konstruktor der Klasse `Class<T>` `private` ist.

Die beiden folgenden Anweisungen stellen eine zusätzliche Möglichkeit dar, wie außer durch den `new`-Operator Objekte einer Klasse geschaffen werden können:

```
Class<?> c = Class.forName ("Klassenname");
Klassenname p2 = (Klassenname) c.newInstance();
```

Eine Referenz auf ein `Class<T>`-Objekt erhält man auch, indem man das Klassenliteral verwendet. Das Klassenliteral ist ein Ausdruck, der sich aus dem Klassennamen eines Referenztyps oder dem Namen eines einfachen Datentyps, gefolgt von einem Punkt `.` und dem Schlüsselwort `class` zusammensetzt.

Kapitel **17.6** befasst sich mit der Generizität bei variablen Parameterlisten.

In Kapitel **17.7** wird schließlich Generizität und Polymorphie besprochen. Generische Klassen und Methoden sind im Bezug auf die Polymorphie eingeschränkt. Lediglich statische Polymorphie ist möglich, da die Generizität zur Kompilierzeit aufgelöst wird. So ist es z. B. durch die Type Erasure nicht möglich, generische Methoden zu überladen. Generell zielt Generizität mit seiner parametrischen Polymorphie darauf ab, Berechnungen für verschiedene unabhängige Typen durchführbar zu machen, während die Inklusions-Polymorphie von herkömmlichen Klassen mit Basisklassen und deren Kindern arbeitet.

17.9 Übungen

Aufgabe 17.1: Kleinteilemagazin

Eine Schlosserei benötigt für die Verwaltung ihrer Kleinteile eine neue Verwaltungssoftware. Es sollen Schrauben, Muttern und Unterlegscheiben katalogisiert werden. Damit die Entwicklung weniger Zeit beansprucht, ist eine Klassenbibliothek für die Abbildung der zu verwaltenden Teile von einer Fremdfirma hinzugekauft worden. Die gekaufte Klassenbibliothek lässt sich folgendermaßen in UML abbilden:

Von der Klassenbibliothek haben Sie auch die Lizenz der Weiterentwicklung und Modifizierung erworben. Das heißt, Ihnen wurde beim Kauf der Quellcode mitgeliefert, der abgeändert und an Ihre Ansprüche angepasst werden kann. Die Klassenbibliothek ist zunächst nicht generisch, sondern besteht aus herkömmlichen Java-Klassen. Die Information, über welche Datenfelder und Methoden die einzelnen Klassen der Bibliothek verfügen, müssen Sie sich direkt aus dem Quellcode beschaffen.

17.1.1 Implementierung der Klasse `KleinteileMagazin`

Schreiben Sie eine generische Klasse `KleinteileMagazin`, mit der es möglich ist, beliebige Kleinteile zu erfassen. Es soll möglich sein, mit einer Instanz einer aktuell parametrisierten Klasse des generischen Typs `KleinteileMagazin` beliebig viele Posten gleichartiger Kleinteile aufzunehmen, z. B. Scheiben von verschiedenem Durchmesser oder Gewindeteile in verschiedenster Ausführung.

Um dies zu bewerkstelligen, soll die generische Klasse `ArrayList<T>` der Java-Klassenbibliothek aus dem Paket `java.util` verwendet werden. Diese Klasse ist eine sogenannte Container-Klasse (siehe Kapitel 19), die es ermöglicht, beliebig viele Referenzen auf Objekte eines frei wählbaren Typs aufzunehmen. Für die Manipulation und Verwaltung der gespeicherten Referenzen stellt die Klasse `ArrayList<T>` unter anderem die folgenden Methoden bereit:

- `public boolean add (T ref)`

  Einfügen einer übergebenen Referenz `ref` vom Typ `T` in das Collection-Objekt.

- `public T get (index)`

  Liefert die Referenz auf das Objekt vom Typ `T`, das sich an der Stelle `index` im Collection-Objekt befindet.

- `public void set (int index, T obj)`

  Setzt die Referenz vom Typ `T` an der Stelle `index` im Collection-Objekt auf die Referenz, die über `obj` übergeben wurde.

- `public int indexOf (Object obj)`

  Liefert den Index des ersten Auftauchens des gesuchten Objektes im Collection-Objekt, auf das die Referenzvariable `obj` zeigt. Die Methode liefert `-1` zurück, falls das gesuchte Objekt nicht im Collection-Objekt enthalten ist.

Implementieren Sie die Klasse `KleinteileMagazin`. Sie soll die folgenden Methoden bereitstellen:

- `public boolean aufnehmen (T posten)`

  Es soll geprüft werden, ob der Posten `posten` vom Typ `T` in dieser Art und Ausführung schon im Magazin enthalten ist. Ist dies der Fall, dann soll die Methode `false` zurückliefern, sonst `true`.

- `public boolean einlagern (String bezeichnung,`
  `String ausfuehrung, int stueckzahl)`

  Einlagern von Kleinteilen in das Magazin. Es muss der Posten schon im Magazin enthalten sein, damit die Kleinteile eingelagert werden können. Ist dies der Fall, dann soll die Stückzahl erhöht und `true` zurückgeliefert werden. Ansonsten soll die Methode den Wert `false` zurückliefern.

- `public boolean entnehmen (String bezeichnung, String ausfuehrung, int stueckzahl)`

  Entnahme von Kleinteilen mit gewünschter Bezeichnung und in entsprechender Ausführung. Sind weniger Teile als gewünscht vorhanden, so soll nichts entnommen und `false` zurückgeliefert werden.

- `public void druckeMagazin()`

  Es soll damit der Magazinbestand ausgegeben werden können.

Verwenden Sie für die Implementierung die folgende Vorlage der Klasse `KleinteileMagazin` in der Datei `KleinteileMagazin_Vorlage.java`. Alle Stellen in der Vorlage, an denen Punkte als Platzhalter stehen, müssen entsprechend ersetzt werden. Bevor Sie die Klasse übersetzen können, müssen Sie die Datei entsprechend umbenennen:

```
// Datei: KleinteileMagazin_Vorlage.java
package magazin;

import java.util.ArrayList;

// Die benötigten Klassen der hinzu gekauften
// Klassenbibliothek importieren
. . . . .

// Die generische Klasse soll Objekte vom Typ Teil verwalten können
public class KleinteileMagazin . . . . .
{
   // Das ArrayList<T>-Objekt dient als interne Datenstruktur
   private ArrayList<T> magazin;


   public KleinteileMagazin()
   {
      magazin = new ArrayList<T>();
   }

   // Aufnehmen des übergebenen Postens in das Magazin
   public boolean aufnehmen (T posten)
   {
      // Prüfen, ob der Posten mit Bezeichnung
      // und Ausführung schon vorhanden ist
      int index = . . . . .

      // Ist der Posten schon vorhanden, so soll
      // die Methode false zurückliefern
      . . . . .

      // Aufnehmen des Postens in das Magazin
      . . . . .
   }

   // Einlagern des Kleinteils in beschriebener Ausführung
   public boolean einlagern (String bezeichnung,
                             String ausfuehrung, int stueckzahl)
```

```
   {
      // Ist der Posten verfügbar?
      // Die vorhandene Stückzahl ist egal.
      int index = . . . . .
      // Wenn ja, dann die vorhandene Stückzahl
      // des Postens um die übergebene Stückzahl erhöhen
      if (index >= 0)
      {
         . . . . .
      }
      // Posten ist nicht vorhanden
      . . . . .
   }

   // Entnehmen des gewünschten Kleinteils
   public boolean entnehmen (String bezeichnung,
                             String ausfuehrung, int stueckzahl)
   {
      // Ist der gesuchte Posten in
      // ausreichender Stückzahl verfügbar?
      int index = . . . . .
      // Wenn ja, dann die gewünschte Stückzahl entnehmen
      if (index >= 0)
      {
         . . . . .
      }

      // Nicht in ausreichender Stückzahl verfügbar
      . . . . .
   }

   // Den Bestand des Magazins ausdrucken
   public void druckeMagazin()
   {
      . . . . .
   }
   // Hilfsmethode, die überprüft, ob der Posten in Bezeichnung,
   // Ausführung und gewünschter Stückzahl schon im Magazin
   // vorhanden ist. Sie liefert dann dessen Index im
   // ArrayList<T>-Objekt zurück, sonst -1
   private int istVerfuegbar (String bezeichnung,
                              String ausfuehrung, int anzahl)
   {
      // Suchen, bis der Posten in Bezeichnung und
      // Ausführung gefunden wurde. Dann den Index
      // im ArrayList<T>-Objekt zurückliefern.
      for (T posten : magazin)
      {
         . . . . .
      }
      // Nicht verfügbar? -1 zurückliefern.
      . . . . .
   }
}
```

Benutzen Sie zum Testen die Klasse `KleinteileMagazinTest`:

```
// Datei: KleinteileMagazinTest.java
package magazin;

import teile.GewindeTeil;
import teile.Mutter;
import teile.Scheibe;
import teile.Schraube;

public class KleinteileMagazinTest
{
   public static void main (String[] args)
   {
      // Anlegen zweier Magazine für Gewindeteile und Scheiben
      KleinteileMagazin<GewindeTeil> gewindeteile =
         new KleinteileMagazin<>();
      KleinteileMagazin<Scheibe> scheiben =
         new KleinteileMagazin<Scheibe>();

      // Anlegen von Gewindeteilen
      Mutter mutter_m5 = new Mutter (1000, Mutter.NORMAL, 6, 5);
      Mutter mutter_m6 = new Mutter (1000, Mutter.SICHERUNG, 6, 6);
      Schraube schraube_m30 = new Schraube (
         1000, Schraube.EINFACHSCHLITZ, Schraube.LINSENKOPF, 30);
      Schraube schraube_m40 = new Schraube (
         1000, Schraube.KREUZSCHLITZ, Schraube.LINSENKOPF, 40);

      // Gewindeteile aufnehmen
      gewindeteile.aufnehmen (mutter_m5);
      gewindeteile.aufnehmen (mutter_m6);
      gewindeteile.aufnehmen (schraube_m30);
      gewindeteile.aufnehmen (schraube_m40);

      // Gewindeteile-Magazin ausdrucken
      gewindeteile.druckeMagazin();

      // Verschiedenste Gewindeteile entnehmen und wieder einlagern
      gewindeteile.entnehmen (mutter_m5.getBezeichnung(),
         mutter_m5.getAusfuehrung(), 600);
      gewindeteile.entnehmen ("Mutter", Mutter.NORMAL, 500);
      gewindeteile.einlagern ("Mutter", Mutter.SICHERUNG, 4000);

      // Gewindeteile-Magazin ausdrucken
      gewindeteile.druckeMagazin();

      // Scheiben anlegen
      Scheibe scheibe_m5 = new Scheibe (1000, Scheibe.FEDER, 6, 10);
      Scheibe scheibe_m6 =
         new Scheibe (1000, Scheibe.NORMAL, 7, 11);

      // Scheiben im Scheiben-Magazin aufnehmen
      scheiben.aufnehmen (scheibe_m5);
      scheiben.aufnehmen (scheibe_m6);

      // Scheiben-Magazin ausdrucken
      scheiben.druckeMagazin();
   }
}
```

17.1.2 Implementierung der generischen Klassenbibliothek

Die zugekaufte Klassenbibliothek zur Abbildung der Kleinteile soll nun an neue Anforderungen angepasst werden. Es soll möglich sein, auch Hochpräzisionsteile abzubilden, deren Abmaße im µm-Bereich angegeben werden müssen. Dazu soll die Klassenbibliothek in eine generische Bibliothek umgeschrieben werden. Der Benutzer soll nun die Möglichkeit haben, beim Anlegen eines Objektes vom Typ `Teil` – oder eines Subtyps davon – den Präzisionsgrad des einzulagernden Kleinteils anzugeben. Dies soll folgendermaßen aussehen: Ein Kleinteil, dessen Erfassung eine niedrige Präzision benötigt, soll angelegt werden zu:

```
Teil<Integer> irgendeinteil = new . . . . .;
```

Ein Teil, für dessen Erfassung eine höhere Präzision notwendig ist, wird folgendermaßen referenziert:

```
Teil<Float> weiteresTeil = new . . . . .;
```

Zusätzlich soll der Klasse `Teil` ein weiteres Datenfeld

```
private T toleranz;
```

hinzugefügt werden. Es muss dann für jedes angelegte Objekt vom Typ `Teil` die zulässige Toleranz der Abmaße über den Konstruktor angegeben werden.

Die zuvor implementierte Klasse `KleinteileMagazin` soll jedoch, abgesehen von minimalen Änderungen zum Unterbringen der neu eingeführten Generizität, von der Änderung der Klassenbibliothek nichts merken, das heißt, die Klassenbibliothek zur Abbildung von Kleinteilen soll nur mit Änderungen für den generischen Parameter der Klasse `KleinteileMagazin` komplett ohne Warnungen lauffähig sein. Zum Testen der neu implementierten Klassenbibliothek verwenden Sie folgende Klasse `KleinteileMagazinTest2`:

```
// Datei: KleinteileMagazinTest2.java
package magazin;

import teile.GewindeTeil;
import teile.Mutter;
import teile.Scheibe;
import teile.Schraube;

public class KleinteileMagazinTest2
{
   public static void main (String[] args)
   {
      // Anlegen zweier Magazine für Gewindeteile und Scheiben
      KleinteileMagazin<GewindeTeil<?>> gewindeteile =
         new KleinteileMagazin<GewindeTeil<?>> ();
      KleinteileMagazin<Scheibe<Float>> scheiben =
         new KleinteileMagazin<>();

      // Anlegen von Gewindeteilen
      Mutter<Float> mutter_m5 =
         new Mutter<Float> (1000, Mutter.NORMAL, 6, 5f, 0.01f);
      Mutter<Integer> mutter_m60 =
         new Mutter<Integer> (1000, Mutter.NORMAL, 6, 60, 1);
```

```
      Schraube<Float> schraube_m3 =
         new Schraube<Float> (1000, Schraube.EINFACHSCHLITZ,
            Schraube.LINSENKOPF, 3f, 0.001f);

      // Gewindeteile aufnehmen
      gewindeteile.aufnehmen (mutter_m5);
      gewindeteile.aufnehmen (mutter_m60);
      gewindeteile.aufnehmen (schraube_m3);

      // Gewindeteile-Magazin ausdrucken
      gewindeteile.druckeMagazin();

      // Scheiben anlegen
      Scheibe<Float> scheibe_m5 =
         new Scheibe<Float>(1000,Scheibe.FEDER, 5.5f, 10.1f, 0.01f);
      Scheibe<Float> scheibe_m2 =
         new Scheibe<Float> (1000, Scheibe.NORMAL, 2.11f,
            3.001f, 0.0001f);

      // Scheiben im Scheiben-Magazin aufnehmen
      scheiben.aufnehmen (scheibe_m5);
      scheiben.aufnehmen (scheibe_m2);

      // Scheiben-Magazin ausdrucken
      scheiben.druckeMagazin();
   }
}
```

Als Hilfe für die Implementierung können Sie wiederum die Vorlagen-Dateien `Teil_Vorlage.java`, `GewindeTeil_Vorlage.java`, `Schraube_Vorlage.java`, `Mutter_Vorlage.java` und `Scheibe_Vorlage.java` verwenden. Auch hier müssen Sie daran denken, die Vorlagen-Dateien entsprechend umzubenennen.

17.1.3 Anpassen der Klassenbibliothek

Die Schlosserei hat in der Vergangenheit immer wieder schlechte Erfahrungen mit ansetzendem Rost an den Kleinteilen gemacht. Sie müssen somit Ihre in Aufgabe 17.1.2 erweiterte Klassenbibliothek zur Abbildung von Hochpräzisionsteilen so anpassen, dass nur noch Objekte vom Typ `Teil` angelegt werden können, deren Hersteller mindestens eine 12-monatige Rostfreiheit garantiert. Dieser Zeitraum ist nach oben hin beliebig anpassbar, indem im Konstruktor der instanziierbaren Kleinteile – Schraube, Mutter und Scheibe – eine zusätzliche Zahl übergeben wird. Wird versucht, ein Kleinteil anzulegen, dessen garantierte Rostfreiheit kleiner als 12 Monate ist, so soll eine Exception vom Typ `RostgarantieNichtErfuelltException` geworfen werden.

Um sicherzustellen, dass die angelegten Objekte vom Typ `Teil` auch wirklich rostfrei sind, muss von der Klassenbibliothek eine zusätzliche Schnittstelle implementiert werden:

```
// Datei: Rostfrei.java

public interface Rostfrei
{
   public int gibZeitraumRostfreiheit();
}
```

Die Methode `gibZeitraumRostfreiheit()` soll dann so implementiert werden, dass bei ihrem Aufruf der Zeitraum der garantierten Rostfreiheit zurückgegeben wird. Die Klasse `KleinteileMagazin` muss nun so angepasst werden, dass nur noch Teile aufgenommen werden können, die auch garantiert rostfrei sind. Sinnigerweise wird dann die Klasse in `RostfreieKleinteileMagazin` umbenannt.

Zum leichteren Einstieg können Sie wiederum die Vorlagen-Dateien verwenden. Zum Testen steht Ihnen die Klasse `KleinteileMagazinTest3` zur Verfügung:

```
// Datei: KleinteileMagazinTest3.java

package magazin;
import teile.GewindeTeil;
import teile.Mutter;
import teile.RostgarantieNichtErfuelltException;
import teile.Scheibe;

public class KleinteileMagazinTest3
{
   public static void main (String[] args)
   {
      // Anlegen zweier Magazine für Gewindeteile und Scheiben
      RostfreieKleinteileMagazin<GewindeTeil<Float>> gewindeteile =
         new RostfreieKleinteileMagazin<GewindeTeil<Float>>();
      RostfreieKleinteileMagazin<Scheibe<Float>> scheiben =
         new RostfreieKleinteileMagazin<Scheibe<Float>>();

      try
      {
         // Anlegen einer Mutter mit garantierter
         // Rostfreiheit des Herstellers
         Mutter<Float> mutter_m5 =
            new Mutter<Float> (1000, Mutter.NORMAL, 6, 5f, 0.01f);
         gewindeteile.aufnehmen (mutter_m5);

         // Anlegen einer Mutter mit 18-monatiger Rostfreiheit
         Mutter<Float> mutter_m6 = new Mutter<Float> (
            1000, Mutter.SICHERUNG, 6, 6f, 0.01f, 18);
         gewindeteile.aufnehmen (mutter_m6);

         // Gewindeteile-Magazin ausdrucken
         gewindeteile.druckeMagazin();

         // Versuch, eine Scheibe mit 11-
         // monatiger Rostfreiheit anzulegen
         Scheibe<Float> scheibe_m5 = new Scheibe<Float> (
            1000, Scheibe.FEDER, 5.5f, 10.1f, 0.01f, 11);
         scheiben.aufnehmen (scheibe_m5);

         scheiben.druckeMagazin();
      }
      catch (RostgarantieNichtErfuelltException e)
      {
         System.out.println(e.getMessage());
      }
   }
}
```

17.1.4 Schnittstelle `Comparator<T>` und Wildcards

Zum leichteren Auffinden von Teilen in einem Magazin, soll eine Möglichkeit geschaffen werden, mit der die Kleinteile der einzelnen Magazine nach ihrer Bezeichnung und Ausführung sortiert werden können.

Die Utility-Klasse `Collections` aus der Java-Klassenbibliothek bietet mit der statischen Methode `sort()` die Möglichkeit des Sortierens von Listen. In der Klasse `RostfreieKleinteileMagazin` soll die Liste vom Typ `ArrayList` mit Hilfe dieser Methode sortiert werden. Mit einer Klasse, die die Schnittstelle `Comparator<T>` implementiert, kann die Sortierreihenfolge für beliebige Klassen definiert werden.

Zu diesem Zweck soll eine generische Utility-Klasse `TeileUtils` im Paket `utils` geschaffen werden. Diese Klasse soll die Schnittstelle `Comparator<T>` implementieren und eine statische, eigenständig generische Methode `sortiereMagazin()` bereitstellen. Zur Sortierung der Liste soll die Variante der Methode `sort()` verwendet werden, die zwei Parameter annimmt.

Der Konstruktor dieser Klasse soll nicht öffentlich sein. Stattdessen soll bei der Verwendung der Methode `sortiereMagazin()` eine neue Instanz der Utility-Klasse erstellt werden.

Der formale Typ-Parameter `T` dieser Klasse, sowie der eigenständig generischen Methode `sortiereMagazin()` soll folgenden Einschränkungen unterliegen: `T` darf alle von `Teil` abgeleiteten Typen darstellen und dieser, von Teil abgeleitete Typ, darf wiederum mit einem beliebigen Typ parametrisiert worden sein.

Die Methode `sortiereMagazin()` soll nach ihrem Aufruf die Methode `sort()` der Klasse `Collections` aufrufen. Als erster Parameter soll die übergebene Liste weitergereicht werden, als zweiter Parameter soll eine neue Instanz der zu implementierenden Utility-Klasse übergeben werden. Die Methode `compare()` aus der Schnittstelle `Comparator<T>` soll – wie beschrieben – lexikografisch zunächst die Bezeichnung und im zweiten Schritt die Ausführung der jeweils zu vergleichenden Teile auswerten. Dazu kann die Methode `compareToIgnoreCase()` der Klasse `String` herangezogen werden. Der Rückgabewert der Methode `compare()` soll dabei den Bedingungen, die in der Java-Klassenbibliothek-Dokumentation beschrieben sind, genügen. Die Methode `equals()` aus der Schnittstelle `Comparator<T>` muss nicht implementiert werden.

Zur Implementierung der Klasse `TeileUtils` muss der folgende Programmcode ergänzt werden. Bevor Sie die Klasse übersetzen können, müssen Sie die Datei entsprechend umbenennen:

```
// Datei: TeileUtils_Vorlage.java

package utils;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import teile.Teil;
```

```
public class TeileUtils<T . . . . > implements Comparator<T>
{
   private TeileUtils()
   {
   }
   public int compare (T t1, T t2)
   {
      // String-Vergleiche machen und Ergebnis zurückgeben
      . . . . . . . .
   }

   public static<T . . . . > void sortiereMagazin(List<T> teile)
   {
      // Klassenmethode sort() der Klasse Collections<T> aufrufen
      . . . . . . . .
   }
}
```

Die Klasse `RostfreieKleinteileMagazin` soll um die Methode `sortiereMagazin()` erweitert werden:

```
public void sortiereMagazin()
```

Diese Methode soll lediglich die Methode `sortiereMagazin()` der geschaffenen Utility-Klasse aufrufen und ihr die intern verwendete Liste `magazin` übergeben.

Es sollen alle Klassen aus der vorherigen Aufgabe verwendet werden. Zum Testen steht die Klasse `KleinteileMagazinTest4` zur Verfügung:

```
// Datei: KleinteileMagazinTest4.java
package magazin;

import teile.GewindeTeil;
import teile.Mutter;
import teile.RostgarantieNichtErfuelltException;
import teile.Scheibe;
import teile.Schraube;

public class KleinteileMagazinTest4
{
   public static void main (String[] args)
   {
      // Anlegen zweier Magazine für Gewindeteile und Scheiben
      RostfreieKleinteileMagazin<GewindeTeil<Float>> gewindeteile =
         new RostfreieKleinteileMagazin<GewindeTeil<Float>>();
      RostfreieKleinteileMagazin<Scheibe<Float>> scheiben =
         new RostfreieKleinteileMagazin<Scheibe<Float>>();

      // Hinzufügen der unterschiedlichen Magazine zu einem Array
      RostfreieKleinteileMagazin<?>[] magazine =
         {gewindeteile, scheiben};

      try
      {
         // Anlegen dreier Gewindeteile
         gewindeteile.aufnehmen
```

```
            (new Mutter<> (1000, Mutter.NORMAL, 6, 5f, 0.01f));
         gewindeteile.aufnehmen (new Mutter<>
            (1000, Mutter.SICHERUNG, 6, 6f, 0.01f, 18));
         gewindeteile.aufnehmen (new Schraube<>
            (1000, Schraube.SENKKOPF, Schraube.KREUZSCHLITZ,
            6f, 0.01f, 18));
         // Anlegen einer Scheibe
         scheiben.aufnehmen (new Scheibe<>
            (1000, Scheibe.FEDER, 5.5f, 10.1f, 0.01f, 12));

         // Das Magazin gewindeteile sortieren
         gewindeteile.sortiereMagazin();

         // Inhalt der Magazine ausdrucken
         for (RostfreieKleinteileMagazin<?> magazin : magazine)
         {
            System.out.println ("Gebe naechstes Magazin aus:");
            magazin.druckeMagazin();
         }
      }
      catch (RostgarantieNichtErfuelltException e)
      {
         System.out.println(e.getMessage());
      }
   }
}
```

Aufgabe 17.2: Tanklasterverwaltung – Wildcards

Für eine Tanklasterabfüllanlage soll eine Verwaltung für die an der Anlage angemeldeten Laster implementiert werden. Dazu stehen die Dateien `Laster.java` und `Tanklaster.java` als fertig implementierte Klassen bereit, deren Klassenhierarchie die folgende Struktur hat:

Für diese Klassenhierarchie soll eine Utility-Klasse implementiert werden, die nicht volle und volle Laster ermitteln kann. Um die Verwaltung der Laster so allgemein wie möglich zu erlauben, sollen die Methoden der Utility-Klasse gegen die Schnittstelle `Collection<E>` aus der Java-Klassenbibliothek programmiert werden. Die Utility-Klasse soll lediglich die vollen und die nicht vollen Laster in einer Collection ermitteln, an die Verwaltung zurückgeben und den Inhalt einer Collection auf dem Bildschirm ausgeben können. Da die Abfüllanlage in Zukunft vielleicht erweitert und noch andere Tanklasterarten verwaltet werden könnten, sollen die Methoden dieser Utility-Klasse generisch sein.

Von den Methoden der Schnittstelle `Collection<T>` wird in dieser Aufgabe nur die Methode `public boolean add (T obj)` benötigt. Diese Methode `add()` setzt

die Referenz vom Typ `T` im Collection-Objekt auf die Referenz, die über `obj` übergeben wurde.

Verwenden Sie für die Implementierung der Utility-Klasse die folgende Vorlage in der Datei `TanklasterUtils_Vorlage.java`. Alle Stellen in der Vorlage, an denen Punkte als Platzhalter stehen, müssen entsprechend ersetzt werden. Bevor Sie die Klasse übersetzen können, müssen Sie die Datei entsprechend umbenennen:

```
// Datei: TanklasterUtils_Vorlage.java

import java.util.Collection;

public class TanklasterUtils
{
   // Statische Methode zum Ermitteln aller nicht vollen Laster in
   // einer Collection. Diese Methode soll als ersten Parameter alle
   // von Collection abgeleiteten Typen akzeptieren, die Referenzen
   // auf Tanklaster-Objekte - oder auf Objekte eines Subtyps -
   // enthalten. Als zweiten Parameter sollen alle von Collection
   // abgeleiteten Typen akzeptiert werden, die Referenzen auf
   // Tanklaster-Objekte - oder auf Objekte eines Basistyps -
   // enthalten.
   public static <T . . . . Collection< . . . . . . . >,
      R . . . . Collection< . . . . . . . >>
      void ermittleNichtVolleLaster
      (T alleLaster, R NichtVolleLaster)
   {
      // Collection der Tanklaster nach nicht vollen Lastern
      // durchsuchen.
      for (Tanklaster l : alleLaster)
      {
         // Prüfe, ob Tanklaster nicht voll.
         if ( . . . . . )
         {
            // Gefundenen Laster zur Collection hinzufügen.
            . . . . . .
         }
      }
   }

   // Statische Methode zum Ermitteln aller vollen Laster in einer
   // Collection.
   // Alle Einschränkungen für Typen, die von der Methode
   // ermittleNichtVolleLaster() akzeptiert werden, gelten auch für
   // diese Methode.
   public static <T . . . . Collection< . . . . . . . >,
      R . . . . Collection< . . . . . . . >>
      void ermittleVolleLaster (T alleLaster, R volleLaster)
   {
      // Collection der Tanklaster wird durchsucht nach vollen
      // Lastern
      for (Tanklaster l : alleLaster)
      {
         // Prüfe, ob Laster voll.
         if ( . . . . . )
         {
```

```
            // Gefundenen Laster zur Collection hinzufügen
            . . . . . . . .
         }
      }
   }

   // Statische Methode, um alle Tanklaster in einer Collection
   // auf dem Bildschirm auszugeben. Diese Methode soll als ersten
   // Parameter alle von Collection abgeleiteten Typen akzeptieren,
   // die Referenzen auf Tanklaster-Objekte - oder auf Objekte
   // eines Subtyps - enthalten.
   public static <T . . . . Collection< . . . . . . . >>
      void printLasterFuellstand (T laster)
   {
      for (Tanklaster l : laster)
      {
         l.print();
         System.out.println ("-----------");
      }
   }
}
```

Zum Testen steht Ihnen die Klasse `TanklasterTest` zur Verfügung:

```
// Datei: TanklasterTest.java

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

public class TanklasterTest
{
   public static void main (String[] args)
   {
      Tanklaster[] lasterArr = new Tanklaster[3];

      // Anlegen dreier neuer Tanklaster
      lasterArr[0] =
         new Tanklaster ("Maimler - Tankstelle Lara", 25000d, 1.5d);
      lasterArr[1] =
         new Tanklaster ("Nam - Tankstelle Muschel", 25000d, 1.7d);
      lasterArr[2] =
         new Tanklaster ("WV - Tankstelle Osse", 2500d, 1.7d);

      // Unterschiedliches Befüllen der angelegten Laster
      lasterArr[0].befuelle (lasterArr[0].getMaxFuellmenge());
      lasterArr[1].befuelle (50000d);
      lasterArr[2].befuelle (1000d);

      // Anlegen einer neuen, leeren Ergebnis-Collection
      ArrayList<Tanklaster> list = new ArrayList<Tanklaster>();

      System.out.println ("Nicht volle Laster:\n");
      // Befüllen der Ergebnis-Collection mit nicht vollen Lastern
      TanklasterUtils.ermittleNichtVolleLaster
         (Arrays.asList (lasterArr), list);
      // Ausgeben der Collection auf dem Schirm
      TanklasterUtils.printLasterFuellstand (list);
```

```
        // Säubern der Ergebnis-Collection, um sie wiederzuverwenden
        list.clear();

        System.out.println ("\nVolle Laster:\n");
        // Befüllen der Ergebnis-Collection mit vollen Lastern
        TanklasterUtils.ermittleVolleLaster
           (Arrays.asList (lasterArr), list);
        // Ausgeben der Collection auf dem Schirm
        TanklasterUtils.printLasterFuellstand (list);
    }
}
```

Kapitel 18

Einführung in Collections

18.1 Das Konzept der Collection-API
18.2 Eigenschaften und Fähigkeiten der Collections
18.3 Zusammenfassung
18.4 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_18

18 Einführung in Collections

Collections sind – wie der Name schon andeutet – **Zusammenstellungen von Daten** im Arbeitsspeicher, genauer gesagt von **Objekten, in gewissen Datenstrukturen** wie Listen oder Mengen. Das bekannte Array ist die einfachste Datenstruktur für die Anordnung von Objekten. Darüber hinaus kennt die Informatik noch eine ganze Reihe weiterer Datenstrukturen mit jeweils spezifischen Eigenschaften.

Klassen für **Datenstrukturen** mit speziellen Fähigkeiten nennt man in Java **Collections**.

Der Begriff **Container**, der in anderen Programmiersprachen für Datenstrukturen verwendet wird, hat in Java eine abweichende Bedeutung und wird im Zusammenhang mit der **Oberflächenprogrammierung** verwendet. Als Container bezeichnet man in Java einen Bereich in einem Fenster, der Komponenten als Bausteine enthalten kann, also einen Behälter für diese Bausteine darstellt.

In diesem Kapitel werden die Collections der Collection-API kurz vorgestellt. Kapitel 18.1 stellt das Konzept der Collection-API vor und zeigt an einem einfachen Beispiel, wie eine Collection-Klasse verwendet werden kann. Des Weiteren wird gezeigt, wie man programmieren kann, damit es zu einem späteren Zeitpunkt sehr einfach ist, eine eingesetzte Collection-Klasse durch eine andere Collection-Klasse zu ersetzen. In Kapitel 18.2 werden wichtige Eigenschaften in der Programmierung mit Collections exemplarisch zusammengestellt. Nach einem kurzen Einblick in die Generizität der Collections wird gezeigt, wie Collection-Klassen auch mit einfachen Datentypen verwendet werden können und wie über den Inhalt einer Collection-Klasse iteriert werden kann.

18.1 Das Konzept der Collection-API

Die zentralen Klassen der Collection-API sind im Paket `java.util` zu finden. Diese Collection-API versorgt einen Programmierer mit einem Satz von Klassen und Schnittstellen, die es wesentlich erleichtern, Zusammenstellungen von Objekten zu verwalten. Man muss dabei überhaupt keine eigenen Collection-Klassen schreiben, sondern kann die **zur Verfügung gestellten Datenstrukturen verwenden**.

Für alle Datenbestände, die sich verändern können (also mal mehr oder mal weniger Datensätze speichern), ist der Einsatz einer Collection-Klasse gegenüber einem Array fester Länge zu empfehlen. Beispiele hierfür gibt es sehr viele, denn die meisten Datenbestände erfordern diese Flexibilität: Eine Firma, die ihren Fuhrpark an Fahrzeugen verwalten möchte (Fahrzeuge werden hinzugefügt oder wieder entfernt), ein Semester aus Studenten (das eine Semester hat 50 Studenten, das andere nur 40, mal werden Studenten zusätzlich immatrikuliert, mal exmatrikuliert), Mitarbeiterdaten einer Firma (der Bestand kann durch Einstellungen und Entlassungen

variieren), usw. Natürlich könnte man auf die Idee kommen und ein sehr großes Array anlegen, um mit einem wachsenden Datenbestand zurecht zu kommen, dies bedeutet allerdings immer eine Speicherverschwendung und letztendlich kann man nie sicher sein, ob die definierte Größe irgendwann nicht doch einmal überschritten wird. In diesem Fall müsste man dann den Programmcode anpassen.

Die Lösung ist die Programmierung eines Arrays, das seine Größe verändern kann. Da eine solche Lösung sehr häufig benötigt wird, ist es sinnvoll, dies in einer Bibliothek (in Java in der Collection-API) allen Programmierern zur Verfügung zu stellen.

Java liefert bereits fertige Collection-Klassen wie Queues, Stacks oder verschiedene Ausprägungen von Listen in seiner Collection-API.

Einer der Vorteile der Collections gegenüber der Verwendung von Arrays ist, dass die Größe einer Collection nicht angegeben werden muss. Eine Collection erweitert sich selbstständig bei Bedarf.

Collections können ähnlich wie Arrays verwendet werden. Sie haben aber im Gegensatz zu diesen keine feste Größe, sondern können dynamisch wachsen und schrumpfen. Sie werden deshalb als **dynamische Datenstrukturen** bezeichnet.

Je nach Anwendungsproblem sollen in einer Collection-Klasse andere Typen von Objekten (Fahrzeuge, Mitarbeiter, Studenten, ...) verwaltet werden. Um Objekte beliebigen Typs in einer Collection-Klasse zu verwalten, könnte auf den Typ `Object` zurückgegriffen werden. Vor der Version Java 5 war genau dies der Fall. Eine Methode `add()` für eine Collection-Klasse besaß einen Übergabeparameter vom Typ `Object`, um eine Referenz auf ein beliebiges Objekt der Collection übergeben und darin speichern zu können.

Damit ein Programmierer präziser festlegen kann, von welchem Typ Objekte sein dürfen, die durch eine Collection verwaltet werden, sind seit der Version Java 5 alle Klassen und Schnittstellen der **Collection-API generisch**.

Collection-Klassen können deshalb für bestimmte konkrete Datentypen instanziiert werden. Damit bieten diese Datenstrukturen die Sicherheit, dass nur Objekte dieses Typs bzw. davon abgeleiteter Klassen in der entsprechenden Datenstruktur gespeichert werden können. Das heißt, der Programmierer legt beispielsweise beim Erzeugen der Collection-Klasse fest, dass Objekte vom Typ `Student` verwaltet werden sollen.

18.1.1 Verwendung der Collection-API

Bevor im nächsten Kapitel die grundlegende Struktur der Collection-API vorgestellt wird, soll zuerst in einem kleinen Beispiel gezeigt werden, wie eine Collection-Klasse verwendet werden kann. Eine Firma möchte die Fahrzeuge ihres Fuhrparkes ver-

walten. Da es sich hier um einen Datenbestand handelt, der sowohl wachsen als auch schrumpfen kann, wird als Datenstruktur beispielhaft die Collection-Klasse `Vector` aus dem Paket `java.util` gewählt. Zuerst wird eine Klasse `Fahrzeug` programmiert, die Fahrzeuge einfachheitshalber durch nur zwei Datenfelder beschreibt. Ergänzend wird die `toString()`-Methode der Klasse `Object` überschrieben, um in der folgenden Klasse `Fuhrpark` eine einfache Ausgabemöglichkeit unter Verwendung der Funktion `System.out.println()` zu implementieren. Zuerst die Klasse `Fahrzeug`:

```
// Datei: Fahrzeug.java

public class Fahrzeug
{
   private String kennzeichen;
   private int anzahlSitze;
   // ... weitere Datenfelder

   public Fahrzeug (String kennzeichen, int anzahlSitze)
   {
      this.kennzeichen = kennzeichen;
      this.anzahlSitze = anzahlSitze;
   }

   public String toString()
   {
      return "Kennzeichen: " + kennzeichen
           + " Sitzplaetze: " + anzahlSitze;
   }
}
```

Die Klasse `Fuhrpark` besitzt die Methoden `aufnehmen()`, `ausbuchen()` und `ausgeben()`. Die Stellen im Programmcode, in denen die Collection-Klasse `Vector` verwendet wird, sind fett hervorgehoben:

```
// Datei: Fuhrpark.java
import java.util.Vector;

public class Fuhrpark
{
   private Vector<Fahrzeug> fahrzeuge = new Vector<Fahrzeug>();

   public void aufnehmen (Fahrzeug ref)
   {
      fahrzeuge.add (ref);
   }

   public void ausbuchen (Fahrzeug ref)
   {
      fahrzeuge.remove (ref);
   }

   public void ausgeben()
   {
      int anzahl = fahrzeuge.size();
      for (int i = 0; i < anzahl; i++)
```

```
      {
         Fahrzeug ref = fahrzeuge.get (i);
         System.out.println (ref); // Aufruf toString()-Methode
      }
   }
}
```

Der privaten Instanzvariablen `fahrzeuge` wird ein Objekt vom Typ `Vector<Fahrzeug>` zugewiesen. Durch Angabe des aktuellen Typparameters `Fahrzeug` wird sichergestellt, dass in die Collection nur Referenzvariablen auf Objekte vom Typ `Fahrzeug` eingefügt werden können. Durch Aufruf der Methode `add()` wird eine Referenz auf ein Objekt hinzugefügt, durch Aufruf der Methode `remove()` wird die übergebene Referenz wieder entfernt, durch Aufruf der Methode `size()` wird die Größe der Collection zurückgegeben und durch Aufruf der Methode `get()` wird die Referenz auf das Objekt an der übergebenen Position zurückgegeben. Neben den Methoden `add()`, `remove()`, `size()` und `get()` besitzt eine Collection-Klasse noch viele weitere nützliche Methoden.

Die folgende Klasse `FuhrparkTest` erzeugt 3 Fahrzeuge, fügt diese dem Fuhrpark (und damit der Collection) hinzu und gibt durch Aufruf der Methode `ausgeben()` die Fahrzeuge in einer Übersicht aus. Danach wird das „mittlere“ Fahrzeug entfernt und erneut der Fuhrpark ausgegeben:

```
// Datei: FuhrparkTest.java

public class FuhrparkTest
{
   public static void main (String[] argv)
   {
      Fuhrpark fuhrpark = new Fuhrpark();
      Fahrzeug f1 = new Fahrzeug ("S AB 123", 5);
      fuhrpark.aufnehmen (f1);
      Fahrzeug f2 = new Fahrzeug ("S XY  45", 4);
      fuhrpark.aufnehmen (f2);
      Fahrzeug f3 = new Fahrzeug ("S QI 456", 7);
      fuhrpark.aufnehmen (f3);
      fuhrpark.ausgeben();

      System.out.println ("Fahrzeug 2 ausbuchen!");
      fuhrpark.ausbuchen (f2);
      fuhrpark.ausgeben();
   }
}
```

Die Ausgabe des Programms ist:

```
Kennzeichen: S AB 123 Sitzplaetze: 5
Kennzeichen: S XY  45 Sitzplaetze: 4
Kennzeichen: S QI 456 Sitzplaetze: 7
Fahrzeug 2 ausbuchen!
Kennzeichen: S AB 123 Sitzplaetze: 5
Kennzeichen: S QI 456 Sitzplaetze: 7
```

18.1.2 Einblick in die Struktur der Collection-API

In diesem Kapitel soll ein Überblick über die in der Java-Klassenbibliothek verfügbaren Collections gegeben werden.

Das Paket `java.util` enthält vier verschiedene Familien von Collections:

- Listen (Lists),
- Warteschlangen (Queues),
- Mengen (Sets)
- und Verzeichnisse (Maps).

Zusätzlich zu den im Paket `java.util` enthaltenen Collections gibt es seit der Version Java 5 auch thread-safe Collections im Paket `java.util.concurrent`.

Zu jeder der Collection-Familien Liste, Warteschlange, Menge und Verzeichnis existiert eine **Schnittstelle** sowie eine **abstrakte Basisklasse**, welche diese **Schnittstellen implementiert** und als **Basis für konkrete Collection-Klassen** dient.

Diese **Aufteilung** zwischen Schnittstelle und Implementierung hat den folgenden Hintergrund: Die **Schnittstelle** definiert, was man mit der Collection machen kann. Sie legt also die **funktionalen Eigenschaften** fest. Die dahinter liegende **Implementierung** bestimmt weitere **nicht funktionale Eigenschaften** wie z. B. die Geschwindigkeit, mit der auf Daten zugegriffen werden kann. Funktionale Eigenschaften einer Collection, d. h. wie auf sie zugegriffen werden kann, sind geordnet/ungeordnet, sortiert/unsortiert und sequenziell/wahlfrei.

Bild 18-1 zeigt einen Auszug der wichtigsten Collection-Schnittstellen und deren Beziehungen:

Bild 18-1 Collection-Schnittstellen in `java.util`

Auf die generellen Eigenschaften von Listen, Warteschlangen, Mengen und Verzeichnisse wird im Folgenden kurz eingegangen. Hierbei wird auf die zugehörigen Schnittstellen und Klassen in der Collection-API Bezug genommen.

Listen sind geordnete Collections, die mittels Indizes einen beliebigen Zugriff (wahlfreier Zugriff, engl. random access) auf ein Element innerhalb einer Collection ermöglichen – analog zu den Arrays. Im Unterschied zu Arrays können Elemente an beliebiger Stelle zusätzlich eingefügt und auch gelöscht werden.

Dabei werden die bereits existierenden Elemente der Liste automatisch verschoben. Die dazu notwendige Rechenzeit hängt von der zugrundeliegenden Implementierung ab.

Die zugehörige Funktionalität einer Liste wird durch die Schnittstelle `List<E>` bereitgestellt. Die Collection-Klassen `ArrayList`, `LinkedList`, `Vector` und `Stack` implementieren die Schnittstelle `List<E>`.

Eine **Warteschlange** bzw. eine **Queue** ermöglicht ebenfalls die Aufbewahrung von Elementen. Die Besonderheit einer Queue ist die Regelung des Zugriffs. Typischerweise verwenden Queues das Zugriffsprinzip "First In-First Out" (abgekürzt FIFO).

Eine Queue bietet keinen wahlfreien Zugriff. In **prioritätsbasierten Queues** kann gesteuert werden, wie neue Einträge die vorhandenen Einträge innerhalb der Warteschlange überholen dürfen.

Die zugehörige Funktionalität einer Queue wird durch die Schnittstelle `Queue<E>` bereitgestellt. Die Collection-Klassen `LinkedList` und `PriorityQueue` stellen die Funktionalität der Schnittstelle `Queue<E>` zur Verfügung. Da eine Queue im Grunde genommen nichts anderes als eine Liste ist, die typischerweise nach dem FIFO-Prinzip abgearbeitet wird, kommt es auch vor, dass Collection-Klassen sowohl die Schnittstelle `List<E>` als auch die Schnittstelle `Queue<E>` implementieren. Dies ist beispielsweise bei der Collection-Klasse `LinkedList` der Fall.

Eine **Menge** oder ein **Set** ist eine ungeordnete Menge von Objekten, wobei jedes dieser Objekte nur ein einziges Mal vorkommen darf.

Der Zugriff beschränkt sich bei diesen Collections im Wesentlichen darauf, Objekte hinzuzufügen oder zu entfernen bzw. zu prüfen, ob ein Objekt enthalten ist oder nicht.

Die zugehörige Funktionalität einer Menge wird durch die Schnittstelle `Set<E>` bereitgestellt. Die Collection-Klassen `HashSet` und `TreeSet` stellen die Funktionalität der Schnittstelle `Set<E>` zur Verfügung.

Bei **Verzeichnissen** (**Maps**) wird statt über einen numerischen Index über ein **eindeutiges** Schlüsselobjekt auf Werte zugegriffen. In einer Map werden also **Schlüssel-Werte-Paare** repräsentiert.

Dies kommt durch die alternative Bezeichnung **Dictionary** zum Ausdruck. Die Typen für Schlüssel und Wert sind beliebig. Da der Zugriff nur über den Schlüssel erfolgt, muss der Wert nicht eindeutig sein. Auch die Reihenfolge des Einfügens von Elementen in eine Map spielt keine Rolle, da diese im Gegensatz zu einer Liste ungeordnet ist.

Die Funktionalität eines Verzeichnisses wird durch die Schnittstelle `Map<K,V>` bereitgestellt. Die Schnittstelle `Map<K,V>` definiert Eigenschaften von assoziativen Speichern, die Schlüssel(wörter) auf Werte abbilden. Damit das funktioniert, darf jeder Schlüssel nur ein einziges Mal vorkommen, Werte hingegen auch mehrfach. Zu beachten ist, dass die Schnittstelle `Map<K,V>` nicht von der Schnittstelle `Collection<E>` abgeleitet ist! Der Grund hierfür ist, dass beim Einfügen von Elementen in eine Map stets zwei Elemente auf einmal eingefügt werden und zwar Schlüssel und Wert, was die Schnittstelle `Collection<E>` nicht vorsieht.

Die Collection-Klassen `HashMap` und `TreeMap` stellen die Funktionalität der Schnittstelle `Map<K,V>` zur Verfügung.

Tabelle 18-1 zeigt die vorgestellten **Collection-Klassen** und deren **Eigenschaften**:

		Organisation	Freiheit beim Zugriff	Zugriff	Duplikate zugelassen	Besondere Eigenschaften
Listen	ArrayList	geordnet	wahlfrei	wahlfrei über Index	ja	schnelles Lesen
	LinkedList	geordnet	wahlfrei	wahlfrei über Index	ja	schnelles Einfügen
	Vector	geordnet	wahlfrei	wahlfrei über Index	ja	synchronisiert
	Stack	geordnet	sequenziell	sequenziell (letztes Element)	ja	LIFO
Queues	LinkedList	geordnet	sequenziell	sequenziell, nächstes Element	ja	schnelles Einfügen (am Rand)
	PriorityQueue	sortiert	sequenziell	sequenziell, nächstes Element	ja	-
Sets	HashSet	ungeordnet	wahlfrei	ungeordnet, Test auf Existenz	nein	schnelles Lesen
	TreeSet	sortiert	wahlfrei	sortiert, Test auf Existenz	nein	-
Maps	HashMap	ungeordnet	wahlfrei	wahlfrei über Schlüssel	Schlüssel nein Werte ja	schnelles Lesen
	TreeMap	sortiert	wahlfrei	wahlfrei über Schlüssel	Schlüssel nein Werte ja	-

Tabelle 18-1 Übersicht über konkrete Collection-Klassen im Paket `java.util`

Die Schnittstelle `Collection<E>` bildet die Wurzel der Schnittstellen der Collection-API. Die Schnittstelle `Collection<E>` wird nicht direkt in einer Collection-Klasse implementiert, jedoch die davon abgeleiteten Schnittstellen wie `Set<E>`, `List<E>` und `Queue<E>`. Mit Ausnahme der assoziativen Arrays wie z. B. `HashMap<K,V>` implementieren damit alle Collection-Klassen die Schnittstelle `Collection<E>`.

Die Schnittstelle `Iterable<E>` ist eine Schnittstelle für alle Klassen, über die man iterieren[220] können soll. Collection-Klassen sind dafür ein typisches Beispiel. Trotzdem ist diese Schnittstelle nicht Bestandteil des Pakets `java.util`, sondern des Pakets `java.lang`, da sie noch eine besondere Bedeutung im Zusammenhang mit der Variante der erweiterten `for`-Schleife zur Iteration über Elemente einer Datenstruktur hat, d. h. sie ist Bestandteil des Sprachkerns und erlaubt die Implementierung von eigenen Klassen, die mit der erweiterten `for`-Schleife funktionieren sollen.

18.1.3 Programmieren gegen die Schnittstelle

Um flexibel zu sein, stehen in Java zu jeder Art von Collection-Schnittstelle verschiedene Klassen als Implementierung zur Verfügung.

Die Kunst bei der Programmierung besteht darin, für jedes Problem zuerst anhand der geforderten Funktionalität die richtige Schnittstelle und dann dazu gemäß der Leistungsanforderungen die richtige Klasse auszuwählen.

Fehlentscheidungen sind dabei natürlich möglich, müssen jedoch bei geschickter Umsetzung nicht endgültig sein. Der Trick besteht darin, stets **"gegen die Schnittstelle"** zu **programmieren**, d. h. alle Variablen, Parameter und Rückgabewerte stets mit Hilfe des Schnittstellentyps zu vereinbaren. Diesem Schnittstellentyp kann dann ein Objekt einer beliebigen Collection-Klasse, die diese Schnittstelle implementiert, zugewiesen werden. Stellt sich später heraus, dass eine ungünstige Collection-Klasse gewählt wurde, kann bei der Instanziierung des Objekts eine andere Collection-Klasse verwendet werden, ohne dass der übrige Code geändert werden muss, da jede dieser Klassen die Collection-Schnittstelle implementiert.

Die Collection-Schnittstellen sind das Herzstück des Collection Frameworks. Ihre Hauptaufgabe besteht darin, es dem Programmierer zu ermöglichen, flexibel mit unterschiedlichen Implementierungen einer bestimmten Art von Collections zu arbeiten.

Die Klasse Fuhrpark aus Kapitel 18.1.1 soll nun derart angepasst werden, dass die Instanzvariable `fahrzeuge` nicht vom Typ `Vector<Fahrzeug>`, sondern vom Schnittstellentyp `List<Fahrzeug>` ist. Damit kann der Referenzvariablen `fahrzeug` entweder ein Objekt vom Typ `ArrayList<Fahrzeug>`, `Vector<Fahr-`

[220] Iterieren bedeutet, dass **schrittweise auf jedes Objekt zugegriffen** wird, das von der Collection aus referenziert wird.

zeug>, LinkedList<Fahrzeug> oder Stack<Fahrzeug> zugewiesen werden. Dies ist deshalb möglich, da die Collection-Klassen ArrayList, Vector, LinkedList und Stack die Collection-Schnittstelle List implementieren (siehe hierzu Tabelle 18-1). Im folgenden Programmcode der Klasse Fuhrpark wurde exemplarisch ein Objekt der Collection-Klasse ArrayList zugewiesen. Die notwendigen Änderungen sind fett hervorgehoben:

```
// Datei: Fuhrpark2.java
import java.util.List;
import java.util.ArrayList;

public class Fuhrpark2
{
   private List<Fahrzeug> fahrzeuge = new ArrayList<Fahrzeug>();

   public void aufnehmen (Fahrzeug ref)
   {
      fahrzeuge.add (ref);
   }

   public void ausbuchen (Fahrzeug ref)
   {
      fahrzeuge.remove (ref);
   }

   public void ausgeben()
   {
      int anzahl = fahrzeuge.size();
      for (int i = 0; i < anzahl; i++)
      {
         Fahrzeug ref = fahrzeuge.get (i);
         System.out.println (ref);
      }
   }
}
```

Im Umkehrschluss bedeutet dies, dass die Methoden add(), remove(), size() und get(), welche zur Referenzvariablen fahrzeuge aufgerufen werden, bereits in der Collection-Schnittstelle List definiert sein müssen. Dies ist auch der Fall. Betrachtet man Bild 18-1, so könnte man auf die Idee kommen, noch „flexibler“ zu programmieren, da die Collection-Schnittstelle List von der Collection-Schnittstelle Collection abgeleitet ist. Dies ist im vorliegenden Beispiel der Klasse Fuhrpark jedoch nicht möglich, da die Methode get() – wie diese in der Methode ausgeben() aufgerufen wird, erst in der Collection-Schnittstelle List – und nicht bereits in der Collection-Schnittstelle Collection – definiert wird.

18.2 Eigenschaften und Fähigkeiten der Collections

In Kapitel 18.2.1 wird nochmals kurz auf die Generizität von Collections eingegangen und in Kapitel 18.2.2 wird gezeigt, wie Objekte von Collection-Klassen dazu genutzt werden können, mittels Auto-Boxing und Auto-Unboxing, Variablen einfacher Datentypen zu verwalten. Kapitel 18.2.3 zeigt wie mit Hilfe der Schnittstelle Itera-

tor<E> über die Elemente eines Objekts vom Typ Collection iteriert werden kann.

18.2.1 Generizität von Collections

Wie bereits erwähnt, sind alle Collection-Klassen und -Schnittstellen generisch. Das heißt, der Programmierer bestimmt durch Angabe eines aktuellen Typparameters beim Definieren von Referenzvariablen und beim Erzeugen von Objekten, welche Objekte durch die Collection verwaltet werden sollen.

Das folgende Beispiel zeigt die Angabe eines aktuellen Typ-Parameters vom Typ String:

```
Collection<String> coll = new ArrayList<String>();
```

Durch Angabe des aktuellen Typ-Parameters String wird sichergestellt, dass der Collection nur Referenzen auf Objekte vom Typ String hinzugefügt werden können.

Der aktuelle Typ-Parameter kann auch – wie in folgendem Beispiel – weggelassen werden:

```
Collection coll = newArrayList();
```

Wird kein aktueller Typ-Parameter beim Definieren von Referenzvariablen und beim Anlegen von Objekten einer Collection-Klassen angegeben, so wird implizit die Klasse Object verwendet.

Die Anweisung entspricht also der folgenden:

```
Collection<Object> coll = new ArrayList<Object>;
```

Durch eine solche Collection können dann Referenzen eines beliebigen Typs verwaltet werden. Generell sollte der Programmierer immer durch einen aktuellen Typ-Parameter angeben, welche Referenzen von welchem Typ durch die Collection verwaltet werden sollen.

Bei der Verwendung von Collections, die ohne aktuellen Typ-Parameter angelegt werden, gibt der Compiler deshalb folgende Warnung aus:

```
Note: <Dateiname>.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Der Compiler signalisiert damit, dass der übersetzte Quellcode Abschnitte enthält, die bei falscher Handhabung Probleme hervorrufen können.

18.2.2 Auto-Boxing und Auto-Unboxing bei Collections

Ein Objekt einer Collection-Klasse ermöglicht die Verwaltung von Referenzen auf Objekte. Variablen einfacher Datentypen können ebenfalls durch eine Collection verwaltet werden – für diese Fälle wird der Auto-Boxing bzw. Auto-Unboxing-

Mechanismus verwendet. Hierzu wird als aktueller Typ-Parameter – beim Definieren von Referenzvariablen und beim Erzeugen eines Objektes einer Collection-Klasse – die dem einfachen Datentyp entsprechende Wrapper-Klasse verwendet. Das folgende Beispiel zeigt dies anhand der Wrapper-Klasse `Integer`:

```
List<Integer> c = new ArrayList<Integer>();
```

Der Collection kann dann einfach eine Variable vom einfachen Datentyp `int` hinzugefügt werden. Beispielsweise wie folgt durch Aufruf der Methode `add()`:

```
c.add (5);          // 5 wird automatisch eingepackt
```

Entsprechend können die Werte wieder aus der Collection ausgelesen werden. Beispielsweise wie folgt durch Aufruf der Methode `get()`:

```
int i = c.get (0) // wird automatisch ausgepackt
```

18.2.3 Iterieren über Collections

Neben den verschiedenen Möglichkeiten, auf einzelne Elemente zuzugreifen, bieten Collections die Möglichkeit, alle ihre Elemente in einer Schleife zu bearbeiten. Der Schlüssel dazu sind sogenannte Iteratoren.

Iteratoren verkörpern das Prinzip, dass auf eine Collection zugegriffen werden kann, ohne den speziellen Typ der Collection zu kennen. Ein Iterator erlaubt es, über alle Elemente der Collection zu laufen, sowie festzustellen, ob noch ein weiteres Element in der Collection ist.

Die Methode `ausgeben()` der Klasse `Fuhrpark2` aus Kapitel 18.1.3 soll derart angepasst werden, dass die Ausgabe mittels eines Iterators erfolgt. Die folgende Klasse `Fuhrpark3` zeigt die angepasste Variante:

```
// Datei: Fuhrpark3.java
import java.util.Collection;
import java.util.ArrayList;
import java.util.Iterator;

public class Fuhrpark3
{
   private Collection<Fahrzeug> fahrzeuge=new ArrayList<Fahrzeug>();

   public void aufnehmen (Fahrzeug ref)
   {
      fahrzeuge.add (ref);
   }

   public void ausbuchen (Fahrzeug ref)
   {
      fahrzeuge.remove (ref);
   }

   public void ausgeben()
   {
```

```
        Iterator<Fahrzeug> iter = fahrzeuge.iterator();

        // Solange noch ein Element in der Collection enthalten ist
        while (iter.hasNext())
        {
           // lies das Element aus der Collection und rufe dessen
           // toString()-Methode auf.
           System.out.println (iter.next());
        }
    }
}
```

Die Referenzvariable `fahrzeuge` ist in diesem Beispiel vom Typ `Collection<Fahrzeug>`. Dies ist deshalb möglich, da alle Methoden, die zur Referenzvariablen `fahrzeuge` aufgerufen werden, bereits in der Collection-Schnittstelle `Collection` definiert sind. Damit konnte durch Einsatz des Iterators eine noch „flexiblere“ Variante programmiert werden.

Der Laufvariablen eines Arrays entspricht bei einer Collection die Referenz `iter` auf den Iterator. Diese Referenz stellt gewissermaßen einen "Zeiger" dar, welcher über die Elemente der Collection gleitet und die jeweils aktuelle Position markiert (ein wahlfreier Zugriff via Index ist ja nicht bei allen Collections möglich!).

Als Abbruchbedingung für einen Durchlauf mit Hilfe des Iterators dient der Rückgabewert der Methode **`hasNext()`**, der anzeigt, ob noch weitere Elemente vorhanden sind.

Zum Zugriff auf ein Element wird anstelle des Array-Index bei Verwendung eines Iterators die Methode **`next()`** verwendet. Sie liefert das aktuelle Element zurück und stellt **gleichzeitig** den Iterator auf das nächste Element.

Die Methoden `hasNext()` und `next()` des Iterators sind in der Schnittstelle `Iterator<E>` deklariert. Jede Collection-Klasse hat ihre eigene Iterator-Klasse, welche diese Schnittstelle implementiert. Diese Iterator-Klasse verbirgt die Implementierungsdetails der Collection und stellt dem Benutzer der Collection eine standardisierte Schnittstelle zum Zugriff auf Elemente zur Verfügung.

Die im Interface `Collection` deklarierte Methode `iterator()` ist dabei in jeder konkreten Collection-Klasse so implementiert, dass sie ein Objekt zurückliefert, dessen Klasse das Interface `Iterator<E>` implementiert und über die Elemente der jeweiligen Collection-Klasse iterieren kann.

Die eigentliche Iterator-Klasse wird nie sichtbar, da auch hier stets "gegen die Schnittstelle" programmiert wird.

Seit der Version Java 5 kann über Collections – wie bei Arrays – mit der **erweiterten `for`-Schleife** (**for-each-Schleife**) iteriert werden. Die Methode `ausgeben()` der folgenden Klasse `Fuhrpark4` gibt die Elemente der Collection mittels der erweiterten `for`-Schleife aus:

```
// Datei: Fuhrpark4.java

import java.util.Collection;
import java.util.ArrayList;

public class Fuhrpark4
{
   private Collection<Fahrzeug> fahrzeuge=new ArrayList<Fahrzeug>();

   public void aufnehmen (Fahrzeug ref)
   {
      fahrzeuge.add (ref);
   }

   public void ausbuchen (Fahrzeug ref)
   {
      fahrzeuge.remove (ref);
   }

   public void ausgeben()
   {
      for (Fahrzeug element : fahrzeuge)
         System.out.println (element);
   }
}
```

Die for-each-Schleife verwendet dabei – für den Programmierer unsichtbar – ebenfalls einen Iterator, auf den über die Schnittstelle `Iterable` zugegriffen wird.

Die for-each-Schleife ist zwar einfacher zu schreiben, aber schwerer zu verstehen, da man nicht mehr sieht, was passiert. Der Iterator tritt nun nämlich gar nicht mehr in Erscheinung – er wird im Verborgenen verwendet.

18.3 Zusammenfassung

Klassen für Datenstrukturen mit speziellen Fähigkeiten nennt man in Java Collections. Für alle Datenbestände, deren Größe sich ändern kann, ist der Einsatz einer Collection-Klasse gegenüber einem Array fester Länge zu empfehlen. Die zentralen Klassen der Collection-API sind im Paket `java.util` zu finden. Es gibt vier verschiedene Collection-Familien: Listen, Warteschlangen, Mengen und Verzeichnisse (siehe Kapitel **18.1**).

Listen sind geordnete Collections, die mittels Indizes einen beliebigen Zugriff (wahlfreier Zugriff, engl. randomaccess) auf ein Element innerhalb einer Collection ermöglichen – analog zu den Arrays. Im Unterschied zu Arrays können Elemente an beliebiger Stelle zusätzlich eingefügt und auch gelöscht werden.

Eine Warteschlange bzw. eine Queue ermöglicht ebenfalls die Aufbewahrung von Elementen. Die Besonderheit einer Queue ist die spezielle Regelung des Zugriffs. Typischerweise verwenden Queues das Zugriffsprinzip "First In-First Out" (abgekürzt FIFO).

Eine Menge oder ein Set ist eine ungeordnete Menge von Objekten, wobei jedes dieser Objekte nur ein einziges Mal vorkommen darf.

Bei Verzeichnissen (Maps) wird statt über einen numerischen Index über ein eindeutiges Schlüsselobjekt auf Werte zugegriffen. In einer Map werden also Schlüssel-Werte-Paare repräsentiert.

Die Funktionalität einer Collection-Familie wird in einer zugehörigen Schnittstelle definiert. Die zugehörige Funktionalität einer Liste wird beispielsweise durch die Schnittstelle `List<E>` bereit gestellt und die Collection-Klassen `ArrayList`, `LinkedList`, `Vector` und `Stack` implementieren die Schnittstelle `List<E>`.

Die Kunst bei der Programmierung besteht darin, für jedes Problem zuerst anhand der geforderten Funktionalität die richtige Schnittstelle und dann dazu gemäß der Leistungsanforderungen die richtige Klasse auszuwählen.

Damit der Programmierer präziser festlegen kann, von welchem Typ Objekte sein dürfen, die durch eine Collection verwaltet werden, sind seit der Version Java 5 alle Klassen und Schnittstellen der Collection-API generisch (siehe Kapitel **18.2**). Das heißt, der Programmierer bestimmt durch Angabe eines aktuellen Typparameters – beim Definieren von Referenzvariablen und beim Erzeugen von Objekten – welche Objekte durch die Collection verwaltet werden sollen.

Ein Objekt einer Collection-Klasse ermöglicht die Verwaltung von Referenzen auf Objekte. Variablen einfacher Datentypen können ebenfalls durch eine Collection verwaltet werden – für diese Fälle wird der Auto-Boxing bzw. Auto-Unboxing-Mechanismus verwendet.

Die Elemente einer Collection können mit einer Schleife bearbeitet werden. Hierzu kommen so genannte Iteratoren zum Einsatz. Iteratoren verkörpern das Prinzip, dass auf eine Collection zugegriffen werden kann, ohne den speziellen Typ der Collection zu kennen. Ein Iterator erlaubt es, über alle Elemente der Collection zu laufen, sowie festzustellen, ob noch ein weiteres Element in der Collection ist.

Seit der Version Java 5 kann über Collections – wie bei Arrays – auch mit der erweiterten `for`-Schleife (for-each-Schleife) iteriert werden. Im Hintergrund kommt hierbei ebenfalls ein Iterator zum Einsatz.

18.4 Übungen

Aufgabe 17.1: Flughafenprojekt – ArrayList

Bisher wurden die bereits eingegebenen Fluggesellschaften, Flugzeugtypen und die Flugzeuge nicht im System zur erneuten Verwendung gehalten. Dies soll nun geändert werden. Hierzu soll die Klasse `Flughafen` um die drei Instanzvariablen `flug-`

`zeuge`, `flugzeugtypen` und `fluggesellschaften` vom Typ `ArrayList` erweitert werden. Auch soll der Flugzeugsimulator so abgeändert werden, dass er nicht nur ein Flugzeug, sondern beliebig viele gleichzeitig verwalten kann. Zusätzlich soll dem Benutzer die Möglichkeit gegeben werden, entweder einen bereits eingegebenen Flugzeugtyp bzw. eine bereits eingegebene Fluggesellschaft auszuwählen oder einen neuen Flugzeugtyp bzw. eine neue Fluggesellschaft einzugeben.

Kapitel 19

Ein-/Ausgabe und Streams

19.1 Klassifizierung von Streams
19.2 Das Stream-Konzept
19.3 Bytestream-Klassen
19.4 Characterstream-Klassen
19.5 Standardeingabe und Standardausgabe
19.6 Ein Beispiel für die Anwendung von Stream-Klassen
19.7 Ein- und Ausgabe der Datenfelder von Objekten
19.8 Zusammenfassung
19.9 Übungen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1_19

19 Ein-/Ausgabe und Streams

Programme laufen fast immer nach dem Schema "Eingabe/Verarbeitung/Ausgabe" ab. Schon daran ist die Wichtigkeit der Ein- und Ausgabe in Programmen zu erkennen. Ein **gutes Ein- und Ausgabe-System** ist für die Akzeptanz einer Programmiersprache von großer Bedeutung, ist aber jedoch nicht einfach zu entwerfen.

Einerseits gibt es sehr viele **Datenquellen** (Tastatur, Dateien, Netzverbindungen), von denen Eingaben gelesen werden, und **Datensenken** (Bildschirme, Dateien, Netzverbindungen), in welche die Ausgaben geschrieben werden, andererseits sollen mit diesen Datenquellen und Datensenken Informationen auch in verschiedenen Einheiten wie z. B. Bytes, Zeichen oder Zeilen ausgetauscht werden.

In Java ist die Ein-/Ausgabe sequenziell. Hierfür wird das sogenannte **Stream-Konzept** verwendet. Ein Stream ist eine **geordnete Folge von Bytes**, ein sogenannter **Bytestrom**.

Solche Byteströme sind meist von **unbekannter Länge**, d. h. die Anzahl der Bytes, die ein Bytestrom transportieren wird, ist im Voraus nicht ermittelbar. Für die Umsetzung des Stream-Konzeptes gibt es in Java die sogenannten **Stream-Klassen** aus dem Paket `java.io`, welche die Funktionen für die Ein- und Ausgabe implementieren.

In Kapitel 19.1 werden Streams in unterschiedliche Arten klassifiziert und die Unterschiede zwischen Eingabe-, Ausgabe- und Verarbeitungs-Streams bzw. zwischen Byte- und Character-Streams werden hervorgehoben. Das Stream-Konzept wird in Kapitel 19.2 erläutert. Die Bytestream-Klassen werden in Kapitel 19.3 und die Characterstream-Klassen in Kapitel 19.4 beschrieben. Die Verwendung der Standardeingabe und der Standardausgabe wird in Kapitel 19.5 erklärt. In Kapitel 19.6 soll anhand eines Beispiels die Verwendung der Stream-Klassen verdeutlicht werden. Abschließend wird in Kapitel 19.7 die Ein- und Ausgabe der Datenfelder von Objekten gezeigt.

19.1 Klassifizierung von Streams

Die Stream-Klassen, die Java zur Verfügung stellt, befinden sich fast alle im Paket `java.io` – einem der größten Pakete der Java-API. Um sich einen Überblick zu verschaffen, ist es hilfreich, zuallererst eine **Klassifizierung der Stream-Klassen** durchzuführen.

Stream-Klassen können auf mehrere Arten klassifiziert werden. Die erste Art ist anhand der **Klassenhierarchie** des Pakets `java.io`. Dadurch erhält man eine Aufteilung in **Byte- und Characterstream-Klassen**.

Die zweite Art, Stream-Klassen zu klassifizieren, ist nach funktionalen Kritieren:

- **in Datensenken schreiben** (**Sinkstream-Klassen**),
- **aus Datenquellen lesen** (**Springstream-Klassen**),
- eine Sink- bzw. Springstream-Klasse benutzen und deren **Funktionalität erweitern** (**Processingstream-Klassen**).

Ein Bytestrom, der aus einer Datenquelle kommt, wird als **Eingabestrom bzw. Inputstream** bezeichnet. Ein Bytestrom, der in eine Datensenke hineingeht, wird als **Ausgabestrom bzw. Outputstream** bezeichnet.

19.1.1 Datensenken und -quellen

Das folgende Bild zeigt Datenquellen und -senken in Java:

Bild 19-1 Datensenken und Datenquellen

In Java können **Dateien**, **Pipes**[221], **`byte`-Arrays**, **`char`-Arrays**, **Strings** und **Netzverbindungen** Datenquellen oder Datensenken sein.

Dabei stellen `byte`-Arrays, `char`-Arrays, Strings und Pipes **interne Datenquellen bzw. interne Datensenken** dar, d. h. sie befinden sich innerhalb eines Java-Programms. Dateien und Netzverbindungen hingegen sind **externe Datenquellen bzw. externe Datensenken**, d. h. sie befinden sich außerhalb eines Java-Programms.

Klassen, die einen Bytestrom von einer Datenquelle einlesen können, bezeichnet man als **Eingabestrom-Klassen**. Klassen, die einen Bytestrom in eine Datensenke schreiben können, bezeichnet man als **Ausgabestrom-Klassen**.

Mit der Notation :`Ausgabestrom`-Klasse wird ein Objekt der Klasse `Ausgabestrom`-Klasse dargestellt. Dabei ist die Klasse Ausgabestrom-Klasse keine existierende Klasse in Java – sie steht stellvertretend für die später vorgestellten Ausgabestrom-Klassen von Java.

Im Bild sind links Instanzen von Ausgabestrom-Klassen zu sehen, die geordnete Bytefolgen in externe und interne Datensenken schreiben. Objekte von Ausgabestrom-Klassen werden von anderen Objekten eines Java-Programms benutzt, um Daten in Datensenken zu schreiben. Rechts sind die entsprechenden Objekte der Eingabestrom-Klassen zu sehen, die geordnete Bytefolgen aus internen und externen Datenquellen lesen. Objekte von Eingabestrom-Klassen werden von anderen Objekten eines Java-Programms benutzt, um Eingaben aus einer Datenquelle zu lesen.

19.1.2 Byte- und Characterstreams

Alle Stream-Klassen des Pakets `java.io` sind von einer der vier abstrakten Basisklassen `InputStream`, `OutputStream`, `Reader` oder `Writer` abgeleitet.

Klassen, die von den Klassen `InputStream` oder `OutputStream` abgeleitet sind, nennt man **Bytestream-Klassen** – sie sind hauptsächlich für die **Verarbeitung einzelner Bytes** verantwortlich.

Klassen, die von den Klassen `Reader` oder `Writer` abgeleitet sind, nennt man **Characterstream-Klassen**. Diese sind für die korrekte **Verarbeitung von Zeichen** verantwortlich.

[221] Eine Pipe ist in Java ein Mechanismus zur Interprozesskommunikation zwischen Threads (siehe Kapitel 16). Die eine Seite der Pipe wird als Datensenke verwendet – auf dieser Seite kann ein Programm Bytes in die Pipe hineinschreiben. Die andere Seite der Pipe wird als Datenquelle verwendet – auf dieser Seite kann ein Programm Bytes aus der Pipe lesen.

Innerhalb der Byte- und Characterstream-Klassen gibt es die sogenannten **Inputstream-Klassen**, die Daten lesen, und die sogenannten **Outputstream-Klassen**, die Daten schreiben. Somit ergibt sich folgende Klassifizierung:

Bild 19-2 Klassifizierung der Stream-Klassen

Bytes und Characters in Java

Objekte von **Bytestream-Klassen** schreiben und lesen **Bytes**, also Werte vom Typ `byte`. Objekte von **Characterstream-Klassen** schreiben und lesen **Characters**, also Zeichen vom Typ `char`. Bei den meisten Programmiersprachen wird ein Zeichen vom Typ `char` durch ein Byte dargestellt. In Java werden jedoch Zeichen des Typs `char` in der UTF-16 Codierung des **Unicodes** (siehe Kapitel 5.2) gespeichert. Für jedes Zeichen vom Typ `char` werden also **zwei Bytes** benötigt. Hierdurch können in Java auch Schriftzeichen von Sprachen wie Chinesisch, Arabisch oder Hebräisch repräsentiert werden.

Java unterstützt seit der Version 1.1 mit den Characterstream-Klassen die Verwendung von Unicode-Zeichen bei der Ein- und Ausgabe. Die bereits von Anfang an vorhandenen **Bytestream-Klassen** werden **durch** die **Characterstream-Klassen erweitert**, jedoch auf keinen Fall ersetzt.

Bytestream-Klassen werden benötigt, um auf Dateien auf Speichermedien zuzugreifen, da das Lesen und Schreiben auf Speichermedien byteweise erfolgt.

Objekte der Bytestream-Klassen können selbst auch Daten vom Typ `char` und `String` – also Zeichen – verarbeiten, jedoch geht dabei das jeweils höherwertige Byte eines jeden Zeichens verloren.

Es muss darauf geachtet werden, dass für die korrekte Ein- und Ausgabe von Zeichen, die für ihre Darstellung mehr als 1 Byte benötigen, Characterstream-Klassen eingesetzt werden.

19.1.3 Sink-, Spring- und Processingstreams

Durch die Klassifizierung nach funktionalen Kriterien kann die Aufteilung der Bytestream- und Characterstream-Klassen noch verfeinert werden.

Bytestream-Klassen und **Characterstream-Klassen besitzen** jeweils spezielle **Klassen**, die sich in die **Kategorien Sink-**, **Spring- und Processingstream** einteilen lassen.

Klassifizierung von Stream-Klassen:

- Ein Objekt einer **Sinkstream-Klasse** kann Daten in eine **Datensenke** schreiben.
- Ein Objekt einer **Springstream-Klasse** kann Daten direkt aus einer **Datenquelle** lesen.
- Ein Objekt einer **Processingstream-Klasse** benutzt intern ein Objekt einer Sinkstream- oder Springstream-Klasse und **erweitert** deren **Funktionalität**, zum Beispiel um `int`-Werte lesen oder gepuffert schreiben zu können.

In den folgenden Kapiteln werden nun alle Stream-Klassen des Paketes `java.io` vorgestellt und den inzwischen bekannten **Kategorien Sinkstream**, **Springstream** und **Processingstream** zugeordnet. In Kapitel 19.3 werden die Bytestream-Klassen behandelt und danach die Characterstream-Klassen in Kapitel 19.4. Bevor jedoch im Detail auf die einzelnen Klassen eingegangen wird, erfolgt eine allgemeine Einführung in das Stream-Konzept.

19.2 Das Stream-Konzept

Objekte von Sink- und Springstream-Klassen werden dazu verwendet, um Bytes oder Zeichen sequentiell in Datensenken zu schreiben oder sequentiell aus Datenquellen zu lesen. Ist eine **erweiterte Funktionalität** erwünscht, kommen **Processingstreams** zum Einsatz. Es sind prinzipiell zwei Möglichkeiten denkbar, wie die Processingstream-Klassen die Funktionalität von Sink- und Springstream-Klassen erweitern und den Quellcode dieser Klassen wiederverwenden können:

- durch Vererbung
- oder durch **Aggregation**.

In Java hat man sich für die Aggregation entschieden und zwar enthält jede Processingstream-Klasse für eine

- Byte-**Inputstream**-Klasse ein privates Datenfeld vom Typ `InputStream`,
- Byte-**Outputstream**-Klasse ein privates Datenfeld vom Typ `OutputStream`,
- Character-**Inputstream**-Klasse ein privates Datenfeld vom Typ `Reader`,
- Character-**Outputstream**-Klasse ein privates Datenfeld vom Typ `Writer`.

Jede Processingstream-Klasse ist von einer der Klassen `InputStream`, `OutputStream`, `Reader` oder `Writer` abgeleitet und aggregiert gleichzeitig ein Objekt der entsprechenden Klasse.

Bild 19-3 zeigt dies stellvertretend an einer Processingstream-Klasse für Byte-Outputstreams:

Bild 19-3 Beziehungen einer Processingstream-Klasse

Ein Objekt einer Processingstream-Klasse, die von `OutputStream` abgeleitet ist, aggregiert ein Objekt vom Typ `OutputStream`. Dies bedeutet, dass eine Processingstream-Klasse eine Referenzvariable vom Typ `OutputStream` als Instanzvariable hat. Eine Referenz vom Typ `OutputStream` zeigt auf ein Objekt, dessen Klasse von der **abstrakten** Klasse `OutputStream` abgeleitet ist.

Die weitere Betrachtung soll der Einfachheit halber nur am Beispiel der **Byte-Outputstream-Klassen** erfolgen. Bild 19-4 zeigt einen Ausschnitt aus der Vererbungshierar-chie für die Byte-Outputstream-Klassen. Die grau hinterlegten Klassen sind Sinkstream-Klassen und die anderen von der abstrakten Klasse `OutputStream` abgeleiteten Klassen sind Processingstream-Klassen.

Bild 19-4 Ausschnitt aus der Klassenhierarchie der Byte-Outputstream-Klassen

Alle **Processingstream-Klassen**, die von der Klasse `OutputStream` abgeleitet sind, besitzen eine Instanzvariable vom Typ `OutputStream`. Diese Referenz kann auf jedes Objekt zeigen, dessen Klasse von der abstrakten Klasse `OutputStream` abgeleitet ist **und damit vom Typ `OutPutStream` ist**.

Die **Klasse** `OutputStream` definiert nur Methoden, mit denen man **Bytes** in eine **Datensenke schreiben** kann. Die Sinkstream-Klassen implementieren diese Methoden für die verschiedenen Datensenken. Möchte man eine erweiterte Funktionalität haben und möchte man beispielsweise Werte elementarer Datentypen wegschreiben, so benötigt man eine der Processingstream-Klassen. Die Processingstream-Klasse `DataOutputStream` ermöglicht zum Beispiel das Schreiben von Werten elementarer Datentypen. Die drei Sinkstream-Klassen `ByteArrayOutputStream`, `FileOutputStream` und `PipedOutputStream` können alle von einem Objekt der Processingstream-Klasse `DataOutputStream` aggregiert werden, da die genannten Klassen von der abstrakten Klasse `OutputStream` abgeleitet sind. Dies ist deshalb möglich, weil ein Objekt einer Sohnklasse sich immer so verhalten kann wie ein Objekt seiner Vaterklasse[222].

Jede Processingstream-Klasse stellt einen Konstruktor mit einem Übergabeparameter vom Typ `OutputStream` bzw. `InputStream` zur Verfügung. Dieser Parameter wird dann der privaten Instanzvariablen vom Typ `OutputStream` bzw. `InputStream` zugewiesen.

Es ist also möglich, jede beliebige Referenz, die auf ein Objekt zeigt, dessen Klasse von der abstrakten Klasse `OutputStream` abgeleitet ist, als Konstruktorparameter einer Processingstream-Klasse zu übergeben. Damit ist die Vorgehensweise für eine Ausgabe mit erweiterter Funktionalität klar:

Zuerst wird ein Objekt einer Sinkstream-Klasse erzeugt. Dann wird ein Objekt einer Processingstream-Klasse erzeugt und im Konstruktoraufruf wird die Referenz auf das erzeugte Sinkstream-Objekt übergeben.

Dabei steht Processingstream stellvertretend für eine konkrete Processingstream-Klasse, z. B. `DataOutputStream`, und Sinkstream steht stellvertretend für eine konkrete Sinkstream-Klasse wie z. B. `FileOutputStream`.

Hier der Pseudocode für die Erzeugung eines Processingstream-Objektes, das ein Sinkstream-Objekt aggregiert:

```
OutputStream sink = new Sinkstream-Klasse();
Processingstream-Klasse pro;
pro = new Processingstream-Klasse (sink);
```

[222] Vorausgesetzt, die Verträge der Basisklasse werden beim Ableiten eingehalten.

Grafisch lässt sich dies folgendermaßen darstellen:

Bild 19-5 Objekt einer Processingstream-Klasse benutzt ein Objekt einer Sinkstream-Klasse

Im Programm selbst werden nun **die komfortablen Methoden der Processingstream-Klasse aufgerufen**. Diese **leiten bei Bedarf die Aufrufe an das aggregierte Sinkstream-Objekt weiter**.

Ein Vorteil dieses Konzepts ist, dass nur eine einzige Processingstream-Klasse pro zusätzlicher Funktionalität geschrieben werden muss und diese Funktionalität zu allen Sinkstream-Klassen individuell hinzugefügt werden kann.

Eine Konstruktion, die es erlaubt, dass mehrere Klassen, die eine gemeinsame Basisklasse besitzen, um eine zusätzliche Funktionalität – hier die Funktionalität einer Processingstream-Klasse – individuell erweitert werden können, ist unter dem Namen **Dekorierer** als **Entwurfsmuster** bekannt.

Hätte man, anstatt das Dekorierer-Entwurfsmuster einzusetzen, das mit einer Aggregationsbeziehung zur Basisklasse arbeitet, die zusätzliche Funktionalität über eine Vererbungsbeziehung realisiert, so müsste für jede unterschiedliche Sinkstream-Klasse eine extra Processingstream-Klasse geschrieben werden.

Die Processingstream-Klasse `DataOutputStream` müsste dann zum Beispiel in drei Varianten vorliegen – für jede Sinkstream-Klasse genau eine spezifische Unterklasse. Durch die Anwendung des Dekorierer-Entwurfsmusters bei den Processingstream-Klassen bleibt die Vererbungshierarchie des Pakets `java.io` also überschaubar.

19.3 Bytestream-Klassen

Bytestream-Klassen arbeiten, wie schon erwähnt, nur mit **einzelnen Bytes**.

Da externe Datensenken und Datenquellen byteweise arbeiten, sind in der Praxis Bytestreams nötig, um in eine externe Datensenke zu schreiben oder um aus einer externen Datenquelle zu lesen. Deshalb gibt es bei **den Characterstream-Klassen**, wie Sie später noch sehen werden, auch **keine Sink- und Springstream-Klassen**, die **auf einer Datei** als externe Datensenke bzw. Datenquelle **arbeiten**.

Geht es um Daten, die mit einem einzigen Byte repräsentiert werden können, so ist eine Stream-Klasse, die Bytes verarbeiten kann, ausreichend.

Geht es um Daten, die durch **mehrere Bytes** dargestellt werden müssen (`short`, `int`, `long`, `float`, `double` und `char`), so benötigt man **spezielle Processingstream-Klassen**, um diese Daten korrekt in eine Folge von Bytes zu wandeln, die dann von einem Bytestream-Objekt verarbeitet werden können.

Umgekehrt müssen solche Daten, wenn sie aus einem Bytestream gelesen werden, von einem Processingstream-Objekt verarbeitet werden, um die einzelnen Bytes wieder korrekt zusammenzufügen.

19.3.1 Outputstream-Klassen

In Bild 19-4 wurde ein Ausschnitt der Byte-Outputstream-Klassenhierarchie gezeigt. Da es sich hier um Outputstream-Klassen handelt – also um Klassen, die Daten in eine Datensenke schreiben – sind in Bild 19-4 keine Springstream-Klassen zu finden. Die grau hinterlegten Klassen sind die Sinkstream-Klassen und die restlichen von der abstrakten Klasse `OutputStream` abgeleiteten Klassen sind Processingstream-Klassen.

19.3.1.1 Die Basisklasse `OutputStream`

Die Methoden der Klasse `OutputStream` ermöglichen das Schreiben von Bytes in eine Datensenke. Die abstrakte Klasse `OutputStream` deklariert die Methoden:

- `abstract void write (int b)`
- `void write (byte[] b)`
- `void write (byte[] b, int off, int len)`
- `void flush()`
- `void close()`

Nicht verwirren lassen darf man sich durch die Tatsache, dass die `write()`-Methode zum Schreiben eines Bytes jedoch einen Übergabeparameter vom Typ `int` verlangt. Dieser `write()`-Methode wird ein `int`-Wert im Wertebereich `0` bis `255` übergeben. Die anderen beiden `write()`-Methoden ermöglichen das Schreiben von `byte`-Arrays bzw. von deren Teilen in eine Datensenke. Es fällt auf, dass eine der drei `write()`-Methoden abstrakt ist. Alle Subklassen der Klasse `OutputStream` überschreiben diese abstrakte `write()`-Methode. Die Klasse `FileOutputStream` überschreibt diese Methode beispielsweise mit einer nativen[223] Methode, die ein Byte in eine Datei schreiben kann. Die anderen beiden `write()`-Methoden sind schon in der abstrakten Klasse `OutputStream` implementiert und rufen die abstrakte `write()`-Methode in einer Schleife auf, um mehrere Bytes hintereinander auszugeben. Diese `write()`-Methoden verlassen sich auf die Unterklassen, die garantieren, dass sie die abstrakte `write()`-Methode implementieren, wenn sie nicht auch abstrakt sein wollen.

[223] Eine native Methode von Java ist eine Methode, die beispielsweise in C oder C++ implementiert ist.

In der Klasse `OutputStream` selbst befindet sich also keine konkrete Ausgabefunktionalität. Diese muss von Subklassen wie z. B. `FileOutputStream` durch Implementieren der `write()`-Methode zur Ausgabe eines Bytes bereitgestellt werden.

Puffert eine Outputstream-Klasse – wie beispielsweise die Processingstream-Klasse `BufferedOutputStream` – die zu schreibenden Bytes, so kann die Methode `flush()` dafür verwendet werden, alle sich im Puffer befindenden Bytes wirklich in die Datensenke zu schreiben. Die Methode `close()`[224] schließt den Ausgabestrom. Nach dem Schließen sollte ein Ausgabestrom nicht mehr verwendet werden.

Wird das seit Java 7 eingeführte **try-with-resources**-Konzept genutzt, so wird die Methode `close()` beim Verlassen des `try`-Blocks implizit aufgerufen und muss nicht mehr wie bisher explizit aufgerufen werden. Mit den folgenden zwei Beispielen soll das neue Konzept verdeutlicht und der Unterschied zur seitherigen Verwendung dargestellt werden. Eine detaillierte Beschreibung des Konzepts erfolgte in Kapitel 13.2.4.

Im folgenden **Beispiel** wird die seitherige Verwendung, bei dem der **Stream explizit geschlossen** wird, dargestellt:

```
// Datei: ExplicitClose.java
import java.io.*;

public class ExplicitClose
{
   public static void main(String[] args)
   {
      FileOutputStream fos = null;
      try
      {
         fos = new FileOutputStream("daten.txt");
         // Daten schreiben
         fos.write(...);
      }
      catch (IOException ex)
      {
         // Es ist ein Fehler beim Öffnen oder Schreiben aufgetreten
      }
      finally
      {
         if (null != fos)
         {
            try
            {
               fos.close();
            }
```

[224] In Java gibt es keine Methode `open()`, um einen Strom zu öffnen. Ein Strom ist nach der Erzeugung mit dem `new`-Operator bereits automatisch geöffnet.

```
            catch (IOException ex)
            {
               // Es ist ein Fehler beim Schließen aufgetreten
            }
         }
      }
   }
}
```

Zuerst wird vor dem `try`-Block die Variable `fos` deklariert und mit `null` initialisiert. Dieses ist notwendig, damit später im `finally`-Block weiterhin auf die Variable `fos` zugegriffen werden kann. Innerhalb des `try`-Blocks wird nun ein Objekt der Klasse `FileOutputStream` angelegt und der Variablen `fos` zugewiesen. Danach werden die Daten mit der Methode `write()` in den Stream geschrieben. Im `finally`-Block muss zuerst geprüft werden, ob das Objekt der Klasse `Stream` angelegt werden konnte. Danach wird der Stream durch die Methode `close()` geschlossen. Hierfür muss allerdings die Methode `close()` von einem `try-catch` Block umschlossen werden.

Nun folgt das **Beispiel** mit dem **try-with-resources**-Konzept, beim dem der **Stream implizit geschlossen** wird.

```
// Datei: TryWithResource.java
import java.io.*;

public class TryWithResource {

   public void main(String[] args)
   {
      try (FileOutputStream fos = new FileOutputStream("daten.txt"))
      {
         // Daten schreiben
         fos.write(...);
      } catch (IOException ex)
      {
         // Es ist ein Fehler beim Öffnen oder Schreiben aufgetreten
      }
   }
}
```

Wie im ersten Beispiel wird ein Objekt der Klasse `FileOutputStream` angelegt und der Variablen `fos` zugewiesen. Diese Variablendefinition wird aber dem `try`-Statement als eine Art Parameter übergeben und steht später im `try`-Block für die weitere Nutzung zur Verfügung. Beim Verlassen des `try`-Blocks wird nun das Objekt der Klasse `FileOutputStream` automatisch geschlossen. Es spielt dabei keine Rolle, ob eine Exception auftritt oder nicht.

19.3.1.2 Sinkstream-Klassen

Es existieren die folgenden **Sinkstream-Klassen**[225] für Bytestreams:

[225] Sinkstream-Klassen gibt es logischerweise nur für Outputstreams und nicht für Inputstreams. Sinkstream bezieht sich immer auf eine Klasse, die etwas in eine Datensenke schreibt.

- `ByteArrayOutputStream`

  Mit einem Objekt der Klasse `ByteArrayOutputStream` kann man Bytes in ein `byte`-Array schreiben. Diese Klasse stellt zwei Konstruktoren zur Verfügung, einen ohne Parameter und einen, der einen `int`-Wert für die anzulegende Größe des `byte`-Arrays entgegennimmt.

- `FileOutputStream`

  Die Bytestream-Klasse `FileOutputStream` arbeitet direkt auf einer externen Datensenke. Sie bietet mehrere Konstruktoren an, unter anderem einen, dem als Parameter der Name der Datei übergeben wird, in die geschrieben werden soll.

- `PipedOutputStream`

  Ein Objekt der Klasse `PipedOutputStream` schreibt Bytes in eine Pipe. Eine **Pipe** wiederum wird in Java durch ein **`byte`-Array realisiert**.

19.3.1.3 Processingstream-Klassen für Byte-Outputstreams

Alle Processingstream-Klassen für Byte-Outputstreams aggregieren ein Objekt vom Typ `OutputStream`. Deshalb verlangen alle Konstruktoren dieser Processingstream-Klassen als ersten Parameter eine Referenz auf ein Objekt vom Typ `OutputStream`.

Die Processingstream-Klasse `FilterOutputStream` ist eine Klasse, die nur die Methoden der abstrakten Klasse `OutputStream` implementiert. Man kann zwar Exemplare der Klasse `FilterOutputStream` erzeugen, jedoch stellen diese noch keine erweiterte Funktionalität im Vergleich zu Objekten der Sinkstream-Klassen zur Verfügung. Alle Aufrufe, die an ein Objekt der Klasse `FilterOutputStream` gehen, werden direkt an das aggregierte Objekt vom Typ `OutputStream` weitergeleitet (delegiert). Die Processingstream-Klassen `BufferedOutputStream`, `DataOutputStream` und `PrintStream` leiten von der Klasse `FilterOutputStream` ab.

Im Folgenden werden einige **Processingstream-Klassen** vorgestellt und erläutert:

- `BufferedOutputStream`

  Die Klasse `BufferedOutputStream` hat als Instanzvariable eine Referenz auf ein `byte`-Array einer bestimmten Größe. Wird die Methode `write()` für ein Objekt der Klasse `BufferedOutputStream` aufgerufen, so wird dieser Aufruf nicht sofort an das aggregierte Objekt vom Typ `OutputStream` weitergeleitet, sondern die zu schreibenden Bytes werden zuerst in das `byte`-Array geschrieben. Erst wenn das `byte`-Array voll ist, wird der Schreibbefehl an das aggregierte Objekt vom Typ `OutputStream` weitergeleitet.

  Durch diese Pufferung ist ein wesentlich effizienteres Schreiben – z. B. in eine Datei – zu erreichen, da nicht jedes einzelne Byte getrennt geschrieben wird, sondern ein ganzes Array – d. h. ein Puffer – auf einmal.

- `DataOutputStream`

  Ein Objekt der Klasse `DataOutputStream` kann **alle primitiven Java-Datentypen** schreiben, auch diejenigen, die durch zwei oder mehr Bytes repräsentiert werden.

  Dabei müssen einige Dinge beachtet werden – z. B. die Reihenfolge, in der die Bytes geschrieben werden. Java schreibt Daten grundsätzlich im "big-endian" Format, d. h. das höherwertige Byte wird zuerst geschrieben.

  Die Klasse `DataOutputStream` kann auch `String`-Objekte schreiben.

  Allerdings sollten String-Objekte, welche in Java durch Unicode-Zeichen (siehe Kapitel 5.2) dargestellt werden, mit der Methode `writeChars (String data)` geschrieben werden. Benutzt man die Methode `writeBytes (String data)`, kann einiges schief gehen. Enthält ein zu schreibender String Unicode-Zeichen, die zu ihrer Darstellung das höherwertige Byte mitbenutzen, so geht die Information, die im höherwertigen Byte steht, einfach verloren. Es wird beim Aufruf der Methode `writeBytes (String data)` nur das niederwertige Byte eines jeden Zeichens geschrieben. Bei der Rekonstruktion erhält man den ursprünglichen String also nicht mehr zurück.

- `PrintStream`

  Die Klasse `PrintStream` bietet Methoden zum **Schreiben aller elementaren Datentypen**. Dabei wandeln die Schreibmethoden alle Datentypen in deren **String-Repräsentation** und leiten diese an das aggregierte Objekt vom Typ `OutputStream` weiter. Diese Klasse wird zur einfachen Textausgabe benutzt.

  Die Standardausgabe benutzt diese Klasse, um die Ausgaben auf dem Bildschirm zu tätigen. Mit `System.out.println()` wird die Methode `println()` eines Objektes vom Typ `PrintStream` aufgerufen, auf das die Referenz `out` zeigt.

- `ObjectOutputStream`

  Die Klasse `ObjectOutputStream` ist eine Processingstream-Klasse zum Schreiben von **Objekten in einen Ausgabestrom** (siehe Kapitel 19.7).

Das folgende Beispiel zeigt die Verwendung der Processingstream-Klassen `BufferedOutputStream` und `PrintStream`. Dabei ist die Klasse `BufferedOutputStream` ein Dekorator und dekoriert die Standardausgabe um eine gepufferte Ausgabe auf dem Bildschirm. Die Puffergröße des Objektes der Klasse `BufferedOutputStream` wird auf 100 Bytes gesetzt. Ebenfalls dient die Klasse `PrintStream` als Dekorator und dekoriert bzw. benutzt das Objekt der Klasse `BufferedOutputStream`. Der `try`-Block mit dem Aufruf der statischen Methode `sleep()` auf der Klasse `Thread` dient lediglich dazu, den Programmablauf zu verlangsamen, damit

am Bildschirm beobachtet werden kann, dass immer mehrere Datensätze aufgrund der Pufferung auf einmal ausgegeben werden.

Hier das **Beispielprogramm**:

```
// Datei: Messdaten.java

import java.io.*;

public class Messdaten
{
  public static void main (String[] args)
  {
    BufferedOutputStream out =
        new BufferedOutputStream (System.out, 100);
    PrintStream print = new PrintStream (out);
    for (int i = 0; i < 10; i++)
    {
      print.println ("Messwert Nr. " + i
          + " hat den Wert: " + getMesswert());
      try
      {
        Thread.sleep(1000);
      }
      catch (InterruptedException ex)
      {
          // die Methode sleep() wurde frühzeitig unterbrochen
      }
    }
    // Dafür sorgen, dass der Puffer geleert wird und damit alles
    // ausgegeben wird.
    print.flush(); // den Rest noch ausgeben
    print.close(); // den Stream schließen
  }

  public static double getMesswert()
  {
    return Math.random() * 1000;
  }
}
```

Die Ausgabe des Programms ist:

```
Messwert Nr. 0 hat den Wert: 998.8001676001394
Messwert Nr. 1 hat den Wert: 620.4308663504445
Messwert Nr. 2 hat den Wert: 254.09528643582325
Messwert Nr. 3 hat den Wert: 565.4049915738923
Messwert Nr. 4 hat den Wert: 599.1219468716811
Messwert Nr. 5 hat den Wert: 223.6336443179907
Messwert Nr. 6 hat den Wert: 50.13028255727614
Messwert Nr. 7 hat den Wert: 700.8958107530149
Messwert Nr. 8 hat den Wert: 259.4003857856323
Messwert Nr. 9 hat den Wert: 387.79674635543824
```

19.3.2 Inputstream-Klassen

Die Byte-Inputstream-Klassen bilden das Gegenstück zu den Byte-Outputstream-Klassen. Einen Ausschnitt aus der Klassenhierarchie der Byte-Inputstream-Klassen ist in Bild 19-6 zu sehen:

Bild 19-6 Ausschnitt aus der Klassenhierarchie der Byte-Inputstream-Klassen

Bei den Inputstream-Klassen sind keine Sinkstream-Klassen zu finden, da mit Inputstream-Klassen aus Datenquellen gelesen wird – es sind also nur Springstream-Klassen vorhanden.

19.3.2.1 Die Basisklasse `InputStream`

Die abstrakte Klasse `InputStream` gibt die Aufrufschnittstelle für alle Byte-Inputstream-Klassen vor. Es werden die folgenden Methoden deklariert:

- `abstract int read()`
- `int read (byte[] b)`
- `int read (byte[] b, int off, int len)`
- `void close()`
- `long skip (long n)`
- `int available()`
- `void mark (int readlimit)`
- `void reset()`
- `boolean markSupported()`

Die drei `read()`-Methoden werden zum Lesen von einzelnen Bytes, `byte`-Arrays und Teilen von `byte`-Arrays aus einer Datenquelle verwendet.

Die `read()`-Methode, die ein einzelnes Byte einliest, ist abstrakt und muss von Subklassen implementiert werden. Die anderen beiden `read()`-Methoden verwenden die abstrakte `read()`-Methode, um mehrere Bytes in ein `byte`-Array einzulesen.

Die Methode `close()` schließt den Eingabestrom und die Methode `skip()` überliest die angegebene Anzahl von Bytes. Mit der Methode `available()` kann geprüft werden, wie viele Bytes zum Einlesen im Eingabestrom bereitstehen. Die drei Methoden `mark()`, `markSupported()` und `reset()` stellen eine Art Lesezeichen-Funktionalität bereit. Unterstützt ein Eingabestrom die Lesezeichen-Funktionalität, so gibt die Methode `markSupported()` den Wert `true` zurück. In diesem Fall kann mit `mark()` eine beliebige Stelle im Eingabestrom markiert werden, um dann später mit `reset()` an diese Stelle zurückspringen zu können.

Das **try-with-resources**-Konzept, das in Kapitel 13.2.5 für OutputStreams beschrieben wurde, kann auch bei InputStreams angewandt werden.

19.3.2.2 Springstream-Klassen

Es existieren die folgenden **Springstream-Klassen** für Bytestreams:

- `ByteArrayInputStream`

  Mit einem Objekt der Klasse `ByteArrayInputStream` kann man Bytes aus einem `byte`-Array lesen. Im Konstruktor übergibt man die Datenquelle, von der gelesen werden soll, nämlich ein Objekt vom Typ `byte`-Array.

- `FileInputStream`

  Die einzige Bytestream-Klasse im Paket `java.io`, die direkt auf einer externen Datenquelle arbeitet, ist die Klasse `FileInputStream`. Sie bietet mehrere Konstruktoren an, unter anderem einen, dem als Parameter direkt der Name der Datei übergeben wird, aus der gelesen werden soll.

- `PipedInputStream`

  Ein Objekt einer `PipedInputStream`-Klasse kann nur im Zusammenhang mit einem Objekt der Klasse `PipedOutputStream` eingesetzt werden. Ein Objekt der Klasse `PipedInputStream` liest aus der Datenquelle, die ein Objekt vom Typ `PipedOutputStream` als Datensenke verwendet. Dabei dient als Datenquelle bzw. als Datensenke ein `byte`-Array.

19.3.2.3 Processingstream-Klassen für Byte-Inputstreams

Alle Processingstream-Klassen für Byte-Inputstreams aggregieren ein Objekt vom Typ `InputStream`. Der erste Parameter eines Konstruktors dieser Klassen ist deshalb vom Typ `InputStream`.

Die Processingstream-Klasse `FilterInputStream` ist eine Klasse, die nur die Methoden der abstrakten Klasse `InputStream` implementiert. Es gilt für diese Klasse das Gleiche wie für die entsprechende Klasse `FilterOutputStream` bei den Outputstreams. Bild 19-7 zeigt ein Objekt einer Processingstream-Klasse, das ein Objekt einer Springstream-Klasse benutzt:

Bild 19-7 Objekt einer Processingstream-Klasse benutzt ein Objekt einer Springstream-Klasse

Das Objekt der **Springstream-Klasse** ist damit das **aggregierte Objekt**. An dieses Objekt leitet das Objekt der Processingstream-Klasse die Aufrufe weiter. Die Processingstream-Klassen für Inputstreams sind zum großen Teil äquivalent zu den Processingstream-Klassen für Outputstreams. Diese wurden bereits detailliert vorgestellt. Daher wird hier nur das Wichtigste zusammengefasst:

- Die Klasse `BufferedInputStream` ermöglicht – mit einer äquivalenten Technik wie die Klasse `BufferedOutputStream` – das gepufferte und damit effizientere Lesen aus einem Eingabestrom.
- Ein Objekt der Klasse `DataInputStream` kann alle primitiven Java-Datentypen aus einem Eingabestrom lesen.
- Die Klasse `PushbackInputStream` stellt Methoden zur Verfügung, mit denen es möglich ist, Bytes in den Eingabestrom zurückzustellen.
- Die Klasse `SequenceInputStream` dient zum Verketten von mehreren Eingabeströmen. Damit können mehrere Eingabeströme hintereinandergehängt werden und es kann gelesen werden, als wenn nur ein einziger Eingabestrom vorhanden wäre.
- Die Klasse `ObjectInputStream` ist eine Processingstream-Klasse, die es ermöglicht, Objekte aus einem Eingabestrom zu lesen. Darauf wird in Kapitel 19.7 genauer eingegangen.

19.3.3 Bytes in eine Datei schreiben und aus einer Datei lesen

Im Folgenden soll ein einfaches **Beispiel** zur Ein- und Ausgabe von einzelnen Bytes betrachtet werden. Zuerst werden mit Hilfe eines Objektes der Klasse `FileOutputStream` 10 Bytes in die Datei `Bytes.txt` geschrieben. Danach werden diese 10 Bytes mit Hilfe eines Objektes der Klasse `FileInputStream` wieder eingelesen und zur Kontrolle ausgegeben. Hier das Programm:

```
// Datei: EinUndAusgabeVonEinzelnenBytes.java

import java.io.*;

public class EinUndAusgabeVonEinzelnenBytes
{
   public static void main (String[] args)
   {
      try (FileOutputStream fos = new FileOutputStream ("Bytes.txt"))
      {
         for (int i = 0; i < 10; i++)
         {
            fos.write (i);
         }
      }
```

```
      catch (IOException ex)
      {
         System.err.println ("IO-Fehler beim Schreiben");
         ex.printStackTrace();
      }

      try (FileInputStream fis = new FileInputStream ("Bytes.txt"))
      {
         for (int i = 0; i < 10; i++)
         {
            System.out.print (fis.read());
            System.out.print (" ");
         }
      }
      catch (IOException ex)
      {
         System.err.println ("IO-Fehler beim Lesen");
         ex.printStackTrace();
      }
   }
}
```

Die Ausgabe des Programms ist:

```
0 1 2 3 4 5 6 7 8 9
```

Um die Verwendung der Processingstream-Klassen zu verdeutlichen, soll im folgenden **Beispiel** die Klasse `BufferedOutputStream` verwendet werden. Diese Klasse erweitert ein Objekt der Klasse `FileOutputStream` um die Fähigkeit, Daten zu puffern, bevor diese in die Datei geschrieben werden.

Hier das Beispiel:

```
// Datei: GepufferteEinUndAusgabeVonEinzelnenBytes.java

import java.io.*;

public class GepufferteEinUndAusgabeVonEinzelnenBytes
{
   public static void main (String[] args)
   {
      try (FileOutputStream fos = new FileOutputStream ("Bytes.txt");
           BufferedOutputStream bos =
                                 new BufferedOutputStream (fos, 5))
      {
         for (int i = 0; i < 10; i++)
         {
            bos.write (i);
         }

         dateiInhaltLesen();
         bos.flush();
      }
```

```
      catch (IOException ex)
      {
         System.err.println ("IO-Fehler beim Schreiben");
         ex.printStackTrace();
      }
      dateiInhaltLesen();
   }

   private static void dateiInhaltLesen()
   {
      try (FileInputStream fis = new FileInputStream ("Bytes.txt"))
      {
         System.out.println();
         int b;

         for (int i = 0; i < 10; i++)
         {
            b = fis.read();

            if (b == -1)
            {
               break;
            }
            System.out.print (b);
            System.out.print (" ");
         }
      }
      catch (IOException ex)
      {
         System.err.println ("IO-Fehler beim Lesen");
         ex.printStackTrace();
      }
   }
}
```

Die Ausgabe des Programms ist:

```
0 1 2 3 4
0 1 2 3 4 5 6 7 8 9
```

In diesem Beispiel werden zuerst 10 Bytes in einen gepufferten Stream geschrieben. Die Puffergröße wurde beim Aufruf des Konstruktors auf **5 Bytes** festgelegt. Dies bedeutet, dass der Inhalt des Puffers in die Datei geschrieben wird, sobald dieser voll ist und ein weiteres Byte in den Puffer geschrieben werden soll. Nach dem Schreiben der 10 Bytes – also beim ersten Aufruf von `dateiInhaltLesen()` – wird der Inhalt der Datei ausgegeben, ohne dass der Puffer des Objektes der Klasse `BufferedOutputStream` explizit mit `flush()` geleert wurde. Es ist zu erwarten, dass zu diesem Zeitpunkt nur die Bytes 0 bis 4 aus der Datei gelesen werden können, da sich die restlichen 5 Bytes noch im Ausgabepuffer befinden und noch gar nicht in der Datei angekommen sind. Die Ausgabe des Programms bestätigt dies. Nach dem Leeren des Puffers und dem Schließen des Streams können alle 10 Bytes aus der Datei gelesen werden.

19.3.4 Byte-Arrays in eine Datei schreiben und aus einer Datei lesen

Das Schreiben in Dateien und Lesen von `byte`-Arrays aus Dateien funktioniert im Prinzip genau gleich wie das Schreiben und Lesen einzelner Bytes. Das folgende **Beispiel** veranschaulicht dies:

```
// Datei: EinUndAusgabeVonByteArrays.java

import java.io.*;

public class EinUndAusgabeVonByteArrays
{
   public static void main (String[] args)
   {
      byte[] b = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
      byte[] c = new byte [10];

      try (FileOutputStream fos = new FileOutputStream ("Bytes.txt"))
      {
         fos.write (b);
      }
      catch (IOException ex)
      {
         System.err.println ("IO-Fehler beim Schreiben");
         ex.printStackTrace();
      }

      try (FileInputStream fis = new FileInputStream ("Bytes.txt"))
      {
         fis.read (c);

         for (int i = 0; i < 10; i++)
         {
            System.out.print (c [i]);
            System.out.print (" ");
         }
      }
      catch (IOException ex)
      {
         System.err.println ("IO-Fehler beim Lesen");
         ex.printStackTrace();
      }
   }
}
```

Die Ausgabe des Programms ist:

```
0 1 2 3 4 5 6 7 8 9
```

Da die Ein- und Ausgabe von `byte`-Arrays grundsätzlich schneller ist als die Ausgabe von einzelnen Bytes, ist es zu empfehlen – wann immer möglich – `byte`-Arrays anstatt einzelner Bytes zu lesen oder zu schreiben.

19.3.5 Streaming von elementaren Datentypen

Bei der Ein- und Ausgabe elementarer Datentypen wie z. B. `short`, `int`, `float` und `double` gibt es einiges zu beachten. Da die Ein- und Ausgabe bei Dateien auf der untersten Ebene nur mit einzelnen Bytes arbeitet, muss bei der Ein- und Ausgabe von Datentypen, die aus mehreren Bytes bestehen, die Reihenfolge der Bytes beachtet werden.

Dieser Sachverhalt soll an folgendem Beispiel veranschaulicht werden: Es soll eine einfache Folge von Bytes wie z. B. ein `byte`-Array mit den vier Bytes `5A`, `42`, `F7` und `1C` in eine Datei geschrieben werden. Dabei gibt es in Bezug auf die Byte-Reihenfolge nichts weiter zu beachten. Da diese Bytes in diesem Fall nicht Teile der Werte von Datentypen sind, die aus mehreren Bytes bestehen, sind sie voneinander unabhängig. Das heißt, es kann einfach Byte für Byte in die Datei geschrieben werden. Also erst `5A`, dann `42`, dann `F7` und zuletzt `1C`.

Wird nun ein `int`-Wert in die Datei geschrieben, muss man sich auf eine bestimmte Byte-Reihenfolge einigen. Der `int`-Wert 447 362 175 besteht beispielsweise aus den vier Bytes `1A`, `AA`, `34` und `7F`. Wird dieser `int`-Wert nun in eine Datei geschrieben, so kann dies auf zwei grundsätzlich unterschiedliche Arten geschehen:

- Eine Möglichkeit ist, das höherwertige Byte – also `1A` – zuerst zu schreiben, dann die Bytes `AA`, `34` und `7F`. Diese Reihenfolge wird "**Big-Endian**" genannt. Java verwendet ausschließlich diese Byte-Reihenfolge zur Speicherung von Daten im Hauptspeicher sowie beim Schreiben von Daten in einen Stream.
- Die zweite Möglichkeit ist, das niederwertige Byte – hier `7F` – zuerst zu schreiben und dann die Bytes `34`, `AA` und `1A`. Diese Reihenfolge wird "**Little-Endian**"" genannt. Werden Daten z. B. von einem C-Programm auf einem Rechner mit Intel-Prozessor in eine Datei geschrieben, so geschieht dies in der "Little-Endian"-Byte-Reihenfolge.

Das folgende Bild visualisiert die Byte-Reihenfolge für Little-Endian und Big-Endian:

Bild 19-8 Byte-Reihenfolge für Little-Endian und Big-Endian

In Java wird immer die "Big-Endian"-Byte-Reihenfolge verwendet.

Die beiden Klassen **`DataOutputStream`** und **`DataInputStream`** werden zur Ein- und Ausgabe elementarer Datentypen benutzt. Sie können auch zur Ein- und Ausgabe von Zeichen verwendet werden. Dies ist jedoch nicht zu empfehlen, da diese

Klassen Unicode-Zeichen nicht korrekt verarbeiten. Das folgende **Beispiel** zeigt, wie elementare Datentypen in eine Datei geschrieben werden können und aus dieser auch wieder ausgelesen werden können:

```
// Datei: EinUndAusgabeVonDatenPrimitiverTypen.java

import java.io.*;

public class EinUndAusgabeVonDatenPrimitiverTypen
{
   public static void main (String[] args)
   {
      try (FileOutputStream fos = new FileOutputStream ("Daten.dat");
           DataOutputStream dos = new DataOutputStream (fos))
      {
         dos.writeInt (1);
         dos.writeDouble (1.1);
         dos.writeInt (2);
      }
      catch (IOException ex)
      {
         System.err.println ("IO-Fehler beim Schreiben");
         ex.printStackTrace();
      }

      try (FileInputStream fis = new FileInputStream ("Daten.dat");
           DataInputStream dis = new DataInputStream (fis))
      {

         System.out.println (dis.readInt());
         System.out.println (dis.readDouble());
         System.out.println (dis.readInt());
      }
      catch (IOException ex)
      {
         System.err.println ("IO-Fehler beim Lesen");
         ex.printStackTrace();
      }
   }
}
```

Die Ausgabe des Programms ist:

```
1
1.1
2
```

Die im Beispiel generierte Textdatei `Daten.dat` enthält die geschriebene Information als Bytes. Der `int`-Wert wird zum Beispiel mit 4 Bytes codiert in die Datei geschrieben. Hierbei spielt es keine Rolle, dass der Wert 1 auch in einem einzigen Byte codiert werden könnte.

Das folgende Bild zeigt die beteiligten Stream-Klassen aus dem Beispielprogramm:

Bild 19-9 Schreiben und Lesen elementarer Datentypen mit Hilfe der Processing-stream-Klassen `DataInputStream` *und* `DataOutputStream`

19.4 Characterstream-Klassen

Characterstream-Klassen arbeiten mit **Zeichen** statt nur mit einzelnen Bytes. Da auf **externe** Datensenken und Datenquellen – wie oben erwähnt – nur **byteweise** gearbeitet werden kann, sind **grundsätzlich Bytestreams** nötig, **um mit externen Datensenken und Datenquellen zu kommunizieren**. Aus diesem Grund gibt es auch keine Characterstream-Klassen, die direkt mit externen Datensenken/-quellen arbeiten. Sollen Zeichen in eine **externe** Datensenke – z. B. eine Datei – geschrieben oder umgekehrt aus dieser gelesen werden, so müssen sogenannte **Bridge-Klassen** verwendet werden.

Bridge-Klassen können Bytes in Zeichen wandeln und umgekehrt.

Auf die genaue Funktionsweise und die damit zusammenhängende Problematik wird in Kapitel 19.4.3 genauer eingegangen.

Interne Datensenken und **-quellen** befinden sich komplett innerhalb eines Java-Programms und stellen letztendlich **`char`-Arrays** dar. Da innerhalb eines Java-Programms alles im **Unicode-Format** verarbeitet wird, muss keine Umsetzung von Bytes in Zeichen und umgekehrt stattfinden. Die Characterstream-Klassen bieten im Wesentlichen die gleiche Funktionalität wie die Bytestream-Klassen. Der Unterschied ist, dass Characterstream-Klassen Zeichen, die aus zwei Bytes bestehen, korrekt verarbeiten können.

19.4.1 Writer-Klassen

Bild 19-10 zeigt einen Ausschnitt der Klassenhierarchie der Character-Outputstream-Klassen. Die hier dargestellten Processingstream-Klassen und Sinkstream-Klassen sind von der abstrakten Klasse `Writer` abgeleitet. Die grau hinterlegten Klassen sind die Sinkstream-Klassen und die Klassen `BufferedWriter`, `FilterWriter` und `PrintWriter` sind die Processingstream-Klassen. Die Klasse `Output-`

`StreamWriter` ist eine sogenannte Bridge-Klasse (siehe Kapitel 19.4.3) und die Klasse `FileWriter` eine sogenannte Bequemlichkeitsklasse (siehe Kapitel 19.4.4). Das folgende Bild zeigt einen Ausschnitt aus der Klassenhierarchie der Character-Outputstream-Klassen:

Bild 19-10 Ausschnitt aus der Klassenhierarchie der Character-Outputstream-Klassen

19.4.1.1 Die Basisklasse `Writer`

Die abstrakte Klasse `Writer` deklariert folgende Methoden:

- `void write (int c)`
- `void write (char[] c)`
- `abstract void write (char[] c, int off, int len)`
- `void write (String s)`
- `void write (String s, int off, int len)`
- `abstract void flush()`
- `abstract void close()`

Wie bei den Bytestreams gibt es auch hier drei grundlegende Ausgabemethoden. Eine Methode für die Ausgabe eines einzelnen Zeichens und zwei weitere für die Ausgabe eines Zeichen-Arrays. Ein Unterschied ist aber, dass die Methode zur Ausgabe von Zeichen-Arrays abstrakt ist und nicht diejenige für die Ausgabe einzelner Zeichen, wie es bei den Bytestreams der Fall war. Das Schreiben eines Zeichen-Arrays muss folglich von den Subklassen von `Writer` implementiert werden. Die Ausgabe eines einzelnen Zeichens ist dagegen in der Klasse `Writer` implementiert. Alle nicht abstrakten `write()`-Methoden der Klasse `Writer` benutzen wiederum gerade die von den Subklassen zu implementierende abstrakte `write()`-Methode für `char`-Arrays, um Zeichen zu schreiben. Die Methode `write (int c)` arbeitet intern beispielsweise folgendermaßen:

```
public void write (int c) throws IOException
{
   // . . . . .
   // writeBuffer ist ein char[]-Array
   writeBuffer [0] = (char) c;
   // Ab der Position 0 im char[]-Array 1 Element schreiben
   write (writeBuffer, 0, 1);
}
```

Weiterhin werden noch zwei Schreibmethoden zum Schreiben von String-Objekten zur Verfügung gestellt. Die abstrakte Methode `flush()` wird bei puffernden Streams zum Schreiben der im Puffer befindlichen Daten in die Datensenke verwendet und die abstrakte Methode `close()` schließt einen Ausgabestrom. Die Methoden `flush()` und `close()` sind abstrakt und müssen von den Sinkstream-Klassen implementiert werden.

19.4.1.2 Sinkstream-Klassen

Als Sinkstream-Klassen für Characterstreams existieren die Klassen **`CharArrayWriter`**, **`PipedWriter`** und **`StringWriter`**. Dabei benutzt ein Objekt der Klasse `CharArrayWriter` ein `char`-Array als Datensenke und ein Objekt der Klasse `StringWriter` ein String-Objekt als Datensenke. Ein Objekt der Klasse `PipedWriter` funktioniert gleich wie ein Objekt der äquivalenten Klasse `PipedOutputStream` bei den Bytestreams mit dem Unterschied, dass als Datensenke kein `byte`-Array, sondern ein `char`-Array verwendet wird.

19.4.1.3 Processingstream-Klassen für Character-Outputstreams

Alle Processingstream-Klassen für Character-Outputstreams aggregieren ein Objekt vom Typ `Writer`. Deshalb verlangen alle Konstruktoren dieser Processingstream-Klassen als ersten Parameter eine Referenz auf ein Objekt vom Typ `Writer`.

Im Bild 19-10 fällt auf, dass die Klasse **`FilterWriter`** abstrakt ist und im Gegensatz zu ihrem Äquivalent `FilterOutputStream` keine Subklassen hat. Eine solche "Umorganisation" der Stream-Klassen ist bei den Characterstreams im Vergleich zu den Bytestreams leider vorzufinden und trägt nicht gerade zur Übersichtlichkeit des Pakets `java.io` bei.

Die Klasse `FilterWriter` kann dazu benutzt werden, Filter-Klassen durch Ableitung selbst zu schreiben. Die Processingstream-Klasse **`BufferedWriter`** ist eine Subklasse der Klasse `Writer`. `BufferedWriter` und **`PrintWriter`** funktionieren analog zu den Bytestream-Klassen `BufferedOutputStream` und `PrintStream`.

19.4.2 Reader-Klassen

Bild 19-11 zeigt einen Ausschnitt der Klassenhierarchie der Character-Inputstream-Klassen. Die gemeinsame Basisklasse ist die abstrakte Klasse `Reader`. Die grau hinterlegten Klassen sind die Springstream-Klassen und die Klassen **`BufferedReader`**, **`LineNumberReader`**, **`FilterReader`** und **`PushbackReader`** sind Processingstream-Klassen. Die Klasse **`InputStreamReader`** ist eine **Bridge-Klasse** (siehe Kapitel 19.4.3) und die Klasse **`FileReader`** ist eine Bequemlichkeits-Klasse (siehe Kapitel 19.4.4).

Das folgende Bild zeigt einen Ausschnitt aus der Klassenhierarchie der Character-Inputstream-Klassen:

Bild 19-11 Ausschnitt aus der Klassenhierarchie der Character-Inputstream-Klassen

19.4.2.1 Die Basisklasse Reader

Die abstrakte Klasse `Reader` deklariert folgende Methoden:

- `int read()`
- `int read (char[] c)`
- `abstract int read (char[] c, int off, int len)`
- `boolean ready()`
- `long skip (long n)`
- `boolean markSupported()`
- `void mark (int readAheadLimit)`
- `void reset()`
- `abstract void close()`

Es existieren drei `read()`-Methoden zum Lesen eines Zeichens und zum Lesen von `char`-Arrays. Dabei ist auch wieder die Methode zum Lesen eines Zeichenarrays abstrakt und muss von den Subklassen der Klasse `Reader` (siehe Bild 19-11) implementiert werden.

Die Methode `ready()` kann mit der Methode `available()` der Klasse `InputStream` verglichen werden. Allerdings lässt sich mit der Methode `ready()` nur ermitteln, ob Zeichen zum Lesen bereitstehen, aber nicht wie viele. Die Methoden `skip()`, `markSupported()`, `mark()`, `reset()` und `close()` haben dieselbe Bedeutung wie bei der Klasse `InputStream`.

19.4.2.2 Springstream-Klassen

Als Springstream-Klasse bei den Character-Inputstreams ist die Klasse `CharArrayReader` zum Lesen von Zeichen aus einem `char`-Array, die Klasse `PipedReader` zum Lesen von Zeichen aus einer Pipe und die Klasse `StringReader` zum Lesen von Zeichen aus einem String-Objekt zu nennen.

19.4.2.3 Processingstream-Klassen für Character-Inputstreams

Als Processingstream-Klasse bei den Character-Inputstreams ist die Klasse `BufferedReader` zum gepufferten Lesen von Zeichen, die Klasse `FilterReader` (abstrakte Klasse) zur Spezialisierung von Filter-Klassen, die Klasse `LineNumberReader` zum zeilenweisen Lesen von Zeichen und die Klasse `PushbackReader` zum Zurückstellen von Zeichen in den Eingabestrom zu nennen. Weitere Processingstream-Klassen sind in der Java-API zu finden.

19.4.3 Bridge-Klassen

Eine Bridge-Klasse wandelt Zeichen in Bytes und umgekehrt. Bridge-Klassen verbinden damit die Characterstream-Klassen mit den Bytestream-Klassen.

Die Umwandlung erfolgt unter Berücksichtigung eines Character Encodings. Die Bridge-Klassen **`OutputStreamWriter`** und **`InputStreamReader`** sind direkte Subklassen von `Writer` bzw. `Reader`.

Diese Bridge-Klassen werden beim **Einlesen** von Zeichen **aus** einer **externen Datenquelle** bzw. bei der **Ausgabe** von Zeichen **in** eine **externe Datensenke** benötigt.

Sollen Zeichen in eine externe Datensenke, z. B. eine Datei, geschrieben werden, so müssen diese zuerst in Bytes umgewandelt werden, sodass ein Objekt einer Bytestream-Klasse diese in eine externe Datensenke schreiben kann. Ebenso können nur einzelne Bytes aus externen Datenquellen gelesen werden. Stellen diese einen Zeichenstrom dar, müssen die einzelnen Bytes wieder in Zeichen umgewandelt werden. Diese Umwandlung geschieht unter der Berücksichtigung eines Character Encodings. Beispiele für **Character-Encodings** sind: ASCII, Latin-1, Big-5, UTF-8 oder UTF-16 des Unicodes.

Wird ein Exemplar der Klasse `OutputStreamWriter` bzw. `InputStreamReader` erzeugt, so kann man im Konstruktor ein Character Encoding angeben. Wird der parameterlose Konstruktor verwendet, so wird das **Default Encoding**[226] der aktuellen Plattform verwendet.

19.4.4 Bequemlichkeits-Klassen

Die Klasse **`FileWriter`** ist keine wirkliche Sinkstream-Klasse, sondern eine Klasse, die nur so tut, als ob sie eine wäre.

Ein Objekt der Klasse `FileWriter` aggregiert einfach ein Objekt der Klasse `FileOutputStream`.

[226] Das Default-Encoding lässt sich mit dem Aufruf `System.getProperty ("file.encoding")` ermitteln.

Die Klasse `FileOutputStream` ist dabei die eigentliche Sinkstream-Klasse. Das folgende Bild zeigt, dass ein Objekt der Klasse `FileWriter` ein Objekt der Klasse `FileOutputStream` benutzt, um in eine Datei zu schreiben:

Bild 19-12 Verwendung eines Objekts der Klasse `FileWriter`

Die Klasse `FileWriter` wird deshalb Bequemlichkeitsklasse genannt, weil sie es dem Benutzer ermöglicht, Zeichen zu schreiben, die dann mit Hilfe der Klasse `FileOutputStream` in einen Bytestrom überführt werden.

Hierzu müsste man eigentlich eine Bridge-Klasse verwenden, die Zeichen in Bytes wandelt. Die Bridge-Klasse wiederum müsste den Bytestrom an ein Objekt der Klasse `FileOutputStream` weiterleiten, um die Bytes in eine Datei zu schreiben.

Der Programmierer spart sich jedoch durch die Verwendung der Klasse `FileWriter` die Benutzung einer Bridge-Klasse, verliert aber an Flexibilität, da immer ein Default-Encoding verwendet wird, um die Zeichen zu schreiben.

19.4.5 Ein- und Ausgabe von Zeichenketten

Die folgenden Zeilen erzeugen ein Objekt vom Typ `OutputStreamWriter`, welches die ihm übergebene Zeichenkette mit dem **Default Encoding** in die Datei `Text.txt` schreibt:

```
FileOutputStream fos = new FileOutputStream ("Text.txt");
OutputStreamWriter osw = new OutputStreamWriter (fos);
osw.write ("Sehr geehrter Herr Mustermann");
```

Das **Default Encoding** kann mit Hilfe des Schlüssels `file.encoding` von den Systemeigenschaften abgefragt werden. Unter Windows ist im Falle der Spracheinstellung "deutsch" das Default Encoding "Cp1252". Cp1252 bedeutet Codepage 1252 und umfasst den westeuropäischen Zeichensatz.

Soll dagegen ein anderes Character Encoding benutzt werden, so muss dieses im Konstruktor der Klasse `OutputStreamWriter` angegeben werden:

```
OutputStreamWriter osw = new OutputStreamWriter (fos, "MacThai");
osw.write ("Sehr geehrter Herr Mustermann");
```

In diesem Fall wird die Zeichenkette "`Sehr geehrter Herr Mustermann`" per "MacThai" Encoding in die Datei geschrieben. Mit "MacThai" werden thailändische Texte auf einem Apple Computer dargestellt.

Ein Objekt der Klasse `OutputStreamWriter` erwartet als ersten Konstruktorparameter einen Parameter vom Typ `OutputStream`. Es kann also eine Referenz auf ein Objekt übergeben werden, deren Klasse von der Klasse `OutputStream` abgeleitet ist. Ein Objekt der Bridge-Klasse `OutputStreamWriter` aggregiert ein Objekt vom Typ `OutputStream`, wie z. B. ein Objekt vom Typ `FileOutputStream`:

Bild 19-13 Ein Objekt der Bridge-Klasse `OutputStreamWriter` aggregiert ein Objekt der Klasse `FileOutputStream`

Zur Ausgabe der Zeichenkette im Unicode-Format in der "Little-Endian"-Byte-Reihenfolge können folgende Anweisungen verwendet werden:

```
FileOutputStream fos = new FileOutputStream ("Text.txt");
OutputStreamWriter osw
              = new OutputStreamWriter (fos, "UnicodeLittle");
osw.write ("Sehr geehrter Herr Mustermann");
```

Das folgende **Beispielprogramm** schreibt die Zeichenkette "Gelb" mit unterschiedlichen Encodings in Dateien. In die Datei `TextUnicodeBig.txt` wird in der "Big-Endian"-Byte-Reihenfolge geschrieben und in die Datei `TextUnicodeLittle.txt` wird in der "Little-Endian"-Byte-Reihenfolge geschrieben. Anschließend wird der Inhalt der beiden Dateien byteweise ausgelesen und ausgegeben, um die unterschiedlichen Encodings sichtbar zu machen.

Hier das Programm:

```
// Datei: AusgabeVonZeichenketten.java

import java.io.*;

public class AusgabeVonZeichenketten
{
   public static void main (String[] args)
   {
      String s = "Gelb";

      try ( FileOutputStream fos =
               new FileOutputStream ("TextUnicodeBig.txt");
            OutputStreamWriter osw =
               new OutputStreamWriter (fos, "UnicodeBig"))
      {
         osw.write (s);
      }
      catch (IOException ex)
      {
         System.err.println ("IO-Fehler beim Schreiben");
         ex.printStackTrace();
      }
```

```
      try (FileOutputStream fos =
              new FileOutputStream ("TextUnicodeLittle.txt");
           OutputStreamWriter osw =
              new OutputStreamWriter (fos, "UnicodeLittle"))
      {
          osw.write (s);
      }
      catch (IOException ex)
      {
         System.err.println ("IO-Fehler beim Schreiben");
         ex.printStackTrace();
      }

      dateiInhaltLesen ("TextUnicodeBig.txt");
      dateiInhaltLesen ("TextUnicodeLittle.txt");
   }

   private static void dateiInhaltLesen (String d)
   {
      try (FileInputStream fis = new FileInputStream (d))
      {
         System.out.println();
         int b = 0;

         while (true)
         {
            b = fis.read();

            if (b == -1)
            {
               break;
            }
            System.out.print (b);
            System.out.print (" ");
         }
      }
      catch (IOException ex)
      {
         System.err.println ("IO-Fehler beim Lesen");
         ex.printStackTrace();
      }
   }
}
```

Die Ausgabe des Programms ist:

```
254 255 0 71 0 101 0 108 0 98
255 254 71 0 101 0 108 0 98 0
```

Um beim Einlesen von Unicode-Zeichen zwischen den beiden Byte-Reihenfolgen "Little-Endian" und "Big-Endian" unterscheiden zu können, werden die Bytes `FE` (254) und `FF` (255) als erste zwei Bytes in eine Datei geschrieben. Wird beim Einlesen zuerst das Byte `FE` (254) und dann `FF` (255) gelesen, so werden die nachfolgenden Bytes entsprechend der "Big-Endian"-Byte-Reihenfolge interpretiert – wird

erst `FF` (255) und dann `FE` (254) gelesen, so werden die nachfolgenden Zeichen entsprechend der "Little-Endian"-Byte-Reihenfolge interpretiert. Die Ausgabe des Programms zeigt in der ersten Zeile die Zeichenkette `"Gelb"`, codiert im "Big-Endian" Unicode-Format. Die zweite Zeile repräsentiert die Zeichenkette `"Gelb"` im "Little-Endian" Unicode-Format. Die Zeichenkette `"Gelb"` entspricht den vier ASCII-Werten `71`, `101`, `108` und `98`.

19.5 Standardeingabe und Standardausgabe

Die Standardeingabe bzw. die Standardausgabe verwenden beide einen Bytestream, um Daten auf dem Bildschirm auszugeben bzw. um Daten von der Tastatur einzulesen.

Die Standardausgabe verwendet ein **Objekt vom Typ `PrintStream`** und die Standardeingabe verwendet ein **Objekt vom Typ `InputStream`**.

Über die Referenz `out` können alle Methoden, welche die Klasse `PrintStream` zur Verfügung stellt, aufgerufen werden – genauso können über die Referenz `in` alle Methoden, welche die Klasse `InputStream` zur Verfügung stellt, aufgerufen werden. Die Referenzen `out` und `in` sind dabei Klassenvariablen der Klasse **`System`** (siehe Anhang C).

Die Klasse `PrintStream` implementiert insbesondere Methoden zur Textausgabe. Hierzu benutzt sie überladene Varianten der Methoden **`print()`** und **`println()`**. So gibt es die Methoden `print()` und `println()` z. B. mit einem Übergabeparameter vom Typ `int`, `long`, `float`, `double`, `String` oder sogar `Object`.

Dabei werden alle Daten in deren String-Repräsentation ausgegeben. Das heißt, wenn ein `int`-Wert 2 ausgegeben wird, werden nicht die vier Bytes 00 00 00 02 ausgegeben, sondern das Zeichen `'2'`.

Wird der Methode `print()` bzw. `println()` eine Referenz als Übergabeparameter übergeben, so wird der String ausgegeben, den die `toString()`-Methode des referenzierten Objektes zurückgibt oder der String `"null"`, wenn die Referenz die `null`-Referenz ist.

Die Methoden der Klasse `InputStream`, die mit der Referenz `in` der Klasse `System` zum Einlesen von der Tastatur benutzt werden können, sind ausführlich in Kapitel 19.3.2.1 erläutert.

19.5.1 Formatierte Ausgabe

In Kapitel 4.7 wurde bereits die Methode **`printf()`** der Klasse `PrintStream` vorgestellt. Die Methode `printf()` ermöglicht eine formatierte Ausgabe auf dem Bild-

schirm. Diese Art der formatierten Ausgabe wird auch durch die Methode **`format()`** der Klasse `java.lang.Formatter` und durch die Methode `format()` der Klasse `String` bereitgestellt.

Die im Formatstring angegebenen Formatelemente von `printf()` folgen einem allgemeinen Aufbau. Elemente in eckigen Klammern [ ] sind dabei optional:

%[Parameterindex$][Steuerzeichen][Feldbreite][.Genauigkeit]Umwandlungszeichen

Im einfachsten Fall werden verschiedene Parameter nacheinander ausgegeben. Zu jedem Parameter gehört dabei jeweils ein eigenes Formatelement, wobei die Reihenfolge der auszugebenden Parameter der Reihenfolge der Formatelemente entspricht.

Mit Hilfe des **Parameterindex** kann die Ausgabereihenfolge der Parameter von der Reihenfolge der Parameter in der Parameterliste von `printf()` abweichen. Dies wird in den folgenden **Beispielen** gezeigt:

```
// Ausgabe entsprechend der Reihenfolge der Parameter.
System.out.printf ("%d %d %n", 3, 4);

// Erst wird der zweite Parameter ausgegeben, dann der erste.
System.out.printf ("%2$d %1$d %n", 3, 4);

// Es wird zweimal der zweite Parameter ausgegeben. Der erste
// Parameter wird nicht ausgegeben.
System.out.printf ("%2$d %2$d %n", 3, 4);
```

Die Ausgabe des Programmausschnitts ist:

```
3 4
4 3
4 4
```

Steuerzeichen (auch flags genannt) sind zusätzliche Formatangaben, mit denen die formatierte Ausgabe erweitert werden kann. Beispielsweise kann mit `'-'` die Ausrichtung der Ausgabe linksbündig erfolgen oder bei der Formatierung von Zahlen mit dem Steuerzeichen `'+'` stets das Vorzeichen mit ausgegeben werden.

Durch die Angabe der **Feldbreite** kann die minimale Anzahl auszugebender Zeichen festgelegt werden. Wird für den Wert des Parameters weniger Platz benötigt, so werden Leerzeichen vorangestellt. Das Feld für die **Genauigkeit** legt in der Regel die Breite der Ausgabe, d. h. die maximale Anzahl an Zeichen für die Ausgabe des Parameters, fest. Wird für den Wert des Parameters mehr Platz benötigt, so werden in der Ausgabe die Zeichen abgeschnitten, welche die maximale Anzahl übersteigen. Bei Gleitkommazahlen bezieht sich die Genauigkeit auf die Anzahl der Stellen hinter dem Komma. Die **Umwandlungszeichen** bestimmen das eigentliche Format der Ausgabe und sind vom Typ des Parameters abhängig. Es werden Parameter aller numerischen Typen wie `int`, `byte`, `long`, `Integer`, `Double`, `Float`, usw. unterstützt,

sowie verschiedene Bibliotheksklassen wie `Calendar` und `Date`. Zudem gibt es Umwandlungszeichen, die generell auf Datentypen angewendet werden können. Dies sind z. B. die Umwandlungszeichen `'s'` oder `'S'`, welche die Methode `toString()` des angegebenen Parameters benutzen.

Bei Formatelementen für die Ausgabe von Datum und Zeit entfällt die Option Genauigkeit, allerdings benötigen sie für alle Umwandlungszeichen das Präfix `'t'` (klein geschrieben) oder `'T'` (groß geschrieben). Das groß geschriebene `T` bewirkt dabei die Ausgabe in Großbuchstaben. Hier ein **Beispiel** für das Umwandlungszeichen bei der Datumsformatierung:

```
// Ein Calendar-Objekt mit dem Datum 31.12.2009 anlegen
Calendar calendar = new GregorianCalendar(2009, 11, 31);

// Zu beachten ist, dass der Monat Dezember durch die Zahl 11
// angegeben wird.

// Den Monat in normaler Schreibweise ausgeben
System.out.printf("%tB %n", calendar);

// Den Monat in Großbuchstaben ausgeben
System.out.printf("%TB %n", calendar);
```

Die Ausgabe des Programmausschnitts ist:

```
Dezember
DEZEMBER
```

Die nachfolgende Tabelle enthält die wichtigsten Umwandlungszeichen:

Umwandlungszeichen	**Typ des Parameters**	**Ausgabe**
`c,C`	Zeichen	Ein einzelnes Zeichen (auch aus einer Zahl)
`s,S`	Generell	Darstellung als Zeichenkette
`d`	Ganzzahl	Dezimale Darstellung
`o`	Ganzzahl	Oktale Darstellung (zur Basis 8)
`x,X`	Ganzzahl	Hexadezimale Darstellung (zur Basis 16)
`f`	Gleitkommazahl	Dezimale Darstellung
`e,E`	Gleitkommazahl	Darstellung als Exponentialzahl
`t,T`	Datum/Zeit	Präfix für alle Datum- und Zeitausgaben
`H`	Zeit	Stunden der 24 Stunden Uhr: 00 - 24
`I`	Zeit	Stunden der 12 Stunden Uhr: 01 - 12
`M`	Zeit	Minuten: 00 - 59
`S`	Zeit	Sekunden: 00 - 59
`B`	Datum	Monatsname "Januar", "Februar", etc.
`b`	Datum	Monatsname abgekürzt zu "Jan", "Feb", etc.
`Y`	Datum	Jahreszahl im vierstelligem Format wie z. B. "2005"
`n`	-	Zeilenumbruch

Tabelle 19-1 Die wichtigsten Umwandlungszeichen für `printf()` und `format()`

Die Umwandlungszeichen, die auch in Großbuchstaben angegeben werden können, bewirken, dass die jeweilige Ausgabe in Großbuchstaben umgewandelt wird. Für eine ausführliche Aufstellung aller unterstützten Formatelemente wird auf die Dokumentation der Java-API für die Klasse `java.util.Formatter` verwiesen.

19.5.2 Die Methode `format()`

Mit Hilfe der Methode `format()` der Klasse `String` kann eine formatierte Ausgabe in eine Stringvariable erfolgen.

Die Formatelemente von `format()` stimmen mit denjenigen von `printf()` überein und sind auszugsweise in Tabelle 19-1 dargestellt.

Im Folgenden wird gezeigt, wie mit der Methode `format()` ein String konstruiert wird:

```
// Das aktuelle Datum wird mit der Klasse Calendar bestimmt
Calendar calendar = Calendar.getInstance();

// Mit dem Parameterindex 1$ wird auf denselben Parameter
// mehrfach zugegriffen. Hier wird der aktuelle Tag, der Monat
// und das Jahr aus dem Objekt calendar hintereinander in
// einen String geschrieben.
String s = String.format ("%1$td %1$tB %1$tY", calendar);
```

Das folgende **Beispielprogramm** zeigt die Anwendung von `format()` und `printf()`:

```
// Datei: FormatTest.java

import java.util.Calendar;

public class FormatTest
{
   // Klassenmethode zur formatierten Ausgabe von Datum und Zeit
   public static void main (String[] args)
   {
      // Das aktuelle Datum wird mit Hilfe der Klasse Calendar
      // bestimmt.
      Calendar calendar = Calendar.getInstance();

      // Mit dem Parameterindex 1$ wird auf denselben Parameter
      // mehrfach zugegriffen.
      String datum = String.format ("%1$td %1$tB %1$tY", calendar);
      String zeit = String.format ("Uhrzeit: %1$tH:%1tM", calendar);

      // Die Reihenfolge der Parameter kann durch die
      // Parameterindices verändert werden. Hier wird an erster
      // Stelle das Datum ausgegeben.
      System.out.printf ("Heute ist der %2$s %n%1$s", zeit, datum);
   }
}
```

Die Ausgabe des Programms ist:

```
Heute ist der 19 Juli 2013
Uhrzeit: 10:49
```

19.6 Ein Beispiel für die Anwendung von Stream-Klassen

Bevor nun auf eine weitere Art von Stream-Klassen eingegangen wird, soll hier an einem **Beispiel** das Verwenden von Stream-Klassen gezeigt werden. Hierfür soll ein Adressbuch entwickelt werden, das die Anschrift und die Handy-Nummer von Personen verwalten kann. Dabei sollen die Daten von der Tastatur eingelesen und später wieder auf dem Bildschirm ausgegeben bzw. in eine Datei geschrieben werden. Um das Adressbuch realisieren zu können, werden die Klassen `Person`, `Addressbuch` und `Test` benötigt.

Als erstes wird die Klasse `Person` vorgestellt:

```
// Datei: Person.java

import java.io.*;
import java.util.*;

public class Person
{
   private String name, vorname, str, ort, handy;
   private int hausNr, plz;

   public Person (Scanner in)
   {
      vorname = in.next();
      name = in.next();
      str = in.next();
      hausNr = in.nextInt();
      plz = in.nextInt();
      ort = in.next();
      handy = in.next();
   }

   public void ausgeben (PrintStream out)
   {
      out.printf ("%-10s %-10s\n%-12s %-3d\n%-5s %-10s\n%-14s\n",
                  vorname, name, str, hausNr, plz, ort, handy);
   }
}
```

Die Klasse `Person` hat einen Konstruktor mit einem Parameter vom Typ `Scanner`. Ein Objekt der Klasse `Scanner` kann auf eine beliebige Datenquelle zeigen und von dort Zeichen einlesen. Die Datenquelle kann somit die Tastatur (`System.in`) oder eine Datei sein. Mit Hilfe des Scanners werden über die `next . . .()`-Methoden die Daten eingelesen und den einzelnen Attributen der Klasse `Person` zugewiesen. Beim Aufruf der Methode `next. . .()` liest der Scanner so lange Zeichen aus der

Datenquelle, bis er auf ein Whitespace-Zeichen – z. B. ein Leerzeichen oder einen Zeilenumbruch – stößt und dann die gelesenen Daten an den Aufrufer zurückgibt.

Mit Hilfe der Methode `ausgeben()` der Klasse `Person` soll es nun möglich sein, die Daten sowohl auf den Bildschirm auszugeben, als auch in eine Datei zu schreiben. Das funktioniert nur, wenn ein Objekt der Klasse `PrintStream` als Parameter der Methode `ausgeben()` übergeben wird. Bei der Ausgabe auf dem Bildschirm muss dabei `System.out` als Parameter übergeben werden. Beim Schreiben in eine Datei muss das Objekt der Klasse `PrintStream` auf eine Datei verweisen.

Als nächstes soll die Klasse `Adressbuch` betrachtet werden. Die Daten des Adressbuches sollen in einer Liste verwaltet und in einer Datei abgelegt werden können. Von dieser Datei sollen die bereits im Adressbuch geführten Personen eingelesen werden können. Zusätzlich sollen neue Personen in das Adressbuch aufgenommen und in eine Datei gespeichert werden können. Alle im Adressbuch verwalteten Personen sollen auf dem Bildschirm ausgegeben werden können.

Hier die Klasse `Adressbuch`:

```
// Datei: Adressbuch.java
import java.io.*;
import java.util.*;

public class Adressbuch
{
   private String dateiname = null;
   private ArrayList<Person> liste = new ArrayList<Person>();

   public Adressbuch (String dateiname)
   {
      this.dateiname = dateiname;
      try (FileInputStream fis = new FileInputStream (dateiname))
      {
         einlesen (fis);
      }
      catch (FileNotFoundException e)
      {
         System.out.println (dateiname + " wird angelegt!");
      }
      catch (IOException e)
      {
         System.err.println("Fehler beim Schliessen der Datei!");
      }
   }

   private void einlesen (InputStream in)
   {
      Scanner scan = new Scanner (in);
      // Solange noch weitere Elemente in der Datei sind,
      // werden Adressdaten von Personen eingelesen.
      while (scan.hasNext())
      {
         addAdresse (new Person (scan));
      }
   }
```

```
   // Methode, um dem Adressbuch einen Eintrag hinzuzufügen.
   public void addAdresse (Person person)
   {
      liste.add (person);
   }

   // Methode, um das Addressbuch zu speichern.
   public void speichern()
   {
      try (PrintStream print = new PrintStream (
          new FileOutputStream (dateiname))
      {
         ausgeben (print);
      }
      catch (IOException ex)
      {
         System.err.println ("IO-Fehler beim Schreiben");
         ex.printStackTrace();
      }
   }

   // Adressbuch auf den Bildschirm oder in eine Datei schreiben.
   public void ausgeben (PrintStream out)
   {
      for (Person person : liste)
      {
         person.ausgeben (out);
      }
   }
}
```

Die Daten des Adressbuches werden in einem Objekt der Klasse `ArrayList-<Person>` verwaltet. Diese Daten werden im Konstruktor aus einer Datei gelesen. Hierfür wird dem Konstruktor des Adressbuches der Dateiname übergeben. Ist die Datei vorhanden, werden mit Hilfe eines Objektes der Klasse `FileInputStream` alle Adressdaten aus der Datei mit der Methode `einlesen()` eingelesen. Dabei werden mit Hilfe eines Objekts der Klasse `Scanner` die Attribute der Klasse `Person` in deren Konstruktor aus dem Objekt der Klasse `Scanner` bzw. dem referenzierten Objekt der Klasse `InputStream` eingelesen. Die jeweils angelegten Objekte der Klasse `Person` werden in der Methode `addAdresse()` der Liste `liste` mit der Methode `add()` hinzugefügt. Die Methode `speichern()` erzeugt ein Objekt der Klasse `FileOutputStream` und übergibt dieses beim Instanziieren dem Konstruktor der Klasse `PrintStream`. Danach wird das Objekt der Klasse `PrintStream` der Methode `ausgeben()` übergeben und die gesamten Adressdaten mit Hilfe der übergebenen Referenz in eine Datei geschrieben. Wird die Methode `ausgeben()` mit dem Parameter `System.out` aufgerufen, so werden die Daten anstatt in eine Datei auf den Bildschirm geschrieben.

Zum Abschluss noch die Klasse `Test`, um das Adressbuch ausführen zu können:

```
// Datei: Test.java

import java.util.*;
```

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

      Adressbuch buch = new Adressbuch (args[0]);

      System.out.println("Bitte in folgendem Format eingeben:");
      System.out.println("Vorname Nachname Str. Nr. PLZ Ort Handy");

      // Erstelle einen Scanner, um die Daten einlesen zu können
      Scanner scanner = new Scanner (System.in);

      // Neue Einträge in das Adressbuch aufnehmen
      do
      {
          buch.addAdresse (new Person(scanner));
          System.out.print("Weitere Adressen? ");
      }
      while ("ja".equals(scanner.next()));

      // Adressbuch auf dem Bildschirm ausgeben
      System.out.println ("******** Adressbucheinträge ********");
      buch.ausgeben (System.out);
      buch.speichern();
   }
}
```

Die Ausgabe des Programms ist:

```
Adressbuch.txt wird angelegt!
Bitte in folgendem Format eingeben:
Vorname Nachname Str. Nr. PLZ Ort Handy
Anna Klein Mozartstr. 17 72878 Leonberg 0172/7878878
Weitere Adressen? ja
Chris Valentin Enzweg 35 89898 Mainz 0173/6564343
Weitere Adressen? nein
******** Adressbucheinträge ********
Anna       Klein
Mozartstr.   17
72878 Leonberg
0172/7878878
Chris      Valentin
Enzweg       35
89898 Mainz
0173/6564343
```

In der Methode `main()` wird zuerst ein Objekt der Klasse `Adressbuch` angelegt. Dabei wird dem Konstruktor der Dateiname als erster Kommandozeilenparameter übergeben. Die zwei darauffolgenden Zeilen geben das Eingabeformat für die Adressen auf dem Bildschirm aus. Danach wird ein Objekt der Klasse `Scanner` mit dem Parameter `System.in` instanziiert. In der `do while`-Schleife werden nun so lange Adressen eingelesen und dem Adressbuch hinzugefügt, solange der Benutzer die Frage `"Weitere Adressen?"` mit `"ja"` beantwortet. Ansonsten wird die Schleife verlassen. Zum Schluss werden die Adressen auf dem Bildschirm ausgegeben und in einer Datei gespeichert.

Beim ersten Aufruf des Programms ist noch kein Adressbuch vorhanden. Die ersten Adressdaten können aber eingeben werden. Der Aufruf des Programms erfolgt mit:

```
java Test Adressbuch.txt
```

Beim erneuten Ausführen des Programms ist die Datei `Adressbuch.txt` nun bereits vorhanden und enthält auch schon Adressen. Diese Adressen werden bei jedem Aufruf des Programms eingelesen und können um neue Adressen erweitert werden.

19.7 Ein- und Ausgabe der Datenfelder von Objekten

In Java kann man nicht nur Werte elementarer Datentypen in eine Datensenke schreiben und aus einer Datenquelle lesen, sondern ganze Objekte. Werden die Werte der **Datenfelder eines Objektes in einen Bytestrom überführt, der sich wieder rekonstruieren lässt**, so spricht man von Objektserialisierung. Dieser Mechanismus wird im folgenden Kapitel erläutert.

19.7.1 Objektserialisierung

Mittels der Objektserialisierung können Objekte genauso wie elementare Datentypen in einen Bytestrom geschrieben werden. Ebenso ist es möglich, aus einem Bytestrom, der aus einer Datenquelle kommt, die Datenfelder eines Objektes wieder einzulesen.

Für die Überführung der Datenfelder eines Objektes in einen Bytestrom ist die Klasse `ObjectOutputStream` zuständig. Für die Rekonstruktion der Datenfelder eines Objektes beim Lesen aus einem Bytestrom, der aus einer Datenquelle kommt, ist die Klasse `ObjectInputStream` zuständig.

Das folgende Bild symbolisiert die Serialisierung eines Objektes:

Bild 19-14 Serialisierung eines Objektes

Die Klasse `ObjectOutputStream` stellt Schreibmethoden für alle elementaren Datentypen zur Verfügung, beispielsweise die Methode `writeDouble()`, die einen `double`-Wert in den Ausgabestrom schreibt, oder die Methode `writeInt()`, die

einen `int`-Wert in den Ausgabestrom schreibt. Außer den Schreibmethoden für die elementaren Datentypen existiert auch eine Methode `writeObject()`, die eine Referenz auf ein Objekt als Parameter übergeben bekommt und die Datenfelder dieses Objektes in den Ausgabestrom schreibt. Hier der Methodenkopf von `writeObject()`:

```
public writeObject (Object data) throws IOException
```

Entsprechend existieren in der Klasse `ObjectInputStream` Lesemethoden für elementare Datentypen und die Methode `readObject()`, um die Datenfelder eines Objektes aus einem Bytestrom auszulesen.

Hier der Methodenkopf der Methode `readObject()`:

```
public final Object readObject() throws OptionalDataException,
                                        ClassNotFoundException,
                                        IOException
```

Wird die Methode `writeObject()` aufgerufen, so wird das Objekt, auf das die übergebene Referenz zeigt, daraufhin überprüft, ob dessen Klasse die **Schnittstelle `Serializable`** implementiert. Die Schnittstelle `Serializable` kennzeichnet ein Objekt als serialisierbar. Implementiert die Klasse des zu serialisierenden Objektes die Schnittstelle `Serializable` nicht, so wird eine Exception vom Typ **`NotSerializableException`** geworfen.

Die **Schnittstelle** `Serializable` dient als **Kennzeichnung, dass** ein **Objekt einer Klasse serialisierbar** ist, d. h. als ein sogenanntes Marker-Interface[227]. Die Schnittstelle selbst deklariert keine Methoden.

Das folgende **Beispiel** zeigt, wie die Datenfelder eines Objektes in eine Datei geschrieben und wieder eingelesen werden:

```
// Datei: Serial.java

import java.io.*;

class Person implements Serializable
{
   private String name;
   private String vorname;
   private int alter;

   public Person (String name, String vorname, int alter)
   {
      this.name = name;
      this.vorname = vorname;
      this.alter = alter;
   }
```

[227] Marker-Interfaces werden in Kapitel 14.7 beschrieben.

```
   public void print()
   {
      System.out.println ("Name: " + name);
      System.out.println ("Vorname: " + vorname);
      System.out.println ("Alter: " + alter);
   }
}

public class Serial
{
   public static void main (String[] args)
   {
      // Datei text.txt zum Schreiben öffnen.
      try (ObjectOutputStream out =
         new ObjectOutputStream (new FileOutputStream ("text.dat")))
      {
         Person pers1 = new Person ("Weiss", "Renate", 12);
         Person pers2 = new Person ("Maier", "Anja", 13);

         // Ein paar primitive Datentypen in die Datei schreiben.
         out.writeInt (1);
         out.writeDouble (1.2);
         // Objekte pers1 und pers2 in die Datei schreiben.
         out.writeObject (pers1);
         out.writeObject (pers2);
      }
      catch (IOException ex)
      {

         System.err.println ("IO-Fehler beim Schreiben");
         ex.printStackTrace();
      }

      // Datei text.txt zum Lesen öffnen.
      try (ObjectInputStream in =
         new ObjectInputStream (new FileInputStream ("text.dat")))
      {
         // Datentypen wieder einlesen und ausgeben.
         System.out.println (in.readInt());
         System.out.println (in.readDouble());
         // Die Methode readObject() gibt eine Referenz vom Typ
         // Object zurück. Es muss ein expliziter Cast erfolgen,
         // damit die Methode print() der Klasse Person aufgerufen
         // werden kann.
         ((Person) in.readObject()).print();
         ((Person) in.readObject()).print();
      }
      catch (IOException ex)
      {
         System.err.println ("IO-Fehler beim Lesen");
         ex.printStackTrace();
      }
      catch (ClassNotFoundException ex)
      {
         System.err.println ("Klasse nicht gefunden");
         ex.printStackTrace();
      }
```

```
    }
}
```

Die Ausgabe des Programms ist:

```
1
1.2
Name: Weiss
Vorname: Renate
Alter: 12
Name: Maier
Vorname: Anja
Alter: 13
```

Das Ganze funktioniert problemlos – und dazu muss recht wenig Aufwand betrieben werden. Es darf lediglich nicht vergessen werden, dass die Klasse der zu serialisierenden Objekte die Schnittstelle `Serializable` implementieren muss. Dabei muss dies, da die Schnittstelle `Serializable` keine Methoden deklariert, nur in der Klassendeklaration mit `implements Serializable` angegeben werden. Wird die Angabe `implements Serializable` im Beispiel bei der Klasse `Person` weggelassen, so wird beim Ausführen des Programms die folgende Meldung ausgegeben:

```
java.io.NotSerializableException: Person
```

Bei der Serialisierung eines Objektes laufen folgende Schritte ab:

- Überprüfen, ob das zu serialisierende Objekt die Schnittstelle `Serializable` implementiert. Ist das nicht der Fall, wird eine Exception vom Typ `NotSerializableException` geworfen.
- Ein eindeutiger Identifikator der Klasse wird codiert in den Bytestrom geschrieben.
- Die Instanzvariablen werden der Reihe nach durch die entsprechende Methode der Klasse `ObjectOutputStream` in den Ausgabestrom geschrieben. Ist ein Datenfeld wiederum eine Referenz, so setzt sich dieser Mechanismus rekursiv beim ersten Schritt fort.

Bei der Rekonstruktion eines Objektes aus einem Eingabestrom werden folgende Schritte durchlaufen:

- Rekonstruktion der Klasse des Objektes.
- Laden der Klasse, wenn diese noch nicht in der Java-Virtuellen Maschine vorhanden ist. Kann die Klasse vom Klassenlader nicht gefunden werden, wird eine Exception vom Typ **`ClassNotFoundException`** geworfen. Für das zu rekonstruierende Objekt wird Speicher angelegt und die Instanzvariablen werden mit den Default-Werten belegt.
- Es werden alle Instanzvariablen der Reihe nach mit der entsprechenden Methode der Klasse `ObjectInputStream` eingelesen. Ist ein Datenfeld wiederum eine Referenz, so setzt sich der Vorgang rekursiv beim ersten Schritt fort.

19.7.2 Strom-eindeutige Bezeichner

Zu jeder Klasse, deren Objekte serialisiert bzw. deserialisiert werden sollen, berechnet die JVM einen eindeutigen Identifikator, die sogenannte **`serialVersionUID`**. Anhand dieser ID erkennt die JVM, ob sie für das serialisierte Objekt die passende Klasse geladen hat.

Dazu vergleicht sie die ID aus dem Eingabestrom mit der ID der geladenen Klasse. Schlägt der Vergleich fehl, wird eine Exception vom Typ `InvalidClassException` geworfen.

Hat man in einem verteilten System eine Klasse, welche das Marker-Interface `Serializable` implementiert, erst einmal verbreitet, ist man für immer an die ursprüngliche Implementierung dieser Klasse gebunden. Jede Änderung und Erweiterung führt dazu, dass sich der Identifikator `serialVersionUID` ändert. Wird also die interne Implementierung einer Klasse geändert, kann man mit dieser Klasse keine Objekte mehr einlesen, die mit der Vorgängerversion dieser Klasse geschrieben wurden. Daher wird eine Nutzung der Standardform der Serialisierung nicht empfohlen, da die physische Repräsentation der Klasse im Identifikator `serialVersionUID` codiert wird.

Diese Problematik kann man umgehen, indem man die `serialVersionUID` für eine Klasse selbst festlegt. Dies geht ganz einfach, indem man eine private Klassenvariable jeder serialisierbaren Klasse hinzufügt:

```
private static final long serialVersionUID = zufallsLongZahl;
```

Welcher Wert hier eingetragen wird, ist prinzipiell nicht von Bedeutung, doch sollte er für diese Klasse eindeutig sein. Üblicherweise lässt man sich den Wert für die Klasse durch das Programm `serialver` errechnen. Das Programm `serialver` liegt im gleichen Verzeichnis wie der Compiler `javac` und der Interpreter `java`. Der Zugriffsmodifikator sollte immer `private` sein, da abgeleitete Klassen nicht dieselbe `serialVersionUID` haben dürfen. Das heißt, in jeder Subklasse ist eine neue eindeutige `serialVersionUID` festzulegen.

Einige Entwicklungsumgebungen bemerken das Fehlen des Datenfeldes `serialVersionUID` in einer Klasse vom Typ `Serializable` und geben eine Warnung aus.

19.7.3 Das Schlüsselwort `transient`

In Kapitel 19.7.1 wurde automatisch jede Instanzvariable eines Objektes in den Ausgabestrom geschrieben bzw. aus einem Eingabestrom rekonstruiert. Es kann aber auch durchaus vorkommen, dass es Instanzvariablen gibt, die nicht serialisiert werden sollen. Zum Beispiel kann es aus Sicherheitsgründen erforderlich sein, dass ein Passwort nicht in einer Datei abgespeichert werden soll. Manchmal werden auch Instanzvariablen angelegt, die nur temporäre Werte annehmen. Speichert eine Instanzvariable zum Beispiel, ob der rechte Mausknopf gedrückt ist oder nicht, so kann

eine solche Information für eine persistente (dauerhafte) Speicherung uninteressant sein.

Die **Serialisierung von Instanzvariablen** kann **unterdrückt werden**, indem das **Schlüsselwort** `transient` bei der Deklaration der Instanzvariablen angegeben wird.

Das folgende **Beispiel** zeigt, wie verhindert wird, dass das Datenfeld `alter` einer Klasse `Person` serialisiert wird:

```
public class Person implements Serializable
{
   private String name;
   private String vorname;
   private transient int alter;
   . . . . .
}
```

Wird nun ein Objekt der Klasse `Person` mit dem Aufruf der Methode `writeObject()` in einen Ausgabestrom geschrieben, so werden nur die beiden Datenfelder `name` und `vorname` serialisiert. Genauso wird bei der Rekonstruktion das Datenfeld `alter` nicht berücksichtigt.

19.7.4 Die Schnittstelle `Externalizable`

Die Objektserialisierung durch die Implementierung der Schnittstelle `Serializable` ist sehr komfortabel. Möchte man jedoch Einfluss auf die Serialisierung nehmen, so kann dies nicht mit Hilfe der Schnittstelle `Serializable` erfolgen. Zum Beispiel kann eine Abänderung des Formates, in welchem die Objekte in den Ausgabestrom geschrieben werden, von Interesse sein.

Will man bei der Serialisierung Einfluss auf das Format oder den Inhalt nehmen, so muss man die Schnittstelle `Externalizable` verwenden.

Die Schnittstelle `Externalizable` ist von `Serializable` abgeleitet und erweitert diese um die folgenden beiden Methoden:

```
public abstract void writeExternal (ObjectOutput out)
                     throws IOException
public abstract void readExternal (ObjectInput in)
                     throws IOException,
                            ClassNotFoundException
```

Um nun Einfluss auf die Serialisierung eines Objektes zu nehmen, muss in dem zu serialisierenden Objekt die Schnittstelle `Externalizable` implementiert werden.

Das folgende Beispiel benutzt die Schnittstelle `Externalizable`:

```
// Datei: Serial2.java

import java.io.*;

class Person2 implements Externalizable
{
   private String name;
   private String vorname;
   private int alter;
   private static final long serialVersionUID = 1234;

   public Person2()
   {
      System.out.println ("Hallo, hier im Default Konstruktor");
   }

   public Person2 (String name, String vorname, int alter)
   {
      this.name = name;
      this.vorname = vorname;
      this.alter = alter;
   }

   public void print()
   {
      System.out.println ("Name: " + name);
      System.out.println ("Vorname: " + vorname);
      System.out.println ("Alter: " + alter);
   }

   public void writeExternal (ObjectOutput out) throws IOException
   {
      System.out.println ("Explizites Schreiben!");
      out.writeObject (name);
      out.writeObject (vorname);
      out.writeInt (alter);
   }

   public void readExternal (ObjectInput in) throws IOException,
                                                ClassNotFoundException
   {
      System.out.println ("Explizites Lesen!");
      name = (String) in.readObject();
      vorname = (String) in.readObject();
      alter = in.readInt();
   }
}

public class Serial2
{
   public static void main (String[] args)
   {
      try (ObjectOutputStream out =
         new ObjectOutputStream (new FileOutputStream ("text.dat")))
      {
         Person2 pers1 = new Person2 ("Mustermann", "Heinz", 45);
         Person2 pers2 = new Person2 ("Heinzelmann", "Max", 30);
```

```
         // Ein paar primitive Datentypen in die Datei schreiben.
         out.writeInt (1);
         out.writeDouble (1.2);
         // Objekte pers1 und pers2 in die Datei schreiben.
         out.writeObject (pers1);
         out.writeObject (pers2);
      }
      catch (IOException ex)
      {
         System.err.println ("IO-Fehler beim Schreiben");
         ex.printStackTrace();
      }

      try (ObjectInputStream in =
         new ObjectInputStream (new FileInputStream ("text.dat")))
      {
         // Datentypen wieder einlesen und ausgeben.
         System.out.println (in.readInt());
         System.out.println (in.readDouble());
         ((Person2) in.readObject()).print();
         ((Person2) in.readObject()).print();
      }
      catch (IOException ex)
      {
         System.err.println ("IO-Fehler beim Lesen");
         ex.printStackTrace();
      }
      catch (ClassNotFoundException ex)
      {
         System.err.println("Klasse nicht gefunden");
         ex. printStackTrace();
      }
   }
}
```

Die Ausgabe des Programms ist:

```
Explizites Schreiben!
Explizites Schreiben!
1
1.2
Hallo, hier im Default Konstruktor
Explizites Lesen!
Name: Mustermann
Vorname: Heinz
Alter: 45
Hallo, hier im Default Konstruktor
Explizites Lesen!
Name: Heinzelmann
Vorname: Max
Alter: 30
```

Im Programm wird der Methode `writeObject()` eine Referenz vom Typ `Person` übergeben. Die Methode `writeObject()` prüft, ob die Referenz auf ein Objekt zeigt, das vom Typ `Externalizable` oder vom Typ `Serializable` ist. Ist das Ob-

jekt vom Typ `Externalizable` – wie im Falle der Klasse `Person` – wird die Methode `writeExternal()` aufgerufen. Wird ein Objekt mit `readObject()` eingelesen, so wird ebenfalls überprüft, ob die Klasse des zu rekonstruierenden Objektes die Schnittstelle `Externalizable` oder `Serializable` implementiert. Implementiert die Klasse die Schnittstelle `Externalizable`, so wird die Methode `readExternal()` aufgerufen.

Zusätzlich zur Implementierung der Schnittstelle `Externalizable` wurde noch ein selbst geschriebener parameterloser Konstruktor eingeführt. Dieser wird bei der Rekonstruktion der Objekte aufgerufen. Nach dem Aufruf des parameterlosen Konstruktors wird die Methode `readExternal()` aufgerufen. Ist kein parameterloser Konstruktor mehr vorhanden, weil er zum Beispiel durch einen Konstruktor mit Parametern überschrieben wurde, so wird zur Laufzeit folgende Fehlermeldung ausgegeben:

```
java.lang.NoSuchMethodError
```

19.8 Zusammenfassung

Programme verarbeiten Daten, die von unterschiedlichen Datenquellen (Tastatur, Dateien, Netzverbindungen) gelesen und in unterschiedliche Datensenken (Bildschirm, Dateien, Netzverbindungen) geschrieben werden. Auch die Übertragung kann auf unterschiedliche Arten erfolgen (Bytes, Zeichen oder Zeilen). Um dieser Problematik gerecht zu werden, wurde in Java das sogenannte Stream-Konzept entwickelt, das in Kapitel 19 vorgestellt wird.

Um einen besseren Überblick über die Vielzahl der Stream-Klassen zu erhalten, werden diese in Kapitel **19.1** zunächst klassifiziert: In Stream-Klassen für Datensenken und Datenquellen, in Bytestream-Klassen zur Verarbeitung von Bytes und Characterstream-Klasse zur Verarbeitung von Zeichen und in Sink- bzw. Springstream-Klassen und Processingstream-Klassen, die die Funktionalität der Sink- bzw. Springstream-Klassen erweitern.

Das eigentliche Stream-Konzept, das die komfortable Benutzung von Streams erlaubt, wird in Kapitel **19.2** vorgestellt. Demnach besitzen die Processingstream-Klassen jeweils eine Referenz auf eine Sinkstream- bzw. Springstreamklasse und erweitern deren Funktionalität durch Methoden, die einen komfortableren Zugriff auf einen Stream erlauben.

Kapitel **19.3** stellt die Bytestream-Klassen vor, die notwendig sind, um mit externen Datensenken und Datenquellen byteweise umzugehen. Die Klasse `OutputStream` stellt die Basisklasse der Byte-Outputstream-Klassen dar und wird sowohl durch Sinkstream-Klassen als auch durch Processingstream-Klassen erweitert. Die Klasse `InputStream` stellt die Basisklasse der Byte-Inputstream-Klassen dar und wird durch Springstream-Klassen und ebenfalls durch Processingstream-Klassen erweitert. Darüber hinaus werden in Kapitel **19.3** Beispiele gegeben, um einzelne Bytes bzw. Byte-Arrays in eine Datei zu schreiben bzw. aus einer Datei zu lesen.

Die Characterstream-Klassen, die mit Zeichen anstatt einzelner Bytes arbeiten, werden in Kapitel **19.4** beschrieben. Characterstream-Klassen werden unterteilt in

Writer-Klassen und Reader-Klassen, die wiederum durch Sinkstream-Klassen und Processingstream-Klassen bzw. durch Springstream-Klassen und Processingstream-Klassen erweitert werden. Darüber hinaus existieren Bridge-Klassen, die eine Verbindung zwischen den Characterstream-Klassen und den Bytestream-Klassen herstellen.

Kapitel **19.5** zeigt auf, dass sowohl die Standardeingabe als auch die Standardausgabe mittels Bytestreams realisiert ist. Dabei verwendet die Standardausgabe ein Objekt vom Typ `PrintStream`, die Standardeingabe ein Objekt vom Typ `InputStream`.

Ein konkretes Beispiel für die Anwendung von Stream-Klassen wird in Kapitel **19.6** gegeben. Realisiert wird ein Adressbuch, das Adressbuchdaten von der Kommandozeile einliest und in einer Datei abspeichert. Beim erneuten Start des Beispielprogramms wird die abgespeicherte Datei eingelesen und dem Benutzer sowohl die neu eingegebenen als auch die von der Datei eingelesenen Adressbuchdaten über die Standardausgabe angezeigt.

Neben Werten elementarer Datentypen lassen sich in Java auch die Datenfelder ganzer Objekte in eine Datensenke schreiben bzw. aus einer Datenquelle lesen. Hierzu werden die Objekte in einen Bytestrom überführt, der sich wieder rekonstruieren lässt. Dieser Vorgang wird Objektserialisierung bzw. Objektdeserialisierung genannt und in Kapitel **19.7** vorgestellt. Um Objekte einer Klasse zu serialisieren, muss die entsprechende Klasse die Schnittstelle `Serializable` implementieren. Diese Schnittstelle deklariert keine Methoden und stellt somit ein sogenanntes Marker-Interface dar. Sollen innerhalb einer Klasse bestimmte Instanzvariablen von der Serialisierung ausgenommen werden, lässt sich dies durch das Schlüsselwort `transient` erreichen. Durch Implementierung der Schnittstelle `Externalizable` lässt sich auf die Serialisierung von Objekten weiteren Einfluss nehmen.

19.9 Übungen

Aufgabe 19.1: Streams

a) Schreiben Sie eine Java-Anwendung, die eine Text-Datei mit Hilfe der Klasse `FileReader` einliest und auf dem Bildschirm ausgibt. Der Name der auszugebenden Datei soll als Parameter an die Anwendung übergeben werden. Kann die Datei nicht geöffnet bzw. gelesen werden, so soll eine Fehlermeldung ausgegeben werden.

b) Erweitern Sie die Java-Anwendung, indem Sie die Zeilen einer Textdatei in umgekehrter Reihenfolge (die letzte Zeile der Datei zuerst) auf dem Bildschirm ausgeben. Sie können zur Implementierung die Klassen `java.lang.String` und `java.lang.StringBuilder` verwenden.

Aufgabe 19.2: Einmaleins

a) Schreiben Sie ein Programm, das die Zahlen des kleinen Einmaleins berechnet und in der Textdatei `einmaleins.txt` ablegt. Die Zahlen sollen jeweils durch einen Tabulator voneinander getrennt werden.

Die Textdatei `einmaleins.txt` sollte folgenden Inhalt haben:

```
1     2     3     4     5     6     7     8     9     10
2     4     6     8     10    12    14    16    18    20
3     6     9     12    15    18    21    24    27    30
4     8     12    16    20    24    28    32    36    40
5     10    15    20    25    30    35    40    45    50
6     12    18    24    30    36    42    48    54    60
7     14    21    28    35    42    49    56    63    70
8     16    24    32    40    48    56    64    72    80
9     18    27    36    45    54    63    72    81    90
10    20    30    40    50    60    70    80    90    100
```

b) Erweitern Sie das Programm so, dass die Größe der berechneten Tabelle durch den Benutzer angegeben werden kann. Die Größe soll als Parameter beim Programmaufruf übergeben werden, wie in folgendem Beispielaufruf:

```
java Einmaleins 100
```

Aufgabe 19.3: Filter

a) Machen Sie sich die Arbeitsweise der Klasse `Filter` klar und testen Sie diese mit einer beliebigen Textdatei.

Aufruf: `java Filter <Suchmuster> <Dateiname>`

Hier die Klasse `Filter`:

```
// Datei: Filter.java
import java.io.*;

public class Filter
{
   public static void main (String args[])
   {
      if (args.length != 2)
      {
         System.err.println ("Bitte 2 Parameter: ");
         System.err.println ("<Suchmuster> <Dateiname>");
         System.exit (1);
      }

      String patt = args [0];
      String fileIn = args [1];

      try (LineNumberReader reader = new LineNumberReader
                                     (new FileReader (fileIn)))
      {
         String str;

         while ((str = reader.readLine()) != null)
         {
            if (str.indexOf (patt) != -1)
            {
```

```
                    int ln = reader.getLineNumber();
                    System.out.println (fileIn + "[" + ln + "]: " +
                                                                str);
                }
            }
        }
        catch (IOException e)
        {
            System.err.println ("IO-Fehler beim Filtern");
            e.printStackTrace();
        }
    }
}
```

b) Modifizieren Sie die Klasse `Filter` so, dass die Ausgabe nicht auf dem Bildschirm erscheint, sondern in eine Datei geschrieben wird. Der Name der Datei, in die geschrieben werden soll, wird als dritter Parameter beim Programmaufruf übergeben.

Aufgabe 19.4: Umleiten

Schreiben Sie ein Programm, welches den Fehler-Stream in die Datei `error.log` umleitet. Testen Sie das Programm, indem Sie mehrere Textzeilen auf der Standardausgabe und in den Fehler-Stream ausgeben.

Aufgabe 19.5: Ein- und Ausgabe

a) Schreiben Sie ein Programm, welches Zufallszahlen vom Typ `int` in eine Datei schreibt.
b) Schreiben Sie ein weiteres Programm, welches diese Zahlen aus der Datei einliest und die Anzahl sowie die Summe und den Durchschnitt der Zahlen berechnet und auf dem Bildschirm ausgibt.

Aufgabe 19.6: Formatierte Datumsausgabe

Der internationale Standard ISO 8601[228] spezifiziert die Ausgabe von Datum und Zeit. Datums-Ausgaben erfolgen nach Notation JJJJ-MM-TT, wobei J für Jahr, M für Monat und T für Tag steht. Zeit-Ausgaben erfolgen nach der Notation hh:mm:ss, wobei h für Stunde (00-23), m für Minute und s für Sekunde steht.

Schreiben Sie ein Programm, welches das aktuelle Datum und die Uhrzeit nach dem Standard der ISO 8601 ausgibt und zur Ausgabe die `format()`-Methode der Klasse `String` verwendet.

Aufgabe 19.7: Objektserialisierung

Es soll die Klasse `BenutzerLogin` geschrieben werden. Diese Klasse kapselt einen Login mit den Attributen `name` vom Typ `String`, `passwort` vom Typ `String` und `online` vom Typ `boolean` und besitzt folgende Methoden:

- einen Konstruktor, der es ermöglicht, den Namen und das Passwort für den neu erzeugten Login zu setzen.

[228] Weitere Informationen zur ISO 8601 sind zu finden unter: http://de.wikipedia.org/wiki/ISO_8601.

- `anmelden (String password)` zum Vergleichen der Passwörter und zum Setzen von `online` auf `true`, falls die Passwörter gleich sind.
- `print()` zur Ausgabe des Benutzernamens und der Information, ob der Benutzer online ist.

a) Implementieren Sie die Klasse `BenutzerLogin`.

b) Erzeugen Sie ein Objekt der Klasse `BenutzerLogin` und melden Sie sich bei diesem Objekt an. Überprüfen Sie hierbei, ob der Anmeldevorgang erfolgreich war, indem Sie die Methode `print()` vor und nach dem Anmelden aufrufen.

c) Speichern Sie das Objekt mit Hilfe der Objektserialisierung (`ObjectOutputStream`) in einer Datei. Beachten Sie hierbei, dass die Variable `online` nicht serialisiert (`transient`) werden soll.

d) Stellen Sie das Objekt mit Hilfe der Klasse `ObjectInputStream` wieder her und überprüfen Sie, ob der Benutzer immer noch angemeldet ist, indem Sie die Methode `print()` des neu erzeugten Objektes aufrufen.

e) Schreiben Sie das Programm um, damit statt der Schnittstelle `Serializable` die Schnittstelle `Externalizable` verwendet wird. Passen Sie das Programm der neuen Schnittstelle an und beachten Sie, dass weiterhin die bisherige Funktionalität erhalten bleiben soll.

Aufgabe 19.8: Flughafen-Projekt – Singleton / Laden und Speichern

Die Klasse `Flughafen` soll nun so erweitert werden, dass sie das Singleton-Pattern implementiert. Das Singleton-Pattern wird in Kapitel 10.5.2 erklärt. Verändern Sie die Klasse `Client` so, dass der Zugriff auf den Flughafen nur noch über das Singleton-Pattern geschieht.

Die im Flughafensystem gehaltenen Informationen sollen nun beim Beenden des Programms gespeichert und beim erneuten Starten des Programms wiederhergestellt werden. Hierzu soll die Methode, welche für das Singleton-Pattern eingeführt wurde, erweitert werden. Diese Methode soll dabei folgendermaßen vorgehen: Existiert noch keine Instanz, wird erst versucht, eine Instanz durch Objektserialisierung aus der gespeicherten Datei wiederherzustellen. Schlägt dies fehl, so wird eine neue Instanz der Klasse `Flughafen` mit `new` erzeugt. Es muss noch eine weitere statische Methode implementiert werden, welche die Instanz der Klasse `Flughafen` vor Programmende mit Hilfe der Objektserialisierung in eine Datei speichert. Diese Methode muss manuell vor Programmende aufgerufen werden.

Begriffsverzeichnis

Abstraktion	**Abstraktion** ist immer damit verbunden, dass man sich mit dem **Wesentlichen** befasst und das Unwesentliche gedanklich weglässt. Die Abstraktion bei einem Objekt befasst sich mit der Festlegung des nach außen sichtbaren Verhaltens eines Objekts, d. h. mit der Definition der Schnittstellen der von außen aufrufbaren Exportmethoden eines Objekts.
Aggregation	Bei der **Aggregation** enthält ein Objekt eine **Referenz auf ein anderes Objekt**. Über die Referenz zeigt das aggregierende Objekt auf ein aggregiertes Objekt. Das aggregierte Objekt kann auch weiterleben, wenn das aggregierende Objekt zerstört wird, und umgekehrt.
Aktor	**Aktoren** sind die Schnittstellen eines Systems zu Fremdsystemen oder Geräten bzw. Nutzern.
Applet	Ein **Applet** ist ein kleines Programm, das in einer HTML-Seite eingebettet ist.
Archiv	**Archive** dienen zur Zusammenstellung mehrerer Dateien – in gepackter oder ungepackter Form – in einer einzigen Datei. Archivdateien haben die Dateierweiterung `jar`. `jar`-Dateien können außer den Bytecode-Daten der Java-Klassen beispielsweise auch Bilder und `xml`-Dateien enthalten. Es können aber auch weitere `jar`-Dateien in einer `jar`-Datei verpackt sein. Häufig enthält ein Archiv auch eine Manifest-Datei, in der beispielsweise Informationen wie die Manifest-Version oder der Name der Start-Klasse enthalten sind sowie bei signierten Archiven die Signatur-Informationen für Klassen.
Array	Ein **Array** ist eine Datenstruktur, die aus Komponenten aufgebaut ist. Das Besondere an einem Array ist, dass **alle Komponenten denselben Datentyp** besitzen müssen. Über einen Array-Index kann gezielt auf eine bestimmte Komponente des Arrays zugegriffen werden.
Basisklasse	siehe Superklasse
Basisklasse, abstrakte	Eine **abstrakte Basisklasse** dient zur Abstraktion in einer Klassenhierarchie. Von einer abstrakten Basisklasse können **keine Objekte instanziiert** werden.
Bibliotheksklasse	Eine **Bibliotheksklasse** ist eine Klasse, welche durch eine Klassenbibliothek zur Verfügung gestellt wird.
Bindung	Wird eine Methode aufgerufen, so ist der entsprechende Programmcode der Methode, d. h. der Methodenrumpf, auszuführen. Die **Zuordnung des Methodenrumpfes zum Aufruf der Methode**, d. h. dem Methodenkopf, nennt man **Bindung**.

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1

Bindung, frühe (statische)	Bei der **frühen Bindung** erfolgt die Zuordnung des Methodenrumpfes zum Methodenkopf der Aufrufstelle **zur Kompilierzeit**.
Bindung, späte (dynamische)	Bei der **späten Bindung** erfolgt die Zuordnung des Methodenrumpfes zum Methodenkopf der Aufrufstelle **zur Laufzeit** des Programms.
Bit	**Binärziffer**, d. h. eine Ziffer, die die Werte 0 oder 1 annehmen kann (engl. bit = **bi**nary digi**t**).
Block	Ein **Block** ist eine besondere Anweisung. Ein Block **kann keine, eine oder mehrere Anweisungen enthalten**, die in der Reihenfolge, wie sie notiert sind, abgearbeitet werden. Von der Syntax der Sprache her zählt ein Block als eine einzige Anweisung. Damit ist der Block ein elegantes Hilfsmittel an Programmstellen, an denen die Syntax der Sprache nur eine einzige Anweisung zulässt, tatsächlich aber mehrere Anweisungen erforderlich sind.
Browser	Ein **Browser** ist ein **Werkzeug**, um hierarchische Strukturen wie z. B. eine Verzeichnisstruktur auf einer Festplatte zu durchlaufen, d. h. um durch diese Struktur zu navigieren. Im Falle des Internets dient ein Browser **zur Navigation** durch das Internet und zur Darstellung der gefundenen Web-Seiten.
Byte	Ein **Byte** stellt eine Folge von 8 zusammengehörigen Bits dar.
Casten	Beim **Casten** wird ein Wert eines bestimmten Datentyps in einen anderen Datentyp gewandelt (**Typkonvertierung**). Hierzu ist eine Typverträglichkeit erforderlich.
Datenfeld	Ein **Datenfeld** ist eine Komponente einer Klasse bzw. eines Objektes, die einen einfachen Wert oder eine Referenz aufnehmen kann.
Datentyp	Ein **Datentyp** ist der **Bauplan für** eine **Variable**. Der Datentyp legt fest, welche Operationen auf einer Variablen möglich sind und wie die Darstellung (Repräsentation) der Variablen im Speicher des Rechners erfolgt. Mit der Darstellung wird festgelegt, wie viele Bytes die Variable im Speicher einnimmt und welche Bedeutung ein jedes Bit dieser Darstellung hat.
Datentyp, abstrakter	Ein **abstrakter Datentyp** deklariert abstrakte Operationen auf abstrakten Daten. Eine Klasse implementiert hierbei die abstrakten Operationen des abstrakten Datentyps als Methoden und definiert konkret die Struktur der Daten.
Datentyp, einfacher	Ein **einfacher Datentyp** ist ein Bauplan für **einfache Variablen**. Das sind Variablen, die atomar sind und nur einen einzigen Wert tragen können.
Datentyp, selbst definierter	**Selbst definierte Datentypen** sind dem Compiler standardmäßig nicht bekannt. Wenn die Programmiersprache hierfür die Sprachmittel anbietet, so ist es dem Programmierer möglich, eigene Datentypen zu erfinden, die für die Modellierung einer An-

wendung von Bedeutung sind, und diese dem Compiler bekannt zu machen. Java bietet hierfür das Sprachkonstrukt der Klasse (`class`) und des Aufzählungstyps (`enum`). Auch Bibliotheksklassen sind selbst definierte Datentypen.

Default-Konstruktor — Ein **Default-Konstruktor** ist ein Konstruktor ohne Übergabeparameter, der vom Compiler zur Verfügung gestellt wird, wenn kein selbst geschriebener Konstruktor für eine Klasse vorhanden ist.

Definition — Eine **Definition** einer **Variablen** umfasst die Deklaration und das Anlegen der Variablen.

Eine **Definition** einer **Methode** umfasst die Deklaration des Methodenkopfes und das Festlegen des Methodenrumpfes.

Bei der **Definition** einer **Klasse** wird zwischen einer normalen Klassendefinition (Typ `class`) und einem Aufzählungstyp (Typ `enum`) unterschieden. Eine Definition einer normalen Klasse umfasst die Deklaration des Klassennamens und das Festlegen des Klassenrumpfes. Optional können eine Vaterklasse und zu implementierende Schnittstellen angegeben werden. Eine Definition eines Aufzählungstyps umfasst den Namen und den Rumpf des Aufzählungstyps. Optional können bei der Definition eines Aufzählungstyps Schnittstellen angegeben werden, die implementiert werden müssen.

Deklaration — Eine **Deklaration** gibt dem Compiler einen Typ und den zugehörigen Namen an wie z. B. bei der Deklaration einer Variablen.

Entity-Klasse — Eine **Entity-Klasse** ist Daten tragend. Entity-Objekte entsprechen Objekten der Realität.

Exception — Eine **Exception** ist eine Ausnahme. In Java werden Exceptions als Klassen modelliert. Tritt eine Ausnahme im Programm auf, so kann von einer aufgerufenen Methode ein Objekt einer Exception an jeweils die aufgerufene Methode bis hin zur virtuellen Maschine "geworfen" werden. Natürlich ist es auch möglich, Exceptions in einem Exception Handler zu behandeln und nicht weiterzuwerfen.

Exemplar — siehe Objekt

GUI-Container — Ein **GUI-Container** ist eine GUI-Komponente, der beliebig viele andere GUI-Komponenten hinzugefügt werden können.

Gültigkeit — Die **Gültigkeit** einer Variablen bedeutet, dass an einer Programmstelle der Name einer Variablen dem Compiler durch eine Deklaration bekannt ist.

Hauptprogramm — Mit dem **Hauptprogramm** beginnt ein klassisches Programm seine Ausführung. In der Programmiersprache C heißt das Hauptprogramm `main()`. In Anlehnung daran beginnt eine Java-Anwendung ihre Ausführung bei der Methode `main()` einer Startklasse.

Information Hiding — Die Daten eines Objekts sind im Idealfall verborgen (**Information Hiding**) und können nur durch die Methoden des Objekts selbst manipuliert werden. Sie sind also nach außen nicht direkt sichtbar. Ein solches Objekt tritt mit seiner Umwelt im Idealfall nur über **wohldefinierte Schnittstellenmethoden** in Kontakt. Die Implementierungseigenschaften der Klasse, d. h. die Struktur der Datenfelder, die Methodenrümpfe der Exportmethoden und die privaten Hilfsmethoden sind ein Implementierungsgeheimnis. Nur die Schnittstellenmethoden und ihre Parameter werden offen gelegt.

Initialisierung — Der Vorgang der **Initialisierung** dient dazu, Variablen mit definierten **Anfangswerten** zu versehen.

Instanziierung — Unter **Instanziierung** versteht man das Erzeugen einer **Instanz einer Klasse** gemäß dem Bauplan der Klasse. Bei der Instanziierung wird eine Instanz der Klasse, mit anderen Worten ein **Exemplar (Objekt)** der Klasse, im Arbeitsspeicher **erzeugt**.

Instanz — siehe Objekt

Instanzmethode — Eine **Instanzmethode** kann nur direkt für eine Instanz (ein Objekt) aufgerufen werden. Eine Instanzmethode arbeitet auf den Datenfeldern des Objektes, den Instanzvariablen, kann aber auch auf Klassenvariablen zugreifen.

Instanzvariable — Eine **Instanzvariable** eines Objektes besitzt den in der Klasse angegebenen Namen und Typ und hat grundsätzlich **für jedes Objekt einen individuellen Wert**. So hat beispielsweise jedes Objekt der Klasse `ZweidimensionalerPunkt` seine individuellen Koordinaten `x` und `y`.

Interface-Klasse — In Objekten von **Interface-Klassen** wird die Abhängigkeit der Objekte einer Anwendung im Rahmen der Systemanalyse von der Hardware/Software der Aktoren gekapselt, damit Entity- und Kontroll-Objekte unabhängig von der speziellen Ausprägung eines Aktors werden. Interface-Objekte werden zwischen den betrachteten Entity-/Kontroll-Objekten und den Aktoren eingeschoben. Interface-Objekte haben kein Gegenstück in der Realität.

Invariante — Invarianten sind Bedingungen, die von allen Methoden einer Klasse, die von einem Kunden aufgerufen werden können, eingehalten werden müssen, um die Korrektheit zu gewährleisten. Man spricht auch von **Klasseninvarianten**. Jedes Objekt einer Klasse muss die Invarianten der zugehörigen Klasse erfüllen. Dabei kann es für verschiedene Kunden verschiedene Invarianten geben.

Iteration — Bei einer **Iteration** wird ein Verarbeitungsschritt mehrmals hintereinander ausgeführt.

Java-Anwendung
: Zu einer **Java-Anwendung** gehört eine Methode `main()`. In der Methode `main()` können Objekte geschaffen und Methoden von Objekten bzw. Klassen aufgerufen werden.

Java-Plattform
: Die **Java-Plattform** umfasst unter anderem die Programmiersprache Java, Werkzeuge zum Erstellen von Programmen, die Java-Virtuelle Maschine sowie eine Klassenbibliothek.

JDK
: Das **J**ava **D**evelopment **K**it stellt eine Entwicklungsumgebung dar, die aus Werkzeugen wie Compiler und Disassemblierer besteht, und darüber hinaus alle Klassen der Java-Klassenbibliothek bereitstellt.

Kapselung
: Daten und die Methoden, die auf diesen Daten arbeiten, werden nicht mehr getrennt wie in der klassischen Programmierung, sondern als Einheit betrachtet – **Daten und Methoden** sind zusammen **in einer Kapsel**. Sie verschmelzen zu einem Objekt bzw. einer Klasse. Diese **Kapselung** ist eines der wichtigsten Konzepte der objektorientierten Programmierung.

Klasse (class)
: Eine **Klasse** bildet ein Objekt der realen Welt in ein Schema ab, das der Compiler versteht, wobei ein Objekt beispielsweise ein Haus, ein Vertrag oder eine Firma sein kann – also prinzipiell jeder konkrete und abstrakte Gegenstand, der für einen Menschen eine Bedeutung hat und den er sprachlich beschreiben kann. Eine Klasse besteht aus dem Klassennamen, Datenfeldern (siehe Datenfeld) und Methoden (siehe Methode).

Klasse, abgeleitete
: siehe Unterklasse

Klassenmethode
: Eine **Klassenmethode** kann nur auf Klassenvariablen arbeiten, nicht jedoch auf Instanzvariablen, es sei denn, man übergibt eine Referenz auf ein Objekt an die Klassenmethode.

Klassenvariable
: Eine **Klassenvariable** stellt eine Variable dar, die **allen Exemplaren** (**Objekten**) **einer Klasse gemeinsam** ist und die in der Klasse selbst und nicht bei einem Objekt gespeichert wird. Eine Klassenvariable ist eine Variable, auf die alle Objekte dieser Klasse zugreifen können.

Komposition
: Bei einer **Komposition** besitzt ein "Groß"-Objekt als Datenfeld ein "Klein"-Objekt. Da das Datenfeld eines Objektes untrennbar mit dem Objekt selbst verknüpft ist, ist die Lebensdauer des "Klein"-Objekts gleich derjenigen des "Groß"-Objektes. Das "Groß"-Objekt muss den Life-Cycle des "Klein"-Objekts managen.

Konstante
: **Konstanten** sind Werte, die während des Programmablaufs nicht geändert werden können. Es gibt literale Konstanten, die einfach angeschrieben werden wie z. B. die Zahl 46. Es gibt auch symbolische Konstanten, die unter einem Namen angesprochen werden, denen aber während eines Programmlaufs ein fester Wert zugeordnet ist. So kann man z. B. eine Konstante `PI` einführen. Muss eventuell der Wert einer Konstanten wie

z. B. die Genauigkeit von `PI` abgeändert werden, so hat man den Vorteil, dass man nur zentral an einer einzigen Stelle, nämlich an der Stelle der Definition der Konstanten, eine Änderung durchführen muss. Die Änderung gilt dann an all den Stellen des Programms, an denen die Konstante mit Namen, d. h. als Symbol, steht.

Konstruktor — Ein **Konstruktor** ist eine spezielle Methode einer Klasse, die dazu dient, um ein **Objekt zu initialisieren**.

Konstruktor mit Parameter — Ein **Konstruktor mit Parametern** erlaubt es, **Werte zur Initialisierung** eines Objekts als aktuelle Parameter an den Konstruktor zu **übergeben**.

Kontrollfluss — Die Reihenfolge der Abarbeitung der Anweisungen eines Programms wird als der **Kontrollfluss** des Programms bezeichnet.

Kontrollklasse — Objekte von **Kontrollklassen** kapseln komplexe Verarbeitungen (Prozeduren) in Objekten und erhalten ihre Daten von Entity-Objekten. Ein Kontroll-Objekt findet man nicht durch "Peilen" in die Realität, sondern durch das Analysieren der Abläufe der Anwendungsfälle. Objekte von Kontrollklassen haben kein Gegenstück im Problembereich.

Kontrollstruktur — Konstrukte einer Programmiersprache, die den Kontrollfluss – d. h. die Abarbeitungsreihenfolge der Anweisungen – steuern, werden **Kontrollstrukturen** genannt.

Laufzeitsystem — siehe Laufzeitumgebung

Laufzeitumgebung — Die **Laufzeitumgebung** (auch Laufzeitsystem genannt) für das auszuführende Programm wird von der Java-Virtuellen Maschine gebildet. Die Laufzeitumgebung stellt einem Programm zusätzliche Funktionen (Routinen) zur Verfügung, welche für die Ausführung des Programms benötigt werden. Dazu gehören unter anderem Funktionen zur Speicherverwaltung oder Fehlererkennung.

Lebensdauer — Die **Lebensdauer** ist die Zeitspanne, in der die Java-Virtuelle Maschine einer Variablen einen **Platz im Speicher** zur Verfügung stellt. Mit anderen Worten, während ihrer Lebensdauer besitzt eine Variable einen Speicherplatz.

Liskovsches Substitutionsprinzip — Will man Polymorphie von Objekten für ein Programm haben, so muss im Programm eine Referenz auf eine Instanz einer abgeleiteten Klasse an die Stelle einer Referenz auf eine Instanz einer Basisklasse treten können. Dies fordert das **Liskovsche Substitutionsprinzip**.

Lösungsbereich — Die **Welt der technischen Lösung**, bei Programmiersprachen der Bereich der Programmkonstruktion.

L-Wert — Ein **L-Wert** ist eine **Variable**. Eine Variable hat eine Adresse im Arbeitsspeicher und kann Werte aufnehmen.

Maschine, virtuelle — Eine **virtuelle Maschine verbirgt** eine **spezielle Rechner-Plattform**, d. h. die Rechner-Hardware und das Betriebssystem, vor einem Programm. Damit wird erreicht, dass Programme plattformunabhängig – mit anderen Worten portabel – werden. Natürlich braucht man für jede Rechner-Plattform die entsprechende virtuelle Maschine.

Maschinencode — **Maschinencode** besteht aus Anweisungen, die ein spezieller Prozessor direkt versteht.

Mehrfachvererbung — Unterstützt eine Programmiersprache das Sprachmittel der **Mehrfachvererbung**, so kann eine **Klasse von mehreren Klassen abgeleitet** werden und erbt dabei Datenfelder und Methoden aus mehreren Klassen.

Methode — Eine Methode stellt eine Funktion dar, die für ein Objekt im Falle einer **Instanzmethode** (siehe Instanzmethode) bzw. für eine Klasse im Falle einer **Klassenmethode** (siehe Klassenmethode) aufgerufen werden kann. Methoden dienen dazu, um Werte an das Objekt (die Klasse) zu übergeben, um mit den im Objekt (in der Klasse) gespeicherten Daten Berechnungen durchzuführen und um Werte vom Objekt (der Klasse) abzuholen.

Modifikator — Ein **Modifikator** dient zur verfeinerten Festlegung der Eigenschaften von Datenfeldern, Methoden, Konstruktoren, Klassen und Schnittstellen.

Nachbedingung — Eine **Nachbedingung** stellt die Bedingung dar, die von einer aufgerufenen Methode eingehalten werden muss.

Nebeneffekt — Ein **Nebeneffekt** liegt vor, wenn ein Operator oder eine Methode nicht nur den Wert eines Ausdrucks beeinflusst, sondern **nebenbei** auch noch einen Wert im Speicher, in der Regel den Wert einer **Variablen, abändert**.

Oberklasse — siehe Superklasse

Object — Die Klasse `Object` ist die Wurzelklasse aller Java-Klassen. Eine jede Klasse in Java stellt einen Untertyp der Klasse `Object` dar.

Objekt — Ein **Objekt** ist eine Variable, die nach dem Datentyp einer Klasse gebaut ist. Andere Begriffe für Objekt sind **Instanz** und **Exemplar**.

Operand — **Operanden** werden durch Operatoren manipuliert. So verknüpft der Operator `+` die beiden Operanden `a` und `b` im Ausdruck `a + b` zur Summe.

Operation — Eine **Operation** stellt in abstrakter Weise eine **Verarbeitungsvorschrift** dar. Eine Operation wird als Methode in einer Klasse implementiert. Eine Operation kann in verschiedenen Klassen verschiedene Implementierungen erhalten, dabei wird jedoch stets die abstrakte Verarbeitungsvorschrift der Operation eingehalten.

Operator

Ein **Operator** ist eine **Rechenvorschrift in Symbolform** wie z. B. der Vorzeichenoperator − (Ändere das Vorzeichen) oder der Additionsoperator + (Bilde die Summe).

Paket

Pakete dienen zur Gruppierung von inhaltlich zusammengehörigen Klassen und Schnittstellen. Ein Paket umfasst einen eigenen Namensraum. Ein Paket stellt eine Bibliothek für einen bestimmten Zweck dar und kann als Einheit für den Zugriffsschutz verwendet werden.

Polymorphie

Polymorphie bedeutet Vielgestaltigkeit. So kann beispielsweise ein Objekt eines Subtyps auch in Gestalt der Basisklasse auftreten.

Polymorphie von Objekten

Eine **Polymorphie von Objekten** gibt es nur bei **Vererbungshierarchien**. **Polymorphie** (**Vielgestaltigkeit**) **von Objekten** bedeutet, dass ein Objekt in einer Vererbungshierarchie von verschiedenem Typ sein kann. Es kann vom Typ einer Basisklasse sein, aber auch vom Typ einer Unterklasse. Die Kundenklassen eines **polymorphen Programms** dürfen es hierbei gar nicht merken, dass plötzlich Objekte einer abgeleiteten Klasse an die Stelle von Objekten der Basisklasse treten. Polymorphie liegt vor, wenn das Liskovsche Substitutionsprinzip eingehalten wird.

Polymorphie von Operationen

Bei der **Polymorphie von Operationen** stellt jede nicht verwandte Klasse einen eigenen Namensraum dar. Verschiedene Klassen können deshalb denselben Methodenkopf in verschiedenen Implementierungen realisieren.

Problembereich

Der **Problembereich** oder **Problem Domain** ist die Welt der **Anwendung**. Er ist derjenige Ausschnitt aus der realen Welt, der später durch die zu realisierende Software abgedeckt werden soll.

Quellcode

Ein in einer Programmiersprache geschriebenes Programm oder Programmstück ist ein **Quellcode**.

Referenztyp

Variablen eines **Referenztyps** stellen **Referenzen** dar. Referenzen sind eine Art von Zeigern. Dies bedeutet, dass die gesuchte Information an einer anderen Stelle steht, nämlich an der Stelle, auf die die Referenz verweist.

Regulärer Ausdruck

Ein **regulärer Ausdruck** ist eine aus einer Menge von Zeichen nach syntaktischen Regeln erstellte Zeichenkette, die dazu dient, beliebige Zeichenketten daraufhin zu überprüfen, ob diese den regulären Ausdruck enthalten.

Rekursion

Ein rekursiver Algorithmus enthält im Falle einer **direkten Rekursion** Abschnitte, die sich selbst direkt aufrufen. Im Falle einer **indirekten Rekursion** rufen sich Abschnitte wechselseitig auf.

R-Wert	Ein **R-Wert** ist ein Wert, dem **keine Adresse** im Speicher zugeordnet ist und der nur temporär existiert.
Selektion	Eine **Selektion** ist eine Fallunterscheidung.
Sichtbarkeit	Die **Sichtbarkeit** einer Variablen bedeutet, dass man von einer Programmstelle aus die Variable sieht, das heißt, dass man auf sie über ihren Namen zugreifen kann.
Sohnklasse	Eine **Sohnklasse** ist eine abgeleitete Klasse, die direkt von einer Vaterklasse abgeleitet ist. Sohnklasse und Vaterklasse stehen in direkt benachbarten Hierarchieebenen.
Standardtyp	Eine Programmiersprache stellt selbst standardmäßig einige Datentypen als **Standardtypen** bereit, die ein Programmierer in der vorgesehenen Bedeutung ohne weitere Maßnahmen verwenden kann. Ein Beispiel hierfür ist in Java der Datentyp `int`.
Startklasse	Eine **Startklasse** ist eine Klasse, die eine Methode `main()` enthält, und damit von der Java-Virtuellen Maschine gestartet werden kann.
String	Ein **String** ist eine **Zeichenkette**, d. h. eine bestimmte Folge von zusammengehörigen Zeichen.
Subklasse	Eine Klasse, die von einer anderen Klasse abgeleitet ist, ist eine **Subklasse** dieser Klasse. Die Subklasse wird auch abgeleitete Klasse oder Unterklasse genannt.
Superklasse	Eine **Superklasse** ist eine Klasse, von der eine andere Klasse abgeleitet ist. Eine Superklasse wird auch Basisklasse oder Oberklasse genannt.
this-Referenz	Eine **this-Referenz** ist eine Referenz (ein Zeiger) auf das aktuell betrachtete Objekt.
Typkonvertierung	siehe Casten
Überschreiben	Definiert eine Unterklasse eine Methode mit demselben Namen, einem kompatiblen Rückgabetypen und einer kompatiblen Parameterliste wie eine Superklasse, so ist in der Unterklasse direkt über den Namen nur die Methode der Unterklasse sichtbar. Dies wird als **Überschreiben** bezeichnet. Physisch findet jedoch kein Überschreiben statt. Wenn auch nicht direkt über den Namen, so kann dennoch über andere Mechanismen auf die überschriebene Methode zugegriffen werden.
Unterklasse	siehe Subklasse
Variable	Eine **Variable** in einer Programmiersprache ist eine benannte Speicherstelle im Arbeitsspeicher des Rechners. Über den **Variablennamen** kann der Programmierer auf die entsprechende Speicherstelle zugreifen. Eine Variable hat vier Kennzeichen: Variablennamen, Datentyp, Wert und Adresse. In Java ist die Adresse einer Variablen dem Programmierer verborgen.

Variable, lokale — Eine **lokale Variable** lebt nur innerhalb einer Methode oder innerhalb eines Blocks. Dies bedeutet, dass nur während der Abarbeitung der Methode bzw. des Blocks für sie Platz im Arbeitsspeicher bereitgestellt wird.

Vaterklasse — Eine **Vaterklasse** ist eine Klasse, von der eine Sohnklasse abgeleitet ist. Sohnklasse und Vaterklasse stehen in direkt benachbarten Hierarchieebenen.

Verdecken — Gibt es in einem Programmstück mehrere gültige Variablen, die den gleichen Namen tragen, wie z. B. ein Datenfeld und eine lokale Variable, so ist über den Namen nur die lokale Variable, erreichbar. Auf die andere kann direkt über den Namen nicht zugegriffen werden, mit anderen Worten, sie ist **verdeckt**.

Vereinbarung, Variable — Eine **Vereinbarung einer Variablen** ist ein Oberbegriff für Deklaration und Definition von Variablen.

Vererbung — Eine Unterklasse **erbt** alle Datenfelder und Methoden ihrer Superklasse, in anderen Worten ihre Struktur und ihr Verhalten. Diese müssen bei der Definition der Unterklasse nicht erneut angeschrieben werden.

Vertrag — Der **Vertrag einer Klasse** umfasst die **Verträge der Methoden**, d. h. ihre Vor- und Nachbedingungen, sowie die **Invarianten der Klasse**. Verschiedenen Kunden einer Klasse können verschiedene Verträge zur Verfügung gestellt werden.

Vorbedingung — Eine **Vorbedingung** (engl. **precondition**) stellt die Einschränkungen dar, unter denen eine Methode korrekt aufgerufen wird.

Wrapper-Klasse — Eine **Wrapper-Klasse** wird gebraucht, wenn man ein **nicht-objektorientiertes Verhalten in eine objektorientierte Form bringen** muss. Bespiele hierfür sind das Wrappen (Einpacken) einer Prozedur in Form einer `main()`-Methode in eine eigene Klasse oder das Einpacken einer Variablen eines einfachen Datentyps in die Gestalt eines Objektes, das in seinem Innern diese einfache Variable enthält.

Zustand, makroskopisch — Ein **Zustand** eines Objektes, der **für eine Anwendung eine Bedeutung** hat. So kann ein Objekt der Klasse Fahrstuhl sich z. B. im Zustand "Fahren" oder "Türen öffnen" befinden.

Zustand, mikroskopisch — Jede Kombination der Werte der Datenfelder eines Objekts stellt einen Zustand dar, der als **mikroskopischer Zustand** eines Objektes bezeichnet wird.

Abkürzungsverzeichnis

AES	Advanced Encryption Standard
API	Application programming interface
ASCII	American Standard Code for Information Interchange
BCPL	Basic Combined Programming Language
CRC	Class-Responsibility-Collaboration
DIN	Deutsches Institut für Normung
EDV	Elektronische Datenverarbeitung
GUI	Graphical User Interface
HTML	Hypertext Markup Language
Java EE	Java Enterprise Edition
Java ME	Java Micro Edition
Java SE	Java Standard Edition
JDBC	Java Database Connectivity
JDK	Java Development Kit
JIT	Just-in-time
JNDI	Java Naming and Directory Interface
JRE	Java Runtime Environment
JSF	JavaServer Faces
JSP	JavaServer Pages
MMI	Mensch-Maschine-Schnittstelle
OSF	Open Software Foundation
RMI	Remote Method Invocation
sh	Bourne-Shell
SHA	Secure Hash Algorithm
UML	Unified Modeling Language
UTF	Unicode Transformation Format
VM	Virtuelle Maschine

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1

Anhang A Der ASCII-Zeichensatz

Der ASCII-Zeichensatz ist die US-nationale Ausprägung des ISO-7-Bit-Codes (ISO 646). Eine weitere nationale Ausprägung des ISO-7-Bit-Codes ist der nach DIN 66003 spezifizierte deutsche Zeichensatz, bei dem die Zeichen `§`, `Ä`, `Ö`, `Ü`, `ä`, `ö`, `ü` und `ß` die Zeichen `@`, `[`, `\`, `]`, `{`, `|`, `}` und `~` des ASCII-Zeichensatzes ersetzen.

Bei manchen Rechnern wie z. B. beim IBM-PC wird aber ein erweiterter ASCII-Zeichensatz eingesetzt, bei dem alle 8 Bits verwendet werden. Die ersten 128 Zeichen stimmen dabei mit dem 7-Bit ASCII-Code in Tabelle A-1 überein. Die Sonderzeichen `Ä`, `Ö`, `Ü`, `ä`, `ö`, `ü` und `ß` befinden sich dabei im Bereich 128-255.

Wie aus Tabelle A-1 und Tabelle A-3 ersichtlich ist, enthält der ASCII-Zeichensatz Buchstaben, Ziffern, Sonderzeichen und Steuerzeichen. Da jedem Zeichen im Rahmen des jeweiligen Codes eine 7- bzw. 8-stellige Binärzahl eindeutig zugeordnet ist und die Binärzahlen selbst eine geordnete Menge darstellen, bilden damit in jedem dieser Codes die Zeichen eine geordnete Menge. Es gibt für die Zeichen also ein vorher (<) und nachher (>), sodass die Zeichen dem Code entsprechend alphabetisch sortiert werden können.

Dez.	Hex.	Ctrl-Ch.	Char.	Dez.	Hex.	Char.	Dez.	Hex.	Char.	Dez.	Hex.	Char.
0	00	^@		32	20		64	40	@	96	60	'
1	01	^A	☺	33	21	!	65	41	A	97	61	a
2	02	^B	☻	34	22	"	66	42	B	98	62	b
3	03	^C	♥	35	23	#	67	43	C	99	63	c
4	04	^D	♦	36	24	$	68	44	D	100	64	d
5	05	^E	♣	37	25	%	69	45	E	101	65	e
6	06	^F	♠	38	26	&	70	46	F	102	66	f
7	07	^G	●	39	27	'	71	47	G	103	67	g
8	08	^H	◘	40	28	(	72	48	H	104	68	h
9	09	^I	○	41	29	)	73	49	I	105	69	i
10	0A	^J	◙	42	2A	*	74	4A	J	106	6A	j
11	0B	^K	♂	43	2B	+	75	4B	K	107	6B	k
12	0C	^L	♀	44	2C	,	76	4C	L	108	6C	l
13	0D	^M	♪	45	2D	-	77	4D	M	109	6D	m
14	0E	^N	♫	46	2E	.	78	4E	N	110	6E	n
15	0F	^O	☼	47	2F	/	79	4F	O	111	6F	o
16	10	^P	►	48	30	0	80	50	P	112	70	p
17	11	^Q	◄	49	31	1	81	51	Q	113	71	q
18	12	^R	↕	50	32	2	82	52	R	114	72	r
19	13	^S	‼	51	33	3	83	53	S	115	73	s
20	14	^T	¶	52	34	4	84	54	T	116	74	t
21	15	^U	§	53	35	5	85	55	U	117	75	u
22	16	^V	▬	54	36	6	86	56	V	118	76	v
23	17	^W	↨	55	37	7	87	57	W	119	77	w
24	18	^X	↑	56	38	8	88	58	X	120	78	x
25	19	^Y	↓	57	39	9	89	59	Y	121	79	y
26	1A	^Z	→	58	3A	:	90	5A	Z	122	7A	z
27	1B	^[	←	59	3B	;	91	5B	[	123	7B	{
28	1C	^\	∟	60	3C	<	92	5C	\	124	7C	|
29	1D	^]	↔	61	3D	=	93	5D	]	125	7D	}
30	1E	^^	▲	62	3E	>	94	5E	^	126	7E	~
31	1F	^_	▼	63	3F	?	95	5F	_	127	7F	⌂

Tabelle A-1 Der ASCII-Zeichensatz mit 128 Zeichen

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1

In der Spalte Control-Character (Steuerzeichen), abgekürzt durch Ctrl-Ch., werden spezielle Tastenkombinationen angegeben, mit denen Steuerzeichen erzeugt werden können. Hierbei kann es je nach Betriebssystem auch geringfügige Modifikationen geben. Die ersten 32 ASCII-Zeichen stellen Steuerzeichen für die Ansteuerung von Peripheriegeräten und die Steuerung einer rechnergestützten Datenübertragung dar. Diese Zeichen tragen die Namen:

Dez.	Symbol	Dez.	Symbol	Dez.	Symbol	Dez.	Symbol	Dez.	Symbol	Dez.	Symbol
0	NUL	6	ACK	12	FF	18	DC2	24	CAN	30	RS
1	SOH	7	BEL	13	CR	19	DC3	25	EM	31	US
2	STX	8	BS	14	SO	20	DC4	26	SUB		
3	ETX	9	HAT	15	SI	21	NAK	27	ESC		
4	EOT	10	LF, NL	16	DLE	22	SYN	28	FS		
5	ENQ	11	VT	17	DC1	23	ETB	29	GS		

Tabelle A-2 Namen der 32 Steuerzeichen des ASCII-Zeichensatzes

So ist beispielsweise FF die Abkürzung für Form Feed, d. h. Seitenvorschub, oder CR die Abkürzung für Carriage Return, dem Wagenrücklauf, der schon von der Schreibmaschine her bekannt ist.

Dez	Hex	Char	Dez	Hex	Char	Dez	Hex	Char	Dez	Hex	Char	Dez	Hex	Char	Dez	Hex	Char
0	00		43	2B	+	86	56	V	129	81	ü	172	AC	¼	215	D7	╫
1	01	☺	44	2C	,	87	57	W	130	82	é	173	AD	¡	216	D8	╪
2	02	☻	45	2D	-	88	58	X	131	83	â	174	AE	«	217	D9	┘
3	03	♥	46	2E	.	89	59	Y	132	84	ä	175	AF	»	218	DA	┌
4	04	♦	47	2F	/	90	5A	Z	133	85	à	176	B0	░	219	DB	█
5	05	♣	48	30	0	91	5B	[	134	86	å	177	B1	▒	220	DC	▄
6	06	♠	49	31	1	92	5C	\	135	87	ç	178	B2	▓	221	DD	▌
7	07	●	50	32	2	93	5D	]	136	88	ê	179	B3	│	222	DE	▐
8	08	◘	51	33	3	94	5E	^	137	89	ë	180	B4	┤	223	DF	▀
9	09	○	52	34	4	95	5F	_	138	8A	è	181	B5	╡	224	E0	a
10	0A	◙	53	35	5	96	60	'	139	8B	ï	182	B6	╢	225	E1	b
11	0B	♂	54	36	6	97	61	a	140	8C	î	183	B7	╖	226	E2	G
12	0C	♀	55	37	7	98	62	b	141	8D	ì	184	B8	╕	227	E3	p
13	0D	♪	56	38	8	99	63	c	142	8E	Ä	185	B9	╣	228	E4	S
14	0E	♫	57	39	9	100	64	d	143	8F	Å	186	BA	║	229	E5	s
15	0F	☼	58	3A	:	101	65	e	144	90	É	187	BB	╗	230	E6	m
16	10	►	59	3B	;	102	66	f	145	91	æ	188	BC	╝	231	E7	t
17	11	◄	60	3C	<	103	67	g	146	92	Æ	189	BD	╜	232	E8	f
18	12	↕	61	3D	=	104	68	h	147	93	ô	190	BE	╛	233	E9	q
19	13	‼	62	3E	>	105	69	i	148	94	ö	191	BF	┐	234	EA	W
20	14	¶	63	3F	?	106	6A	j	149	95	ò	192	C0	└	235	EB	d
21	15	§	64	40	@	107	6B	k	150	96	û	193	C1	┴	236	EC	µ
22	16	▬	65	41	A	108	6C	l	151	97	ù	194	C2	┬	237	ED	Æ
23	17	↨	66	42	B	109	6D	m	152	98	ÿ	195	C3	├	238	EE	e
24	18	↑	67	43	C	110	6E	n	153	99	Ö	196	C4	─	239	EF	Ç
25	19	↓	68	44	D	111	6F	o	154	9A	Ü	197	C5	┼	240	F0	º
26	1A	→	69	45	E	112	70	p	155	9B	¢	198	C6	╞	241	F1	±
27	1B	←	70	46	F	113	71	q	156	9C	£	199	C7	╟	242	F2	³
28	1C	∟	71	47	G	114	72	r	157	9D	¥	200	C8	╚	243	F3	£
29	1D	↔	72	48	H	115	73	s	158	9E	₧	201	C9	╔	244	F4	ì
30	1E	▲	73	49	I	116	74	t	159	9F	ƒ	202	CA	╩	245	F5	þ
31	1F	▼	74	4A	J	117	75	u	160	A0	á	203	CB	╦	246	F6	÷
32	20		75	4B	K	118	76	v	161	A1	í	204	CC	╠	247	F7	»
33	21	!	76	4C	L	119	77	w	162	A2	ó	205	CD	═	248	F8	°
34	22	"	77	4D	M	120	78	x	163	A3	ú	206	CE	╬	249	F9	·
35	23	#	78	4E	N	121	79	y	164	A4	ñ	207	CF	╧	250	FA	×
36	24	$	79	4F	O	122	7A	z	165	A5	Ñ	208	D0	╨	251	FB	Ö
37	25	%	80	50	P	123	7B	{	166	A6	ª	209	D1	╤	252	FC	ⁿ
38	26	&	81	51	Q	124	7C	|	167	A7	º	210	D2	╥	253	FD	²
39	27	'	82	52	R	125	7D	}	168	A8	¿	211	D3	╙	254	FE	■
40	28	(	83	53	S	126	7E	~	169	A9	⌐	212	D4	╘	255	FF	
41	29	)	84	54	T	127	7F	⌂	170	AA		213	D5	╒			
42	2A	*	85	55	U	128	80	Ç	171	AB	½	214	D6	╓			

Tabelle A-3 Der erweiterte ASCII-Zeichensatz (256 Zeichen)

Anhang B Gültigkeitsbereiche von Namen

Unter dem **Gültigkeitsbereich eines einfachen Namens** versteht man den **Bereich im Programm**, innerhalb dessen die **Deklaration des Namens bekannt** ist.

Qualifizierte Namen dienen zum Zugriff auf die Komponenten eines **Pakets** und auf die Methoden und Datenfelder von **Referenztypen** wie einer Klasse oder Schnittstelle.

Soll beispielsweise in einer Klasse eine Klasse eines anderen Pakets benutzt werden, so erfolgt der Zugriff mit Hilfe des **voll qualifizierten Namens**. Ein **voll qualifizierter** Name bezeichnet den Namen einer Klasse, der die Klasse durch Angabe der Paketstruktur gefolgt von einem Punkt und dem eigentlichen Klassennamen identifiziert wie z. B. `java.lang.String`. Hierbei stellt `String` den eigentlichen Klassennamen, einen einfachen Klassennnamen, dar und `java.lang` ist die Paketstruktur.

Ähnliche Zugriffsformen sind die Zugriffe auf Datenfelder und Methoden von Objekten über Referenzen. Diese Zugriffe erfolgen auch mit Hilfe des Punkt-Operators.

Bei den im Folgenden erörterten Gültigkeitsbereichen geht es nicht um Zugriffe von außen, sondern um die Gültigkeit von Namen, die bei ihrer **Deklaration**, d. h. wenn sie dem Compiler bekannt gegeben werden, stets **einfache Namen** sind.

Deklarationen können in Teilen ihres Gültigkeitsbereichs **verdeckt werden** durch andere Deklarationen mit demselben Namen. Wird der entsprechende Name durch eine weitere Einheit mit demselben Namen verdeckt, so ist der Name immer noch gültig, aber nicht sichtbar, da bei seiner Verwendung der Zugriff auf die verdeckende Einheit erfolgt. Ein Beispiel für das Verdecken ist das Verdecken eines Datenfeldes mit dem Namen `x` durch eine lokale Variable mit dem Namen `x`, oder das Verdecken einer Marke in einem äußeren Block durch eine Marke mit demselben Namen in einem inneren Block.

Wird ein Name nicht verdeckt, so kann innerhalb des Gültigkeitsbereichs über den Namen auf die entsprechende Einheit Bezug genommen werden.

Nicht immer führt ein gleicher Name dazu, dass ein anderer Name verdeckt wird. Durch die Verwendung von Kontexten kann Java Namenskonflikte minimieren. So dürfen beispielsweise Typen, Methoden und Datenfelder in Java denselben Namen tragen. Aus der Verwendung wird dabei klar, um was es sich jeweils handelt.

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1

Man unterscheidet zwischen den Gültigkeitsbereichen:

- von Paketen,
- von importierten Klassen und Schnittstellen,
- eines Klassen- oder Schnittstellennamens,
- von Datenfeldern und Methoden innerhalb einer Klasse oder einer Schnittstelle,
- von formalen Parametern einer Methode oder eines Konstruktors,
- von lokalen Variablen innerhalb eines Blocks,
- von lokalen Variablen, die in der Initialisierungsklausel einer `for`-Schleife definiert werden,
- von lokalen Variablen, die in einer try-with-resources-Anweisung definiert werden,
- und eines Exception-Parameters in einem `catch`-Block.

B 1 Pakete

Pakete entsprechen Ordnern im Dateisystem eines Speichermediums. Auf welche Pakete auf der obersten Ebene zugegriffen werden kann, d. h. welche Paketnamen auf der obersten Ebene gültig sind, wird durch Einstellungen auf der Betriebssystem-Ebene festgelegt.

B 2 Importierte Klassen und Schnittstellen

Wird ein Klassenname oder ein Schnittstellenname in einer Übersetzungseinheit (Datei) importiert, so ist er ab der `import`-Vereinbarung in der ganzen Übersetzungseinheit sichtbar. Dies gilt für einen vollständig qualifizierten Namen wie z. B.:

```
import java.awt.Frame;
```

ebenso wie für die Verwendung von Wildcards

```
import java.awt.*;
```

Werden Wildcards verwendet, so werden alle entsprechenden `public`-Typen des genannten Pakets importiert, sofern sie benötigt werden. Die Gültigkeit der importierten Namen erstreckt sich nur auf die Datei, in der die Namen importiert werden, nicht auf das gesamte Paket.

B 3 Klassen und Schnittstellennamen

Der Gültigkeitsbereich einer in einem Paket definierten Klasse oder Schnittstelle bezieht sich auf alle Übersetzungseinheiten (Dateien) des Pakets. Eine Vorwärtsdeklaration ist nicht erforderlich.

B 4 Datenfelder und Methoden innerhalb einer Klasse oder einer Schnittstelle

Wenn ein Attribut, d. h. ein Datenfeld oder eine Methode, in einer Klasse oder Schnittstelle definiert wird oder von einer Klasse bzw. Schnittstelle geerbt wird, so ist das Attribut – unabhängig von der Reihenfolge der Attribute – in der gesamten Defi-

nition der Klasse bzw. Schnittstelle gültig, es sei denn, es wird ein anderes Datenfeld zur Initialisierung eines Datenfeldes benutzt. Ist dies der Fall, dann muss das zur Initialisierung verwendete andere Datenfeld bereits selbst definiert sein, wenn es zur Initialisierung eines Datenfeldes herangezogen wird. Deshalb wird beim folgenden **Beispiel** ein Fehler erzeugt:

```
// Datei: Inittest.java
class Inittest
{
   int alpha = beta; //nicht zulässig
   int beta  = 3;
}
```

Die Ausgabe des Programms ist:

```
Inittest.java:5: illegal forward reference
   int alpha = beta; //nicht zulässig
```

Zulässig ist aber:

```
// Datei: Inittest2.java
class Inittest2
{
   Inittest2()
   {
      alpha = 3;
   }

   int beta = 1;
   int alpha;
}
```

Ebenso ist zulässig:

```
// Datei: Inittest3.java
class Inittest3
{
   int alpha = beta;
   static int beta = 3;
}
```

da eine Klassenvariable bereits beim Laden der Klasse initialisiert wird.

B 5 Formale Parameter bei Methoden und Konstruktoren

Der Gültigkeitsbereich des Namens eines formalen Parameters ist der ganze Methodenrumpf. Es ist nicht zulässig, den Namen eines formalen Parameters zu verdecken, d. h. der Name eines formalen Parameters kann nicht für die Definition einer lokalen Variablen oder eines Exception-Parameters innerhalb der Methode verwendet werden. Auf formale Parameter kann nur unter Verwendung ihres einfachen Namens, nicht aber mit einem qualifizierten Namen Bezug genommen werden. Formale Parameter eines Konstruktors werden wie formale Parameter von Methoden behandelt.

B 6 Lokale Variablen innerhalb eines Blocks

Der Gültigkeitsbereich einer lokalen Variablen ist der Rest des Blocks ab der Definition der Variablen. Lokale Variablen verlieren am Ende ihres Blockes ihre Gültigkeit.

B 7 Lokale Variablen in einer `for`-Anweisung

Der Gültigkeitsbereich einer lokalen Variablen, die im Initialisierungskonstrukt einer `for`-Schleife definiert wird, erstreckt sich über die Initialisierungsklausel `Initialisierungsklausel` mit der eigenen Definition und den Definitionen rechts davon, über den Ausdruck `BoolescherAusdruck`, die Aktualisierungs-Ausdrucksliste `Aktualisierungs-Ausdrucksliste` und schließlich die Anweisung `Anweisung`. An die Stelle der Anweisung `Anweisung` kann ein Block treten. Nach der Schleife verlieren die in der Initialisierungsklausel definierten Variablen ihre Gültigkeit.

Das folgende **Beispiel** demonstriert den Gültigkeitsbereich einer in der Initialisierungsklausel definierten lokalen Variablen `i`:

```
// Datei: Calc.java
class Calc
{
   public static void main (String[] args)
   {
      for (int n = 10; n <= 15; n++)
      {
         int summe = 0;
         for (int i = 1, j = n; i <= n / 2; i++, j--)
            summe = summe + i + j;
         // i ist nicht mehr gültig
         if ( n % 2 != 0)
            summe = summe + n / 2 + 1;
         System.out.println ("Die Summe von 1 bis " + n + " ist: "
                             + summe);
      }
           // n ist nicht mehr gültig
   }
}
```

Die Ausgabe des Programms ist:

```
Die Summe von 1 bis 10 ist: 55
Die Summe von 1 bis 11 ist: 66
Die Summe von 1 bis 12 ist: 78
Die Summe von 1 bis 13 ist: 91
Die Summe von 1 bis 14 ist: 105
Die Summe von 1 bis 15 ist: 120
```

Wird versucht, auf `i` bzw. `n` nach der jeweiligen Schleife zuzugreifen, so resultiert ein Kompilierfehler.

B 8 Lokale Variablen in einer try-with-resources-Anweisung

Der Gültigkeitsbereich einer lokalen Variablen, die in einer try-with-resources-Anweisung definiert wird, erstreckt sich über die try-with-resources-Anweisung mit der eigenen Definition und den Definitionen rechts davon sowie über den `try`-Block. Nach dem `try`-Block – also auch in darauf folgenden `catch`-Blöcken – verlieren die definierten Variablen ihre Gültigkeit.

Das folgende Beispiel demonstriert den Gültigkeitsbereich zweier in der try-with-resources-Anweisung definierten Variablen `fis` und `bis`:

```
// Datei: Except1.java
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;

class Except1
{
   Except1()
   {
     try (FileInputStream fis = new FileInputStream ("c:\\datei.txt");
            BufferedInputStream bis = new BufferedInputStream (fis))
     {
        bis.read();
     }
     catch (IOException e)
     {
        // fis und bis sind hier nicht mehr gueltig
        System.err.println (e.getMessage());
     }
     // fis und bis sind hier nicht mehr gueltig
   }
}
```

B 9 Parameter eines Exception Handlers

Ein `catch`-Block besitzt genau einen Parameter, den Exception-Parameter, als formalen Parameter in der Liste der Übergabeparameter:

```
catch (Exceptiontyp name)
{                            //
   . . . . .                 // catch-Block
}                            //
```

Dies trifft auch bei dem Fangen von mehreren Exceptions in einem Block (siehe Kapitel 13) zu:

```
catch (Exceptiontyp1 | Exceptiontyp2 name)
{                            //
   . . . . .                 // catch-Block
}                            //
```

Der Gültigkeitsbereich des Exception-Parameters erstreckt sich auf den `catch`-Block. Der Name des Exception-Parameters darf nicht verdeckt werden, d. h. es darf kein lokaler Parameter oder keine Exception mit demselben Namen im `catch`-Block

definiert werden. Nach dem `catch`-Block verliert der Exception-Parameter seine Gültigkeit. Wird er nach dem `catch`-Block verwendet, so resultiert ein Kompilierfehler. Dies zeigt das folgende **Beispiel**:

```
// Datei: Except2.java

class Except2
{
   Except2()
   {
      try
      {
      }
      catch (ArrayIndexOutOfBoundsException e)
      {
         e.getMessage();
      }
      e.getMessage();
   }
}
```

Die Ausgabe des Programms ist:

```
Except.java:14: cannot find symbol
symbol  : variable e
location: class Except
      e.getMessage();
```

Auf Exception-Parameter kann nur über einfache Namen, nicht über qualifizierte Namen Bezug genommen werden.

Anhang C Die Klasse `System`

Die Klasse `System` stellt eine Reihe systemnaher Methoden und Variablen zur Verfügung. Da alle Methoden und Variablen dieser Klasse `static` sind, ist es nicht notwendig, eine Instanz dieser Klasse zu erzeugen. Dies geht auch gar nicht, da alle Konstruktoren der Klasse `System` als `private` deklariert sind. Die Klasse `System` befindet sich im Paket `java.lang`. Im Folgenden werden die wichtigsten Methoden der Klasse `System` aufgezeigt.

C 1 Die Klassenvariablen `in`, `out` und `err`

Die wohl am häufigsten benötigten Elemente der Klasse `System` sind mit Sicherheit die beiden Klassenvariablen `in` und `out`. Sie dienen zur Ein- bzw. Ausgabe von Zeichen und Zeichenketten. So wurde in den Buchkapiteln schon häufig der Aufruf `System.out.println()` verwendet, um Text auf dem Bildschirm auszugeben. Analog dazu wurden oftmals Texteingaben des Benutzers mit Hilfe von `System.in.readLine()` über die Tastatur eingelesen. Ein Aufruf sah z. B. so aus:

```
System.out.println ("Geben Sie Ihren Namen ein: ");
String name = System.in.readLine();
```

Neben `in` und `out` beinhaltet die Klasse `System` mit `err` eine weitere Klassenvariable zur Textausgabe. Diese Variable funktioniert analog zu `out`, dient aber dazu, Fehlermeldungen auszugeben. Dadurch ist eine Trennung zwischen normaler Textausgabe und erweiterten Meldungen wie z. B. Fehlermeldungen möglich. Wird die Standardausgabe `out` in eine Datei umgeleitet, so können dennoch Fehlermeldungen über die Variable `err` auf den Bildschirm ausgegeben werden.

Hinter den drei Variablen `in`, `out` und `err` stehen Objekte verschiedener Stream-Klassen. Die beiden Klassenvariablen `out` und `err` sind vom Typ `PrintStream`. `PrintStream` ist eine Klasse, die zum Ausgeben von Text auf der Konsole verwendet wird. Die Klassenvariable `in` ist vom Typ `InputStream` und wird zum Einlesen von Zeichen verwendet.

Die Klasse `System` ermöglicht es dem Programmierer, den Datenstrom auf eigene Stream-Objekte umzulenken. Beispielsweise können Textausgaben in eine Datei umgeleitet werden oder die Daten für eine Texteingabe aus einer Datei gelesen werden. Hierzu werden die folgende Methoden zur Verfügung gestellt:

- `static void setIn (InputStream in)`
- `static void setOut (PrintStream out)`
- `static void setErr (PrintStream err)`

C 2 Properties

Properties bieten Informationen über das Betriebssystem. Eine Property ist ein Wertepaar bestehend aus einem Schlüssel und dem dazugehörenden Wert. Diese Daten werden jeweils in einem String gehalten. Die Klasse `System` beinhaltet eine

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1

Instanz der Klasse `java.utils.Properties`. Auf diese Instanz kann über die Klassenmethode `getProperties()` zugegriffen werden. Über die Klassenmethode `setProperties()` kann diese Instanz gesetzt werden.

Das direkte Arbeiten mit der Klasse `java.utils.Properties` ist in der Regel nicht notwendig, da die Klasse `System` die folgenden Methoden zur Verfügung stellt:

- `static String setProperty (String key, String value)`
- `static String getProperty (String key)`

So lässt sich mit der Methode `setProperty()` eine bestehende Property abändern oder eine neue Property anlegen. Dies sieht folgendermaßen aus:

```
System.setProperty ("MyProperty", "Meine eigene Property");
System.out.println (System.getProperty ("MyProperty"));
```

Die Ausgabe des Programms ist:

```
Meine eigene Property
```

Die Laufzeitumgebung stellt standardmäßig folgende Properties zur Verfügung:

Property	Bedeutung
`file.separator`	Trennzeichen für die Bestandteile eines Pfadnamens
`java.class.path`	Aktueller Klassenpfad
`java.class.version`	Versionsnummer der verwendeten Java-Klassenbibliothek
`java.home`	Installationsverzeichnis der Java-Laufzeitumgebung
`java.vendor`	Herstellerspezifische Zeichenkette der Java-Laufzeitumgebung
`java.vendor.url`	Internet-Link zum Hersteller der Java-Laufzeitumgebung
`java.version`	Versionsnummer der verwendeten Java-Laufzeitumgebung
`line.separator`	Zeilenende-Zeichen
`os.arch`	Betriebssystem-Architektur
`os.name`	Name des Betriebssystems
`os.version`	Versionsnummer des Betriebssystems
`path.separator`	Trennzeichen zwischen einzelnen Pfadnamen
`user.dir`	Aktuelles Arbeitsverzeichnis
`user.home`	Home-Verzeichnis des angemeldeten Benutzers
`user.name`	Name des angemeldeten Benutzers

Tabelle C-1 Properties

C 2.1 Beenden der Java-Laufzeitumgebung

Die Klasse `System` beinhaltet eine Klassenmethode zum sofortigen Beenden der Laufzeitumgebung. Diese Methode ist folgendermaßen definiert:

```
static void exit (int status);
```

Durch den Aufruf dieser Methode werden das Programm und die dazugehörige Laufzeitumgebung beendet. Der Parameter dient als Statuscode. Es ist eine Konvention, dass ein von 0 verschiedener Statuscode eine abnormale Beendigung anzeigt. Ein Ablauf im Fehlerfall könnte folgendermaßen aussehen:

```
if (isVerbindungVerfuegbar() == false)
{
   System.out.println ("Verbindung nicht verfügbar!");
   System.exit (5);
}
```

C 2.2 Weitere Methoden

Die Klasse System bietet weitere Methoden an, von denen zwei hier noch näher erläutert werden sollen. Die anderen Methoden werden der Vollständigkeit halber nur kurz erwähnt.

Manchmal ist es wünschenswert, den Garbage Collector direkt zu starten. Hierfür bietet die Klasse System eine Methode namens gc() an, mit der eine Speicherbereinigung durch die Garbage Collection angefordert werden kann.

Es ist allerdings zu beachten, dass dadurch der Garbage Collector nicht explizit aufgerufen wird. Der Methodenaufruf kann eine Speicherbereinigung durch den Garbage Collector nur erbitten und keinesfalls erzwingen.

Die Methode gc() ist wie folgt definiert:

```
static void gc();
```

Die zweite Methode, die hier vorgestellt werden soll, ermöglicht es dem Programmierer, die seit dem 1.1.1970 verstrichene Zeit in Millisekunden zu ermitteln. Diese Methode hat die folgende Syntax:

```
static long currentTimeMillis();
```

Es folgt ein kleines Beispiel, welches 5 Sekunden lange wartet:

```
long timeSnap = System.currentTimeMillis();
while (System.currentTimeMillis() < timeSnap + 5000);
```

Anhang D jar-Dateien

jar-Dateien sind Java-Archiv-Dateien. Sie dienen zur Zusammenstellung mehrerer Dateien – in gepackter oder ungepackter Form – in einer einzigen Datei. Archivdateien haben die Dateierweiterung jar. Sie dienen dazu, Java-Applikationen und Java-Bibliotheken Anwendern bzw. Entwicklern zur Verfügung zu stellen. jar-Dateien können außer Bytecode-Daten beispielsweise auch Bilder und xml-Dateien enthalten. Es können aber auch weitere jar-Dateien in einer jar-Datei verpackt sein. Typischerweise enthält eine jar-Datei Bytecode-Daten (.class-Dateien) und sogenannte Metadaten. Bei den Metadaten eines Archivs handelt es sich hierbei hauptsächlich um die Datei MANIFEST.MF, in der beispielsweise Informationen wie die Version der Manifest-Spezifikation oder Signatur-Informationen für Klassen, wenn eine jar-Datei signiert wird, hinterlegt werden. jar-Dateien haben das Format einer zip-Datei und können dadurch mit jedem herkömmlichen Archiv-Programm geöffnet werden.

D 1 Verwendung von jar-Dateien

Um die in den jar-Dateien enthaltenen Klassen zu verwenden, müssen die jar-Dateien dem Suchpfad – entweder über die Umgebungsvariable CLASSPATH oder über die Option classpath des Compilers javac oder des Interpreters java – mit ihrem vollständigen Pfad hinzugefügt werden. Sollen z. B. die jar-Dateien a.jar und b.jar im Verzeichnis C:\test zur Umgebungsvariablen CLASSPATH hinzugefügt werden, so muss das Kommando

```
set CLASSPATH=%CLASSPATH%;C:\test\a.jar;C:\test\b.jar;
```

innerhalb der Konsole ausgeführt werden. Es dürfen keine Wildcards wie C:\test*.jar verwendet werden. Es muss darauf geachtet werden, dass hierbei die einzelnen jar-Dateien durch ein Semikolon getrennt sind und die letzte jar-Datei ebenfalls mit einem Semikolon abschließt. Benötigen Klassen in einem Java-Archiv selbst Klassen aus anderen Java-Archiven, können diese anderen jar-Dateien über einen sogenannten ClassPath-Eintrag in der Manifest-Datei eingebunden werden. Dabei ist zu beachten, dass die jar-Dateien, anders als bei der Variable CLASSPATH, durch ein oder mehrere Leerzeichen getrennt werden müssen.

Eine Ausnahme bildet die rt.jar-Datei der Java Standard Edition. Sie enthält alle Klassen, die die Java Standard Edition zur Verfügung stellt. Die Datei rt.jar wird immer von der virtuellen Maschine geladen und muss nicht explizit dem Suchpfad hinzugefügt werden.

Damit die in den jar-Dateien befindlichen Klassen in einem Java-Programm verwendet werden können, müssen sie im Programmcode importiert werden. Dies geschieht, wie schon in Kapitel 12.4 erwähnt, über das Schlüsselwort import.

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1

D 2 Aufbau von Java-Archiven

Eine `jar`-Datei enthält Klassen mit ihrer Paketstruktur und immer einen Ordner `META-INF`, in dem Meta-Informationen abgelegt werden. Die wichtigste Datei im `META-INF`-Ordner ist die Manifest-Datei `META-INF/MANIFEST.MF`. In ihr kann man unter anderem:

- Den Programmeinstiegspunkt angeben (die Klasse mit der `main()`-Methode), um die `jar`-Datei ausführbar zu machen.
- Den Suchpfad (`CLASSPATH`) angeben, sodass die von den Klassen in der `jar`-Datei referenzierten Klassen aus anderen `jar`-Dateien gefunden werden können. Die referenzierten `jar`-Dateien werden als relativer Pfad angegeben und dürfen keine Wildcards enthalten.
- Klassen als Java-Beans[229] kennzeichnen.

Bei den Einträgen der Manifest-Datei dürfen zwischen dem Schlüsselwort und dem Doppelpunkt keine Leerzeichen stehen. Ebenfalls muss die Manifest-Datei immer mit einer leeren Zeile enden.

Für eine ausführbare `jar`-Datei `MainProgram.jar`, die z. B. die Klasse `Main` mit der Methode `main()` enthält und Abhängigkeiten zu Klassen aus den Bibliotheken `Dependency1.jar` und `Dependency2.jar` in einem Unterverzeichnis `lib` besitzen soll, könnte die Manifest-Datei wie folgt aussehen:

```
Manifest-Version: 1.0
Class-Path: lib/Dependency1.jar lib/Dependency2.jar
Main-Class: Main
<leere Zeile>
```

Da sich `jar`-Dateien mit einem ZIP-Archivierungsprogramm problemlos öffnen lassen, sind weitere Beispiele leicht zu finden.

D 3 Erstellung von `jar`-Dateien

Zur Erstellung von `jar`-Dateien stellt das JDK das Werkzeug `jar` zur Verfügung. Wer schon einmal unter Linux mit dem Programm `tar` gearbeitet hat, wird sich schnell zurechtfinden.

Um Dateien zu einem Java-Archiv hinzuzufügen, wäre die folgende Eingabe möglich:

```
jar -cfm MeineJarDatei.jar Manifest.txt Datei1.class ...
```

Beim Erstellen eines Java-Archivs können beliebig viele Dateien angegeben werden. Die übergebene Manifest-Datei muss dabei nach dem Dateinamen der zu erstellenden `jar`-Datei stehen und wird somit in das neu erstellte Java-Archiv in dem darin angelegten Unterverzeichnis `META-INF` übernommen.

[229] JavaBeans werden im Rahmen dieses Buchs nicht behandelt.

Literaturverzeichnis

Cun86 Cunningham, W. und Beck, K., "A Diagram for Object-Oriented Programs", Proceedings of OOPSLA-86, Oktober 1986

Dij72 Dijkstra, E.W., "Go To Statement Considered Harmful", in Communications of the ACM, vol. 15, no. 10, p. 859, October 1972

DIN 66001 DIN 66001, "Sinnbilder und ihre Anwendung", Beuth-Verlag, 1983

DIN 66261 DIN 66261, "Sinnbilder für Struktogramme nach Nassi-Shneiderman", Beuth-Verlag, 1985

Eck00 Eckel, B., "Thinking in Java", Second Edition, Prentice Hall, 2000

Gos97 Gosling, J., Joy, B., Steele, G., "Java – Die Sprachspezifikation", Addison-Wesley, 1997

Hru98 Hruschka, P., "Ein pragmatisches Vorgehensmodell für die UML", OBJEKTspektrum 2/98, Seite 34

IEEE 754 IEEE 754 Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985

Jac92 Jacobson, I., "Object-Oriented Software Engineering, A Use Case Driven Approach", Addison-Wesley, ISBN: 0-201-54435-0, 1992

Ker78 Kernighan, B.W., Ritchie, D.M., "The C Programming Language", Prentice Hall, 1978

langer http://www.angelikalanger.com/Articles/EffectiveJava/05.Clone-Part1.html

Lis88 Liskov, B., "Liskov Substitution Principle, Data Abstraction and Hierarchy", SIGPLAN Notices, 23 , 5, (Mai 1988)

Mey97 Meyer, B., "Object-Oriented Software Construction", 2nd ed., Prentice Hall, 1997

Mon06 Monson-Haefel R., "Enterprise JavaBeans 3.0", 5. Auflage, O'Reilly, ISBN: 059600978X, 2006

Nas73 Nassi I., Shneiderman B.: "Flowchart Techniques for Structred Programming", in SIGPLAN, S. 12-26, Aug.1973

Now04 Nowak, J., "Fortgeschrittene Programmierung mit Java 5, Generics, Annotations, Concurrency und Reflection – mit allen wesentlichen Neuerungen des J2SE 5.0", dpunkt Verlag, 2004

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1

Oak98 Oaks, S., "Java Security", OReilly Associates, 1998

oraexe http://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html

Sch64 Schmeil, 0., "Tierkunde", Quelle & Meyer Heidelberg, 1964

Uni07 "The Unicode Standard", Version 5.0, Addison-Wesley, 2007, ISBN: 0-321-48091-0

Index

"is a"-Beziehung 415
Ableitung.....Siehe Vererbung
`abstract` 183, 451, 596
abstrakte Basisklasse.....451, 612
abstrakte Methode.....596
Abstraktion.....43, 588
Abstraktionsebene.....47
Adresse 20
Aggregation 49, 107, 414
Aktualisierungs-Ausdrucksliste.....301
aktuell parametrisierte Klasse.....727
  Repräsentation ~.....730
aktueller Parameter 327
aktueller Typ-Parameter 727
Algorithmus.....4, 11
Alphabet 18
  Binäralphabet.....18
  Dezimalalphabet 18
Alternative
  einfach 13
  mehrfach.....14
Anführungszeichen
  doppeltes ~ 131
  einfaches ~ 131
Annotation 229
  Typ-~.....232
Annotationen
  wiederholende ~.....232
anonyme Klasse 647
  Realisierung ~.....661
anonymes Paket.....521
Anweisung 6, 248
  `assert`-~.....249, 575
  `break`-~.....299, 306
  `continue`-~ 308
  `do-while`-~ 305
  `else`-.....293
  `else-if`-~.....295
  `for`-~ 300
  `if`- 293
  `if-else`-~.....294
  Iterations-~.....249
  leere ~.....249, 320
  `return`-~.....325
  Selektions-~ 249
  Sprung-~ 249
  `super()`-~..... 438
  `switch`-~..... 295
  `synchronized`-~.....249, 699, 701
  `throw`-~.....249, 555
  `try`-~ 249, 549
  zusammengesetzte ~..... 292
API
  ~-Dokumentation..... 76
  Java-~ 76
Applet 71
Archiv..... 83
arithmetischer Ausdruck 14
Array 184
  aus Basisklassen 192
  aus einfachen Datentypen 186
  aus Referenzen..... 189
  Casten auf einen ~-Typ..... 436
  Datenfeld `length`..... 187
  Default-Wert..... 187
  eindimensionales ~ 184
  Erzeugung ~.....186, 187, 189
  Grenzen 186
  Initialisierung 186, 188, 190
  Initialisierungsliste 188, 191
  Komponente..... 184
  Länge 184
  mehrdimensionales ~..... 192
  Objektcharakter..... 191
  offenes ~ 192
  Referenzen auf ~ 187
  Syntax-Varianten..... 195
Array-Index 184
Array-Objekt..... 185
Array-Variable..... 187
ASCII-Zeichensatz..... 19
`assert`..... 575
Assoziativität.....252, 275
Aufrufschnittstelle 42, 591, 605
  Sohnklasse..... 420
  Vaterklasse 420
Aufzählungskonstante 142, 196
Aufzählungstyp 196
Ausdruck.....92, 246, 248
  bedingter ~..... 273
  Rückgabewert eines ~s..... 248
Ausdrucksanweisung..... 249

© Springer Fachmedien Wiesbaden 2016

J. Goll, C. Heinisch, *Java als erste Programmiersprache*, DOI 10.1007/978-3-658-12118-1

ausführbares Programm 77
Ausgabe
  formatierte ~.....117
Ausgabestrom 811
Ausgabestrom-Klasse 812
Ausnahme ..... *Siehe* Exception
Auswertungsreihenfolge
  Operatoren.....251
Backslash 131
Basisklasse
  abstrakte ~ 451, 612
batch-Betriebssysteme 670
bedingte Anweisung 293
Bedingung 14
Bedingungsoperator 273
Betriebsmittel.....673
  exklusiv benutzt 674
  räumlich aufteilbar.....674
  virtuell.....672
  zeitlich aufteilbar 674
Betriebssystem-Prozess.....672, 675
Bewertungsreihenfolge
  Operanden 253
Bezeichner.....138
Bibliotheksklasse 24
Big-Endian 830
Bindung 425
  dynamische ~...*Siehe* späte Bindung
  finale Methode 428
  frühe ~.....425, 426
  private Methode 428
  späte ~ 425, 426
  statische ~..... *Siehe* frühe Bindung
Bit 18
Bit-Operatoren 268
  logische ~.....268
blank.....131
Block.....12, 292, 318
  leerer ~.....320
  Schachtelung 321
blocked 678
`boolean` 158
boolesche Konstante 142
boolescher Ausdruck 13
boolescher Wert 147
`break`.....299, 306
BREAK-Anweisung.....16
Bridge-Klasse 832, 834, 836
Buchstaben.....130
`byte`.....158
Bytecode.....71, 74, 78
Bytecode-Interpreter 75
Bytestream-Klasse.....813, 817
call by value 328
`case` 296, 297
case sensitiv 132
`case`-Marke 296, 297
cast-Operator.....96, 276, 433
`char` 158
Character Encoding 836
Characterstream
  Klasse 813, 832
Checked Exception..... 559
`class` 162
`Class<T>` 766
`-classpath` 521
`CLASSPATH` 85, 520
Code 19
Codeblöcke
  synchronisierte ~ 699
Codierung 18
Coercion 774
Cohesion 588
Collections 794
  ~-Schnittstellen 798
  Lists..... 798
  Maps 798
  Queues 798
  Sets..... 798
Compiler 21
concurrent..... *Siehe* nebenläufig
`continue` 308
Coupling 588
critical section .....
  ..... *Siehe* kritischer Abschnitt
Dämon-Thread..... 714
Dateiende 116
Dateinamenserweiterung..... 78
Dateipuffer 116
Dateizeiger 116
Datenabstraktion..... 160
Datenfeld 40, 166
  Default-Initialisierung..... 379
  klassenbezogenes ~ 362
  manuelle Initialisierung..... 380
  objektbezogenes ~ 362
  Verdecken 444, 445
Datenquelle..... 810
Datensenke..... 810
Datentyp 21, 156
  abstrakter ~ 160
  Bibliotheksklasse..... 24

boolean 158
byte 158
char 158
double 158, 160
einfacher ~ 21, 23, 157, 158
float 22, 158, 160
int 21, 158
Klasse 35
Klassen-Typ 160
Konvertierung von ~ 276
long 158
null 157
Referenztyp 160
selbst definierter ~ 24, 35, 157
short 158
Standardtyp 21
Übersicht 156
Wandlung 218
Deadlock 709
default 525
Default-Wert 379
defensive Programmierung 263
Definition 169
Klassen-~ 162
Variable 169
Deklaration 169
Methoden-~ 163
Deklarationsanweisung 319
Dekorierer 817
Delegationsprinzip 108
Design by Contract 466
Dezimalzahl 143
Diamant-Operator 725, 727
Division
ganzzahlige ~ 258
double 158, 160
do-while-Schleife 16, 305
Down-Cast 431, 433
dynamisches Laden 78
Eigenschaften 414
Einfachvererbung
Schnittstelle 611
Eingabedatei 116
Eingabestrom 116, 811
Eingabestrom-Klasse 812
eingebettete Klasse .... *Siehe* geschachtelte Klasse
Einheit
Memory Management 675, 676
Scheduling 675, 676
Elementklasse 636, 638
Einschränkungen 641
Realisierung 657
Syntax 639
Zugriffsschutz 638
else-if 295
Encoding 836
Setzen eines ~ 837
Endian
Big-~ 830
Little-~ 830
Endlos-Schleife 300, 304
Entwurfsmuster
Dekorierer 817
enum 196
enumeration type .... *Siehe* Aufzählungstyp
Ersatzdarstellung 131, 148
in konstanten Zeichenketten 148
in Zeichenkonstanten 148
oktal 150
Unicode ~ 131, 149
Erstellung von Paketen 511
Erweitern 420
Euklid 5, 17
Exception 546
ableiten 555
Ankündigen einer ~ 569
AnnotationFormatError 560
ArithmeticException ...258, 560
ArrayIndexOutOfBoundsException 186, 562
AssertionError 560
CertificateException 560
Checked ~ 559
ClassCastException 435, 560
ClassNotFoundException 851
CloneNotSupportedException 398, 560, 619
DataFormatException 560
Error 558
Exception 558
fangen 551, 562
IllegalMonitorStateException 706
IllegalThreadStateException 680
IndexOutOfBoundsException 560
InterruptedException .680, 681

IOError 560
Klassenbaum 555
NoSuchMethodError 856
NotSerializableException 849
NullPointerException .558, 560
OutOfMemoryError 395
Propagieren einer ~ 552
RuntimeException 558
Throwable 557
Unchecked ~ 559
VirtualMachineError 560
Weiterreichen einer ~ 564
werfen 551
Exception Handler 551
Reihenfolge 563
Exception-Hierarchie 557, 563
executable program 77
Exemplar ....*Siehe* Instanz
Exponent 22, 160
Exponential-Anteil 145
Exponentialzahl 22
extends 595, 596
Extension 78
Externalizable
writeExternal() 853
Fallunterscheidung 8
final 597
finale Klasse 449
finale Methode 449
finalize() 397, 399
flache Kopie 620
Fließkommakonstanten 145
float 158, 160
Flussdiagramm 12
for 300, 303
formaler Parameter 327
formaler Typ-Parameter 724, 726
Ersetzung beim Kompilieren 733
formatierte Ausgabe 117, 840
for-Schleife 300
friendly ....*Siehe* default
ganze Zahl 21
Ganzzahlige Konstante 142, 143
Garbage Collector 182, 394
Gegenschrägstrich 131
Generalisierung 47, 418, 419
Generics 724
generische Klasse
Code Sharing 728
generische Klassen
Implementierung von ~ 726
Instanzmethoden von ~ 729
Speicherabbild von ~ 733
Subtyping und ~ 735
Typ-Parameter-Sektion 734
Übersetzen von ~ 728
Vererbungsbeziehungen und ~ .. 738
generische Methode 742
Notation 743
generische Parameter
Bounds und Schnittstellen 758
generische Schnittstelle
Comparable 762
Implementierung von ~ 755, 757
Notation 755
generische Schnittstellen
Vererbungsbeziehungen und ~ .. 761
generischer Konstruktor
Notation 743
generischer Parameter
Raw Type 731
Generizität 724, 773
Bounded Typ-Parameter 747
generische Klasse 726
generische Methode 742
generische Schnittstellen 755
generischer Konstruktor 743
Klasse Class<T> 766
Lower Bound Wildcard 753
Polymorphie und ~ 772
Unbounded Wildcard 749
Upper Bound Wildcard 752
geschachtelte Klasse
anonyme Klasse 385, 647
Elementklasse 636, 638
lokale Klasse 636, 642
Realisierung 656
statisch ~ 653
geschachtelte Schnittstelle 637
Gleitpunktkonstante 142, 145
Exponent 145
Mantisse 145
Typ-Suffix 146
Gleitpunktzahl 21, 160
IEEE-Format 160
Gültigkeit 322
Gültigkeitsbereich 168
Klassenname 521
Halbbyte 144
Heap 174, 176, 181, 373
Hierarchie 47

"is a"-~ 47
"kind-of"-~ 47
Exception-~.....557
Schnittstellen-~ 596
Hilfsklasse 512
Hilfsmethode.....39
IEEE 754 .....146, 160
`if-else` 294
Imperative Sprachen 19
`implements`.....598, 605
`import`.....514
Information Hiding 43, 362, 588
Initialisierung.....176, 379
automatische ~.....
..... *Siehe* Default-Initialisierung
Datenfeld.....379
Default-~ 379
Initialisierungsblock.....384
manuelle ~ 179, 380
mit Initialisierungsblock 382
mit Konstruktor.....99
Reihenfolge.....388
Variable.....10
von Arrays.....190
Initialisierungsblock 382
nicht statischer ~.....384
statischer ~ 382
Initialisierungsklausel.....301
Initialisierungsliste
für Arrays 191
Initialisierungsreihenfolge 388
Initialwerte 20
Inklusions-Polymorphie 774
innere Klasse.....
.....*Siehe* geschachtelte Klasse
Inputstream..... *Siehe* Eingabestrom
InputStream
`available()` 824
`close()` 824
`mark()` 824
`markSupported()` 824
`read()` 824
`reset()` 824
`skip()` 824
`instanceof`.....474, 619
Instanz.....53
Instanziierung 53, 392
Ablauf bei der ~.....393
Instanzmethode 46, 102
Instanzmethoden
synchronisierte ~..... 698
Instanzvariable.....46, 178, 362
`int` 158
Integer .....*Siehe* `int`
Integer-Erweiterung 279
integral promotion .....
..... *Siehe* Integer-Erweiterung
`interface` 589, 595
Interpreter 78
Invariante.....468, 469
is a-Beziehung 111
Iteration.....6, 7, 299, 345, 804
for-each Schleife 806
Iterator..... 804
`java` 3
Java 7
binäre Konstante 143
Konstante mit Unterstrich 145
multi-catch..... 550
try-with-resources..... 553
Java 8
`FunctionalInterface` 233
Typ-Annotationen..... 232
wiederholende Annotationen..... 232
Java Development Kit.....74, 82
Java Runtime Environment..... 74
`JAVA_HOME` 84
Java-Anwendung 86
Java-API 76
Java-Applet..... 86
JavaBeans 72
`javac`..... 3, 78
`javadoc`..... 135
Java-Plattform..... 73
JavaScript 86
JavaServer Faces 86
JavaServer Pages 86
Java-Virtuelle Maschine..... 74
JDK.....*Siehe* Java Development Kit
Bezugsquelle..... 82
JIT-Compiler 75
JRE. *Siehe* Java Runtime Environment
Kapselung.....39, 43
Klasse.....24, 37, 508
abgeleitete ~ 111
aktuell parametrisierte ~..... 727
anonyme ~ 647
`BigInteger`..... 22
`BufferedInputStream` 826
`BufferedOutputStream` 821

BufferedReader 834
BufferedWriter 834
ByteArrayInputStream 825
ByteArrayOutputStream 821
ByteInputStream 824
CharArrayReader 834
CharArrayWriter 834
DataInputStream 826, 830
DataOutputStream 822, 830
Definition einer ~ 162
Element-~ 636, 638
Error 558
Exception 558
FileInputStream 825
FileOutputStream 821
FileReader 834
FileWriter 836
FilterReader 834
FilterWriter 834
finale ~ 449
Formatter 841
generische ~ 726
InputStream 813, 814, 840
InputStreamReader 834
InputStreamWriter 836
Instanziierung einer ~ 392
LineNumberReader 834
lokale ~ 636, 642
NullPointerException 150
Object 76, 398
ObjectInputStream 826, 848
ObjectOutputStream 822, 848
OutputStream 813, 814, 818
OutputStreamWriter 836
PipedInputStream 825
PipedOutputStream 821
PipedReader 834
PipedWriter 834
PrintStream 822, 840
PrintWriter 834
PushbackInputStream 826
PushbackReader 834
Reader 813, 814, 835
RuntimeException 558
Scanner 117, 119
selbst definierte ~ 85
SequenceInputStream 826
Start-~ 82
static geschachtelte ~ 636
String 205
StringBuffer 205, 211
StringBuilder 205, 216
StringReader 834
StringWriter 834
Thread 682
Throwable 554, 557
Vertrag 470
Writer 813, 814, 832, 833
Klassenbaum
Exception 557
Klasseninvariante 469
Klassenlader 78, 82
Klassenliteral 768
Klassenmethode 46, 367
synchronisierte ~ 697
Klassennamen 52
Klassensuchpfad 518
Klassen-Typ 160
Klassenvariable 46, 178, 362, 365
kombinierte Zuweisungsoperatoren 261
Kommentar 52, 134
~block 135
Dokumentations~ 135
Zeilen~ 135
Kompilierung 78
Komposition 49
Konkatenationsoperator 149
Konstante 133, 141, 142
Aufzählungs~ 196
binäre ~ 143
boolesche ~ 142
dezimale ~ 143
false 147
ganzzahlige ~ 142, 143
Gleitpunkt-~ 142, 145
hexadezimale ~ 143
literale ~ 142
MAX_VALUE 146
MIN_VALUE 146
NaN 146
NEGATIVE_INFINITY 146
Null-~ 142, 150, 157
oktale ~ 143
POSITIVE_INFINITY 146
String-~ 142
symbolische ~ 93, 94, 141
symbolische Gleitpunkt-~ 146
true 147

Unterstrich....145
Zeichen-~....142, 147
konstante Variable....182
konstante Zeichenkette 149, 205
Konstruktor
abgeleitete Klasse....437
Aufruf eines ~ im ~....391
Default-~ 101, 389, 390, 439
mit Parameter 389, 390
parameterloser ~....389
`super()` 437
vordefinierter Default-~ 101
Zugriffsschutz....530
Kontextwechsel 673
Kontravarianz 482
Kontrollfluss....6, 292
Kontrollstruktur 7, 292
Iteration....7
Selektion 7
Sequenz....7, 292
Kooperation 675
Kopie
flache ~ 620
tiefe ~ 621
Kovarianz....482
kritischer Abschnitt 675, 693, 696
bedingt 696
Laden
dynamisches ~....78
Laufzeit....77
Laufzeitsystem....174
Lebensdauer....168, 322
leere Anweisung....320
leerer Block....320
Leerzeichen....131, 134
leichtgewichtiger Prozess 675
Lesezeiger .... *Siehe* Dateizeiger
Lexikalische Analyse 79
lexikalische Einheiten 133
LIFO 180
linksassoziativ....252
liskovsches Substitutionsprinzip 453, 454
~ und generische Klassen....736
Literale....142
Little-Endian....830
logische Bit-Operatoren....268
logische Operatoren 265
lokale Klasse 636, 642
Einschränkungen 643
in einer Instanzmethode....644
in einer Klassenmethode.... 644
lokale Variable 178, 320
`long` 158
Lösungsbereich 36
Lower Bound Wildcard.... 753
lvalue ....*Siehe* L-Wert
L-Wert.... 254
makroskopischer Zustand.... 40
Mantisse 22, 145, 160
Marke....296, 297, 306
Anweisung.... 306
mehrfache Alternative.... 295
Mehrfachvererbung
Schnittstelle.... 611
Mehrfachzuweisung.... 260
Memory Management.... 675
Method-Area 179, 182, 373, 476
Methode....40, 163, 318, 324
abstrakte ~ 596
Aufruf einer ~ 324
`clone()` 398, 618, 620
`close()` 818
`compareTo()` 763
Definition 324
Deklaration 324
eigenständig generische ~ 742
`equals()` 398
finale ~ 449
`finalize()` 397, 399
`flush()` 818
`format()` 841, 843
`forName()` 768
`getMessage()` 557
`getPriority()` 691
`hasNext()` 805
Instanz-~ 46
`interrupt()` 680, 681
intValue().... 221
`isAlive()` 682
`isDaemon()` 714
`join()` 680
Klassen-~ 46, 367
mit Parametern....166, 325
`newInstance()` 768
`next()` 120, 805
`nextInt()` 120
`notify()` 679, 705
`notifyAll()` 679
ohne Parameter 165, 324
`parseInt()` 219

print() 840
printf() 117, 841
println() 840
readDouble() 831
readExternal() 853
readInt() 831
run() 683, 685
setDaemon() 714
setPriority() 691
sleep() 680
start() 680, 683
toString() 398
Überladen einer ~ 337
Überschreiben einer ~....424
valueOf() 220, 221, 226
Vertrag einer ~ 468
wait() 679, 705
write() 818
writeDouble() 831
writeInt() 831
yield() 680, 690
Methoden
überschreiben 421
Methodendeklaration 163
Methodenkopf....104, 163, 425
Methodenrumpf 104, 163, 324, 425
Methodentabelle 476
mikroskopischer Zustand....40
Modifikator 169, 183
abstract....183, 451, 596
final 597
synchronized 183, 697, 701
transient....183, 852
Modulo-Operator 23
Monitor 703
Monitorkonzept
~ allgemein 695
~ in Java 697
most significant bit 159
MSB.... *Siehe* most significant bit
multi-catch-Block 550
Multiplexen 672
Multithreading 676
mutual exclusion....
.... *Siehe* wechselseitiger Ausschluss
Nachbedingung 468
Nachricht 42
Name 133, 138
einfacher ~ 510
Variablen-~ 168
voll qualifizierter ~ 510
Namensraum 510, 637
Nassi-Shneiderman-Diagramm.... 11
Nebeneffekt 250
Ausführungszeitpunkt.... 250
Operator 268
nebenläufig 670
Nebenwirkung.... *Siehe* Nebeneffekt
nested class....
....*Siehe* geschachtelte Klasse
new 174
Nullkonstante 142, 150, 157
null-Referenz 157, 170
null-Typ 157, 170
Oberflächenprogrammierung
Adapter-Klasse.... 643
ObjectOutputStream
readObject() 849
writeObject() 849
Objekt 37
zusammengesetzt 107
Objekterzeugung 97, 392
Objektserialisierung 848
offenes Array 192
Operand.... 246
Operator 133, 151, 246
Additions-~ 257
Additions-Zuweisungs-~.... 261
arithmetischer ~ 255, 257
Bedingungs~ 273, 478
binärer ~....23, 246, 248
Bit-Operatoren.... 268
cast-~ 96, 276
Divisions-~ 258
dreistelliger ~.... 246
einstelliger ~....246, 251
Exklusiv-ODER-Operator für Bits 270
Gleichheits-~ 262
Größer-~ 264
Größergleich-~ 264
instanceof-~ 474
Kleiner-~.... 264
Kleinergleich-~ 264
Linksshift-~.... 273
logischer ~.... 265
logischer Exklusiv-ODER-~.... 266
logischer Negations~.... 267
logischer ODER-~ 266
logischer UND-~.... 265
mehrstelliger ~ 251, 252
Modulo-~ .... *Siehe* Restwertoperator

Multiplikations-~ 257
Negationsoperator für Bits 270
negativer Vorzeichen-~ 256
`new`-~ 174
ODER-Operator für Bits 269
positiver Vorzeichen-~ 255
Postfix-Dekrement-~ 256
Postfix-Inkrement-~.....256
Präfix-Dekrement-~.....256
Präfix-Inkrement-~ 256
Prioritätentabelle 274
Punkt-~ 166
Punkt-~ für Referenzen.....478
Rechtsshift-~.....272
relationaler ~ 262
Restwert-~ 23, 259
Shift-~ 268, 271
Subtraktions-~.....257
ternärer ~ 246, 248
unärer ~ 23, 246
UND-Operator für Bits.....269
Ungleichheits-~ 263
Vergleichs-~.....94, 262
Verkettungs-~.....478
Vorzeichen-~.....144
Zuweisungs-~.....94, 260, 479
zweistelliger ~ 246
`OutOfMemoryError` 174
Outputstream..... *Siehe* Ausgabestrom
Outputstream-Klasse.....818
Overloading .....Siehe überladen
`package` 511
Paket 72, 508, 509
anonymes ~ 521
Deklaration.....511
`java.lang`.....516
`java.util`.....794
Komponente.....513
Static imports 516
Unterpaket 515
Paketname 511
eindeutiger ~ 521
vollständiger ~.....520
parallel.....671
Parameter
aktueller ~ 106, 327
Auswertungsreihenfolge aktuelle ~ 331
einfacher Datentyp.....328
formaler ~.....106, 327
Referenztyp.....329
Parameterliste variabler Länge..... 340
Parameterübergabe
Programm 342
Parser 133
`PATH`-Variable..... 84
Pipe 704
Plattform 73
Polymorphie.....414, 453
Inklusions-~..... 774
von Methoden 335
von Objekten 453
von Operationen..... 453
Portierbarkeit 73
Postfix-Operator..... 247
Präfix-Operator 247
preemptive scheduling..... 672
`printf()` 117
Formatelement..... 117
Formatstring..... 117
Priorität 275
logische Operatoren..... 267
Operatoren..... 252
prioritätengesteuert..... 690
Prioritätentabelle
Operatoren..... 274
`private`..... 525
Problembereich.....34, 46
Processingstream-Klasse 814, 821
Programm..... 4, 11
Programmieren im Kleinen 70
Programmierstil.....94, 139
Programmierung
defensive..... 263
Propagieren 552
`protected` 525
Prozess
leichtgewichtiger ~..... 675
schwergewichtiger ~.....673, 675
Prozesskontext 672
Prozesskonzept 670, 671
Prozessumgebung..... 673
`public`..... 525
Pufferung
zeilenweise ~ 116
Punktoperator 166
für Referenzen 478
quasi-parallel 671
Quellcode-Datei 78
Quelldatei 78
Queue..... *Siehe* Warteschlange
Race Condition 691

Reader
close() 835
mark() 835
markSupported() 835
read() 835
ready() 835
reset() 835
skip() 835
Reader/Writer-Problem 691
ready-to-run 678
Rechner-Plattform 71
rechtsassoziativ 252, 253
reelle Zahlen 23
Referenz 49, 172
null-~ 157, 170
this-~ 372
Zuweisung an eine ~ 172
Referenztyp 160
Schnittstelle 599
Referenzvariable 170, 171, 599
Rekursion 345
direkt 345
indirekt 345
Relationale Operatoren 262
reserviertes Wort . *Siehe* Schlüsselwort
return-Anweisung 325
RTTI ....
.. *Siehe* Run Time Type Identifikation
Rückgabetyp 248
kovarianter ~ 484
Rückgabewert 246, 248, 325
Run Time Type Identifikation 475
running 678
rvalue .... *Siehe* R-Wert
R-Wert 254
Satzzeichen 131, 133, 150
Scanner 133
Scheduler 672
Scheduling 675
Schleife 92
abweisende ~ 15, 300
annehmende ~ 15, 305
endlose 300, 304
for-each-~ 303, 806
ohne Bedingungsprüfung 16
Schlüsselobjekt 700, 701
Schlüsselwort 133, 139
final 449
Schnittstelle 509, 588
als Datentyp 599
Cloneable 615
Collection<E> 801
Comparable 762
Deklaration 595
Externalizable 853
generische ~ 755
Implementieren einer ~ 590, 598
Iterable<E> 801
Iterator<E> 805
Mehrfachvererbung 611
Runnable 682, 685
Serializable 849
static geschachtelte 637, 653
Typsicherheit 601
Vererbung 611
Schnittstellen
Implementieren mehrerer ~ 602
schwergewichtiger Prozess 673, 675
Seiteneffekt .... *Siehe* Nebeneffekt
Seitenvorschub 134
Selektion 6, 7, 8, 293
bedingte Anweisung 293
bedingte Verarbeitung 14
einfache Alternative 13, 293
mehrfache Alternative 14, 295
Semantik
dynamische ~ 80
statische ~ 80
Semantische Analyse 80
Semaphor 693
signal()-Befehl 693
wait()-Befehl 693
Warteschlange 693
Semaphorkonzept 693
Sequenz 6, 7, 8, 292
Serialisierung 848
serialVersionUID 852
Service-Klasse .... *Siehe* Hilfsklasse
Service-Methode... *Siehe* Hilfsmethode
Servlet 86
short 158
Sichtbarkeit 322, 510
Klasse 525
Schnittstelle 525
single entry 8
single exit 8
Singleton 393
Sinkstream-Klasse 814, 820, 834
Sohnklasse 414, 420
Sonderzeichen 131

Spezialisierung 48, 418, 419
Sprachen
  formal 11
  natürlich 11
Springstream-Klasse 814, 825
Sprunganweisung 306
Stack 176, 180
Standardausgabe 840
Standardeingabe 840
Standardtyp 21
Standard-Wert 379
Stapel .....*Siehe* Stack
Startklasse 78, 82
statisch geschachtelte Klasse.....653
statisch geschachtelte Schnittstelle 653
statische Top-Level-Klasse.....653
statische Top-Level-Schnittstelle 653
Stellenwert-Tabelle 144
Steuerzeichen.....131
Stream-Konzept.....814
String 2, 205
  explizite Erzeugung.....206
  implizite Erzeugung.....208
  Methode `equals()` 209
  Methode `length()` 206, 209
  Methode `substring()` 210
  Methode `trim()` 210
  Methoden der Klasse ~ 209
  Vergleichen 209
  Verkettung.....214, 215
string constant 149
string literal 149
`StringBuffer`
  Erzeugung.....211
  Vergleichen 214
  Verkettung.....214
`StringBuilder`.....216
String-Konstante 142, 149
String-Objekt.....206
Strom-eindeutige Bezeichner 852
Struktogramm 11
Strukturierte Programmierung ..8, 9, 11
Styleguide.....139
  Paketname.....512
Subsystem.....588
Subtyping.....111
`super`.....446
`super()` 113, 437, 438
`switch`.....295
  `case` 296
  `default`.....297
symbolische Gleitpunkt-Konstanten 146
symbolische Konstante 93, 94
Synchronisation 674, 675
    Instanzmethoden 698
  Blöcke 702
  Codeblöcke 699
  Instanzmethoden..... 702
  Klassenmethode 697
  klassenübergreifend..... 703
  mutual exclusion 702
  Schlüsselobjekt 701
`synchronized`.....183, 697, 701
Syntaxanalyse 79
Tabulator 134
Tastatur-Eingabespeicher..... 116
Tastaturpuffer 114, 115
Teilsystem .....*Siehe* Subsystem
`this` 372
`this()` 391
`this`-Referenz 372
Thread 508, 670
  Beenden von ~ 687
  Erzeugung..... 682
  Priorität..... 691
  Scheduling 690
`throw` 555
`throws`.....553, 559, 569
tiefe Kopie..... 621
Time Slice 672
Token..... 133,
  ..... *Siehe* lexikalische Einheiten
Top-Level-Klasse 656
Top-Level-Schnittstelle 656
Trace-Tabelle..... 9
`transient` 183, 852
Trenner 133
`try`-Anweisung 549
  geschachtelt 568
try-with-resources 553
Typ
  Aufzählungs-~ 196
  `null`-~ 157
Typ Erasure
  Bounded Typ-Parameter 747
Typ-Annotationen 232
Typbegriff..... 615
Type Erasure 731, 761
Typerkennung..... 475
Typkonvertierung
  einfacher Datentypen 277
  explizite ~ 96, 276, 430

für binäre Operatoren.....279
für numerische Typen 278, 281
implizite ~.....96, 276, 430
implizite ~ von Referenzen.....430
Referenz 429
Vorschriften.....281
Typmodifikator..... *Siehe* Modifikator
Typ-Parameter
aktueller ~ 727
Ersetzung des formalen ~ 728
formaler ~.....724, 726
mehrere formale ~.....734
Typ-Parameter-Sektion
~ bei generischen Konstruktoren 743
~ bei generischen Methoden.....743
Typsicherheit
~ von Schnittstellen.....601
Erhöhung der ~ 725
Typ-Suffix 145
Typumwandlung ..... *Siehe* Typkonvertierung
Übergabeparameter 178
Überladen.....337
Überschreiben 421
Zugriffsmodifikator 532
zur Optimierung 423
zur Verfeinerung 423
Umwandlungszeichen 841, 842
Unchecked Exception.....559
Unicode 19
Unicode-Ersatzdarstellungen.....149
Unicode-Zeichen
Ersatzdarstellung 132
Up-Cast 431, 433
UTF-16 132
UTF-32 132
UTF-8 132
varargs 340
Variable 9, 19, 156
Datentyp.....21
Definition.....169
dynamische ~.....166, 174
Instanz-~ 46, 178
Klassen-~.....46, 178, 365
konstante ~ 182
lokale ~ 167, 178, 320
ohne Namen 181
`PATH`-~.....84
Referenz-~ 170
statische ~.....166, 168
Variablennamen.....19, 21, 168
Wert 19, 21
variable Zeichenkette.....205, 211
Variablennamen..... 19
Vaterklasse 420
Verarbeitungsschritt..... 12
Sinnbild 12
verdecken 444, 445
Vereinbarung 169
`import`-~..... 514
Vererbung.....111, 414
Exception 555
Schnittstelle.....596, 611
Vererbungshierarchie 418
Verfeinerung 589, 590, 613
Vergleichsoperator.....94, 262
Verkettungsoperator 96, 149
~ für String-Objekte 478
Verknüpfungsreihenfolge 267
Verschiebeoperator .....
.....*Siehe* Shift-Operator
Vertrag..... 466
einer Klasse 470
einer Methode 468
virtueller Prozessor 671
Vorbedingung 468
Vorzeichenbit..... 160
Vorzeichenoperator 144
Wahrheitstabelle
bitweise Negation..... 270
Exclusiv-ODER 270
ODER..... 269
Wahrheitswert.. *Siehe* boolescher Wert
Warteschlange..... 798
Web-Server..... 72
wechselseitiger Ausschluss 675, 693
Whitespace-Zeichen 133, 134
Wiederverwendung
Quellcode..... 414
Wildcard
Lower Bound ~..... 753
Unbounded ~ 749
Upper Bound ~..... 752
Wrapper-Klasse..... 146
`Void` 219
Writer
`close()` 833
`flush()` 833
`write()` 833
Writer-Klasse 832
Zahlensysteme 143
Zahlenüberlauf..... 22

Zeichen 18
Buchstaben 18
einlesen von ~ 114
Sonderzeichen 18
Steuerzeichen 18
Whitespace-~ 133, 134
Ziffern 18
Zeichenkette 2, 205
konstante ~ 131, 149, 205
variable ~ 205, 211
Zeichenkonstante 142, 147
Zeichenliteral 147
Zeichensatz ..... *Siehe* Zeichenvorrat
Unicode 130
Zeichenvorrat 18, 19
Zeilentrenner 134
Zeitscheibe 672
Zeitscheiben-Betriebssystem 672
Ziffer 18, 131
Zugriffsmodifikator 53, 524
beim Überschreiben 531
`private` 524
`protected` 524
`public` 524
Schnittstelle 597
Zugriffsschutz 524
Datenfeld 525
default 524
Elementklasse 638
Klasse 525
klassenbezogen 526
Konstruktor 530
Methode 525
objektbezogen 526
Schnittstelle 525
Übersicht 529
zusammengesetzte Anweisung 318
zusammengesetzte Objekte 107
Zusicherung 468
Invariante 467, 469
Nachbedingung 467
Vorbedingung 468
Zustand
blocked 678
eines Objektes 40
makroskopischer ~ 40
mikroskopischer ~ 40
ready-to-run 678
running 678
Zustandsübergangsdiagramm
für Betriebssystem-Prozesse 678
Thread 679, 681
Zuweisung 10
zuweisungskompatibel 277
Zuweisungsoperator 94, 259, 479
Zuweisungsoperatoren
kombiniert 261
Zweierkomplement 159
Zwischencode 79

Lizenz zum Wissen.

Sichern Sie sich umfassendes Technikwissen mit Sofortzugriff auf tausende Fachbücher und Fachzeitschriften aus den Bereichen: Automobiltechnik, Maschinenbau, Energie + Umwelt, E-Technik, Informatik + IT und Bauwesen.

Exklusiv für Leser von Springer-Fachbüchern: Testen Sie Springer für Professionals 30 Tage unverbindlich. Nutzen Sie dazu im Bestellverlauf Ihren persönlichen Aktionscode **C0005406** auf *www.springerprofessional.de/buchaktion/*

Springer für Professionals.
Digitale Fachbibliothek. Themen-Scout. Knowledge-Manager.

- Zugriff auf tausende von Fachbüchern und Fachzeitschriften
- Selektion, Komprimierung und Verknüpfung relevanter Themen durch Fachredaktionen
- Tools zur persönlichen Wissensorganisation und Vernetzung

www.entschieden-intelligenter.de

Springer für Professionals

Springer