

Erwin Merker  
Roman Merker

## **Programmieren lernen mit JAVA**

## Aus dem Bereich IT erfolgreich lernen

### **Lexikon für IT-Berufe**

von Peter Fetzter und Bettina Schneider

### **Grundkurs IT-Berufe**

von Andreas M. Böhm und Bettina Jungkunz

### **Java für IT-Berufe**

von Wolf-Gert Matthäus

### **Prüfungsvorbereitung für IT-Berufe**

von Manfred Wünsche

### **Grundlegende Algorithmen**

von Volker Heun

### **Algorithmen für Ingenieure – realisiert mit Visual Basic**

von Harald Nahrstedt

### **Grundkurs Programmieren mit Delphi**

von Wolf-Gert Matthäus

### **Grundkurs Visual Basic**

von Sabine Kämper

### **Visual Basic für technische Anwendungen**

von Jürgen Radel

### **Grundkurs Smalltalk – Objektorientierung von Anfang an**

von Johannes Brauer

### **Grundkurs Software-Entwicklung mit C++**

von Dietrich May

### **Grundkurs JAVA**

von Dietmar Abts

### **Aufbaukurs JAVA**

von Dietmar Abts

### **Grundkurs Java-Technologien**

von Erwin Merker

### **Java ist eine Sprache**

von Ulrich Grude

### **Middleware in Java**

von Steffen Heinzl und Markus Mathes

### **Das Linux-Tutorial – Ihr Weg zum LPI-Zertifikat**

von Helmut Pils

### **Rechnerarchitektur**

von Paul Herrmann

### **Grundkurs Relationale Datenbanken**

von René Steiner

### **Grundkurs Datenbankentwurf**

von Helmut Jarosch

### **Datenbank-Engineering**

von Alfred Moos

### **Grundlagen der Rechnerkommunikation**

von Bernd Schürmann

### **Netze – Protokolle – Spezifikationen**

von Alfred Olbrich

### **Grundkurs Verteilte Systeme**

von Günther Bengel

### **Grundkurs**

### **Mobile Kommunikationssysteme**

von Martin Sauter

### **Grundkurs Wirtschaftsinformatik**

von Dietmar Abts und Wilhelm Müller

### **Grundkurs Theoretische Informatik**

von Gottfried Vossen und Kurt-Ulrich Witt

### **Anwendungsorientierte Wirtschaftsinformatik**

von Paul Alpar, Heinz Lothar Grob, Peter Weimann

und Robert Winter

### **Business Intelligence – Grundlagen und praktische Anwendungen**

von Hans-Georg Kemper, Walid Mehanna und Carsten Unger

### **Grundkurs**

### **Geschäftsprozess-Management**

von Andreas Gadatsch

### **Prozessmodellierung mit ARIS®**

von Heinrich Seidlmeier

### **ITIL kompakt und verständlich**

von Alfred Olbrich

### **BWL kompakt und verständlich**

von Notger Carl, Rudolf Fiedler, William Jórasz und Manfred Kiesel

### **Masterkurs IT-Controlling**

von Andreas Gadatsch und Elmar Mayer

### **Masterkurs Computergrafik und Bildverarbeitung**

von Alfred Nischwitz und Peter Haberäcker

### **Grundkurs Mediengestaltung**

von David Starmann

### **Grundkurs Web-Programmierung**

von Günter Pomaska

### **Web-Programmierung**

von Oral Avcı, Ralph Trittman und Werner Mellis

### **Grundkurs MySQL und PHP**

von Martin Pollakowski

### **Grundkurs SAP R/3®**

von André Maassen und Markus Schoenen

### **SAP®-gestütztes Rechnungswesen**

von Andreas Gadatsch und Detlev Frick

### **Kostenträgerrechnung mit SAP R/3®**

von Franz Klenger und Ellen Falk-Kalms

### **Masterkurs Kostenstellenrechnung mit SAP®**

von Franz Klenger und Ellen Falk-Kalms

### **Controlling mit SAP®**

von Gunther Friedl, Christian Hilz

und Burkhard Pedell

### **Logistikprozesse mit SAP R/3®**

von Jochen Benz und Markus Höflinger

### **IT-Projekte strukturiert realisieren**

von Ralph Brugger

### **Programmieren lernen mit Java**

von Erwin Merker und Roman Merker

Erwin Merker  
Roman Merker

# **Programmieren lernen mit Java**

**Leicht verständlich –  
Griffige Beispiele –  
Ausführbare Programme**

Mit 52 Abbildungen



Bibliografische Information Der Deutschen Bibliothek

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

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

Höchste inhaltliche und technische Qualität unserer Produkte ist unser Ziel. Bei der Produktion und Auslieferung unserer Bücher wollen wir die Umwelt schonen: Dieses Buch ist auf säurefreiem und chlorfrei gebleichtem Papier gedruckt. Die Einschweißfolie besteht aus Polyäthylen und damit aus organischen Grundstoffen, die weder bei der Herstellung noch bei der Verbrennung Schadstoffe freisetzen.

1. Auflage Februar 2006

Alle Rechte vorbehalten

© Friedr. Vieweg & Sohn Verlag | GWV Fachverlage GmbH, Wiesbaden 2006

Lektorat: Dr. Reinald Klockenbusch / Andrea Broßler

Der Vieweg Verlag ist ein Unternehmen von Springer Science+Business Media.  
[www.vieweg.de](http://www.vieweg.de)



Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlags unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Konzeption und Layout des Umschlags: Ulrike Weigel, [www.CorporateDesignGroup.de](http://www.CorporateDesignGroup.de)

Umschlagbild: Nina Faber de.sign, Wiesbaden

Druck und buchbinderische Verarbeitung: Těšínská tiskárna, a.s.; Tschechische Republik

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier.

Printed in the Czech Republic

ISBN 3-8348-0068-6

---

## Vorwort

---

Dies ist ein Arbeitsbuch, kein Nachschlagewerk. Das Ziel dieses Buches ist es, Schülern und Studenten eine grundlegende Einführung in das Programmieren von Computern zu geben. Als Programmiersprache haben wir Java gewählt, weil Java eine moderne, elegante und leicht zu lernende Sprache ist. Es werden keine Programmierkenntnisse vorausgesetzt, auch nicht in einer anderen Sprache.

Der Leser benötigt die kostenlose Entwicklungsumgebung der Firma Sun und einen beliebigen Texteditor, um alle Beispiele in diesem Buch selbst ausführen zu können. Hinweise zum Bezug und zur Installation dieser Produkte geben wir im ersten Kapitel des Buches.

Alle Datenverarbeitung mit Computern basiert darauf, dass sich diese Daten im Arbeitsspeicher befinden. Also beginnt das Buch damit, die Datenrepräsentation im Arbeitsspeicher zu erläutern. Danach werden die Basiskonzepte der Java-Sprache beschrieben. Wir werden die Datentypen, Kontrollstrukturen, Algorithmen und Ausdrücke besprechen, so wie sie auch in den meisten anderen Programmiersprachen heute zu finden sind.

Java ist eine objektorientierte Sprache. Deshalb ist ein Schwerpunkt des Buches, das Denken in Klassen und Instanzen zu trainieren. In den Kapiteln 10 bis 16 haben wir beschrieben, wie mitgelieferte Klassen benutzt werden, wie eigene Klassen und Methoden erstellt werden und wie mit den Objekten gearbeitet wird. Und immer wieder gibt es Hinweise darauf, worauf zu achten ist, dass "gute" Programme entstehen, die verständlich sind und die Wartung erleichtern.

Das Buch enthält keine ausführliche Beschreibung aller einzelnen Klassen der Standard-Bibliothek. Die findet der Leser in der API-Dokumentation von Sun. Wichtiger war uns, dass die Denkweisen und allgemeingültigen Konzepte einer objektorientierten Programmiersprache erlernt werden.

Häufig stellt sich ein Verständnis erst dann ein, wenn Fehler gemacht und korrigiert wurden. [Deswegen empfehlen wir dringend, die Beispiele dieses Buches selbst zu editieren und zu testen.](#) Im zweiten Schritt sollten die vorgestellten Programme modifiziert und erneut getestet werden. Hinweise dazu geben wir an vielen Stellen in diesem Buch.

Steinfurt, Januar 2006

Roman Merker, Erwin Merker

**[www.merkeredv.de](http://www.merkeredv.de)**

---

## Hinweise für den Lehrenden

Dies ist ein Java-Lehrbuch ohne die Beschreibung von Applets. Um einfache Java-Applets zu schreiben (oder auch nur zu verstehen), sind Kenntnisse in den Internet-Standards wie HTML und HTTP notwendig. Außerdem müssen die Techniken der Objektorientierung, insbesondere die Vererbungsmechanismen erlernt werden. Darüber hinaus muss man mit der Programmierung von grafischen Oberflächen vertraut sein. Selbstverständlich ist auch ein tiefes Verständnis für die Nutzung der umfangreichen Klassenbibliotheken notwendig (die Standard-Edition enthält die verwirrende Vielfalt von fast 5000 Klassen, und jede Klasse kann viele Methoden enthalten). Und wenn dann noch das Programmieren unter dem Einsatz von komplexen Entwicklungsumgebungen wie z.B. Eclipse erfolgen soll, ist ein Anfänger hoffnungslos überfordert.

Vielleicht ist das der Grund für das häufig gehörte Vorurteil, Java sei ungeeignet als Sprache zum Erlernen der Programmierung. Dieses Buch zeigt, dass Java eine sehr einfache, leicht zu lernende und logisch aufgebaute Programmiersprache ist. Java besteht aus gerade einmal 50 Schlüsselwörtern und ist hervorragend geeignet, um das Programmieren von guten, übersichtlichen und gut strukturierten Programmen zu lehren und zu lernen. Das ist das Ziel dieses Buches.

Damit dies gelingt, sind allerdings Beschränkungen notwendig. Das Buch

- verzichtet auf komplexe Algorithmen, um die einführenden Beispiele möglichst einfach zu halten,
- kann nicht die gesamte Fülle der mitgelieferten API-Klassen und Interfaces erläutern, sondern beschränkt sich auf die Basiskonzepte der Java-Sprache und Objektorientierung,
- beschreibt nicht weiterführende Techniken wie Threads, Polymorphismus, grafische Oberflächen oder das Arbeiten mit generischen Klassen.

### Was kann der Leser dafür erwarten?

Das Buch ist ein Lehrbuch für den Neueinsteiger in die Programmierung von EDV-Anlagen. Es werden keine Vorkenntnisse in einer anderen Programmiersprache vorausgesetzt. Erwartet werden aber Grundkenntnisse im Arbeiten mit dem Computer und die Bereitschaft zum Üben und Experimentieren. Der Lernende wird interaktiv geführt und angeleitet zum selbstständigen Ausprobieren und Bewerten.

Programmieren ist eine faszinierende Tätigkeit. Über Erfolg oder Nicht-Gelingen wird so unmittelbar entschieden wie bei kaum einer anderen Arbeit. Dabei wird von Anfängern oft übersehen, dass das Erstellen von "guten" Programmen eine Entwicklungstätigkeit ist, die neben Disziplin umfangreiche Sachkenntnis und viel Erfahrung verlangt. Die theoretischen Voraussetzungen werden durch dieses Buch geschaffen.

---

# Inhaltsverzeichnis

---

Hinweise für den Lehrenden .....	VI
----------------------------------	----

1	Einleitung: Die Arbeit vorbereiten .....	1
---	--	---

1.1	Das JDK (Java Development Kit).....	1
1.2	Die JDK-Dokumentation.....	3
1.3	Der Java-Editor.....	4
1.4.	Das erste Java-Programm erstellen, umwandeln und ausführen.....	6

2	Java im Überblick: Erste Schritte machen.....	11
---	---	----

2.1	Was ist ein Java-Programm?.....	12
2.2	Elemente eines Java-Programms .....	16
2.3	Schlüsselwörter, Syntax und Semantik .....	19
2.4	Bezeichner (identifier) und Namensregeln.....	20
2.5	Einige Hinweise zu möglichen Fehlern .....	22
2.6	Empfehlungen für lesbaren Quelltext .....	25

3	Informationen maschinell darstellen .....	27
---	---	----

3.1	Zahlensysteme und der Binärcode .....	27
3.2	Informationsformen .....	29
3.3	ASCII-Code .....	31
3.4	Erweiterungen des ASCII-Code .....	34
3.5	Rein binäre Codierung von Zahlen .....	35
3.6	Unicode .....	36

4	Klassen und andere Typen beschreiben ("declaration")	43
---	--	----

4.1	Deklarationsanweisung .....	44
4.2	Was ist der Datentyp?.....	45
4.3	Referenztypen .....	48
4.4	Spezialfall: Primitive Datentypen .....	50

---

5	RAM verwalten: Variable und Objekte erzeugen .....	63
5.1	Was sind Variablen? .....	63
5.2	Primitive Variablen .....	64
5.3	Referenzvariablen .....	68
5.4	Konstanten .....	74
5.5	Literale .....	76
5.6	Zusammenfassung .....	85
6	Eingabe und Ausgabe durchführen ("i/o-operation ") ....	87
6.1	Stream-Konzept .....	87
6.2	Standard-Eingabe .....	94
6.3	Standard-Ausgabe .....	98
6.4	Dateiverarbeitung .....	102
7	Ausdrücke verstehen ("expression") .....	113
7.1	Operanden und Operatoren .....	114
7.2	Arithmetische Operatoren .....	116
7.3	Vergleichsoperatoren .....	131
7.4	Logische Operatoren .....	134
7.5	Bitweise Operatoren .....	142
7.6	Auswertungs-Reihenfolge (Präzedenzregeln) .....	145
8	Anweisungen kodieren ("statements") .....	147
8.1	Einfache und zusammengesetzte Anweisungen .....	148
8.2	Wertezuweisung .....	150
8.3	Steueranweisungen .....	154
8.4	Verzweigungen (Selektion, Auswahl) .....	155
8.5	Schleifen (Iteration, Wiederholung, Loop) .....	167
8.6	Sprung-Anweisungen (break, continue) .....	179
8.7	Lösungsmuster für Schleifen .....	184
8.8	Stilfragen: Konventionen zum Programmierstil .....	188



---

9	Softwaresysteme entwickeln (Projekte realisieren) .....	192
9.1	Herausforderungen und Vorgehensweisen .....	193
9.2	Modelle zur Vorgehensweise .....	201
9.3	Prinzipien und Methoden der Anwendungsentwicklung.....	203
9.4	Java als Projektsprache .....	206
9.5	Entwurfssprachen .....	208
9.6	Komplettbeispiel.....	216
10	Methoden erklären, implementieren und benutzen.....	219
10.1	Was sind Methoden?.....	220
10.2	Mitgelieferte Methoden benutzen .....	222
10.3	Methodenaufruf .....	228
10.4	Eigene Methoden erstellen .....	232
10.5	Methodenblock implementieren .....	237
10.6	Parameter übergeben und empfangen.....	240
10.7	Rückgabewert .....	248
10.8	Zusammenfassung .....	251
11	Klassen beschreiben und benutzen.....	252
11.1	Was steht in einer Klassenbeschreibung? .....	252
11.2	Arbeiten mit Instanzen der Klassen .....	255
11.3	Mitgelieferte Klasse benutzen.....	257
11.4	Eigene Klassen erstellen .....	262
11.5	Konstruktoren .....	269
11.6	Vererbung ("inheritance").....	274
11.7	Statische Elemente einer Klasse .....	284
11.8	Weitere Sprachmittel für Referenztypen (interface, enum).....	286
11.9	Zusammenfassung .....	292
12	Module entwerfen, kapseln und dokumentieren .....	294
12.1	Was ist ein Modul?.....	295
12.2	Motivation für Modulbildung .....	296
12.3	Objektorientierte Systementwicklung.....	298

---

12.4	Unified Modeling Language (UML) .....	307
12.5	Pattern und Frameworks.....	309
<b>13</b>	<b>Reihungen benutzen ("arrays").....</b>	<b>313</b>
13.1	Erzeugen von Arrays .....	313
13.2	Initialisieren von Arrays .....	315
13.3	Zugriff auf die Array-Komponenten .....	317
13.4	Objekte in Arrays sammeln .....	321
13.5	Methoden der Class <i>Arrays</i> .....	322
13.6	Mehrdimensionale Arrays .....	323
13.7	Arrays als Parameter und Returnwert bei Methoden .....	325
13.8	Zusammenfassung .....	326
<b>14</b>	<b>Zeichenketten anwenden ("strings").....</b>	<b>329</b>
14.1	Erstellen von String-Objekten.....	329
14.2	Methoden der Class <i>String</i> .....	332
14.3	Methoden der Class <i>StringBuilder</i> .....	338
14.4	Strings als Commandline-Parameter .....	339
14.5	Zerlegen von Text .....	341
14.6	Reguläre Ausdrücke .....	343
14.7	Strings und Unicode.....	347
<b>15</b>	<b>Typumwandlungen verstehen ("casting").....</b>	<b>349</b>
15.1	Erweiternde Konvertierung bei einfachen Typen .....	350
15.2	Einschränkende Konvertierung bei einfachen Typen .....	353
15.3	Verallgemeinernde Konvertierung bei Referenztypen.....	355
15.4	Spezialisierende Konvertierung bei Referenztypen .....	356
15.5	Typ-Umwandlung zwischen einfachen und Referenztypen .....	358
<b>16</b>	<b>Modifier richtig einsetzen ("access control") .....</b>	<b>362</b>
16.1	Lokale Variable und Member-Variable.....	362
16.2	Sichtbarkeit und Gültigkeit von Variablen.....	364
16.3	Welchen Anfangswert haben die Variablen? .....	365

---

16.4	Lebensdauer von Variablen .....	366
16.5	Zugriffsrechte von außerhalb einer Klasse ("access control") .....	370
16.6	Bedeutung der Package-Namen .....	372
16.7	Zugriffsmodifizier <i>private</i> , <i>public</i> , <i>protected</i> .....	375
A	Installationshinweise J2SE SDK 5.0.....	379
B	Meta-Sprachen zur Syntaxbeschreibung.....	384
C	Die ersten 256 Unicode-Zeichen (0000-ffff).....	386
D	Komplettbeispiel einer verteilten Application.....	394
E	Glossar .....	396
	Sachwortverzeichnis.....	412

# 1

## Einleitung: Die Arbeit vorbereiten

---

In diesem Kapitel erfahren Sie,

- was Sie für die Arbeit mit diesem Buch benötigen,
- woher Sie diese kostenlosen Werkzeuge bekommen und
- wie Sie damit umgehen.

Für das Arbeiten mit den Beispielen in diesem Buch werden ein (beliebiger) Text-Editor und das JDK (Java Development Kit) von Sun benötigt. Alle Installationshinweise und Erläuterungen zum Umwandeln und Testen der Programme in diesem Buch beziehen sich zwar auf die Windows-Umgebung, können aber für das Linux-/Unix-Umfeld in ähnlicher Weise übernommen werden.

Es wird dem Anfänger nicht empfohlen, von Beginn an mit einer komfortablen Entwicklungsumgebung wie *Eclipse*, *NetBeans* oder *JBoss* zu arbeiten. Diese komplexen "Integrierten Entwicklungsumgebungen" (IDE für Integrated Development Environment) sind zwar sehr komfortabel, um diese jedoch zu verstehen und ihre Vorteile auszunutzen, sind Kenntnisse im Programmieren und in Java erforderlich.

Als Editor könnten unter MS-Windows zwar die mitgelieferten Programme "*Editor*" oder "*WordPad*" genutzt werden, besser sind aber spezielle Produkte für die Java-Entwicklung. Diese gibt es als kostenlose Freeware-Tools wie z.B. *BlueJ*. Möglich ist natürlich auch der Einsatz von kostenpflichtigen Programmen wie *TextPad*. Wir haben alle Beispiele in diesem Buch mit dem Freeware-Programm "*JOE*" erstellt und ausgeführt. Hinweise zum Bezug und zur Installation dieser Produkte erfolgen im Abschnitt 1.3.

### 1.1 Das JDK (Java Development Kit)

#### Was ist das JDK?

Die klassische Entwicklungsumgebung für Java ist das JDK von Sun. Dieses Software-Paket enthält außer einem Editor alles, was zum Entwickeln und Ausführen von Javaprogrammen notwendig ist, nämlich

- einen **Übersetzer** ("Compiler"), der den Quelltext in den Bytecode übersetzt, und
- einen **Ausführer** ("Java Run Time-Environment JRE"), der als Teil der "Java Virtual Machine" JVM den Bytecode interpretiert und ausführt.

Außerdem enthält die JDK die Implementierung der wichtigsten Standardklassen, die als Bestandteil der Java-Sprache für die Umwandlung und Ausführung auch der ein-

fachsten Programme unbedingt benötigt werden (z.B. für Strings, Input/Output, Collections, Applets, GUI oder Network).

Die Beispiele in diesem Buch basieren auf dem Stand der "Standard-Edition" Version 5 (oder höher). Die offizielle Bezeichnung des aktuellen Software-Pakets ist "Java 2 Plattform Standard-Edition" (J2SE 5.0). Es kann vom Server der Firma SUN bezogen werden.

### Bezug und Installation der JDK

Hier eine Anleitung zum Beschaffen der JDK für Version 5.0:

- Im Browser die Java-URL der Firma SUN eingeben: **<http://java.sun.com>**
- Dort entweder direkt die Seite "J2SE 5.0" unter der Überschrift "Popular Downloads" anwählen oder zunächst die Informationen, die unter dem Link "J2SE (Core/Desktop)" gefunden werden, lesen und danach die Download-Schaltfläche anklicken.
- Auf der dann folgenden Seite werden zur Auswahl drei Produkte angeboten:
  - entweder das komplette JDK inclusive NetBeans IDE
  - oder nur das komplette JDK
  - oder nur die JRE (Run-Time-Umgebung).
- Sie benötigen das komplette JDK, allerdings ohne Netbeans IDE.
- Jetzt ist nur noch zu entscheiden, für welche Plattform die JDK benötigt wird. Für MS-Windows wird dann z.B. der Download der Datei "*jdk-1\_5\_0\_nn-windows-i586-p.exe*" (ca. 50 MB) gestartet.

Weitere detaillierte Hinweise für die **Installation** der SDK unter MS-Windows siehe Anhang A.

### Muss unbedingt die J2SE 5.0 (oder höher) installiert werden?

Ja, denn alle Beispiele in diesem Buch sind mit der Version 5.0 getestet und lauffähig. Einige Programme enthalten Neuerungen, die erst mit der Version 5.0 eingeführt worden sind - ältere Versionen verursachen Umwandlungs- oder auch Laufzeitfehler.

### Können mehrere JDK-Versionen installiert sein?

Ja, auf einer Maschine können mehrere unterschiedliche Versionen der JDK installiert und auch parallel genutzt werden. So können unter einem Betriebssystem z.B. sowohl die ältere Version 1.4 als auch die Version 5 vorhanden sein. Welche dieser Versionen jeweils genutzt wird, wird durch Angaben in einer PATH-Environment-Variablen entschieden. Diese Variablen sind dynamisch änderbar, in MS-Windows für jede DOS-Box individuell (siehe Hinweise zur Installation im Anhang A).

## 1.2 Die JDK-Dokumentation

Auf der Download-Seite von Sun werden nicht nur die JDK-Produkte angeboten, sondern auch die J2SE-Dokumentation, etwa 45 MB. Es wird dringend empfohlen, auch diesen Download durchzuführen.

### Was enthält die JDK-Dokumentation?

Durch den Download bekommen Sie Zugriff auf die komplette Dokumentation der J2SE Plattform Standard Edition. Im Wesentlichen sind dies die beiden folgenden Teile:

**The Java Language Specification:** Eine komplette Beschreibung der Basis-Sprachelemente, also die Definition der grammatikalischen und lexikalischen Struktur, die Beschreibung der primitiven Datentypen und der Statements, die den Kern der Sprache bilden. Diese Beschreibungen sind auch als HTML- oder PDF-Versionen zum Download verfügbar unter

`http://java.sun.com/docs/books/jls/`

**Java 2 Platform API Specification:** Eine ausführliche Beschreibung aller Standard-Klassen, die eingebaut sind und zum Lieferumfang der Standard-Edition gehören. API ist die Abkürzung für "Application Programmer Interface", damit bezeichnet man die Schnittstellen, die dem Javaprogrammierer für das Schreiben seiner Anwendung zur Verfügung stehen, und das sind alle Klassen und Interfaces des "Java 2 Platform-Packages". Eine Fülle von Informationen sind in dieser Dokumentation enthalten: Tutorials, Demos und Beispielpprogramme, Glossary usw. Alle Informationen sind auch online verfügbar unter:

`http://java.sun.com/reference/`

### Bezug und Installation der API-Dokumentation für lokale Nutzung

- Im Browser die Java-URL der Firma SUN eingeben: `http://java.sun.com`
- Im Startbild direkt die Schaltfläche "J2SE 5.0" unter der Überschrift "Popular Downloads" anklicken.
- Danach die Schaltfläche "Download" für die J2SE 5.0 Documentation anklicken, die Lizenzvereinbarungen akzeptieren und mit "continue" den Download-Vorgang starten (vorher noch die englische Version auswählen).
- Es wird die Datei `jdk-1_5-doc.zip` übertragen (Größe ca. 44 MB).
- Die Installation wird gestartet durch das Entzippen dieser Datei.
- Für das Arbeiten mit der Dokumentation wird empfohlen, eine Verknüpfung auf dem Desktop anzulegen. Die Startdatei ist: `c:\install\docs\index.html`
- Hinweise zum richtigen Umgang mit der API-Dokumentation erfolgen später.

## 1.3 Der Java-Editor

Ein Editor ist ein Programm, das es erlaubt, eine Textdatei zu erstellen und zu bearbeiten. Prinzipiell kann jeder einfache Texteditor benutzt werden, um die Java-Quelltexte einzutippen. Ein Textverarbeitungsprogramm wie MS-Word hat weitergehende Aufgaben (Schriftarten, Textformatierungen ...), deswegen ist so ein Programm nicht geeignet. Auch die Text-Editoren, die Teil des Betriebssystems MS-Windows sind, eignen sich nicht so gut (allein schon deswegen nicht, weil sie hartnäckig die Dateieignung selbstständig vergeben). Auch "vi" unter Linux wird nicht empfohlen.

### Welcher Editor sollte genommen werden?

Im Internet gibt es unter dem Suchbegriff "Texteditoren" einige Dutzend kostenlose Programme zum Erstellen und Bearbeiten von ASCII-Text. Für Java-Quelltexte gibt es außerdem eine Reihe von speziellen Editoren, z.B.

- JCreator (von [www.jcreator.com/download.html](http://www.jcreator.com/download.html) (kostenlose Freeware-Version)
- BlueJ (von [www.bluej.org](http://www.bluej.org), kostenlose Freeware-Version)
- JOE (von [www.javaeditor.de](http://www.javaeditor.de); kostenlose Freeware-Version)

Wir werden in diesem Buch mit dem "Java Oriented Editor JOE" arbeiten. Deswegen dazu einige Hinweise.

### Bezug und Installation des Java-Editors "JOE"

Der Editor steht zum Download unter folgender Webadresse zur Verfügung:

<http://www.javaeditor.de>

Durch den Download erhält man die Datei "joe.zip" (Größe: ca. 1.3 MB). Zur Installation muss diese Datei entzippt werden. Alle Fenster des Installationsprogrammes sind unverändert zu akzeptieren. Auch im folgenden Fenster ist normalerweise mit "JA" zu antworten.



Abb. 1.1: Verknüpfung des JOE-Editors mit der installierten Java-Version

Dadurch wird erreicht, dass die Einstellungen (Optionen) im JOE-Programm automatisch ergänzt werden um die Suchpfade zu den *bin*-Dateien, damit der Interpreter und der Compiler gefunden werden. Auch die Verknüpfung zur Dokumentation wird hergestellt, damit aus dem Editor heraus über den Menüpunkt *Hilfe (?)* direkt dahin verzweigt werden kann.

Gestartet wird die Entwicklungsumgebung mit:

```
START|PROGRAMME|Fantastic-Bits|JOE|JOE
```

Empfehlung: Legen Sie eine Verknüpfung auf dem Desktop an.

### Prüfen, ob JOE korrekt installiert ist

Über den Menüpunkt *Optionen|Einstellungen* sollte überprüft werden, ob die Pfade zum Compiler und zum Interpreter korrekt eingetragen sind. Eventuell muss dies manuell nachgeholt werden. Wahlweise kann hier auch der Zielpfad für die Ausgabe des Compilers eingetragen werden.

### Anlegen eines neuen Ordners für die Übungsdateien

Für das Arbeiten mit den Beispielen in diesem Buch ist es hilfreich, wenn ein separates Verzeichnis angelegt wird. Der Name ist frei wählbar, z.B. "*JavaProg*" oder der Name des Nutzers. Die Beispiele in diesem Buch stehen im Ordner *e:\merker*.

Danach testen wir, ob alle Voraussetzungen für einen erfolgreichen Start erfüllt sind.

### Übung: Prüfen, ob Java korrekt installiert ist

- Öffnen Sie ein Fenster für die Eingabeaufforderung (Linux-Shell bzw. unter MS-Windows eine DOS-Eingabebox)
- In diesem Console-Fenster verzweigen Sie in den soeben neu angelegten Ordner (z.B. *cd e:\merker*)
- Dort bitte eingeben: "*java -version*". Dadurch wird der Interpreter aufgerufen und die Java-Version angezeigt. Wenn mit J2SE 5.0 gearbeitet wird, sieht die Ausgabe wie folgt aus:

```
E:\merker>java -version
java version "1.5.0_01"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_01-b08)
Java HotSpot(TM) Client VM (build 1.5.0_01-b08, mixed mode, sharing).
```

- Geben Sie bitte nur "*java*" ein. Dadurch rufen Sie den Interpreter (die JVM) auf. Weil weitere Informationen für den Interpreter fehlen, werden Hilfe-Informationen ausgegeben. Die Syntax des Befehls wird beschrieben und alle möglichen Optionen (z.B. "*-version*") aufgelistet.



## 1.4. Das erste Java-Programm erstellen, umwandeln und ausführen

Zur Einführung werden wir das simpelste Java-Programm, das es gibt, erstellen und ausführen. Das Programm hat nur wenige Zeilen. Es geht aber nicht darum, ein Verständnis für dieses Programm zu bekommen, sondern es handelt sich lediglich um einen technischen Test.

### Programm Test01: Das erste Java-Programm

```
public class Test01 {  
    public static void main(String[] args) {  
        System.out.println("Erstes Programm");  
    }  
}
```

Sind die Voraussetzungen, die auf den vorherigen Seiten beschrieben worden sind, erfüllt, gibt es zwei Möglichkeiten, wie Sie arbeiten können:

- Entweder verzichten Sie ganz auf eine IDE (also auch z.B. auf JOE) und starten die Programme (Editor, Compiler, Interpreter usw.) direkt durch Textkommandos in einem Commandprompt. Dies kann die DOS-Eingabeaufforderung in MS-Windows oder eine Unix-Shell sein. Dadurch verzichtet man zwar auf Komfort, denn es entfallen mögliche Automatisierungen, aber das Verständnis für die Abläufe wird erleichtert, weil alle Möglichkeiten ungefiltert zur Verfügung stehen.
- Oder Sie nutzen die Entwicklungsumgebung, also z.B. JOE, nicht nur zum Editieren, sondern auch, um menügesteuert umzuwandeln und zu testen.

Wir werden beide Varianten demonstrieren. Für den absoluten Neueinsteiger empfehlen wir dringend, zumindest einmal auch die Commandline-Variante auszuprobieren. Das ist gut für das Verständnis der Vorgänge Editieren, Compilieren und Testen.

### Was bedeutet "umwandeln" (compilieren)?

Der Programmierer erstellt einen Quelltext ("Sourceprogramm") mit einem Editor. Dieses Quellenprogramm muss in ein maschinenlesbares Format, den Bytecode, umgewandelt werden. Dafür benötigt man ein spezielles Programm, den Compiler. Der Compiler ist Teil des JDK. Er liest das Quellenprogramm und überprüft es auf formale Korrektheit. Wenn der Quelltext ohne Syntaxfehler ist, erzeugt der Compiler daraus eine zusätzliche neue Datei. Diese hat die Dateierweiterung *class* und enthält einen Zwischencode, den so genannten Bytecode.

Es gibt Programmiersprachen (z.B. C++), bei denen das Source-Programm in ein Maschinenprogramm umgewandelt wird. Bei dieser Umwandlung wird ein ausführbares Programm für eine genau definierte Plattform (welcher Prozessor, welches Betriebssystem) erzeugt - bei einem Wechsel der Plattform muss neu kompiliert werden.

### 1.4.1 Version 1: Arbeiten mit der Eingabeaufforderung

#### Übung 1: Editieren und Umwandeln des Programms *Test01.java*

- Erstellen Sie den Quelltext: Tippen Sie die fünf Zeilen des Programms mit einem beliebigen Texteditor ein. Dabei ist unbedingt auch auf die Groß-/Kleinschreibung zu achten.
- Sichern Sie die Quelldatei: Der Quelltext muss unter dem Namen *"Test01.java"* gesichert werden. Dieser Dateiname ist Pflicht, denn er muss mit dem internen (Klassen-)Namen in Zeile 1 übereinstimmen.
- Wandeln Sie die Quelldatei um: Dazu muss in den Ordner, der die Quelldatei enthält, verzweigt werden. Dort eventuell mit *"set pathb"* prüfen, ob das Compilerprogramm auch gefunden wird. Dann wird die Umwandlung gestartet mit: *"javac Test01.java"*.

Der Java-Compiler wird aufgerufen durch: *javac Test01.java*.

#### Was ist das Ergebnis der Umwandlung?

Die fehlerfreie Umwandlung des Programms (erkennbar daran, dass keine explizite Fehlermeldung ausgegeben wird) führt dazu, dass eine neue Datei erstellt wird. Diese hat die Dateiendung *.class*, und sie enthält den ausführbaren Code (bytecode). Sollte eine Fehlermeldung kommen, so versuchen Sie, diese zu interpretieren (Hinweise dazu siehe Abschnitt 1.4.3). Zumindest die Zeilennummer der fehlerhaften Zeile kann eine Hilfe sein.

Korrigieren Sie eventuelle Tippfehler und vergessen Sie nicht, danach die Umwandlung erneut zu starten. Erst wenn die Compilierung fehlerfrei möglich ist, wird eine neue Datei mit der Endung *.class* im Arbeitsordner erzeugt. Das ist die Datei für die Ausführung.

#### Wie wird die Ausführung gestartet?

Für die Ausführung benötigt man die Run-Time-Umgebung von Java (Laufzeitumgebung). Sie enthält den Java-Interpreter, das ist ein Programm, das den Bytecode interpretiert (d.h. Zeile für Zeile in den Maschinencode umwandelt) und ausführt.

Der Java-Interpreter wird aufgerufen durch: *java Test01*

Durch den Aufruf des Programms *java* wird die "Java Virtuelle Maschine" mit dem Interpreter gestartet. Diese lädt die Class-Datei mit dem Namen *Test01* und führt sie aus.

#### Übung 2: Ausführen (Testen des Programms)

Die Ausführung des umgewandelten Programms wird wie folgt gestartet:

*java Test01*

Achtung: Die Dateiendung *.class* darf für *Test01* nicht angegeben werden. Die Datei *java.exe* startet eine virtuelle Java-Maschine (JVM) und beendet diese, sobald das Java-Programm *Test01.class* beendet ist.

Hier ist ein Protokoll der durchgeführten Arbeiten unter MS-Windows:



```
E:\merker>set path
Path=D:\WINNT\system32;D:\WINNT;D:\WINNT\System32\Wbem;E:\Programme\Java\jdk1.5.
0_01\bin
PATHEXT=.COM;.EXE;.BAT;.CMD;.UBS;.UBE;.JS;.JSE;.WSF;.WSH

E:\merker>javac Test01.java

E:\merker>java Test01
Erstes Programm

E:\merker>
```

Abb.1.2: Umwandeln und Testen per Commandline in einer DOS-Sitzung

### 1.4.2 Version 2: Arbeiten mit JOE

Und jetzt die gleiche Übung, diesmal mit Hilfe von JOE. Nach dem Start von JOE ist eine neue Datei anzulegen und wie folgt zu editieren:

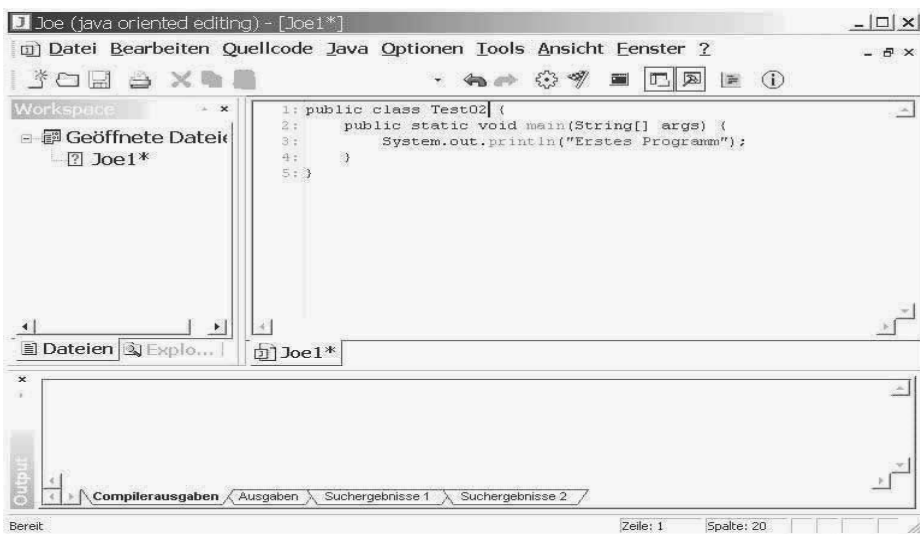


Abb. 1.3: Programm *Test02.java* editieren mit JOE

Anschließend erfolgt über das Datei-Menü die Sicherung des Quelltextes. Achtung: der Dateiname ist jetzt *Test02* (die Dateiergung *.java* wird automatisch vergeben).

Die Umwandlung und Ausführung des Programms kann ebenfalls menügesteuert erfolgen.

- **Umwandeln** durch: JAVA | COMPILIEREN
- **Ausführen** durch: JAVA | STARTEN.

Alternativ können Sie auch über den Menü-Eintrag

- JAVA | MSDOS Eingabeaufforderung

in eine Commandbox verzweigen und von dort aus den Compiler und den Interpreter per Command aufrufen.

### 1.4.3 Hinweise auf mögliche Probleme beim Compilieren und Ausführen

- Groß- und Kleinschreibung ist wichtig. Beim Editieren des Quelltextes und beim Aufrufen der Programmnamen für die Umwandlung und Ausführung ist darauf zu achten, dass Java sehr wohl unterscheidet zwischen Klein- und Großbuchstaben.
- Achten Sie besonders auf die Klammernbildung: öffnende und schließende Klammern immer paarweise z.B. { } oder ( ).
- Wenn der Compiler oder der Interpreter nicht gefunden werden, ist die PATH-Environment-Variable zu überprüfen. In dieser Suchpfadliste müssen die Hinweise auf die *bin*-Dateien des Installationsordners stehen, z.B.  
Path=D:\WINNT\...E:\Programme\Java\jdk1.5.0\_01\bin  
Hinweise, wie dieser Fehler korrigiert wird, finden Sie im Anhang A.
- Wenn die Quelltextdatei nicht gefunden wird, überprüfen Sie bitte, ob der Compileraufruf aus dem richtigen Ordner heraus erfolgt ist (nämlich aus dem Arbeitsordner, in dem sich das Quellenprogramm befindet).
- Manche Editoren ergänzen den Dateinamen mit dem Quelltext hartnäckig um die Dateiergung *.txt*. Dann muss z.B. mit Hilfe des Windows-Explorers oder mit dem DOS-Command *rename* der gewünschte Dateiname hergestellt werden.
- Wenn bei der Ausführung die Class-Datei nicht gestartet werden kann, überprüfen Sie, ob nach dem Aufruf des Interpreters durch "*java*" der korrekte Dateiname für das ausführbare Javaprogramm angegeben ist (nämlich mit dem richtigen Dateinamen, allerdings ohne die Endung *.class* ), und ob dieser Dateiname exakt übereinstimmt mit dem Namen der Klasse im Quelltext (Groß-/Kleinschreibung?).
- Wenn eine Programmänderung durchgeführt worden ist, muss unbedingt neu umgewandelt werden. Andernfalls wird die Änderung nicht wirksam.

### Übung zum Programm *Test01.java*

Ein Java-Programm kann Kommentarzeilen enthalten. Entweder beginnt der Kommentar mit zwei Schrägstrichen `//`, dann gilt dies nur für den Rest einer Zeile, oder er beginnt mit `/*`, dann kann sich der Kommentar auch über mehrere Zeilen erstrecken und wird mit `*/` abgeschlossen.

Ergänzen Sie das Programm um die folgende Kommentarzeile:

```
"Dies ist das erste Testprogramm"
```

wandeln Sie das Programm neu um und testen Sie es erneut. Versuchen Sie die zwei Möglichkeiten, wie ein Kommentar eingefügt werden kann. Überprüfen Sie die Größe der erzeugten Class-Datei, um zu klären, ob der Kommentar Bestandteil der Byte-Datei ist.

Dadurch bekommen Sie die Antwort auf die Frage: Verändert sich die Größe der ausführbaren Datei, wenn man sehr viel Dokumentartext im Source-Programm verwendet?

### Zusammenfassung

Die Voraussetzungen für ein erfolgreiches Arbeiten mit Java haben Sie geschafft. Sie haben auch bereits die drei wichtigsten Schritte zur Programmerstellung ausgeführt:

<b>Editieren:</b>	Eingabe des Quelltexts mit Hilfe eines Editors
<b>Compilieren:</b>	Erzeugen des Bytecodes mit Hilfe eines Compiler-Programms, durch: <code>javac dateiname.java</code>
<b>Interpretieren:</b>	Testen des Programms durch Ausführung in einer JVM durch: <code>java dateiname</code>

Für diese Arbeiten ist die Installation einer Java-Entwicklungsumgebung (z.B. JDK von Sun) erforderlich.

Und für das Umwandeln und Ausführen ist es notwendig, dass die entsprechenden Systemprogramme (das Umwandlungsprogramm ***javac*** und das Interpreterprogramm ***java***) im Suchpfad des Betriebssystems gefunden werden. Dazu wurde die *path*-Environment-Variable angepasst und um das *bin*-Verzeichnis der JDK ergänzt.

Für das Editieren der Quelltexte eines Java-Programms ist ein spezieller Java-Texteditor wie z.B. JOE hilfreich. Dann können einige Arbeiten automatisiert und menügesteuert ausgeführt werden, und die Anzeige des Quelltexts wird übersichtlich gestaltet, z.B. durch verschiedenfarbige Textbausteine, um Schlüsselwörter, Variable und Befehle optisch kenntlich zu machen.

---

# 2

## Java im Überblick: Erste Schritte machen

---

In diesem Kapitel erhalten Sie Antworten auf folgende Fragen:

- Wie ist ein einfaches Javaprogramm aufgebaut?
- Was versteht man unter Syntax und Semantik einer Programmiersprache?
- Was sind die wichtigsten Bestandteile eines ausführbaren Javaprogramms (einer "Application")?
- Welche besondere Bedeutung hat die *main*-Methode?
- Was sind die Unterschiede zwischen Schlüsselwörtern und Programmiererwörtern?
- Welche Empfehlungen gibt es für die Namensvergabe und für die äußere Form eines Javaprogramms?

Dieses Kapitel beschreibt also die Vorschriften und Empfehlungen für den Aufbau eines Javaprogramms. Wie bei jeder Programmiersprache, gibt es auch für Java lexikalische und syntaktische Regeln, die vom Programmierer eingehalten und vom Compiler überprüft werden müssen.

Der Compiler erstellt nur dann ein ausführbares Programm (also den Bytecode), wenn der Quelltext keine formalen Fehler enthält. Jeder Verstoß gegen die Syntaxregeln der Java-Sprache wird vom Compiler als Fehlermeldung dokumentiert, und es gehört einige Übung und Erfahrung dazu, diese Fehlermeldung zu interpretieren und die Korrektur durchzuführen.

Wir empfehlen noch einmal dringend, alle Beispiele in diesem Buch selbst zu editieren. In den meisten Fällen sind es nur wenige Zeilen, die darüber hinaus auch kopiert und dann modifiziert werden können. Aber nur so können Sie programmieren lernen, nicht nur durch theoretisches Nachvollziehen der Beispiele.

Neben den Syntaxvorschriften gibt es Konventionen, die eingehalten werden *sollten*, damit "gute" Programme entstehen (was gute Programme sind, müssen wir noch klären). Einige dieser Empfehlungen werden wir bereits in diesem Kapitel vorstellen.

Die Java-Programmiersprache ist objektorientiert. Jeder Programmcode ist organisiert in Klassen. Deshalb werden Sie bereits in diesem Kapitel die wichtigen Begriffe "Klassen" und "Methoden" aus der objektorientierten Programmierung kennen lernen. Sie werden Klassen erstellen und benutzen.

## 2.1 Was ist ein Java-Programm?

Generell kann ein Programm definiert werden als Vorschrift an einen Prozessor, eine bestimmte Arbeit auszuführen. Es wird in einer formalisierten Sprache geschrieben. Dazu muss dem Computer mitgeteilt werden

- womit
- was
- in welcher Reihenfolge

getan werden soll.

Jedes EDV-Programm muss definieren, **womit** es arbeitet - das sind in Java die Datenbeschreibungen (Deklarationen der Variablen). Außerdem enthält ein Programm die eigentlichen Befehle, die dem Prozessor vorschreiben, **was** zu tun ist - das sind in Java die Methoden. Und zusätzlich kann der Programmierer festlegen, **in welcher Reihenfolge** die Befehle ausgeführt werden sollen - dafür gibt es Steuerbefehle.

In Java ist ein ausführbares Programm in einer Klasse beschrieben. Der Quelltext für ein vollständiges Programm hat mindestens folgende Struktur:

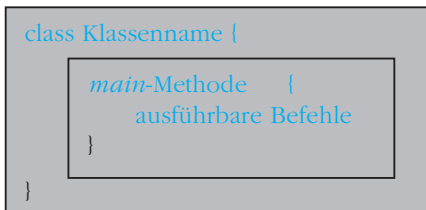


Abb. 2.1: Minimale Java-Klasse (selbstständig ausführbar)

Alle Variablen und Methoden einer Klasse werden innerhalb der geschweiften Klammern platziert. Im Minimum hat eine ausführbare Klasse die Methode *main*. Die Klasse kann auch umfangreicher sein - sie kann beliebig viele Methoden enthalten. Außerdem kann sie Datenbeschreibungen innerhalb und außerhalb von Methoden haben.

### Programm01: Das erste ausführbare Javaprogramm (mit einer Methode)

```
public class Einfuehrung01 {  
    public static void main(String[] args) {  
        // Datenbeschreibung      = Womit?  
        // Methoden                = Was?  
        // Steueranweisungen       = In welcher Reihenfolge?  
    }  
}
```

### 2.1.1 Anatomie des ersten Java-Programms

In der ersten Zeile wird die Klasse definiert. Sie bekommt den Namen *Einfuehrung01*. Alles, was danach folgt, wird als Klassenrumpf bezeichnet und ist in geschweiften Klammern eingefasst. Die Klasse endet mit der letzten geschweiften Klammer.

```
public class Einfuehrung01 {  
    ...  
}
```

In der zweiten Zeile beginnt die Beschreibung der Methode *main*. Jede Java-Application muss genau eine *main-Methode* enthalten - und diese muss mit einer Zeile, die genau diesen Aufbau hat, beginnen:

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

Eine Methode besteht aus dem **Methodenkopf** und dem **Methodenrumpf**. Die erste Zeile wird als Methodenkopf bezeichnet. Die genaue Bedeutung der einzelnen "Token" im Kopf wird später noch gründlich erläutert.

An dieser Stelle ist lediglich wichtig, dass alles, was zu dieser Methode gehört, als Methodenrumpf (oder Methodenblock) bezeichnet wird und wiederum in geschweiften Klammern eingefasst ist. Dort stehen die Anweisungen für die Ausführung durch den Prozessor. Die Methode endet in der vorletzten Zeile dieses Programms.

Innerhalb der Methode *main* stehen drei **Kommentarzeilen**. Kommentare haben für die Ausführung eines Programms keinerlei Bedeutung; sie sollten vom Programmierer geschrieben werden, um (für sich selbst oder für Kollegen) Programmteile näher zu erläutern. Kommentare können auch über mehrere Zeilen gehen, dann beginnen sie mit */\** und enden mit *\*/*. Der Compiler ignoriert diesen Quelltext.

#### Übungen zum Programm *Einfuehrung01*

**Übung 1:** Editieren Sie das o.a. Programm. Speichern Sie es unter dem Namen "*Einfuehrung01.java*" ab. Wandeln Sie es um und starten Sie die Ausführung. Aber erwarten Sie nicht zuviel: dieses Programm macht - gar nichts. Aber für den Anfang ist es ein schöner Erfolg, wenn es fehlerfrei compiliert und getestet werden kann.

Eine Quelltextdatei kann eine oder mehrere Klassen enthalten.

**Übung 2:** Fügen Sie der Quelltextdatei eine zweite Klasse hinzu. Der Name der neuen Klasse soll sein: "*Einfuehrung01a*". Diese Klasse soll leer sein (ohne eine Methode). Wandeln Sie die Quelltextdatei um. Überprüfen Sie, welche neuen Class-Dateien durch eine fehlerfreie Compilierung erstellt worden sind.



**Übung 3:** Ändern Sie die Syntax der Kommentarzeilen. Benutzen Sie nicht die `//` pro Zeile, sondern beginnen Sie den mehrzeiligen Kommentar mit `/*` und beenden Sie ihn mit `*/`.

### Lösungsvorschlag:

```
public class Einfuehrung01 {
    public static void main(String[] args) {
        /*
            ...
        */
    }
}
class Einfuehrung01a {}
```

Achtung: Der zusätzlichen Klasse darf **nicht** das Wort *public* vorangestellt sein.

### Was ist eine Umwandlungseinheit ("compilation unit")?

Eine Quelltextdatei nennt man auch Umwandlungseinheit. Sie kann mehrere Klassen enthalten. Aus formalen Gründen ist strikt zu unterscheiden, ob die Quelltextdatei auch ein selbstständig ausführbares Java-Programm enthält oder ob sie lediglich aus Klassenbeschreibungen besteht, die nicht selbstständig lauffähig sind, sondern nur von anderen Programmen aufgerufen werden können. **Für eine Umwandlungseinheit gelten folgende Regeln:**

- Innerhalb einer Quelltextdatei darf nur **eine** Klasse die *main*-Methode haben, und dies ist auch die einzige Klasse, die dann das Wort *public* haben darf.
- Der Name der Quelltextdatei **muss** identisch sein mit dem Namen der ausführbaren Klasse, also mit dem Namen der Klasse, die die Methode *main* enthält.
- Durch die Umwandlung wird aus jeder Klasse in einer Umwandlungseinheit eine eigene *class*-Datei. Diese enthalten den Bytecode für die Ausführung.

Nach der Umwandlung von *Einfuehrung01* haben Sie also in Ihrem Arbeitsverzeichnis zwei zusätzliche Dateien, jeweils mit der Dateiendung *class*. Aber nur das Programm *Einfuehrung01* ist ein ausführbares Programm, denn nur dies enthält die Methode *main*. Wie Sie bereits gesehen haben, wird ein selbstständig lauffähiges Java-Programm gestartet, indem der Interpreter mit *java.exe* aufgerufen wird und dabei der Name der ausführbaren Klasse als Parameter mitgegeben wird.

### Übung zum Programm *Einfuehrung01a*

Versuchen Sie das Programm *Einfuehrung01a* zu starten. Es sollte folgende Fehlermeldung kommen: "NoSuchMethodError: main" - und das ist der eindeutige Hinweis darauf, dass eine Klasse ohne *main*-Methode nicht selbstständig ausführbar ist.

### 2.1.2 Was sind Java-Applicationen?

Die Beispiele in diesem Buch sind überwiegend **ausführbare Programme**, d.h. sie sind Klassen, die eine *main*-Methode enthalten. Diese werden bezeichnet als Java-Application. Sie sind im Gegensatz zu anderen Java-Klassen völlig autonom und können per Betriebssystembefehl ("*java progname*") gestartet werden. Java-Programme benötigen für die Ausführung eine spezielle Laufzeitumgebung, die virtuelle Maschine (JVM). Diese wird durch Aufruf von "*java*" gestartet - und sie sorgt dann dafür, dass die Datei mit dem Bytecode in den Arbeitsspeicher geladen wird. Die Programmausführung beginnt mit dem ersten Befehl in der *main*-Methode.

Aus Betriebssystemsicht wird durch "*java*" ein neuer *Prozess* gestartet - und zwar zunächst die JVM, die dann die Klasse startet. Es gibt andere Java-Programmtypen, die **nicht selbstständig ausführbar** sind. Dazu gehören Klassen, die keine *main*-Methode haben, z.B. Applets, Servlets, EJBs und Webservices. Diese Klassen benötigen spezielle Serverprogramme, in die sie eingebettet werden, damit sie dort (als Thread und nicht als Prozess) ausgeführt werden können. Diese Programmtypen sind nicht Gegenstand dieses Buches.

Und dann gibt es die Klassen, die als Schablone benutzt werden, damit andere Klassen von ihnen Objekte erzeugen können. Davon enthält allein die *Java 2 Standard Edition* mehrere Tausend. Ohne sie kann ein Java-Programm nicht geschrieben werden. Sie enthalten vorgefertigte Lösungen (wiederverwendbaren Code) für eine Gruppe von Objekten.

Wenn ein Programm diesen vorgefertigten Code nutzen will, erzeugt es von der Klasse eine konkrete Instanz (ein Objekt) im Arbeitsspeicher und ruft dann die damit verbundenen Methoden auf. Dies ist der Kern der objektorientierten Programmierung.

Wir halten also fest:

Eine Java-Application ist ein Programm, das eine *main*-Methode enthält. Das Programm wird gestartet durch einen Befehl an das jeweilige Betriebssystem. Dadurch wird eine JVM gestartet - und diese sorgt dafür, dass der Programmablauf mit der ersten Anweisung der Hauptmethode *main* beginnt. Innerhalb der *main*-Methode können beliebig viele andere Klassen benutzt werden.

Die Klassen, die in einer Application referenziert werden, müssen bei der Umwandlung (und natürlich auch später bei der Ausführung) im Zugriff sein. Dafür ist der richtige Einsatz der *classpath*-Variable wichtig (Hinweise siehe Anhang A).

Es gibt auch Klassen, die *nicht* selbstständig ausführbar sind. Diese Klassen haben keine *main*-Methode. Deswegen können sie nur von anderen Klassen genutzt werden (Wiederverwendung von Programmcode). Durch das Zusammenspiel von mehreren Klassen entsteht eine Java-Anwendung ("application").

### 2.1.3 Wie entsteht ein Programm?

Die Vorgehensweise für das Erstellen eines Programms ist immer gleich:

- Der Programmierer muss das Problem verstanden haben ("Problemanalyse"),
- Danach werden die Lösungsmöglichkeiten für das neue Programm(-system) geplant ("Systemplanung").
- Nach der Entscheidung über die weitere Vorgehensweise erfolgt die "Detailorganisation". Diese legt den Aufbau und Ablauf des Programms fest.
- Erst dann beginnt die eigentliche Programmiertätigkeit, das Kodieren der Datenbeschreibungen ("Deklarationen") und das Formulieren des Algorithmus (der Anweisungen zur Datenmanipulation und zur Ablaufsteuerung).

Im engeren Sinn besteht das Programmieren also aus dem Schreiben des Quelltextes ("codieren") und der Umwandlung durch den Compiler. Die letzte Phase der Programmentwicklung ist der Test und die Abnahme durch den Anwender (Benutzer).

## 2.2 Elemente eines Java-Programms

Natürlich ist auch unser erstes echtes Programm ein "Hallo Welt"-Programm. Weil Java eine objektorientierte Sprache ist, soll dieses Programm auch bereits (fast) alle Elemente eines objektorientierten Programms enthalten.

### Programm *Einfuehrung02*: "Hallo Welt" - komplett objektorientiert

```
public class Einfuehrung02 {  
    public static void main (String[] args)    {  
        private String text;                // Objektreferenz erzeugen  
        text = new String("Hallo ");        // Objekt im Speicher anlegen  
        text = text.concat("Welt");         // Nachricht senden  
        System.out.println(text);           // Nachricht senden  
    }  
}
```

Kleiner Trost für den Neueinsteiger: Dieses Programm ist für die nächsten 50 Seiten das schwierigste. Es enthält eine Vielzahl von speziellen Java-Techniken: Zugriffsmodifizier, Arbeiten mit Referenzen und Objekten, mit Arrays und Methodenaufrufe, Parameterübergabe und Returnwerte bzw. *void*. Außerdem wird mit der nicht ganz selbsterklärenden Technik für die Standard-Ausgabeeinheit *System.out* gearbeitet. Also: Das Programm hat es in sich, und es enthält auch all das, was uns in den nächsten Kapiteln beschäftigen wird.

In diesem Kapitel geht es zunächst vor allem darum, zu verstehen, wie ein Javaprogramm aufgebaut ist und welche formalen Vorschriften einzuhalten sind.

### 2.2.1 Aufbau eines Java-Programms

Der Quelltext eines Programms ("Sourcecode") besteht aus einzelnen Wörtern und Symbolen (auch "Token" genannt), die (meistens) durch Leerstellen abgetrennt sind. Beginnen wir mit einer grundsätzlichen Unterscheidung der einzelnen Token: es gibt "Schlüsselwörter" (keywords), die Bestandteil der Javasprache sind, und es gibt frei vom Programmierer gewählte Wörter ("Programmiererwörter", Bezeichner).

#### Was sind Schlüsselwörter und was sind Programmiererwörter?

- **Schlüsselwörter** sind z.B. *class*, *public*, *static*, *void*. Die Schreibweise dieser Begriffe (ihre **Syntax**) ist exakt vorgegeben, und auch die Bedeutung (ihre **Semantik**) ist in der Sprachdefinition festgelegt. Es gibt etwa 50 Schlüsselwörter in Java.
- **Programmiererwörter** sind Namen ("Bezeichner", identifier), die der Programmierer frei gewählt hat, z.B. für Klassen, Methoden oder für Daten. Das Programm *Einfuehrung02* enthält z.B. folgende Programmiererwörter: *Einfuehrung02*, *args* und *text*.

#### Hinweise zum Arbeiten mit JOE

Wenn Sie mit JOE (oder einem anderen Java-Editor) arbeiten, erkennen Sie, dass diese Unterschiede auch farblich dargestellt werden.

Das Programm *Einfuehrung02* besteht aus nur einer Methode, der Methode *main*. Innerhalb von Methoden werden folgende typische Arbeiten ausgeführt:

- es werden Daten beschrieben und dem Programm zur Verfügung gestellt,
- danach werden die Daten verarbeitet (manipuliert), und
- zum Schluss wird das Ergebnis ausgegeben.

Ausgedrückt mit einigen Fachausdrücken der objektorientierten Programmierung kann man den Ablauf auch wie folgt beschreiben: Wir benutzen die mitgelieferte Klasse *String*, um einen neuen Datentyp im Arbeitsspeicher zu beschreiben. Dazu wird zunächst eine Objektreferenz (man sagt auch: eine Instanzvariable) erstellt, sie bekommt den Bezeichner *text*. In der nächsten Zeile wird das Objekt selbst im Speicher erzeugt und durch das Senden einer Nachricht mit dem Text "Hallo" gefüllt. Der vorletzte Befehl ergänzt diesen Text um das Wort "Welt". Danach wird mit der Methode *println* der Inhalt des Objekts ausgegeben am Bildschirm.

#### Übung zum Programm *Einfuehrung02*

Editieren Sie das Programm, wandeln Sie es um und rufen Sie es auf zur Ausführung. Achten Sie auf Groß-/Kleinschreibung, und geben Sie beim Aufruf des Interpreters nicht die Dateierweiterung *.class* für das ausführbare Programm an. Das Ergebnis dieses Programmaufrufs ist die Ausgabe des Textes "Hallo Welt". Variieren Sie danach das Programm, geben Sie beliebige andere Texte aus.

### 2.2.2 Aufruf des Programms

Für die Ausführung des Programms muss in einer Commandline folgender Befehl eingegeben werden:

```
java Einfuehrung02
```

Dadurch wird vom Betriebssystem eine *.exe*-Datei gestartet mit dem Namen *java.exe*. Dieses Programm startet die Java Virtuelle Maschine (JVM), die dann ihrerseits die Klasse lädt, deren Name als Aufrufparameter mitgegeben wurde (*Einfuehrung02.class*, aber ohne die Dateiergung *.class*). Diese Start-Klasse ist die Hauptklasse einer Java-Anwendung. Sie muss einige spezielle Anforderungen erfüllen:

- sie muss eine ausführbare Klasse sein, d.h.
  - sie sollte *public* sein,
  - sie muss eine *main*-Methode enthalten.

Die JVM enthält einen Classloader, der die Datei mit dem Bytecode auf dem externen Speicher (i.d.R. ist das die Festplatte) sucht und in den Arbeitsspeicher lädt. Gesucht wird die Datei innerhalb des aktuellen Verzeichnisses. (Wenn der Aufruf aus einem anderen Ordner erfolgt, muss mit Hilfe der *classpath*-Variablen der Suchpfad entsprechend ergänzt werden, siehe dazu Anhang A.)

Jedes Java-Programm startet mit dem ersten Befehl in der *main*-Methode. Von dort aus wird dann alles Weitere gesteuert. Im Programm *Einfuehrung02* wird zunächst die mitgelieferte Klasse *String* geladen und dann davon eine Instanz erzeugt. Während der Laufzeit eines Javaprogramms können beliebig viele Instanzen von einer oder mehreren Klassen erzeugt werden. Dazu werden die jeweiligen Klassen dann bei Bedarf, also wenn sie das erste Mal gebraucht werden, in den Arbeitsspeicher geholt. Man sagt, die Klassen werden dynamisch geladen. Wenn die Klassen und die Instanzen nicht mehr benötigt werden, sorgt ein eingebauter Mechanismus der JVM dafür, dass der belegte Speicherplatz wieder frei gegeben wird ("Garbage Collector").

Wenn das Programm *Einfuehrung02.class* insgesamt beendet ist, wird auch die JVM beendet.

### Hinweise zum Arbeiten mit JOE

Durch Drücken der Tasten UMSCHALTUNG|STEUERUNG|F10 wird ein Grundgerüst einer jeden Java-Applikation in das Editierfenster eingefügt. Leider entspricht das Gerüst nicht exakt den Empfehlungen, die von der Firma Sun in den "Java Code Conventions" gegeben werden, denn sowohl die geschweiften wie auch die eckigen Klammern stehen nicht dort, wo sie stehen sollten. Unser Vorschlag: Bitte korrigieren Sie das Programmgerüst entsprechend (als Muster siehe Programm *Einfuehrung02.java*).

## 2.3 Schlüsselwörter, Syntax und Semantik

Zur Grammatik der Java-Sprache gehören die Definition der Schlüsselwörter, die Beschreibung der formalen Regeln für das Codieren des Quelltexts und auch die Beschreibung der Bedeutung, die die Sätze im Quelltext haben.

### 2.3.1 Schlüsselwörter (keywords)

Schlüsselwörter bilden den reservierten Teil des Sprachumfangs. Sie dürfen nicht verwendet werden, um Namen für Speicherplätze, Klassennamen oder Methodennamen zu bilden. Es gibt etwa 50 reservierte Namen in Java, z.B. gehören dazu *class*, *int*, *switch*, *while* usw. Im JOE-Editor sind diese Schlüsselwörter farblich gekennzeichnet. Alle Schlüsselwörter bestehen ausschließlich aus ASCII-Zeichen (siehe nachfolgendes Kapitel).

### 2.3.2 Syntaxregeln

Jede Programmiersprache hat Regeln für den Zusammenbau der Anweisungen. [Durch diese Syntaxbeschreibung wird festgelegt:](#)

- [Art und Aufbau der Datenbeschreibung,](#)
- [Art und Aufbau der Befehle für die Programmausführung.](#)

Die Beschreibung der Daten wird auch Deklaration oder Definition genannt. (Java unterscheidet diese beiden Begriffe nicht so streng, wie das in anderen Sprachen üblich ist). Das Kapitel 4 befasst sich ausführlich mit dem Thema "Beschreibung der Daten".

Die Befehle werden auch Anweisungen, Operationen oder Statements genannt. In den Kapiteln 5 und folgende werden die unterschiedlichen Arten der Anweisungen und ihre inhaltliche Bedeutung (ihre Semantik) erläutert.

Zunächst werden wir grundsätzliche Regeln für die Programmiersprache Java besprechen. Eine Sprache besteht aus einer Folge von Wörtern (und Sonderzeichen), die nach bestimmten Regeln aneinander gereiht werden.

Häufig wird die Syntax und die Grammatik einer Programmiersprache in einer besonderen Notation (in einer "Metasprache") formal beschrieben. So gibt es grafische Beschreibungssprache oder auch Syntax-Diagramme, z.B. **Backus-Naur-Form (BNF)**, siehe Anhang B.

Wichtige Syntaxregeln der Java-Sprache sind:

- Java-Programme bestehen aus einzelnen Wörtern und Symbolen ("token"), die durch festgelegte Trennzeichen ("delimiter") getrennt werden.
- Bei den Token werden Schlüsselwörter (reservierte Wörter) und Programmierwörter (frei gewählte Namen, "Bezeichner", identifier) unterschieden.

- Die Schreibweise im Quelltext ist formatfrei. Eine maximale Zeilenlänge ist nicht festgelegt. Die Steuerzeichen wie Zeilenschaltung (line-feed) oder Tabulator (tab) spielen keine Rolle für die Interpretation des Quelltexts.
- Java unterscheidet zwischen Groß- und Kleinschreibung (man sagt, Java ist "case-sensitiv").
- Am Ende eines Befehls steht ein Semikolon.
- Kommentare können beliebig eingefügt werden:
  - am Zeilenende (dann beginnen sie mit //) oder auch
  - über mehrere Zeilen (dann beginnen sie mit /\* und enden mit \*/)

### Übung zum Programm Einfuehrung02

Versuchen Sie zu klären, was die Schlüsselwörter und was die Identifier in diesem Programm sind (Hinweis: Schlüsselwörter werden farblich abgehoben im JOE-Editierfenster). Ändern Sie danach das erste Token in diesem Programm von *class* auf *Class*. Testen Sie die Reaktion des Compilers auf diese Änderung.

## 2.4 Bezeichner (identifier) und Namensregeln

### 2.4.1 Wofür werden Namen vergeben?

Identifier sind frei gewählte Namen für die Speicherplätze (Variablen), für Klassen, Methoden, Dateien, DB-Tabellen oder Pakete. Jedes Wort endet, wenn vom Compiler ein Trennzeichen (delimiter) erkannt wird. Trennzeichen sind also Zeichen, die zwei Elemente ("token") voneinander trennen und abgrenzen. Dies können "whitespaces" wie Leerstellen ("blank") sein oder Tabulator- oder Zeilenvorschub-Zeichen.

### Programm Einfuehrung03: Schlüsselwörter und Bezeichner

```
public class Einfuehrung03 {  
    public static void main (String[] args) {  
        int zahl;  
        zahl = 5;  
        System.out.println(zahl);  
    }  
}
```

### Übung zum Programm Einfuehrung03

Editieren Sie das Programm. Achtung: ein Abtippen ist immer sinnvoll! Fehler machen und Fehler beseitigen helfen beim Lernen einer Programmiersprache. Wandeln Sie das Programm um und führen Sie es aus. Wenn es fehlerfrei läuft: Herzlichen Glückwunsch. Wenn es Umwandlungsfehler produziert, korrigieren Sie diese selbstständig - es können nur Tippfehler sein.

### 2.4.2 Regeln für Namensvergabe

Die Regeln für das Bilden dieser Bezeichner sind einfach. Das erste Zeichen muss ein lateinischer Buchstabe sein, danach ist fast jeder Buchstaben aus jedem Alphabet (siehe Hinweise zum Thema Unicode) und jede Ziffer erlaubt. In der Länge gibt es auch keine Begrenzungen. Zu beachten ist lediglich:

- Die Unterscheidung der **Groß-/Kleinschreibung ist wichtig**.
- Schlüsselwörter sind reserviert und dürfen nicht für das Bilden von Bezeichnern benutzt werden.
- Namen dürfen keine Leerzeichen enthalten (besonders wichtig auch bei Datei- oder Verzeichnisnamen, selbst wenn das Betriebssystem dies gestattet).

Weil Java kaum einengende Vorschriften enthält für das Bilden von Namen, ist die Disziplin des Programmierers gefordert. Einige Konventionen zu diesem Thema folgen auf den nächsten Seiten.

**Gültige** (wenn auch nicht unbedingt empfohlene) Namen sind z.B.

```
kundennummer
weiß123           // Nationales Sonderzeichen ß
bool
grün              // Umlaut, nicht empfohlen
a_bereich         // Unterstrich, besser vermeiden
diesIstEinIdentifizier
a                 // wenig aussagefähig
```

**Ungültige** Namen sind z.B.

```
-gross           // Erste Stelle kein Buchstabe
lkdnr           // dto.
boolean         // Reserviertes Wort
class           // Reserviertes Wort
```

Die Syntaxregeln der Java-Sprache sind strikt einzuhalten. Verstöße werden normalerweise vom Compiler erkannt und führen zu "compile-time-errors".

#### Programm Syntax01: Was ist hier falsch?

```
public class Syntax01 {
    public static void main(String args[]) {
        int summe;
        Summe = 2000;
        System.out.println(summe);
    }
}
```



### Übung zum Programm Syntax01

Editieren und compilieren Sie dieses Programm. Die Fehlermeldung beim Umwandeln lautet: "Syntax01.java:4: cannot find symbol". Sinngemäß bedeutet dies: In der Quelldatei *Syntax01.java* in Zeile 4 befindet sich ein Symbol, das unbekannt ist. Um welches Symbol es sich handelt, steht in der nächsten Zeile der Fehlermeldungen.

Korrigieren Sie den Fehler und führen Sie das Programm aus.

## 2.5 Einige Hinweise zu möglichen Fehlern

Beim Entwickeln von neuen Programmen können Fehler zu unterschiedlichen Zeitpunkten auftreten bzw. entdeckt werden. Man unterscheidet: Compile-Time-Fehler, Run-Time-Fehler und logische Fehler.

### 2.5.1 Compile-Time-Fehler

Diese entstehen durch Verstöße gegen die Syntaxregeln der Sprache. Sie werden in jedem Fall vom Compiler erkannt.

#### Programm Syntax02: Gegen welche Syntaxregel wird verstoßen?

```
public class Syntax02 {  
    public static void main(String[] args) {  
        String str  
        System.out.println(str);  
    }  
}
```

### Übung zum Programm Syntax02

Editieren und compilieren Sie dieses Programm. Die Fehlermeldung beim Umwandeln lautet: "Syntax02.java:4: ';' expected". Sinngemäß bedeutet dies: In der Quelldatei *Syntax02.java* in Zeile 4 fehlt das Semikolon.

Korrigieren Sie den Fehler und führen Sie das Programm aus.

Bei der Fehleranalyse muss bedacht werden, dass die Fehlerursache nicht unbedingt auch in der angegebenen Zeile liegt, sie wird evtl. nur dort entdeckt. In diesem Fall fehlt das Semikolon natürlich in der Zeile 3.

Außerdem können Syntaxfehler zu Folgefehlern führen. Dadurch kann es zu einer ganzen Kette von Meldungen kommen, obwohl es vielleicht nur eine Ursache gibt. Deswegen sollte die Fehlerkorrektur immer mit der ersten Fehlermeldung des Compilers beginnen.

Besonders tückische Fehlermeldungen werden produziert, wenn die Klammerungen durch geschweifte oder runde Klammern nicht richtig oder nicht paarweise erfolgt. Dazu ein Beispiel, das allerdings in vollem Umfang noch nicht verstanden werden kann - es geht hier lediglich um die Syntax.

### Programm Syntax03: Geschweifte Klammern treten immer paarweise auf

```
public class Syntax03 {  
    public static void main(String[] args) {  
        A a = new A();    // Instanz der Klasse A erzeugen  
        System.out.println("Hallo");  
    }  
}  
class A {}
```

#### Übung zum Programm Syntax03

Das Programm müsste fehlerfrei umwandelbar und ausführbar sein. Es liefert als Ausgabe einen Gruß ("Hallo"). Das Programm funktioniert also. Jetzt löschen Sie bitte die drittletzte Zeile mit der geschlossenen geschweiften Klammer. Wandeln Sie neu um, und prüfen Sie die Meldung des Compilers. Was passiert, wenn Sie die Klammer, wie die Meldung suggeriert, in Zeile 7 einfügen?

Die Erkenntnis aus der obigen Übung ist: Die Fehlermeldung meldet zwar einen Fehler in der Zeile 3, tatsächlich jedoch liegt die Ursache einige Zeilen weiter hinten.

## 2.5.2 Run-TimesFehler

Es kann aber auch sein, dass der Compiler einen Fehler nicht erkennen kann, weil dieser erst zur Laufzeit des Programms offenbar wird. Dann handelt es sich um so genannte *run-time-error*. Sie werden in Java "Exceptions" genannt. Natürlich haben die Erfinder der Java-Sprache sich bemüht, diese Fälle möglichst auszuschließen, und wann immer möglich, werden Fehlerquellen bereits beim Compilieren erkannt und verhindert. Aber trotzdem kann es vorkommen, dass zur Ausführungszeit eine so genannte Exception (Ausnahmesituation) entsteht, die sogar zum Programmabbruch führen kann.

### Programm Laufzeitfehler01: Run-Time-Error produzieren

```
public class Laufzeitfehler01 {  
    static void main(String[] args) {  
        String name = new String("Merker");  
        System.out.println(name);  
    }  
}
```

#### Übung zum Programm Laufzeitfehler01

Versuchen Sie das Programm umzuwandeln und auszuführen. Ergebnis: Es gibt keinen compile-time-error, allerdings bricht die Ausführung ab mit folgendem run-time-error: "Main method not public".

Korrigieren Sie den Fehler und führen Sie das Programm aus.

### 2.5.3 Logische Fehler

Und dann gibt es natürlich logische oder semantische Fehler, die von dem Computer gar nicht erkannt werden (können), die aber zu falschen Ergebnissen führen, weil der Programmierer Anweisungen zwar syntaktisch korrekt, aber von der Bedeutung her falsch eingesetzt hat.

Das folgende Programm enthält einen logischen Fehler. Es handelt sich um das Beispielprogramm *Einfuehrung02*, das den Text "Hallo Welt" ausgeben soll. Das Programm wurde allerdings an einer entscheidenden Stelle modifiziert.

#### Programm *Logische01*: Logischer Fehler im Programm

```
class Logische01 {  
    public static void main(String[] args) {  
        String text;  
        text = new String("Hallo ");  
        System.out.println(text);  
        text = text.concat("Welt");  
    }  
}
```

#### Übung zum Programm *Logische01*

Wenn Sie das Programm ausführen, sollte eigentlich der Text "Hallo Welt" ausgegeben werden. Leider fehlt die "Welt", weil der Programmierer (versehentlich oder unwissend) nicht bedacht hat, dass die Befehle sequentiell ausgeführt werden, wenn er nichts anderes vorsieht.

Korrigieren Sie das Programm und führen Sie es aus.

Lösungshinweis: Die Reihenfolge der Befehle im Quelltext stimmt nicht.

Das Auffinden von logischen Fehlern in Programmen gehört zu den schwierigsten und aufwändigsten Aufgaben im Software-Entwicklungsprozess. Man nennt diesen Vorgang "Testen". Hinzu kommt, dass beim Testen von eigenen Programmen ein psychologisches Problem zu überwinden ist: Man soll herausfinden, welche Fehler das eigene Programm enthält, wo man doch viel eher nachweisen möchte, dass es fehlerfrei ist.

Vielleicht hilft folgende Weisheit: Wenn der Fehler nicht da ist, wo man ihn sucht, dann ist er woanders.

Manchmal wird auch der Begriff "[Debugging](#)" für die Fehlersuche und -bereinigung benutzt (als "bug" wird ein Computerfehler bezeichnet). Ein Werkzeug (tool) zum Debuggen nennt man Debugger-Programm. Damit kann ein Programm kontrolliert und schrittweise (mit einzelnen Haltepunkten zwischen den Schritten) ausgeführt werden. Ein Debugger ist meistens Bestandteil einer Entwicklungsumgebung (IDE).

## 2.6 Empfehlungen für lesbaren Quelltext

### 2.6.1 Guter Stil für die Namensvergabe

Wie bereits beschrieben, gibt es in Java kaum Einschränkungen beim Bilden von Bezeichnern. Deswegen ist es wichtig, an die Disziplin und an den guten Willen des Programmierers zu appellieren: Um die Lesbarkeit des Programms zu verbessern, halten Sie sich bitte bei der Namensvergabe an folgende [freiwillige Konventionen](#):

- Die Namen sollen sprechend sein. Einzelne Buchstaben wie *x* oder *y* sind in der Regel zu wenig aussagefähig. Die Bezeichner sollen möglichst die Funktion beschreiben (wenn eine Variable *brutto* heißt, soll dies auch dem Inhalt entsprechen).
- Es wird dringend empfohlen, lediglich die Groß- und Kleinbuchstaben aus dem ASCII-Code (siehe nachfolgendes Kapitel) zu verwenden. Benutzen Sie z.B. keine deutschen Umlaute wie *ä* oder *ü*, auch keinen Unterstrich oder das Dollarzeichen *\$*, selbst wenn Java dies erlaubt. Dadurch ist die Lesbarkeit und Austauschbarkeit der Programme gewährleistet.
- Klassennamen beginnen mit einem Großbuchstaben, z.B. *class Kunden*
- Variablen- und Methodennamen werden klein geschrieben, z.B. *kdnr* oder *drucken*
- Hauptwörter (Nomen) eignen sich häufig gut für Klassennamen, während sich Eigenschaftswörter (Adjekte) für Variablen und Zeitwörter (Verben) für Methoden anbieten.
- Wenn Variablennamen oder Methodennamen aus mehreren Wörtern bestehen, werden die Wörter ohne Trennzeichen zusammen geschrieben, und jedes Teilwort beginnt mit einem Großbuchstaben. Beispiele: *mustBerechnen* oder *druckenRechnungen*.
- Konstantennamen werden komplett groß geschrieben, z.B. *MWSTPROZENT*
- Aus historischen Gründen sind zwar der Unterstrich *\_* und das Dollarzeichen *\$* als erstes Zeichen für Identifier erlaubt, ihre Verwendung wird aber nicht empfohlen.

### 2.6.2 Guter Stil für Formatierung von Quellenprogrammen

Wenn auch generell der Quellcode formatfrei ist, haben diese ersten Beispiele bereits gezeigt, dass es unbedingt hilfreich ist, wenn der Quelltext so formatiert wird, dass er für den Menschen leicht lesbar ist. So sollten die geschweiften Klammern, die immer paarweise codiert werden müssen, auch optisch als Paar erkennbar sein. Und das wird dadurch erreicht, dass die schließende Klammer auf derselben Spalte

(nicht verwechseln mit Zeile) steht wie der Beginn des Blocks. Zur Abschreckung hier ein Beispiel, wie **nicht** codiert werden sollte:

### Programm Trash01: So nicht!

```
public class Trash01 { public static void main
(String[] args){int
    summe; summe=
    2000; System.out.println(summe);}}
```

Wie man (schwer) erkennen kann, handelt es sich um das bereits bekannte Programm *Syntax01.java*. Es erfüllt auch immer noch dieselbe Aufgabe, denn die Syntaxregeln wurden eingehalten. Deswegen kann auch dieses Programm fehlerfrei umgewandelt und gestartet werden. Aber welcher menschliche Leser kann den Quelltext verstehen?

Deswegen hier weitere Empfehlungen für das Kodieren von übersichtlichen Programmen:

- Pro Zeile möglichst nur eine Anweisung.
- Kommentare schreiben für Quellcode, der ansonsten schwer verständlich wäre.
- Programmblöcke optisch kennzeichnen durch Einrücken nach rechts. Alle Zeilen innerhalb eines Klammerpaares werden eingerückt, dadurch wird die Verschachtelung von Programmteilen deutlich.
- Kein Whitespace (siehe hierzu **Glossar** im Anhang E) zwischen Methodennamen und öffnender Klammer.
- Die Programme sollten insgesamt einen konsistenten Aufbau haben. Hilfreich sind Programmierrichtlinien, die unternehmensweit eingehalten werden.

# 3

## Informationen maschinell darstellen

---

Bevor Sie in den nachfolgenden Kapiteln detailliert erfahren, wie in Java Klassen definiert und Variablen im Arbeitsspeicher erzeugt und mit Hilfe von Methodenaufrufen verarbeitet werden, wollen wir in diesem Kapitel erläutern,

- wie Daten codiert werden, so dass sie maschinell gespeichert und verarbeitet werden können und welche Codes es gibt,
- welche Bedeutung der von Java genutzte Unicode hat und
- welche Herausforderungen bei der Entwicklung von international einsetzbaren Softwaresystemen und beim globalen Datenaustausch gelöst werden müssen.

### 3.1 Zahlensysteme und der Binärcode

#### 3.1.1 Bits und Bytes

Im Arbeitsspeicher einer binären Rechanlage gibt es ausschließlich Bitmuster, die jeweils nur einen von zwei möglichen Werten darstellen können ("bits"). Die Werte dieser binären Darstellung nennt man "Null" oder "Eins" bzw. "ja" oder "nein". Jedes Zeichen einer Sprache, insbesondere auch die alphabetischen Zeichen, jeder Zahlenwert und auch alle Multimediadaten wie Bilder oder Töne werden in Form dieser **bits** verschlüsselt, stellen also ein Muster aus Nullen und Einsen dar.

Die Vorteile dieser binären Verschlüsselung sind:

- Sie ist eine ideale Grundlage für den Computerbau. Die zwei Zustände lassen sich technisch leicht realisieren (optisch: an oder aus; elektrisch: zwei verschiedenen Spannungswerte; magnetisch: magnetisiert oder nicht magnetisiert...).
- Nicht nur Texte und Zahlen lassen sich binär kodieren, sondern auch Töne, Bilder und Grafiken (Multimedia-Daten).
- Für numerische (mathematische) Probleme: das Zweiersystem (Dualsystem) ist ein vollwertiges Stellenwertsystem zum Rechnen.
- Für logische Probleme: es gibt eine zweiwertige Logik (z.B. Boolesche Algebra). Die beiden Wahrheitswerte *wahr* und *falsch* lassen sich vielfältig verknüpfen, z.B. mit UND oder mit ODER (siehe auch Kapitel 7: Logische Operatoren).

Die Bits sind die kleinste Informationseinheit in einem Computer. Jedoch kann Java nicht auf einzelne Bits zugreifen. Die kleinste adressierbare Einheit ist ein Byte, und das besteht aus einer Gruppe von 8 bits.

#### 3.1.2 Stellenwertsysteme

Ein Java-Programmierer kann in seinem Quelltext mit unterschiedlichen Zahlensystemen arbeiten: Dezimal- Oktal, Hexadezimal- oder Dualsystem. Allen ist gemeinsam, dass sie Stellenwertsysteme sind: die Zahlen setzen sich aus einzelnen Ziffern zusammen, und jede Ziffer hat einen Nennwert (im Dezimalsystem von 0 - 9, im Dualsystem von 0 bis 1 usw.). Der Nennwert sagt noch nichts aus über den tatsächlichen Wert, den eine Ziffer hat. Dieser hängt ab von der Stelle der Ziffer innerhalb der Zahl. Das heißt, abhängig von der Position einer Ziffer innerhalb einer Zahl verändert sich der tatsächliche Wert.

##### Beispiel 1:

Im **Dezimalsystem** muss der Nennwert einer Position so oft mit 10 multipliziert werden, wie es die Position erfordert (beginnend rechts mit Null):

$$\begin{array}{rclcl} \text{die Zahl 532 hat den Wert} & 5 * 10^2 & = & 500 \\ & + & 3 * 10^1 & = & 30 \\ & + & 2 * 10^0 & = & 2 \end{array} \quad (\text{Summe ist: 532})$$

##### Beispiel 2:

Im **Dualsystem** muss der Nennwert einer Position so oft mit 2 multipliziert werden, wie es die Position erfordert (beginnend rechts mit Null):

$$\begin{array}{rclcl} \text{die Zahl 101 hat den Wert} & 1 * 2^2 & = & 4 \\ & 0 * 2^1 & = & 0 \\ & 1 * 2^0 & = & 1 \end{array} \quad (\text{Dezimal: 5}).$$

Die Stellenwerte sind die Potenzen der Basis. Die Basis sind die Anzahl der Ziffern, die das Zahlensystem hat. Das Dualsystem arbeitet zur Basis 2. Ein Byte hat 8 Bits, das sind  $2 * 2 * 2 * 2 * 2 * 2 * 2 * 2 = 256$  Kombinationsmöglichkeiten.

##### Beispiel 3:

Das **hexadezimale Zahlensystem** hat insgesamt 16 Ziffern (von 0 - 9 und zusätzlich die "Ziffern" A, B, C, D, E und F), es arbeitet also zur Basis 16. Folglich hat die hexadezimale Zahl AE3 den folgenden dezimalen Wert:

$$\begin{array}{rclcl} \text{die Zahl AE3 hat den Wert} & A(=10) * 16^2 & = & 2560 \\ & E(=14) * 16^1 & = & 224 \\ & 3 & * 16^0 & = & 3 \end{array} \quad (\text{Dezimal: 2787}).$$

Der Computer arbeitet generell mit dem Dualsystem, der Mensch rechnet mit dem Dezimalsystem. Das hexadezimale System wird lediglich benutzt, damit es für Menschen leichter ist, den Inhalt eines Bytes (oder besonders von mehreren Bytes) benennen zu können.

Die praktische Bedeutung des Oktalsystems ist gering, im Wesentlichen findet man ihn als ein theoretisches Thema in EDV-Lehrbüchern.

## 3.2 Informationsformen

Beim Verarbeiten von Daten durch den Computer müssen unterschiedliche Arten von Informationen unterschieden werden:

### Textcodierung im Computer

- Codierte Einzelinformationen, die aus fest vereinbarten Zeichen eines Alphabets, aus Ziffern oder aus Zeichen eines Sachgebiets wie Physik oder Mathematik bestehen. Für diese Zeichen kann die Art der Bitverschlüsselung und die inhaltliche Bedeutung der bits einmalig und allgemeingültig festgelegt werden. Die Zuordnung der Bedeutung zu den Bitfolgen wird Code genannt. So gibt es z.B. den ASCII- und den EBCDIC-Code.
- Eine Codetabelle legt nicht nur die Bitfolgen fest, sondern auch die textliche Beschreibung für jedes vereinbarte Zeichen ("benannte Zeichen"). Nicht festgelegt ist in einem Code die Darstellungsform (also das Aussehen oder die Größe) des Zeichens. Die englische Bezeichnung für Zeichen ist "character".

### Zahlencodierung im Computer

- Zahlen bestehen aus einzelnen Ziffern (und damit aus mehreren Character). Sie werden aber als ganze Einheit behandelt und rein binär verschlüsselt. Dabei wechselt das Stellenwertsystem. Man geht ganz vom Dezimalsystem weg und benutzt für die komplette Zahl die Stellenwertigkeit 2.
- Bei dieser Art der maschinellen Darstellung benötigt man keine Code-Tabelle, sondern orientiert sich allein an der Stellenwertigkeit des Zahlensystems. Die jeweilige Programmiersprache legt dann nur noch fest, wieviel Stellen für Zahlendarstellung reserviert werden (man spricht dann von "eingebauten Datentypen", z.B. ist der eingebaute Datentyp *int* bei Java immer 4 Bytes lang).
- Weil bei der Wandlung einer Dezimalzahl in die Zweierdarstellung Ungenauigkeiten entstehen können, gibt es das BCD-System, bei dem jede einzelne Ziffer einer Dezimalzahl einzeln in einen Binärwert verschlüsselt wird. Wir werden auch hierauf zurückkommen.

### Multimedia-Daten im Computer

- "Uncodierte" Informationen, für die es keine festgelegten Zeichen oder Muster gibt, sondern eine praktisch unbegrenzte Anzahl von unterschiedlichen Ausprägungen und damit Interpretationen. So gibt es für Bilder z.B. die punktförmige Anordnung von bits in einer Bitmap-Datei ("pixel") oder für akustische Informationen oder für Videos Bitfolgen in kompletten Dateien. Hier gibt es keinen Zeichensatz und keine byteweise Ordnung, sondern nur eine (unbegrenzte) Aneinanderreihung von einzelnen Bits, die von speziell dafür geschriebenen Pro-



grammen interpretiert werden. Dies geschieht häufig unter Angabe des so genannten Mime-Typs, z.B. image/gif.

Das folgende Beispiel zeigt die Verarbeitung einer Datei mit der Dateiergung *.wav* (Audio-Datei). Dies ist das einzige Beispiel in diesem Buch, das mit Multimedia-Daten (Sound, Video, Bilder oder Grafiken) arbeitet. Im Regelfall werden dafür Spezialkenntnisse, insbesondere auch Kenntnisse in der Programmierung von grafischen Benutzeroberflächen, benötigt, und das ist nicht Thema dieses Buches.

#### Programm Sound01: Audio-Datei abspielen per Java-Programm

```
import java.net.*;
import java.applet.*;
public class Sound01 {
    public static void main(String[] args) throws Exception {
        URL url = new URL("file://c:/windows/media/chord.wav");
        AudioClip clip = Applet.newAudioClip(url);
        clip.play();
    }
}
```

Hinweise zu den Beispielen in diesem Kapitel

Wir empfehlen, die Beispiele in diesem Kapitel auf jeden Fall selbst zu editieren, umzuwandeln und mit ihnen zu arbeiten. Dadurch bekommen Sie Übung im Codieren und vor allem auch in der Fehlerbearbeitung. Sie werden vertraut mit der Syntax von Java (auch wenn die Semantik noch nicht in jedem Detail verstanden werden kann).

Das Verständnis für die interne Darstellung und für die unterschiedlichen Codes wird später immer wieder benötigt, z.B. bei den bitweisen Operatoren (Kapitel 7) oder bei der Diskussion um multinationale Anwendungen.

#### Übung zum Programm Sound01

Denken Sie beim Testen des Programms daran, den Lautsprecher zu aktivieren. Ändern Sie danach das Programm so, dass eine (beliebige) andere Sound-Datei abgespielt wird.

Die bekanntesten Verschlüsselungsverfahren **für Zeichen** sind der ASCII-Code (verbreitet im PC- und UNIX-Bereich), der EBCDIC-Code (verbreitet auf den IBM-Großrechnern) und der Unicode. Die ersten beiden gruppieren die einzelnen Bits zu Einheiten von 8 Nullen und Einsen - genannt Bytes. Bytes sind in den heute verbreiteten Rechnern die kleinste adressierbare Einheit. Sie bieten die  $2^8$  Möglichkeiten, insgesamt also 256 verschiedene Verschlüsselungen, die mit einer Bedeutung belegt werden können. Im Unicode dagegen sind viele Tausend Zeichen verschlüsselt. Dafür stehen 16 bits zur Verfügung.

3.3 ASCII-Code

Im ASCII-Code (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) wird jedes Zeichen mit 7 bit codiert. Es sind also 128 unterschiedliche ASCII-Zeichen möglich, und dies sind (wie der Name schon vermuten lässt) die Zeichen aus dem englischen Sprachraum (Alphabet und Ziffern). Eine gebräuchliche Bezeichnung für diesen Code ist deshalb auch US-ASCII. Sonderzeichen aus anderen Sprachen, z.B. die deutschen Umlaute ä, ü oder ö, sind im 7-bit-ASCII-Code nicht vorgesehen. Ursprünglich benötigten die Computer ein weiteres Bit pro Zeichen als Prüfbit, so dass auch der US-ASCII-Code mit Bytes (= 8 bits) arbeitet.

Die 128 Zeichen des ASCII-Codes sind durchnummeriert. In einer Code-Tabelle werden den Buchstaben des Alphabets, den Ziffern und einigen Sonderzeichen die Platznummern 0 - 127 zugeordnet, und für die interne Darstellung werden diese dezimalen Platznummern umgewandelt in eine binäre Darstellung. So hat der Großbuchstabe 'A' die Platznummer 65 (auch Dezimalwert 65 genannt). In der binären Darstellung bekommen die 7 bits des ASCII-Codes eine Stellenwertigkeit von 1 beginnend und dann von rechts nach links jeweils den doppelten Wert. Für den Buchstaben A sieht die maschinelle Darstellung und ihre Interpretation also wie folgt aus:

Bitmuster	0	1	0	0	0	0	0	1	
Stellenwert	128	64	32	16	8	4	2	1	
Umrechnung	-	64	-	-	-	-	-	1	64 + 1 = 65

Abb.3.1: Binäre Darstellung des Buchstabens 'A' im ASCII-Code

**Sie finden im Anhang C eine Code-Tabelle.** Diese enthält zwar die ersten 256 Zeichencodierungen des Unicode (siehe Abschnitt 3.6), doch die ersten 128 Zeichen sind identisch mit dem ASCII-Code. Dort sind z.B. die Platznummern 65 und 66 wie folgt beschrieben:

Platz-Nr.	hexadezimaler Wert	Symbol	Name
65	0x41	A	Latin Capital Letter A
66	0x42	B	Latin Capital Letter B

Abb. 3.2: Ausschnitt aus ASCII-Tabelle für Platz 65 und Platz 66

Die hexadezimale Darstellung dient ausschließlich dem Zweck, die Bitfolge eines Zeichens prägnant wiederzugeben. Es ist eben einfacher zu sagen, das Byte enthält "hex. 41" als "1000001". Zur Kennzeichnung, dass dies ein hexadezimaler Wert ist, werden die beiden Zeichen 0x dem eigentlichen hex-Wert vorangestellt. Weder mit

dem binären Inhalt noch mit der hexadezimalen Bezeichnung des Inhalts wird der Programmierer im Programm normalerweise arbeiten.

#### Programm Ascii01: Arbeiten mit dem Buchstaben 'A' im Java-Programm

```
public class Ascii01 {  
    public static void main(String args[]) {  
        char zeichen = 'A';  
        System.out.println(zeichen);  
    }  
}
```

Es ist auch möglich, anstelle des Zeichens dessen Platznummer in der Tabelle des ASCII-Codes auszugeben.

#### Programm Ascii02: Ausgeben der Platznummer eines Zeichens

```
public class Ascii02 {  
    public static void main(String args[]) {  
        char zeichen = 'A';  
        System.out.println((byte) zeichen);  
    }  
}
```

Sie werden aber in den nachfolgenden Beispielen sehen, dass für internationale Programme (gekennzeichnet durch mehrere Sprachen, unterschiedliche Sonderzeichen, kulturelle Besonderheiten bei der Darstellung von Zahlen oder Datumsangaben usw.) es häufig notwendig ist, Zeichen umzuformen oder individuell zu interpretieren. Dazu sind detaillierte Kenntnisse der Codierungsformen und der verwendeten Zeichencodes erforderlich.

Deswegen soll das nächste Programm demonstrieren, wie der Programmierer sowohl den binären Inhalt als auch die hexadezimale Repräsentation eines Bytes ausgeben kann.

#### Programm Ascii03: Binärer und hexadezimaler Wert eines Zeichens

```
public class Ascii03 {  
    public static void main(String args[]) {  
        char zeichen = 'A';  
        System.out.println(Integer.toBinaryString(zeichen));  
        System.out.println(Integer.toHexString(zeichen));  
    }  
}
```

Die ersten 32 Zeichen des ASCII-Codes sind für Steuerzeichen ("control character") reserviert, etwa für die Steuerung eines Druckers. Diese (undruckbaren) Kontrollzei-

chen sind historisch begründet und haben heute keine große Bedeutung mehr. Ausnahmen sind:

Tabulator	Dezimalwert 9
Line Feed (LF, Zeilenvorschub)	Dezimalwert 10
Carriage Return (CR, Wagenrücklauf)	Dezimalwert 13

Mit dem Dezimalwert 32 wird das Leerzeichen (space, blank) dargestellt. Man beachte: Auch das Leerzeichen ist für den Digitalrechner ein Schriftzeichen, nur dass es meistens ausschließlich durch die Hintergrundfarbe am Bildschirm dargestellt wird. Ab Dezimalwert 33 werden in der Tabelle die wichtigsten druckbaren Zeichen (aus der englischen Sprache) verschlüsselt: die lateinischen Groß- und Kleinbuchstaben A - Z, die Ziffern 0 - 9 und einige Sonderzeichen.

### Übung:

Bitte ermitteln Sie anhand der Tabelle im Anhang C, wie das Zeichen ' > ' (Größer als) im ASCII-Code dargestellt wird. Geben Sie für dieses Zeichen den hexadezimalen Wert, den Dezimalwert und den binären Wert an. Überprüfen Sie Ihre Überlegungen per Programm.

Im nächsten Beispiel wollen wir einen **Binärwert** umwandeln in ein Zeichen.

#### Programm Ascii04: Bitstring in Zeichen umwandeln

```
public class Ascii04 {
    public static void main(String args[]) {
        String bits = "1000001";
        System.out.println((char)Integer.parseInt(bits, 2));
    }
}
```

Es ist auch möglich, einen **hexadezimalen** Wert in eine Dezimalzahl umzuwandeln. Das nächste Programm übernimmt diese Aufgabe und interpretiert zusätzlich diesen Dezimalwert als Positionsnummer in der Codetabelle.

#### Programm Ascii06: Hexadezimalen String in Character umwandeln

```
public class Ascii06 {
    public static void main(String args[]) {
        String bits = "5A";
        System.out.println((char)Integer.parseInt(bits, 16));
    }
}
```

#### 3.4 Erweiterungen des ASCII-Code

Aufbauend auf 7-bit-ASCII existieren mehrere erweiterte Zeichensätze ("codepages") mit nationalen Sonderzeichen. Prüfbits sind im ASCII-Code seit vielen Jahren nicht mehr erforderlich, deswegen konnte man das achte Bit auch für die Zeichenverschlüsselung verwenden. Dadurch hat sich der "Wertebereich" verdoppelt, anstatt 128 Verschlüsselungsmöglichkeiten gibt es somit 256 verschiedene Bitkombinationen in einem Byte.

Weil auch diese Möglichkeiten nicht ausreichen, um alle existierenden Zeichen zu codieren, gibt es eine Vielzahl von **unterschiedlichen Zeichensätzen** ("Codepages"), in denen für die 128 zusätzlichen Möglichkeiten länder- oder plattform-spezifischen Sonderzeichen zugeordnet sind. Sehr gebräuchlich ist die von ISO definierte Zeichensatz-Familie mit der Bezeichnung ISO8859-x. Diese wird von Linux/Unix und auch von MS-Windows (außer im DOS-Fenster) verwendet. Es gibt 15 verschiedene Ausprägungen, alle enthalten 256 Zeichen, wobei die ersten 128 Zeichen identisch sind mit dem ASCII-Zeichensatz und die nächsten 128 Zeichen je nach Kulturkreis oder Land unterschiedlich belegt sind. So gibt es z.B.

- ISO 8859-1 LATIN-1, enthält die westeuropäischen Sonderzeichen
- ISO 8859-2 LATIN-2, osteuropäische Sprachen wie Polnisch, Kroatisch...
- ISO 8859-5 kyrillisch, wie Russisch, Ukrainisch, Bulgarisch
- ISO 8859-7 neugriechisch

Bei MS-Windows werden folgende Codepages verwendet:

- CP 1252 (für MS-Windows-Programme, z.B. MS-Word...)
- CP 437 (DOS Latin-US - Amerikanisch, für DOS-Fenster von MS-Windows)
- CP 850 (DOS Latin-1 (West - Europa)
- CP 852 (DOS Latin-2 (Ost - Europa)

Viele Codierungen sind mehrfach belegt. **Die Bedeutung einer bestimmten Bitkombination kann nur in Verbindung mit der eingestellten Codepage erkannt werden.** Dieser fehlende universelle Standard behindert natürlich den internationalen Datenaustausch und hat zur Folge, dass viele Programme nicht kompatibel sind.

Die Tabelle im Anhang C zeigt für die Platznummern 128 bis 255 zusätzlich zum Unicode die Bedeutung, die diese Bitkombinationen in der Windows-Welt haben. Dabei ist der Codeset CP1252 praktisch identisch mit dem westeuropäischen Standard ISO 8859-1 und damit auch mit den ersten 256 Stellen des Unicodes, allerdings mit einer Ausnahme. Im ISO 8859-1 werden die Zeichen 128 - 159 nicht genutzt, Microsoft jedoch hat diese mit einer Bedeutung belegt.

#### **Programm Ascii07: Arbeiten mit dem Euro-Zeichen**

```
public class Ascii07 {  
    public static void main(String args[]) {  
        char zeichen = '€';           // Euro-Zeichen
```

```

        System.out.println(zeichen);
    }
}

```

### Übungen zum Programm *Ascii07* (nur für MS-Windows)

**Übung 1:** Editieren Sie das Programm. Das Eurozeichen sollte auf der Tastatur (beim Buchstaben E?) vorhanden sein. Laut CP1252-Codepage hat dieses Zeichen die Platznummer 128. Wenn Sie das Programm jedoch in einem DOS-Fenster testen, wird der Inhalt dieses Zeichens aber als 'C' interpretiert und ausgegeben, denn dort gilt die Codepage 850.

**Übung 2:** Wenn Sie jedoch die Ausgabe des Programms umleiten durch folgenden Befehl in der Commandline der DOS-Box: "java Ascii07 > datei.txt", können Sie sich anschließend den Dateiinhalt in einem Editor oder Textprogramm anschauen - dort steht das €-Zeichen.

Was können Sie aus diesen Übungen lernen? Auf jeden Fall die Erkenntnis, dass nur die ersten 128 Zeichen des ASCII-Codes genormt sind. Alle anderen Zeichen, insbesondere auch die deutschen Umlaute, können Probleme bereiten, sobald Sie die Plattform (oder auch nur das Programm) wechseln. Natürlich ist diese fehlende Kompatibilität der Daten in der Praxis nicht akzeptabel, und Java bietet auch dafür eine sehr komfortable Lösung. Sie werden diese später kennenlernen.

## 3.5 Rein binäre Codierung von Zahlen

Im ASCII-Code sind auch die Binärverschlüsselungen für die 10 Ziffern des Dezimalsystems enthalten.

### Übung

Bitte klären Sie anhand der Tabelle im Anhang C, wie die Ziffer 2 im ASCII-Code als Zeichen verschlüsselt wird (hexadezimal).

Eine Ziffer, die als Zeichen angesehen wird, wird anderes codiert als eine Ziffer, die eine Zahl ist. Eine Zahl kann aus mehreren Ziffern bestehen, z.B. besteht die Zahl 270 aus drei Ziffern. Um diese Zahl als Ganzes zu codieren, benötigt man keinen ASCII-Code, sondern man verschlüsselt diese Dezimalzahl als rein binären Wert. Hierbei geht man vom Dezimalsystem weg und verwendet die Stellenwertigkeit des Dualsystems. Man codiert also anhand der Stellenwertigkeit der einzelnen Bits.

Beispiel: Wie wird die Dezimalzahl 270 als rein binäre Zahl codiert?

Stellenwert:	256	128	64	32	16	8	4	2	1
Bitdarstellung:	1	0	0	0	0	1	1	1	0

Abb.3.3: Stellenwertigkeit der einzelnen Bits (Dezimalzahl 270 rein binär)

#### Programm Ascii08: Bitkombination im Arbeitsspeicher für die Zahl 270

```
public class Ascii08 {  
    public static void main(String args[]) {  
        int zahl = 270;  
        System.out.println(Integer.toBinaryString(zahl));  
    }  
}
```

Für die rein binäre Codierung spielt die ASCII-Tabelle keine Rolle, denn es wird hier nicht ein einzelnes Zeichen, sondern die Zahl als Ganzes verschlüsselt. Aus der Tabelle Abb. 3.3 kann man z.B. ablesen, dass die Dezimalzahl 65 rein binär wie folgt codiert wird: 1000001 - und bitte beachten Sie, dass sie damit genau so codiert wird wie der Großbuchstabe 'A'.

#### Übung

Codieren Sie die Zahl 2 rein binär und vergleichen Sie das Ergebnis mit der vorherigen Übung, bei der die einzelne Ziffer (das Zeichen) '2' verschlüsselt wurde.

Die Interpretation einer Bitkombination hängt also davon ab, ob eine Zahl oder ein Zeichen (*char*) an dieser Stelle erwartet wird. Anders gesagt: Es gibt nicht nur verschiedene Darstellungen eines Wertes (abhängig vom verwendeten Code), sondern die Bitfolgen können auch dadurch eine unterschiedliche Bedeutung bekommen, dass sie unabhängig von einem Code als rein binäre Zahl interpretiert werden.

## 3.6 Unicode

### 3.6.1 Arbeiten mit dem Unicode-Standard

Der Unicode-Standard wurde ursprünglich entworfen als eine feste 16-bit-Verschlüsselung pro Zeichen. Später kamen Ergänzungen auf mehr als 2 Bytes pro Zeichen hinzu, die aktuelle Version Unicode 4 erweitert den Bereich auf 21 bit. Außerdem gibt es historische Varianten, die zwar untereinander kompatibel sind, aber z.T. andere Bezeichnungen haben: Double-Byte-Character-Set, Universal Character Set (UCS), Standard ISO 10646. Ausführliche Informationen zum Thema Unicode finden Sie unter der Adresse

<http://www.unicode.org>.

Der Unicode fasst die Bits zu Gruppen von jeweils 16 (in neueren Versionen bis zu 32) Nullen und Einsen zusammen, um ein Zeichen zu verschlüsseln. **Java benutzt den 16-bit-Unicode.** Dieser bietet  $2^{16}$ , das sind 65.536 unterschiedliche Bitkombinationen. Das ist ausreichend, um alle weltweit gebräuchlichen Zeichen wie Arabisch, Hebräisch, Griechisch, Kyrillisch und auch chinesische, japanische und koreanische Schriftzeichen kodieren zu können. Im Unicode konnte man alle bestehenden Zeichensätze zusammenführen und zusätzlich noch die Codierung von vielen mathema-

tischen und technischen Sonderzeichen standardisieren. Für Internet-Anwendungen ist der Unicode unverzichtbar.

Die Tabelle im Anhang C zeigt die Unicode-Zeichen von 0x0000 - 0x00FF (die ersten 128 Zeichen von 0000 - 007F sind identisch mit dem ASCII-Code). Aufgeführt werden die festgelegten Zeichen und ihre dezimale und hexadezimale Repräsentation (hier "codepoint" genannt) sowie die offiziellen Namen der Zeichen. So hat z.B. das kleine "ä" den Codepoint 00E4, das "ß" den Codepoint 00DF und das Eurozeichen den Codepoint 20AC.

### Wie kommen die Unicodezeichen in den Computer?

Theoretisch ist durch die Einführung des weltweit gültigen Unicodes jedes Kompatibilitätsproblem beim Datenaustausch gelöst. Praktisch gilt dies - jedenfalls derzeit - noch nicht. Denn die Frage, wie die Unicode-Zeichen eigentlich entstehen, ist noch nicht befriedigend gelöst. Die heutigen **Eingabegeräte** und Editoren kennen häufig nur die Byte-Verschlüsselung. Die Tastatur benutzt für die Eingabe den 7-bit-ASCII-Code, ergänzt um nationale Besonderheiten für die nächsten 128 Kombinationen. Das bedeutet, dass beim Eintippen von Zeichen über die Tastatur eine Konvertierung dieser 8-bit-ASCII-Codierung in den Unicode erfolgen muss. Allerdings gibt es bei einigen Editoren die Möglichkeit, mit Hilfe von Umschalttasten (z.B. ALT-C-INSERT) Unicodezeichen einzugeben.

Umgekehrt gibt es auch bei der **Ausgabe** von Unicodezeichen die Notwendigkeit, die Zeichen zu transformieren. Zunächst einmal sind die heutigen Betriebssysteme noch nicht in der Lage, durchgehend mit dem Unicode zu arbeiten, obwohl einige Dateisysteme wie NTFS in Windows und diverse in Linux bereits Unicodezeichen erlauben. Aber die Peripheriegeräte (Drucker, herkömmliche Datei- und Datenbanksysteme auf externen Datenträgern wie Magnetplatten oder DVDs) arbeiten weitgehend noch mit 8-bit-ASCII-Informationen. Also ist auch bei der Ausgabe der Daten eine Umsetzung erforderlich von dem intern in der JVM verwendeten Unicode in den Zeichensatz, den das Betriebssystem verwendet.

Sie werden im Kapitel 6 (Eingabe und Ausgabe) weitere Informationen zu diesem Thema bekommen. Dort gibt es ausführliche Hinweise, wie mit Hilfe von Umsetzungstabellen ("encoding schema") aus ASCII-Daten Unicode-Zeichen transformiert werden und umgekehrt.

### Unicode-Zeichen innerhalb eines Programms verwenden

Die ersten 256 Zeichen des Unicodes sind in der Regel auf der Eingabetastatur verfügbar. Für weitere Sonderzeichen fehlt häufig eine bequeme Eingabemöglichkeit. Eine Möglichkeit, trotzdem Unicodezeichen zu verwenden, besteht darin, diese direkt im Programm zu erzeugen. Dazu sucht man zunächst in der Unicode-Tabelle das entsprechende Symbol und den dazugehörigen hexadezimalen Wert. Die Eingabe erfolgt dann in Form von so genannten "Unicode Escapes". Diese haben die



Form `\uxxxx`, wobei `xxxx` der hexadezimale Wert ist, den das Zeichen im Unicode hat, der Codepoint. Der Präfix `\u` steht für Unicode.

Im nachfolgenden Programm soll ein Unicode-Zeichen bearbeitet werden, das auch mit einem "normalen" Font angezeigt werden kann: das Prozentzeichen `%`. Angenommen, unsere Tastatur enthält dieses Zeichen nicht, dann könnte es im Programm erzeugt werden. Das Zeichen hat im Unicode den Codepoint 0025.

#### Programm *Unicode01*: Unicode-Zeichen im Programm erzeugen

```
public class Unicode01 {
    public static void main(String args[]) {
        char c = '\u0025';
        System.out.println(c);
    }
}
```

Das `%`-Zeichen ist ein ASCII-Zeichen, es liegt im Bereich der ersten 128 Zeichen, und dort ist der Unicode identisch mit dem ASCII-Code. Etwas problematischer wird das Arbeiten mit Zeichen, die zum erweiterten ASCII-Code gehören, also im Bereich 0080 - 00FF liegen. Hier wird ein "Encoding" notwendig, und dafür wird eine Umsetzungstabelle eingesetzt.

Die nachfolgenden Beispiele sind Spezialfälle, die für Einsteiger zunächst weniger Bedeutung haben und deshalb nicht unbedingt am Anfang durchgearbeitet werden müssen. Jedoch wird ein Programmierer in der Praxis immer wieder auf Probleme stoßen, die mit der Codierung von Zeichen ("Encoding") zu tun haben. Insbesondere beim internationalen Datenaustausch, bei Internetanwendungen, beim Arbeiten mit HTML und XML, ist dieses Thema wichtig. Im Bedarfsfall kann dieses Thema also nachgeschlagen werden.

#### 3.6.2 Java-Encodings UTF-8, UTF-16 und ISO-8859-1

Der Unicode selbst ist lediglich eine Tabelle, in der jedes Zeichen eine Platznummer (Codepoint) hat. Damit ist zwar die Bedeutung eindeutig festgelegt, aber der Codepoint sagt noch nichts darüber aus, wie die Zeichen im Computer oder auf einem Datenträger wirklich abgebildet werden. Die einfachste Form wäre es, wenn jedes Zeichen in 2 Bytes (bzw. 3 oder 4 Bytes beim erweiterten Unicode) gespeichert würde. Das ist aber z.B. immer dann uneffektiv, wenn die Daten zum Großteil im Bereich von `\u0000` bis `\u00FF` liegen, weil dafür ein Byte ausreichen würde.

Aus diesem Grund wurden die verschiedenen Versionen des "**Universal Character Set Transformation Format**" (UTF) entworfen. Während Java intern mit dem UTF-16-Format arbeitet, bei dem alle Zeichen in 16 bit verschlüsselt werden, kann für die Auslagerung der Daten auf einem Datenträger ein anderes Format, z.B. UTF-8, ge-

wählt werden. Dieses Format produziert kompakte Dokumente für englischsprachige Texte. Der Transformationsvorgang wird "Encoding" genannt.

Derzeit sind drei Encodings im Einsatz: UTF-8, UTF-16 und UTF-32. Die Zahlen 8, 16 oder 32 geben an, wieviel Bits standardmäßig für die Speicherung eines Zeichens genommen werden.

### UTF-16

Java arbeitet intern ausschließlich mit dem UTF-16-Encoding. Das bedeutet, dass ein (normales) Zeichen innerhalb der JVM mit 16 bit verschlüsselt wird. Wenn das nicht reicht, trifft das System besondere Vorkehrungen, um größere Einheiten zu bilden, d.h. dann wird ein Zeichen (ausnahmsweise) in 3 oder 4 Bytes verschlüsselt und entsprechend gekennzeichnet. Das ist relativ aufwändig, aber man geht davon aus, dass es sich dabei um Ausnahmen handelt.

### UTF-8

Wenn die zu speichernden Zeichen zum Großteil den westeuropäischen Sprachen entstammen, ist der UTF-16-Code natürlich die reinste Platzvergeudung, denn dann reicht 1 Byte pro Zeichen aus. Darum wurde der UTF-8-Standard definiert. Dieses Encoding kann z.B. angegeben werden beim Schreiben von Texten auf externe Datenträger oder beim Austausch mit anderen Systemen. Innerhalb der JVM bleibt es beim UTF-16-Encoding für *char*- und *String*-Typen (siehe Kapitel 4), lediglich für den Datenaustausch kann etwas anderes angegeben werden.

### ISO-8859-1 Latin

Bei der Installation des JDK ist eine Default-Umsetztabelle - abhängig von dem Ländercode - festgelegt worden. In Westeuropa ist dies ISO-8859-1 Latin-Alphabet. Diese Tabelle ist weitgehend identisch mit den ersten 256 Zeichen der Unicode-Tabelle.

Eine Ausnahme ist z.B. das Eurozeichen. Im Default-Characteraset ISO-8859-1 hat das Eurozeichen die Platznummer 80, im Unicode dagegen den Codepoint 20AC. Wir werden nun klären, was passiert, wenn mit diesem Zeichen gearbeitet wird.

### Programm *Unicode02*: Zeichen außerhalb des ASCII-Code

```
public class Unicode02 {  
    public static void main(String args[]) {  
        char c = '\u20AC';  
        System.out.println(c);  
    }  
}
```

Bei der Ausführung dieses Programms z.B. in einer DOS-Box unter MS-Windows wird das Eurozeichen nicht angezeigt. Das liegt daran, dass Microsoft dort mit einem anderen Characteraset arbeitet (mit CP850). Aber wenn Sie die Ausgabe des Programms *Unicode02* umleiten in eine Datei (mit dem Command: *java Unicode02 >*

*a.txt*) und diese Datei dann mit einem hexadezimalen Editor anzeigen, bekommen Sie folgenden Dateiinhalt angezeigt:



Abb. 3.4: Inhalt der Ausgabedaten (umgeleitet in eine Datei)

Die Datei enthält also **80 0D 0A**. Das sind die drei hexadezimalen Platznummern für das Eurozeichen, für CR und für LF im erweiterten ASCII-Code. Damit ist bewiesen, dass eine automatische Umsetzung ("encoding") stattgefunden hat: innerhalb der JVM gilt die Unicode-Verschlüsselung UTF-16, beim Schreiben wurde das Zeichen '\u20AC' umgesetzt in den ASCII-Wert 80 (entsprechend den ISO-8859-1-Codierregeln).

#### Übungen Programm *Unicode02*

**Übung 1:** Starten Sie das Programm erneut aus einer Eingabeaufforderung (DOS-Box). Geben Sie dazu folgenden Befehl ein: `java Unicode02 > test.html`. Dadurch wird die Ausgabe umgeleitet in eine HTML-Datei. Diese können Sie dann in einem Browser anzeigen. Wenn dort als Codierung "Westeuropäisch ISO" gewählt wurde, müsste das Eurozeichen angezeigt werden. Probieren Sie auch eine andere Codierung (im Internet Explorer durch ANSICHT|CODIERUNG).

**Übung 2:** Ändern Sie den Codepoint von \u20AC in \u2030. Damit wird im Unicode das Promille-Zeichen dargestellt. Testen Sie, wie dieses Zeichen von den unterschiedlichen Programmen dargestellt wird.

Nur zur Klarstellung: der Unicode und die Encoding-Tabellen sagen nichts aus über die Art, wie die Zeichen am Bildschirm oder auf Papier dargestellt werden. Dazu werden die Fonts mit ihren unterschiedlichen Schriftarten eingesetzt.

Und zum Nachschlagen hier noch ein Beispiel, wie der UTF-16-Unicode konvertiert werden kann in UTF-8-Code. In dem Programm *Unicode03* werden Informationen des Datentyps *byte* als UTF-8-Code-Informationen interpretiert und in einen String umgewandelt und danach wieder zurück. Dabei wechselt das Encoding von UTF-8 nach UTF-16. Natürlich kann das Programm noch nicht komplett verstanden werden, es dient ausschließlich der Dokumentation und kann später (am Ende des Buches) bei Bedarf als Muster verwendet werden.

**Programm Unicode03: Musterprogramm zum Konvertieren zwischen UTF-16 und UTF-8**

```
public class Unicode03 {
    public static void main (String args[]) throws Exception {

        // ASCII als nach Unicode konvertieren
        byte[] ascii = {'a', 'b', 'c'};
        String str = new String(ascii, "ISO-8859-1");
        System.out.println(str);

        // Unicode (UTF-16) nach UTF-8 konvertieren
        byte[] utf8 = str.getBytes("UTF8");
        for (int i=0; i<utf8.length; i++)
            System.out.println((char)utf8[i]);
    }
}
```

**3.6.3 Sind alle Kompatibilitätsprobleme durch Unicode gelöst?**

Wenn alle Computersysteme und Programme mit dem Unicode arbeiten würden und auch alle vorhandenen Datenbestände im Unicode gespeichert wären: ja. Solange es jedoch noch ASCII-Informationen gibt: nein. Für die Interpretation von ASCII-Informationen ist es erforderlich, den bei der Kodierung benutzten Standard zu kennen (z.B. ISO 8859-1). Deswegen erlauben Technologien, die den Unicode benutzen (wie XML oder Java) die Angabe einer Codepage. Dieses "Encoding" spielt eine entscheidende Rolle bei internationalen Anwendungen und beim Datenaustausch. In Java gibt es eine ganze Reihe von Klassen, die sich mit der "Lokalisierung", also auch mit der Auswahl der Encoding-Tabelle, befassen.

Aber die Art der Codierung (ASCII, EBCDIC, Unicode) und die Darstellung der Zeichensätze durch verschiedene Fonts sind nicht die einzigen Hindernisse beim Datenaustausch zwischen den Computern.

**Folgende Fragen sind zusätzlich zu klären, damit eine fehlerfreie Verständigung zwischen Partnern, die sich auf verschiedenen Plattformen befinden, möglich ist:**

- Wie ist die "Byte-Order", also die Anordnung der bits innerhalb eines Bytes. Damit ist gemeint, wie ist die Wertigkeit der Bits vereinbart: stehen die hochwertigen bits links oder beginnt man rechts mit der Interpretation? Hierfür gibt es die Fachausdrücke "little endian" und "big endian". Welches Verfahren gewählt wird, hängt u.a. ab von der Hardware des Prozessors.
- Wenn Dateien ausgetauscht werden, sind Informationen über das Dateiformat notwendig. Je nach Betriebssystem gibt es Stream-orientierte Dateikonzepte, wo

die Interpretation eventueller Steuerzeichen durch das Programm erfolgen muss, oder Record-/Block-Formate, bei denen das Betriebssystem die Struktur erkennt.

- Sollen programmiersprachen-interne Datentypen wie z.B. Integer- oder Floating-Point-Daten (siehe Kapitel Datentypen) ausgetauscht werden? Diese sind nur in Ausnahmefällen kompatibel. So gibt es zwar z.B. in Java *und* in C++ den eingebauten Datentyp *int*, doch nur in Java ist genormt, wieviel Bytes für solche Zahlen vorgesehen sind.
- Werden Textdaten, Informationen aus Datenbanken, objektorientierten Anwendungen ("serialisierte Instanzen") oder Multimediadaten wie Töne oder Bilder ausgetauscht? Allein die Darstellung eines Zeilenwechsels in Textdateien wird unterschiedlich codiert: in MS-Windows durch die ASCII-CR/LF-Zeichen, in Unix nur durch ein LF-Zeichen und im Apple-System u.U. nur durch ein CR-Zeichen.

#### Zusammenfassung

Im Arbeitsspeicher und auf externen Datenspeichern eines Computers finden sich niemals Zeichen oder Dezimalzahlen, sondern immer [nur Bitmuster](#). Diese werden interpretiert anhand von Codetabellen. Diese Tabellen enthalten Regeln für die Zuordnung von Zeichen zu binären Zahlen. Jedem Zeichen ist eine eindeutige Platznummer zugewiesen, die dann binär verschlüsselt wird. Es gibt verschiedene Codetabellen: ASCII, EBCDIC oder Unicode.

In Java werden die Werte von [char- und String-Variablen intern als Unicode](#) (UTF-16-Encoding) gespeichert. Auch Literale (siehe Kapitel 5) können Unicode-Zeichen enthalten. Außerdem können Kommentare und Identifier aus Unicode-Zeichen zusammengesetzt sein.

Da sowohl die Dateisysteme als auch die Ein- und Ausgabegeräte sowie die Programmierertools heute noch nicht durchgängig mit Unicode arbeiten, ist immer dann, wenn die Daten die "Java Virtuelle Maschine" (JVM) verlassen, eine Konvertierung nötig. Das gleiche gilt für das Einlesen der Daten in die JVM.

Die Transformation von Bytecodierungen in Unicode ist nur möglich, wenn zusätzliche Informationen (über die verwendete Codepage) zur Verfügung stehen.

Der Quelltext eines Java-Programmes kann Unicode enthalten, entweder direkt mit geeigneten Editoren eingegeben oder als so genannte Escape-Sequenz `\uxxxx`, wobei `xxxx` die Codepoint-Zahl des Zeichens ist.

---

# 4

## Klassen und andere Typen beschreiben ("declaration")

---

Java ist eine typisierte Programmiersprache. Das bedeutet, dass die zu verarbeiten den Daten, die im Arbeitsspeicher stehen, zu einem bestimmten Datentyp gehören müssen. Dies wird vom Compiler überprüft, z.B. bei einer Wertezuweisung an den Speicherplatz.

In diesem Kapitel erfahren Sie,

- welche Bedeutung die Datentypen haben,
- wie der Datentyp für einen Arbeitsspeicherplatz festgelegt wird,
- welche Datentypen es gibt,
- wodurch sich primitive Datentypen von Referenzdatentypen unterscheiden.

In einer objektorientierten Sprache wie Java werden die unterschiedlichen Datentypen in **Klassen** beschrieben. In den Klassen ist definiert, aus welchen Datenfeldern sich der Datentyp zusammensetzt und welche Verarbeitungsmöglichkeiten es dafür gibt. Viele dieser Klassen sind integrierter Bestandteil der Java-Sprache ("Standard-Klassen"), darüber hinaus kann der Programmierer selbst neue Klassen erstellen, d.h. der Programmierer kann seine Typen selbst definieren. Wenn der Programmierer mit diesen Typen arbeiten will, muss er **Objekte** im Speicher erzeugen.

Für diese Erläuterungen haben wir zwei Begriffe benutzt, die fundamental sind für jede objektorientierte Programmiersprache: **Klassen und Objekte**. Häufig wird gesagt: In Java ist alles ein Objekt. Oder: Alles, was in Java geschieht, ist in Klassen codiert. Die Abgrenzung zwischen den beiden Begriffen ist jedoch manchmal etwas unscharf, und ganz falsch ist es, wenn man sie als Synonyme benutzt.

Wir werden uns auch mit diesen Themen befassen und in diesem Kapitel eine erste Einführung geben in die Bedeutung von Klassen als benutzerdefinierte Datentypen.

Im nächsten Kapitel 5 werden wir uns dann ausführlich mit dem Erzeugen von Objekten (und einfachen Variablen) im Arbeitsspeicher befassen, bevor dann im Kapitel 8 detailliert darauf eingegangen wird, wie Sie eigene Klassen erstellen und benutzen können.

Ein Schwerpunkt dieses Kapitels ist die Abgrenzung zwischen primitiven Datentypen und Referenztypen.

Sie werden mit Referenztypen arbeiten und die acht primitiven Typen, die in die Java-Sprache eingebaut sind, mit vielen Beispielen kennen lernen.

## 4.1 Deklarationsanweisung

Damit Daten von Programmen verarbeitet werden können, ist es zwingend erforderlich, dass sich diese im internen Arbeitsspeicher ("Hauptspeicher", "RAM") befinden, andernfalls ist eine Verarbeitung mit Java-Befehlen nicht möglich. Das bedeutet, dass diese Daten vorher von einem externen Speicher gelesen oder über die Tastatur eingegeben werden müssen. Dem Interpreter müssen dann die Position der Daten im Arbeitsspeicher, ihre Struktur, d.h. die interne Darstellung der Daten, und ihre Verarbeitungsmöglichkeiten, bekannt sein. Und genau dafür gibt es die Deklarationsanweisung in Java.

Durch die Datenbeschreibung ("declaration", Vereinbarung) wird dem Compiler bekannt gemacht, welche Daten das Programm verarbeitet, wie sie aufgebaut sind, welche Verarbeitungsmöglichkeiten es für sie gibt, und wie sie referenziert, d.h. adressiert, werden. In Java sieht die Deklarationsanweisung ganz allgemein wie folgt aus:

```
<datentyp> <identifizier>
```

Beispiel für eine Datendeklaration:

```
String    text;
```

Mit dieser Anweisung wird dem Compiler mitgeteilt, dass vom Datentyp *String* ein Objekt benötigt wird, das unter dem Bezeichner *text* adressiert werden kann. Der Bezeichner *text* wird auch als Referenzvariable bezeichnet, denn er kann eine Referenz auf ein bestimmtes (*String*-)Objekt enthalten.

Der Datentyp *String* ist in einer mitgelieferten Klasse der Standard-Bibliothek exakt beschrieben. Dort steht, woraus dieser Datentyp besteht (ein *String* ist aus einzelnen Zeichen zusammengesetzt) und welche Verarbeitungsmöglichkeiten dem Programmierer zur Verfügung stehen (z.B. kann man *String* ausgeben, vergleichen, kopieren und konkatenieren, also zwei Strings zusammenfassen zu einem einzigen usw.).

Eine Besonderheit ist, dass bei der Deklaration mehrere Bezeichner angegeben werden können, z.B.

```
double gehalt, stundenlohn, pension, rente;
```

Mit dieser Deklaration werden vier Bezeichner bekannt gemacht, dabei ist der Datentyp nur einmal angegeben. In der Aufzählung werden die Bezeichner, wie für Aufzählungen in Java generell üblich, durch Komma getrennt. Für jede dieser vier Variablen gilt, dass der Datentyp *double* ist. Allerdings wird diese Schreibweise nicht empfohlen, weil die Lesbarkeit des Quelltextes dadurch leidet.

Die Deklarationsanweisung kann zusätzlich eine Klausel zum [Initialisieren](#) enthalten. Diese Klausel (der "Initializer") besteht aus dem Zuweisungsoperator = und einem Ausdruck, der den Anfangswert spezifiziert, z.B.

```
int zahl = 4700;
```

## 4.2 Was ist der Datentyp?

Jede Variable hat einen Datentyp. Im nachfolgenden Programm wird eine Stringvariable (ein Objekt) angelegt und verarbeitet. Dazu muss in der Deklaration der Name der Klasse *String* angegeben werden.

### Programm Deklaration01: Variable vom Typ *String* anlegen und benutzen

```
public class Deklaration01 {
    public static void main(String[] args) {
        String text;           //Referenzvariable erzeugen
        text = new String("Hallo "); //Objekt erzeugen
        System.out.println(text); //Objektwert ausgeben
    }
}
```

### 4.2.1 Warum unterschiedliche Datentypen?

Der Datentyp beschreibt

- mit welchen Operationen (Methoden) die Variablen bearbeitet werden können, (evtl. gibt es auch gleichnamige Operationen mit unterschiedlicher Wirkung, abhängig vom Datentyp),
- die Struktur der Objekte (wie setzen sich die Daten im Arbeitsspeicher zusammen und wie groß ist der Speicherbereich?).

Der Compiler überprüft, ob der Einsatz der Daten entsprechend ihrem Typ erfolgt, z.B. beim Initialisieren, beim Einlesen vom externen Speicher, bei einer Wertezuweisung, beim Methodenaufruf und bei der Parameterübergabe. Dies wollen wir in den nachfolgenden Beispielen praktisch überprüfen.

#### a) Der Datentyp legt die Verarbeitungsmöglichkeiten fest

##### Übung 1 zum Programm Deklaration01

Bitte überprüfen Sie, wie der Compiler reagiert, wenn Sie versuchen, in dem Programm mit der *String*-Variablen zu rechnen. Fügen Sie hierzu unmittelbar vor dem Ausgabebefehl folgende Zeile ein: `text = text - 5;`

Der Umwandlungsfehler wird lauten: "operator - cannot be applied to String", denn durch den Datentyp wird der Befehlsvorrat (die Menge der möglichen Operationen) für eine Variable festgelegt. Das Minuszeichen ist für *String*-Variable nicht erlaubt.

#### b) Der Datentyp legt die Bedeutung von Operationen fest

##### Übung 2 zum Programm Deklaration01

Bitte überprüfen Sie, wie der Compiler reagiert, wenn Sie versuchen, den folgenden Befehl einzufügen: `text = text + 5;`



Die Umwandlung wird fehlerfrei durchgeführt. Wenn Sie das Programm danach zur Ausführung starten, wird auf der Console ausgegeben: "Hallo 5". Offensichtlich hat der Compiler das Plus-Zeichen (+) nicht als arithmetischen Befehl für eine Addition interpretiert, sondern er hat die beiden Werte "Hallo " und 5 verkettet (concateniert) und dann ausgegeben. Abhängig vom Datentyp führt die JVM also die richtige Aktion aus.

### Übung 3 zum Programm Deklaration01

Bitte überprüfen Sie, wie die Ausgabe lautet, wenn Sie im Ausgabebefehl zwei numerische Werte mit dem Plus-Zeichen verbinden, z.B. `println(5 + 3);`

#### c) Der Datentyp bestimmt die Art der Datenwerte (den "Wertebereich")

Nicht jeder Datentyp basiert auf einer Klassenbeschreibung. Einige einfache ("primitive") Datentypen sind in die Sprache eingebaut und können vom Programmierer benutzt werden, ohne dass eine Klassenbeschreibung als Grundlage vorliegt.

Für jeden eingebauten Datentyp ist exakt festgelegt, wieviel Speicherplatz für ihn reserviert wird. So hat z.B. eine Variable vom Datentyp *char* eine Größe von zwei Bytes (wegen Unicode), und sie kann genau *ein* einzelnes Zeichen aufnehmen.

### Programm Deklaration02: Variable vom Typ *char*

```
public class Deklaration02 {  
    public static void main(String[] args) {  
        char c = 'A';  
        System.out.println(c);  
    }  
}
```

In der dritten Zeile wird die Variable *c* vom Datentyp *char* definiert **und** gleichzeitig initialisiert, d.h. es wird ihr ein Anfangswert zugewiesen.

### Übung 1 zum Programm Deklaration02

Testen Sie die Reaktion des Compilers, wenn Sie versuchen, die Variable *c* mit den beiden Zeichen 'AB' zu initialisieren.

Es kommt die Fehlermeldung: "unclosed character literal", weil nach dem ersten Zeichen das abschließende Hochkomma fehlt. Eine Variable vom Datentyp *char* kann nur ein einzelnes Zeichen speichern.

#### d) Der Compiler überwacht die Verarbeitung typisierter Daten

Vor einem Zugriff auf den Wert einer Variablen überprüft der Compiler (soweit möglich), ob die Variable deklariert ist und ob ein korrekter Inhalt vorhanden ist. So muss vor der ersten Benutzung ein passender Wert zugewiesen sein, entweder durch eine Initialisierung oder durch eine Wertezuweisung.

## Übung 2 zum Programm Deklaration02

Testen Sie, ob eine fehlerfreie Umwandlung möglich ist, wenn Sie die Initialisierung in der dritten Zeile entfernen bzw. mit `//` als Kommentar kennzeichnen.

**Fazit:** Datentypen haben folgende Aufgaben:

- Verringerung der Programmierfehler. Mögliche Fehler sollten entdeckt werden, bevor ein Programm zur Verarbeitung an die JVM übergeben wird. Also sollte der Compiler bereits verhindern, dass Speicherbereiche falsch verwendet werden.
- Vermeidung von Fehlern während der Ausführung (weil spätestens die Run-Time-Umgebung die normale Fortführung eines Programms verhindern wird, wenn für Daten nicht erlaubte Operationen aufgerufen werden).
- Problemgerechte Anpassung von internen Objekten. Durch die Deklaration kann eine Optimierung der internen Repräsentation der Daten erfolgen.
- Die Unterscheidung von Datentypen je nach Aufgabenstellung erleichtert dem Menschen das Verständnis und die Überschaubarkeit von Quellenprogrammen.

### 4.2.2 Welche Datentypen gibt es?

Java unterscheidet grundlegend zwischen zwei Arten von Datentypen:

- **Referenztypen** (Klassentypen), die jeweils in eigenen Klassendateien beschrieben sind. Es gibt eine Fülle von Klassen, die als Bestandteil des JDK mitgeliefert werden ("Standardklassen"). Es können aber zusätzlich vom Programmierer beliebig viele neue Klassen erstellt werden ("benutzerdefinierte Datentypen"), dadurch wird der Sprachumfang von Java beliebig erweitert.
- **Primitive Datentypen**, die in die Sprache eingebaut sind und ohne weitere Vorkehrungen genutzt werden können. Java kennt 4 Arten von primitiven Datentypen:
  - die Integer-Typen *int*, *byte*, *long*, *short* für Ganzzahlen
  - die Gleitkomma-Typen *float* und *double* für gebrochene Zahlen
  - den logischen Typ *boolean* für Wahrheitswerte
  - den *char*-Typ für einzelne Zeichen.

Die primitiven Typen werden auch "einfache Datentypen" genannt, weil sie nur jeweils genau einen Wert aufnehmen können. Dagegen sind die Referenztypen zusammengesetzte (aggregierende) Typen, die auch mehrere Werte enthalten können.

Auf den grundlegenden Unterschied zwischen eingebauten Datentypen und den Klassentypen werden wir in diesem Buch immer wieder eingehen.

Eine zusammenfassende Übersicht der Unterschiede finden Sie im nächsten Kapitel.

### 4.3 Referenztypen

Normalerweise sind in Java die Datentypen in Klassen beschrieben, d.h. der Bauplan für die Zusammensetzung und die Verarbeitungsmöglichkeiten von Objekten steht in Klassen. Die Ausnahme bilden acht Datentypen, die in die Sprache eingebaut sind.

Durch den Klassentyp wird beschrieben,

- wie sich der Datenteil eines Objekts zusammensetzt (welche Attribute, d.h. welche Felder mit welchen Typen, werden benötigt, um ein konkretes Objekt zu beschreiben?)
- welche Methoden können auf diesen Datenteil angewendet werden (welche Fähigkeiten hat das Objekt?)

Zum Sprachumfang von Java gehören einige Tausend Klassen, die der Programmierer für die Deklaration von Objekten benutzen kann, z.B. die Klasse *String*, die Klasse *Date* oder die Klasse *Point*.

Die Syntax für die Deklaration von Referenzvariablen unterscheidet sich nicht von der Deklaration einer primitiven Variablen. In beiden Fällen besteht die Deklaration in der einfachsten Form aus dem Datentyp und dem Namen. So kann eine primitive Variable wie folgt definiert werden:

```
char buchstabe;
```

Dadurch wird der Bezeichner *buchstabe* vereinbart **und** ein passender Speicherbereich (in diesem Fall 2 Bytes) bereitgestellt.

Eine Referenzvariable kann wie folgt definiert werden:

```
String text;
```

Dadurch wird der Bezeichner *text* dem Compiler bekannt gemacht und ein "passender" Speicherbereich zur Verfügung gestellt. Aber es gibt einen gravierenden Unterschied zu primitiven Objekten: der "passende" Speicherplatz ist nicht der Bereich für das eigentliche Objekt, sondern lediglich der Speicherplatz für die Referenz auf das Objekt. Das eigentliche Objekt wird durch diese Definition noch nicht angelegt. Dazu ist ein besonderes Schlüsselwort erforderlich: *new*.

```
String text = new String("ABC");
```

Mit *new* wird

- Speicherplatz für ein neues Objekt im Arbeitsspeicher angelegt,
- die Anfangsadresse dieses Platzes in die Referenzvariable übertragen und
- dieser Platz mit standardisierten oder individuellen Werten initialisiert.

Am Beispiel des bereits bekannten Programms *Deklaration01* wollen wir jetzt genauer klären, wie die Abläufe im Arbeitsspeicher (RAM) sind, wenn mit Referenzvariablen gearbeitet wird.

**Programm Deklaration01: Referenzvariable und Objekt**

```

public class Deklaration01 {
    public static void main(String[] args) {
        String text;           //Referenzvariable erzeugen
        text = new String("Hallo "); // Objekt erzeugen
        System.out.println(text); // Objektwert ausgeben
    }
}

```

Beim Starten dieses Programms geschieht im Speicher folgendes:

Im ersten Schritt wird ein Platz reserviert für die Referenzvariable *text*. Diese Variable ist vom Typ *String*, d.h. sie kann auf ein Objekt mit den Attributen und Verarbeitungsmöglichkeiten, die in der Klasse *String* beschrieben sind, verweisen. Durch diese Definition hat die Referenzvariable *text* aber **noch keinen Inhalt** ("Wert").

Im zweiten Schritt wird mit dem Schlüsselwort *new* das Stringobjekt selbst erzeugt. Außerdem bekommt das Objekt einen Anfangswert, nämlich "Hallo" zugewiesen. Und - ganz wichtig! - die Speicheradresse des Objekts wird in die Referenzvariable *text* übertragen. Die folgende grafische Darstellung soll dies verdeutlichen:

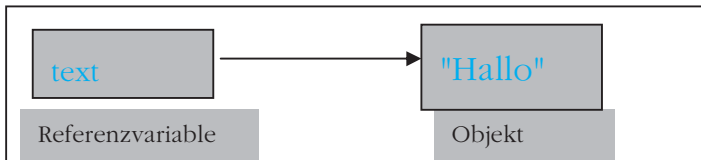


Abb. 4.1: Objekt im Arbeitsspeicher

**Objekte werden durch Referenzvariable manipuliert**

Im dritten Schritt wird mit dem Objekt gearbeitet. Der Wert wird am Bildschirm ausgegeben. Dabei wird der Name (Bezeichner, Identifier) der Referenzvariablen benutzt (das Objekt selbst hat keinen Namen).

**Übung zum Programm Deklaration01**

Modifizieren Sie das Programm so, dass eine weitere Referenzvariable mit dem Identifier *name* angelegt wird. Das Objekt soll vom Typ *String* sein. Der Anfangswert soll aus Ihrem Namen bestehen. Danach wird über *System.out* der Wert des Objekts ausgegeben.

**Fassen wir zusammen:**

In der objektorientierten Programmierung werden zunächst Objekte im Arbeitsspeicher erzeugt, bevor sie durch Methodenaufrufe ("Nachrichten") manipuliert werden können.

Basis für die Objekterzeugung ist eine Klasse. Eine Klasse kann wie ein neuer, selbst definierter Datentyp gesehen werden. Dort ist für eine Gruppe von Objekten beschrieben, aus welchen Daten sich die Objekte zusammensetzen und welche Operationen ("Methoden") damit ausgeführt werden können. Die Objekterzeugung erfolgt zur Laufzeit des Programms, dazu sind **zwei Schritte erforderlich**:

- **Deklaration der Referenzvariablen.** Hierzu gehört, dass der Programmierer festlegt, welchen Datentyp das Objekt haben soll und unter welchem Namen das Objekt referenziert werden soll. Um den Datentyp des Objekts festzulegen, wird bei der Deklaration der Name der entsprechenden Klasse angegeben.
- **Anlegen des Speicherplatzes** im Arbeitsspeicher und Initialisieren dieses Platzes für das Objekt. Dazu wird das Schlüsselwort *new* benutzt. Man sagt auch: Es wird eine Instanz der Klasse erzeugt. Als Ergebnis dieses Vorganges wird die Anfangsadresse der Instanz in die Referenzvariable übertragen.

Die Verarbeitungsmöglichkeiten für die Instanz (das Objekt) sind in der Klasse, die beim Deklarieren der Referenzvariablen angegeben wurde, festgelegt. Ein Lesen oder Ändern der Objektinhalte kann nur erfolgen durch Methodenaufrufe (mit Hilfe der Referenzvariablen).

### 4.4 Spezialfall: Primitive Datentypen

Das beschriebene Vorgehen, Daten als Objekte im Arbeitsspeicher zu deklarieren und mit Methoden zu verarbeiten, ist relativ aufwändig. Deshalb haben die Java-Entwickler entschieden, für bestimmte Grund-Datentypen, die immer wieder benötigt werden, ein vereinfachtes Verfahren zu verwenden. In der Sprachreferenz sind insgesamt acht Datentypen festgelegt, die in die Sprache "eingebaut" sind. Sie können benutzt werden, ohne den Weg über eine Klasse mit Referenzvariablen und Methodenaufrufe gehen zu müssen. Diese eingebauten Datentypen werden auch "primitive" oder "einfache" Typen genannt.

Durch den primitiven Datentyp wird für eine einzelne Variable festgelegt:

- die Anzahl der zu reservierenden Speicherstellen,
- der "Wertebereich" für die Daten, die dort gespeichert werden können,
- die erlaubten Operationen für diese Speicherstellen.

Die Deklaration legt noch ein weiteres Merkmal der Variablen fest: den "Scope", das ist der Gültigkeitsbereich, der beschreibt, wo diese Variable bekannt ist und genutzt werden kann, z.B. nur innerhalb einer Methode (lokale Variable) oder von allen Methoden (weitere Hinweise dazu siehe Kapitel 16).

Sie lernen in den folgenden Abschnitten, welche einfachen Typen es gibt, wie deren Deklaration erfolgt und wie mit ihnen gearbeitet werden kann.

#### 4.4.1 Deklaration mit primitiven Datentypen

Einfache Variable sind keine Objekte; sie werden direkt referenziert, d.h. der vereinbarte Identifier ist ein Platzhalter für den Wert der Variablen. Ganz anders dagegen die Deklaration eines Objekts: Der Identifier für ein Objekt enthält lediglich eine Adresse als Verweis auf die eigentlichen Objektfelder. Technisch ausgedrückt: In einem Maschinenbefehl gibt es keine symbolischen Namen, sondern nur echte Speicheradressen. So wird aus dem Identifier einer einfachen Variablen die Adresse für den tatsächlichen Wert, aber aus dem Identifier einer Referenzvariablen wird die Adresse für die Adresse des Objekts.

Für die Manipulation des Speicherplatzes gibt es so genannte Operatoren. Ein häufig benutzter Operator ist der Zuweisungsoperator, das Gleichheitszeichen `=`. Damit wird der Variablen ein neuer Wert zugewiesen. Die möglichen Operatoren werden ausführlich im Kapitel 7 besprochen. Das folgende Programm zeigt, wie eine Variable vom einfachen Typ deklariert und danach (per Wertezuweisung) mit einem Wert gefüllt wird.

##### Programm Deklaration03: Einfache Datentypen deklarieren und verarbeiten

```
public class Deklaration03 {
    public static void main(String[] args) {
        int zahl1;
        zahl1 = 100;
        System.out.println(zahl1);
    }
}
```

Mit der Anweisung `int zahl1;` wird die Variable `zahl1` deklariert. Der Compiler reserviert den benötigten Speicherplatz, abhängig vom Datentyp. In diesem Beispiel ist als Typ `int` angegeben, die Abkürzung für Integer, engl. Ganzzahl. Die Sprachspezifikation von Java legt fest, dass für den Datentyp `int` immer 4 Bytes im Speicher angelegt werden. Und die interne Repräsentation der Werte dieses Objekts ist damit auch festgelegt - nämlich rein binär. Somit wird die Dezimalzahl 100 im Arbeitsspeicher wie folgt dargestellt (ohne Berücksichtigung des Vorzeichens):

0000 0000	0000 0000	0000 0000	0110 0100
-----------	-----------	-----------	-----------

Die obige Bitkombination könnte vom System auch als der kleine Buchstabe 'd' interpretiert werden. Denn dieser Buchstabe hat die Platznummer 100 im Unicode. Damit der Speicherinhalt als Zeichen (und nicht als Zahl) interpretiert wird, müsste dann bei der Deklaration als Datentyp `char` (für Character, engl. Zeichen) angegeben werden.

##### Übung zum Programm Deklaration03

Bitte ändern Sie den Datentyp von `int` auf `char` und testen Sie das Programm.

Die eingebauten Datentypen sind das Fundament einer Programmiersprache. Deswegen werden wir - bevor wir uns in den nachfolgenden Kapiteln mit selbst definierten Datentypen, also mit den Klassen, befassen - zunächst die acht Basisdatentypen besprechen.

Bei der Deklaration einer einfachen Variablen wird der Speicherplatz für einen möglichen Wert gleich reserviert. Bei der Deklaration einer Referenzvariablen dagegen wird lediglich der Platz für die Adresse reserviert, danach ist ein zweiter Schritt, die Instanziierung mit *new*, erforderlich, damit ein Objekt entsteht. Hierzu ein Hinweis: In einigen anderen Programmiersprachen wird zwischen Deklaration und Definition streng unterschieden. Als Deklaration wird der Vorgang bezeichnet, der einen neuen Bezeichner (z.B. für eine Variable) bekannt macht, und unter Definition wird dann nicht nur die Deklaration, sondern auch die Speicherallokation für die Variable verstanden. Java unterscheidet nicht zwischen diesen beiden Begriffen.

Auf den nächsten Seiten wird jeder Datentyp detailliert besprochen. Die Beispielprogramme haben immer den gleichen Aufbau. Damit Sie ein Verständnis für die Struktur und die wichtigsten Sprachelemente dieser immer gleich aufgebauten Programme bekommen, erläutern wir zunächst ein Musterprogramm.

#### Programm Typen01: Ein typisches EDV-Programm sieht in Java so aus:

```
import java.util.Scanner;
public class Typen01 {
    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        int zahl;
        zahl = eingabe.nextInt();
        zahl = zahl + 50;
        System.out.println("Ergebnis ist: " + zahl);
    }
}
```

Das Programm hat die typische Struktur vieler EDV-Programme: Eingabe, Verarbeitung, Ausgabe. Zunächst werden die benötigten Ressourcen bekannt gemacht ("deklariert"), dann werden Daten **eingelesen**, im Arbeitsspeicher **verarbeitet** und die Ergebnisse **ausgegeben**. Damit das Beispielprogramm nicht zu umfangreich wird, enthält es keinerlei Prüfungen, auch keine Formalprüfungen, die sicherstellen, dass der Bediener auch wirklich nur Daten des geforderten Typs eingeben kann. Diese formale Prüfung sollte selbstverständlich in einem praxisrelevanten Programm vorhanden sein.

Im Einzelnen enthält das Programm folgende Anweisungen:

- Die *import*-Anweisung in Zeile 1 enthält Hinweise darauf, wo bereits fertig codierte Klassen zu finden sind, die in diesem Programm benutzt werden. Durch diese Angabe wird dem Programm bekannt gemacht, dass aus dem Paket *ja-*

*va.util* die Klasse *Scanner* benötigt wird. Weitere Hinweise zu Packages und der *import*-Anweisung folgen später.

- Danach wird [ab Zeile 2](#) die Klasse beschrieben. In diesem Fall besteht die Klasse aus nur einer Methode, nämlich aus der Methode *main()*. Die Klasse hat den Namen *Typen01*, sie endet nach der letzten geschweiften Klammer.
- [Ab Zeile 3](#) ist die Methode *main()* definiert. Die Definition einer Methode besteht grundsätzlich aus den beiden Teilen Methodenkopf und Methodenblock. Im Methodenkopf werden die Eigenschaften der Methode (Name, Zugriffsrechte, Parameter ...) beschrieben. Der Methodenblock enthält die eigentlichen Ausführungsanweisungen, er ist eingefasst in geschweiften Klammern und enthält in diesem Beispiel die folgenden fünf Anweisungen:

- Definition des Eingabegeräts:

```
Scanner eingabe = new Scanner(System.in);
```

- Definition des Speicherplatzes für eine Variable:

```
int zahl;
```

- Einlesen eines Wertes vom definierten Eingabegerät:

```
zahl = eingabe.nextInt();
```

- Veränderung des Variableninhalts durch Addition:

```
zahl = zahl + 50;
```

- Ausgabe des Ergebniswertes:

```
System.out.println("Ergebnis ist: " + zahl);
```

### Übungen zum Programm *Typen01*

**Übung 1:** Starten Sie das Programm. Die Ausführung des Programms stoppt an der Stelle, wo die Eingabe einer Zahl erwartet wird. Geben Sie eine ganze Zahl ein, z.B. 25. Klären Sie, in welcher Zeile die Verarbeitung so lange unterbrochen wird, bis der Bediener die Eingabetaste ("Enter") drückt.

**Übung 2:** Rufen Sie das Programm erneut zur Ausführung auf. Klären Sie, was passiert, wenn Sie keine Ganzzahl eingeben, sondern z.B. eine Kommazahl oder einen Buchstaben.

Uns interessiert an dieser Stelle besonders die Deklaration des Speicherplatzes für eine primitive Variable. Dies geschieht mit der folgenden Anweisung:

```
int zahl;
```

Das Schlüsselwort *int* legt den Datentyp fest, danach wird der Identifier (Bezeichner) für die Variable definiert. Der Datentyp *int* ist einer von **acht "eingebauten" (oder elementaren, einfachen oder primitiven) Datentypen**. Einfache Datentypen gehören zum Java-Sprachumfang, sie werden über Schlüsselwörter (keywords) benutzt.



### 4.4.2 Die Datentypen für Ganzzahlen

Eine ganze Zahl ist eine Zahl ohne Kommastellen. Sie kann positiv oder negativ sein. Ganze Zahlen werden in Java normalerweise rein binär gespeichert, d.h. man verlässt die Stellenwertigkeit des Dezimalsystems und codiert die Zahl mit der Stellenwertigkeit 2. Die Unicode-Tabelle wird nicht benötigt.

Java kennt vier Ganzzahlen-Typen: *byte*, *short*, *int* und *long*. Die jeweils höchste (bzw. niedrigste) Zahl hängt ab von der Anzahl der bits, die für den Datentyp vorgesehen sind. Dabei wird in jedem Fall auch das Vorzeichen gespeichert (das hat zur Folge, dass sich der Wertebereich halbiert!) - es gibt in Java keine vorzeichenlose Ganzzahlentypen.

Datentyp	Anzahl bits	Anzahl Bytes	Wertebereich von:	Wertebereich bis:
<b>byte</b>	8	1	127	-127
<b>short</b>	16	2	32767	-32768
<b>int</b>	32	4	2.147.483.647	-2.147.483.647
<b>long</b>	64	8	9223372 Mrd.	-9223372 Mrd.

Abb. 4.2: Übersicht der ganzzahligen Datentypen

#### 4.4.2.1 Der Datentyp *int*

Der wichtigste Ganzzahlentyp ist *int*, das ist die Abkürzung für Integer, engl. ganze Zahl. Damit wird festgelegt, dass dieser Speicherplatz **nur** für die Speicherung von ganzen Zahlen genutzt werden kann, dass dafür immer 4 Bytes im Arbeitsspeicher zur Verfügung stehen, unabhängig von der Plattform und von der Maschinenarchitektur, und dass die 32 bits rein binär (mit der Stellenwertigkeit von 2er Potenzen) interpretiert werden. Weiterhin ist durch den Datentyp festgelegt, welche Operationen mit dieser Variablen erlaubt sind (z.B. rechnen, aber kein konkatenieren).

#### Programm *Integer01*: Arbeiten mit dem Datentyp *int*

```
import java.util.Scanner;
public class Integer01 {
    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        int zahl; // Deklaration
        zahl = eingabe.nextInt(); // Einlesen
        System.out.println("Eingelesen wurde: " + zahl);
    }
}
```

### Übungen zum Programm Integer01

**Übung 1:** Bitte testen Sie das Programm, indem Sie zunächst einen beliebigen Integerwert eingeben. Dann ist alles in Ordnung.

**Übung 2:** Testen Sie danach mit einem nicht-numerischen Wert (z.B. mit Buchstaben) oder mit einem gebrochenen Anteil, entweder in amerikanischer Schreibweise, bei der anstelle des Komma ein Punkt eingetippt wird (z.B. 15.20) oder auch die deutsche Schreibweise. Dann erzeugt die JVM einen Laufzeitfehler ("Exception").

**Übung 3:** Testen Sie jetzt das Programm, indem Sie eine negative Ganzzahl eingeben (Achtung: das Vorzeichen muss vor der Zahl stehen, also z.B. -125).

#### 4.4.2.2 Der Datentyp *byte*

Das nächste Programm erwartet vom Bediener Werte vom Datentyp *byte*. Dies ist ebenfalls ein numerischer Datentyp. Wie die Tabelle 4.2 zeigt, ist der höchste Wert in einer Byte-Variablen 127. Erläuterung: Eine *byte*-Variable enthält immer ein Vorzeichenbit, deswegen stehen für die Zahl selbst nur 7 bits zur Verfügung. Das gibt 128 Möglichkeiten, beginnend mit der Dezimalzahl 0, maximal dann 127 positiv oder minimal 127 negativ.

#### Programm *Byte01*: Arbeiten mit dem Datentyp *byte*

```
import java.util.Scanner;

public class Byte01 {
    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        byte zahl; // Deklaration
        zahl = eingabe.nextByte(); // Eingabe
        System.out.println("Eingegeben wurde: " + zahl);
    }
}
```

#### Übung zum Programm *Byte01*

**Übung 1:** Testen Sie das Programm zunächst mit einer korrekten numerischen Eingabe (z.B. 28) und danach mit einem Wert, der außerhalb des Wertebereichs einer *byte*-Variablen liegt (z.B. 128 oder einem nicht-numerischen Wert).

**Übung 2:** Erlaubt dieser Datentyp die Eingabe einer negativen Zahl (z.B. -5)?

**Übung 3:** Versuchen Sie selbstständig, einen *long*-Datentyp zu benutzen. Lösungshinweise: Dazu ist in Zeile 5 der Datentyp auf *long* zu ändern, und in der Zeile 6 lautet der Lesebefehl:

```
    zahl = eingabe.nextLong();
```

### 4.4.3 Die Datentypen für die Gleitkomma-Zahlen (floating-point)

Zahlen mit Nachkomma-Stellen werden als Gleitkommazahlen bezeichnet. Java kennt die beiden Typen *float* und *double*. Die Art der Verschlüsselung (Codierung) von Kommazahlen ist im IEEE-Standard 754 normiert - natürlich wiederum völlig unabhängig von der Unicode-Verschlüsselung, denn es sollen keine einzelnen Zeichen, sondern Zahlenwerte als Ganzes codiert werden.

Bei der Gleitkommadarstellung wird die Dezimalzahl in zwei Teile zerlegt:

- in eine Festkommazahl (Mantisse), diese kann sich aus einer Vor- und Nachkommazahl zusammensetzen und
- in die Angabe einer Zehnerpotenz, mit der diese Zahl multipliziert wird. Dieser Exponent zur Basis 10 bezeichnet die Anzahl Stellen, um die das Komma nach rechts oder links verschoben wird (also "gleitet").

#### Allgemeine Formel:

$$\begin{aligned} \text{zahl} &= x * 10^y \\ x &= \text{Mantisse} && (= \text{die Zahl selbst}) \\ y &= \text{Exponent zur Basis 10} && (= \text{Anzahl Gleitstellen}). \end{aligned}$$

#### Beispiele:

$$\begin{aligned} 10000 &= 1 \text{ (Mantisse)} \quad 5 \text{ (Exponent)} \\ 325000 &= 325 \text{ (Mantisse)} \quad 3 \text{ (Exponent)} \\ 0.0001 &= 1 \text{ (Mantisse)} \quad -4 \text{ (Exponent)} \end{aligned}$$

Der Exponent gibt also an, um wieviel Stellen das Komma verschoben werden soll. Ist er positiv, gleitet das Komma nach rechts; ist er negativ, gleitet das Komma nach links.

In Computerprogrammen (und auch in Java) benutzt man häufig die so genannte normalisierte (oder wissenschaftliche) Schreibweise, bei der vor den Exponent der Buchstabe E (oder e) geschrieben wird:

$$\begin{aligned} 10e7 &= 100000000 \\ 10E-7 &= 0,0000010, \end{aligned}$$

Diese beiden Beispiele zeigen einen der Vorteile der Gleitkomma-Darstellung: die Schreibweise kann bei großen oder sehr kleinen Zahlen sehr kompakt sein. Der Wertebereich ist entsprechend hoch: für *double*-Typen ist die größte Zahl 1.8E308.

Die Aufteilung in Mantisse und Exponent ist wahlfrei und kann unterschiedlich gewählt werden. So kann die Dezimalzahl 37,5900 z.B. wie folgt dargestellt werden:

$$\begin{aligned} 3759 &\text{ als Mantisse und } -2 \text{ als Exponent, also: } 3759E-2 && \text{ oder} \\ 37590 &\text{ als Mantisse und } -3 \text{ als Exponent, also: } 37590E-3 && \text{ oder} \\ 3,759 &\text{ als Mantisse und } 1 \text{ als Exponent, also: } 3,759e1. \end{aligned}$$

Java kennt zwei unterschiedliche Gleitkomma-Typen, *float* und *double*. Sie unterscheiden sich in der Anzahl Speicherstellen, die zur Verfügung stehen:

- *float* stellt 32 bits zur Verfügung (davon 7-8 signifikante Dezimalstellen) und
- *double* hat doppelt so viele, nämlich 64 bits (davon 14 signifikante Dezimalstellen).

Das folgende Programm speichert den Eingabewert in einer Variablen vom Datentyp *double*. Damit ist es für diese Variable in der Lage, gebrochene Zahlen (mit Kommastellen) zu lesen und zu verarbeiten.

#### Programm Double01: Arbeiten mit dem Datentyp *double*

```
import java.util.Scanner;
public class Double01 {
    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        double zahl; // Deklaration
        zahl = eingabe.nextDouble(); // Einlesen
        System.out.println("Eingegeben wurde: " + zahl);
    }
}
```

#### Übungen zum Programm Double01

**Übung 1:** Für die Eingabe von Gleitkommazahlen gibt es mehrere Möglichkeiten: entweder wird die Standarddarstellung gewählt (z.B. 5,34) oder es wird die wissenschaftliche Notation verwendet (z.B. 7E5). Bitte testen Sie beide Varianten. Achtung: Das Programm ist angepasst an die deutsche Schreibweise, d.h. es wird ein Komma erwartet und kein Dezimalpunkt.

**Übung 2:** Ändern Sie im Quelltext den Datentyp der Variablen *zahl* in *float*. Lösungshinweis: Denken Sie daran, dass sich auch der Methodenname ändert - von *nextDouble* auf *nextFloat*. Testen Sie das Programm mit den folgenden vier Gleitkommazahlen: -5E2, 302e-2, 3,00E8, 2,0.

Die Gleitkomma-Darstellung hat neben dem Vorteil der kompakten Darstellung (durch Beseitigung von überflüssigen Nullen) noch den zusätzlichen Vorteil, dass die meisten Prozessoren schneller damit arbeiten können als mit binären Ganzzahlen oder mit binär codierten Dezimalzahlen (diese so genannten BCD-Zahlen werden wir später genauer behandeln).

Aber Achtung: Durch den Wechsel des Stellenwertsystems (vom Zehnerrechnen zur Zweierdarstellung) können sich Ungenauigkeiten ergeben. Der Grund ist die begrenzte Stellenanzahl, die für Bruchzahlen zur Verfügung steht. Zwar ist das Binärsystem nicht mehr oder weniger genau als das Dezimalsystem, aber es entspricht eben nicht unserer Denkweise. Im Kapitel 7 gibt es dazu weitere Hinweise.

### 4.4.4 Der Datentyp *boolean*

Eine Aussage kann wahr oder falsch sein. Um einen dieser beiden "Wahrheitswerte" speichern zu können, ist in Java der Datentyp *boolean* eingebaut. Dieser Typ kennt also nur diese beiden Werte, dies ist sein Wertebereich. Die beiden möglichen Wahrheitswerte werden *false* und *true* genannt.

#### Programm *Boolean01*: Arbeiten mit dem Datentyp *boolean*

```
public class Boolean01 {
    public static void main (String args[]) {
        boolean aussage = false;
        System.out.println("Aussage ist: " + aussage);
        System.out.println("d.h. sie ist nicht: " + !aussage);
    }
}
```

Das Programm *Boolean01.java* erzeugt eine Boolesche Variable und initialisiert sie mit dem Anfangswert *false*. Dieser wird im ersten Ausgabebefehl unverändert ausgegeben, im zweiten *println*-Aufruf jedoch wird der Wahrheitswert durch Verwendung des ! (Ausrufezeichen) umgekehrt (aus *false* wird *true*).

#### Programm *Boolean02*: Einlesen eines Wahrheitswertes

```
import java.io.*;
import java.util.Scanner;
public class Boolean02 {
    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        boolean wahr;
        wahr = eingabe.nextBoolean(); // Eingabe
        System.out.println("Ergebnis ist: " + wahr); // Ausgabe
    }
}
```

#### Übung zum Programm *Boolean02*

Testen Sie das Programm, indem Sie jeweils einmal folgende Werte eingeben: "true", "false" und "TRUE" und "FaLse".

Man sagt, *true* und *false* sind eingebaute Literale, und sie müssen exakt so geschrieben werden. Aber: die Groß- oder Kleinschreibung spielt **ausnahmsweise** keine Rolle, denn die Methode *nextBoolean()* ist so geschrieben, dass sie den eingegebenen String automatisch in Kleinbuchstaben umwandelt.

Eingesetzt wird der *boolean*-Datentyp für die Speicherung von Zuständen z.B. nach Vergleichen. Boolesche Variablen können so als Schalter dienen für das Festhalten von Vergleichsergebnissen, von denen dann weitere Abläufe abhängig gemacht werden. Die Steuerbefehle (siehe Kapitel 8) arbeiten alle mit booleschen Werten.

### 4.4.5 Der Datentyp *char*

Der Zeichentyp *char* ist der einzige primitive Datentyp, der mit einer vereinbarten Codierungstabelle arbeitet, d.h. während die Bitkombinationen in den vier Ganzzahldatentypen als Dezimalzahlen interpretiert werden, wird der Inhalt einer *char*-Variablen als ein einzelnes Zeichen interpretiert. In Java wird intern für jedes Zeichen aus dem Unicode die vereinbarte Platznummer (codepoint) gespeichert, wenn der Datentyp *char* vereinbart wird. Im Unicode ist für jedes Zeichen die Codierung festgelegt (anders als bei Zahlen, wo nicht die Ziffernzeichen einzeln verschlüsselt werden, sondern wo der Zahlenwert als Ganzes umgerechnet wird in eine Dualzahl mit der Stellenwertigkeit 2).

Für jedes Zeichen werden in Java zwei Bytes belegt. Vorzeichen gibt es beim Datentyp *char* nicht. Damit liegt der hexadezimale Wertebereich des Datentyps *char* zwischen `\u0000` (niedrigster Wert) und `\uFFFF` (höchster Wert).

Die direkte Eingabe eines Unicodezeichens ist über die Tastatur (*System.in*) nicht so ohne weiteres möglich, denn wo gibt es Tastaturen mit einigen Tausend Unicodezeichen? Deshalb haben die Java-Entwickler entschieden, dass einem Programm zunächst nur der numerische Wert des Zeichens übergeben wird, wenn es von der Tastatur liest. Konkret wird dem Programm der Codepoint aus der Unicode-Tabelle geliefert. Dieser Wert muss zunächst in einer *int*-Variablen empfangen werden und kann dann je nach Bedarf interpretiert werden. Diese Vorgehensweise gilt auch für herkömmliche ASCII-Zeichen, die im Unicode als Untermenge enthalten sind.

#### Programm *Character01*: Arbeiten mit dem Datentyp *char*

```
public class Character01 {
    public static void main(String[] args) throws Exception {
        int z;
        z = System.in.read(); // Liefert den Codepoint
        char c = (char)z;      // Interpretation als (Unicode-)Zeichen
        System.out.println("Ergebnis ist: " + c);
    }
}
```

#### Übung zum Programm *Character01*

Testen Sie das Programm, indem Sie ein beliebiges Zeichen des ASCII-Codes (z.B. den Buchstaben A oder a per Tastatur eingeben. Überprüfen Sie, wie das Programm reagiert, wenn Sie ein Zeichen des erweiterten ASCII-Code (z.B. einen deutschen Umlaut) eingeben. Offensichtlich ist dies nicht so ohne weiteres möglich. Sie werden im Kapitel 6 Lösungen für dieses Problem kennen lernen.

Der Kopf der Methode enthält die Klausel **"throws Exception"**. Damit wird dokumentiert, dass innerhalb der Methode ein Laufzeitfehler ("Exception") auftreten kann, der nicht abgefangen und speziell behandelt wird, sondern "unbehandelt" durchge-

reicht wird an den Aufrufer dieser Methode. Da es sich in diesem Fall um die *main*-Methode handelt, wird ein eventuell auftretender Laufzeitfehler von der JVM bearbeitet (normalerweise bedeutet dies Programmabbruch). Wenn die Klausel fehlt, führt das zu folgendem Umwandlungsfehler:

```
...unreported exception java.io.IOException; must be caught  
or declared to be thrown...
```

Auf der Basis von einzelnen Zeichen (*char*-Typ) können weitere Texttypen gebildet werden. So gibt es in Java die Klassen *String* und *StringBuffer*, die ein komfortables Arbeiten mit Zeichenketten ("strings") ermöglichen.

### 4.4.6 Wrapper-Klassen

In Java gibt es für jeden primitiven Datentyp eine korrespondierende Klasse, also für den Typ *char* die Klasse *Character*, für den Typ *byte* die Klasse *Byte* usw. Die Klassen werden Wrapper-Klassen genannt. Sie können anstelle der einfachen Datentypen benutzt werden.

Notwendig kann der Einsatz von Wrapper-Klassen sein, wenn Objektvariable benötigt werden (also einfache Datentypen nicht erlaubt sind, z.B. bei Methodenaufrufen) oder wenn zusätzliche Verarbeitungsmöglichkeiten genutzt werden sollen.

#### Programm Wrapper01: Einen Integerwert als Objekt erzeugen

```
public class Wrapper01 {  
    public static void main (String[] args)  {  
        Integer zahl = new Integer(25);  
        System.out.println(zahl + 3);  
    }  
}
```

Das Programm *Wrapper02* zeigt, wie mit Hilfe einer Methode der Wrapper-Klasse *Character* abgefragt werden kann, ob es sich bei einem Zeichen um einen Großbuchstaben handelt.

#### Programm Wrapper02: Methode *isUpperCase()* aus Klasse *Character*

```
public class Wrapper02  {  
    public static void main(String[] args)    {  
        char zeichen1 = 65;  
        if (Character.isUpperCase(zeichen1))  
            System.out.println("Es ist ein Großbuchstabe");  
    }  
}
```

## 4.5 Zusammenfassung

### Deklarationsanweisungen benötigen immer die Angabe des Datentyps

Die Deklaration von einfachen Variablen sieht genau so aus wie die Deklaration von Klassentypen:

```
<datentyp> <identifizier>.
```

Beispiele:

```
int zahl;           // primitiver Datentyp
String text;        // Referenz-Datentyp
```

### Vorteile des Typkonzepts

- Die Programmsicherheit wird erhöht (weil der Compiler die korrekte Verwendung der Speicherplätze überprüft).
- Ressourcen-Optimierungen sind möglich (der Speicherbedarf und die optimale Verarbeitungsform kann anhand des Typs ermittelt werden).
- Verständlichkeit, weil Absicht und Wirkung der Verarbeitung von Daten deutlicher wird. Weil z.B. Methodensignaturen explizit den Datentyp enthalten, wird ihre Bedeutung klarer.

Je differenzierter die Datentypen in einer Programmiersprache sind, umso umfangreicher können die eingebauten Prüfungen schon zur Umwandlungszeit vorgenommen werden. Aufgabe des Programmierers ist es, den Datentyp auszuwählen, der von der Aufgabenstellung her am engsten passt.

Java ist eine typisierte und objektorientierte Programmiersprache. Prinzipiell sind die Daten, die verarbeitet werden sollen, allesamt **Objekte**. Das heißt, sie werden zusammen mit ihren Verarbeitungsmethoden beschrieben in Klassen, von denen dann zur Ausführungszeit Referenz-Variablen (Instanzen) erzeugt werden. Nur die in der Klasse vorgesehenen Methoden können dann mit diesen Daten arbeiten.

Ein Spezialfall sind die primitiven Datentypen, die aus Vereinfachungsgründen bereits in die Sprache "eingebaut" worden sind. Sie sind keine Objekte und können mit vordefinierten Schlüsselwörtern benutzt werden (*int*, *char*, *boolean* usw.).

### Warum gibt es primitive Datentypen?

- Die *Größe* eines primitiven Typs wird von Java bestimmt, sie ist unabhängig von der Hardware-Architektur und vom Betriebssystem.. Auch der *Aufbau* im Speicher ist auf allen Plattformen gleich: Java benutzt grundsätzlich das Big-Endian-Format. Damit wird gewährleistet, dass Java-Programme portabel sind.



- Ihre Nutzung ist für den Programmierer einfacher und für die JVM effizienter als das Arbeiten mit Referenz-Variablen, weil z.B. keine Methoden aufgerufen werden.

#### Welche primitiven Datentypen gibt es?

Es gibt eingebaute Datentypen für ganze Zahlen, für reelle Zahlen (mit Kommastellen), für einzelne Unicode-Zeichen und für logische Werte. Im Einzelnen sind dies die folgenden acht Datentypen:

- vier ganzzahlige Typen (*byte*, *short*, *int*, *long*), die sich nur unterscheiden durch die Länge des reservierten Speicherplatzes (1, 2, 4 oder 8 Byte),
- zwei reale Datentypen (die Gleitkommazahlen *float* oder *double*), die sich in der Genauigkeit, das heißt, in der Anzahl der Stellen vor und hinter dem Komma, unterscheiden,
- Boolescher Datentyp (*boolean*), der in der Lage ist, die Wahrheitswerte *true* oder *false* zu speichern,
- Zeichentyp (*char*), der jedes einzelne Zeichen anhand der Unicode-Tabelle, d.h. in 2 Bytes, also 16 bits, verschlüsselt.

#### Welche Regeln gelten für primitive Datentypen?

- Der Platzbedarf im Arbeitsspeicher ist durch die Sprachspezifikation festgelegt, unabhängig von der Hardwareplattform und von der Betriebssystemumgebung (das ist *ein* Grund für die Kompatibilität von Javaprogrammen).
- Damit ist auch der "Wertebereich" (der höchste und niedrigste darstellbare Wert in diesem Speicherplatz) festgelegt. Der Wertebereich ergibt sich durch  $2^n$ , wobei  $n$  die Anzahl der bits ist, die zur Verfügung stehen.
- Allerdings wird evtl. 1 bit benötigt für die Vorzeichendarstellung. Alle numerischen Typen werden mit Vorzeichen gespeichert (außer *char* und *boolean* sind alle Typen numerisch).

Referenztypen wie Arrays und Strings oder beliebige andere benutzerdefinierte Datentypen ("Klassen") werden aus diesen atomaren Bausteinen zusammengesetzt.

# 5

## RAM verwalten: Variable und Objekte erzeugen

---

Wenn Daten von einem Programm angesprochen und verarbeitet werden sollen, dann müssen sich diese Daten zwingend im RAM (Random Access Memory), d.h. im internen Arbeitsspeicher (Hauptspeicher), befinden. Daten auf einem anderen Medium sind nicht direkt manipulierbar. Also ist es notwendig, die Daten entweder einzulesen oder die Daten als Teil des Quellcodes direkt im Programm zu kodieren (als "Litereale"). Für die Aufnahme der eingelesenen Daten und für die Verwaltung des Arbeitsspeichers steht ein Konzept zur Verfügung, das als *Variablenkonzept* bezeichnet wird. Mit diesem Thema befassen wir uns in diesem Kapitel.

Im Kapitel 4 haben wir erläutert, welche Möglichkeiten der Typisierung der Speicherplätze es gibt, wie dadurch eine unterschiedliche Repräsentation und eine optimierte Behandlung möglich wird und vor allem, wie dadurch die Sicherheit der Verarbeitung erhöht wird. In diesem Kapitel lernen Sie,

- was Variablen und Konstanten sind,
- wie sie im Arbeitsspeicher erzeugt und verarbeitet werden,
- warum es wichtig ist, dabei zwischen Objekten und einfachen Variablen zu unterscheiden,
- wie Literale im Programm benutzt werden und welche alternativen Darstellungen möglich sind,
- was Escape-Sequenzen sind und welche Arten es gibt.

### 5.1 Was sind Variablen?

Variablen sind Arbeitsspeicherplätze, die typisiert sind und einen Namen ("identifier") haben. Sie werden durch eine Deklarationsanweisung im Programm definiert, dadurch bekommt jede Variable einen Datentyp und einen Identifier:

```
int zahl;           oder
String text;
```

Java unterscheidet streng zwischen Variablen, die

- einen primitiven Datentyp haben ("**primitive Variablen**") und Variablen, deren
- Datentyp eine Klasse ist ("**Referenz-Variablen**").

Die Unterscheidung zwischen diesen beiden Arten von Variablen ist von grundlegender Bedeutung, und wir werden beide Arten ausführlich besprechen.

## 5.2 Primitive Variablen

### 5.2.1 Deklaration von primitiven Variablen

Bevor ein Speicherplatz in Java benutzt werden kann, muss dafür eine Deklaration ausgeführt worden sein. Eine Variablendeklaration enthält immer mindestens zwei Angaben: den Datentyp und den Namen der Variablen. Primitive Variablen haben einen der acht eingebauten Basis-Datentypen. Sie sind keine Objekte, die explizit mit *new* erzeugt werden, sondern sie werden im Speicher angelegt mit Ausführung der Definition.

Der Datentyp einer Variablen bestimmt

- wieviel Speicherplatz für die Variable belegt werden soll,
- welchen Wert eine Variable enthalten (speichern) kann und
- welche Operationen damit ausgeführt werden können.

Speicherplätze, die als Variablen definiert werden, können ihren Inhalt (ihren "Wert") während der Laufzeit eines Programmes beliebig oft ändern ("die Inhalte sind variabel"). Hiervon gibt es Ausnahmen, die wir später besprechen werden: es gibt "konstante Variablen", deren Inhalt einmal festgelegt wird und dann unveränderlich ist, und es gibt "immutable" Datentypen, deren Inhalt nur scheinbar änderbar ist.

Somit bestehen einfache Variablen aus:

- einem **Namen** (Bezeichner, Identifier), z.B. *zahl*,
- einem der eingebauten acht einfachen **Datentypen** (z.B. *int*),
- einem **Inhalt** ("Wert"), z.B. 28 oder ' A ' .

Der **Name** und der **Datentyp** wird durch eine Vereinbarungsanweisung (Deklaration) vom Programmierer festgelegt. Mit dem Namen wird die Variable adressiert, dieser Bezeichner ist im Quelltext der Stellvertreter für den aktuellen Inhalt.

Beispiel:

```
double gehalt;
```

Ein **Wert** muss vor dem ersten Ansprechen dieser Variablen vorhanden sein. Er kann als so genannter Initialwert entweder automatisch durch Java oder bei der Deklaration bzw. durch Wertezuweisung vom Programmierer festgelegt worden sein.

Danach kann der Wert der Variablen beliebig oft geändert werden. Man sagt, es wird ein neuer Wert *zugewiesen*, und das bedeutet, dass der bisherige Inhalt komplett überschrieben wird. Diese "Wertzuzuweisung" verändert also den Inhalt einer Variablen. Beim Lesen der Variablen bleibt der Inhalt unverändert bestehen, es wird lediglich eine Kopie zum Empfänger des Lesebefehls transportiert.

### 5.2.2 Initialisierung von primitiven Variablen

Beispiel einer Deklaration inklusive einer Initialisierung:

```
double gehalt = 2000.00;
```

Mit dieser Deklaration wird dem Programm bekannt gemacht, dass ein Speicherbereich reserviert werden soll, der 64 bit groß ist und Gleitkommazahlen speichern kann. Der Anfangswert ist 2000,00. Sein aktueller Wert kann unter dem Namen *gehalt* gelesen oder verändert werden.

#### Programm *Variablen01*: Deklaration von Variablen

```
public class Variablen01 {  
    public static void main(String[] args) {  
        char zeichen;  
        double gehalt;  
        boolean vorhanden;  
    }  
}
```

#### Übungen zum Programm *Variablen01*

**Übung 1:** Wandeln Sie das Programm um und führen Sie es aus. Formal ist alles in Ordnung, aber es passiert natürlich nichts. Ändern Sie das Programm so, dass der Wert der Variablen *gehalt* auf *System.out* ausgegeben wird, und versuchen Sie, das Programm umzuwandeln. Die Fehlermeldung lautet sinngemäß: Die Variable ist nicht initialisiert.

**Übung 2:** Ändern Sie also das Programm erneut, indem Sie Ihr Wunschgehalt als Initial-Wert in die Variable schreiben.

#### Namen (identifier) für Variablen

Die Namen sind vom Programmierer frei wählbar. Ausgeschlossen sind allerdings die Schlüsselwörter der Sprache, denn diese haben eine festgelegte Bedeutung. Die Länge ist beliebig, als Zeichen ist (fast) jedes Unicodezeichen erlaubt, nur die erste Stelle muss ein Buchstabe sein.

Empfehlungen für die Namensvergabe: Der Identifier sollte möglichst aussagefähig sein. Damit es keine Probleme beim Anzeigen oder Ausdrucken des Quelltexts gibt, sollten die Bezeichner keine Zeichen außerhalb des reinen ASCII-Codes enthalten. Auch deutsche Umlaute sind zu vermeiden.

Abhängig von der Position, wo eine Variable deklariert ist, unterscheidet man noch die "Scope". Das ist die Sichtbarkeit der Deklaration, d.h. sie beschreibt, wo die Variable genutzt werden kann und wo nicht. Dies ist ein Thema, das im Kapitel 16 erläutert wird.

### 5.2.3 Wertezuweisung an Variablen

Der Inhalt einer Variablen kann z.B. durch folgende Wertezuweisung überschrieben werden:

```
gehalt = 3000.00;
```

#### Übung zum Programm *Variablen01*

Codieren Sie eine Wertezuweisung für die Variable *gehalt*. Das neue Gehalt soll 2100 sein. Testen Sie diese Änderung.

Variablen müssen einen Wert haben, bevor sie zum ersten Mal in einem Statement angesprochen werden können. Wenn eine Variable nicht Teil einer Klasse ist, sondern innerhalb einer Methode deklariert worden ist, so handelt es sich um eine lokale Variable. Lokale Variablen müssen explizit vom Programmierer einen Initialwert bekommen, bevor sie genutzt werden können. (Variablen, die Mitglieder einer Klasse sind, werden außerhalb von Methoden deklariert, und diese Variablen werden automatisch initialisiert. Dazu in späteren Kapiteln mehr.)

Im Programm *Variablen01* sind alle drei Variablen lokal. Deshalb müssen sie explizit einen Wert als Anfangswert bekommen. Das kann erfolgen

- entweder bereits bei der Deklaration, indem das Gleichheitszeichen und dahinter der gewünschte Initialwert eingetragen werden,
- oder später zur Laufzeit des Programms, durch eine Zuweisung, mit der ein Wert in die Variable geschrieben wird.

#### Programm *Variablen02*: Initialisieren von lokalen Variablen

```
public class Variablen02 {  
    public static void main(String[] args) {  
        char zeichen = 'x';  
        double gehalt = 2000.00;  
        boolean vorhanden = true;  
        System.out.println(vorhanden);  
    }  
}
```

#### Übung zum Programm *Variablen02*

Ändern Sie das Programm wie folgt: Führen Sie **keine** Initialisierung bei der Deklaration aus, sondern codieren Sie stattdessen eine Wertezuweisung.

Die Ausgabe des Programms verändert sich dadurch nicht, sondern nur das Verhalten: jetzt wird der Wert nicht bereits zur Umwandlungszeit eingetragen, sondern erst zur Laufzeit. Bei jeder Wertezuweisung überprüft der Compiler, ob der gesendete Wert kompatibel ist zu dem Datentyp der Empfängervariablen ("Zuweisungskompatibilität").

**Lösungsvorschlag: Programm Variablen03: Wertezuweisung an Variablen**

```
public class Variablen03 {  
    public static void main(String[] args) {  
        char zeichen;  
        double gehalt;  
        boolean vorhanden;  
        zeichen = 'x';  
        gehalt = 2000;  
        vorhanden = true;  
        System.out.println(vorhanden);  
    }  
}
```

Der Variablenname steht für den Wert der Variablen. Das heißt, wenn in einem Befehl der Name der Variablen verwendet wird, so wird der Wert der Variablen dafür eingesetzt und damit gearbeitet. Beispiel:

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

Durch diese Anweisung wird der aktuelle Inhalt des Speicherbereichs mit dem Identifier *vorhanden* ermittelt und auf der Standardausgabereinheit ausgegeben. Man sagt, der Ausdruck *vorhanden* wird ausgewertet ("evaluiert"), und mit dem Ergebnis wird dann die gewünschte Operation (hier: *println*) ausgeführt.

Das folgende Programm *Tauschen01* soll den Inhalt (die Werte) der Variablen *z1* und *z2* tauschen und danach am Bildschirm ausgegeben.

**Programm Tauschen01: Tauschen der Werte von 2 Variablen**

```
import java.util.Scanner;  
public class Tauschen01 {  
    public static void main(String[] args) {  
        Scanner eingabe = new Scanner(System.in);  
        int z1;  
        int z2;  
        System.out.println("Bitte 2 Zahlen eingeben");  
        z1 = eingabe.nextInt();  
        z2 = eingabe.nextInt();  
        System.out.println("Zahl1 = " + z1 + " Zahl2 = " + z2);  
        int hilfsvARIABLE = z1;  
        z1 = z2;  
        z2 = hilfsvARIABLE;  
        System.out.println("Zahl1 = " + z1 + " Zahl2 = " + z2);  
    }  
}
```

Die Besonderheit in diesem Programm ist, dass eine Hilfsvariable definiert werden muss, die temporär einen Wert zwischenspeichert.

### Übung zum Programm *Tauschen01*

Bitte notieren Sie auf einem Blatt Papier, welche Arbeitsspeicherplätze definiert werden, wie groß diese sind und wie sich deren Inhalte zur Programmlaufzeit verändern.

## 5.3 Referenzvariablen

### 5.3.1 Deklaration einer Referenzvariablen

Bei der Deklaration einer Variablen kann als Datentyp nicht nur einer der acht eingebauten Datentypen angegeben werden, sondern dort kann als Typ auch ein Klassenname stehen. Dabei kann es sich eine der mitgelieferten Klassen handeln (aus der Standardbibliothek der J2SE) oder um eine Klasse, die zugekauft oder selbst geschrieben wurde. Damit ist die Sprache Java beliebig erweiterbar - das Hinzufügen von Klassen entspricht einem Hinzufügen von weiteren Datentypen mit ihren Verarbeitungsmöglichkeiten.

Wir haben bereits mehrfach mit der Standardklasse *String* gearbeitet, z.B.

```
String text;
```

Durch diese Deklarationsanweisung wird eine Variable deklariert, die die Fähigkeit hat, auf ein (beliebiges) Stringobjekt zu verweisen. Die Variable *text* wird auch Referenzvariable genannt, weil sie eine Referenz auf ein Objekt enthalten soll. Eine Referenz ist eine Adresse auf den Anfang der Objektwerte. Aber noch enthält diese Variable keinen Wert, denn es existiert noch kein Stringobjekt.

### Programm *String01*: Referenzvariable deklarieren

```
public class String01 {  
    public static void main(String[] args) {  
        String text;  
    }  
}
```

### Übung 1 zum Programm *String01*

Ergänzen Sie das Programm um einen Ausgabebefehl, der den Inhalt der Variablen *text* am Bildschirm ausgeben soll. Versuchen Sie eine Umwandlung und interpretieren Sie die Fehlermeldung.

Der Grund für die Fehlermeldung ist: Die Variable *text* soll ein Objekt referenzieren, aber dieses Objekt ist im Speicher noch gar nicht vorhanden.

### 5.3.2 Objekt erzeugen ("Instanzieren") und initialisieren

Wir haben bereits in früheren Programmen gesehen, dass ein Objekt mit dem Schlüsselwort *new* erzeugt wird. Hinter dem Schlüsselwort *new* wird der Klassename wiederholt, gefolgt von runden Klammern, in denen ein Anfangswert für das Objekt angegeben sein kann:

```
text = new String("Merker");
```

Durch diese "Instanziierung" wird der Speicherplatz für das Objekt zur Verfügung gestellt. Und es wird auch gleich ein Initialwert hinein geschrieben ("Merker").

#### Übung 2 zum Programm *String01*

Ergänzen Sie das Programm um ein Statement, das ein weiteres Objekt der Klasse *String* erzeugt. Dabei soll gleichzeitig als (Anfangs-)Text Ihr Name an das Objekt übergeben werden. Der Identifier ist frei wählbar. Danach soll zuerst das neue und dann das Objekt *text* ausgegeben werden.

Das folgende Programm *String02* benutzt wiederum die Klassenbeschreibung der mitgelieferten Klasse *String*. In dieser Klasse ist vorprogrammiert,

- woraus eine Zeichenkette besteht (nämlich aus einer Folge von einzelnen *char*-Zeichen) und
- wie damit gearbeitet werden kann (welche Methoden zur Ausführung aufgerufen werden können, z.B. zum Vergleichen, Zusammenfügen oder Aufteilen von Strings).

Das Programm hat die Aufgabe, zwei Zeichenketten zu erstellen und diese dann zu *einer* Zeichenkette zu verbinden ("konkatenerieren"). Dafür gibt es die Methode *concat*. Das Ergebnis soll am Konsolbildschirm ausgegeben werden.

#### Programm *String02*: Instanzen aus mitgelieferten Klassen erzeugen

```
public class String02 {
    public static void main(String[] args) {
        String str1 = new String("Java");
        String str2 = new String("buch");
        String ergebnis;
        ergebnis = str1.concat(str2);
        System.out.println(ergebnis);
    }
}
```

Die Zeile

```
String str1 = new String("Java");
```



fasst drei Vorgänge zusammen. Es ist die Kurzschreibweise für:

```
String str1;  
str1 = new String("Java");
```

Zunächst wird eine Referenzvariable mit dem Bezeichner *str1* deklariert, und danach wird das eigentliche Stringobjekt erzeugt (durch das Schlüsselwort *new*). Und drittens wird dieses Objekt mit dem Initialwert "Java" belegt.

### Übung zum Programm *String02*

Das Programm erzeugt insgesamt drei Objekte. Bitte identifizieren Sie die drei Objekte. Notieren Sie die Speicher-Belegung dafür auf einem Blatt Papier. Überlegen Sie, wann und in welchen Schritten die benötigten Plätze angelegt werden. Zeichnen Sie auch den Inhalt (den "Wert") ein, den die Objekte im Laufe der Programmausführung haben.

### Genereller Hinweis zum Arbeiten mit String-Objekten

In jeder Programmiersprache werden Datentypen, mit denen Texte (Zeichenketten, "Strings") beschrieben werden können, sehr häufig benötigt. Dies gilt natürlich auch für Java. Deswegen haben die Entwickler der Sprache eine Vereinfachung speziell für das Arbeiten mit *String*-Objekten ermöglicht. So können *String*-Variable im Arbeitsspeicher erzeugt werden, ohne explizit das Schlüsselwort *new* zu benutzen. Entweder kann die Objekterzeugung bereits zusammen mit der Deklaration der Referenzvariablen erfolgen:

```
String text = "Hallo";
```

oder später in einer separaten Wertezuweisung:

```
String text;  
...  
text = "Hallo";
```

Wir haben in den bisherigen Beispielen zur Objekterzeugung allerdings auch für Strings das Schlüsselwort *new* benutzt, um zu demonstrieren, wie die Instanzerzeugung normalerweise, d.h. bei allen anderen Klassen, zu erfolgen hat.

### Beispiel für den Datentyp *Point*

Das nächste Programm verwendet den Datentyp *Point*. Diese Standardklasse beschreibt zwei Integerfelder *x* und *y* (zur Speicherung einer Position in einer Koordinate). Als Verarbeitungsmöglichkeiten für diesen Datentyp sind etwa zehn Methoden vorprogrammiert, u.a. auch die Methode *toString()*.

### Programm *Point01*: Instanz der Klasse *Point* erzeugen und ausgeben

```
import java.awt.*;  
class Point01 {
```

```
public static void main(String[] args) {  
    Point p;  
    p = new Point(5,3);  
    System.out.println(p.toString());  
}
```

Das Programm *Point01* definiert die Referenzvariable *p*, erzeugt mit *new* ein Objekt vom Typ *Point* mit den Werten 5 und 3 und ruft anschließend die Methode *toString* auf, damit die Werte über *System.out* als String angezeigt werden.

### 5.3.3 Wertzuweisung bei Referenztypen

Wenn eine Referenzvariable verarbeitet werden soll, muss sie als Wert entweder *null* enthalten (wenn für sie noch kein Objekt existiert) oder eine Referenz, um ein Objekt zu erreichen. Dabei können durchaus mehrere Referenzvariable auf dasselbe Objekt verweisen.

#### Programm Point02: Zwei Referenzen, aber nur 1 Objekt

```
import java.awt.*;  
public class Point02 {  
    public static void main(String[] args) {  
        Point p1 = new Point(5,3);  
        Point p2;  
        p2 = p1;           // Wertezuweisung  
        System.out.println(p2.toString());  
    }  
}
```

So wie primitiven Variablen ein Wert zugewiesen werden kann, so ist dies auch für Referenzvariablen erlaubt. Allerdings gilt für beide Variablenarten: die Werte müssen kompatibel sein, d.h. sie müssen zu dem Empfänger passen. Nicht erlaubt z.B. ist die Zuweisung eines primitiven Typs an einen Referenztyp, denn diese Variablen sind nicht **zuweisungsverträglich**. Einer Referenzvariablen kann nur ein Wert zugewiesen werden, der dem Klassentyp entspricht, mit dem sie definiert worden ist.

#### Übung zum Programm Point02

**Übung 1:** Überlegen Sie (zunächst theoretisch) die Auswirkung folgender Programmänderung: Weisen Sie der Variablen *p1* unmittelbar vor dem Ausgabebefehl *null* zu (durch: "*p1* = null;"). Kann *p2* trotzdem korrekt ausgegeben werden? Überprüfen Sie dies praktisch.

**Übung 2:** Versuchen Sie danach, die Werte des Objekts *p1* auszugeben. Lösungshinweis: Es müsste folgende Fehlermeldung kommen: "Exception in thread "main" java.lang.NullPointerException".

Im Programm *Point02* wurde der Wert von *p1* der Variable *p2* zugewiesen. Damit waren im Arbeitsspeicher folgende Plätze belegt.

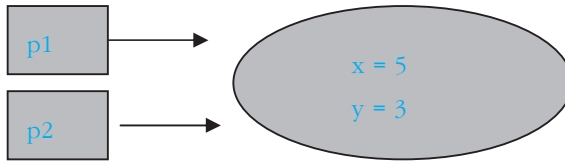


Abb. 5.1: Zwei Referenzvariablen, aber nur 1 Objekt

Besonders wichtig ist die Erkenntnis, dass durch die Zuweisung im Programm *Point02* nicht das Objekt dem Empfänger zugewiesen wird, sondern dass lediglich die Referenz auf das Objekt kopiert und in die Variablen *p2* transportiert worden ist.

Das Objekt selbst ist nur einmal im Arbeitsspeicher.

Das nächste Programm *Point03* beweist diese Aussage. Dort werden nämlich die beiden Werte von *x* und *y* geändert (d.h. jetzt wird das Objekt selbst verändert).

### Programm *Point03*: Wertezuweisung an das Objekt selbst

```
import java.awt.*;
public class Point03 {
    public static void main(String[] args) {
        Point p1 = new Point(5,3);
        Point p2;
        p2 = p1;           // Wertezuweisung
        p2.x = 3333;
        p2.y = 4444;
        System.out.println(p1.toString());
        System.out.println(p2.toString());
    }
}
```

Die Ausgabe ist in beiden Fällen gleich. Damit wird deutlich, dass sowohl *p1* als auch *p2* auf dasselbe Objekt referenzieren.

### Übung zum Programm *Point03*

Bitte geben Sie mit der *println*-Methode den *y*-Wert der Instanz *p1* aus. Lösungshinweis: diese Variable kann referenziert werden durch "*p1.y*".

### 5.3.4 Wodurch unterscheiden sich primitive und Referenzvariablen?

In einem Javaprogramm können Variablen einen primitiven Datentyp oder einen Referenztyp haben. Demzufolge können auch zwei Arten von Variablenwerten unterschieden werden: primitive Werte und Referenzwerte. Wo liegen die grundsätzlichen Unterschiede zwischen den Referenzvariablen und einfachen Variablen?

- Primitive Variablen **bestehen aus genau einem Wert** dieses Datentyps, Referenzvariablen können auf zusammengesetzte Objekte verweisen, die aus unterschiedlichen Typen zusammengesetzt sind.
- Die **Syntax, wie sie erzeugt werden**, ist unterschiedlich: für Referenzvariablen benötigt man das Schlüsselwort *new*.
- **Art und Umfang** des Speicherplatzes ist unterschiedlich: Objekte benötigen zusätzlich zu den eigentlichen Daten noch eine Referenzvariable, die auf den Zielwert verweist.
- Der Zeitpunkt, **wann die Größe des Arbeitsspeicherplatzes für diese Variable bestimmt** wird, ist unterschiedlich: bei einfachen Datentypen kann dies bereits zur Umwandlungszeit (compile-time) und bei Referenztypen erst zur Ausführungszeit (run-time) bestimmt werden.
- Der Zeitpunkt, **wann der Speicherplatz angefordert** wird (Allokierung), ist unterschiedlich: einfache Variablen können bereits beim Laden des Programms in den Arbeitsspeicher den angeforderten Platz belegen, Referenzvariablen werden nach Bedarf während der Programmausführung ("dynamisch") mit dem Schlüsselwort *new* erzeugt.
- Obwohl eine **Wertezuweisung** für beide Variablentypen mit dem Gleichheitszeichen `=` erfolgen kann, ist die Wirkung sehr unterschiedlich: bei einfachen Variablen wird der Wert selbst zugewiesen, bei Referenztypen wird die Referenz auf das Objekt zugewiesen.

In späteren Kapiteln werden Sie **weitere Unterschiede** dieser beiden Variablenarten kennenlernen. Der Vollständigkeit halber werden diese bereits hier aufgeführt:

- Die Syntax, wie sie verarbeitet werden (z.B. vergleichen, zuweisen, kopieren usw.) ist unterschiedlich. Bei Referenztypen werden normalerweise Methoden aufgerufen, bei einfachen Typen wird meistens mit Operatoren wie `+` (Addition) oder `<` (Vergleich) gearbeitet.
- Die Übergabe als Parameter beim Aufruf von Methoden unterscheidet sich bei Objekten grundlegend von der Art, wie einfache Variable übergeben werden. Es wird zwar immer der Wert der Variablen kopiert, aber nur bei einfachen Variablen ist es der tatsächliche Wert, bei Objekten handelt es sich um den Wert der Referenzvariablen (also um die Adresse von dem Objekt), der kopiert und übergeben wird.

## 5.4 Konstanten

### Deklaration von Konstanten

Konstanten sind in Java Namen für Speicherwerte, die unveränderbar sind. Es ist dem Programmierer lediglich möglich, diese Plätze **einmal** zu füllen - danach ist eine Änderung nicht mehr erlaubt.

Konstanten bestehen aus

- einem Namen, z.B. MWST. Empfehlung: Der Name sollte immer aus Großbuchstaben bestehen, um ihn als Name für eine Konstante zu kennzeichnen.
- einem Datentyp, z.B. *int*.
- einem unveränderlichen Inhalt, z.B. 28.

Eine Konstante wird durch das Schlüsselwort *final* deklariert, z.B.

```
final double MWST;
```

Der Wert der Konstanten kann entweder bereits bei der Deklaration festgelegt werden:

```
final double MWST = 19.0;
```

oder später durch eine Wertezuweisung, wie das folgende Programm demonstriert.

### Programm *Konstanten01*: Konstanten definieren und mit ihnen arbeiten

```
public class Konstanten01 {  
    public static void main(String[] args) {  
        final double MWST;  
        MWST = 19.5;  
        System.out.println(MWST);  
    }  
}
```

Wenn der Wert der Konstanten einmal feststeht, kann er danach nicht mehr modifiziert werden (nur durch Änderung des Quelltextes mit anschließender Umwandlung). Wo liegt der Sinn? Zunächst einmal wird durch das Schlüsselwort *final* verhindert, dass eine versehentliche Modifikation des Speicherinhalts erfolgt. Außerdem wird die Lesbarkeit erhöht, denn der Begriff "MWST" ist aussagefähiger als die Zahl 18.0. Und zusätzlich wird immer dann die Wartungsfreundlichkeit verbessert, wenn der Wert an mehreren Stellen im Programm benutzt wird, denn bei eventuellen Änderungen muss nur an einer Stelle modifiziert werden.

### Programm *Konstanten02*: Können Konstantenwerte geändert werden?

```
public class Konstanten02 {  
    public static void main(String[] args) {
```

```

        final double PI = 3.14159;
        System.out.println("PI betraegt = " + PI);
    }
}

```

### Übungen zum Programm Konstanten02

**Übung 1:** Kann der Initialwert auch weggelassen werden? Welche Meldung kommt bei der Umwandlung des Programms?

**Übung 2:** Fügen Sie die Initialisierung wieder ein. Bitte versuchen Sie dann, den Wert von *PI* um 3 zu erhöhen durch Einfügen von folgender Anweisung: *PI = PI + 3*; Wie lautet die Fehlermeldung, die bei der Umwandlung erzeugt wird? Klären Sie danach die Frage, ob evtl. eine neue Wertezuweisung möglich ist, z.B. in der Form: *PI = 3.14*?

### Eingebaute Konstanten

In den mitgelieferten Klassen des Java-API gibt es als Bestandteil von Klassen häufig eingebaute Konstanten. Sie werden anstelle von numerischen Werten benutzt, weil sie dadurch aussagefähiger werden.

So gibt es z.B. in den Wrapper-Klassen für einfache Variablen eingebaute Konstanten, die den jeweils höchsten oder niedrigsten Wert des Datentyps enthalten. Das folgende Beispiel zeigt, wie Maximalwerte von diversen primitiven Datentypen mit Hilfe von vordefinierten Konstanten angezeigt werden können.

### Programm Konstanten03: Arbeiten mit eingebauten Konstanten

```

public class Konstanten03 {
    public static void main(String[] args) {
        // Ganzzahlen
        byte b    = Byte.MAX_VALUE;
        short s   = Short.MAX_VALUE;
        int z     = Integer.MAX_VALUE;
        long l    = Long.MAX_VALUE;
        // Reale Zahlen
        float f   = Float.MAX_VALUE;
        double d  = Double.MAX_VALUE;
        System.out.println("b = " + b);
        System.out.println("s = " + s);
        System.out.println("z = " + z);
        System.out.println("l = " + l);
        System.out.println("f = " + f);
        System.out.println("d = " + d);
    }
}

```

### Übung zum Programm Konstanten03

Überprüfen Sie, wie der Compiler reagiert, wenn Sie versuchen, durch folgende Zeile einen MAX-VALUE zu ändern: `Float.MAX_VALUE = 15;`

#### Wann sollten Konstanten eingesetzt werden?

Konstanten sind nicht nur dann sinnvoll einsetzbar, wenn der Wert wirklich für immer konstant ist (wie bei dem Wert für PI), sondern auch, wenn zu erwarten ist, dass Änderungen eintreten (wie bei dem Wert für den MWST-Satz). In solchen Fällen wird dringend empfohlen, nicht einen festen Wert ("18 %") in den verschiedenen Anweisungen einer oder mehrerer Klassen zu verwenden, sondern eine Konstante zu definieren (damit bekommt der Wert einen Namen). Dann kann in allen Statements dieser Name verwendet werden. Bei einer Änderung des MWST-Satzes wird dann lediglich die Initialisierung der Konstante geändert, und in allen Statements wird mit dem neuen Wert gearbeitet.

**Fazit:** Eine Konstante kann in einem Programm wie eine normale Variable benutzt werden. Einzige Ausnahme: eine Wertezuweisung ist nur **einmal** möglich, üblicherweise bei der Deklaration als Initialwert. Die Syntax für eine Konstante erfordert, dass das Schlüsselwort *final* benutzt wird. Außerdem wird empfohlen, dass der Identifier komplett aus Großbuchstaben besteht.

## 5.5 Literale

### 5.5.1 Was sind Literale?

Literale sind Werte, die im Programm direkt benutzt werden. Ohne dass vorher ein Speicherbereich für eine Variable oder Konstante deklariert wurde, kann der Wert im Quelltext benutzt werden.

Literale sind somit Werte ohne Identifier, sie werden "einfach so" in einer Anweisung codiert. Der Wert eines Literals entspricht einem einfachen Datentyp oder einem String.

Beispiel für ein Literal in einer Wertezuweisung:

```
zahl = 15;
```

Hier ist 15 ein Literal. Der Datentyp eines Literals wird vom Compiler automatisch festgelegt, abhängig von der Schreibweise. Allerdings ist nicht immer eindeutig erkennbar, von welchem Typ das Literal ist, dazu müssen die Regeln bekannt sein, z.B. ist 47.25 ein *float*- oder *double*-Typ?

Welche Literale gibt es?

- Ganzzahlige Werte sind vom Typ *int*.
- Numerische Werte in Dezimalnotation sind vom Typ *double*

- Wahrheitswerte sind vom Typ *boolean*.
- Einzelne Zeichen in einfachen Anführungsstrichen (Apostroph) sind vom Typ *char*, z.B. *'a'*.
- Zeichenketten müssen in doppelten Anführungsstrichen (double quotes) stehen, sie sind vom Typ *String*, z.B. *"Hamburg"*.

In Zuweisungen muss der Datentyp des Literals evtl. an den Typ der Empfängervariablen angepasst werden. Diese Konvertierung geschieht automatisch, solange die beiden Typen kompatibel sind. Wir kommen später auf das Thema Typumwandlung und Casting zurück, dediziert wird es in Kapitel 15 behandelt.

### 5.5.2 Integer-Literale

Ganzzahlige Literale werden als Ziffernfolge ohne Hochkomma geschrieben. Sie werden unabhängig von ihrer Größe immer als Datentyp *int* interpretiert. Beispiele:

```
17
0
5000
```

Standardmäßig wird angenommen, dass es sich um eine Dezimalzahl handelt (also zur Basis 10). Aber das gilt nicht immer, es gibt auch Schreibweisen für ganzzahlige Literale in anderen Stellenwertsystemen:

- Enthält die Zahl eine führende Null, so wird sie als Oktalzahl (also zur Basis 8) interpretiert, z.B. 015
- Beginnt die Zahl mit den Zeichen 0x, so wird dahinter eine Hexadezimalzahl (also zur Basis 16) erwartet, die den Binärwert der Dezimalzahl repräsentiert und durch die ASCII-Zeichen 0-F dargestellt werden. Beispiel: 0xA5FF oder 0xA.

#### Programm *Literal01*: Ganzzahl 17 in verschiedenen Notationen

```
public class Literal01 {
    public static void main(String[] args) {
        int zahl1 = 17;           // Dezimal 17
        int zahl2 = 021;          // Oktal 17
        int zahl3 = 0x0011;       // Hexadezimal 17
        System.out.printf("%d %d %d", zahl1, zahl2, zahl3);
    }
}
```

Das letzte Statement enthält eine Formatierung der Ganzzahlen, damit die Ausgabe einheitlich als Dezimalzahl erfolgt. Die Methode *printf* (siehe Kapitel 6) erlaubt die Angabe von Optionen für die Formatierung, diese stehen in Anführungsstrichen vor den Namen der auszugebenden Variablen und beginnen jeweils mit einem %-Zeichen. Das *d* steht für *decimal* (siehe Abschnitt 6.3.2).



### Übungen zum Programm *Literal01*

**Übung 1:** Schreiben Sie auf ein Blatt Papier, wie die binären Inhalte dieser drei Variablen sind. (Frage: Benötigt man dazu die ASCII- oder Unicode-Tabelle im Anhang? Antwort: Nein.) Hilfreich könnte es jedoch sein, wenn das folgende Statement in das Programm eingefügt wird:

```
System.out.println(Integer.toBinaryString(zahl1));
```

**Übung 2:** Ändern Sie den Datentyp wahlweise in *byte*, *long* oder *short*, um zu überprüfen, ob die Zahlen auch für diese Typen erlaubt sind.

**Übung 3:** Was passiert, wenn die Zahl nicht in den Wertebereich des angegebenen Datentyps passt? Beispiel: *byte zahl1 = 128*; Was ist die höchste Zahl, die in der Variablen *zahl1* gespeichert werden kann?

Das folgende Programm soll die Dezimalzahl 19 ausgeben. (Hinweis: Oktalzahlen werden mit einer führenden Null als Literal dargestellt.)

### Programm *Literal02*: Fehlerhafte Anwendung eines ganzzahligen Literals

```
public class Literal02 {  
    public static void main(String[] args) {  
        int zahl1 = 019;  
        System.out.println(zahl1);  
    }  
}
```

### Übung zum Programm *Literal02*

**Übung 1:** Das Programm liefert folgenden Umwandlungsfehler: "integer number too large: 019". Das erscheint zunächst seltsam. Aber gibt es eine Oktalziffer 8 oder 9? Korrigieren Sie den Fehler.

**Übung 2:** Ändern Sie den Initialwert der Variablen *zahl1* auf einen negativen Wert. Hinweis: Das Vorzeichen steht vor der Zahl.

Was haben Sie durch die obige Übung gelernt? Eine führende Null verändert den Wert einer Zahl, z.B. bedeutet 25 etwas ganz anderes als 025. Warum ist das so? Zum einen ist dies eine Erblast von C++, zum anderen wollten die Entwickler wohl garantieren, dass schwierige Fehler eingebaut werden können. :-)

Ganzzahlen-Literale sind mindestens vom Typ *int*. Es gibt keine Literale vom Typ *short* oder *byte* (obwohl sie umgewandelt werden in diesen Typ, falls dies erforderlich, sinnvoll und ohne Informationsverlust möglich ist).

Wie aber erreicht man, dass ein Literal als *long*-Typ interpretiert wird. Dafür ist eine besondere Angabe erforderlich, nämlich der Buchstabe *l* (oder *L*) hinter der Zahl.

**Programm *Literal03*: Ganzzahliges Literal als *long*-Datentyp interpretieren**

```
public class Literal03 {  
    public static void main(String[] args) {  
        long zahl1 = 12345678901L;  
        System.out.println(zahl1);  
    }  
}
```

**Übungen zum Programm *Literal03***

**Übung 1:** Dem Literal ist der Buchstabe L angehängt. Entfernen Sie den Buchstaben L. Welche Fehlermeldung gibt es bei der Umwandlung?

**Übung 2:** Kann auch ein Kleinbuchstabe l benutzt werden? Prüfen Sie, ob die Lesbarkeit leidet (besteht etwa Verwechslungsgefahr mit der Ziffer 1)?

**5.5.3 Boolean-Literale**

In Java gibt es für die Wahrheitswerte "wahr" oder "falsch" entsprechende Literale: *true* und *false*. Natürlich müssen sie exakt so geschrieben werden.

**Programm *Literal04*: Arbeiten mit Boolean-Literalen**

```
public class Literal04 {  
    public static void main(String[] args) {  
        boolean luege = true;  
        System.out.println(luege);  
    }  
}
```

**Übung zum Programm *Literal04***

Die Zeile 3 enthält eine typische Deklaration, sie enthält die drei Teile Name, Typ und Initialisierung. Bitte identifizieren Sie diese drei Teile.

**Programm *Literal05*: In jeder Anweisung steckt ein Fehler!**

```
public class Literal05 {  
    public static void main(String[] args) {  
        boolean luege = "false";  
        System.out.println(false + 3);  
    }  
}
```

**Übung zum Programm *Literal05***

Jede der beiden Zeilen in der Methode *main* enthält einen Fehler. Bitte korrigieren Sie dies. Hinweis: Datentyp *String* passt nicht, mathematische Befehle nicht erlaubt.

### 5.5.4 Gleitkomma-Literale

Für Real-Zahlen, also Zahlen mit Dezimalstellen hinter dem Komma, gibt es unterschiedliche Schreibweisen. Entweder enthalten sie (nur) den Dezimalpunkt oder auch den Buchstaben e (bzw. E) mit nachfolgendem Exponenten. Beispiele:

```
123.45           // Standard-Schreibweise
123.4 E 3         // "Wissenschaftliche" Schreibweise
```

In jedem Fall aber ist, wie allgemein in Programmiersprachen üblich, die angelsächsische Schreibweise für Dezimalzahlen gefordert, also Dezimalpunkt statt Dezimalkomma. In der so genannten wissenschaftlichen Schreibweise für Gleitkomma-Literale wird nach dem Buchstaben "E" ein Exponent angegeben.

#### Programm *Literal06*: Gleitkomma-Literal in Standardschreibweise

```
public class Literal06 {
    public static void main(String[] args) {
        double d1 = 123.0;
        System.out.println(d1);
    }
}
```

#### Programm *Literal07*: Gleitkomma-Literal in wissenschaftlicher Schreibweise

```
public class Literal07 {
    public static void main(String[] args) {
        double d1 = 12345E-2;
        System.out.println(d1);
    }
}
```

#### Übungen zum Programm *Literal07*

**Übung 1:** Definieren Sie eine weitere Variable *d2* und initialisieren Sie diese mit dem Anfangswert 0,0375. Benutzen Sie dabei die wissenschaftliche Schreibweise (Beispiel für eine mögliche Lösung: 3.75e-2). Spielt die Groß-/Kleinschreibung des Buchstaben e eine Rolle?

**Übung 2:** Ändern Sie den Datentyp von *double* auf *float*? Ist eine fehlerfreie Umwandlung möglich?

#### Wie werden *float*-Literale benutzt?

Weil die Übung 2 zu einem Umwandlungsfehler geführt hat, bleibt zu klären, wie ein *float*-Typ mit einem Literal gefüllt werden kann, denn offensichtlich wird jedes Gleitkomma-Literal standardmäßig als *double*-Typ interpretiert.

Soll ein Literal mit einem Dezimalpunkt als *float*-Wert und nicht als *double*-Wert interpretiert werden, so muss dies ausdrücklich mitgeteilt werden, und zwar durch Angabe des Buchstabens *f* (bzw. *F*) hinter dem Literal.

### Programm *Literal08*: Arbeiten mit Float-Literalen

```
public class Literal08 {
    public static void main(String[] args) {
        float d1 = 5e-3f;
        System.out.println(d1);
    }
}
```

Gleitkomma-Literale ohne explizite Angabe von *f* oder *F* werden als *double*-Typ interpretiert. Die Notation in hexadezimal oder oktal ist für Gleitkomma-Werte nicht erlaubt.

#### 5.5.5 *char*-Literale

Zeichenliterale haben den Datentyp *char*. Sie werden im Quelltext in einfache Anführungsstriche eingefasst und können dargestellt werden durch

- das Zeichen selbst, z.B. 'A' oder durch
- den Codepoint im Unicode z.B. '\u0009', angegeben als hexadezimale Darstellung der beiden Bytes im Arbeitsspeicher, oder durch
- so genannte vordefinierte Escapesequenzen, z.B. '\n'.

Für das Arbeiten mit dem Codepoint und den vordefinierten Escapesequenzen ist das Codieren des Rückwärtsstrichs (backslash) \ notwendig. Der Codepoint wird als hexadezimaler, vierstelliger Wert eingegeben, angeführt von dem \ und einem kleinen u (für Unicode). Eine etwas verwirrende Eigenart ist die mögliche Angabe als Oktalzahl, z.B. '\317'.

### Programm *Literal09*: Arbeiten mit *char*-Literalen

```
public class Literal09 {
    public static void main(String[] args) {
        char zeichen = 'A';
        System.out.println("Das Zeichen ist: " + zeichen);
    }
}
```

#### Übung zum Programm *Literal09*

Ändern Sie das Programm so, dass als Zeichenliteral nicht der Buchstabe selbst, sondern sein Codepoint als Initialwert angegeben wird. Lösungshinweis: *int*-Typ.

### 5.5.6 Vordefinierte Escape-Sequenzen

Literale werden benutzt, um primitive Werte oder Strings in einem Programm direkt benutzen zu können. Was ist aber, wenn dieser Wert nicht per Tastatur in den Sourcecode eingegeben werden kann. Beispielsweise kann das Zeilenvorschubzeichen nicht so ohne weiteres Teil des Quelltexts sein, denn es würde direkt beim Editieren einen Zeilenvorschub auslösen.

Ist also zur Programmlaufzeit ein Vorschub auf eine neue Zeile erforderlich, so muss ein spezielles Escapezeichen benutzt werden. Die Escapesequenzen für Zeilenvorschub ist `'\n'`, das entspricht dem Codepoint `'\u000a'`.

#### Programm *Literal10*: Arbeiten mit vordefinierten Escape-Sequenzen

```
public class Literal10 {  
    public static void main(String[] args) {  
        char lf = '\n';  
        System.out.println("Zeile1" + lf + "Zeile2");  
    }  
}
```

Besonders ist den Fällen, wo es sich bei dem Character-Literal um nicht-darstellbare Zeichen (im ASCII-Code die Zeichen 0- 31) handelt, ist es sinnvoll, auf vordefinierte Darstellungen zurückzugreifen. Sie beginnen alle mit `\` (backslash).

Es gibt eine ganze Reihe von vordefinierten Escapesequenzen. Allen gemeinsam ist, dass sie eine besondere Bedeutung für den Compiler haben, man "flüchtet" also aus der normalen Codierung und verlangt eine besondere Leistung vom Compiler. Weitere Escapesequenzen sind z.B.

<code>'\t'</code>	= Tabulator (dezimal 9, hex. 0x09)
<code>'\f'</code>	= Formfeed (dezimal 12, hex. 0x0C)
<code>'\"'</code>	= Darstellung des doppelten Hochkomma
<code>'\''</code>	= Darstellung des einfachen Hochkomma
<code>'\\'</code>	= Darstellung des Backslash

Besonders die drei letzten Zeichen sind interessant. Sie werden benötigt, wenn innerhalb eines Programms die Zeichen `"` oder `'` oder `\` als Teil eines Textes dargestellt werden müssen.

#### Übung zum Programm *Literal10*

**Übung 1:** Ändern Sie das Programm so, dass zwischen den beiden Teilstrings `"Zeile1"` und `"Zeile2"` anstelle des Linefeed a) ein `"` (Anführungszeichen) und b) ein `\` (Backslash) erscheint.

**Übung 2:** Ändern Sie das Programm so, dass zwischen den beiden Teilstrings ein Alarm ertönt. Das Unicodezeichen dafür ist `\u0007`.

### 5.5.7 String-Literale

String-Literale bestehen aus einem oder mehreren Zeichen, die in doppelten Hochkommas eingeschlossen sind. (Zur Erinnerung: ein *char*-Literal wird in einfache Hochkommas eingeschlossen.) Wie alle anderen Literale auch, so können Stringlitterale eingesetzt werden

- für die Initialisierung,
- bei Wertezuweisungen,
- als Parameter bei einem Methodenaufruf (siehe Kapitel 10).

#### String-Literale bei der Objekt-Initialisierung

Die mitgelieferte Klasse *String* bietet eine vereinfachte Möglichkeit der Instanz-Erzeugung. Normalerweise wird eine Instanz mit dem Schlüsselwort *new* erzeugt. Eine String-Instanz kann aber durch einfache Initialisierung oder auch durch eine bloße Wertezuweisung erzeugt werden. Und dabei können auch Stringlitterale eingesetzt werden. Alle folgenden drei Beispiele haben exakt die gleiche Wirkung:

// Ausführliche Schreibweise mit dem Schlüsselwort *new*:

```
String str = new String("abc");
```

// Kurzschreibweise durch Initialisierung:

```
String str = "abc";
```

// Objekterzeugung durch Wertzuweisung:

```
String str;  
str = "abc";
```

In allen Fällen wird ein String-Literal benutzt (mit den 3 Zeichen "abc").

Java kennt für Referenzvariablen noch ein besonderes Literal: *null*. Es wird benutzt, um anzuzeigen, dass die Referenzvariable leer ist, dass also noch kein Objekt existiert. Der Wert *null* wird entweder explizit vom Programmierer zugewiesen oder - wenn es sich nicht um lokale Variable, sondern um ein Klassenmember (siehe Kapitel 11) handelt, als Initialwert automatisch eingesetzt.

#### Programm Literal11: Schlüsselwort *null* für Referenzvariablen

```
public class Literal11 {  
    public static void main(String[] args) {  
        String s1 = null;  
        System.out.println(s1);  
    }  
}
```

### Sonderzeichen im *String*-Literal

So wie es bei *char*-Literalen die Möglichkeit gibt, mit so genannten Escapesequenzen problematische Zeichen des Unicodes darzustellen, so gibt es auch bei *String*-Literalen die Möglichkeit, Escape-Sequenzen oder Codepoints innerhalb eines solchen *String*-Literalen zu verwenden.

### Programm Literal12: Arbeiten mit *String*-Literalen und Escape-Sequenzen

```
public class Literal12 {  
    public static void main(String[] args) {  
        String str = "a\u0062c";  
        System.out.println("Der String enthaelt: " + str);  
    }  
}
```

Wenn ein Seitenvorschub innerhalb des Strings ausgelöst werden soll, so muss das Escape-Zeichen ' \n ' dafür eingetragen werden.

### Übungen zum Programm Literal12

Ersetzen Sie den Codepoint des Buchstabens b durch die entsprechende Escape-Sequenz für den Tabulator.

Bestimmte Zeichen sind in einem String also **nicht direkt** verwendbar, z.B. Zeilenvorschub. Damit darf ein Stringliteral nicht über zwei oder mehr Zeilen gehen. Sollte das Literal zu lang sein für eine Zeile, so muss es auf mehrere Zeilen aufgeteilt und mit dem +-Operator konkateniert werden.

### Programm Literal13: Ein langes *String*-Literal über mehrere Zeilen

```
public class Literal13 {  
    public static void main(String[] args) {  
        String str = "Dies ist ein " +  
                     "langer Text";  
        System.out.println(str);  
    }  
}
```

### Übung zum Programm Literal13

Beantworten Sie folgende Frage: Wie kann man sehr leicht erkennen, ob der Datentyp einer Variablen ein primitiver Datentyp ist oder ob es sich um eine Klassenbeschreibung handelt?

Lösungshinweis: Klassennamen sollten mit einem Großbuchstaben beginnen. Somit unterscheiden sich diese Identifier von den eingebauten Typen, die - wie alle Schlüsselwörter - klein geschrieben werden.

## 5.6 Zusammenfassung

In diesem Kapitel haben wir die Möglichkeiten besprochen, wie Variable im Arbeitsspeicher erzeugt und verwaltet werden können. Üblicherweise können die Arbeitsspeicherplätze einen variablen Inhalt haben. Das geschieht so, dass der RAM für jedes Programm immer wieder neu und individuell in Einzelplätze aufgeteilt wird und dass diesen Plätzen ein Name (ein Identifier) und ein Typ zugeordnet wird. "Variabel" sind sie deswegen, weil der Inhalt dieser Speicherplätze (der "Wert") während der Laufzeit durch Programmbefehle änderbar ist.

- Wenn die Arbeitsspeicherplätze einen Namen haben *und* änderbar sind, bezeichnet man sie als **Variablen**.
- Wenn die Arbeitsspeicher-Plätze lediglich einen Namen haben, aber nicht änderbar sind, so sind es **Konstante** (oder auch "konstante Variablen").
- Wenn die Daten im RAM keinen Namen haben, sondern Teil des Quellcodes sind (und somit auch nicht änderbar), so handelt es sich um **Literale**.

Die Variablen (und Konstanten) müssen vor ihrer Verwendung deklariert werden, Literale werden "einfach so" im Quelltext benutzt. Den Variablen muss außerdem ein Anfangswert zugewiesen sein, bevor sie genutzt werden können.

### Variablen

Java unterscheidet grundsätzlich zwischen zwei Arten von Variablen:

- **primitive Variablen** und
- **Referenz-Variablen** (Objekt, Instanz).

Primitive Variablen haben einen der acht eingebauten einfachen Datentypen, Referenz-Variablen sind vom Typ einer Klasse. (Der Vollständigkeit halber wird an dieser Stelle bereits darauf hingewiesen, dass es nicht nur Referenztypen, die von Klassen erzeugt werden, gibt, sondern darüber hinaus noch weitere Referenztypen, z.B. Interface-, Array- und Enumerationentypen. Diese werden ausführlich in den nächsten Kapiteln vorgestellt.)

Primitive Variablen enthalten direkt den Wert eines einfachen Typs, Referenz-Variablen zeigen auf ein Objekt, das zusammengesetzt sein kann aus mehreren Variablen.

Eine andere Unterscheidung der Variablen ist möglich nach der Art der Zugehörigkeit:

- **lokale Variablen** (als Teil einer Methode oder eines Anweisungsblocks und auch nur dort verwendbar)
- **Member-Variablen** (als Teil einer Klasse, kann von allen Methoden dieser Klasse benutzt werden). Sie werden auch als **Felder** der Klasse bezeichnet (mehr dazu im Kapitel 11).



### Konstanten

Durch das Schlüsselwort "*final*" bei der Variablendeklaration kann festgelegt werden, dass dieser Variablen nur einmal ein Wert zugewiesen werden kann. Danach kann nur noch lesend darauf zugegriffen werden. Vorteile bei der Verwendung von Konstanten: Programme werden lesbarer, verständlicher und leichter wartbar.

### Literale

Literale sind namenlose Werte. Auch Literale haben einen Datentyp. Der Compiler erkennt den Datentyp, den ein Literal hat, automatisch. Ganzzahlen z.B. sind vom Typ *int*, gebrochene Zahlen sind vom Typ *double*. Von dieser automatischen Festlegung kann jedoch abgewichen werden durch zusätzliche Angaben des Programmiers:

- Bei Gleitkomma-Literalen kann der Buchstabe *f* (groß oder klein geschrieben) hinter dem Literal die Umwandlung des Wertes in *float* erzwingen
- Bei Ganzzahlen kann der Buchstabe *L* (groß oder klein geschrieben) hinter dem Literal die Umwandlung des Wertes in *long* erzwingen.

Es gibt unterschiedliche Möglichkeiten der Darstellung, so kann z.B. die Ganzzahl 25 wie folgt angegeben werden:

25	(= als Dezimalzahl)
19	(= als hexadezimale Zahl)
31	(= als Oktalzahl).

Es gibt **vordefinierte Literale**, z.B. *true* oder *false*. Außerdem können für spezielle Sonderzeichen vordefinierte **Escapesequenzen** benutzt werden, z.B. '*\n*' für den Zeilenvorschub.

Die Verwendung von Literalen in Ausdrücken sollte möglichst vermieden werden, denn durch den fehlenden Bezeichner sind sie wenig aussagefähig für den menschlichen Leser. Besser ist oft der Einsatz von Konstanten anstelle von Literalen, diese haben den zusätzlichen Vorteil, dass bei Änderungen nicht der gesamte Quelltext durchsucht werden muss.

# 6

## Eingabe und Ausgabe durchführen ("i/o-operation")

---

In diesem Kapitel erhalten Sie Antwort auf folgende Fragen:

- Wie erfolgt in Java die Ein- und Ausgabe (Input/Output)?
- Welche Bedeutung hat dabei das *Stream*-Konzept?
- Was ist der Unterschied zwischen byte- und characterorientierter Ein- und Ausgabe?
- Welche Klassen werden eingesetzt, um ASCII-Daten in Unicode-Daten umzuwandeln und was versteht man in dem Zusammenhang unter "encoding"?
- Wie können die Standard-Eingabeeinheiten bzw. Standard-Ausgabeeinheiten (die Console) in Java-Programmen genutzt werden?
- Wie hilft die Klasse *Scanner* bei der Analyse und Interpretation der eingelesenen Daten?
- Wie hilft die Klasse *Formatter* bei der Aufbereitung der Ausgabe?
- Wie können auszugebende Daten mit der *format()*-Methode aufbereitet werden?
- Welche Möglichkeiten der Dateiverarbeitung gibt es?

### 6.1 Stream-Konzept

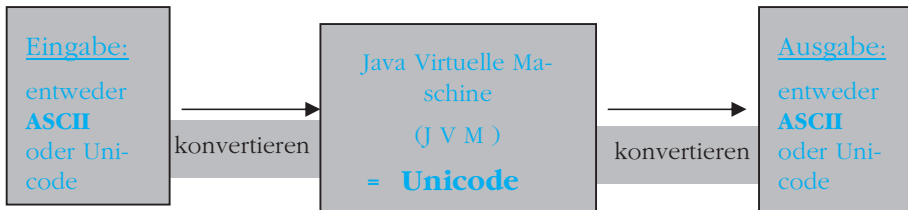
Unter Eingabe (Input) versteht man das Einlesen von Daten in die "Java Virtuelle Maschine" (JVM). Bei der Ausgabe (Output) verlassen die Daten die JVM. Für diese Arbeiten enthält Java ein generelles Konzept: das Stream-Konzept.

Ein Stream (Strom) beschreibt, wie Daten transportiert werden von einem Sender (der "Quelle") zu einem Empfänger (der "Senke"). Der Datenfluss wird **unabhängig von dem Typ** der Ausgabe oder von dem Typ der Eingabe in immer gleicher Weise beschrieben und genutzt.

Die mitgelieferte Klassenbibliothek bietet eine Fülle von vorprogrammierten Klassen mit vielen Methoden, in der alle wichtigen Ein- und Ausgabevarianten vorprogrammiert sind. Diese etwa 50 Klassen sind im Paket *java.io* zusammengefasst.

Ein wichtiges Designziel der Java-Entwickler war es, universelle und kompatible Verarbeitungsmöglichkeiten anzubieten. Dabei war ein grundsätzliches Problem zu lösen: Während Java intern bereits mit dem Unicode arbeitet, ist es außerhalb der JVM nach wie vor Standard, mit einem 8-bit-Code (z.B. ASCII) zu arbeiten. Das bedeutet:

Beim Einlesen muss das 8-bit-Zeichen in Unicode und beim Ausgeben das Unicode-Zeichen in eine 8-bit-Codierung transformiert werden.



**Bild 6.1:** Codetransformation bei der Ein- und Ausgabe

Dies ist ein Grund, warum in Java die Ein- und Ausgabe grundsätzlich **zeichenweise** erfolgt - und nicht, wie häufig bei anderen Plattformen oder Programmiersprachen, **zeilenweise** oder **satzweise**. Aber keine Sorge: es gibt natürlich auch in Java zusätzliche Klassen mit komfortablen Methoden, die all das können, was von einem leistungsfähigen I/O-System erwartet wird: zeilen- oder wortweise transportieren, einfache Datentypen oder Objekttypen senden und empfangen, dabei die Daten komprimieren und in ZIP-Files packen oder verschlüsseln und signieren nach Kryptographiestandards. Und das Arbeiten mit relationalen Datenbanken ist natürlich auch in das Java-API integriert.

Leider ist es so, dass bestimmte Aufgaben nur im Zusammenspiel von mehreren Klassen durchgeführt werden können. In diesen Fällen wird das Streamobjekt aus mehreren Klassen erzeugt. Dadurch wird die Einarbeitung in das Thema auch nicht einfacher.

Aber alle Verfahren basieren auf zeichenweises Lesen und Schreiben von Streams. Wir wollen in diesem Kapitel das Grundsätzliche, das Prinzipielle besprechen und deswegen wird zunächst der wichtigste Eingabebefehl, die Methode `read()`, anhand mehrerer Beispiele demonstriert.

### 6.1.2 Was bietet die Methode `read()`?

Sie hatten bereits gelernt: Zum Konzept der Java-Sprache gehört es, dass die Programme universell einsetzbar sind. Sie laufen unverändert auf jeder Hardware-Plattform, unter jedem Betriebssystem, sofern eine JVM zur Verfügung steht. Für jeden Sprachraum oder Kulturkreis sollen die dort vorhandenen Zeichen verarbeitet werden können. Darüber hinaus soll das Streamkonzept immer in der gleichen Art und Weise funktionieren, unabhängig von der Art der beteiligten Ein- und Ausgabegeräte. Das Lesen oder Schreiben von Magnetplattendaten soll prinzipiell nicht anders funktionieren als das Lesen oder Schreiben von Daten, die über eine Internet-Leitung (TCP/IP mit Sockets) mit chinesischen oder arabischen Geschäftspartnern ausgetauscht werden.

Das sind also die Gründe, warum die Methode `read()` zunächst lediglich die Unicode-Repräsentation des eingelesenen Zeichens an das Programm liefert.

Somit ist man in der Lage, jede denkbare Bitkombination zu empfangen und **abhängig vom Kontext zu interpretieren**, z.B.

- als einzelnes (Nutz-)zeichen oder
- als Zeichen, das nur im Verbund mit anderen Bytes interpretiert werden kann wie bei typisierten Daten oder
- als Steuerzeichen für Zeilen- oder Dateiende usw.

Java ist für alle Fälle und vor allem für zukünftige Entwicklungen gerüstet!

Die meisten Beispiele in diesem Buch benutzen die Konsole als Benutzeroberfläche ("user-interface"). Das Wort "Konsole" bedeutet hier, dass eine Tastatur als Eingabegerät und ein zeichenorientierter Bildschirm als Ausgabegerät benutzt werden. Wir arbeiten also nicht - wie in der Praxis üblich - mit der Maus und auch nicht mit einer grafischen Oberfläche.

Diese textbasierte Ein- und Ausgabe am Bildschirm ist völlig ausreichend, um den gesamten Stoff des Buches an Beispielen zu demonstrieren und zu üben. So kann der Leser sich ganz auf das Erlernen der Programmiersprache und auf die objektorientierte Denkweise konzentrieren.

Fast alle Betriebssysteme und Programmiersprachen kennen standardisierte Verfahren, wie die Eingabe (**Standard-Input**) und die Ausgabe (**Standard-Output**) über Bildschirm und Tastatur bereitgestellt werden. In Java werden diese Geräte als Ziel oder Quelle eines *Streams* gesehen.

Das Programm *Stream01.java* enthält ein Beispiel für das Lesen einer Bediener Eingabe am Konsolbildschirm. Dieses Programm soll helfen, die Frage zu klären, was liefert die Methode `read()` nun wirklich an das Programm?

### Programm *Stream01*: Console als Eingabestream

```
import java.io.*;

public class Stream01 {
    public static void main(String[] args) throws Exception {
        InputStreamReader eingabe = new InputStreamReader(System.in);
        int zeichen = eingabe.read();
        System.out.println(zeichen);
    }
}
```

Zunächst wird eine Instanz der Klasse *InputStreamReader* erzeugt. Dadurch ist es möglich, die Methode `read` zu nutzen. Wir wollen die Frage klären: Was liefert `read()`? Die Antwort ergibt sich aus den nachfolgenden Übungen.

## Übungen mit dem Programm Stream01

**Übung 1:** Bitte wandeln Sie das Programm um und testen Sie es. Geben Sie den Buchstaben A ein (oder ein Zeichen aus dem ASCII-Code von 0 - 127) und überprüfen Sie, ob die Ausgabe dem Codepoint in der Unicode-Tabelle entspricht. Handelt es sich um den hexadezimalen Wert oder um die Angabe als Dezimalzahl?

**Übung 2:** Was gibt das Programm aus, wenn Sie (unter MS-Windows) nach dem Programmstart lediglich die Enter-Taste drücken, ohne ein Zeichen einzugeben? Wenn Sie in der ASCII-Tabelle (siehe Anhang C) die Textbeschreibung zu diesem Codepoint herausuchen, finden Sie die Antwort.

**Übung 3:** Für die erste Annäherung an das Thema Unicode geben Sie bitte einen deutschen Umlaut ein (ö, ü, ä). Die Ausgabe bleibt unverständlich, wir werden später eine Erklärung dafür geben.

Fazit: Die *read*-Methode liefert einen *int*-Wert, der den Codepoint der eingelesenen Bitkombination aus der Unicode-Tabelle repräsentiert. Der Grund dafür ist, dass damit eine individuelle Interpretation und Konvertierung der Bitkombinationen möglich ist - je nach Situation und Umfeld.

### 6.1.2 Byte- oder characterorientiertes Input/Output (I/O)

Die vielfältigen Möglichkeiten der Stream-Verarbeitung sind in etwa 50 Klassen vorprogrammiert. Es gibt I/O-Methoden, die gepuffert arbeiten (aus Performanzgründen), es gibt Methoden, die verarbeiten komplette Objekte, oder andere, die filtern den Ein- oder Ausgabestrom nach bestimmten Regeln, es gibt spezielle Methoden für den Austausch von Informationen zwischen Threads (Programmteile) usw.

Um die Übersicht zu erleichtern, werden wir die Klassen gruppieren. Zunächst kann man trennen zwischen Eingabeklassen und Ausgabeklassen. Und dann kann man byteorientierte und characterorientierte Streams unterscheiden.

	Eingabe		Ausgabe	
	Basisklasse	Basismethode	Basisklasse	Basismethode
Byte-Orientiert	InputStream	read() = <i>int</i> von 0 - 255	OutputStream	write() = <i>int</i> von 0-255
Character-Orientiert	Reader	read() = <i>int</i> von 0-65535	Writer	write() = <i>int</i> von 0-65535

Bild 6.2: Namen der Basisklassen und -Methoden (für die gesamte I/O-Hierarchie)

(Fast) alle Klassen des I/O-Systems sind von diesen vier Basisklassen abgeleitet. Je nach Ein-/Ausgabegerät und Aufgabenstellung gibt es spezialisierte Klassen, die die vier I/O-Methoden unterschiedlich implementieren und ergänzen.

#### Die byteorientierten Streams (Byteströme)

- basieren auf *InputStream*- bzw. *OutputStream*-Klassen. Sie arbeiten mit einzelnen Bytes, d.h. ihre Verarbeitung ist limitiert auf einen 8-bit-Code (standardmäßig arbeiten sie mit dem ISO-Latin-1-Code).

#### Die characterorientierten Stream (Zeichenströme)

- basieren auf *Reader*- bzw. *Writer*-Klassen. Sie können jedes Zeichen des UTF-16-Unicode-Zeichensatzes verarbeiten.

Das nächste Beispielprogramm demonstriert, wie flexibel die Java-Sprache ist, wenn sich das Eingabemedium ändert. Wir wollen die gleiche Aufgabe wie im Programm *Stream01* lösen, nämlich das Lesen eines einzelnen Zeichens von einem Eingabestrom. In diesem Beispiel soll der Eingabestrom allerdings eine Datei sein.

#### Programm *Stream02*: Datei als Eingabestrom (byteorientiert)

```
// Kommentarzeile (das erste Zeichen wird gelesen)
import java.io.*;

public class Stream02 {
    public static void main(String[] args) throws Exception {
        InputStream eingabe = new FileInputStream("Stream02.java");
        int zeichen = eingabe.read();
        System.out.println(zeichen);
    }
}
```

Das Programm *Stream02* liest mit der Methode *read()* aus der Klasse *FileInputStream* ein einzelnes Zeichen aus der Datei "*Stream02.java*". Das Ergebnis dieses Lesevorganges steht dann in der *int*-Variablen *zeichen*. Bitte beachten Sie: diese Variable hat den Datentyp *int*. Anschließend wird das gelesene Zeichen ausgegeben. Das Ergebnis ist 47. Laut Unicode-Tabelle ist dies der Codepoint (oder die Platz-Nr.) für den Schrägstrich (denn das ist das erste Zeichen in der Eingabedatei)..

Übrigens kann man in diesem Fall auch in der ASCII-Tabelle nachsehen, denn die ersten 128 Zeichen des ASCII-Codes sind identisch mit den ersten 128 Zeichen des Unicodes.

#### Übung zum Programm *Stream02*

Bitte ändern Sie das Programm so, dass das vierte Zeichen der Datei eingelesen und dessen Codepoint ausgegeben wird. Lösungshinweis: Es sind vier Lesebefehle erforderlich.

### 6.1.3 Unterschiedliche Interpretation der Eingabe

Das Programm *Stream02* liest Daten ein. Im Programm muss entschieden werden, wie die eingelesene Bitkombination interpretiert werden soll. Mit einfachen Mitteln kann der eingelesene *int*-Wert z.B. als ein Unicode-Zeichen (character) interpretiert werden. Dies demonstriert das nachfolgende Programm.

#### Programm *Stream03*: Interpretation der Eingabe als *char*

```
import java.io.*;
public class Stream03 {
    public static void main(String[] args) throws Exception {
        InputStreamReader eingabe = new InputStreamReader(System.in);
        int zeichen = eingabe.read();
        System.out.println((char) zeichen);
    }
}
```

Notwendig ist lediglich die Angabe "(*char*)" im Ausgabebefehl. Dadurch wird der Inhalt der Integervariablen als Zeichen interpretiert, und es wird anstelle des Zahlenwerts der Schrägstrich ausgegeben.

#### Übungen zum Programm *Stream01* (!)

**Übung 1:** Wir kommen zurück auf das Programm *Stream01*. Bitte testen Sie dieses Programm (= Lesen von *System.in*), indem Sie eine Zahl, bestehend aus einer Ziffer, z.B. 5, eingeben.

a) Was wird vom Programm ausgegeben, wenn Sie dieses einzelne Zeichen ohne eine zusätzliche Interpretation angeben? Hinweis: Das ist die Platz-Nummer in der Unicode-Tabelle.

b) Was wird vom Programm ausgegeben, wenn Sie dieses einzelne Zeichen auch als *char* interpretieren?

**Übung 2:** Ändern Sie die Interpretation des eingelesenen *int*-Wertes so, dass die Zahl als Byte-Wert ausgegeben wird. Hinweis: Der Datentyp *char* muss durch *byte* geändert werden.

Ein weiteres Beispiel (Programm *Stream04*) soll die Flexibilität der Java-Sprache auf andere Weise demonstrieren. Wir ändern erneut die Quelle unseres Eingabestroms. Jetzt soll über eine TCP/IP-Verbindung von einem entfernten ("remoten") System gelesen werden. Diese Verbindung wird als Socket-Verbindung bezeichnet. Sie setzt voraus, dass zwei Partnerprogramme (ein Client- und ein Serverprogramm) in unterschiedlichen Adressräumen aktiv sind und über eine Kommunikationsleitung miteinander verbunden sind. Außerdem müssen sie sich abgestimmt haben, wer der Sender ist und wer der Empfänger.

Dieses Beispielprogramm demonstriert nur das Lesen.

**Programm *Stream04*: TCP/IP-Verbindung als Eingabestrom**

```
import java.net.*;
import java.io.*;
public class Stream04 {
    public static void main(String[] args) throws Exception {
        ServerSocket ss = new ServerSocket(1500);
        Socket s = ss.accept();
        DataInputStream ein = new DataInputStream(s.getInputStream());

        int zahl = ein.read();
        System.out.println(zahl);
    }
}
```

Um über eine TCP/IP-Leitung Daten empfangen zu können, müssen die beiden ersten Zeilen der *main*-Methode zusätzlich codiert werden. Damit wird eine so genannte Socket-Verbindung über den Port 1500 aufgebaut. Zum Testen dieser Anwendung fehlt allerdings auch noch ein Senderprogramm. Der Anspruch dieses Buches ist es, einfache und konzentrierte Beispiele zu bieten. Und: dies sollen keine Codefragmente oder Programmausschnitte sein, sondern komplette, ausführbare Programme. Damit das Versprechen auch für dieses TCP/IP-Beispiel erfüllt wird, beschreiben wir im Anhang D, wie diese kleine verteilte Anwendung komplettiert werden muss und wie sie getestet werden kann.

Am Ende dieses Unterkapitels, in dem Sie das Stream-Konzept kennen gelernt und anhand vieler Beispiele ausprobiert haben, zeigen wir Ihnen ein weiteres Beispiel für das Lesen eines Datenstroms (Programm *Stream05*). Diesmal soll der Eingabestrom im selben Adressraum liegen wie das Programm selbst. Auch das ist mit Stream-Techniken möglich; es muss also nicht immer ein Ein- oder Ausgabegerät sein, mit dem kommuniziert wird. Sie sehen: Auch innerhalb des Arbeitsspeichers, sogar innerhalb eines Adressraums (einer JVM), ist die *read*-Methode einsetzbar.

**Programm *Stream05*: Lesen eines Streams im selben Adressraum**

```
import java.io.*;
public class Stream05 {
    public static void main(String[] args) throws Exception {
        String str = "Dies ist ein Text im Arbeitsspeicher";
        StringReader text = new StringReader(str);
        int c = text.read();
        System.out.print((char)c);
    }
}
```



## 6.2 Standard-Eingabe

Für das Einlesen von Zeichen, die der Bediener am Bildschirm eingetippt hat, gibt es eine große Vielfalt unterschiedlicher Methoden. Alle haben als Basis das Einlesen von Einzelzeichen, doch gibt es auch komfortable Methoden, die für das Programm ganze Zeichenketten, komplette Zeilen oder typisierte Daten einlesen und diese nach bestimmten Regeln analysieren und für die Verarbeitung zur Verfügung stellen.

### Programm *Console01*: Zeichenweise Lesen von der Konsole (diesmal mit *ByteStream* und mit verkürzten Schreibweise)

```
import java.io.*;
public class Console01 {
    public static void main(String[] args) throws Exception {
        int zeichen1 = System.in.read();
        System.out.println(zeichen1);
    }
}
```

Das Programm *Console01* erwartet vom Bediener die Eingabe eines Zeichens. Mit der Methode *read()* wird versucht, aus dem Eingabestrom ein Zeichen zu lesen. Wenn der Stream leer sein sollte, stoppt das Programm und es beginnt die Arbeit erst dann wieder, wenn ein Zeichen über *System.in* eingegeben wurde und der Bediener die Entertaste gedrückt hat.

### Übung zum Programm *Console01*

Geben Sie das Zeichen ein, das als Antwort "50" ausgibt. Lösungshinweis: Bitte in der Unicode-Tabelle nachschauen, um zu ermitteln, welches Zeichen den numerischen Wert 50 hat.

Das nächste Beispiel demonstriert das Einlesen von mehreren Zeichen. An dieser Stelle fehlen noch Kenntnisse zur Schleifenbildung in Java. Natürlich wird das Einlesen von *mehreren* Zeichen oder Zeilen z.B. durch einen *while*-Befehl realisiert, und praktische Beispiele dazu finden sich reichlich im Kapitel 8. Im Augenblick geht es lediglich um das Lesen und die Interpretation von Bildschirmeingaben.

### Programm *Console02*: Lesen von Steuerzeichen (CR/LF) von *System.in*

```
import java.io.*;
public class Console02 {
    public static void main(String[] args) throws Exception {
        int zeichen1 = System.in.read();
        int zeichen2 = System.in.read();
        int zeichen3 = System.in.read();
        System.out.printf("%d %d %d", zeichen1, zeichen2, zeichen3);
    }
}
```

### Übung zum Programm *Console02*

Bitte starten Sie das Programm und tippen Sie lediglich *ein* Zeichen ein. Wenn Sie danach die Enter-Taste drücken, werden trotzdem drei Zeichen eingelesen und am Bildschirm ausgegeben. Prüfen Sie anhand der Unicode-Tabelle im Anhang C, um welche Zeichen es sich handelt.

#### 6.2.1 Arbeiten mit der Klasse *Scanner*

Es gibt natürlich auch Situationen, wo nicht einzelne Zeichen eingelesen und interpretiert werden müssen, sondern das Programm typisierte Daten (also *int*, *float* usw.) erwartet. Es kann auch sein, dass eventuell vorhandene Steuerzeichen erkannt und automatisch interpretiert werden sollen. Die Standard-Bibliothek von J2SE enthält die Klasse *Scanner*, die eine komfortable Möglichkeit bietet, Texte und Java-interne Datentypen einzulesen.

Die eingelesenen Daten werden von Methoden dieser Klasse analysiert und nach vorgegebenen Regeln interpretiert. So können komplette Zeilen eingelesen und ihr Inhalt richtig erkannt werden. Dieser Vorgang wird "Parsen" oder "Scannen" genannt. Beispielsweise kann eine Zeile neben ganzen Wörtern auch die Java-internen Datentypen wie *int* oder *double* enthalten. Die Methoden der Klasse *Scanner* unterscheiden die einzelnen Wörter (hier "Token" genannt) anhand von Trennzeichen, die frei wählbar sind. Standardmäßig werden die einzelnen Token einer Zeile abgetrennt durch Leerstellen ("blank").

Mit dem folgenden Beispielprogramm wird eine komplette Zeile vom Bildschirm gelesen. Dabei wird davon ausgegangen, dass diese Zeile folgenden Aufbau hat: zunächst wird eine beliebige Zeichenfolge (String) erwartet; beendet wird dieser String beim Auftreten des Trennzeichens Blank. Dann folgt als nächstes Token eine ganze Zahl, und das letzte Token in dieser Zeile muss eine Gleitkommazahl in wissenschaftlicher Schreibweise (mit dem Buchstaben *e*) sein.

#### Programm *Scanner01*: Eingabe von typisierten Daten (*String*, *int* und *double*)

```
import java.util.Scanner;
public class Scanner01 {
    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        String str = eingabe.next();           // Komplettes Wort lesen
        int zahl1 = eingabe.nextInt();         // Ganzzahl lesen
        double zahl2 = eingabe.nextDouble();  // E-Format z.B. 5e3 lesen
        System.out.printf("%s | %d | %f", str, zahl1, zahl2);
    }
}
```

### Übungen zum Programm *Scanner01*

**Übung 1:** Eine korrekte Eingabezeile sieht etwa so aus: `aaaa 4700 5e3`. Können die drei Daten auch in jeweils einer eigenen Zeile stehen?

**Übung 2:** Was passiert, wenn "falsche" Daten im Eingabestrom stehen?

## 6.2.2 Fehlerbehandlung und Prüfungen

Generell wird das Thema Fehlerbehandlung in diesem Buch nicht besprochen, denn die Programmierung von Routinen zur Fehlerbehandlung ("Exceptionhandling") erfordert in Java einige tiefergehende Kenntnisse der Objektorientierung. Und vor allem würde die Einfachheit der Programme durch die Programmierung der Fehlerbehandlung leiden und dem Einsteiger die Übersicht erschweren.

Dennoch werden wir mit dem nachfolgenden Programm eine kurze Einführung in dieses Thema geben. Das Programm *TryCatch01.java* liest von der Console das erste Token ein und prüft, ob der eingegebene Wert numerisch ist. Wenn nicht, wird eine Fehlermeldung ausgegeben, andernfalls wird "ok" ausgegeben.

### Programm *TryCatch01*: Lesen mit Formalprüfung (auf numerisch)

```
import java.util.Scanner;
public class TryCatch01 {
    public static void main(String[] args) {
        int zahl;
        Scanner eingabe = new Scanner(System.in);
        String str = eingabe.next();          // Erstes Wort lesen
        try {
            zahl = Integer.parseInt(str);     // Konvertieren in Ganzzahl
        }
        catch (NumberFormatException e) {
            System.out.println("Es wurde keine Ganzzahl eingegeben");
            System.exit(8);
        }
        System.out.println("Der eingegebene Text ist ok");
    }
}
```

Die Behandlung von Fehlern erfolgt in Java mit Hilfe der *try-catch*-Anweisung. Innerhalb des *try*-Block werden die fehlerträchtigen Anweisungen ausgeführt, und von dem *catch*-Block werden eventuell aufgetretene Fehler behandelt. Wir haben in allen anderen Programmen in diesem Buch auf diese Art der Ausnahmebehandlung verzichtet und haben durch die Klausel "**throws Exception**" im Kopf der Methode festgelegt, dass mögliche Fehler vom Laufzeit-System (und nicht individuell) behandelt werden sollen.

### Übung zum Programm *TryCatch01*

Ändern Sie das Programm so, dass auf eine individuelle Fehlerbehandlung verzichtet wird. Testen Sie das Programm und beobachten Sie das Systemverhalten. Was passiert, wenn der Bediener falsche Daten eintippt?

### Programm *Scanner03*: Lösungsvorschlag für den Verzicht auf individuelle Fehlerbehandlung

```
import java.util.Scanner;
public class Scanner03 {
    public static void main(String[] args) throws Exception {
        int zahl;
        Scanner eingabe = new Scanner(System.in);
        String str = eingabe.next();          // Erstes Wort lesen
        zahl = Integer.parseInt(str);         // Konvertieren in Ganzzahl
        System.out.println("Der eingegebene Text ist ok");
    }
}
```

Die Fehlerbehandlung hat in der Praxis eine sehr große Bedeutung, und die Empfehlung an den Programmierer lautet: Jeder denkbare Fehlerfall muss vom Programm individuell behandelt werden (mit *try-catch*).

### 6.2.3 Ändern des Delimiter (des Begrenzungszeichens) zum Auftrennen der Begriffe

Die Klasse *Scanner* bietet dem Programmierer die Möglichkeit, den Delimiter (Trennzeichen) je nach Situation zu ändern. Das Trennzeichen für einzelne Token kann durch Aufruf der Methode *useDelimiter* geändert werden. Sehr häufig werden die einzelnen Begriffe der Eingabedaten durch Komma abgetrennt, und das folgende Programm zeigt eine Lösung für diese Aufgabenstellung.

### Programm *Scanner 04*: Ändern des Begrenzungszeichens

```
import java.util.Scanner;
public class Scanner04 {
    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        eingabe.useDelimiter(",");           // Delimiter aendern
        String str = eingabe.next();          // Komplettes Wort lesen
        int zahl1 = eingabe.nextInt();        // Ganzzahl lesen
        double zahl2 = eingabe.nextDouble(); // E-Format z.B. 5e3 lesen
        System.out.printf("%s | %d | %f", str, zahl1, zahl2);
    }
}
```

Beispiel für eine korrekte Eingabezeile:

```
Javabuch,17,3e5,
```

Das letzte Komma in der Eingabezeile ist zwingend notwendig, sonst wird der Eingabevorgang des letzten Token nicht beendet. Außerdem dürfen in diesem (sehr einfachen) Beispiel keine Leerzeichen zwischen den Token stehen. Natürlich ist auch hier eine flexible Eingabegestaltung möglich, aber das ist ein Thema für spätere Kapitel. Denn die *Scanner*-Klasse ist besonders komfortabel im Zusammenhang mit "Regulären Ausdrücken", siehe hierzu Kapitel 14.

### Wie wird das Zeilenende erkannt?

Beim Einlesen einer kompletten Zeile ist folgendes Problem zu lösen: das Zeilenende wird auf unterschiedlichen Plattformen unterschiedlich dargestellt (in MS-Windows durch CR/LF, in Unix durch CR und in MAC-OS durch LF). Auch dafür bietet Java eine Lösung. In so genannten Systemproperties sind diese Eigenschaften für die jeweilige Plattform gespeichert, und es gibt Methoden, diese abzufragen. Das folgende Beispiel zeigt eine Anwendung.

### Programm *Scanner05*: Plattformunabhängiges Einlesen von Zeilen

```
import java.util.Scanner;
public class Scanner05 {
    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        String zeilenende = System.getProperty("line.separator");
        eingabe.useDelimiter(zeilenende); // Delimiter aendern
        String zeile1 = eingabe.next();    // Komplettes Zeile lesen
        String zeile2 = eingabe.next();    // nächste Zeile lesen
        System.out.printf("%s %s", zeile1, zeile2);
    }
}
```

## 6.3 Standard-Ausgabe

Wird der Bildschirm als Standard-Ausgabeeinheit vom Programm angesprochen, so ist damit ein eigenes Fenster gemeint. Dieses arbeitet zeilenorientiert, und das bedeutet, dass nicht ein einzelnes, grafisches Pixel adressiert werden kann, sondern dass mit Zeilen und Spalten für die Zeichendarstellung gearbeitet wird. In einer MS-Windowsumgebung ist dies die "Eingabeaufforderung" (DOS-Box).

Die Ausgabe über die Standard-Ausgabeeinheit ist sehr komfortabel. So stehen folgende Methoden zur Verfügung:

- *System.out.println()* = für die Ausgabe von Java-Datentypen in eine Zeile. Danach wird auf eine neue Zeile vorgeschoben.

- `System.out.print()` = für die Ausgabe von Java-Datentypen in eine Zeile, ohne dass danach auf eine neue Zeile vorgeschoben wird
- `System.out.format()` = für die formatierte Ausgabe von Java-Datentypen (wie `printf`)
- `System.out.printf()` = für die formatierte Ausgabe von Java-Datentypen (wie `format`)

Ein Statement mit den Bezeichnern `System.out` bewirkt, dass die Klasse `System` geladen wird. In dem `static`-Feld `out` steht eine Referenz auf die aktuelle Standard-Ausgabeeinheit. Ein Umleiten ist jederzeit möglich, wie das folgende Programm zeigt.

#### Programm *Redirect01*: Umleiten der Standard-Ausgabe in Datei

```
import java.io.*;

public class Redirect01 {
    public static void main(String[] args) throws Exception {
        System.setOut(new PrintStream(new FileOutputStream("a.txt")));
        System.out.println("Der Text wird umgeleitet in Datei");
    }
}
```

##### 6.3.1 Arbeiten mit `println` und `print`

Das Ergebnis einer Ausgabe mit den verschiedenen `print`- bzw. `format`-Methoden ist immer ein String. **Typisierte Daten** wie `int` oder `float` werden vor der eigentlichen Ausgabe automatisch in eine String-Repräsentation umgewandelt.

#### Programm *Print01*: Ausgabe von Java-Datentypen auf Console

```
public class Print01 {
    public static void main(String args[]){
        int monat= 5;
        float gehalt = 2123.45f;
        String s1= "Im Monat ";
        String s2= "haben Sie ";

        System.out.print(s1 + monat + " ");
        System.out.print(s2 + gehalt + " ");
        System.out.println("verdient");
    }
}
```

Der Ausgabestrom kann also aus einer Kombination von beliebigen Java-Datentypen bestehen. Die primitiven Datentypen werden automatisch in einen String umgewan-

delt, und das Plus-Zeichen sorgt für das Konkatenieren, fügt also die einzelnen Werte so aneinander, dass *ein* Ausgabestring entsteht.

Wenn ein Ausgabestrom dagegen **Referenz-Datentypen** enthält, so muss der Programmierer sicherstellen, dass die Umwandlung in String erfolgt. Dies kann dadurch geschehen, dass er eine Methode *toString()* programmiert, diese wird dann automatisch aufgerufen und ausgeführt. Andernfalls wird eine mitgelieferte Standardmethode *toString* ausgeführt. Dieses Thema wird in Kapitel 14 detailliert besprochen.

### 6.3.2 Arbeiten mit class *Formatter*

Soll die Ausgabe der Daten in besonderer Weise formatiert erfolgen, so sollte mit der Methode *format()* der Klasse *Formatter* oder wahlweise auch mit *printf()* gearbeitet werden. Diese beiden Methoden bieten identische Möglichkeiten zum Formatieren und Aufbereiten der Ausgabe. Es können z.B. Angaben zur Länge, zum Einfügen von führenden Leerstellen, zur rechts- oder linksbündigen Aufbereitung, zu der Anzahl der Dezimalstellen und einiges mehr gemacht werden.

#### Programm *Format01*: Gleitkomma mit der Ausgabemethode *format()*

```
public class Format01 {
    public static void main(String args[]) {
        float gehalt = 2000.00f;
        System.out.format("%f \n", gehalt);
        System.out.format("%e \n", gehalt);
        System.out.format("%g \n", gehalt);
    }
}
```

#### Übung zum Programm *Format01*

Ersetzen Sie den Methodenname *format* durch *printf* (die beiden Methoden haben den gleichen Leistungsumfang). Ändert das etwas an der Ausgabe?

Auch der syntaktische Aufbau der beiden Methoden ist gleich:

```
format(String format, Object ... args);
printf(String format, Object ... args);
```

Erläuterungen: Beide Methoden erwarten zwei Argumente, die durch Komma getrennt werden. Das erste Argument ist ein String, der Formatierungsangaben enthält, wahlweise gemischt mit Literalen. Das zweite Argument hat eine **variable Anzahl** (festgelegt durch die drei Punkte) und umfasst die zu formatierenden Objekte.

Die Formatierungselemente beginnen mit dem %-Zeichen, danach muss mindestens ein Zeichen stehen, das die Art der Konvertierung angibt, z.B.

f für float,  
t für time,  
d für decimal oder  
s für String.

Zusätzlich können aber auch Angaben zur Größe gemacht werden, z.B.

.2 für zwei Stellen hinter dem Komma  
6.2 für sechs Stellen vor, zwei Stellen hinter dem Komma.

Im folgenden Programm soll die Ganzzahl 90 in unterschiedlichen Varianten am Bildschirm ausgegeben werden (als Dezimalzahl, als Oktalzahl und zweimal als Hexadezimalzahl, jeweils getrennt durch Schrägstrich).

### Programm *Format02*: Ganzzahl mit der Ausgabemethode *format()*

```
import java.io.*;
public class Format02 {
    public static void main(String args[]) {
        int zahl = 90;
        System.out.format("%1$d / %o / %x / %<X ", zahl);
    }
}
```

### Übung zum Programm *Format02*

Bitte ändern Sie das Programm so, dass als Trennzeichen nicht ein Schrägstrich, sondern ein Komma ausgegeben wird.

Zum Abschluss soll ein Beispiel die Ausgabe von Literalen und typisierten Variablen demonstrieren.

### Programm *Format03*: Formatierung mit Literalen und typisierten Variablen

```
import java.io.*;
public class Format03 {
    public static void main(String args[]) {
        int monat1 = 5;
        float gehalt1 = 2123.45f;
        String s1 = "Im Monat ";
        String s2 = "haben Sie ";
        String s3 = "%s %02d %s %4.2f verdient\n";
        System.out.format(s3, s1, monat1, s2, gehalt1);
        System.out.format(s3, s1, 11, s2, 983f);
    }
}
```



## 6.4 Dateiverarbeitung

Das Streamkonzept wird auch beim Lesen oder Schreiben von externen Datenträgern (Platten- oder DVD-Dateien) eingesetzt. Allerdings gibt es eine große Fülle von Klassen, die jeweils spezialisiert sind und für ergänzende Aufgaben ihre Dienste anbieten. Das Arbeiten mit diesen Klassen ist voll objektorientiert. Deswegen noch einmal der Hinweis: Es fehlen an dieser Stelle noch einige Voraussetzungen, um alle Codiertechniken der folgenden Beispiele zu verstehen. Trotzdem wollen wir demonstrieren, wie Daten auf einen externen Speicher, z.B. in eine Magnetplattendatei, geschrieben (man sagt: "persistent" gemacht) werden und wie diese anschließend wieder gelesen werden können. Das Kapitel kann also vom Einsteiger zunächst überflogen und später bei Bedarf nachgearbeitet werden.

Dieser Abschnitt enthält drei Themenbereiche:

- Die beiden Programme *Datei01* und *Datei02* demonstrieren das Schreiben und Lesen von Dateien mit Hilfe von Stream-Klassen. Weil die Datendarstellung in einem [Java-spezifischen Format](#) erfolgt, ist diese Art der Dateiverarbeitung jedoch nur sinnvoll, wenn kein Datenaustausch mit anderen Systemen notwendig ist (oder wenn lediglich mit rein binären Daten und nicht mit Texten und Zeichen gearbeitet wird).
- Die nächsten vier Programme *Konvert01-04* zeigen, dass ein Dateiinhalt unterschiedlich interpretiert werden kann: im ersten Programm wird eine Datei erzeugt, die einen Integerwert enthält. In den folgenden Programmen werden die vier *int*-Bytes aber als einzelne Bytes interpretiert und "encodiert" in Unicode und zum Schluss wieder konvertiert in ASCII-Code. Dazu ist es notwendig, so genannte [Brückenklassen](#) einzusetzen, die als Transformationshilfe zwischen der ASCII- und der Unicodewelt fungieren.
- In den beiden letzten Programm *Encode01-02* vertiefen wir die gewonnenen Einsichten zum Thema "Wie werden aus ASCII-Zeichen Unicode-Zeichen?". Die Programme werden zeigen, dass dies über den Einsatz der richtigen [Encoding-Tabelle](#) gesteuert wird.

### 6.4.1 Lesen und Schreiben in Datei

Nun zum ersten Thema: Welche Möglichkeiten bieten die Streamklassen, um Daten in Dateien zu schreiben? Das Programm soll sowohl primitive Variablen (*int*-Werte) als auch Texte in die Datei *"test1.dat"* ausgeben.

#### [Programm Datei01: Schreiben von typisierten Variablen in eine Datei](#)

```
import java.io.*;
public class Datei01 {
    public static void main(String[] args) throws Exception {
        OutputStream aus = new FileOutputStream("test1.dat");
        DataOutput datenAus = new DataOutputStream(aus);
```

```
    datenAus.writeInt(4700);  
    datenAus.writeUTF("Merker");  
    datenAus.writeUTF("Steinfurt");  
}  
}
```

Das Programm *Datei01* benutzt die Klasse *DataOutputStream*. Diese enthält für alle eingebauten Java-Datentypen eigene Ausgabemethoden, z.B. für Ganzzahlen die Methode *writeInt*. Außerdem können Unicode-Zeichen einzeln (mit *writeChar*) oder als String (mit *writeChars*) ausgegeben werden. Dabei belegt jedes Zeichen zwei Bytes in der Ausgabedatei.

### Was bedeutet UTF?

Im Programm *Datei01* benutzen wir die Methode *writeUTF* für die String-Ausgabe, und die hat eine Besonderheit: Um in der Ausgabedatei Platz zu sparen, wird jedes Unicodezeichen nach bestimmten Verfahren transformiert, bevor es in den Ausgabestrom geschrieben wird. Dabei gelten folgende Transformationsregeln:

- Zeichen im Bereich `\u0001` - `\u007f` benötigen 1 Byte
- Zeichen im Bereich `\u0080` - `\u07ff` benötigen 2 Bytes
- Zeichen im Bereich `\u0800` - `\uffff` benötigen 3 Bytes.

Anders gesagt, die ASCII-Zeichen benötigen wenig Platz, alle anderen entsprechend mehr. (Wir werden später das UTF-8-Format kennen lernen. - ein Standardformat, das mit den *Reader-/Writer*-Klassen von Java übernommen wurde. Die Transformation mit *writeUTF* entspricht leider nicht exakt diesem UTF-8-Standard, denn es handelt sich hierbei noch um ein Java-internes Format.)

### Was bedeutet Little/Big Endian ?

Der Vollständigkeit halber soll an dieser Stelle noch auf ein weiteres Kompatibilitätsproblem hingewiesen werden. Immer wenn mehrere Bytes nicht einzeln, sondern im Verbund interpretiert werden müssen (z.B. bei Integer-Variablen oder bei Unicode-Zeichen, die mit zwei oder drei Bytes dargestellt werden), müssen Sender und Empfänger eines Streams sich einigen, wie die Byte-Reihenfolge ist: stehen die höherwertigen Bytes vorne oder hinten? Der Fachausdruck dafür ist: Little Endian oder Big Endian. Auch dies ist im Unicode plattform-unabhängig standardisiert. Wir werden darauf zurückkommen.

Nachdem das Programm *Datei01* ausgeführt wurde, ist die Datei *test1.dat* in demselben Ordner angelegt, in dem sich die Class-Datei befindet. Die Datei enthält 23 Bytes, die sich zusammensetzen aus 4 Bytes Ganzzahl, 6 und 9 Bytes für die Strings, und zusätzlich enthält jeder String 2 Zeichen für die Stringlänge. Zumindest die Textdaten kann man mit einem beliebigen Editor (oder auch an der Console mit einem Betriebssystembefehl z.B. *type*) anzeigen. Java arbeitet standardmäßig mit dem Big-Endian-Format.

### Übungen zum Programm *Datei01*

**Übung 1:** Klären Sie folgende Fragen durch Ausprobieren:

- a) Was passiert, wenn Sie dieses Programm erneut aufrufen? Sind die Daten in der Datei zweimal vorhanden oder wird die Datei überschrieben?
- b) Wo steht die Datei, wenn Sie in der vierten Zeile innerhalb der Anführungsstriche folgendes angeben: "c:\test1.dat" ?

**Übung 2:** Ändern Sie die Ausgabemethode in *writeBytes()*. Wie ändert sich der Inhalt der Ausgabedatei?

Anschließend wollen wir ein Programm codieren, das die Daten aus der Datei *test1.dat* wieder in den Arbeitsspeicher einliest und von dort in einem Konsolfenster ausgibt.

### Programm *Datei02*: Einlesen von typisierten Variablen aus einer Datei

```
import java.io.*;
public class Datei02 {
    public static void main(String[] args) throws Exception {
        InputStream ein = new FileInputStream("test1.dat");
        DataInput datenEin = new DataInputStream(ein);
        int plz = datenEin.readInt();
        String kdname = datenEin.readUTF();
        String ort = datenEin.readUTF();
        System.out.println(plz + " " + kdname + " " + ort);
    }
}
```

### Übungen zum Programm *Datei02*

**Übung 1:** Ändern Sie zunächst das Schreib-Programm *Datei01.java* so ab, dass eine zweite Adresse in die Datei geschrieben wird.

**Übung 2:** Danach soll auch das Lese-Programm *Datei02.java* so geändert werden, dass diese zweite Adresse eingelesen und angezeigt wird.

## 6.4.2 Unicode und die unterschiedliche Interpretation von Bitkombinationen

Der Unicode ist ein Regelwerk, wie "jedem Zeichen dieser Welt" ein numerischer Wert, eine Platznummer, zugeordnet wird. Diese Platznummer wird Codepoint genannt. Die real existierenden Betriebssysteme, Ein-/Ausgabegeräte und Programmiersprachen arbeiten jedoch noch nicht vollständig mit dem Unicode. In vielen Bereichen gilt nach wie vor die 8-Bit-Codierung. Das bedeutet, dass der Austausch von Daten zwischen verschiedenen Plattformen in der Praxis noch für längere Zeit ein heikles Thema bleiben wird.

### Warum ist das "Encoding" ein wichtiges Thema?

Ob moderne verteilte Anwendungen erstellt werden sollen oder ob Daten mit so genannten Legacy-Anwendungen (Alt-Systeme z.B. von COBOL, RPG oder auch von hierarchischen Datenbanken) ausgetauscht werden, immer ist ein tiefgehendes Verständnis für die interne Darstellung der Daten und für das Arbeiten mit diversen Codpages und Charactersets notwendig. Eine Encoding-Tabelle spielt bei allen Anwendungen eine Rolle, die "internationalisiert" werden müssen - und dies ist zumindest bei den meisten Internet-Anwendungen erforderlich. Deshalb sind auch gerade die modernen Internettechnologien wie XML und HTTP und Java ausgestattet mit der Fähigkeit, unterschiedliche Encoding-Tabellen zu verarbeiten. Die Angabe, welches Encoding-Schema eingesetzt wird (z.B. "*characterset=iso-8859-3*"), kann in XML-Dateien, im HTTP-Header beim Arbeiten mit Formular-Parametern, in HTML-Dokumenten und natürlich in Java-Programmen stehen.

### Datei mit *int*-Wert erstellen und unterschiedlich interpretieren

Die nachfolgenden vier Programme demonstrieren kurz und übersichtlich, wie in Java durch Einsatz der characterorientierten Klassen *Reader* und *Writer* (in Zusammenarbeit mit den byteorientierten Streamklassen)

- ein einzelner Dezimalwert in einer Datei gespeichert wird (*Konvert01.java*),
- diese Zahl als vier Einzelzeichen interpretiert und als UTF-8-/UTF-16-Daten konvertiert in einer Datei gespeichert werden kann (*Konvert02.java*),
- diese Unicode-Datei wieder zurück in eine ASCII-Datei umgestellt werden kann (*Konvert03.java*) und wie zum Schluss
- die vier ASCII-Zeichen wieder als die Dezimalzahl interpretiert werden, die sie ursprünglich dargestellt haben (*Konvert04.java*).

### Programm *Konvert01*: Erstellen der Ausgangsdatei *Stream01.txt*

```
import java.io.*;
public class Konvert01 {
    public static void main(String[] args) throws Exception {
        OutputStream aus = new FileOutputStream("Stream01.txt");
        DataOutput ausgabe = new DataOutputStream(aus);
        ausgabe.writeInt(174);
    }
}
```

Die Methode *writeInt()* codiert die Dezimalzahl 174 als reine binäre Zahl und schreibt sie in die Datei *Stream01.txt*. In Java sind *int*-Typen immer 4 Bytes lang, diese 32 bits haben die Stellenwertigkeit des Binärsystems. Also hat die Datei folgenden Inhalt:

binär:	00000000	00000000	00000000	10101110
hexadezimal ausgedrückt:	00	00	00	AE

Dieser Inhalt kann auch als eine Folge von 4 einzelnen Bytes interpretiert werden. Das letzte Byte enthält 0xAE (hexadezimal AE), das ist ein Wert, der jenseits der ASCII-Standard-Codierung von 00 - 7F liegt. Und wir wissen: alle Zeichen, die oberhalb von 0x7F liegen, sind in ASCII nicht standardisiert. Hier ist entscheidend, mit welchem Characterset gearbeitet wird. Auf der MS-Windows-Plattform wird bei der Installation ein Standard-Characterset, abhängig vom Länderschlüssel, festgelegt. Dies ist in Westeuropa der Characterset CP-1252. Leider hat die DOS-Box unter MS-Windows eine davon abweichende Codierung, nämlich CP-850.

Im Anhang C sind beide Zeichencodierungen zusammen mit dem Unicode aufgeführt. Für die hex-Verschlüsselung AE finden wir dort folgende Zeile:

Codepoint:	UnicodeZeichen/Text:	CP-1252:	CP-850:
174 0x00AE	® REGISTERED SIGN	®	«

Im Unicode und im MS-Windows-Standard wird der Dezimalwert 174 also als "registered" (registriertes) Zeichen interpretiert; in einer DOS-Box dagegen wird der doppelte Pfeil ausgegeben.

### Übung zum Programm *Konvert01*

Bitte überprüfen Sie, wie der Inhalt der Datei *Stream01.txt* in

- einer DOS-Box (mit dem DOS-Command "type dateiname") und wie der Inhalt
- in einem Editor (z.B. MS-Editor oder MS-Word)

angezeigt wird.

### 6.4.3 Transformieren mit Brückenklassen und Encoding-Tabellen

Das nächste Programm soll den Inhalt der Datei als vier einzelne Bytes interpretieren und diese als Unicode-Zeichen mit Hilfe einer "Encoding-Tabelle" ausgeben. Die wichtigsten Hilfsklassen bei dieser Transformation sind die Brückenklassen zwischen der 8-bit-ASCII-Welt und der 16-bit-Unicodewelt (*InputStreamReader* und *OutputStreamWriter*). Und die wichtigste Aufgabe für den Programmierer ist die Auswahl der richtigen Encoding-Tabelle (Zeichensatz, Characterset). Die bekanntesten Zeichensätze sind:

ISO 646	Standard-US-ASCII (7-bit = 128 Zeichen von hex. 00 bis 7F)
ISO 8859-x	8-bit-ASCII (256 Zeichen, von hex. 80-FF, , Latin-x, mehrere Versionen)
CP1252	MS-Windows-Version für ISO 8859-1 (CP für Codepage)
CP850	MS-DOS (Zeichensatz für Consolefenster in Windows)
UTF-8	8-bit-Unicode, die ersten 255 Zeichen sind identisch mit ISO 8859-1
UTF-16	16-bit-Unicode (auch UCS-2), wird in Java verwendet
UTF-32	32-bit-Unicode (auch UCS-4)

Im UTF-16-Code ist das Unicode-Zeichen 65279 (hexadezimal feff) das "Byte Order Mark" (BOM). Damit kann festgelegt werden, ob die Byte-Ordnung der Big-Endian-

oder der Little-Endian-Variante entspricht. Darüber hinaus gibt es noch zwei Varianten von UTF-16, die sich auch mit diesem für die Portabilität der Daten so wichtigen Thema beschäftigen, nämlich

UTF-16-BE (BE für Big Endian, d.h. das höherwertige Byte zuerst) und

UTF-16-LE (LE für Little Endian, d.h. das niederwertige Byte zuerst).

Die Encoding-Tabelle kann bei der Erzeugung eines *InputStreamReader*-Objekts bzw. eines *OutputStreamWriter*-Objektes angegeben werden. Wenn die Angabe fehlt, wird mit einer Standard-Tabelle, die automatisch bei der Java-Installation festgelegt wird, gearbeitet.

### Programm *Konvert02*: Vier Bytes als vier Unicodezeichen "encodieren"

```
import java.io.*;
public class Konvert02 {
    public static void main(String[] args) throws Exception {
        BufferedReader eingabe = new BufferedReader(
            new InputStreamReader(new FileInputStream(
                "Stream01.txt"), "CP1252"));
        BufferedWriter ausgabe = new BufferedWriter(
            new OutputStreamWriter(new FileOutputStream(
                "Stream02.txt"), "UTF-16"));
        int zeichen = eingabe.read();
        ausgabe.write(zeichen);
        zeichen = eingabe.read();
        ausgabe.write(zeichen);
        zeichen = eingabe.read();
        ausgabe.write(zeichen);
        zeichen = eingabe.read();
        System.out.println((char) zeichen);
        ausgabe.close();
    }
}
```

In dem Programm *Konvert02* sorgen die Brückenklassen *InputStreamReader* bzw. *OutputStreamWriter* für das Encoding (von 8-bit-ASCII nach UTF-16 Unicode). Das Ergebnis dieser Konvertierung steht in der Datei *Stream02.txt*. Diese Datei ist insgesamt 10 Bytes groß, neben den 2 Bytes je Zeichen enthält sie auch ein Steuerzeichen.

### Übungen zum Programm *Konvert02*

**Übung 1:** Bitte editieren Sie die Datei *Stream02.txt* mit einem MS-Windows-Editor (z.B. Editor oder MS-Word). Angezeigt wird das Registerzeichen (laut Characterset UTF-16).

**Übung 2:** Bitte ändern Sie das Programm so, dass das vierte gelesene Zeichen am Bildschirm ausgegeben wird mit folgendem Befehl:

```
ausgabe.write((char) zeichen);
```

Das Ergebnis dieser Ausgabe ist der doppelte Pfeil (laut Characterset CP-850).

**Übung 3:** Prüfen Sie, wie sich der Dateinhalt der Ausgabedatei verändert, wenn Sie für die Ausgabe statt UTF-16 den Characterset UTF-8 benutzen.

**Übung 4:** Muss für die Eingabedatei der Characterset CP1252 zwingend angegeben werden (oder ist dies die Standardannahme)? Kann also die Angabe entfallen?

Das nächste Programm wird den Inhalt der Datei *Stream02.txt* als vier Unicode-Zeichen interpretieren und diese wieder zurückführen in ASCII-Daten.

### Programm *Konvert03*: Einlesen von 4 Unicodezeichen und in ASCII umwandeln

```
import java.io.*;
public class Konvert03 {
    public static void main(String[] args) throws Exception {
        BufferedReader eingabe = new BufferedReader(
            new InputStreamReader(new FileInputStream(
                "Stream02.txt"), "UTF-16"));
        BufferedWriter ausgabe = new BufferedWriter(
            new OutputStreamWriter(new FileOutputStream(
                "Stream03.txt"), "8859_1"));
        int zeichen = eingabe.read();
        ausgabe.write(zeichen);
        zeichen = eingabe.read();
        ausgabe.write(zeichen);
        zeichen = eingabe.read();
        ausgabe.write(zeichen);
        zeichen = eingabe.read();
        ausgabe.write(zeichen);
        ausgabe.close();
    }
}
```

Das Ergebnis nach Ausführung des Programms *Konvert03* steht in der Datei *Stream03.txt*. Aus den 10 Bytes der Eingabedatei wurden wieder die 4 Bytes, die auch schon die Datei *Stream01.txt* hatte.

Jetzt könnten wir in einem weiteren Programm diese 4 Bytes als Integerwert interpretieren und am Bildschirm als Dezimalzahl 174 ausgeben:

**Programm *Konvert04*: Interpretieren von 4 ASCII-Bytes als Integerwert**

```
import java.io.*;
public class Konvert04 {
    public static void main(String[] args) throws Exception {
        InputStream ein = new FileInputStream("Stream03.txt");
        DataInput eingabe = new DataInputStream(ein);
        int zeichen = eingabe.readInt();
        System.out.println(zeichen);
    }
}
```

**6.4.4 Beispielprogramme für Encoding und Characterset**

Zur Vertiefung dieses Themas folgen noch zwei Beispiele, die zeigen sollen, wie wichtig der richtige Einsatz des Encoding-Mechanismus ist.

Das erste Beispiel gibt einen Text aus, ohne besondere Hinweise zu einem Encoding; im zweiten Beispiel wird ein individuelles Encoding explizit angegeben.

**Programm *Encode01*: Arbeiten mit Standard-Encoding**

```
import java.io.*;
public class Encode01 {
    public static void main(String[] args) throws Exception {
        Writer writer = new FileWriter("ausgabe.txt");
        writer.write("Java\u1234");
        writer.close();
    }
}
```

Nach der Ausführung dieses Programms enthält die Ausgabedatei 5 Bytes, obwohl das letzte Zeichen ein Unicode-Zeichen ist (1234 ist der Codepunkt für das Promille-Zeichen). Die Datei enthält an dieser Stelle ein ? (Fragezeichen) - und das ist das Standardzeichen für alle Unicodewerte, die nicht im ASCII-Code enthalten sind.

Die korrekte Ausgabe von Unicode-Zeichen, die nicht im Bereich der 128 ASCII-Zeichen liegen, erfordert die ausdrückliche Angabe einer Encoding-Tabelle. Zusätzlich muss die Ausgabeklasse geändert werden. Das nachfolgende Programm demonstriert, wie beliebige Unicodezeichen in eine Datei ausgegeben werden können.

**Programm *Encode02*: Arbeiten mit individuellem Encoding-Schema durch Angabe des Charactersets**

```
import java.io.*;
public class Encode02 {
    public static void main(String[] args) throws Exception {
        FileOutputStream fos = new FileOutputStream("aus.txt");
```



```
Writer writer = new OutputStreamWriter(fos, "UTF-8");
writer.write("testing\u2030");
writer.close();
}
}
```

### Klasse *OutputStreamWriter*

Das obige Programm benutzt die bereits mehrfach erwähnte Klasse *OutputStreamWriter* für die Ausgabe. Diese Klasse bildet eine Brücke zwischen der Unicode-Welt und ASCII-Welt. Bei der Ausgabe von Character- oder Stringwerten werden die Zeichen anhand einer Umwandlungstabelle neu verschlüsselt. Für die Ausgabe werden die intern benutzten Unicode-Zeichen in das Format konvertiert, das durch die Encoding-Tabelle spezifiziert wird. Fehlt diese Angabe, so wird der Defaultwert des darunter liegende Betriebssystems benutzt (meistens ASCII-Zeichen). Für die Eingabe gibt es denselben Mechanismus: Mit Methoden der Klasse *InputStreamReader* werden die eingelesenen ASCII-Informationen umgesetzt in Unicode-Zeichen.

Die Umwandlungstabelle ("encoding schema") hat in diesem Programm die Bezeichnung "UTF-8", damit ist eine Variante des Unicoes gemeint, der die 16-bit langen Einzelzeichen umformt in 1, 2, 3 oder 4-Byte lange Zeichen, abhängig von der Häufigkeit des Auftretens. So werden alle Zeichen, die im ASCII-Code enthalten sind, in 8-bit-Verschlüsselungen umgeformt. Chinesische Zeichen dagegen werden umgeformt in 24-bit/32-bit-Verschlüsselungen. Warum diese zusätzliche Umformung der Unicode-Zeichen bei der Ausgabe? Der Grund ist Speicherplatzeinsparung: häufig benutzte Zeichen können komprimiert werden, selten benutzte dagegen benötigen mehr Platz.

### Übungen zum Programm *Encode02*

**Übung 1:** Prüfen Sie den Inhalt der Datei *aus.txt*. Wieviel Bytes enthält die Datei? Lassen Sie sich den Inhalt in einem Unicode-fähigen Textprogramm (z.B. MS-Word) anzeigen. Das Ergebnis muss so aussehen: "Java%"

Natürlich ist es auch möglich, den Inhalt der Variablen in der JVM als "normalen" Unicode (also 16-bit pro Zeichen) ausgeben zu lassen. Dazu muss lediglich das richtige Encoding angegeben werden.

**Übung 2:** Ändern Sie den Namen der Encoding-Tabelle von UTF-8 auf UTF-16 und starten Sie das Programm erneut. Prüfen Sie danach, wie lang die Ausgabedatei nun ist und lassen Sie sich den Inhalt über MS-Word anzeigen.

**Übung 3:** Ändern Sie das Programm so ab, dass ein Unicodezeichen aus dem Bereich 00-7F ausgegeben wird, z.B: mit folgender Zeile:

```
writer.write(0x0078);
```

Prüfen Sie die Ausgabedatei und kontrollieren Sie, ob das Zeichen der Unicode-Tabelle entspricht.

**Übung 4:** Ändern Sie das Programm so ab, dass eine *int*-Variable definiert und ihr der Dezimalwert 184 zugewiesen wird. Geben Sie dann diese Variable in die Datei aus und kontrollieren Sie das Ergebnis wiederum mit einem Unicode-fähigen Textprogramm.

#### 6.4.5 Hinweise zum UTF-8

Der Standard UTF-8 (Unicode Transformation Format) beschreibt, wie Unicode-Zeichen so transformiert werden, dass ASCII-Zeichen in *einem* Byte und alle anderen in 2 bis 4 Byte verschlüsselt werden. Dieses Format ist das Standard-Format für den Austausch von Unicode-Daten. Es kann genutzt werden, um Unicode-Daten in Dateien zu schreiben und auch für den Transport der Daten z.B. im Internet. Der Vorteil liegt im geringeren Platzbedarf gegenüber dem normalen Unicode, soweit es sich um amerikanische Texte handelt, denn diese werden auf ein Byte zurückgeführt.

Die Transformation von Unicode, der innerhalb der JVM zum Einsatz kommt, in UTF-8 für die externe Darstellung erfolgt nach einem ausgeklügelten Algorithmus. So gelten beispielsweise folgende Regeln:

- liegt der Wertebereich des UTF-16-Zeichens zwischen **00 und 7f**, wird es transformiert in 1 Byte mit folgendem Bitmuster: 0xxxxxxx
- liegt der Wertebereich des UTF-16-Zeichens zwischen **80 und 7ff**, wird es transformiert in 2 Bytes mit folgenden Bitmustern: 110xxxxx 10xxxxxx

Die wichtigste Information steht in den ersten Bits eines Bytes. Steht dort eine 0, so handelt es sich um eine 7-Bit-ASCII-Codierung; stehen dort 110, so handelt es sich um das 1. Byte einer 16-Bit-Darstellung usw. Die folgende Tabelle verdeutlicht diese Verschlüsselung.

Unicode	wird transformiert in UTF-8	
<i>dezimaler Wert</i>	<i>Erstes Bytes</i>	<i>Anzahl Bytes</i>
0 - 127	0xxxxxxx	1
128-2048	110xxxxx	2
2048-65.535	1110xxxx	3
65.535-131.071	11110xxx	4
usw.		

Abb. 6.3: Umformung von Unicode nach Standard-UTF-8

#### 6.4.6 Zusammenfassung

Die Input-/Output-Möglichkeiten in Java sind sehr umfassend, aber nicht unbedingt intuitiv verständlich. Um das Gesamtkonzept richtig zu verstehen, ist viel Erfahrung in objektorientierter Programmierung erforderlich.

Jede der mehr als 50 Klassen, die sich mit diesem Thema befassen, erfüllt eine spezielle Aufgabe, und manches Programmierproblem lässt sich nur durch das Zusammenspiel von mehreren Streamklassen lösen. So sind diese Klassen einerseits sehr stark über Vererbungstechniken miteinander verbunden, andererseits können sie sich gegenseitig benutzen, und auch eine Verknüpfung über Konstruktoren ist möglich.

Wir können an dieser Stelle nur eine grundsätzliche Einführung in das gesamte I/O-Konzept geben.

#### Streams beschreiben ein abstraktes Konzept für den Transport von Daten

Das wichtigste Einsatzgebiet von Streams ist der Transport von Daten zwischen einer JVM und beliebigen Peripheriegeräten (Konsole, Magnetplatte, TCP-Kommunikationsleitung usw.).

#### Unterscheidung zwischen byte- und characterorientierter Ein- und Ausgabe

Diese Unterscheidung ist für die Verarbeitung von codierten Zeichen wichtig, denn byteorientierte Verarbeitung erlaubt nur die Darstellung von 256 unterschiedlichen Zeichen, während die characterorientierten Klassen die Angabe einer **Encoding-Tabelle** (z.B. UTF-16, UTF-8 oder 8859-1) erlauben und damit den Unicode voll unterstützen.

#### Encoding

Weil außerhalb der JVM in den meisten Fällen mit Bytes für die Zeichendarstellung gearbeitet wird, ist ein Verständnis für die Konvertierung zwischen 8- und 16-bit-Darstellung wichtig. Wir haben die Bedeutung der Encoding-Tabellen und ihren Einsatz mit verschiedenen Beispielen demonstriert.

---

# 7

## Ausdrücke verstehen ("expression")

---

Unter einem Ausdruck ("expression") versteht man eine Verarbeitungsvorschrift, mit der ein Wert ermittelt wird. Im Java-Quelltext besteht ein Ausdruck aus **Operanden** (das können Variablen, Konstanten oder Literale sein), die durch **Operatoren** (z.B. + oder /) verknüpft werden. Hier einige [Beispiele für Ausdrücke](#):

```
zahl = 25           // Zuweisungsausdruck
a + 5 / b           // Mathematischer Ausdruck
a > 3               // Vergleichsausdruck
```

Ein Ausdruck ist kein selbstständiger Befehl, sondern Teil einer Anweisung. Die obigen Beispielausdrücke können entweder durch Anhängen eines Semikolons zu einer [vollständigen Anweisung](#) gemacht werden oder sie können innerhalb von anderen Anweisungen benutzt werden. Auch hierzu einige Beispiele:

```
zahl = 25;           // Zuweisungsanweisung
System.out.println(a + 5 / b); // Ausdruck als Methodenparameter
boolean b = a > 3;    // Ausdruck als Teil der Zuweisung
```

Ein Ausdruck kann überall da stehen, wo ein Wert benötigt wird. Er kann auch aus nur einem Operanden bestehen, im einfachsten Fall aus einer Variablen oder aus einem Literal. Das ist deswegen wichtig zu erwähnen, weil überall da, wo laut Syntaxbeschreibung ein Ausdruck erwartet wird, auch eine einzige Variable oder ein Literal stehen darf. Das Ergebnis eines Ausdrucks, der nur aus einer Variablen besteht, ist also der Wert dieser Variablen.

Ein Ausdruck kann auch aus einem Methodenaufruf bestehen. Die Operanden beim Methodenaufruf sind normalerweise Objekte. Eine Besonderheit ist, dass es Methodenaufrufe gibt, die *kein* Ergebnis liefern. Wir werden das Arbeiten mit Methoden im Kapitel 10 ausführlich besprechen.

[In diesem Kapitel geht es ausschließlich um Operanden aus einfachen Datentypen und um die verschiedenen Operatoren, die dazu dienen, diese Operanden zu verarbeiten. Sie werden die wichtigsten Arten von Ausdrücken kennen lernen und in vielen praktischen Beispielen anwenden.](#)

Die beiden **wichtigsten Einsatzgebiete** für Ausdrücke sind

- die Steuerung des Programmablaufs, z.B. in einer If-Anweisung: *if (a == b) ...*
- die Wertezuweisung, z.B. *a = 5 + y*.

Dies sind dann die Themen für das Kapitel 8.

## 7.1 Operanden und Operatoren

Ein Ausdruck soll einen Wert ausdrücken. Die Auswertung eines Ausdrucks liefert also ein (und nur *ein*) Ergebnis. Dieses Ergebnis muss vom Programm in irgendeiner Form weiterverarbeitet werden, andernfalls ist die Anweisung unvollständig und führt zu einem Umwandlungsfehler.

Ein Ausdruck kann mehrere Operanden haben, die dann verknüpft werden durch Operatoren. So entsteht ein komplexer Ausdruck.

### Programm *Ausdruck01*: Beispiel für einen komplexen Ausdruck

```
class Ausdruck01 {
    public static void main(String[] args) {
        double gehalt = 2000;
        gehalt = gehalt * 1.15 - 8;
        System.out.println("Das neue Gehalt ist: " + gehalt);
    }
}
```

Hinweis: In der ersten Zeile fehlt der Modifier *public*, das bedeutet, dass der Zugriff auf diese Klasse begrenzt ist auf das Package (weitere Erläuterungen siehe 16.7.3).

Die Zeile 4 enthält einen zusammengesetzten Ausdruck. Er besteht aus Operanden und aus Operatoren. Die Operatoren sind die beiden Symbole *\** und *-*. Die Operanden sind *gehalt*, *1.15* und *8*. Ein **Operator** ist ein spezielles Symbol (oder eine Kombination von mehreren Symbolen), das eine bestimmte Aktion veranlasst, z.B. eine Addition durch das Pluszeichen *+* oder ein Vergleich von Werten durch Größer- oder Kleinerzeichen *>* *<* ("was soll getan werden?"). Ein **Operand** ist eine Variable, eine Konstante oder ein Literal, mit dem gearbeitet wird ("womit soll etwas gemacht werden?").

### Ein Ausdruck hat einen Datentyp

Ein Ausdruck hat immer einen Datentyp, und das ist der Datentyp des Ergebnisses. Und dieser wird bestimmt vom Typ der einzelnen Operanden. Beispiel:

$5 + 7$

Hier ist der Fall klar. Der Datentyp dieses Ausdrucks ist *int* (Integer), weil beide Operanden vom Typ *int* sind. Etwas komplexer sind die Regeln, wenn die einzelnen Operanden unterschiedliche Datentypen haben. Dann werden sie "gleichnamig" gemacht, dazu später mehr.

### Ausdrücke mit Nebeneffekt

In Ausnahmefällen kann es sein, dass ein Ausdruck nicht nur einen neuen Wert ermittelt, der dann weiter verarbeitet wird, sondern dass gleichzeitig auch der Wert eines der Operanden im Speicher verändert wird, z.B. beim Inkrement *a++*;. Diese

Anweisung enthält den Ausdruck *a++*. Die Wirkung des Ausdrucks besteht darin, dass der Wert der Variablen *a* um 1 erhöht wird. Gleichzeitig hat er den Nebeneffekt, dass dieser neu ermittelte Wert auch der Variablen *a* zugewiesen wird. In solchen Fällen spricht man von einem "Ausdruck mit Nebeneffekt". Obwohl der Begriff Nebeneffekt in der Informatik häufig verbunden ist mit der Vorstellung von unerwünschten Nebenwirkungen bei der Verarbeitung von Variablen, ist in diesem Fall der Zusatzeffekt erwünscht.

### Reihenfolge der Auswertung

Die Auswertung von zusammengesetzten Ausdrücken geschieht in folgender Reihenfolge: Zunächst wird der Wert der einzelnen Operanden ermittelt (bei Variablen ist dies der augenblickliche Inhalt des Speicherplatzes dieser Variablen). Danach werden - normalerweise von links nach rechts - anhand der Operatoren die Ergebnisse der Teilausdrücke evaluiert. Beispiel:

```
gehalt * 1.15 - 8
```

In diesem Beispiel wird also zunächst der Wert der Variablen *gehalt*, dann das Ergebnis der Multiplikation und erst danach die Differenz ermittelt. Da in Java bei arithmetischen Operatoren die übliche Prioritätsreihenfolge ("Punktrechnung vor Strichrechnung") gilt, stimmt in diesem Fall die Regel, dass von links beginnend die einzelnen Teilausdrücke berechnet werden. Durch das Setzen von Klammern kann die Reihenfolge jedoch geändert werden. Beispiel:

```
gehalt * (1.15 - 8)
```

In diesem Fall wird zuerst die Differenz ermittelt und danach multipliziert. Die umfangreichen Regeln zur Reihenfolge der Evaluierungsschritte ("Präzedenz") werden wir später detailliert besprechen.

### Übung zum Programm *Ausdruck01*

Fügen Sie in den Ausdruck der Zeile 4 die runden Klammern ein. Vergleichen Sie das Ergebnis. Testen Sie danach andere Varianten der Klammersetzung.

Wie wichtig das Setzen von Klammern sein kann, demonstriert das folgende Programm. Durch die Klammerung der Multiplikation wird dieser Rechengang zuerst ausgeführt, bevor dann das Ergebnis in einen *int*-Typ konvertiert wird.

### Programm *Ausdruck02*: Klammern verändern die Auswertungsreihenfolge

```
class Ausdruck02 {
    public static void main(String[] args) {
        // int z1 = (int)1.23 * 100; // Zuerst 1.23 in int konvertieren
        int z1 = (int) (123 * 100); // Zuerst multiplizieren
        System.out.println(z1);
    }
}
```

## 7.2 Arithmetische Operatoren

### 7.2.1 Die Grundrechenarten

Es gibt für die vier Grundrechenarten jeweils einen Operator: + - \* /. Die Operanden müssen numerische Typen sein. Zu den numerischen Datentypen gehören alle primitiven Typen mit einer Ausnahme: Boolean-Typen sind nicht numerisch, mit ihnen kann man nicht rechnen.

#### Programm *Arithmetik01*: Beispiel für die vier Grundrechenarten

```
class Arithmetik01 {
    public static void main(String[] args) {
        float a = 25.7f;
        float b = 5f;
        short x = 212;
        int y = 148;

        System.out.println("x + y = " + (x + y));
        System.out.println("x - y = " + (x - y));
        System.out.println("x / y = " + (x / y));
        System.out.println("a / b = " + (a / b));
    }
}
```

#### Übungen zum Programm *Arithmetik01*

**Übung 1:** Starten Sie das Programm und überprüfen Sie die Ergebnisse. Ist das letzte Ergebnis korrekt? Wahrscheinlich wird ein Programmieranfänger zusammenzucken: Können Computer nicht rechnen? Wir werden Ursache und Lösung dieses Phänomens später besprechen.

**Übung 2:** Bitte ändern Sie das Programm so, dass die vier Variablen a, b x und y negative Initialwerte haben (z.B. -212. Überprüfen Sie die Ergebnisse.

Neben den vier Grundrechenarten gibt es noch einen zusätzlichen arithmetischen Operator: % (Prozentzeichen). Dieser wird auch Modulo- oder Rest-Operator genannt. Man kann dadurch den Rest einer Division ermitteln. Beispiel: Der ganzzahlige Rest der Division 5 geteilt durch 3 ist 2.

#### Programm *Arithmetik02*: Anwendung des Modulo-Operators

```
class Arithmetik02 {
    public static void main(String[] args) {
        int y = 15;
        System.out.println(y % 4);
    }
}
```

### Rechnen mit *byte*- oder *char*-Datentypen

Etwas ungewöhnlich ist sicherlich, dass auch mit Variablen der Datentypen *byte* oder *char* gerechnet werden kann. Technisch ist dies keine Besonderheit, denn auch diese Daten werden binär dargestellt und zwar abhängig von der Position des Zeichens im Unicode.

Auch wenn es wahrscheinlich nur in Ausnahmefällen sinnvoll ist, Rechenoperationen auf diese Datentypen anzuwenden, soll das nächste Beispiel eine mögliche Anwendung demonstrieren. Dabei benutzen wir ein so genanntes Inkrement. Dies erhöht den Inhalt einer numerischen Variablen um 1 (siehe unäre Operatoren).

#### Programm *Zeichen01*: Rechnen mit *char*-Variablen

```
public class Zeichen01 {
    public static void main(String args[]) {
        char zeichen = 'A';
        System.out.println(++zeichen);
    }
}
```

Der kleinste Datentyp, mit dem in Java gerechnet werden kann, ist *int*, d.h. schmalere Typen werden zu *int* konvertiert.

#### Programm *Byte01*: Ergebnisvariable hat falschen Datentyp

```
public class Byte01 {
    public static void main(String args[]) {
        byte b1 = 10;
        byte b2 = 11;
        byte erg = b1 + b2;
        System.out.println(erg);
    }
}
```

Wenn Sie versuchen, dieses Programm umzuwandeln, gibt es eine Fehlermeldung ("*possible loss of precision, found int, required byte*"). Sinngemäß bedeutet das, dass als Ergebnis der Addition in der Zeile 5 ein Integerwert gefunden wurde, deshalb muss auch die Ergebnisvariable mindestens den Typ *int* haben.

#### Übungen zum Programm *Byte01*

**Übung 1:** Bitte ändern Sie das Programm so, dass das richtige Ergebnis (21) ausgegeben wird.

**Übung 2:** Überlegen Sie, ob das obige Programm fehlerfrei umgewandelt werden kann, wenn der Datentyp aller drei Variablen von *byte* nach *char* geändert wird. Lösungshinweis: Eine *char*-Variable ist 2 Byte lang, eine *int*-Variable ist 4 Byte lang. Überprüfen Sie das Ergebnis Ihrer Denkarbeit.



### Rechnen mit Gleitkomma-Datentypen

Wenn bei ganzzahligen Datentypen eine Division durch Null versucht wird, liefert die Run-Time-Umgebung einen Laufzeitfehler ("Exception"). Bei *float*- und *double*-Zahlen dagegen bewirkt die Division durch Null keine Exception, sondern liefert als Ergebnis den speziellen Wert "unendlich" (infinity).

#### Programm Gleitkomma01: Falsche Operationen mit Gleitkomma-Zahlen

```
public class Gleitkomma01 {
    public static void main(String[] args) {
        double zahl1 = 15.21;
        double zahl2 = 0.0;
        double erg    = zahl1 / zahl2;
        System.out.println("Ergebnis ist: " + erg);
    }
}
```

Variable vom Typ Gleitkomma kennen neben "*infinity*" noch einen anderen speziellen Wert: "nicht definiert" bzw. "keine Zahl" (Not-a-Number NaN). Auch dieser Wert kann als Ergebnis von unzulässigen Rechengvorgängen entstehen. Beide Werte können abgefragt oder zugewiesen werden durch spezielle eingebaute Konstanten.

#### Programm Gleitkomma02: Spezielle Werte in Gleitkomma-Variablen

```
public class Gleitkomma02 {
    public static void main(String[] args) {
        double zahl1 = 0.0 / 0.0;
        float  zahl2 = Float.NaN;
        System.out.println(zahl1 + " " + zahl2);
    }
}
```

### Gemischte Datentypen in einem Ausdruck und "numeric promotion"

Ein Ausdruck liefert immer nur *ein* Ergebnis. Der Datentyp dieses Ergebnisses bestimmt somit den Datentyp des gesamten Ausdrucks. Das heißt, wenn das Ergebnis eine Integer-Variable ist, dann handelt es sich um einen Integer-Ausdruck, und wenn das Ergebnis eine Float-Variable ist, dann handelt es sich um einen Float-Ausdruck.

Diese Unterscheidung ist wichtig, weil davon der Datentyp einer möglichen Ergebnisvariablen abhängt.

Aber welcher Ergebnistyp gilt, wenn es sich um einen zusammengesetzten Ausdruck handelt, der Operanden mit unterschiedlichen Datentypen enthält? Generell hat dann das Ergebnis den größten Datentyp aller Operanden, d.h. es findet eine automatische Typenerweiterung statt - alle Operanden der Teilausdrücke werden angepasst ("gleichnamig gemacht").

Intern werden die Rechenvorgängen nur mit dem Datentyp *int* (oder größer) ausgeführt. Das heißt, kleiner als 4 Byte geht nicht! Wenn die Operanden vom unterschiedlichen Typ sind, wird mindestens in den *int*-Typ konvertiert. Dieser Vorgang wird auch als "**numeric promotion**" bezeichnet.

### Programm Byte02: Numeric Promotion bei unterschiedlichen Datentypen

```
public class Byte02 {
    public static void main(String args[]) {
        byte z1 = 10;
        z1 = z1 + 5;
        System.out.println(z1);
    }
}
```

### Übung zum Programm Byte02

Die Umwandlung dieses Programms endet mit einer Fehlermeldung. Bitte interpretieren Sie die Fehlermeldung und ändern Sie das Programm entsprechend.

## 7.2.2 Unäre arithmetische Operatoren

Java-Operatoren verlangen entweder ein, zwei oder drei Operanden. Die bisher besprochenen arithmetischen Operatoren sind so genannte **binäre** (dyadische) Operatoren, sie stehen zwischen zwei Operanden. Darüber hinaus gibt es **unäre** (monadische) Operatoren, die nur einen Operanden erfordern. Und es gibt sogar einen Operator, der drei Operanden erfordert, den Bedingungsoperator (siehe unter If-Befehl). Dieser wird **tenär** (triadisch) genannt. Unäre Operatoren verlangen also nur einen Operanden. Bei den arithmetischen Befehlen gibt es zwei unterschiedliche Ausprägungen der unären Operatoren: für die Vorzeichendarstellung und für die Inkrementbildung.

### Vorzeichen

Mit dem Minuszeichen kann also nicht nur die Differenz zwischen zwei Operanden ermittelt werden, es kann auch als unärer Operator unmittelbar vor einem Operanden stehen. Dann hat es die Bedeutung eines Vorzeichens.

### Programm Arithmetik05: Arbeiten mit Vorzeichen

```
public class Arithmetik05 {
    public static void main(String args[]) {
        byte z1 = -10;
        long z2 = -11;
        long erg = z1 + -z2;
        System.out.println(erg);
    }
}
```

### Inkrement/Dekrement

Für die Addition und Subtraktion gibt es abgekürzte Schreibweisen. Auch diese erfordern nur einen Operanden (und werden deshalb ebenfalls als unäre Operatoren bezeichnet). Es handelt sich um das Inkrement ++ (increment, engl. Zuwachs) und das Dekrement --.

Durch den Operator ++ wird der Inhalt einer Variablen um 1 erhöht. Ein typisches Einsatzgebiet ist die Schleifenbildung, bei der Laufvariablen rauf- oder runtergezählt werden (siehe Kapitel 8.5.3).

#### Programm Inkrement01: Arbeiten mit Inkrement

```
class Inkrement01 {
    public static void main(String[] args) {
        int zahl = 0;
        zahl++;          // Kurzschreibweise für: zahl = zahl + 1;
        System.out.println(zahl);
    }
}
```

Der Ausdruck `zahl++`; ist eine komplette Anweisung, denn er wird abgeschlossen mit einem Semikolon. Er hat dieselbe Wirkung wie die etwas ausführlichere Anweisung `zahl = zahl + 1`;

Durch den Operator -- wird der Inhalt einer Variablen um 1 reduziert.

#### Programm Dekrement01: Arbeiten mit Dekrement

```
class Dekrement01 {
    public static void main(String[] args) {
        int zahl = 0;
        zahl--;          // Kurzschreibweise für: zahl = zahl - 1;
        System.out.println(zahl);
    }
}
```

### Pre- oder Postfix

Beim Arbeiten mit Inkrement- bzw. Dekrement-Operatoren ist noch eine wichtige Besonderheit zu beachten. Beide Operatoren können sowohl in Präfix- als auch in der Postfix-Notation verwendet werden:

Präfix	Postfix
<code>++zahl</code>	<code>zahl++</code>
<code>--zahl</code>	<code>zahl--</code>

Man kann also den Operator vor oder hinter den Operanden schreiben. Und die Wirkung dieser unterschiedlichen Schreibweise ist auch unterschiedlich: In der Prä-

fix-Version wird die Variable zuerst verändert und dann benutzt, in der Postfix-Version wird sie erst benutzt und dann um 1 modifiziert. Praktische Auswirkung hat diese Unterscheidung nur dann, wenn das Hochzählen oder Runterzählen in einem zusammengesetzten Ausdruck erfolgt

### Programm Dekrement02: Präfix-Notation

```
class Dekrement02 {
    public static void main(String[] args) {
        int a = 1;
        System.out.println(--a);
    }
}
```

### Übungen zum Programm Dekrement02

**Übung 1:** Ändern Sie das Programm *Dekrement02.java* so, dass erst der aktuelle Variablenwert ausgegeben und danach die Rechenoperation ausgeführt wird.

**Übung 2:** Angenommen, Sie haben den zusammengesetzten Ausdruck  $9 / ++a$ , dann wird zuerst eine 1 auf den Wert der Variablen *a* addiert und erst danach die Division durchgeführt. Bitte testen Sie diese Anweisung mit dem Programm *Dekrement02*.

**Übung 3:** Dagegen wird in dem zusammengesetzten Ausdruck  $9 / a++$  zuerst die Division durchgeführt und anschließend *a* um 1 erhöht. Bitte testen Sie diese Anweisung mit dem Programm *Dekrement02* und vergleichen Sie die Ergebnisse.

Das folgende Programm demonstriert eine ähnliche Aufgabenstellung. Es zeigt noch einmal die Auswirkungen der unterschiedlichen Schreibweisen beim Inkrementieren. Und vor allem macht es deutlich, dass diese Ausdrücke schwer verständlich sind und deswegen besser vermieden werden sollten.

### Programm Inkrement02: Ausdrücke, schwer verständlich, nicht empfohlen!

```
public class Inkrement02 {
    public static void main(String args[]) {
        int x = 1;
        int y = 7 * ++x;    // Präfix, Erst erhöhen, dann rechnen
        System.out.println(y);

        int z = 7 * x++;    // Postfix Erst rechnen, dann erhöhen.
        System.out.println(z);
    }
}
```

Beide Multiplikationen lauten  $7 * 2$ , d.h. in beiden Fällen ist das Ergebnis 14. Trotzdem ist die Wirkung unterschiedlich.

### Übung zum Programm Inkrement02

Überlegen Sie, welchen Inhalt die Variablen `x` am Programmende hat. Überprüfen Sie Ihre Überlegung dadurch, indem Sie das Programm um entsprechende Ausgabeanweisungen ergänzen.

#### Beispiel zum Nebeneffekt

Die Inkrement- bzw. Dekrement-Operatoren haben noch eine weitere Besonderheit: sie bewirken einen "Nebeneffekt". Während ein Ausdruck normalerweise lediglich einen Wert ermittelt, aber keinen Variablenwert verändert (es sei denn, man benutzt ausdrücklich den Operator `=` für eine Wertezuweisung), haben die besprochenen Operatoren `++` und `--` eine zweifache Auswirkung: zum einen wird dadurch ein neuer Wert errechnet und außerdem (so ganz nebenbei, als Zusatzeffekt) dieser Wert auch der beteiligten Variablen zugewiesen.

### Programm Dekrement03: Wie wirkt der Nebeneffekt?

```
public class Dekrement03 {
    public static void main(String args[]) {
        int a = 0;
        System.out.println(a + 1);
        System.out.println(a++);
    }
}
```

### Übung zum Programm Dekrement03

Überlegen Sie, ob und wie der Inhalt der Variablen `a` sich zur Programmlaufzeit verändert. Verifizieren Sie dies durch einen zusätzlichen Ausgabebefehl.

## 7.2.3 Probleme beim Rechnen mit primitiven Datentypen

Der Begriff "primitive" Datentypen bedeutet eigentlich nicht, dass diese Typen wenig komfortabel sind, sondern er soll ausdrücken, dass es sich um die Basistypen einer Programmiersprache handelt und dass sie elementare Typen sind, die nicht weiter unterteilt werden können.

Aber beim Rechnen mit den eingebauten Typen sind einige Restriktionen zu beachten, die dazu führen können, dass diese "primitiven" Typen nicht in jedem Fall geeignet sind, um mathematische Aufgabenstellungen fehlerfrei zu lösen. Dann muss auf mitgelieferte Klassen zurückgegriffen werden (natürlich bedeutet der zusätzliche Komfort der Klassen auch Verlust an Performance, deswegen ist im Einzelfall immer abzuwägen, welche Lösung besser ist).

Insbesondere sind es drei Themenkreise, die problematisch sind und mit besonderer Sorgfalt behandelt werden müssen:

- Wie werden die Stellen **rechts vom Komma** behandelt? Wieviel Stellen stehen zur Verfügung? Wird abgeschnitten oder gerundet?
- Wie werden die Stellen **links vom Komma** behandelt? Was passiert, wenn der Wertebereich der Ergebnisvariablen nicht groß genug ist, um alle denkbaren Ergebnisse auch wirklich aufnehmen zu können? Wie reagiert das System bei einem so genannten "Überlauf"?
- Wie kann es zu **Ungenauigkeiten** beim Rechnen mit Gleitkommazahlen kommen? Wieso liefern die primitiven Datentypen eventuell merkwürdige Ergebnisse?

Wir werden diese Themen in den nachfolgenden Beispielen diskutieren, es werden falsche Beispiele gezeigt, aber Sie werden vor allem auch die korrekten Lösungen kennen lernen.

Eine wichtige Regel ganz zu Anfang:

Der Programmierer sollte für mathematische Berechnungen mit Wertangaben in Währungen (z.B. Euro-Beträgen) im Zweifel die mitgelieferte Klasse *BigDecimal* benutzen und nicht mit primitiven Datentypen arbeiten. Nur so kann sichergestellt werden, dass keine Ungenauigkeiten auftreten.

### 7.2.3.1 Datentyp falsch gewählt - oder: wieviel ist $1/3$ mal 3?

Das folgende Programm soll das Ergebnis aus  $1/3$  multipliziert mit 3 errechnen und ausgeben.

#### Programm *Arithmetik06*: 1. Versuch, leider ist das Ergebnis falsch

```
public class Arithmetik06 {
    public static void main(String args[]) {
        double d2 = 1/3 * 3;
        System.out.println(d2);
    }
}
```

Als Ergebnis wird 0.0 ausgegeben. Warum? Die Grund ist: Der Ausdruck wird von links nach rechts evaluiert. Der Teilausdruck  $1 / 3$  besteht aus Integerwerten, deswegen ist das Ergebnis die Ganzzahl 0. Und 0 multipliziert mit 3 ergibt 0.

Das nächste Programm zeigt eine mögliche Lösung dieses Problems. Jetzt sind auch die Zwischenergebnisse korrekt, weil die Operanden des Ausdrucks den korrekten Datentyp haben.

#### Programm *Arithmetik07*: 2. Versuch, jetzt ist alles richtig

```
public class Arithmetik07 {
    public static void main(String args[]) {
```

```
double d3 = 1d / 3d * 3;  
System.out.println(d3);  
}  
}
```

### Übung zum Programm *Arithmetik07*

Überprüfen Sie, ob das richtige Ergebnis auch zu erzielen ist durch folgende Anweisung: `double d3 = 1.0 / 3.0 * 3;` ?

#### 7.2.3.2 Überlauf-Probleme

Ein weiteres Problem kann entstehen, wenn das Ergebnis einer arithmetischen Operation mit Integerwerten zu groß ist für den gewählten Datentyp. Dann kommt es zum Überlauf von Integer-Werten.

#### Programm *Arithmetik08*: Multiplikationsergebnis ist 10 Mrd.

```
public class Arithmetik08 {  
    public static void main(String args[]) {  
        int z1 = 100000;  
        int z2 = 100000;  
        System.out.println(z1 * z2);  
    }  
}
```

Dass in diesem Fall der Compiler keine Warnung gibt oder spätestens zur Run-Time ein Überlauf-Fehler gemeldet wird, ist eine problematische Schwäche in Java. Es liegt also in der Verantwortung des Programmierers, dafür zu sorgen, dass ausreichend Speicherplatz für Zwischenergebnisse und für Ergebnisvariablen zur Verfügung steht.

### Übung zum Programm *Arithmetik08*

Wie muss das Programm geändert werden, damit das Ergebnis korrekt ist? Gibt es einen primitiven Datentyp, dessen Wertebereich groß genug ist? Lösungshinweis: Versuchen Sie es mit *long*.

Natürlich gibt es auch Situationen, wo selbst der Wertebereich des *long*-Datentyps nicht ausreicht. Das soll im nächsten Programm demonstriert werden.

#### Programm *Arithmetik09*: Überlauf bei Integertypen

```
public class Arithmetik09 {  
    public static void main(String args[]) {  
        long z1 = Long.MAX_VALUE;  
        long z2 = Long.MAX_VALUE;  
        System.out.println(z1);  
    }  
}
```

```
        System.out.println(z2);
        System.out.println(z1 * z2);
    }
}
```

Die eingebaute Konstante `Long.MAX_VALUE` enthält den höchsten Wert, den eine Variable vom Typ `long` aufnehmen kann. Wenn beide Höchstwerte multipliziert werden, reicht natürlich der Platz für das Zwischenergebnis vom Typ `long` nicht aus.

### **Lösung: Arbeiten mit Klassen (anstelle von primitiven Datentypen)**

Wenn mit Zahlen dieser Größenordnung gearbeitet werden muss, bleibt nur die eine Möglichkeit: man muss den Klassentyp `BigInteger` verwenden, Instanzen erzeugen und Methoden aufrufen.

#### **Programm Arithmetik10: Class `BigInteger` anstelle von `long`-Typen**

```
import java.math.*;
public class Arithmetik10 {
    public static void main(String args[]) {
        BigInteger z1 = new BigInteger("1234567890123456");
        BigInteger z2 = new BigInteger("9876543210987654");
        System.out.println(z1.multiply(z2));
    }
}
```

### **7.2.3.3 Warum gibt es unverständliche Ergebnisse beim Überlauf?**

Das folgende Programm ist für den Neueinsteiger nicht unbedingt wichtig. Aber es demonstriert, wie die internen Abläufe bei der Arithmetik sind und liefert Erkenntnisse für die Interpretation von "kryptischen" Ergebnissen. Diese Informationen können auch zum Nachschlagen benutzt werden.

Das Programm `Ueberlauf01` liefert als Ergebnis der Addition der beiden Zahlen 2 Millionen und 1.8. Millionen -494967296. Die Ursache dafür ist, dass das Ergebnis zu gross ist für die 31 bit einer Integervariablen. Aber woher kommt diese seltsame Zahl, die zusätzlich auch noch negativ ist?

#### **Programm Ueberlauf01: Ergebnis passt nicht in 31 bits (Integertyp)**

```
public class Ueberlauf01 {
    public static void main(String args[]) {
        int z1 = 2000000000;
        int z2 = 1800000000;
        System.out.println(z1 + z2);
    }
}
```



Damit die internen Arbeitsvorgänge verstanden werden können, muss man zunächst wissen, wie negative Ganzzahlen von Java dargestellt werden. Das höchstwertige Bit repräsentiert das Vorzeichen (0 = positiv, 1 = negativ). Zusätzlich wird ein Verfahren eingesetzt, das sich "**Zweierkomplement**" nennt. Dabei werden negative Zahlen so gespeichert, dass (aus technischen Gründen) zunächst die Bits invertiert werden (d.h. aus 0 wird 1 und aus 1 wird 0), und dann wird noch eine 1 addiert.

**Beispiel 1:** Die **positive** Dezimalzahl 20 wird binär in einer *int*-Variablen wie folgt dargestellt:

```
0000 0000 0000 0000 0000 0000 0001 0100
```

**Beispiel 2:** Um eine **negative** 20 möglichst effizient zu speichern und verarbeiten zu können, passiert folgendes:

```
invers:  1111 1111 1111 1111 1111 1111 1110 1011
+ 1      0000 0000 0000 0000 0000 0000 0000 0001
Ergebnis: 1111 1111 1111 1111 1111 1111 1110 1100
```

Also: Zweierkomplement = Einerkomplement + 1.

Übertragen auf das Beispielpogramm *Ueberlauf01* bedeutet das:

Dezimal:	hexadezimal	rein binär
2.000.000.000	77 35 94 00	0111 0111 0011 0101 1001 0100 0000 0000
+ 1.800.000.000	6b 49 d2 00	0110 1011 0100 1001 1101 0010 0000 0000
= 3.800.000.000	e2 7f 66 00	1110 0010 0111 1111 0110 0110 0000 0000

Das höchstwertige Bit ist 1, damit wird das Ergebnis als negativer Wert interpretiert. Für die Ausgabe wird jetzt die Codierung "rückgängig gemacht", es wird eine 1 subtrahiert und dann die Inversion zurückgenommen:

```
Ergebnis ist:      1110 0010 0111 1111 0110 0110 0000 0000
- 1                0000 0000 0000 0000 0000 0000 0001
=                  1110 0010 0111 1111 0110 0101 1111 1111
Umkehrung der bits: 0001 1101 1000 0000 1001 1010 0000 0000
```

Und diese Bitkombination repräsentiert die Dezimalzahl 494967296.

### 7.2.3.4 Ungenauigkeiten bei Gleitkomma-Zahlen

Ein besonders heikles Thema kann entstehen beim Rechnen mit Gleitkommazahlen. Denn unter bestimmten Umständen kommt es zu Ungenauigkeiten. Der Grund dafür ist der Wechsel des Stellenwertsystems.

#### Wichtige Regeln für das Arbeiten mit Gleitkomma-Zahlen

Nicht alle Dezimalzahlen können im Speicher exakt repräsentiert werden. Weil dort nur eine begrenzte Stellenanzahl zur Verfügung steht, können nur Näherungswerte gespeichert werden (insbesondere bei periodischen Zahlen).

Warum ist das so? Unser Zahlensystem basiert auf dem Stellenwert 10 (Dezimalsystem), Gleitkommawerte im Rechner werden als rein binäre Zahlen codiert (mit dem Stellenwert 2). Das dabei entstehende Problem: nicht jeder Zehnerbruch ist exakt im Dualsystem darstellbar. Zum Beispiel hat die Dezimalzahl 0,1 in der Zweierdarstellung ein unendliches Ergebnis, nämlich 0,0(0011). Verständlich wird dies, wenn man berücksichtigt, dass die Stellenwertigkeit natürlich nicht nur *vor* dem Komma gilt, sondern sich rechts vom Komma fortsetzt. Je weiter rechts eine Ziffer steht, umso kleiner wird ihr Stellenwert - im Dezimalsystem 1/10, dann 1/100, 1/1000 usw., im Dualsystem 1/2, dann 1/4, 1/8 usw. Hier eine Gegenüberstellung für die Stellenwertigkeit der Positionen rechts vom Komma:

<u>Dezimalwert:</u>	<u>das heißt:</u>	<u>entspricht im Dualsystem:</u>
0,5	1/2	0,1
0,250	1/4	0,01
0,125	1/8	0,001
0,0625	1/16	0,0001
0,03125	1/32	0,00001      usw.

Die Dualzahl 0.00011 ist also umgerechnet als Dezimalzahl  $1/32 + 1/16 = 0,09375$ .

Um den Dezimalwert 0,10000 binär darzustellen, benötigen wir eine unendlich lange Dualzahl. Da es aber keine komplette Darstellung einer unendlich langen Zahl gibt, muss der Rechner diese abkürzen (abschneiden), z.B. als 0,0001100110011 speichern. Damit wird dieser Dezimalbruch im Dualsystem ungenau.

### Programm Arithmetik11: Ungenauigkeit bei Wechsel des Stellenwertsystems

```
public class Arithmetik11 {
    public static void main(String args[]) {
        double d1 = 0.17;
        float f1 = 0.000001f;
        System.out.println(d1 / f1);
        System.out.println(d1 * f1);
    }
}
```

### Lösung: Arbeiten mit der Klasse *BigDecimal* anstelle von Gleitkommazahlen

Die Lösung des Problems besteht auch hier im Wechsel des Datentyps. Anstelle der primitiven (und sehr effizienten) Datentypen *double* oder *float* muss der Klassentyp *BigDecimal* genommen werden.

Dieser Datentyp arbeitet weiterhin mit dem Stellenwert 10, das heißt, jede Ziffer einer Zahl wird für sich allein binär verschlüsselt (z.B. in einem Halbbyte), und die einzelnen Halbbytes behalten jeweils die Wertigkeit des Dezimalsystems. Diese Codierung nennt man BCD (binär codierte Dezimalzahlen).

**Beispiel:** Die Dezimalzahl 123 unterschiedlich verschlüsselt:

rein binär:	0000	0000	0111	1011
BCD-Code (pro Ziffer ein Halbbyte):	0000	0001	0010	0011

Der Wechsel vom einfachen Datentyp zum Klassentyp bedeutet auch, dass nicht mit den mathematischen Operatoren für die Grundrechenarten (+ - / \*) gearbeitet werden kann, sondern dass Methoden für die Verarbeitung aufgerufen werden müssen. Außerdem ist es so, dass die meisten Computer spezielle Hardware enthalten für das Rechnen mit Gleitkommazahlen. Das alles führt dazu, dass das Arbeiten mit der Klasse *BigDecimal* aufwändiger ist als das Arbeiten mit den eingebauten Typen. Dafür sind die Ergebnisse aber auch in jedem Fall richtig.

### Programm *BigDecimal01*: Jetzt ist das Ergebnis korrekt

```
import java.math.*;
public class BigDecimal01 {
    public static void main(String args[]) {
        BigDecimal d1 = new BigDecimal("0.17");
        BigDecimal d2 = new BigDecimal("0.000001");
        System.out.println(d1.divide(d2));
        System.out.println(d1.multiply(d2));
    }
}
```

### Übung zum Programm *BigDecimal01*

Überprüfen Sie, ob die Ausgabe des Programms auch mit folgendem Ausgabebefehl möglich ist: `System.out.printf("%s", d1.divide(d2));`

#### 7.2.3.5 Rundung der Ergebnisse

Jetzt bleibt nur noch eine Frage: Und wie ist es mit dem Auf- oder Abrunden der Dezimalstellen? Gerade im Alltag und im kaufmännischen Bereich wird mit einer vereinbarten Genauigkeit der Dezimalstellen hinter dem Komma gearbeitet. Beim Arbeiten mit Währungen hat das Ergebnis normalerweise nur zwei Stellen hinter dem Komma, danach wird auf- oder abgerundet.

Die Klasse *BigDecimal* erlaubt dem Programmierer die volle Kontrolle über die verschiedenen Möglichkeiten des Rundens. Es gibt insgesamt sieben verschiedene Möglichkeiten. Diese sind über Namen aufrufbar, sie sind als eingebaute Konstanten (hinter denen *int*-Werte stehen) vordefiniert, z.B. bedeutet "ROUND\_HALF\_UP" kaufmännisch runden, das heißt, wenn der Nachkommwert größer/gleich 0.5 ist, wird aufgerundet, andernfalls wird abgerundet.

**Programm *BigDecimal02*: Zwei der möglichen Rundensarten**

```
import java.math.BigDecimal;
class BigDecimal02 {
    static public void main(String[] args) {

        BigDecimal d1 = new BigDecimal(0.17);
        BigDecimal d2 = new BigDecimal(0.000001);
        BigDecimal d3 = new BigDecimal(0.0);

        d3 = d1.divide(d2, 2, BigDecimal.ROUND_HALF_UP);
        System.out.printf("%s\n", d3);
        System.out.println(d3);    // Alternative Ausgabe

        // Multiplizieren
        d3 = d1.multiply(d2);
        // Anschließend runden
        BigDecimal d4 = d3.setScale(2, BigDecimal.ROUND_UP);

        // Ergebnis ausgeben
        System.out.printf("%s", d4);
    }
}
```

Das folgende Programm ist ein primitiver Euro-Umrechner, es rechnet einen DM-Wert um in Euro.

**Programm *Euro01*: Rechnen, ohne das Ergebnis zu runden**

```
public class Euro01 {
    public static void main(String args[]) {
        double dm = 100.00;
        double euro = dm / 1.95583;
        System.out.println(euro);
    }
}
```

Das Ergebnis ist: 51.12918811962185

**Übung zum Programm *Euro01***

Bitte ändern Sie das Programm *Euro01.java* so, dass es mit der Klasse *BigDecimal* arbeitet und dass eine kaufmännische Rundung auf zwei Stellen bei der Ausgabe erfolgt.

### Programm Euro02: Lösungsvorschlag

```
import java.math.*;
public class Euro02 {
    public static void main(String args[]) {
        BigDecimal dm    = new BigDecimal("100.00");
        BigDecimal kurs   = new BigDecimal("1.95583");
        BigDecimal euro   = dm.divide(kurs, 2, BigDecimal.ROUND_HALF_UP);
        System.out.printf("%s", euro);
    }
}
```

Das folgende Programm errechnet die Verzinsung eines Sparbetrages von 100.000 Euro aus (für 20 Jahre, bei einem Zinssatz von 6,5 %, ohne Zinseszins).

### Programm Arithmetik12: Zinsrechnung

```
public class Arithmetik12 {
    public static void main(String[] args) {
        double start = 100000.0;
        int jahre = 20;
        float zinssatz = 1.065f;    // 6.5%
        double total;
        total = zinssatz * jahre * start;
        System.out.println("Neues Kapital: " + total);
    }
}
```

### Übung zum Programm BigDecimal03

Bitte ändern Sie das Programm so, dass es mit der Klasse *BigDecimal* arbeitet, und vergleichen Sie die Ergebnisse.

### Programm BigDecimal03: Lösungsvorschlag

```
import java.math.*;
public class BigDecimal03 {
    public static void main(String[] args) {
        MathContext def = MathContext.DECIMAL32;
        BigDecimal start = new BigDecimal(100000);
        BigDecimal jahre = new BigDecimal(20);
        BigDecimal zinssatz = new BigDecimal("1.065"); // 6.5%
        BigDecimal total;
        total = zinssatz.multiply(jahre).multiply(start);
        System.out.println("Neues Kapital: " + total.toString());
    }
}
```

**Zusammenfassung: Finger oder Faust?**

Der Programmierer hat beim Rechnen in Java die Wahl, das sehr effiziente Dualsystem zum Rechnen zu benutzen oder im Dezimalsystem zu bleiben. Wenn die Variablen einen der eingebauten Datentypen *int*, *float* usw. haben, dann wird sehr schnell und sehr einfach gearbeitet. Aber der Programmierer muss wissen, was er tut. Es kann nämlich zu Überlaufproblemen kommen oder auch zu Ungenauigkeiten beim Arbeiten mit Kommastellen.

Wenn dies nicht tolerierbar ist, dann muss der Datentyp *BigDecimal* oder *BigInteger* genommen werden. Diese Klassen arbeiten weiterhin mit dem Dezimalsystem ((BCD-Codierung). Wenn allerdings die Performance der wichtigste Aspekt ist und die Auswirkungen überschaubar sind, kann es sinnvoll sein, mit den primitiven Datentypen zu arbeiten.

Für diese primitiven Datentypen bietet Java Operatoren für die Grundrechenarten an. Wenn darüber hinaus spezielle mathematische oder wissenschaftliche-technische Funktionen benötigt werden (z.B. Logarithmus, trigonometrische Berechnungen...) muss auf die Klassen *Math* (und evtl. auch *BigDecimal*) zurückgegriffen werden.

## 7.3 Vergleichsoperatoren

Mit einem Vergleichsoperator (relationalem Operator) können Vergleiche zwischen zwei Werten durchgeführt werden. Das Ergebnis ist ein boolescher Wert (*true* oder *false*).

**Programm Vergleich01: Zwei Zahlenwerte vergleichen**

```
public class Vergleich01 {  
    public static void main(String[] args) {  
        int z1 = 10;  
        int z2 = 15;  
        System.out.println(z1 < z2);  
    }  
}
```

Das Ergebnis dieses Vergleichs ist natürlich *true*. Andere Vergleichsoperatoren sind:

< größer  
< kleiner  
== gleich  
!= ungleich (nicht gleich).

**Übung zum Programm Vergleich01**

**Übung 1:** Machen Sie eine der beiden Zahlen negativ. Wie lautet dann das Ergebnis?

**Übung 2:** Variieren Sie die Tests und arbeiten Sie mit beliebigen anderen Vergleichsoperatoren.

Das nächste Beispiel vergleicht die Werte von zwei *char*-Variablen auf "gleich" und "ungleich".

### Programm *Vergleich02: char-Variablen vergleichen*

```
public class Vergleich02 {
    public static void main(String[] args) {
        char z1 = 'a';
        char z2 = 'b';
        System.out.println(z1 != z2);
        System.out.println(z1 == z2);
    }
}
```

Der Operator für das Prüfen auf Gleichheit ist das doppelte Gleichheitszeichen `==`. Dies ist eine sehr beliebte Fehlerquelle. Häufig wird dieser Operator verwechselt mit dem einfachen Gleichheitszeichen, das ist aber der Zuweisungsoperator.

Das folgende Beispiel enthält einen Fehler. Die Aufgabe des Programms soll es sein, die beiden Variablen *z1* und *z2* auf Gleichheit zu prüfen.

### Programm *Vergleich03: Falsches Ergebnis*

```
public class Vergleich03 {
    public static void main(String[] args) {
        char z1 = 'a';
        char z2 = 'b';
        System.out.println(z1 = z2);
    }
}
```

### Übung zum Programm *Vergleich03*

Das Programm gibt den Buchstaben *b* aus. Erklären Sie, warum dies so ist und korrigieren Sie das Programm, so dass der Vergleich korrekt ausgeführt wird.

Vorsicht ist geboten beim Vergleich von Gleitkommazahlen. Durch den Wechsel des Stellenwertsystems kann es zu Ungenauigkeiten kommen.

### Programm *Vergleich04: Vergleich von Gleitkommazahlen*

```
public class Vergleich04 {
    public static void main(String[] args) {
        float zahl1 = 0.1f;
        zahl1 = zahl1 / 0.0001f;
        System.out.println(zahl1 == 1000);
        System.out.println(zahl1);
    }
}
```

Das Beispiel *Vergleich04* zeigt, dass der Test auf exakte Gleichheit bei Gleitkommazahlen problematisch ist. Es liefert als Ergebnis *false*.

Eine bessere Lösung besteht darin, das Ergebnis entweder zu runden oder - noch besser - abzufragen, ob die Werte "so ungefähr" gleich sind. Dies kann durch Testen einer (tolerierbaren) Differenz abgefragt werden.

### Vergleich von Objekten

Die Operatoren `==` (für Gleichheit) und `!=` (für Ungleichheit) sind auch auf objektwertige Variablen einsetzbar. Aber: Vorsicht, auch hier muss man wissen, was man tut, wie das folgende Programm zeigt.

#### Programm *Vergleich05*: Vergleich auf Identität

```
public class Vergleich05 {
    public static void main(String[] args) {
        String s1 = new String("Hallo");
        String s2 = new String("Hallo");
        System.out.println(s1 == s2);
    }
}
```

Beide Objekte enthalten die gleichen Daten, trotzdem wird als Ergebnis des Vergleichs *false* ausgegeben. Wie kommt das? Die Antwort: Java arbeitet streng logisch, es werden nämlich die beiden Referenzvariablen *s1* und *s2* verglichen.

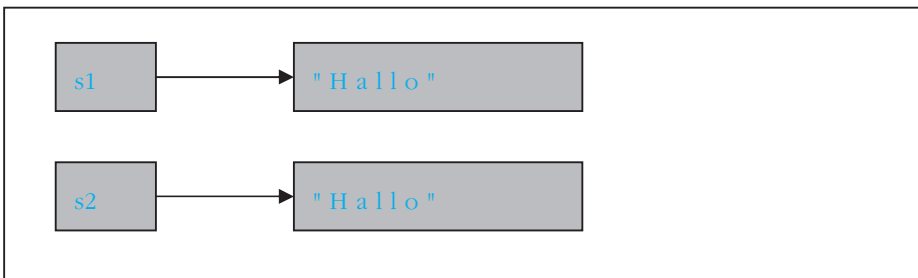


Abb. 7.1. Zwei *String* - Objekte (jeweils mit Referenzvariable und Objektwert)

Und *s1* und *s2* enthalten als Wert die Adressen der beiden Strings. Man sagt, dies ist ein Vergleich auf Identität. Denn so ein Vergleich prüft, ob es sich um dasselbe Objekt handelt. Und die Antwort ist "nein", weil es zwei Objekte im Arbeitsspeicher gibt.

Ein inhaltlicher Vergleich ist nur möglich über den Aufruf einer Methode.



### Programm *Vergleich06*: Vergleich auf inhaltliche Gleichheit

```
public class Vergleich06 {  
    public static void main(String[] args) {  
        String s1 = new String("Hallo");  
        String s2 = new String("Hallo");  
        System.out.println(s1.equals(s2));  
    }  
}
```

Jetzt lautet das Ergebnis *true*. Beide Objekte sind inhaltlich gleich.

**Fazit:** Die Operatoren für Zuweisungen (=) und für den Test auf Gleichheit (==) wirken auf die Werte der Operanden. Das bedeutet: sind die Operanden Referenzvariablen, wirken sie auf den Referenzwert. In den meisten Fällen entspricht das nicht der Aufgabenstellung, deshalb müssen für Referenzen Methoden aufgerufen werden.

## 7.4. Logische Operatoren

### 7.4.1 Was sind logische Operatoren?

Vergleichsoperatoren werden häufig kombiniert mit logischen Operatoren. Dadurch kann man mehrere Vergleiche verknüpfen zu einer Gesamtaussage. Die wichtigsten logischen Operatoren ("conditional operator") sind:

&&    logische UND-Verknüpfung  
||    logische ODER-Verknüpfung, einschließendes OR (= zwei senkrechte Striche)  
!    logische Verneinung (= Ausrufungszeichen)  
^    logische ODER-Verknüpfung, ausschließendes OR

### Programm *Logik01*: Prüfung, ob Wert der Variablen *x* zwischen 0 und 10 liegt

```
class Logik01 {  
    public static void main(String[] args) {  
        int x = 7;  
        System.out.println(x > 0 && x < 10);  
    }  
}
```

Dieser zusammengesetzte Ausdruck hat als Ergebnis *true*, weil beide Einzelbedingungen ( $x > 0$  und  $x < 10$ ) jeweils für sich *true* sind und damit die Verknüpfung durch das logische UND auch *true* ergibt.

### Übung zum Programm *Logik01*

Bitte überlegen Sie, wieviel Testfälle Sie benötigen, um sicherzustellen, dass das Programm auch wirklich fehlerfrei ist (unter "Testfall" ist ein Wert von *x* zu verstehen).

### Lösungshinweise

Es sollte mindestens mit folgenden x-Werten getestet werden:

- Ein "normaler" Wert, der zwischen 1 und 9 liegt, z.B. 5 (= *true*).
- Dann sollten unbedingt die beiden Grenzwerte getestet werden und zwar einmal so, dass das Ergebnis *true* liefert (also z.B. 1 und 9) und auch so, dass das Ergebnis *false* liefert (also z.B. 0 und 10).

### Zusammenfassung

Die logischen Operatoren werden benutzt, um Teilaussagen, z.B. einzelne Vergleiche, zu verknüpfen. Die Operatoren können direkt als boolesche Werte angegeben sein:

`(a && b)`

oder als Vergleiche codiert werden, die boolesche Werte liefern:

`(x>5) && (y<3) .`

Das Ergebnis ist für die gesamte Aussage ein einzelnes *true* oder *false*.

## 7.4.2 Aussagenlogik

Das Binärsystem der heutigen Rechner ist die ideale Grundlage nicht nur für das Arbeiten mit numerischen Werten, sondern auch für das Lösen von logischen Problemen, bei denen mit Aussagen gearbeitet wird, die entweder "wahr" oder "falsch" sind. Die Bereiche der formalen zweiwertigen Logik (z.B. die Boolesche Algebra, die Aussagenlogik und die Prädikatenlogik) sind Wissenschaften, die sich mit den Regeln befassen für die formale Richtigkeit bei der Verknüpfung von Aussagen.

Die Aussagenlogik geht von bereits festgestellten Einzeltatsachen aus. Diese werden Aussagen genannt, und es gilt der fundamentale Grundsatz: Eine Aussage *a* ist wahr oder falsch, etwas Drittes gibt es nicht. Die Aussagen können durch "Junktoren" verknüpft werden, so dass logische "Formeln" entstehen. Bei der Verknüpfung von 2 Aussagen *a* und *b* gibt es theoretisch 16 Möglichkeiten für Regelfestlegungen.

Begründung: Wenn man sich die verknüpfte Aussage als Black-Box vorstellt, die zwei Eingänge (= nämlich die zwei Aussagen) hat und wir abhängig von dem Input dieser zwei Eingänge den Ausgangswert bestimmen (vorhersagen) wollen, dann müssen wir von 4 unterschiedlichen Eingangskombinationen ausgehen - und jede Eingangskombination kann als Verarbeitungsregel der Black-Box dazu führen, dass, je nachdem, was gewünscht wird, der Ausgang mit *wahr* oder *falsch* "gefüttert" wird.

### Beispiel:

Ausgegangen wird von folgenden zwei Aussagen: Klaus raucht(a), Peter raucht (b). Demnach gibt es folgende vier Eingangskombinationen:

a (Klaus raucht)	b (Peter raucht)	Mögliches Ergebnis einer Regel:
true	true	wahr oder falsch
true	false	wahr oder falsch
false	true	wahr oder falsch
false	false	wahr oder falsch

Die Aussagenlogik erstellt nun Regeln für das Ergebnis, das die "Black Box" liefert, abhängig von den Eingangswert a und b. Dabei können die beiden Wahrheitswerte der vier Ergebnisse beliebig miteinander zu einer Regel verknüpft werden. Dadurch ergeben sich  $4 \text{ hoch } 2 = 16$  Kombinationen.

a	b	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	f13	f14	f15	f16
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
AND								XOR OR									

Abb. 7.2: Die 16 zweistelligen Funktionen

Beispiele für Anwendungen dieser Regeln:

- Nur wenn a und b denselben Wahrheitswert haben, so ist der Ausgang *true* (Äquivalenz, Regel f10).
- Nur wenn a und b unterschiedliche Werte haben, ist der Ausgang *true* (Antivalenz, Regel f7).

Die in Programmiersprachen gebräuchlichsten Verknüpfungsregeln sind

- die **UND**-Verbindung (Konjunktion, das heißt, das Gesamtergebnis ist nur dann wahr, wenn sowohl a als auch b wahr sind) und
- die **ODER**-Verbindung (Disjunktion, das heißt, das Gesamtergebnis ist nur dann falsch, wenn sowohl a als auch b falsch sind). Diese ODER-Verbindung wird auch **einschließendes ODER** genannt. Darüber hinaus gibt es in Java noch
- die **ausschließende ODER**-Verbindung (exklusives ODER, XOR bzw. entweder...oder), die nur dann wahr ist, wenn eine der beiden Aussagen, aber nicht beide gleichzeitig, wahr ist.

## Boolesche Algebra

Mit Hilfe der Booleschen Algebra werden die Gesetzmäßigkeiten der Aussagenlogik in ein mathematisches System gebracht. Sie wurde von dem englischen Mathemati-

ker George Boole (1815 - 1864) entwickelt. Sie besteht aus einer Menge von zwei Elementen (Aussagen) und nur drei möglichen Junktoren (Operatoren):

- Addition (UND-Funktion)
- Multiplikation (ODER-Funktion)
- Komplementbildung (NICHT-Funktion, Negation).

### Wo sind Anwendungen für die Boolesche Algebra?

Die grundlegenden Schaltungen in den Computern folgen den Gesetzen der Booleschen Algebra. Auch für das Formulieren von Abfragen für Internet-Suchmaschinen und relationalen Datenbanken werden Boolesche Operatoren eingesetzt. Und für das Codieren von logischen Ausdrücken in Java ist das Verständnis der Booleschen Algebra ebenfalls eine Voraussetzung.

**Binäre Verknüpfungen** machen aus zwei Aussagen eine neue Aussage. In Java gibt es die folgenden logischen Operatoren für binäre Verknüpfungen:

- && (das kaufmännische Und-Zeichen, Ampersand) für die UND-Verbindung
- || (zwei senkrechte Striche) für das einschließende ODER
- ^ für das ausschließende ODER.

Die Tabelle Abb. 7.3. enthält einen Überblick über die Wahrheitwerte der Booleschen Algebra, ergänzt um die exklusiv-ODER-Verbindung (XOR). Dabei steht w für wahr/true und f für falsch/false.

Aussage a	Aussage b	UND- Verbindung &&	incl. ODER- Verbindung 	excl. ODER (XOR) ^
w	w	w	w	f
w	f	f	w	w
f	w	f	w	w
f	f	f	f	f

Abb. 7.3: Tafel der Wahrheitwerte für binäre Verknüpfungen

Außerdem kennt Java einen Operator (die Negation) für eine **unäre Verknüpfung**. Eine unäre Verknüpfung macht aus einer Aussage eine neue Aussage.

- ! (Ausrufungszeichen) für die Negation, d.h. für die Umkehrung des Wahrheitswertes.

### Übung für binäre Verknüpfungen

Das Programm *Logik02* soll die folgende Aufgabe lösen: Für alle Werte, die zwischen 0-10 und 80-90 liegen, soll das Programm *true* liefern.

#### Lösungsvorschlag: Programm *Logik02* - Wertebereiche abfragen

```
class Logik02 {
    public static void main(String[] args) {
        int x = 85;
        boolean richtigeZahl = ((x>0 && x<10) || (x>80 && x<90));
        System.out.println(richtigeZahl);
    }
}
```

Die Gesamtaussage des Ausdrucks ist wahr, wenn der Wert der Variablen *x* zwischen 0 und 10 **oder** zwischen 80 und 90 liegt. Hier zeigt sich, dass umgangssprachlich die Begriffe *und* bzw. *oder* manchmal etwas unscharf benutzt werden. In der strengen Logik der Booleschen Algebra muss es in diesem Fall die Oder-Verbindung sein, obwohl laut Aufgabenstellung alle Werte gesucht werden, die "zwischen 0 und 10 **und** zwischen 80 und 90 liegen".

### Übung zum Programm *Logik02*

Kann statt des Operators für das einschließende Oder auch der Operator für das ausschließende Oder eingesetzt werden? Bitte ändern Sie das Programm und testen Sie das Ergebnis.

### Übung für unäre Verknüpfung (Negation)

Codieren Sie ein Programm, das folgende Aufgabe löst: Es soll geprüft werden, ob ein Wert *x* **nicht** zwischen 0-10 oder nicht zwischen 80-90 liegt.

#### Lösungsvorschlag: Programm *Logik03* - Wertebereich ausschließen

```
class Logik03 {
    public static void main(String[] args) {
        int x = 85;
        boolean richtigeZahl = !((x>0 && x<10) || (x>80 && x<90));
        System.out.println(richtigeZahl);
    }
}
```

Die einzige Änderung, die notwendig ist im Vergleich zum Programm *Logik02*, ist die Umkehrung des Wahrheitswertes der Gesamtaussage - und dies erfolgt durch Codierung des NOT-Operators ! (Ausrufungszeichen, nicht zu verwechseln mit dem einfachen senkrechten Strich).

Die Negation in komplexen Bedingungen ist manchmal schwer verständlich. Ein Beispiel ist folgende Regel: "Wenn auf einem Flugticket nicht ein A oder ein B steht und wenn das Datum nicht 11 oder 12 ist, dann ist der Flug nicht gecancelt."

Wie bitte? Wann findet der Flug statt?

#### Programm Ausdruck03: Negierte Teilaussagen verknüpfen

```
class Ausdruck03 {
    public static void main(String[] args) {
        char ticket = 'C';
        int datum = 13;
        if (!(ticket == 'A') || !(ticket == 'B') &&
            (datum == 11 || datum == 12))
            System.out.println("Kein Flug");
        else
            System.out.println("Flug");
    }
}
```

Eine **Umformulierung** in eine positive Aussage macht das Programm verständlicher:

"Der Flug findet statt, wenn auf dem Ticket A oder B und als Datum 11 oder 12 steht."

Ein gültiges Ticket kann also eine der folgende vier Kombinationen enthalten: A11, A12, B11 oder B12. Alle anderen sind ungültig.

#### Programm Ausdruck04: Positive Teilaussagen verknüpfen

```
class Ausdruck04 {
    public static void main(String[] args) {
        char ticket = 'B';
        int datum = 12;
        if ((ticket == 'A' || ticket == 'B') &&
            (datum == 11 || datum == 12))
            System.out.println("Flug");
        else
            System.out.println("Kein Flug");
    }
}
```

Die Regel für die Umformung nennt man die "**de Morgansche-Regel**":

aus	((not (a or b)) wird	((not a) and (not b)) bzw.
aus	((not (a and b)) wird	((not a) or (not b)).

### Logikaufgabe

Zur Vertiefung des Themas "Arbeiten mit logischen Operatoren" werden Sie jetzt ein komplettes Beispiel lösen. Die **Aufgabenstellung** ist wie folgt:

"Julia plant ihre Party. Diese soll aber nur stattfinden, wenn folgende Bedingungen erfüllt sind:

1. Wenn Andreas (A) kommt, dann muss auch Bernd(B) kommen.
2. Entweder kommen Andreas(A) und Christian(C) oder es kommt keiner von beiden.
3. Entweder kommt Bernd oder Christian."

Die Party findet nur statt, wenn alle drei Bedingungen erfüllt sind. Das heißt, die drei Teil-Aussagen müssen wahr sein, damit die Gesamtaussage wahr ist. Bitte formulieren Sie den Ausdruck und codieren Sie anschließend ein Java-Programm, das diese Aufgabe löst.

Lösungshinweis: Es soll lediglich mit den logischen Operatoren UND, ODER und NICHT gearbeitet werden.

### Lösungsvorschlag für die formale Beschreibung der Bedingungen:

1. Teilaussage: NICHT-A ODER (A UND B)
2. Teilaussage: (A UND C) ODER (NICHT-A UND NICHT-C)
3. Teilaussage: (B UND NICHT-C) ODER (NICHT-B UND C)

### Programm *Logik04*: Lösung, codiert in Java

```
class Logik04 {  
    public static void main(String[] args) {  
        boolean andreas    = true;    // Andreas kommt  
        boolean bernd       = false;   // Bernd kommt nicht  
        boolean christian    = true;    // Christian kommt  
        boolean party = ((!andreas || andreas && bernd) &&  
            ((andreas && christian) || (!andreas && !christian)) &&  
            (bernd && !christian) || (!bernd && christian));  
        System.out.println(party);  
    }  
}
```

Das Ergebnis ist *false* (d.h. die Party findet nicht statt), denn bereits die erste Teilaussage ist nicht erfüllt. Wichtig für die Formulierung der Lösung in Java ist, dass die Priorität der Operatoren bekannt ist: NOT als unärer Operator besitzt die höchste Priorität, danach gilt die "UVO-Regel": **Und** **vor** **Oder**.

Die dringende Empfehlung ist, immer Klammern zu setzen. Damit werden eventuelle Unsicherheiten ausgeschlossen und die Lesbarkeit von Aussagen, besonders in dieser Komplexität, eindeutig verbessert.

### Übung zum Programm Logik04

Bitte ändern Sie die Boolean-Werte so, dass die Party stattfinden kann. (Lösungshinweis: Julia möchte mit Bernd allein sein, d.h. die beiden anderen müssen *false* bekommen).

### Kurzschluss-Auswertung ("Short-Circuit-Evaluation")

Die logischen Operatoren `&&` (AND) und `||` (OR) werden auch als Short-Circuit-Evaluation-Operatoren bezeichnet, denn der Ausdruck wird nur solange ausgewertet, bis das Ergebnis eindeutig feststeht. Die Berechnung wird sofort beendet, wenn klar ist, welcher logische Wert sich ergibt. So kann es passieren, dass ein rechts stehender Teilausdruck nicht mehr ausgewertet wird, weil er für das Gesamtergebnis keine Bedeutung mehr hat. Beispiel: Wenn bei `&&`-Verknüpfung *ein* Teilausdruck *false* liefert, **muss** der Gesamtausdruck auch *false* werden, egal, was noch an weiteren Teilausdrücken folgt.

Im Gegensatz zu dieser Arbeitsweise gibt es in Java noch die folgenden logischen Operatoren, die immer und in jedem Fall den gesamten Ausdruck auswerten:

- `&` (= das einfache und nicht doppelte Ampersand), für die UND-Verknüpfung,
- `|` (= der einfache und nicht doppelte Strich), für die ODER-Verbindung,

Diese beiden Operatoren werden auch als non-SCE ("nicht-Short-Circuit-Evaluation") Operatoren bezeichnet, weil die Evaluierung des Ausdrucks nicht abgebrochen wird, auch wenn die weitere Auswertung keine Rolle mehr für das Gesamtergebnis hat. Diese Art der Evaluierung ist natürlich aufwändiger als die "normalen" logischen Ausdrücke. Warum kann es trotzdem sinnvoll sein, mit diesen Operatoren zu arbeiten?

### Programm Sce01: Short-Cut-Evaluation (SCE) mit UND-Operator

```
public class Sce01 {  
    public static void main(String args[]) {  
        boolean b1 = false;  
        long z2 = 11;  
        System.out.println(b1 && z2++ > 5);  
        System.out.println(z2);  
    }  
}
```

Als Ergebnis wird ausgegeben, dass der Gesamtausdruck *false* ist und dass der Inhalt der Variablen `z2` unverändert bei 11 geblieben ist.

### Übung zum Programm Sce01

Ändern Sie den logischen UND-Operator ab, so dass der Ausdruck auf jeden Fall komplett ausgewertet wird. Jetzt muss der Wert der Variablen `z2` sich verändert haben.



Das nächste Beispiel zeigt eine ähnliche Aufgabenstellung. Durch den logischen Operator `||` wird die Auswertung abgebrochen, wenn das Ergebnis zweifelsfrei feststeht.

### Programm *Sce02*: Short-Cut-Evaluation (SCE) mit ODER-Operator

```
public class Sce02 {
    public static void main(String args[]) throws Exception {
        int a = 0;
        if (a==0 || (a = System.in.read()) > 0)
            System.out.println(a);
    }
}
```

Die *read*-Methode wird nicht ausgeführt, weil bereits der erste Teilausdruck erfüllt ist (bei ODER genügt das!).

### Übung zum Programm *Sce02*

Ändern Sie das Programm so ab, dass in jedem Fall auch der Einlesevorgang stattfindet.

## 7.5 Bitweise Operatoren

Es gibt in Java Operatoren, die auf Bitebene arbeiten, obwohl normalerweise das Byte (8 bits) die kleinste adressierbare Einheit ist. Mit Hilfe dieser bitweisen Operatoren kann man die einzelnen Bits verschieben ("shiften") oder logisch verknüpfen.

### 7.5.1 Verschieben von Bits innerhalb eines Operanden

Für das Verschieben von Bits innerhalb einer Variablen oder innerhalb eines Literals gibt es folgende Operatoren:

`<<` verschieben nach links  
`>>` verschieben nach rechts

Die Operanden müssen einen ganzzahligen Datentyp haben, also *int*, *short*, *long*, *byte* oder *char*.

### Programm *BitShift01*: Einzelne Bits nach links oder nach rechts verschieben

```
class BitShift01 {
    public static void main(String[] args) {
        char zeichen1 = 'A';
        int zeichen2, zeichen3;
        zeichen2 = zeichen1 << 1;    // nach links
        zeichen3 = zeichen1 >> 1;    // nach rechts
    }
}
```

```
        System.out.println(zeichen2);  
        System.out.println(zeichen3);  
    }  
}
```

Für die Interpretation der Ergebnisse (130 und 32) wird die Unicode-Tabelle benötigt.

Der Buchstabe A wird wie folgt codiert:	0100 0001	hex. 41
Nach dem Shiften nach links um 1 Bit:	1000 0010	als <i>int</i> -Wert (rein binär): 130
Nach dem Shiften nach rechts um 1 Bit:	0010 0000	als <i>int</i> -Wert (rein binär): 32

### 7.5.2 Binäre Verknüpfung eines Operanden

Für die binäre Verknüpfung gibt es folgende Operatoren:

- & logische UND-Verknüpfung
- | logische ODER-Verknüpfung
- ^ logische XOR-Verknüpfung
- ~ logisches NOT (Einerkomplement, alle bits werden invertiert)

Das Ergebnis einer solchen bitweisen Verknüpfung ist ein *int*-Wert (= rein binär).

#### Programm *BitVerknuepf01*: Einzelne Bits logisch verknüpfen

```
class BitVerknuepf01 {  
    public static void main(String[] args) {  
        char zeichen1 = 'A';  
        char zeichen2 = 'B';  
        int erg;  
  
        erg = zeichen1 & zeichen2;           // UND  
        System.out.print(erg + " = ");  
        System.out.println((char)erg);  
  
        erg = zeichen1 | zeichen2;           // ODER  
        System.out.print(erg + " = ");  
        System.out.println((char)erg);  
  
        erg = zeichen1 ^ zeichen2;           // XOR  
        System.out.print(erg + " = ");  
        System.out.println((char)erg);  
    }  
}
```

Die Ausgabe des Programms ist:

```
64 = @
67 = C
3  = ♥
```

Für die Interpretation der Ausgabe benötigen wir die Unicode-(ASCII-)Tabelle. Dort finden sich die Zeichen für die binären Inhalte der Variablen *erg*. Das Ergebnis wurde vom Programm wie folgt ermittelt:

Der Buchstabe A wird wie folgt codiert:	0100 0001	hex. 41
Der Buchstabe B wird wie folgt codiert:	0100 0010	hex. 42
Die AND-Verbindung (&) der Bits ergibt:	0100 0000	als Integer: 64
Die OR-Verbindung ( ) der Bits ergibt:	0100 0011	als Integer: 67
die XOR-Verbindung (^) der Bits ergibt:	0000 0011	als Integer: 3.

Eine interessante, aber praktisch wohl wenig eingesetzte Aufgabenstellung ist das Bilden des bitweisen **Komplements** eines ganzzahligen Wertes, z.B. durch `~'A'`:

Der Buchstabe A wird wie folgt codiert:	0000 0000	0100 0001	hex 0041
Bitweises Komplement bilden:	1111 1111	1011 1110	
Wird als negativ interpretiert, d.h. - 1	0000 0000	0000 0001	
Ergebnis	1111 1111	1011 1101	
Weil negativ, Komplement bilden	0000 0000	0100 0010	hex.0042 = 'B' .

### Programm BitVerknuepf02: Not-Operator für Komplement-Bildung

```
class Bitverknuepf02 {
    public static void main(String[] args) {
        byte b = 3;
        System.out.println(~b);
    }
}
```

Die Ausgabe des Programms ist -4, denn

0000 0011	entspricht 3
1111 1100	Komplement, entspricht -4 (Erläuterung siehe Kap. 7.2.3.3)

### Übung zum Programm Bitverknuepf02

**Übung 1:** Notieren Sie für sich auf einem Blatt Papier, wie dieses Ergebnis zu Stande gekommen ist. (Lösungshinweis: Der Wert wird als negative Zahl interpretiert, danach das Komplement gebildet und darauf eine 1 addiert.)

**Übung 2:** Warum werden die Operatoren & bzw. | in bestimmten Anweisungen als logische Operatoren und in anderen Anweisungen als bitweise Verknüpfungsoperatoren behandelt?

Lösungshinweis: dies hängt ab von dem Datentyp der Operanden.

## 7.6 Auswertungs-Reihenfolge (Präzedenzregeln)

### 7.6.1 Vorrang (Priorität)

Der Vorrang regelt die Reihenfolge, in der die verschiedenen Operatoren ausgewertet werden. Enthält ein Ausdruck *unterschiedliche* Operatoren, so gelten auch unterschiedliche Prioritäten bei der Auswertung. **Unäre** Operatoren werden zuerst ausgewertet.

Für die **binären** Operatoren gilt:

- Arithmetische Operatoren werden zuerst ausgewertet. Dabei geht Punkt-Arithmetik vor Strich-Arithmetik.
- Danach werden relationale Operatoren (Vergleiche) durchgeführt.
- Und dann werden evtl. vorhandene logische Verknüpfungen der Ergebnisse dieser Vergleiche ausgewertet.
- Es gibt nur ein Ergebnis - und das wird ganz zum Schluss einer Variablen zugewiesen, falls es einen Zuweisungsoperator gibt.

#### Programm Prioritaet01: Beispiel 1 für unterschiedliche Prioritäten

```
class Prioritaet01 {
    public static void main(String[] args) {
        int x = 5;
        int y = 9;
        System.out.println(x + 15 / 5);
        System.out.println(x != 6 && y < 8);
        System.out.println(x == 5 ^ y > 0);
        System.out.println(x < 3 || y == 5);
    }
}
```

#### Programm Prioritaet02: Beispiel 2 für unterschiedliche Prioritäten

```
class Prioritaet02 {
    public static void main(String[] args) {
        System.out.println(2 + 15 / 3 < 7 && 5 < 3);
    }
}
```

Der Ausdruck im Programm *Prioritaet02.java* besteht aus zwei Teilausdrücken, diese sind mit dem logischen UND verknüpft. Das Ergebnis ist *false*, weil der erste Teilausdruck  $(3 + 15 / 2) < 7$  falsch ist.

Durch den Einsatz von Klammern können diese Regeln umgangen werden. Es wird aber dringend empfohlen, die Klammern auch zu verwenden, wenn die normale Auswertungsreihenfolge unverändert gelten soll (damit Lesbarkeit verbessert wird).

## Übung zum Programm Prioritaet02

Wie kann allein durch das Einfügen von Klammern ( ) dieser Ausdruck *true* liefern?  
Zur Erinnerung: Wenn durch Klammern nichts anderes erzwungen wird, hat die Multiplikation eine höhere Priorität als die Addition.

### 7.6.2 Assoziativität

Dadurch ist festgelegt, in welcher Reihenfolge ein Ausdruck ausgewertet wird, der mehrere Operatoren **mit gleicher Priorität** enthält. Die normale Auswertungsreihenfolge ist von links nach rechts. Davon gibt es zwei Ausnahmen: Unäre Operationen z.B. in der Form `++zahl` und Zuweisungen in Form von: `x = y = z = 0;`.

#### Programm Assoziativitaet01: Ausnahmsweise rechts beginnen

```
class Assoziativitaet01 {
    public static void main(String[] args) {
        int x,y,z;
        x = y = z = 5;
        System.out.println(x);
        y = y + ++z;
        System.out.println(y);
    }
}
```

### Zusammenfassung

Ein Ausdruck ("expression") ist eine Sprachkonstruktion in Java, die einer mathematischen Formel ähnelt. Er wird ausgewertet nach exakt festgelegten Regeln und kann genau **einen** Wert als Ergebnis liefern. Ein Ausdruck besteht aus einer Kombination von Operanden, Operatoren und Methodenaufrufen.

Sie haben in diesem Kapitel die wichtigsten Operatoren kennengelernt. Operatoren arbeiten typischerweise mit primitiven Datentypen.

Objekttypen dagegen werden mit Methoden bearbeitet, und dieses Thema wird detailliert in den Kapiteln 10 und 11 behandelt. Es gibt (in Ausnahmefällen) aber auch die Möglichkeit, Operatoren für Referenztypen einzusetzen, z.B.

- String-Konkatenierung ist sowohl durch eine Methode (*concat*) als auch mit einem Operator (+) möglich
- Der Zuweisungsoperator = und auch die Vergleichsoperatoren wie == oder != sind auch für Referenzvariablen einsetzbar. Aber in den meisten Fällen führt dies nicht zu dem gewünschten Ergebnis. Besser ist es in jedem Fall, bei Vergleichen mit der Methode *equals()* zu arbeiten und die Zuweisung mit der *clone()*-Methode durchzuführen.

---

# 8

## Anweisungen kodieren ("statements")

---

Eine Anweisung (Befehl, "statement") beschreibt eine komplette Ausführungsoperation für die JVM. Äußerliches Merkmal für eine komplette Anweisung ist, dass sie durch ein Semikolon abgeschlossen wird.

In den Kapiteln 4 und 5 haben wir bereits eine wichtige Anweisung ausführlich behandelt: das Deklarationsstatement. Dort haben Sie gesehen, wie im Arbeitsspeicher Platz für eine Variable deklariert wird, was ein Bezeichner ist und welche Bedeutung die Datentypen haben.

Weitere wichtige Anweisungen sind

- Methodenaufrufe
- Zuweisungsanweisung und
- Steueranweisungen.

Methodenaufrufe wurden schon im Kapitel 6 (für die Ein- und Ausgabe) benutzt und werden ausführlich ab Kapitel 10 behandelt.

In diesem Kapitel werden Sie die Zuweisungen und Steueranweisungen kennenlernen. Dabei wird detailliert erläutert,

- was Zuweisungen sind und was dabei zu beachten ist,
- welche unterschiedlichen Arten von **Steueranweisungen** es gibt und wie sie eingesetzt werden,
- dass es für ein und dieselbe Aufgabenstellung verschiedene Lösungsmuster ("pattern") gibt. Anhand einer Schleife, die Datensammlungen sequentiell bearbeitet, wird beispielhaft gezeigt, wo die Vor- und Nachteile der unterschiedlichen Lösungen liegen.
- welche Konventionen eingehalten werden sollten, damit gut lesbare und übersichtliche Quelltexte entstehen.

Nur kurz erwähnt werden Leeraanweisungen, die aus nur einem Semikolon bestehen und in manchen Situationen unangenehme Fehler bewirken können.

Der Schwerpunkt dieses Kapitels liegt in der Beschreibung der unterschiedlichen Steueranweisungen. Diese Statements steuern den Ablauf des Programms, und Sie werden lernen, wie Programmteile **alternativ** (mit der *if*- bzw. *switch*-Anweisung) oder **mehrfach** (mit der *while*- bzw. *do*- und *for*-Schleife) ausgeführt werden können.

## 8.1 Einfache und zusammengesetzte Anweisungen

Java unterscheidet zwischen einfachen und zusammengesetzten Anweisungen. Zusammengesetzte Anweisungen sind Anweisungen, die andere Anweisungen ("substatements") enthalten. Die Liste der Substatements wird dabei eingeschlossen in geschweifte Klammern { }. Es gibt zwei typische Formen der zusammengesetzten Anweisungen ("compound statements", Verbundanweisungen): den Anweisungsblock und die Steueranweisungen.

### 8.1.1 Anweisungsblock

Ein Block ist eine Folge von einfachen Statements, die von geschweiften Klammern eingefasst sind. Beispiel für eine einfache Anweisung ("simple Statement"):

```
x = 42;
```

Beispiel für einen Block als zusammengesetzte Anweisung (Verbundanweisung oder "compound Statement"):

```
{  
    x = 42;  
    y = 3;  
    z = x + y;  
}
```

Wo liegt der Sinn solcher Klammerung?

- Ganz allgemein kann die Blockbildung der Strukturierung dienen. In Verbindung mit dem Einrücken innerhalb der Zeilen wird eine übersichtliche Gliederung erreicht.
- Außerdem bedeutet diese Klammerung, dass die Anweisungen im "compound statement" behandelt werden wie *eine* einzelne Anweisung. Sie können überall dort stehen, wo die Syntax eine einzelne Anweisung verlangt.
- Darüber hinaus ist es möglich, dass innerhalb dieses Blocks Variablen deklariert werden, die dann "lokale Variablen" genannt werden, weil sie nur innerhalb dieses Blocks angesprochen (gelesen oder verändert) werden können und weil sie nur existieren für die Zeit der Ausführung dieses "compound-statements". Das folgende Programm demonstriert das Arbeiten mit lokalen Variablen, allerdings wird bei dem Versuch, dieses Programm umzuwandeln, eine Fehlermeldung ausgegeben.

#### Programm Lokal01: Arbeiten mit lokalen Variablen (Fehlerhaft)

```
public class Lokal01 {  
    public static void main(String[] a) {  
        int zahl1 = 10;  
    }  
}
```

```
{
    int zahl2 = 5;
    System.out.println(zahl2++);
}
System.out.println(zahl1 + zahl2);
}
```

### Übungen mit Programm Lokal01

**Übung 1:** Bitte korrigieren Sie den Fehler (Hinweise zur Ursache siehe vorherigen Absatz: Blockbildung und lokale Variablen).

**Übung 2:** Klären Sie durch Programmänderung die Frage, ob der Variablenname *zahl1* auch innerhalb des Blocks, z. B. in der *println*-Methode, benutzt werden darf.

Die Erkenntnis aus diesen Übungen ist, dass ein Anweisungsblock den Gültigkeitsbereich von Variablen definiert. Ist eine Variable in einem Block definiert, so ist sie nur dort gültig. Sie kann außerhalb dieses Blocks nicht angesprochen werden. Man nennt diese Variable "lokale Variable".

Blocks können beliebig geschachtelt werden. Allerdings müssen innerhalb einer Methode die Bezeichner der lokalen Variablen eindeutig sein, d.h. ein einmal deklarierter Bezeichner darf nicht noch einmal deklariert werden, auch nicht in einem Unterblock.

## 8.1.2 Steueranweisungen

Mit Hilfe von Steueranweisungen kann der Programmierer den Ablauf des Programms steuern. Es gibt Steueranweisungen für Verzweigungen (*if* und *switch*) und für die Schleifenbildung (*for*, *while*, *do*). Sie sind ebenfalls zusammengesetzte Anweisungen, denn sie enthalten andere Anweisungen, die dann entweder selektiv (gar nicht, weil die Bedingung nicht erfüllt ist) oder einmal bzw. mehrmals ausgeführt werden (wenn die dafür notwendige Bedingung erfüllt ist).

Allen Steueranweisungen gemeinsam ist, dass der weitere Ablauf des Programms von einem Ausdruck gesteuert wird, der vom Typ *boolean* sein muss.

### Programm Anweisung01: Boolescher Ausdruck im if-Befehl

```
class Anweisung01 {
    public static void main(String[] args) {
        if (5 > 3)
            System.out.println("5 ist groesser als 3");
    }
}
```



Der IF-Befehl beginnt mit dem Schlüsselwort *if*. Dann folgt in runden Klammern ein Ausdruck, der als Ergebnis einen booleschen Wert liefert (ja oder nein). Im Beispiel enthält die *if*-Anweisung danach eine weitere Anweisung, nämlich den Aufruf einer *println*-Methode. Dieser Block wird nur ausgeführt, wenn der boolesche Ausdruck *true* liefert. Auf jeden Fall ist der gesamte IF-Befehl erst nach dem Semikolon beendet. Das Semikolon schließt hier das Statement mit dem Methodenaufruf ab. Zusammengesetzte Anweisungen, in diesem Fall also das *if*-Statement selbst, werden *nicht* mit einem Semikolon abgeschlossen.

### Programm *Anweisung02*: Anweisungsblock anstelle der Einzelanweisung

```
class Anweisung02 {
    public static void main(String[] args) {
        if (5 > 3) {
            System.out.println("Das Ergebnis der Prüfung ist: ");
            System.out.println("Fuenf ist groesser als 3");
        }
    }
}
```

In dem Programm *Anweisung02* besteht der Ja-Zweig der *if*-Anweisung aus mehr als einer Anweisung. Deswegen muss ein Anweisungsblock gebildet werden, der durch die geschweiften Klammern zusammengehalten wird. Der gesamte IF-Befehl ist nach diesem Block beendet. Es wird am Ende kein Semikolon gesetzt.

## 8.2 Wertezuweisung

Durch eine Wertezuweisung ("assignment") wird der Inhalt einer Variablen verändert, sie bekommt einen neuen Wert zugewiesen. Der Operator für die Zuweisung ist das Gleichheitszeichen =.

### Programm *Zuweisung01*: Variablen einen neuen Wert zuweisen

```
class Zuweisung01 {
    public static void main(String[] args) {
        int zahl = 15;           // Initialisierung
        System.out.println("Vorher: " + zahl);
        zahl = 1;                // Wertezuweisung
        System.out.println("Nachher: " + zahl);
    }
}
```

Rechts vom Gleichheitszeichen steht der "Sender", links vom Gleichheitszeichen steht der "Empfänger". Der Sender ist ein Ausdruck, also eine Variable, Konstante oder ein Methodenaufruf oder eine beliebige Kombination daraus. Der Empfänger kann (nur) aus einer Variablen bestehen, d.h. vor dem Gleichheitszeichen steht der

Identifizier der Variablen, die den neuen Wert des Ausdrucks bekommt. Der bisherige Wert der Variablen wird *komplett* ersetzt durch den evaluierten Wert des Ausdrucks. Zur Wiederholung noch einmal der Hinweis: ein Ausdruck berechnet *einen* (und nur einen) Wert.

### Programm *Zuweisung02*: Ein Ausdruckswert wird evaluiert und zugewiesen

```
class Zuweisung02 {
    public static void main(String[] args) {
        int zahl = 15;
        System.out.println("Vorher: " + zahl);
        zahl = (1 + 6) / 2;
        System.out.println("Nachher: " + zahl);
    }
}
```

Jeder Ausdruck hat also nur *einen* Wert. Im Beispiel *Zuweisung02.java* ist dieser zweifellos vom Datentyp *int*, denn alle Operanden dieses Statements haben denselben Typ und es gibt deswegen auch keinerlei Anpassungsprobleme. Damit ist auch gleichzeitig der Datentyp des Ausdrucks festgelegt. Dieser Datentyp des Sender-Ausdrucks muss übereinstimmen mit dem Datentyp der Empfängervariablen.

Was passiert aber, wenn

- der Datentyp des Sender-Ausdrucks nicht übereinstimmt mit dem Datentyp der empfangenden Variablen oder
- die Operanden des Ausdrucks unterschiedliche Datentypen haben?

### Unterschiedliche Datentypen bei Sender und Empfänger

Generell kann der Empfänger nur Daten speichern, die mit seinem Typ übereinstimmen. Deswegen wird der Datentyp des Ausdrucks an den Typ des Empfängers angepasst, allerdings nur, wenn dadurch nicht die Gefahr besteht, dass Informationen verloren gehen ("implizite Typumwandlung"). Andernfalls gibt es Fehler bei der Umwandlung.

### Programm *Zuweisung03*: Umwandlungsfehler wegen falscher Zuweisung

```
public class Zuweisung03 {
    public static void main(String args[]) {
        int zahl1 = 10;
        short zahl2 = 5;
        zahl2 = zahl1;
        zahl1 = zahl2;
    }
}
```

Das Programm *Zuweisung03.java* kann nicht fehlerfrei umgewandelt werden, weil die empfangende Variable *zahl2* zu klein ist für einen *int*-Wert.

### Übung zum Programm *Zuweisung03*

Korrigieren Sie den Fehler, indem Sie den Datentyp der empfangenden Variablen ändern.

Ein nicht ganz so offensichtlicher Fehler steckt im Programm *Zuweisung04*.

### Programm *Zuweisung04*: Umwandlungsfehler wegen falscher Zuweisung

```
public class Zuweisung04 {  
    public static void main(String args[]) {  
        float s1 = 5.0;  
    }  
}
```

### Übung zum Programm *Zuweisung 04*

Korrigieren Sie den Fehler im Programm *Zuweisung04.java*. Hinweise zur Lösung finden Sie im Kapitel 5 beim Thema Floatingpoint-Literale. Dort ist beschrieben, wie ein Literal vom Typ *float* codiert wird.

### Unterschiedliche Datentypen innerhalb eines Ausdrucks

Typanpassungen sind auch notwendig, wenn die Operanden innerhalb eines Ausdrucks unterschiedliche Typen haben (siehe hierzu auch Kapitel 7).

### Beispiel *Zuweisung05*: Typanpassung der Operanden im Ausdruck

```
public class Zuweisung05 {  
    public static void main(String args[]) {  
        String str = "Bahnhofstr.";  
        int nr = 125;  
        System.out.println(str + nr);  
    }  
}
```

Beim Aufruf der Methode *println()* werden zwei Argumente übergeben, ein *String*-Argument und ein *int*-Argument. Die Werte müssen "gleichnamig" gemacht werden (denselben Typ bekommen) und dies geschieht automatisch ohne explizite Angabe durch den Programmierer, weil kein Informationsverlust dadurch entsteht.

### 8.2.2. Zusammengesetzte Zuweisungsoperatoren

Neben dem bisher besprochenen Gleichheitszeichen `=` als Operator für die Wertezuweisung gibt es in Java zusammengesetzte Zuweisungsoperatoren. Sie werden in folgender Form verwendet:

```
empfänger    operator=    sendeausdruck;
```

Das Gleichheitszeichen wird also kombiniert mit einem weiteren Operator z.B. einem Rechen- oder einem Vergleichsoperator. Zwischen Operator und dem Gleichheitszeichen darf kein Leerzeichen stehen. Beispiel:

```
zahl  += 5;
```

Die Bedeutung dieses Ausdrucks ist äquivalent zu:

```
zahl = zahl + 5;
```

### Programm *Zuweisung06*: Kombinierte Zuweisungsoperatoren

```
public class Zuweisung06 {
    public static void main(String args[]) {
        int x = 1;
        int y = 10;
        x += 13;                // x = x + 13;
        System.out.println(x);
        y *= y + 5;             // y = y * (y + 5);
        System.out.println(y);
    }
}
```

Obwohl das Arbeiten mit diesen kombinierten Operatoren zu einer verkürzten Schreibweise führt, wird der Einsatz nicht empfohlen, weil die Lesbarkeit darunter leidet. Dies demonstriert besonders die vorletzte Zeile des Programms *Zuweisung06.java*.

## 8.2.3 Mehrere Operanden als Empfänger der Zuweisung

Eine weitere Variante des Zuweisungsoperators erlaubt das Arbeiten mit mehreren Empfänger-Operanden.

### Programm *Zuweisung07*: Mehrere Empfängeroperanden

```
public class Zuweisung07 {
    public static void main(String args[]) {
        int x = 1;
        int y = 10;
        x = y = 0;
        System.out.println("x= " + x + "  y= " + y);
    }
}
```

Für diese Codierung ist eine wichtige Besonderheit zu beachten: die Reihenfolge der Ausführung ist nicht - wie sonst allgemein üblich - von links nach rechts, sondern

sie beginnt rechts mit:  $y = 0$ , danach wird dann ausgeführt:  $x = y$ . Es wird jedoch empfohlen, diese Schreibweise zu vermeiden.

### Programm Zuweisung08: Stilfragen für Zuweisung

```
public class Zuweisung08 {
    public static void main(String args[]) {
        int zahl;
        int zahl1;
        System.out.println(zahl = 2);
        System.out.println(zahl +=5);           // Vermeiden!
        System.out.println(zahl1 = zahl = 6);   // Vermeiden!
    }
}
```

### Übung zum Programm Zuweisung08

Bitte ändern Sie das Programm so, dass es leichter lesbar ist. Vermeiden Sie den zusammengesetzten Zuweisungsoperator und auch die Verwendung von mehreren Operanden als Empfänger einer Zuweisung.

## 8.3 Steueranweisungen

Ein Programm ist eine Lösungsbeschreibung ("Algorithmus"), formuliert durch die Zusammenstellung von Anweisungen in einer symbolischen Programmiersprache. Fast alle bisherigen Beispiele dieses Buchs bestanden aus einer linearen Folge von elementaren Schritten, die **nacheinander** (sequentiell) und auch **nur jeweils einmal** ausgeführt wurden.

Diese Programme entsprechen natürlich nicht den realen Anforderungen. Sie nutzen auch in keiner Weise die Vorteile der Computer, nämlich

- die Fähigkeit, sehr schnell sehr viele Daten zu verarbeiten und
- die Möglichkeit, die Verarbeitung abhängig zu machen von Bedingungen.

Es sind also Kontrollstrukturen notwendig, die die Ordnung der Ausführung (die Reihenfolge und die Häufigkeit) festlegen. Seit in den 70er Jahren die Ideen der "Strukturierten Programmierung" von fast allen Programmiersprachen übernommen worden sind, gilt natürlich auch für objektorientierte Sprachen die Beschränkung auf wenige, genau definierte Elemente für die Ablaufsteuerung:

- Sequenz (= ein Befehl nach dem anderen)
- Iteration (= ein Befehl oder Befehlsblock wird **mehrfach** ausgeführt)
- Selektion (= ein Befehl oder Befehlsblock wird **alternativ** ausgeführt)
- Delegation (= Aufruf eines Unterprogrammes).

Für die **Sequenz** ist kein besonderes Schlüsselwort erforderlich, denn dies ist der Default-Ablauf: Die Befehle werden von links nach rechts (wenn mehrere Statements

in einer Zeile stehen) und dann von oben nach unten ausgeführt. Die **Selektion** und **Iteration** werden wir in diesem Kapitel behandeln. Unter **Delegation** versteht man den Aufruf einer Unterroutine, damit dort dann die gewünschte Aufgabe ausgeführt wird, u.U. sogar parallel zum weiteren Ablauf ("Thread"). Der Aufruf solcher Unter Routinen geschieht in Java durch das Senden von Nachrichten - das heißt, durch Methodenaufruf (siehe Kapitel 10 und folgende). Es gibt in Java noch eine weitere Form der Ablaufsteuerung, die in diesem Buch jedoch nur kurz behandelt wird: Für die Ausnahmebehandlung (exception handling, Fehlerbehandlung) gibt es die Schlüsselwörter: *try-catch-finally* und *throw*.

Die in der Abbildung 8.1 aufgelisteten Formen von "control flow-statements" in Java werden wir in diesem Kapitel ausführlich besprechen.

Anweisungstyp	Schlüsselwörter
Verzweigung (Selektion)	if-else, switch-case
Wiederholung (Loop, Schleife, Iteration)	while, do-while, for
Sprunganweisungen	break; continue; label; return;

Abb. 8.1: Drei Arten von Steueranweisungen

Allen Steueranweisungen gemeinsam ist, dass sie mit Ausdrücken arbeiten, die vom Boolean-Datentyp sein müssen, d.h. diese müssen nach der Auswertung entweder *true* oder *false* liefern.

Die Java-Sprachmittel dieser algorithmischen Grundformen unterscheiden sich kaum von den Sprachelementen in anderen Programmiersprachen. Sie sind z.B. fast identisch mit der Syntax der Steueranweisungen in der C++-Sprache. Deswegen können die Denkmuster, die in diesem Kapitel eingeübt werden, durchaus auf andere Sprachen übertragen werden.

## 8.4 Verzweigungen (Selektion, Auswahl)

### 8.4.1 *if*-Statement

Das *if*-Statement ermöglicht es dem Programmierer, andere Anweisungen selektiv ausführen zu lassen.

#### 8.4.1.2 Die Grundformen der *if*-Anweisung

Die Syntax für die einfachste Form der *if*-Anweisung ist:

```
if (ausdruck) {
    // Ja-Block (Dann-Block);
}
```

Der Ausdruck **muss** in runden Klammern stehen! Nur wenn die Auswertung des Ausdrucks *true* ergibt, wird der Ja-Block ausgeführt (bedingte Anweisung, conditional statement). Liefert der Ausdruck den Wert *false*, so wird der Block übersprungen. Zusätzlich können Anweisungen programmiert werden, die nur ausgeführt werden, wenn der Ausdruck *false* ist. Damit hat eine komplette *if*-Anweisung folgende Syntax:

```
if (ausdruck) {  
    // Ja-Block (Dann-Block);  
}  
else  
    // Nein-Block (Andernfalls-Block);  
}
```

Der Ausdruck wird evaluiert und liefert entweder *true* oder *false*. Wenn der Ausdruck wahr ist, werden die Anweisungen im Ja-Block ausgeführt, die Anweisungen im Nein-Block werden dann übersprungen. Liefert der Ausdruck den Wert *false*, so werden nur die Anweisungen des Nein-Blocks ausgeführt.

### Programm *If01*: Komplettes *if*-Beispiel mit Ja-Block und Nein-Block

```
public class If01 {  
    public static void main(String args[]) throws Exception {  
        if (5 < 3) {  
            System.out.println("5 ist kleiner 3");  
        }  
        else {  
            System.out.println("5 ist nicht kleiner als 3");  
        }  
    }  
}
```

Die geschweiften Klammern können entfallen, wenn der Block aus nur einer Anweisung ("simple statement") besteht. Doch wird empfohlen, immer mit Klammern zu arbeiten, damit ein Block entsteht ("compound statement"), weil die Programme dadurch besser lesbar und wartbar werden.

### Struktogramme

Gemeinsam mit den Ideen der "Strukturierten Programmierung" entstanden Darstellungsformen für Algorithmen, mit denen die Einhaltung der Entwurfs- und Codiertechniken erzwungen wurde: die Nassi-Shneidermann-Diagramme (Struktogramme, siehe Kapitel 9). Wir werden einige der grundlegenden Symbole dieser Notation bereits einführen, bevor dann im Kapitel 9 eine ausführliche Darstellung und Einordnung der Diagramme erfolgt.

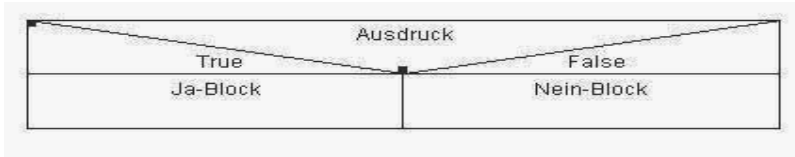


Abb. 8.2: Struktogramm für die If-Anweisung

**Programm If02: Beispiel für einseitige IF-Anweisung**

```
public class If02 {
    public static void main(String args[]) {
        char c = 'A';
        if (Character.isUpperCase(c)) {
            System.out.println("Der Zeichen " + c +
                               " ist ein Grossbuchstabe");
        }
    }
}
```

**Programm If03: Beispiel für zweiseitige IF-Anweisung**

```
public class If03 {
    public static void main(String args[]) {
        char c = 'a';
        if (Character.isUpperCase(c)) {
            System.out.println("Der Zeichen " + c +
                               " ist ein Grossbuchstabe");
        }
        else {
            System.out.println("Das Zeichen " + c +
                               " ist ein Kleinbuchstabe");
        }
    }
}
```

Das folgende Beispiel enthält einen hinterhältigen Fehler. In der Zeile 3 steht ein Semikolon, dieses wirkt wie eine Leeranweisung und beendet die If-Anweisung.

**Programm Leeranweisung01: Beliebter, aber schwer zu entdeckender Fehler**

```
public class Leeranweisung01 {
    public static void main(String[] args) {
        if (3>5);
        System.out.println("3 ist größer 5");
    }
}
```



### Stilfragen (oder: Einrücken ist wichtig)

Zwar ist Java eine formatfreie Sprache (d.h. es gibt keine Regeln für die Aufteilung einer Zeile), trotzdem sollten einige bewährte Regeln für die äußere Form des Quelltexts eingehalten werden. Zur besseren Lesbarkeit sind besonders die Einrückungsregeln ("indentation rules") zu beachten. Dies gilt auf jeden Fall für das Codieren von Alternativen oder Wiederholungen, denn durch diese Steueranweisungen bekommen Programme (oder konkreter: Methoden) eine bestimmte Struktur.

Und die Erfahrung zeigt, dass die Fehlersuche im Quelltext und die Einarbeitung in (eigene oder fremde) Programme erheblich erleichtert werden, wenn diese Struktur äußerlich erkennbar wird ("wenn der dynamische Programmablauf aus der statischen Niederschrift ersichtlich ist").

### Arbeiten mit einem komplexen Ausdruck

Ein Ausdruck, der den weiteren Ablauf steuert, kann beliebig komplex sein, d.h. es können beliebig viele Aussagen durch logische Operatoren verknüpft werden.

### Programm *If04*: Komplexer Ausdruck für die Ablaufsteuerung

```
public class If04 {
    public static void main(String args[]) {
        boolean fehler = false;
        int monat = 12;
        if ((!fehler) && (monat > 0) && (monat < 13))
            System.out.println("Alles o.k.");
        else
            System.out.println("Fehler aufgetreten oder Monat falsch");
    }
}
```

Hinter dem Schlüsselwort *if* muss ein Ausdruck in runden Klammern stehen, der zwar aus beliebig vielen Teilbedingungen bestehen kann, am Ende aber lediglich einen einzigen Wahrheitswert liefern muss: *true* oder *false*.

Besonders interessant ist in dem Programm *If04* die Überprüfung der booleschen Variablen *fehler*. Wenn die Bedingung erfüllt ist, enthält die Variable *fehler* den Wert *true*. Und das soll bedeuten, dass ein Fehler aufgetreten ist. In diesem Programm bekommt sie den Wert *false*, und das heißt, es ist alles in Ordnung, es ist kein Fehler aufgetreten. Deswegen muss in dem Ausdruck der NICHT-Operator ! benutzt werden.

### Übung zum Programm *If04*

Testen Sie das Programm mit dem Monat 13. Ändern Sie danach die Bedingung so ab, dass nur die Monate des 3. Quartals (von 7-9) als korrekt zugelassen werden.

### 8.4.1.2 Geschachtelte IF-Anweisung

Noch komplexer wird eine *if*-Anweisung, wenn innerhalb des Ja-Blocks oder innerhalb des Nein-Blocks eine weitere *if*-Anweisung steht. Die Aufgabe des nachfolgenden Programms soll es sein, zwei Zahlen zu vergleichen.

#### Programm If05: If-Anweisung innerhalb des Nein-Zweiges (schlecht codiert)

```
public class If05 {
    public static void main(String args[]) {
        int a = 2500;
        int b = 1500;
        if (a < b) {
            System.out.println("a ist kleiner als b ");
            if (b > 1000)
                System.out.println("und b ist groesser als 1000");
        }
        else {
            System.out.println("a ist nicht kleiner als b ");
            if (b > 1000)
                System.out.println("und b ist groesser als 1000");
        }
    }
}
```

Die IF-Schachtelung kann beliebig tief erfolgen. Aus Gründen der Übersichtlichkeit wird empfohlen, nicht mehr als 2 oder 3 geschachtelte Alternativen innerhalb eines Zweiges zu kodieren. Ausnahmen können so genannte *if*-Kaskaden sein, bei denen jeweils im *else*-Zweig wiederum eine *if*-Anweisung steht:

#### Programm If06: If-Else-Kaskaden

```
public class If06 {
    public static void main(String args[]) {
        char c1 = 'z';
        char c2 = 'z';
        if (c1 == ' ')
            System.out.println("c1 ist leer");
        else if (c1 < c2)
            System.out.println("c1 ist kleiner c2");
        else if (c1 > c2)
            System.out.println("c1 ist groesser c2");
        else if (c1 == c2)
            System.out.println("c1 ist gleich c2");
    }
}
```

### 8.4.1.3 Dangling Else

Bei geschachtelten *if* s kann es zu mehrdeutigen Situation kommen, nämlich dann, wenn es eine unpaarige Anzahl von *if* und *else*-Schlüsselwörtern gibt. Dann kann es passieren, dass ein *else* "in der Luft hängt" (dangle, engl. für baumeln). Beispiel:

```
if (a > b)
    if (a > 0 ....
else
    ....
```

Daraus ergibt sich die Frage: Zu welchem *if* gehört der *else*-Zweig? Antwort: *else* gehört immer zum unmittelbar davor stehenden *if*.

#### Programm If07: if-else-Schachtelung

```
public class If07 {
    public static void main(String args[]) {
        int dollar = 0;
        boolean kreditkarte=true;
        if (!kreditkarte) {
            if (dollar == 0)
                System.out.println("Weder Dollar noch Karte");
        }
        else
            System.out.println("Kreditkarte vorhanden");
    }
}
```

Die Regel, wie die Mehrdeutigkeit aufgelöst wird, sollte auch aus der Schreibweise erkennbar werden (richtig einrücken!).

#### Übung zum Programm If07

In diesem Programm besteht der Ja-Zweig des ersten *If* s aus nur einer Anweisung. Deswegen könnte man (fälschlicherweise) auf die Idee kommen, die geschweiften Klammern (in Zeilen 5 und 8) wegzulassen. Formal ist das in Ordnung, logisch leider falsch. Die Ausführung läuft dann nämlich nicht so, wie es die Einrückung erwarten lässt. Bitte testen Sie dies durch entsprechende Programmänderung.

Das nachfolgende Programm *If08* soll folgende Aufgabe lösen: Zunächst soll überprüft werden, ob es stimmt, dass *a* kleiner ist als *b*. Wenn das stimmt, soll abgefragt werden, ob *b* > 1000 ist und abhängig davon soll zusätzlich ausgegeben werden: "*b* ist größer 1000".

#### Lösungsvorschlag (fehlerhaft):

```
public class If08 {
    public static void main(String args[]) {
```

```

int a = 2500;
int b = 1500;
if (a < b)
    if (b > 1000)
        System.out.println("a ist kleiner b " +
            "und b ist groesser als 1000");
    else
        System.out.println("a ist nicht kleiner b ");
System.out.println("Programmende");
}
}

```

### Übung zum Programm *If08*

Der Lösungsvorschlag *If08.java* ist falsch, es wird lediglich "Programmende" ausgegeben. Bitte prüfen Sie, wo der Fehler liegt und korrigieren Sie das Programm.

Hinweise zur Lösung: Das Problem des "dangling else" kann nur vorkommen, wenn die Anweisungen nicht durch Blockklammern begrenzt wurden.

#### 8.4.1.4 Bedingungsoperator (Fragezeichenoperator)

Es gibt einen Operator, dessen Wirkung dem If-Befehl sehr ähnelt, der Bedingungsoperator. Deswegen werden wir ihn an dieser Stelle besprechen. Der Bedingungsoperator besteht aus drei Operanden und den beiden Zeichen ? und :. Er ist der einzige "ternäre" Operator, alle anderen sind "unäre" (z.B. ++ oder --) oder "binäre" (z.B. + oder &&) Operatoren.

Allgemeine Syntax eines Bedingungsausdrucks:

*Boolescher ausdruck* ? *ausdruck1* : *ausdruck2*;

Der boolesche Ausdruck wird ausgewertet. Wenn er den Wert *true* ergibt, wird *ausdruck1* evaluiert **und** als Ergebnis geliefert, andernfalls der *ausdruck2*. Beide Ausdrücke müssen den gleichen Datentyp haben, dies wird vom Compiler überprüft.

Beispiel:

$a > b ? x + 5 : y / 5$

Wie bei einem Ausdruck üblich, wird *ein* Ergebnis evaluiert. Abhängig von der Bedingung wird entweder  $x + 5$  oder  $y / 5$  als Ergebnis geliefert. Somit kann ein derart zusammengesetzter Ausdruck nur Teil einer kompletten Anweisung sein, z.B. ein Parameter eines Methodenaufrufs oder Teil einer Zuweisung.

#### Programm *Fragezeichen01*: Bedingungsoperator

```

public class Fragezeichen01 {
    public static void main(String[] args) {

```

```
        System.out.println((6 > 5) ? 17 + 5 : 25 / 5);
    }
}
```

Die beiden nächsten Beispiele zeigen, dass der Bedingungsoperator eine kompakte Alternative für einen ausführlichen If-Befehl sein kann.

### Programm *Fragezeichen02*: Zunächst die *if*-Anweisung

```
public class Fragezeichen02 {
    public static void main(String[] args) {
        String text = " ";
        if (6 > 5)
            text = "Groesser";
        else
            text = "Kleiner/Gleich";
        System.out.println(text);
    }
}
```

Wenn die Bedingung in einen Ausdruck eingebettet wird, kann auf die Hilfsvariable *text* verzichtet werden.

### Programm *IfOperator02*: Und nun der Bedingungsoperator

```
public class IfOperator02 {
    public static void main(String[] args) {
        System.out.println((6 > 5) ? "groesser" : "kleiner/gleich");
    }
}
```

Die Syntax für den Bedingungsoperator ist allerdings nicht immer leicht zu lesen. Deswegen gilt die Empfehlung: Verwenden Sie den Operator *?:* nur in begründeten Ausnahmefällen.

## 8.4.2 Switch-Statement

Wenn der weitere Ablauf eines Programms mehr als zwei Möglichkeiten bietet, muss eine *if-else*-Kaskade programmiert werden, denn der If-Befehl kennt nur eine Alternative: entweder - oder. Wenn aber alle Zweige von dem Wert einer einzigen Variablen abhängen, so kann evtl. mit der *switch*-Anweisung gearbeitet werden.

**Beispiel:** Ein Programm fordert vom Bediener einen numerischen Wert für einen beliebigen Monat des 1. Halbjahres an (1 bis 6). Die weitere Verarbeitung hängt allein von diesem einen Wert ab.

### Programm *Switch01*: Monatstexte für das 1. Halbjahr mit *if* ausgeben

```
import java.util.*;
```

```

public class Switch01 {
    public static void main(String[] args) {
        System.out.println("Bitte geben Sie den Monat 1.Halbjahr ein: ");
        Scanner eingabe = new Scanner(System.in);
        int monat = eingabe.nextInt();
        if (monat == 1) System.out.println("Januar");
        else if (monat == 2) System.out.println("Februar");
        else if (monat == 3) System.out.println("März");
        else if (monat == 4) System.out.println("April");
        else if (monat == 5) System.out.println("Mai");
        else if (monat == 6) System.out.println("Juni");
        else System.out.println("Falsche Eingabe");
    }
}

```

Für diese Aufgabenstellung bietet sich an, den *switch*-Befehl einzusetzen. Dieser ist eine Variante des *If*-Befehls. Immer wenn eine Ganzzahl auf mehrere unterschiedliche Werte zu prüfen ist, kann *switch* eine Alternative zu geschachtelten *if* s sein.

Allgemeine Syntax:

```

switch (ausdruck) {
    case wert: anweisung-1;
    case wert: anweisung-2;
    default: anweisung-n;
}

```

Das Ergebnis des Ausdrucks wird mit jedem Wert verglichen. Wenn einer übereinstimmt, werden die nachfolgenden Anweisungen ausgeführt. Wenn keiner passt, wird die Anweisung hinter *default* ausgeführt.

### Was sind die Beschränkungen für den "ausdruck" und für den "wert"?

Der "**ausdruck**" muss als Ergebnis einen numerischen Wert liefern. Dieser Wert muss ganzzahlig - oder noch genauer: vom *int*-Typ, sein. Ergibt die Auswertung des Ausdrucks einen *short*-, *byte*- oder *char*-Typ, wird dieser umgewandelt in *int*-Typ. Nicht erlaubt sind die primitiven Typen *boolean*, *long*, *float* oder *double*.

Der "**wert**" hinter dem Schlüsselwort *case* muss ein "konstanter Ausdruck" sein. Darunter versteht man einen Wert, der zur Compilezeit bekannt ist, z.B. 5 oder (5 \* 3). Nicht erlaubt sind Variablen. Danach folgen der Doppelpunkt und eine einzelne Anweisung oder auch mehrere Anweisungen.

#### Kritischer Hinweis

Dies alles sind gravierende Einschränkungen für die Einsatzmöglichkeiten des *switch*-statements. Dieses Sprachkonstrukt ist nicht annähernd so leistungsfähig wie

ähnliche Möglichkeiten in anderen Sprachen (es fehlt z.B. die Abfrage von Wertebereichen oder die Möglichkeit, einen Ausdruck hinter *case* zu formulieren).

### Besonderheiten der *switch*-Syntax

Die Anweisungen hinter dem *case*-Schlüsselwort müssen nicht als Block geklammert sein. Ein "Fall" wird durch das nächste Schlüsselwort *case* beendet.

Aber Achtung: Grundsätzlich gilt, dass beim ersten Auftreten des gesuchten Wertes die Verarbeitung beginnt und **alle** nachfolgenden "case" ausgeführt werden, und zwar solange, bis ein *break* gefunden wird oder bis die *switch*-Anweisung beendet ist.

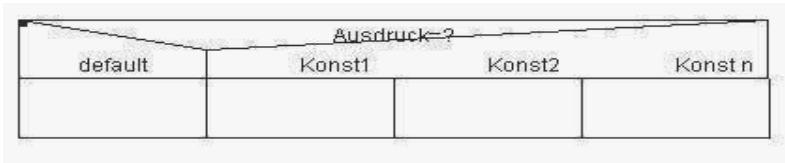


Abb. 8.3: Struktogramm für Switch (Mehrfachverzweigung)

### Programm *Switch02*: Monatstexte für das 1. Halbjahr mit *switch* ausgeben

```
import java.util.*;
public class Switch02 {
    public static void main(String[] args) {
        System.out.println("Bitte Monat eingeben (1.Halbjahr)");
        Scanner eingabe = new Scanner(System.in);
        int monat = eingabe.nextInt();
        switch (monat) {
            case 1: System.out.println("Januar"); break;
            case 2: System.out.println("Februar"); break;
            case 3: System.out.println("Maerz"); break;
            case 4: System.out.println("April"); break;
            case 5: System.out.println("Mai"); break;
            case 6: System.out.println("Juni"); break;
            default: System.out.println("Der Monat ist falsch");
        }
        System.out.println("Programm-Ende");
    }
}
```

### Übung zum Programm *Switch02*

Bitte prüfen Sie durch Programmänderung, ob anstelle der konstanten Zahlen hinter dem Schlüsselwort *case* auch ein Variablenname stehen kann.

Zwei Angaben sind besonders zu beachten:

- Zum einen die Anweisung *break* in jedem *case*. Sie beendet die weitere Ausführung der Switch-Anweisung und übergibt die Steuerung an den nächsten Befehl hinter dieser Switch-Anweisung. Wenn das *break*-Statement fehlt, wird mit dem nächsten *case* weitergemacht.
- Zum anderen das Schlüsselwort *default*. Diese Angabe ist optional. Damit können alle nicht abgefragten Fälle abgefangen und behandelt werden.

### Programm *Switch03*: Die Wirkung der *break*-Anweisung

```
import java.util.*;
public class Switch03 {
    public static void main(String[] args) throws Exception {
        System.out.println("Bitte einen Buchstaben eingeben:");
        Scanner eingabe = new Scanner(System.in);
        byte c = eingabe.nextByte();
        switch (c) {
            case 'A': System.out.println("a");
            case 'B': System.out.println("b");
            case 'C': System.out.println("c");
            case 'E': System.out.println("e");
            default : System.out.println ("Alle anderen Buchstaben");
        }
    }
}
```

Die Methode *nextByte()* erwartet den Integerwert des ASCII-Zeichens, z.B. 65 für A.

### Übung zum Programm *Switch03*

Testen Sie das Programm durch Eingabe der Zahl 66 (für das ASCII-Zeichen 'B'). Überprüfen Sie die Ausgabe. Da diese offensichtlich falsch ist, korrigieren Sie das Programm. Hinweis: Es fehlt die *break*-Anweisung.

Das folgende Beispiel errechnet die Anzahl Tage, die ein Monat hat (unter Berücksichtigung der Schaltjahre). Ein Schaltjahr ist durch 4, aber nicht durch 100 teilbar, es sei denn, es ist durch 400 teilbar.

### Programm *Switch04*: Anzahl Tage eines Monats ermitteln

```
public class Switch04 {
    public static void main(String[] args) {
        int monat = 2;
        int jahr = 2000;
        int tage = 0;
```



```
switch (monat) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        tage = 31; break;
    case 4:
    case 6:
    case 9:
    case 11:
        tage = 30; break;
    case 2:
        if ( ((jahr % 4 == 0) && !(jahr % 100 == 0))
            || (jahr % 400 == 0) ) tage = 29;
        else
            tage = 28; break;
}
System.out.println("Anzahl der Tage = " + tage);
}
```

Die leeren *case*-Klauseln wirken wie ein einschließendes Oder. Sobald der Ausdruck mit der Konstanten übereinstimmt, beginnt die Ausführung aller nachfolgenden Anweisungen bis zum Ende des Switch-Statements, und sie wird nur abgebrochen, wenn ein *break*-Statement auftritt.

### enum-Typen in switch-Anweisung verwenden

Neben den "normalen" Klassen gibt es auch *enum*-Typen, die ebenfalls in Klassen beschrieben werden können (Erläuterungen siehe Kapitel 11.9.2). Enum-Typen werden auch Aufzählungstypen genannt, weil die möglichen Werte, die eine Variable dieses Typs aufnehmen kann, in Form einer Aufzählung angegeben werden. Dann kann eine Variable vom *enum*-Typ als *case*-Wert benutzt werden.

### Programm *Switch05*: Abfrage von einem *enum*-Typ abhängig machen

```
public class Switch05 {
    enum Wochentage {montag, dienstag, mittwoch, donnerstag,
        freitag, samstag, sonntag}

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

```

Wochentage w = Wochentage.samstag;
switch (w) {
    case montag: System.out.println("Wochenbeginn");
    case samstag: System.out.println("Arbeitsfrei");
}
}
}

```

Fazit: Die *switch*-Anweisung wird eingesetzt, wenn vom Wert einer numerischen Variablen (die kann auch ein *enum*-Typ sein) mehrere Alternativen abhängen. Voraussetzung ist: Es wird der Wert einer ganzzahligen Variablen abgefragt (*byte*, *short*, *char* oder *int*), *long* ist nicht erlaubt.

## 8.5 Schleifen (Iteration, Wiederholung, Loop)

Mit Schleifenbefehlen kann man erreichen, dass Programmteile wiederholt, also mehr als einmal, ausgeführt werden. Eine Schleife wird normalerweise beendet, wenn ein bestimmter Zustand erreicht ist. Sie besteht aus einem Kopf und dem Rumpf. Die allgemeine Syntax sieht so aus:

```

schleifenbefehl (Boolescher ausdruck) {
    // Schleifenrumpf (Ausführungsblock)
}

```

Das Ergebnis des Booleschen Ausdrucks entscheidet darüber, ob der Block durchlaufen wird oder nicht. Erforderlich sind normalerweise Anweisungen im Schleifenrumpf, die die Bedingung so modifizieren, dass ein bestimmter Zustand erreicht wird, der die Schleifendurchführung beendet. Andernfalls würde eine Endlosschleife programmiert.

Abhängig vom Zeitpunkt, wann diese Überprüfung stattfindet, unterscheidet man in Java zwei Arten von Schleifen-Anweisungen:

- die **kopfgesteuerte** *while*-Anweisung (abweisende Schleife) und
- die **fußgesteuerte** *do*-Anweisung (nicht-abweisende Schleife).

Außerdem gibt es eine weitere Art, nämlich

- die **Zählschleife** (*for*-Schleife), das ist eine besondere Form der abweisenden Schleife.

### 8.5.1 While-Schleife

Mit der While-Schleife kann man erreichen, dass Programmteile solange wiederholt werden, bis eine Bedingung *false* ergibt.

Syntax der *while*-Schleife:

```
while (Boolescher Ausdruck) {  
    // Schleifen-Block  
}
```

Der Boolesche Ausdruck wird ausgewertet. Wenn er *true* ergibt, wird der Schleifenblock ausgeführt. Danach wird der Ausdruck erneut ausgewertet und abhängig vom Ergebnis der Block noch einmal wiederholt oder nicht. Wenn der Boolesche Ausdruck *false* liefert, wird mit dem nächsten Befehl nach dem Schleifenrumpf fortgesetzt.

Weil *vor* Ausführung des Blocks der Ausdruck überprüft wird, nennt man die *while*-Schleife "kopfgesteuert" oder "abweisend". Der Schleifenrumpf wird solange ausgeführt, wie die Bedingung erfüllt ist.

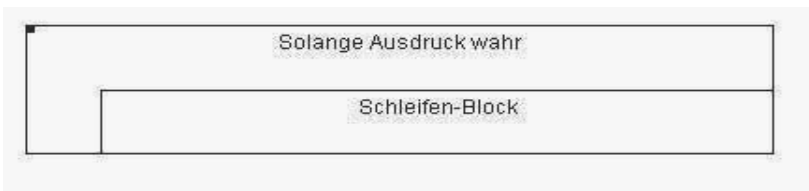


Abb. 8.4: Struktogramm für *while*-Schleife (abweisende Schleife)

### Programm *While01*: Ausgabe der Zahlen von 1 bis 4

```
public class While01 {  
    public static void main(String[] args) {  
        int zahl = 1;  
        while (zahl < 5)  
            System.out.println(zahl++);  
    }  
}
```

Eine Schleife hat drei Bestandteile:

- Kopf der Schleife
- Rumpf der Schleife
- Modifikation der Bedingung (andernfalls Endlosschleife).

### Übungen zum Programm *While01*

**Übung 1:** Identifizieren Sie für dieses Programm die 3 Bestandteile einer jeder vollständigen Schleife, indem Sie die jeweilige Zeilen-Nr. angeben.

**Übung 2:** Modifizieren Sie das Programm so, dass die Summe der Zahlen 1 - 4 gebildet und am Ende der Schleife ausgegeben wird.

**Programm While02.java: Ausgabe der Kleinbuchstaben von a bis z**

```
public class While02 {  
    public static void main(String[] args) {  
        char buchstabe = 'a';  
        while (buchstabe <= 'z')  
            System.out.println(buchstabe++);  
    }  
}
```

Das Programm *While03.java* modifiziert das Programm *While02.java* so, dass zwar weiterhin Kleinbuchstaben iteriert werden, aber der entsprechende Großbuchstabe ausgegeben wird. Laut Unicode-Tabelle ist die Platz-Nummer des Kleinbuchstabens um 32 höher als die Platz-Nummer des Großbuchstabens.

**Programm While03.java: Ausgabe der Großbuchstaben von A bis Z**

```
public class While03 {  
    public static void main(String[] args) {  
        char buchstabe = 'a';  
        while (buchstabe <= 'z') {  
            System.out.println((char) (buchstabe - 32));  
            buchstabe++;  
        }  
    }  
}
```

**Programm While04.java: Endlos-Schleife**

```
import java.util.*;  
public class While04 {  
    public static void main(String[] args) {  
        Scanner eingabe = new Scanner(System.in);  
        String zeile;  
        while (true) {  
            System.out.println("Bitte Text eingeben:");  
            zeile = eingabe.next();  
            System.out.println("Eingegeben wurde: " + zeile);  
        }  
    }  
}
```

Das Programm *While04.java* liest zeilenweise von *System.in*. Die Wörter einer Eingabezeile werden aufgesplittet und in jeweils einer eigenen Zeile ausgegeben ("parzen" der Eingabezeile). Weil die Bedingung der *while*-Schleife das Literal *true* ent-

hält, ist sie immer wahr. Deswegen kann diese Endlosschleife nur gewaltsam (unter Windows mit CTRL-C) abgebrochen werden

Im Programm *While05.java* werden vom Bediener Zahlen angefordert, die im Programm solange aufaddiert werden, bis die Summe den Wert von 100 erreicht hat.

### Programm *While05*: Zahlen aufsummieren

```
import java.util.*;
public class While05 {
    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        int zahl;
        int summe = 0;
        while (summe < 100) {
            System.out.println("Bitte Zahl eingeben:");
            zahl = eingabe.nextInt();
            summe = summe + zahl;
        }
        System.out.println("Die Summe betraegt: " + summe);
    }
}
```

### Programm *While06*: Logischer Fehler

```
public class While06 {
    public static void main(String[] args) {
        boolean ichBinMillionaeer = false;
        while (ichBinMillionaeer == true); {
            System.out.println("Ich bin Millionaeer");
        }
    }
}
```

Das Programm gibt (wenn Sie es genau so abtippen wie vorgegeben!) den Text aus: " Ich bin Millionaeer " - aber das ist gelogen, denn die Variable *ichBinMillionaeer* enthält den Wert *false*.

### Übung zum Programm *While06*

Versuchen Sie selbstständig, den Fehler zu finden und korrigieren Sie das Programm.

Lösungshinweis: Das Programm *Leeranweisung01.java* im Abschnitt 8.4. enthält denselben Fehler.

### 8.5.2 Do-Schleife

Mit der Do-Anweisung wird eine fußgesteuerte Schleife realisiert. Dadurch kann man erreichen, dass ein Programmteil mindestens einmal ausgeführt und danach immer wieder, bis eine Bedingung *false* ergibt. Die Syntax der Do-Schleife ist:

```
do {
    // Schleifen-Block
} while (Boolescher Ausdruck);
```

Der Schleifenblock wird ausgeführt und danach der Boolesche Ausdruck evaluiert. Wenn das Ergebnis *true* ist, beginnt ein neuer Durchlauf. Der Schleifenblock wird dann erneut ausgeführt, und am Ende wird geprüft, ob ein weiterer Durchlauf erforderlich ist usw.

Diese Art der Schleifenbildung nennt man "fußgesteuert" oder "nicht-abweisend", denn die Prüfung erfolgt **nach** dem Schleifendurchlauf. Der Block wird auf jeden Fall einmal, vielleicht sogar mehrmals durchlaufen.



Abb.8.5: Struktogramm für Do-Schleife (Prüfung am Ende)

#### Programm Do01: Ausgabe, solange zahl < 5

```
public class Do01 {
    public static void main(String[] args) {
        int zahl = 5;
        do
            System.out.println(zahl++);
        while (zahl < 5);
    }
}
```

Also: Obwohl die Bedingung nicht erfüllt ist, wird der Schleifenrumpf ausgeführt, weil bei der Do-Schleife die Prüfung erst am Ende der Schleife stattfindet.

#### Übung zum Programm Do01

Ändern Sie das Programm so, dass es (analog zum Programm *While01.java*) die Zahlen 1 - 4 ausgibt. Hinweis: Es ist nur die Änderung eines Literals notwendig.

Das Programm *Do02.java* hat eine ähnliche Aufgabenstellung, allerdings sollen jetzt die Zahlen von 5 - 1 ausgegeben werden (d.h. von 5 beginnend rückwärts).

### Programm Do02: Ausgabe der Zahlen von 5 bis 1 (rückwärts)

```
public class Do02 {
    public static void main(String[] args) {
        int zahl = 5;
        do
            System.out.println(zahl--);
        while (zahl > 0);
    }
}
```

### Übung zum Programm Do02

Lösen Sie die Aufgabe mit der *while*-Schleife (anstelle der *do*-Anweisung).

### Lösungsvorschlag

```
public class While99 {
    public static void main(String[] args) {
        int zahl = 5;
        while (zahl > 0)
            System.out.println(zahl--);
    }
}
```

Im Programm *Do03.java* soll eine Zahl geraten werden. Die Zahl liegt zwischen 0 und 9. (Die richtige Antwort ist als Literal im Programm fest codiert. Sie lautet 4.)

### Programm Do03: Eine Zahl raten

```
import java.util.*;
public class Do03 {
    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        int zahl;
        do {
            System.out.println("Bitte Zahl zwischen 0-9 eingeben:");
            zahl = eingabe.nextInt();
        }
        while (zahl != 4);
        System.out.println("Treffer! Richtig geraten");
    }
}
```

### 8.5.3 For-Schleife

#### 8.5.3.1 Die Grundform der For-Schleife

Die For-Schleife ermöglicht eine sehr kompakte Schreibweise für eine Loop. Syntax:

```
for (Initialisierung; Bedingung; Modifikation) {
    // Schleifenblock
}
```

In den runden Klammern hinter dem Schlüsselwort *for* stehen alle Angaben, die für die Steuerung einer Schleife notwendig sind:

- Initialisierung: Festlegen der Anfangswerte für die Entscheidung, ob ein Schleifendurchlauf erfolgen soll oder nicht.
- Bedingung: Formulieren des Ausdrucks, der über die Beendigung der Schleife entscheiden soll.
- Modifikation: Änderungen der Werte, die Teil der Bedingung sind, damit die Schleife (irgendwann) terminiert werden kann.

Als vierter Teil einer jeden Schleife muss dann noch der Schleifenblock codiert werden.



Abb. 8.6: Struktogramm für For-Schleife (Zählschleife)

#### Programm *For01*: Zahlen von 1 – 10 ausgeben

```
public class For01 {
    public static void main(String[] args) {
        for (int zahl = 1; zahl < 11; zahl++)
            System.out.println(zahl);
    }
}
```

Im Kopf dieser Schleife stehen die drei notwendigen Angaben:

- Deklaration und Initialisierung der Bedingungsvariablen **zahl** (auch "Laufvariable" genannt). Weil die Laufvariable hier deklariert wird, handelt es sich um eine lokale Variable, die auch nur im Block der For-Schleife ansprechbar ist. Außerhalb der Schleife ist sie nicht "sichtbar".
- Formulierung der Bedingung selbst (**zahl < 11**)



- Modifikation der Laufvariablen (**zahl++**).

Die For-Schleife wird auch von-bis-Schleife genannt, weil die Laufvariable den Startwert angibt, die typischerweise durch Inkrement hochgezählt wird, bis der Endwert erreicht ist.

### In welcher Reihenfolge wird die *For*-Schleife ausgeführt?

Vor Eintritt in die Schleife wird der Startwert festgelegt (durch Initialisierung der Laufvariablen). Dies erfolgt nur einmal, danach nie wieder. Anschließend wird die Bedingung überprüft. Ist das Ergebnis *true*, so wird erst der Schleifenblock und anschließend die Modifikation ausgeführt. Bei *false* wird an das Ende der Schleife verzweigt und dort mit dem folgenden Befehl weiter gemacht. Somit ist die For-Schleife auch kopfgesteuert, sie wird null-mal, einmal oder mehrfach durchgeführt.

Die Schleifenkonstrukte sind austauschbar.

### Programm While07: *for*-Schleife im Programm For01 ersetzen durch *while*

```
public class While07 {
    public static void main(String[] args) {
        int zahl = 1;
        while (zahl < 11) {
            System.out.println(zahl);
            zahl++;
        }
    }
}
```

### Übung zum Programm For01

Modifizieren Sie das Programm so, dass nur jede zweite Zahl von 1 bis 10 ausgegeben wird, also die ungeraden Zahlen 1, 3, 5, 7, 9.

### Lösungshinweis:

Es reicht, die Inkrementierung der Laufvariablen zu ändern, d.h. die Schrittweite von 1 auf 2 zu erhöhen.

Das Programm *For02.java* modifiziert das Programm *For01.java* so, dass die Summe der Zahlen von 1 - 10 ermittelt und ausgegeben wird:

### Programm For02: Summe der Zahlen von 1 bis 10

```
public class For02 {
    public static void main(String[] args) {
        int sum = 0;
        for (int zahl = 1; zahl <= 10; zahl++) {
```

```
        sum = sum + zahl;
    }
    System.out.println("Summe ist = " + sum);
}
}
```

### Übung

Bitte codieren Sie ein neues Programm *For03.java*, das Sonderzeichen des Unicodes ausgibt, und zwar die Zeichen mit den Codepoints `\u0021` bis `\u002F`. Es soll mit einer FOR-Schleife gearbeitet werden. Denken Sie daran, dass auch mit *char*-Typen "gerechnet" werden kann.

### Lösungsvorschlag

```
public class For03 {
    public static void main(String[] args) {
        for (char c = '\u0021'; c < '\u002F'; c++)
            System.out.println(c);
    }
}
```

### FOR-Schleife: Abweisend oder nicht-abweisend?

Das letzte Beispiel soll noch einmal eindeutig die Frage klären, ob die FOR-Schleife eine abweisende oder eine nicht-abweisende Schleife ist. Dazu schreiben wir ein Programm, das zeigt, wie der Ablauf des Programms ist, wenn die Testbedingung der FOR-Schleife niemals erfüllt sein kann.

### Programm *For04*: Die Bedingung ist nie erfüllt

```
public class For04 {
    public static void main(String[] args) {
        for (int i=0; i < 0;)
            System.out.println("Ich werde nie ausgegeben");
        System.out.println("Programmende");
    }
}
```

Fazit: Die For-Schleife ist eine kopfgesteuerte, eine abweisende Schleife - in dieser Hinsicht vergleichbar mit der *while*-Schleife, weil die Prüfung des booleschen Ausdrucks vor Eintritt in die Schleife erfolgt.

### Besonderheiten:

- Die drei Ausdrücke im Kopf der For-Schleife müssen nicht immer alle vorhanden sein. Fehlt der erste oder der zweite Ausdruck, wird er durch ein Semikolon ersetzt.

### Programm For05: Initialisierung außerhalb, Modifikation innerhalb

```
public class For05 {
    public static void main(String[] args) {
        int i= 0;
        for (; i < 1;) {
            System.out.println("Ich werde einmal ausgegeben");
            i++;
        }
    }
}
```

Eine weitere Besonderheit zeigt das folgende Programm. Die erste und die dritte Komponente im Kopf einer FOR-Schleife können aus einer Aufzählung von mehreren Teilausdrücken bestehen. Die einzelnen Teilausdrücke werden (wie generell bei Aufzählungen in Java) dann durch Komma abgetrennt.

### Programm For06: Mehrere Modifikationen im Schleifenkopf

```
public class For06 {
    public static void main(String[] args) {
        for (int i=1, j=12; i<5 || j < 13; i++, j=j+2)
            System.out.println("Ich werde viermal ausgegeben");
    }
}
```

Achten Sie also besonders auf die Kommas zwischen den einzelnen Teilausdrücken.

### Übung zum Programm For06

Die Bedingung besteht aus zwei mit ODER verknüpften Teilaussagen. Wie ändert sich das Programmverhalten, wenn die beiden Bedingungen mit dem UND-Operator verknüpft werden?

Das nächste Programm zeigt eine fehlerhafte Prüfbedingung. Die Prüfung auf Schleifenende sollte generell von Ganzzahlen abhängig gemacht werden - nicht zu empfehlen ist die Abfrage eines Gleitkommawerts. Dies kann zu Endlosschleifen führen.

### Programm For07: Ungewollte Endlosschleife, weil Gleitkomma-Wert ungenau

```
public class For07 {
    public static void main(String[] args) throws Exception {
        for (float i=0; i != 5.0; i=i+0.2f) {
            System.out.println(i);
            if (i > 5) break;
        }
    }
}
```

### 8.5.3.2 Variante der For-Schleife: For-Each ("Enhanced For-Loop")

Wenn die For-Schleife zur Verarbeitung von Datensammlungen (z.B. Array oder Collection) eingesetzt wird und wenn nur lesend auf die einzelnen Komponenten zugegriffen wird, dann gibt es eine vereinfachte Variante der Basis-For-Schleife, die so genannte *for-each*-Schleife. Sie durchläuft die Datensammlung, von vorne beginnend, lückenlos bis zum Ende und stellt im Schleifenblock den Wert einer jeden Komponente zur Verfügung. Zum kompletten Verständnis sind Array-Kenntnisse erforderlich (siehe Kapitel 13).

#### Programm *ForEach01*: Sequentielles Lesen eines Arrays

```
class ForEach01 {
    public static void main(String[] args) {
        int[] sammlung = {1,5,7,3};
        for (int zahl : sammlung)
            System.out.println(zahl);
    }
}
```

Um ein Verständnis für die Syntax zu bekommen, wird der Doppelpunkt wie "in" gelesen, also "für jede Zahl in Sammlung...".

Weitere Informationen und Beispiele zu dieser Form einer *for-each*-Loop gibt es im Kapitel 13: Arrays).

#### Bewertung der unterschiedlichen Schleifen-Konstrukte

Es besteht kein grundsätzlicher Unterschied zwischen den drei Schleifenformen *while*, *do* und *for*. Jede FOR-Schleife kann auch mit WHILE gemacht werden und umgekehrt. Ebenso kann eine DO-Schleife durch eine FOR- oder WHILE-Schleife ersetzt werden. Allerdings ist - je nach Aufgabenstellung - mal die eine Form übersichtlicher und mal die andere.

Grundsätzlich soll die Art der Codierung die Problemstruktur widerspiegeln und auch das ausdrücken, was der Programmierer beabsichtigt, damit das Programm verständlich wird. So ist z.B. die For-Schleife besonders geeignet für das Durchlaufen von Arrays, andererseits ist das Codieren einer Endlosschleife durch *for(;;)* zwar möglich, verständlicher ist aber eine *while(true)*-Schleife.

Wenn bereits beim Kodieren der Schleife die Anzahl der Durchläufe festgelegt werden kann, bietet sich die Zählschleife an (von-bis-Schleife).

Wenn die Laufvariable auch nach Schleifenende benötigt wird, ist die *For*-Schleife nicht geeignet, dann sollte die *While*-Schleife eingesetzt werden. Wenn die Laufvariable im Rumpf manipuliert wird, ist die For-Schleife ebenfalls nicht so gut einsetzbar.

### 8.5.4 Schachtelung von Schleifen

Genau so wie IF-Befehle geschachtelt werden können, ist auch eine Schachtelung von Schleifenbefehlen möglich, z.B. kann eine Wiederholung eine weitere Wiederholung enthalten, wenn innerhalb einer *for*-Anweisung im Rumpf eine weitere *for*-Schleife steht. Alle Steuerbefehle sind auch beliebig kombinierbar.

#### Programm *For08*: For-Schleife schachteln

```
public class For08 {  
    public static void main(String[] args) {  
        for (int zahl1 = 1; zahl1 <= 3; zahl1++) {  
            for (int zahl2 = 1; zahl2 <=5; zahl2++)  
                System.out.println(zahl1 + " " + zahl2);  
        }  
    }  
}
```

Die Ausgabe des Programms *For08.java* sieht wie folgt aus:

```
1 1  
1 2  
2 1  
2 2  
3 1  
3 2
```

#### Übung

Bitte codieren Sie das Programm *For09.java*. Es soll mit einer For-Schleife arbeiten. Der Schleifenrumpf soll 3-mal durchlaufen werden und folgende Aufgaben erfüllen:

- Der Text "Aussenschleife" soll ausgegeben werden.
- Außerdem soll innerhalb dieses Rumpfs eine innere FOR-Schleife codiert werden, die zweimal durchlaufen wird und in der lediglich der Text "Innenschleife" ausgegeben wird.

#### Lösungsvorschlag

```
public class For09 {  
    public static void main(String[] args) {  
        for (int i=0; i < 3; i++) {  
            System.out.println("Aussenschleife");  
            for (int j = 0; j < 2; j++)  
                System.out.println("Innenschleife");  
        }  
    }  
}
```

### Übung zum Programm For09

Bitte lösen Sie die Aufgabenstellung der Programms *For09.java* mit einer geschachtelten *while*-Schleife.

#### Lösungsvorschlag

```
public class While08 {
    public static void main(String[] args) {
        int zahl1 = 1;
        while (zahl1 <= 3) {
            int zahl2 = 1;
            while (zahl2 <= 5) {
                System.out.println(zahl1 + " " + zahl2);
                zahl2++;
            }
            zahl1++;
        }
    }
}
```

Insbesondere für das Arbeiten mit mehrdimensionalen Arrays bieten geschachtelte FOR-Schleifen eine sehr übersichtliche Schreibweise (siehe Kapitel 13: Arrays).

## 8.6 Sprung-Anweisungen (break, continue)

In manchen Situationen ist es notwendig, die normale Komplettierung eines Schleifendurchlaufs zu verhindern. Dann muss der Rest der Schleifenrumpfs übersprungen werden. Beispiel: In einer Datei, die sequentiell gelesen wird, stehen alle Kundendaten eines Unternehmens. Es werden im Programm nur die Kunden mit einem bestimmten Mindestumsatz benötigt. Also muss eine Schleife programmiert werden, die etwa wie folgt aussieht:

```
{
    Lese aus der Datei solange noch Sätze vorhanden sind
    Prüfe, ob Mindestumsatz vorhanden
        Wenn nein, beginne die Schleife von vorn
        Wenn ja, verarbeite den gelesenen Satz
}
```

Der vorzeitige Abbruch eines Schleifendurchlaufs bedeutet, dass die restlichen Befehle des Schleifenrumpfs übersprungen werden. Die Ablaufsteuerung wird an eine andere Stelle transferiert.

Wie es danach weiter geht, hängt in Java von der Art der Sprung-Anweisung ab:

- mit einer *break*-Anweisung wird die Schleife **komplett** abgebrochen und

- mit einer *continue*-Anweisung wird **nur** der aktuelle Durchlauf vorzeitig beendet.

**Übung:** Versuchen Sie festzustellen, welche Art von vorzeitigem Schleifenabbruch in dem beschriebenen Fall der Umsatzdatei vorliegt.

Antwort: Der aktuelle Durchlauf wird beendet, aber es wird geprüft, ob ein erneuter Durchlauf notwendig ist. Es handelt sich also um die *continue*-Anweisung).

### 8.6.1 break-Anweisung

Mit der Break-Anweisung wird die aktuelle Schleife komplett beendet. Der Schleifenrumpf wird verlassen und die Verarbeitung mit der ersten Anweisung **nach** der Schleife fortgesetzt.

Das folgende Programm *Break01.java* soll eine Iteration maximal 10mal durchführen und dabei Zahlen aufsummieren. Die Summe der Zahlen darf jedoch den Wert 1000 nicht überschreiten. Ist dies der Fall, wird die Schleife vorzeitig abgebrochen.

#### Programm *Break01*: Abbruch der Schleife, wenn Summe erreicht

```
import java.util.*;
public class Break01 {
    public static void main(String[] args) {
        int zahl;
        int summe = 0;

        Scanner eingabe = new Scanner(System.in);

        for (int i=0; i < 10; i++) {
            System.out.println("Bitte Zahl eingeben: ");
            zahl = eingabe.nextInt();
            if ((summe + zahl) > 1000) {
                System.out.println("Die Summe ist > 1000");
                break;
            }
            else {
                summe = summe + zahl;
                System.out.println("Summe = " + summe);
            }
        }
    }
}
```

Typisches Anwendungsbeispiel für die *break*-Anweisung ist der Abbruch einer Endlosschleife:

```
for (; true ;) {
    ...
    if (....) break;    // Die komplette Schleife wird beendet
}
```

Bei geschachtelten Schleifen wird nur die innere Schleife abgebrochen.

### 8.6.2 continue-Anweisung

Mit der Continue-Anweisung wird (nur) der aktuelle Schleifendurchlauf vorzeitig beendet. Es wird zum Schleifenanfang gesprungen, und es wird eine erneute Auswertung des Booleschen Ausdrucks erzwungen.

#### Beispiel:

```
for (int i=0; i<10; i++) {    // 10mal die Schleife ausführen
    ...
    if (....) continue;      // Diesen Durchlauf abbrechen
    ...
}
```

Bei geschachtelten Schleifen wird nur die innere Schleife abgebrochen.

Das folgende Programm *Continue01.java* errechnet den Kehrwert der Zahlen -10 bis +10. Übersprungen werden soll bei dieser Berechnung aber die Null (0).

#### Programm *Continue01*: Innerhalb der Schleife prüfen und evtl. überspringen

```
public class Continue01 {
    public static void main(String[] args) {
        for (int i= -10; i <11; i++) {
            if (i == 0)
                continue;    // Laufende Iteration beenden
            System.out.println("Kehrwert von " + i + " = " + 1.0 / i);
        }
    }
}
```

#### Geschachtelte Schleifen abbrechen

Bei geschachtelten Schleifen wird nur der innere Schleifenrumpf abgebrochen und nicht etwa die gesamte Schleife.

#### Programm *Continue02*: Abbruch der inneren Schleife

```
import java.util.*;
public class Continue02 {
    public static void main(String[] args) {
```



```
int zahl;  
int summe = 0;  
  
for (int i=0; i < 3; i++) {  
    System.out.println("Aussenschleife " + i);  
    for (int j=0; j < 4; j++) {  
        if (i == 1) continue;  
        System.out.println("Innenschleife " + i);  
    }  
}  
}  
}
```

Wenn die Aufgabenstellung es jedoch verlangt, dass alle Schleifen abgebrochen werden müssen, so kann dies mit dem Beschriften von Schleifen durch "Label" sehr leicht realisiert werden.

### 8.6.3 Benannte Schleifen ("Label")

Sowohl die *Break*- als auch die *Continue*-Anweisung können um einen Identifier ("Marke") ergänzt werden, z.B.

```
continue hauptschleife;
```

In diesem Fall handelt es sich bei "hauptschleife" um den Namen der Schleife, die vorzeitig beendet werden soll. Der Name wird am Beginn der Schleife, die abgebrochen werden soll, codiert, gefolgt von einem Doppelpunkt ..

#### Programm Label01: Gezieltes Ende bei geschachtelten Schleifen

```
public class Label01 {  
    public static void main(String[] args) {  
        aussen:                                // Label vergeben  
        for (int i=0; i <4; i++) {  
            for (int j=0; j<4; j++) {  
                System.out.println("innen: j = " + j);  
                if (j == i) {  
                    System.out.println("\n");  
                    continue aussen;  
                }  
            }  
        }  
    }  
}
```

## Übungen zum Programm Label01

**Übung 1:** Testen Sie das Programmverhalten, wenn bei der *continue*-Anweisung das Label (die "Marke") weggelassen wird.

**Übung 2:** Bitte prüfen Sie, ob Label immer am Schleifenbeginn stehen müssen oder ob sie an einer beliebigen Stelle (auch z.B. weiter hinten in der Schleife) deklariert und angesprungen werden können.

Hinweise zur Lösung: Label sind in Java nur erlaubt als Namen für Schleifen. Es sind keine allgemeinen Sprungziele wie z. B. in manchen Script-Sprachen. Und wenn eine *break*- oder *continue*-Anweisung mit einem Label versehen wird, dann muss dieses Label auch in den aktuellen Schleifen gefunden werden. Es darf also nicht außerhalb stehen.

## Programm Label02: Fehlerhafte Codierung einer benannten Anweisung

```
public class Label02 {
    public static void main(String[] args) {
        for (int i=0; i < 3; i++) {
            if (i==2)
                break aussen;    // Umwandlungsfehler
        }
        aussen: for (int x=1; x < 0; x++) {
            System.out.println("So gehts nicht");
        }
    }
}
```

## *Break* und *Continue* - nur eine verschleierte GOTO-Variante?

Andere Programmiersprachen bieten einen GOTO-Befehl, um Sprünge innerhalb eines Programms zu realisieren. GOTO ist aber zu Recht verpönt (und im Java-Sprachumfang nicht vorhanden). Wodurch aber unterscheiden sich die besprochenen *break*- und *continue*-Befehle von einem GOTO? Antwort: Ihre Wirkung ist eindeutig festgelegt. Sie erlauben nur Sprünge von innen nach außen. Die Programmsteuerung wird entweder an den Schleifenanfang (bei *continue*) oder an das Schleifenende (bei *break*) übergeben. Es können **keine beliebigen Ziele** vorne oder hinten im Programm angesprungen werden wie beim GOTO. Dadurch bleibt die Übersicht erhalten, und Spagetticode wird vermieden.

Als Zusammenfassung noch ein klassisches Anwendungsbeispiel im Pseudocode (siehe Kapitel 9) für die beiden Möglichkeiten, einen Schleifendurchlauf vorzeitig abubrechen. Das Programmfragment soll eine Endlosschleife enthalten, in der die Daten gelesen werden. Innerhalb der Schleife wird überprüft, ob eine Verarbeitung der Daten erforderlich ist oder nicht. Bei Dateiende ist die Schleife beendet.

```
while (true) {
    if (dateiende) break;                // Schleife beenden
    if (Not-VerarbeitungErforderlich) continue; // Schleife von vorn beginnen
    ...
}
```

### 8.6.4 Return

Mit der Return-Anweisung wird ein Unterprogramm (eine "Methode") beendet und die Steuerung an die aufrufende Anweisung zurückgegeben. Gegebenenfalls wird auch ein Ergebnis zurückgeliefert. Weitere Erläuterungen stehen im Kapitel 10.

## 8.7 Lösungsmuster für Schleifen

Die folgenden Hinweise sind für den Anfänger nicht unbedingt wichtig. Sie können auch später bei Bedarf durchgearbeitet werden.

Typisches Einsatzgebiet für eine Schleifenbildung ist das Lesen von Datensammlungen, entweder von einem externen Speicher oder innerhalb des internen Speichers z.B. aus einer *Collection* von Objekten. Dabei ist folgendes Problem zu lösen: Häufig ist die Bedingung für die Wiederholung das Vorhandensein weiterer Daten im Eingabestrom oder in der Collection.

Also wird diese Bedingung **vor** der Ausführung des Schleifenrumpfs geprüft. Aber wie ist diese Prüfung möglich, wenn doch (beim ersten Durchlauf) noch gar nicht versucht worden ist zu lesen?

Folgendes Beispiel soll diese Problematik verdeutlichen.

### Programm *Next01*: Text von `System.in` lesen und verarbeiten

```
import java.io.*;
public class Next01 {
    public static void main (String[] args) throws Exception
    {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
        String str = " ";
        while (str != null) {
            System.out.println("Bitte Text eingeben: ");
            str = in.readLine();
            System.out.println("Verarbeitet wird; " + str);
        }
        System.out.println("Programmende");
    }
}
```

Das Programm soll Text vom Bildschirm lesen und als Echo wieder ausgeben. Die Leseschleife wird beendet, wenn die Eingabe vom Bediener *null* ist (wenn also keine Daten gelesen worden sind). Dies ist dann der Fall, wenn das Programm beendet wird ohne Dateneingabe (unter MS-Windows durch CTRL-C).

### Übung zum Programm *Next01*

Bitte testen Sie das Programm durch Eingabe einiger Zeilen.

Beenden Sie dann das Programm und starten Sie es neu. Geben Sie diesmal gar nichts ein, sondern beenden Sie es sofort. Wird die Eingabe *null* verarbeitet?

Wie verhält sich das Programm, wenn die Variable *str* mit *null* initialisiert wird?

Die Lösung im Programm *Next01.java* ist nicht befriedigend, weil die Wiederholungsbedingung (*str != null*) beim ersten Durchlauf abgefragt wird, obwohl sie erst später gesetzt wird durch das Lesen.

Für dieses immer wiederkehrende Problem, dass eine Abfrage der Bedingung beim ersten Durchlauf noch nicht möglich ist, weil die Voraussetzung fehlt, gibt es verschiedene Lösungsvarianten.

### Programm *Next02*: Erster Lösungsvorschlag - eine Endlosschleife

```
import java.io.*;
public class Next02 {
    public static void main (String[] args) throws Exception {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
        String str = " ";
        while (true) {
            System.out.println("Bitte Text eingeben: ");
            str = in.readLine();
            if (str == null) break;
            System.out.println("Verarbeitet wird: " + str);
        }
        System.out.println("Programmende");
    }
}
```

Die Lösung funktioniert gut, doch wird sie dadurch leicht unübersichtlich, weil nicht erkennbar ist, dass die Endlosschleife nur vorgetäuscht ist, weil es sehr wohl eine klar zu formulierende Bedingung für die Entscheidung über das Schleifenende gibt.

Der Quelltext soll das ausdrücken, was der Programmierer beabsichtigt. Dieses Ziel wird hier nicht erreicht.

### Programm *Next03*: Zweiter Lösungsvorschlag - Probelesen vor dem ersten Schleifendurchlauf

```
import java.io.*;
public class Next03 {
    public static void main (String[] args) throws Exception {
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
        System.out.println("Bitte Text eingeben: ");
        String str = in.readLine();
        while (str != null) {
            System.out.println("Verarbeitet wird: " + str);
            System.out.println("Bitte Text eingeben: ");
            str = in.readLine();
        }
        System.out.println("Programmende");
    }
}
```

Dieses Programm arbeitet richtig. Der Aufbau ist logisch und verständlich. Allerdings hat es den Nachteil, dass der Lesevorgang zweimal programmiert werden muss.

### Programm *Next04*: Dritter Lösungsvorschlag - Einsatz von Standardmethoden für Iteration

```
import java.util.Scanner;
public class Next04 {
    public static void main (String[] args) {
        Scanner cons = new Scanner(System.in);
        String ein = null;
        System.out.println("Bitte Text eingeben: ");
        while (cons.hasNext()) {
            ein = cons.next();
            if (ein.equals("ende")) break;
            System.out.println("Eingabe ist " + ein);
        }
        System.out.println("Programmende");
    }
}
```

Es gibt einige Java-Standardklassen, die enthalten für die Iteration besondere Methoden, die das "Probelesen" durchführen. In diesem Beispiel ist das die Methode *hasNext* der Klasse *Scanner*. Sie liefert als Ergebnis des Vorauslesens den Wert *true*, wenn der Eingabestrom noch weitere Daten enthält, oder *false*, wenn keine Daten mehr vorhanden sind.

**Programm Next05: Viertes Beispiel - Class *Iterator* für eine Collection benutzen**

```
import java.util.*;
public class Next05 {
    public static void main (String[] args) {
        // Erstellen einer Objektsammlung im Speicher
        ArrayList list = new ArrayList();
        list.add(new String("Erstes Objekt"));
        list.add(new String("Zweites Objekt"));
        list.add(new String("Drittes Objekt"));
        // Iterieren durch die Objektsammlung
        Iterator i = list.iterator();
        while(i.hasNext()) {
            System.out.println(i.next());
        }
    }
}
```

Die Standardbibliothek von Java enthält verschiedene Klassen für die Verwaltung von Objekten im Arbeitsspeicher. Dieses Beispiel benutzt hierfür die Klasse *java.util.ArrayList*. Dann wird eine Instanz der Klasse *Iterator* erzeugt. Und dort ist ein Standardalgorithmus für das Iterieren durch derartige Datensammlungen in den Methoden *hasNext()* und *next()* vorprogrammiert.

**Übung zum Programm Next05**

Wandeln Sie das Programm um (bei der Umwandlung mit J2SE 5.0 kommt eine Warnung, die jedoch ignoriert werden kann). Testen Sie das Programm. Ändern Sie es danach so ab, dass ein viertes Objekt hinzugefügt wird und danach das 2. Objekt (über den Index, mit der Methode *remove*) gelöscht wird.

**Hinweise für den Lehrenden:**

Die Klasse *ArrayList* ist Teil des Collection-Frameworks. Seit J2SE 5.0 sind die Collection als parametrisierbare Typen ("Generics") programmiert. Das Arbeiten mit Generics ist ein Thema für Fortgeschrittene und wird deshalb in diesem Buch nicht behandelt. Um die Warnung bei der Umwandlung zu eliminieren, ist das Programm wie folgt zu ändern:

**Programm Next06: J2SE 5.0 - ready**

```
import java.util.*;
public class Next06 {
    public static void main (String[] args) {

        // Erstellen einer Objektsammlung im Speicher
```

```
ArrayList<String> list = new ArrayList<String>();
list.add(new String("Erstes Objekt"));
list.add(new String("Zweites Objekt"));
list.add(new String("Drittes Objekt"));

// Iterieren durch die Objektsammlung
Iterator<String> i = list.iterator();
while(i.hasNext()) {
    System.out.println(i.next());
}
}
```

Die Klasse *StringTokenizer* bietet ebenfalls Methoden an, die "probeweise" lesen. Diese testen, ob weitere Token im String verfügbar sind und liefern *true* oder *false* als Ergebnis. Erst danach erfolgt dann mit *nextToken()* das echte Lesen.

### Programm *Next07*: Fünftes Beispiel - Class *StringTokenizer*

```
import java.util.*;
public class Next07 {
    public static void main (String[] args) {
        String s ="Dies ist nur ein Test";
        StringTokenizer st = new StringTokenizer(s);
        while (st.hasMoreTokens())
            System.out.println(st.nextToken());
    }
}
```

## 8.8 Stilfragen: Konventionen zum Programmierstil

In der Praxis sind immer wieder Änderungen und Anpassungsarbeiten an bestehender Software notwendig. Während der gesamten Software-Lebenszeit wird modifiziert, ergänzt oder korrigiert, z.B. um Fehler zu beseitigen, wegen geänderter Aufgabenstellung, aufgrund gesetzlicher Änderungen usw. Ein Indiz dafür: Das Einspielen von Updates und Servicepacks gehört zu den ständig wiederkehrenden Hauptaufgaben der Systemverwalter.

Deswegen ist es wichtig, übersichtliche und leicht verständliche Quellenprogramme zu schreiben.

Die nachfolgenden Standards entsprechen den Empfehlungen der Java-Entwickler. Diese sind ausführlich dokumentiert in den "Java Code Conventions" und können über die Adresse "<http://java.sun.com/docs/codeconv/>" kostenlos bezogen werden.

### 8.8.1 Empfehlungen für die Quelltextdatei

- Kommentarzeilen zu Beginn (Aufgabe der Klasse beschreiben, Version definieren). Weitere Kommentare immer dort, wo zusätzliche Erläuterungen erforderlich sind, weil der Quelltext allein nicht aussagefähig ist.
- Eine Zeile sollte nicht mehr als 80 Zeichen enthalten, damit die Bildschirmanzeige und Druckausgabe übersichtlich bleibt.
- Wenn eine Quellendatei mehrere Klassen enthält, so sollte die *public*-Class, die die *main*-Methode enthält, die erste Klasse in dieser Umwandlungseinheit sein.
- Zwei Leerzeilen zwischen den einzelnen Klassen einer Quelldatei; eine Leerzeile zwischen den Methoden einer Klasse
- Leerstellen (blanks) zwischen den einzelnen "Token" der Java-Sprache eingeben.
- Es gibt eine besondere Form für Kommentare - das sind die "Dokumentationskommentare". Sie werden begrenzt durch `/** ... /**` und können durch mitgelieferte Tools extrahiert und in einer HTML-Datei aufbereitet werden, ähnlich wie die API-Dokumentation. Weitere Informationen unter der Adresse:

<http://java.sun.com/products/jdk/javadoc/>

### 8.8.2 Empfehlungen zum Codierstil für Statements

- Generell gilt: Die Schreibweise muss konsistent, d.h. durchgehend gleich sein, z.B. sollen die Regeln zum Einrücken nicht variieren, sondern konsequent benutzt werden.
- Die Anweisungen sollen möglichst einfach sein und dem menschlichen Leser einen unmittelbaren Einblick in ihre Aufgabe und Wirkung geben. Dies gilt besonders für die Steuerbefehle. Das Zusammenspiel der einzelnen Konstrukte muss aus dem Quelltext erkennbar sein.
- Eine Zeile im Quelltext sollte nicht mehr als ein Statement enthalten. Beispiel:

```
a++;           // korrekt
x- -;         // korrekt
a++; a- -;    // Kein guter Stil!
```

- Bei Kontrollanweisungen für Schleifenbildung und Alternativen soll eingerückt werden, um die Struktur (z.B. einer Verschachtelung) optisch gut sichtbar zu machen.

Beispiele für IF:

```
if (condition) {
    statements;
}
```



Das *else* muss auf derselben Spalte stehen wie das dazu gehörende *if*.

Besonders wichtig ist das Einrücken bei geschachtelten Anweisungen:

```
if (condition) {
    if (condition)
        statements;
    else
        statements;
else
    statements;
}
```

- Jedes *switch*-Statement sollte ein *default*-Case haben. Außerdem sollte grundsätzlich das *break*-Schlüsselwort benutzt werden. Ausnahmen sind zu dokumentieren.
- Für die besprochenen Steueranweisungen *if*, *else*, *while*, *for* und *do* sollte immer ein Block codiert werden, auch wenn dieser aus nur einer Zeile besteht oder sogar leer ist. Beispiel:

```
while (condition) {
    statement;
}
```

- Java erlaubt an vielen Stellen eine Kurzschreibweise, alternativ zu der ausführlichen Codierung. Beispiel: `a = b = c`; Diese verkürzte Schreibweise bedeutet nicht, dass auch die Ausführung schneller ist. Aber sie erschwert dem menschlichen Leser das Verständnis für den Quelltext ("Programme werden häufiger gelesen als geschrieben"). Deswegen ist es häufig besser, die verständliche Langschreibweise zu wählen.
- Manche Programmierer neigen zu "eleganten" Formulierungen, wobei elegant dann leider nur eine Umschreibung ist für unverständliche (und oft fehleranfällige) Programmiertricks ist.

### 8.8.3 Empfehlungen für die Namensvergabe

- Die **Identifizier** (frei gewählte Namen für Klassen, Methoden usw.) sollen sprechend, also möglichst aussagefähig sein, und nicht aus einzelnen Zeichen bestehen.
- **Klassennamen** sollten aus einem Hauptwort bestehen und immer mit einem Großbuchstaben beginnen. Wenn mehrere Wörter zusammengesetzt werden, so beginnt jedes Wort mit einem Großbuchstaben, z.B. *BufferedInputStream*.
- **Methodennamen** sollten aus einem Verb bestehen und immer mit einem Kleinbuchstaben beginnen. Wenn mehrere Wörter zusammengesetzt werden, so beginnen die nachfolgenden Wörter mit einem Großbuchstaben, z.B. *getDatum*.

- **Feldnamen** sind oft Adjektive, sie beginnen mit einem Kleinbuchstaben. Bei zusammengesetzten Begriffen wird jedes nachfolgende Wort groß geschrieben z.B. *mußtBetrag*.
- **Konstanten** werden komplett groß geschrieben und die einzelnen Wörter durch einen Unterstrich getrennt, z.B. *MAX\_VALUE*.
- **Package-Namen** werden grundsätzlich klein geschrieben. Wenn sie global eindeutig sein müssen, so wird eine Anlehnung an die Internet-Domain-Namen empfohlen, zB. *com.ibm* oder *com.java.sun*.
- **Setter- und Getter-Methoden** sind für jedes private Attribut sinnvoll. Das sind Methoden, die das Lesen (GET) und das Schreiben (SET) dieser Variablen ermöglichen. Diese Methoden werden auch manchmal Accessoren (die Getter-Methoden) und Mutatoren (die Setter-Methoden) genannt. Für das Erstellen von Java-Beans sind sogar bestimmte Namensregeln für Setter- und Getter-Methoden obligatorisch: die Methodennamen beginnen mit dem Präfix *set* oder *get*, danach folgt der Name des Feldes, zB. *getDatum* oder *setAdresse*.
- Die ungarische Notation, bei der mit besonderen Zeichen der Datentyp oder die Lokation gekennzeichnet wird, ist zu vermeiden, weil sie zu unübersichtlichem Quelltext führt.

#### 8.8.4 Hinweise zum Testen

Testen ist eine undankbare Aufgabe, denn das Ziel besteht darin, Fehler in der eigenen Arbeit zu finden. Wahrscheinlich ist es für viele Programmierer einfacher, Fehler in fremden Programmen zu finden, als den Nachweis zu erbringen, dass ihre eigenen Programme fehlerhaft sind.

Es kann sinnvoll sein, Testfälle **vor** dem Codieren der Klasse zu erstellen, um zu überprüfen, ob das Klassendesign komplett ist.

# 9

## Softwaresysteme entwickeln (Projekte realisieren)

---

Die Beispiele in diesem Buch sind Miniprogramme, die ausschließlich darauf ausgerichtet sind, den Lernstoff so einfach wie möglich darzustellen. Sie können jeweils isoliert codiert, umgewandelt, getestet und genutzt werden, es gibt keine Abhängigkeiten zwischen den einzelnen Programmen. Damit sind sie auf keinen Fall Muster für reale Anwendungen - dazu haben wir zu stark vereinfacht. Beispielsweise enthalten die Programme keine formalen oder logischen Prüfungen und auch keine Fehlerbehandlung.

In der Praxis bestehen Projekte aus dem Zusammenspiel von vielen Einzelprogrammen, die jeweils um ein Vielfaches größer sind als jedes in diesem Buch besprochene Beispiel.

Folgende Anekdote soll dies unterstreichen: Der berühmte Edgar Dijkstra, einer der großen "Päpste" der Informatik, hielt vor einigen Jahrzehnten einen Vortrag, bei dem er erklärte: "Ich werde immer getadelt, weil meine Beispiele so klein sind, manchmal sind sie nur 5 Zeilen lang. Ich nehme den Tadel an und bringe Ihnen heute mal ein sehr komplexes Beispiel aus der Praxis". Und dann brachte er ein Programm, das war zwei DIN-A4-Seiten lang. Dazu gibt es in einem Gespräch mit Hasso Plattner, dem SAP-Gründer, folgenden Kommentar: "... hier zeigt sich das Hauptproblem der Informatik. Wir reden bei uns nicht über zwei Seiten, sondern über 40000 DIN-A4-Seiten, und das ist eine ganz andere Kategorie..."

Dieses Kapitel hat genau diese Problematik als Thema. Weil Softwareprojekte nicht isoliert von Einzelpersonen realisiert werden, sondern das Gemeinschaftswerk sind von vielen Spezialisten, die in einem Team zusammenarbeiten, hängt der Gesamterfolg eines Projekts nicht nur davon ab, dass funktionierende Einzelprogramme erstellt werden.

Mindestens ebenso wichtig ist es, dass sich diese sauber integrieren lassen zu einem Gesamtsystem.

Sie werden in diesem Kapitel

- eine Einführung bekommen in die grundlegenden Vorgehensmodelle, Prinzipien und Methoden der Anwendungsentwicklung;
- erfahren, warum gerade Java als Projektsprache für verteilte und internationalisierte Anwendungen besonders gut geeignet ist;
- lernen, welche Entwurfssprachen in der Design- und Realisierungsphase eingesetzt werden und welche Vor- und Nachteile damit verbunden sind.

## 9.1 Herausforderungen und Vorgehensweisen

Nicht wenige Projekte scheitern. Die Gründe dafür sind bekannt: mangelnde Vorbereitung, fehlende Systematik, unstrukturierte Vorgehensweise, geringe Transparenz. Vor allem aber: **die Systeme werden immer komplexer**. Das mag zum einen an der Aufgabenstellung liegen; häufig tragen aber unrealistische Zielvorstellungen dazu bei, dass Projekte gar nicht oder nur mit zusätzlichem, nicht geplantem Aufwand zu Ende geführt werden können. Projekte sind selten daran gescheitert, dass die Lösungen im ersten Anlauf unvollständig waren. Aber häufig blieben große Softwareprojekte unvollendet, weil gleich zu Beginn zu viel auf einmal versucht wurde.

Die erfolgreiche Realisierung von EDV-Projekten erfordert deshalb präzise Vorbereitung, großes Wissen, viel Erfahrung sowie systematisches Arbeiten der Teammitglieder bei der Realisierung. Nicht die Kreativität des Einzelnen steht im Vordergrund, sondern diszipliniertes Arbeiten, orientiert an festen Regeln und vorgegebenen Standards. Als Hinweis darauf, dass Software-Erstellung eine Ingenieursdisziplin ist, wird auch der Begriff "Software-Engineering" verwendet.

Wir wollen die Prinzipien und Methoden beschreiben, die für das Erstellen von Einzelprogrammen gelten ("**Programmieren im Kleinen**"), aber vor allem auch eine Einführung bieten in die Techniken beim Realisieren von größeren Projekten ("**Programmieren im Großen**").

### 9.1.1 Vorgehen bei Entwickeln einzelner Programm ("**Programmieren im Kleinen**")

Basis eines jeden Programmsystems sind immer die einzelnen Programme. Herausforderungen beim Kodieren dieser Programme bestehen im Umsetzen der Anwendungslogik in "Datenstrukturen und Algorithmen". Der Programmierer muss die richtigen Befehle und Datenbeschreibungen codieren und mit Kontrollstrukturen den Ablauf des Programms steuern. Dazu ist es in den meisten Fällen hilfreich, dass das Schreiben des Quelltexts erst erfolgt, wenn die Struktur und die Logik der Programmbausteine geklärt sind.

#### **Schrittweise Verfeinerung**

Eine typische Entwurfstechnik für Algorithmen ist die schrittweise Verfeinerung. Dabei wird zunächst ein grober Lösungsvorschlag mit abstrakten Operationen erstellt, der dann nach und nach abschnittsweise durch konkrete Operationen verfeinert und implementiert wird ("top-down-Entwurf"). Das folgende Beispiel soll dieses Vorgehen demonstrieren. **Die Aufgabe des Programms *TopDown01.java* ist es**, eine Zahl zu potenzieren. Die Eingabedaten werden vom Bildschirm gelesen und das Ergebnis am Bildschirm ausgegeben. Also muss das Programm folgende **Teilaufgaben** erfüllen:

- Daten einlesen und prüfen
- verarbeiten (potenzieren)
- ausgeben.

Wir zerlegen die Gesamtaufgabe in einzelne "Module", jede dieser Teilaufgaben ist eine eigene Methode. Die Implementierung startet zunächst nur mit der *main*-Methode, ergänzt um die Realisierung **nur eines** dieser Module (z.B. *potenzieren*).

### Programm TopDown01a: Erster Schritt - Methode *potenzieren()*

```
class TopDown01a {
    public static void main(String[] args) {
        int erg = potenzieren(5,3);
        System.out.println(erg);
    }
    static int potenzieren(int z1, int z2) {
        int erg = z1;
        if (z2 == 0) return(1);
        for (; z2>1; z2--) {
            erg = erg * z1;
        }
        return erg;
    }
}
```

Wenn dieses Rumpfprogramm funktioniert, wird schrittweise jeweils eine weitere Methode hinzugefügt und getestet, z.B. im zweiten Schritt die Methode *ausgabe()* und danach im dritten Schritt die Methode *eingabe()* ergänzt.

### Übungen zum Programm TopDown01a

Bitte versuchen Sie selbstständig, die Programme TopDown01b (ergänzt um die Methode *ausgabe*) und Topdown01c (ergänzt um die Methode *eingabe*) zu erstellen. Orientieren Sie sich dabei an dem folgenden Lösungsvorschlag für das endgültige Programm.

### Lösungsvorschlag: Das komplette Programm, erstellt in 4 Schritten (a-d)

```
import java.io.*;
class TopDown01d {
    public static void main(String[] args) {
        System.out.println("Bitte eine Zahl als Basis eingeben: ");
        int basis = eingabe();
        System.out.println("Bitte eine Zahl als Exponent eingeben: ");
        int exponent = eingabe();
        int erg = potenzieren(basis, exponent);
        ausgabe(erg);
    }
    static int eingabe() {
```

```
String str = null;
try {
    str = new DataInputStream(System.in).readLine();
}
catch (Exception e) {
    System.out.println(e);
}
return Integer.parseInt(str);
}
static int potenzieren(int z1, int z2) {
    int erg = z1;
    if (z2 == 0) return(1);
    for (; z2>1; z2--) {
        erg = erg * z1;
    }
    return erg;
}
static void ausgabe(int erg) {
    System.out.println("Das Ergebnis ist: " + erg);
}
}
```

Das fertige Programm produziert leider noch einen Umwandlungsfehler:

"TopDown01d.java uses or overrides a **deprecated API**."

Erläuterung: Als "deprecated" werden Sprachbestandteile gekennzeichnet, die veraltet sind und die in späteren Versionen der Java-Sprache entfallen werden. Deswegen sollten wir unser Programm noch einmal modifizieren, um es zukunftssicher zu machen. Dazu werden wir im Eingabemodul die veraltete *readLine*-Methode **komplett gegen eine neue Version austauschen**. Damit ist (hoffentlich) ein wichtiger Vorteil der Modularisierung eindrucksvoll demonstriert.

### Übung zum Programm TopDown01d

Tauschen Sie das veraltete Modul *eingabe* aus gegen folgende verbesserte Version.

#### Methode *eingabe()*: Neu programmiert

```
static int eingabe() {
    Scanner ein = new Scanner(System.in);
    int zahl = 0;
    try {
        zahl = ein.nextInt();
    }
    catch (Exception e) {
        System.out.println(e);
    }
}
```

```
    }  
    return zahl;  
}  
}
```

### Was sind "gute" Programme?

Natürlich ist das wichtigste Ziel bei der Entwicklung von Programmsystemen, dass die Programme richtig funktionieren, d.h. sie sollten fehlerfrei sein. Was macht darüber hinaus ein "gutes" Programm aus?

Hier einige Antworten:

- Gute Programme sollen benutzerfreundlich (einfach und robust) sein.
- Gute Programme sollen übersichtlich und für den Menschen leicht lesbar sein.
- Gute Programme sollen wiederverwendbar und portabel sein.
- Gute Programme sollen effizient sein (schnell, mit wenig Speicherbedarf).

Es gibt natürlich Situationen, wo die Effizienz eines Programms das wichtigste Kriterium ist ("real-time-Systeme"). In der Regel sind jedoch die anderen Ziele wichtiger: Die Benutzerakzeptanz ist gesichert, wenn die Programme einfach und robust sind, die Wartung der Programme wird erleichtert, wenn sie übersichtlich und leicht lesbar sind. Die zukünftige Gebrauchsfähigkeit der Programme wird erhöht, wenn die Programme wiederverwendbar sind und wenn sie auch auf anderen Hardware-Plattformen und Betriebssystemen lauffähig sind.

Zur Wartungsfreundlichkeit eines Programms gehört es, dass der Quellcode transparent und überschaubar codiert worden ist, Kommentare an den notwendigen Stellen enthält, die Namensvergabe plausibel erfolgt und dass Standards eingehalten worden sind, damit der Quellcode leicht analysiert werden kann.

Darüber hinaus gelten folgende Prinzipien:

### Prinzip des Information Hiding (Geheimnisprinzip)

- Die internen Abläufe bleiben dem Benutzer einer Klasse oder Methode verborgen. Variablen möglichst privat (bei Klassen) oder lokal (bei Methoden) deklarieren (siehe hierzu Kapitel 16). Globale Variable sollten vermieden werden.

### Prinzip der "Strukturierten Programmierung"

Die Forderung nach übersichtlichen Programmen beschreibt das wichtigste Ziel der "strukturierten Programmierung". Dabei werden mehrere, bereits bekannte Techniken miteinander verbunden:

- Prinzip der Modulbildung (abgeschlossene Programmblöcke, die eine funktionale Einheit bilden und deren Aufgabe mit möglichst einem Wort beschrieben werden kann).

- Einsatz einer begrenzten Anzahl von Steuerbefehlen (Sequenz, Alternative und Wiederholung).
- Vermeidung von Sprungbefehlen, die zu so genanntem "Spaghetticode" führen. In Java werden Sprünge im Programm realisiert durch die Schlüsselwörter *break*, *continue* und *return*. Es gibt zwar das reservierte Wort *goto*, dieses wird in Java aber nicht genutzt.
- Das Programm sollte von oben nach unten lesbar sein. Dabei sollte die statische Niederschrift einer Methode möglichst übereinstimmen mit dem dynamischen Ablauf des Programms.
- Die Strukturblocke haben nur einen Eingang (ist in Java auch nicht anders möglich). Sollte ein Block mehrere Ausgänge haben (*break*, *return*, *continue*), so ist dies entsprechend zu kennzeichnen.
- Eine besondere Möglichkeit der Ablaufsteuerung hat der Java-Programmierer durch spezielle Sprachmittel, um Methoden parallel ausführen zu lassen (Multi-Threading). Dies ist ein Thema für Fortgeschrittene.
- Der Programmierung im Sinne von "Kodieren" muss unbedingt eine detaillierte Entwurfstätigkeit vorausgehen. Dazu gehört eine exakte Klärung der Aufgabenstellung und ein detailliertes Entwerfen der Programmlogik, am besten mit Unterstützung von grafischen oder verbalen Hilfsmitteln wie Struktogramme, Ablaufpläne oder Entscheidungstabellen (Erläuterungen hierzu später in diesem Kapitel).

### Prinzip der Übersichtlichkeit

- Darunter versteht man den Einsatz von Standards für die Namensvergabe, das Einrücken, Einfügen von Kommentaren usw. Diese Regeln können auch die Art der Dokumentation betreffen, Vorschriften für das Arbeiten mit grafischen Modellbeschreibungen (UML) enthalten, die Verwendung von Design-Pattern betreffen oder aus allgemeinen Programmierregeln bestehen.
- Beispiele für Programmierregeln sind:
  - Benutzen Sie, wann immer möglich, die Klassen der Standardbibliotheken (z.B. müssen Queue- oder Stackklassen nicht mehr selbst codiert werden, sie werden bereits mitgeliefert).
  - Codieren Sie - wo immer Fehler denkbar sind - individuelles Exception-Handling (Behandlung von Ausnahmesituationen)
  - Benutzen Sie anstelle von festen Zahlenwerten z.B. für MWST-Sätze oder Arraygrößen besser Konstanten mit beschreibenden Namen.



## 9.1.2 Projekte: Programmieren im Großen

### 9.1.2.1 Tendenzen

Seit vielen Jahren sind ganz klar zwei Tendenzen in der Softwareentwicklung zu erkennen:

#### Modularisierung der Bauteile

- Das Gesamtsystem wird aufgeteilt in kleine, überschaubare Teilbereiche ("Klassen"). Und das Einzelprogramm wird aufgeteilt in Unterprogramme ("Methoden").

#### Spezialisierung der Mitarbeiter

- Die Zeit der Generalisten ist vorbei, jeder Teilbereich erfordert Spezialwissen. So gibt es Experten für Graphische Oberflächen, für Server- und Client-Anwendungen, für das Deployment, Security, Internet-Technologien, XML- oder Datenbank-Anwendungen.

Die einzelnen Teilaufgaben müssen (über definierte Schnittstellen) im Verbund funktionieren; die beteiligten Mitarbeiter müssen im Team (über definierte Schnittstellen) kommunizieren. Das heißt, das größte Problem sind die Schnittstellen.

#### Wie kann die Anzahl der Schnittstellen kontrolliert werden?

Die Komplexität eines Systems hängt wesentlich von der Anzahl der Schnittstellen ab. Bei einer netzwerkartigen Verbindung der einzelnen Systemkomponenten kann es schnell zu einer unübersehbaren Fülle von Schnittstellen kommen (kombinatorische Explosion).

#### Schnittstellen zwischen den Programmen

Versuchen Sie, die Breite der Schnittstellen bei der Software gering zu halten, indem Sie - wo immer möglich - Aufgaben in abgeschlossenen Einheiten ("Modulen") kapseln. Dadurch wird der erforderliche Datenaustausch minimiert.

#### Schnittstellen zwischen den Mitarbeitern

Die Schnittstellen zwischen den Mitarbeitern müssen standardisiert werden. Konventionen und Richtlinien helfen, Kommunikationsprobleme zu vermeiden. Wichtig ist der Einsatz von Design- und Dokumentationssprachen wie UML oder Entscheidungstabellen. Unterstützt werden die Teammitglieder durch maschinelle Entwicklungswerkzeuge ("tools").

### 9.1.2.2 Wo liegen die konkreten Herausforderungen?

#### Termine planen und einhalten

Der Zeitaufwand für die Projektrealisierung ist keineswegs durch die Addition des Aufwands pro Einzelprogramm zu ermitteln. Die Integration der Teilaufgaben und ihre Implementierung machen einen großen Teil des Gesamtaufwands aus.

### Kosten kontrollieren

Qualität kostet Geld. Geld ist knapp, also: Hier sind Zielkonflikte aufzulösen. Nicht alle Ziele sind gleichzeitig erreichbar. Häufig leidet die Qualität, weil die Zeit knapp wird. Oder die Kosten erhöhen sich bei höherem Zeitaufwand. Die Aufgabe der Verantwortlichen ist es, das Optimum zu finden.

### Qualität sichern

Zeitdruck darf nicht dazu führen, dass die Qualität leidet. Wenn die Qualität des neuen Systems unzureichend ist, leidet entweder die Akzeptanz, weil der Aufwand für den Nutzer der Programme größer oder unbefriedigend ist, oder es muss nachgebessert werden. Und beides wird wahrscheinlich teurer.

### Nicht sparen auf Kosten der Qualität

Softwaresysteme sind in der Regel viele Jahre im Einsatz. Noch heute werden Programme genutzt, die vor Jahrzehnten erstellt worden sind. Untersuchungen über Softwarekosten haben ergeben, dass die Kosten für die Wartung von Programmen die Erstellungskosten häufig übersteigen.

Nachlässigkeiten oder Fehler, die in der Entwicklungsphase gemacht werden, können später nur mit erheblichem Mehraufwand behoben werden. Es ist in jedem Fall kostengünstiger, Fehler zu vermeiden, als entstandene Fehler nachträglich zu suchen und zu korrigieren.

### Der Kampf mit der Komplexität

Das zentrale Problem der Softwareentwicklung ist der Kampf mit der Komplexität. Die neuen Anwendungen werden immer größer, immer anspruchsvoller (Grafik, Bedienerkomfort), internationaler und sind immer stärker vernetzt (Internet und Komponententechnologien). Sind solche Anwendungen, die durchaus aus einigen Millionen Codierzeilen bestehen können, noch überschaubar? Sind Wartungsarbeiten noch gefahrlos möglich? Jede Änderung, jeder einzelne Fehler kann eine millionenfache Auswirkung haben. Das Software-Engineering muss als ingenieurmäßiges Vorgehen zur Entwicklung von EDV-Systemen angesehen werden:

- Der Entwickler versucht, die Komplexität zu reduzieren durch "**Abstraktion**", d.h. durch Erkennen des Wesentlichen und Vernachlässigung von Nebensächlichkeiten.
- Die Vorgehensweise ist schrittweise, vom Allgemeinen zum Speziellen.
- Der Entwickler wird von Werkzeugen ("Tools") unterstützt, z.B. Integrierte Entwicklungsumgebungen (IDE) wie Eclipse
- Es gibt anerkannte Maßnahmen zur Qualitätssicherung (Dokumentation, Namensvergabe, Vorgehen beim Testen usw.).

### 9.1.3 Qualitätskriterien für Softwaresysteme

Die Qualität eines Softwaresystems hängt ab von der Qualität jedes einzelnen Moduls und von der Qualität der Beziehungen zwischen diesen Modulen. Für die Beurteilung der Software-Qualität gibt es verschiedene Ansätze. Ein Modell ist definiert nach ISO 9126. Wir wollen Software-Qualität unterscheiden

- nach der unmittelbaren und
- nach der zukünftigen Gebrauchsfähigkeit.

Das wichtigste Kriterium für die **unmittelbare Gebrauchsfähigkeit** ist die Korrektheit der Programme. Systeme, die die gestellten Aufgaben nicht oder nicht korrekt erfüllen, sind natürlich auf keinen Fall einsetzbar. Darüber hinaus gelten folgende Nebenziele:

#### Benutzerakzeptanz

- Diese hängt wesentlich ab von der Akzeptanz der Bedieneroberfläche. Wie ist diese Schnittstelle ("interface") zum "User" gestaltet? Unübersichtliche Dialoge verwirren den Benutzer, überfrachtete Fenster verlängern den Bildaufbau, und lange Antwortzeiten verärgern den Anwender.

#### Zuverlässigkeit

- Diese hängt zusammen mit der Robustheit der Programme. Sind sie so geschrieben, dass Bedienfehler keine unkontrollierbaren Reaktionen zur Folge haben? Was passiert, wenn es trotzdem zu einem Abbruch des Programms oder des Systems kommt? Gibt es ein Wiederanlaufverfahren?

#### Effizienz

- Hierunter versteht man das Ziel, die Ressourcen des Programms (Arbeitsspeicher, Prozessor) optimal einzusetzen und die Antwortzeit gering zu halten. Man bezeichnet dies als die "performance" des Programms.

Die **zukünftige Gebrauchsfähigkeit** beschreibt einerseits die Chancen für die Wiederverwendbarkeit und andererseits die Wartungsfreundlichkeit des Programms.

#### Wiederverwendbarkeit

- Je allgemeingültiger das Programm erstellt worden ist, umso besser sind die Möglichkeiten zur Wiederverwendung.

#### Portabilität

- Damit der zukünftige Einsatz der Software nicht von vornherein eingegrenzt wird, sollten die Programme keine spezifischen Eigenarten einer speziellen Plattform ausnutzen (z.B. sollte keine Programmiersprache gewählt werden, die nur auf bestimmten Systemen lauffähig ist).

## 9.2 Modelle zur Vorgehensweise

Software-Entwicklung erfolgt schrittweise, es beginnt mit der Produktdefinition, danach erfolgt die Realisierung und nach dem erfolgreichen Testen der Anwendung muss das Programmsystem an die Produktionsumgebung übergeben ("implementiert") werden. Seit Beginn der Programmierstätigkeit von einigen Jahrzehnten wurden immer wieder neue Prinzipien und Methoden entwickelt, die diesen Software-Entwicklungsprozess systematisieren und standardisieren sollen.

Einigkeit herrscht darüber, dass verschiedene Schritte ("Phasen") der Softwareerstellung unterschieden werden können.

### 9.2.1 Wasserfallmodell

Für die Planung der gesamten Tätigkeiten galt lange das so genannte **Wasserfall-Modell**. Dies beschreibt den Ablauf der Software-Entwicklung in verschiedenen Tätigkeitsblöcken, die in eigenständige Phasen eingeteilt werden:

- **Analyse** (Problem- und Bedarfsanalyse, Ermitteln der Schwachstellen im Ist-Zustand)
- **Design** (Entwurf eines Grobkonzept, Lösungsalternativen entwickeln, Datendesign, Funktionsdiagramme)
- **Detailentwurf** (Datenstrukturen festlegen, Module und ihre Schnittstellen beschreiben, Algorithmen entwerfen)
- **Realisierung** (Implementierung, Programmierung/Kodieren, Compilieren, Testen)
- **Einführung** (Integration, Auslieferung, Abnahme und Inbetriebnahme)

Die Phasen im Wasserfall-Modell laufen streng sequentiell ab. Jede Phase hat einen wohldefinierten Start- und Endpunkt. Es gibt keine parallel ablaufenden oder sich überlappende Tätigkeiten, denn jede Phase schließt mit einem eindeutig definierten Ergebnis ab, und die nachfolgende Stufe wird erst begonnen, wenn die vorherige Phase abgeschlossen ist. Die Ergebnisse fallen - wie bei einem Wasserfall - von einer Stufe zur nächsten.

**Bewertung dieses Modells:** Es besteht die Gefahr, dass bei einer zu starren Festlegung auf diese Vorgehensweise der Auftraggeber lediglich in der ersten Phase beteiligt ist, weil dort das "Pflichtenheft" verabschiedet wird und danach gibt es (theoretisch) kein Zurück mehr.

### 9.2.2 Spiralmodell

Mit Beginn der objektorientierten Programmierung wuchs die Erkenntnis, dass die Software-Entwicklung kein einmaliger sequentieller Prozess ist, der sich in abgeschlossenen Phasen vollzieht, sondern dass dies als ein evolutionärer Zyklus gesehen werden muss, bei dem die einzelnen Phasen wiederholt durchlaufen werden.

Jede Phase kann mehrfach durchlaufen werden, z.B. pro Teilprodukt. Dieses Modell wird als **Spiralmodell** bezeichnet. Es beschreibt den Entwicklungsprozess als einen iterativen Zyklus, bei dem sich das Projekt langsam den Zielen annähert.

Das Programmsystem wird inkrementell entwickelt. Man beginnt mit einer Basisimplementierung, eventuell sogar nur mit Prototypen, um die Ziele und Machbarkeit zu klären. Diese werden dann ständig verbessert und ergänzt. Dabei wiederholen sich die einzelnen Phasen (auch unter Einbeziehung der Nutzer) solange, bis alle Bedürfnisse erfüllt sind und das Produkt vom Auftraggeber akzeptiert und abgenommen wird.

Das Spiralmodell kann also durch **folgende Merkmale** beschrieben werden:

- Jede Spiraldrehung umfasst eine der 5 Phasen (steps) aus dem Wasserfallmodell.
- Jeder Step hat eine Rückkopplung auf den vorherigen Step, weil eine ständige Überprüfung der Zwischenprodukte stattfindet (review).
- Deswegen ist jede Entwicklung ein zyklischer (iterativer) Prozess.
- Sinnvoll ist vor Beginn eines neuen Steps ein Prototyping.
- Die Ergebnisse werden laufend validiert (überprüft), spätestens nach einer "Spiraldrehung".
- Die einzelnen Schritte sind nicht so streng getrennt wie beim Wasserfallmodell, bei dem die nächste Phase erst starten kann, wenn die vorherige komplett abgeschlossen ist. Beim Spiralmodell ist überlapptes Arbeiten erlaubt.

**Bewertung dieses Modells:** Der Vorteil dieser Vorgehensweise ist es, dass nicht versucht wird, am Anfang einen vollständigen Anforderungskatalog zu erstellen, der dann als unveränderlich gilt, sondern dass pragmatisch vorgegangen wird.

Natürlich besteht bei diesem Vorgehen die Gefahr, dass die Designphase zu kurz kommt. Man muss unbedingt vermeiden, dass codiert wird, ohne vorher die komplette Aufgabenstellung zu verstehen. Auf Systematik und Planung kann nicht verzichtet werden. Ausprobieren und solange basteln, bis die Programme "irgendwie" laufen, führen in jedem Fall zu erhöhten Gesamtkosten. Also: auch beim Spiralmodell liegt die Betonung bei der Planung. Dem Drang zum unmittelbaren Codieren darf nicht nachgegeben werden.

### 9.2.3 Lebenszyklus-Modell

Es gibt außerdem Modelle, die nicht nur die Entwicklungsphasen beschreiben, sondern ganz ausdrücklich auch die Wartungs- und Pflegearbeiten einbeziehen.

Ein solches Modell nennt man **Lebenszyklus-Modell** (life-cycle-model). Es beschreibt die Wartung und Pflege der Software nach der gleichen Vorgehensweise wie die Entwicklung selbst.

### 9.2.4 Resümee

In der Praxis sind die beschriebenen Prozess-Modelle in dieser reinen Form selten. Meistens findet man gemischte ("hybride") Systeme.

Als Kombination wäre folgende Vorgehensweise denkbar:

Man entwickelt einen Prototyp, der die typischen Eigenschaften im praktischen Einsatz demonstriert. Danach beschreiben Auftraggeber und Entwickler die Gesamtleistung nach dem Phasenmodell. Somit hat man die Basis für die Vertragsgestaltung, für die Termin- und Kostenplanung sowie für die Projektfortschrittskontrolle. Nach dem Testen der Einzelprogramme erfolgt die Systemintegration, das Zusammenmontieren der getesteten Komponenten. Danach ist es erforderlich, das neue System auf der Produktionsmaschine zu implementieren und im Zusammenspiel mit anderen Anwendungen einem Lasttest zu unterziehen. Diese Tätigkeiten können sehr aufwändig und auch komplex sein, man denke nur an den Deployvorgang für Web-server-Anwendungen.

Danach beginnt die letzte (und hoffentlich längste) Phase im *lifecycle* von Software-Produkten: die permanente Weiterentwicklung der Programme ("maintenance"). Änderungen sind z.B. notwendig, um Design- oder Programmierfehler zu bereinigen ("bug-fixing" durch "patches") oder um das System zu ergänzen, zu verbessern oder neuen Anforderungen anzupassen. Es gibt Schätzungen, dass die Entwicklung eines neuen Systems nur etwa ein Drittel der Gesamtkosten des Software-Produktes ausmachen. Die ständigen Updates und Releasewechsel sind die Ursachen für den größeren Kostenblock.

## 9.3 Prinzipien und Methoden der Anwendungsentwicklung

Das Erstellen von Software-Anwendungen kann verglichen werden mit dem Herstellen von Modulen als Industrieprodukte. Notwendig sind

- **Prinzipien**, die allgemein anerkannt sind und
- **Methoden** und Verfahren, nach denen gearbeitet wird.

### 9.3.1 Prinzipien der Anwendungsentwicklung

Prinzipien sind allgemein gültige Grundsätze, die die Vorgehensweise bei der Entwicklung theoretisch beschreiben:

#### Prinzip des Prototyping

- Die Anwender sollten grundsätzlich so weit wie möglich beteiligt werden am Entwurf des neuen Systems und beim Entwickeln der Anwendungsmodule. Dies geht am besten durch den Einsatz von Modellen (Prototypen). Ein Prototyp ist zwar ein ablauffähiges Muster des Zielsystems, erfüllt aber noch nicht die kompletten Anforderungen.

Der Vorteil dieser Vorgehensweise besteht darin, dass der Endbenutzer einen ersten Eindruck bekommt, wie das System funktionieren soll. Designfehler können u. U. rechtzeitig erkannt werden, und der Entwickler kann den Aufwand und die Machbarkeit abschätzen. Die Erstellung des endgültigen Zielsystems erfolgt dann durch evolutionäre Entwicklung.

### Prinzip der Abstraktion

- Dies ist eine allgemeine Problemlösetaktik. Unabhängig von Details werden durch Verallgemeinerungen zunächst die wichtigsten Aufgabenstellungen geklärt und konkrete Themen oder Spezialfälle erst später hinzugefügt. Es geht darum, durch Vereinfachung und Modellbildung ein grundsätzliches Verständnis zu bekommen, ohne sich in Details oder Sonderfällen zu verlieren. Besonders hilfreich können Modellbildungen sein, um dadurch grobe Lösungsansätze anschaulich zu machen.

### Prinzip der Modulbildung

- Das Gesamtsystem wird in überschaubare Einzelteile ("Module") zerlegt. Große, monolithische Programme sollen vermieden werden. Die einzelnen Programm-Module sollen aus übersichtlichen, klar voneinander abgegrenzten Bausteinen bestehen. Aus ihnen wird dann das Programmsystem zusammengesetzt. Für das Entwerfen und Codieren von Modulen gibt es Empfehlungen, die wir später auch detailliert besprechen werden.

### Prinzip der Lokalität

- Bei der Bildung von Modulen soll örtlich zusammengefasst werden, was zusammen gehört bzw. getrennt werden, wo keine Zusammengehörigkeit notwendig ist. Befehle sollen dort wirken, wo sie stehen, damit keine unerwünschten Seiteneffekte entstehen. Als Seiteneffekt wird eine Veränderung an einer nicht-lokalen Variablen durch ein gerufenes Unterprogramm bezeichnet.

### Prinzip der Uniformität

- Durch Standards in der Namensgebung und im äußeren Erscheinungsbild der Quelltexte, aber auch im Programmverhalten und bei den Benutzerschnittstellen soll es sowohl dem Nutzer der Programme als auch den für die Wartung zuständigen Kollegen so einfach wie möglich gemacht werden.

### Prinzip der integrierten Dokumentation

- Neben dem Schreiben von Kommentaren im Quelltext (dort, wo es notwendig ist), hilft der Einsatz von Planungshilfen wie Struktogramme oder UML (siehe Kapitel 12). Sowohl beim Designen des Gesamtkomplexes als auch bei der Entwicklung der Algorithmen kommen grafische Notationen oder umgangssprachliche Entwürfe (Pseudocode) zum Einsatz, bevor mit dem eigentlichen Codieren begonnen wird.

### 9.3.2 Methoden der Anwendungsentwicklung

Methoden definieren planmäßiges Vorgehen. Ihnen liegt ein Prinzip zugrunde. Während also die Prinzipien die Theorie definieren, beschreiben die Methoden die angewandte Praxis.

Für den fachlichen Entwurf der Programmsysteme gibt es unterschiedliche Ansätze, je nachdem, was als Basis der Analyse gesehen wird und welche Programmiersprachen zur Verfügung stehen:

#### Datenmodellierung

- Der Schwerpunkt liegt in der Analyse der Daten. Die Zusammenhänge werden grafisch dargestellt (zumeist in Entity-Relationship-Diagrammen oder Data-Dictionary-Einträgen). Wichtig ist, dass die Relationen zwischen den Daten beschrieben werden.
- Das wichtigste Ziel dieses Modells ist die Überleitung in ein relationales Datenbank-Design.

#### Prozessmodellierung

- Der Schwerpunkt liegt bei der Analyse der Abläufe und Informationsflüsse (Strukturierte Analyse). Daraus ergibt sich das Programm-Design. Das Ergebnis ist die Beschreibung von Funktionen. Dies geschieht u.a. durch Verwendung folgender Elemente: Datenflussdiagramme, Kontextdiagramme, Entscheidungstabellen.
- Es wird die Art der Aktionen und ihre Kommunikation untereinander, also der Datentransfer, zwischen den Prozessen untersucht und dargestellt, z.B. in "Zustandsübergangsdiagrammen".

#### Objektmodellierung

- Hierbei werden Daten und Funktionen als Einheit gesehen. Die Haupttätigkeit liegt bei der Beschreibung von Klassen und ihren Beziehungen. Konsequenz: Die Realisierung kann nur durch eine objektorientierte Programmiersprache erfolgen.
- Für die Beschreibung des fachlichen Entwurfs spielt bei objektorientierten (OO-) Systemen die Unified Modeling Language (UML-Notation) eine wichtige Rolle (siehe Kapitel 11). Durch diese Notation wird der Entwurf unterstützt, die Übersicht erleichtert und die Programmsicherheit erhöht.
- Die Vorgehensweise dabei ist wie folgt: nachdem Klassen und ihre Beziehungen definiert sind, erfolgt der Detailentwurf des Programms. Dies geschieht dadurch, dass die Verknüpfung der einzelnen Module (Klassen, Methoden) beschrieben wird und auch die Ausführungsreihenfolge, der Algorithmus, innerhalb eines Teilproblems festgelegt wird.



## 9.4 Java als Projektsprache

### 9.4.1 Java ist eine Sprache, die Abstraktion unterstützt

Java ist eine Sprache, die sehr stark abstrahiert. So kann ein Java-Programm völlig unabhängig von einer konkreten Hardware-Umgebung oder von einem Betriebssystem entwickelt und eingesetzt werden. Beim Codieren arbeitet der Programmierer mit "abstrakten Datentypen" (in der Objektorientierung werden diese jedoch als "Klassen" bezeichnet) und mit "abstrakten Operationen" (das sind die Methoden, von denen nur die Schnittstelle, nicht jedoch die Implementierung bekannt ist).

#### Abstraktion als allgemeines Mittel zur Bewältigung der Komplexität

Die Abstraktion ist ein allgemeines Prinzip zur Problembewältigung. Es spielt sowohl beim "Programmieren im Großen" als auch beim Realisieren von einzelnen Programmen eine entscheidende Rolle. Deshalb ist die Geschichte der Programmiersprachen auch eine Geschichte der fortschreitenden Abstraktion: von der hardware-nahen Assembler-Programmierung bis zu der objektorientierten Komponententechnologie.

Die folgende Tabelle beschreibt stichwortartig die Abstraktionsziele und jeweiligen Sprachmittel, durch die diese Ziele erreicht worden sind, beginnend mit den "Urzeiten" der elektronischen Datenverarbeitung in den 50er Jahren bis hin zu den neuesten Entwicklungen.

Abstraktionsziel:	Mittel der Realisierung:
einzelne Bits bezeichnen	Darstellung als Hexadezimalcode z.B. 7F
Codierung der Maschinenbefehle	Symbole für Operationen, z.B. add
Adressierung der Speicherplätze	Symbole für Speicherplätze (identifizieren)
Einzelne Maschinenbefehle	Zusammenfassung zu Ausdrücken
Befehlssequenzen zusammenfassen	Unterprogramme / Prozeduren
spezielle Berechnungsvorschriften	Funktionen mit Rückgabe und Parameter
Daten und Operationen kapseln	Abstrakte Datentypen (ADT) und Klassen
Ähnliche Operationen	Overloading, Override von Methoden
Ähnliche Objekte/Wiederverwenden	Vererbungsmechanismus
Hardware/Betriebssystem	Virtueller Rechner (z.B. in Java die JVM)
Adressraumgrenzen überwinden	Komponententechnologie (z.B. EJB in Java)

Abb. 9.1: Abstraktion in der Geschichte der Programmiersprachen

Und Java bildet die derzeitige Endstufe dieser Entwicklung. Insbesondere auch die beiden letzten Zeilen der obigen Tabelle treffen exakt auf Java zu: Die Sprache abstrahiert von einer konkreten Plattform und ist hervorragend geeignet, um Anwendungen zu schreiben, die über Adressraumgrenzen hinweg miteinander kommunizieren. Für diese fortgeschrittene Technologie gibt es in Java EJBs und Webservices.

### 9.4.2 Welche Gründe sprechen außerdem für Java?

Java ist eine objektorientierte Programmiersprache. Dadurch ergeben sich Vorteile, die sozusagen "eingebaut" sind in die Spezifikation der Sprache:

- Das **Prinzip der Abstraktion** wird dadurch eingehalten, dass zunächst mit allgemeinen Beschreibungen von Objekten begonnen wird. Es werden auf hohem Level Gemeinsamkeiten zwischen den Objekten gesucht und in Klassen beschrieben. Danach findet eine Spezialisierung statt. Weitere Klassen beschreiben Abweichungen oder zusätzlich Möglichkeiten. Die Klassen werden geordnet und miteinander verbunden
- Das **Prinzip der Modularität** wird dadurch gewahrt, dass jede Klasse auch gleichzeitig ein Modul ist. Sie beschreibt eine Gruppe von Objekten mit gleichen Eigenschaften und Fähigkeiten. Innerhalb der Klasse entstehen überschaubare, kleine Einheiten durch das Codieren von Methoden und Blöcken. Im Kapitel 12 wird ausführlich auf die Vorgehensweise bei der Modulbildung (Beschreibung von Klassen) eingegangen.
- Das **Prinzip der Lokalität** ist ebenfalls ein in alle OO-Sprachen eingebautes Prinzip. Die Module (Klassen) und auch die Methoden der Klassen fassen zusammen, was zusammengehört, d.h. was allen beschriebenen Objekten gemeinsam ist.
- Das **Geheimnisprinzip** wird unterstützt durch das "Verstecken" von Algorithmen und Variablen durch Klassenbildung mit **privaten** Elementen. Die Daten einer Klasse sind nur ansprechbar über Methoden dieser Klasse, auf keinen Fall sind sie von außen veränderbar.
- Java ist eine sehr einfache Programmiersprache (sie spezifiziert lediglich etwa 50 Schlüsselwörter). **Allerdings ist Java mehr als eine Programmiersprache, sie ist eine umfangreiche, komplexe Technologie.** Sie bietet eine komplette Infrastruktur, einsetzbar in allen Bereichen der IT- und Kommunikationsbranche. Im Vergleich zur Konkurrenztechnologie *.net* von Microsoft, die mehr als ein Dutzend Programmiersprachen integriert, hat die Java-Technologie außerdem den Vorteil, dass Java durchgängig die einzige Programmiersprache ist - verwendbar für alle Aufgabenstellungen: auf Client- **und** Serversystemen, für Spiele- oder Unternehmensanwendungen, für Handys und PDAs, für grafische Oberflächen und für Datenbank- und XML-Anwendungen. Diese Einheitlichkeit vereinfacht natürlich nicht nur die Ausbildung der Mitarbeiter, sondern vor allem auch die Wartung der Programme.

- Im Bereich der **Internet-Anwendung** und der **Internationalisierung** von Applikationen bietet Java beste Voraussetzungen: Die Programme sind universell einsetzbar: durch Standardisierung der eingebauten Datentypen, durch die Verwendung des Unicode und vor allem durch die Möglichkeiten der Lokalisierung, d.h. durch die Anpassung an regionale oder länderspezifische Besonderheiten. **Beispiel für englische und deutsche Währungsaufbereitung:**

```
import java.util.*;
import java.text.*;
class Locale01 {
    public static void main(String[] args) {
        // Default-Locale (wenn nicht deutsch, dann bitte angegeben)
        NumberFormat nf1 = NumberFormat.getInstance();
        System.out.println(nf1.format(1245.23));
        // Englisches Format
        NumberFormat nf2 = NumberFormat.getInstance(Locale.ENGLISH);
        System.out.println(nf2.format(1245.23));
    }
}
```

- Java Programme sind sicher durch vielfältige eingebaute **Security-Mechanismen**.

In der Praxis haben sich im Bereich der Anwendungsentwicklung einige Vorgehensweisen entwickelt, die gerade auch **für Java typisch** sind:

- Es wird sehr stark mit **Modellbildung**, z.B. zum Erkennen von Design-Fehlen, gearbeitet ("Prototyping").
- Es wird häufig auf **Muster** zurückgegriffen, die sich für ähnliche Lösungen bewährt haben ("Pattern"), z.B. Model-View-Control-Designpattern. Das Thema "Codierpattern" haben wir im Abschnitt 8.7 "Lösungsmuster für Schleifen" bereits behandelt, das Thema "Designpattern" werden wir im Kapitel 12 besprechen.

### 9.5 Entwurfssprachen

Bei OO-Systemen wird ein Programmsystem nicht primär als eine Anzahl von Funktionen gesehen, sondern als eine Ansammlung von kooperierenden Objekten (Modulen oder konkreter: Klassen) betrachtet. Dabei kann jedes Objekt mit allen anderen kommunizieren durch das Senden von Nachrichten. Die Aufgabe des Systementwicklers besteht darin, durch geeignete Design-Methoden die Klassen zu definieren und ihre Verbindungen festzulegen. Für diese Tätigkeit stehen ihm Entwurfssprachen zur Verfügung, die wir im Kapitel 12 besprechen werden (z.B. UML).

Für den Entwurf der Algorithmen, also für die Beschreibung der Abläufe innerhalb von Methoden und von Programmblöcken, gibt es ebenfalls diverse, programmiersprachen-unabhängige Hilfsmittel: z.B. Pseudocode oder Struktogramme. Diese

Hilfsmittel werden verwendet, um in einer frühen Entwicklungsphase Struktur und Ablauffolge der Einzelbefehle zu entwerfen und zu dokumentieren.

Damit bilden diese Darstellungen, häufig in Form einer grafischen Notation, die Grundlage für die eigentliche Codierung, d.h. sie werden anschließend in den Quelltext der tatsächlichen Programmiersprache umgesetzt.

Wir werden jetzt einige dieser Planungs- und Dokumentationshilfen vorstellen.

### 9.5.1 Programmablaufplan

Der Programmablaufplan (PAP) ist eine grafische Darstellung eines Lösungsweges (Algorithmus). Die Symbole des PAP sind genormt nach DIN 66001.

Die Sinnbilder des PAP zeigen

- die **Funktion** des Arbeitsschrittes (= durch die Form des Symbols = Was?)
- die **Reihenfolge** der Operationen (= durch die Ablauflinien = Wann?).

Die Beschriftung der Sinnbilder beschreibt die konkrete Operation (allerdings sind die Texte der Innenbeschriftung nicht genormt).

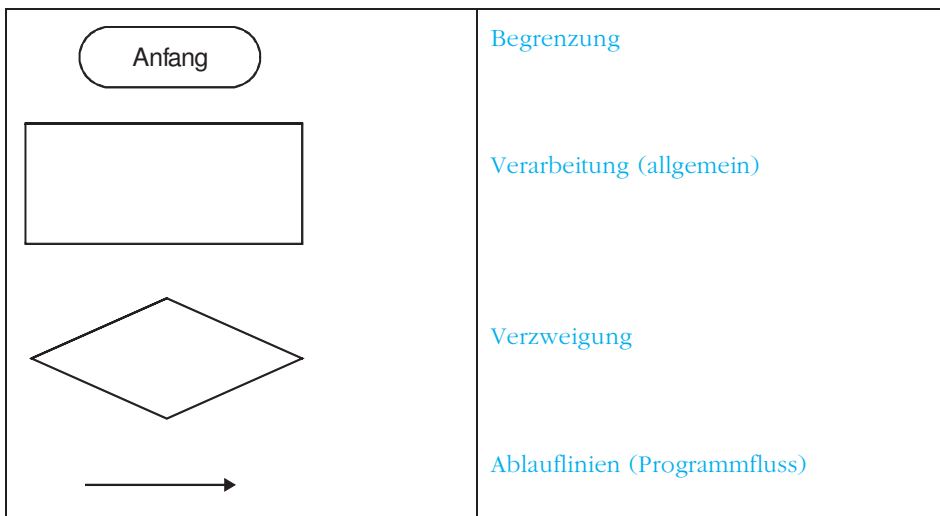


Abb. 9.2: Einige Sinnbilder nach DIN 66001

Ein Programmablaufplan (Flussdiagramm) stellt die Verarbeitungsfolgen in einem Programm dar. Die Verbindungslinien zeigen dabei die Reihenfolge der Verarbeitung auf. Daten (Variablendeklarationen) werden nicht dargestellt.

**Bewertung dieser Notation:** Der PAP erlauben zwar Konstruktionen nach den Konventionen der Strukturierten Programmierung, schließt aber andere Formen (wie GO TO rückwärts) nicht aus, d.h. der PAP lässt bei der Darstellung der Ablaufsteuerung große Freiheit und vertraut auf die Disziplin des Entwicklers.

## 9.5.2 Struktogramme

Struktogramme sind ineinander verschachtelte Rechtecke. In DIN 66261 sind die Sinnbilder und deren Anwendung genormt. Sie wurden von Nassi und Shneidermann entworfen und werden deshalb nach ihren Vätern auch "Nassi-Shneidermann-Diagramme" genannt. Das Ziel war der Ersatz der Programmablaufpläne, und zwar so, dass die Empfehlungen der "Strukturierten Programmierung" nicht nur unterstützt, sondern konsequent erzwungen werden.

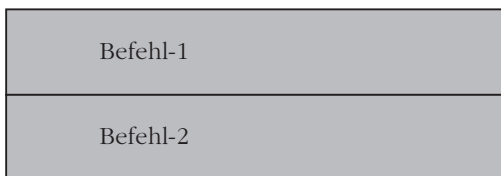
Dabei sind vor allem folgende Gesichtspunkte wichtig:

- Es ist mit den Struktogrammen keine Darstellung von Programmsprüngen möglich. Charakteristisch ist das völlige Fehlen von Verbindungslinien (Pfeilen) bei der Dokumentation von Algorithmen. Die Ablauflogik ergibt sich aus der Form der Symbole, Sprünge sind nicht darstellbar.
- Die Struktogramme werden immer ausschließlich von oben nach unten gelesen.
- Die Diagramme werden auch eingesetzt als grafisches Entwurfshilfsmittel für gut strukturierte Programme. Deswegen darf jedes Element nur einen Eingang und einen Ausgang haben.

**Bewertung dieser Notation:** Die Nassi-Shneidermann-Diagramme haben in der Praxis wenig Bedeutung. Sie eignen sich aber für den Programmieranfänger sehr gut, um die Denkweisen der "Strukturierten Programmierung" zu trainieren.

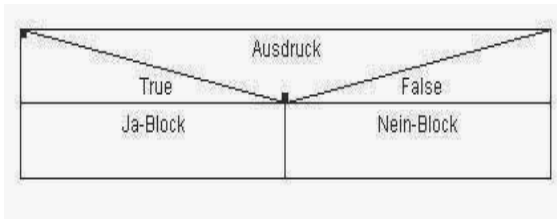
Für die Designphase in der praktischen Projektarbeit sind sie nicht so gut geeignet. Die Erstellung ist unhandlich, und Modifikationen (Änderungen, Löschungen oder Hinzufügen einzelner Schritte) sind oft kaum möglich, ohne dass die Grafik neu erstellt werden muss. Auf jeden Fall benötigt man die Unterstützung entsprechender Werkzeuge ("tools"), um mit den Struktogrammen praktisch zu arbeiten.

Die wichtigsten Symbole sind:



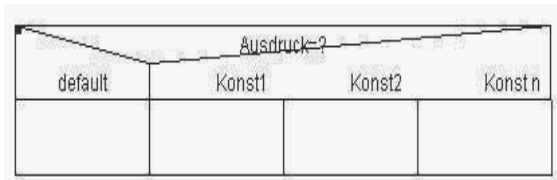
### **Befehl/Funktion:**

Lineare Abfolgen werden durch lückenloses Untereinandersetzen von Rechtecken ausgedrückt



### If-Befehl (Entscheidung/ Verzweigung)

Dieses Symbol zeigt, welcher Zweig einer Alternative ausgeführt wird. Üblicherweise steht der Ja-Fall auf der linken Seite



### Switch-Befehl (Mehrfach- Alternative)

Fall-Unterscheidung



### While-Schleife

abweisende Schleife: Der Block wird nur durchlaufen, wenn die Bedingung erfüllt ist



### Do-Schleife

nicht-abweisende Schleife: Der Block wird mind. einmal durchlaufen, erst danach wird geprüft, ob die Bedingung erfüllt ist u. evtl. wiederholt wird



### For-Schleife

Zählschleife; im Kopf stehen die Initialisierung, die Abbruchbedingung und die Inkrementierung

Abb.9.3: Einige Sinnbilder nach DIN 66261 (Struktogramme)

Das nachfolgende Beispiel zeigt eine Lösungsbeschreibung für das Errechnen der Fakultät, also für die Ermittlung eines Produkts aus natürlichen Zahlen, z.B. ist die Fakultät von 4 gleich  $1 * 2 * 3 * 4$  (d.h. also 24).

### Struktogramm: Errechnen der Fakultät von n

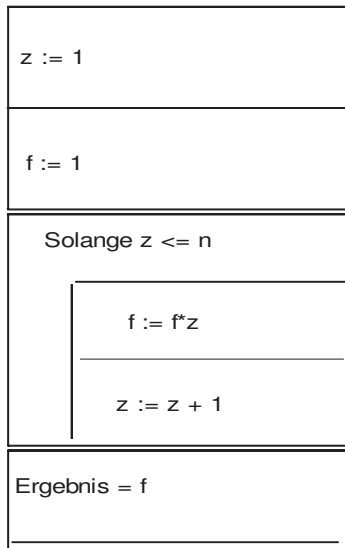


Abb. 9.4: Darstellung des Algorithmus für das Errechnen der Fakultät

**Übung:** Bitte codieren Sie die Aufgabe, die in der Abb. 9.4 als Struktogramm beschrieben ist, in Java.

### Lösungsvorschlag für n = 4:

```
public class Fakultaet {
    public static void main(String[] args) {
        int z = 1;
        int f = 1;

        while (z <= 4) {
            f = f * z;
            z++;
        }
        System.out.println(f);
    }
}
```

### 9.5.3 Pseudocode

Eine verbale, keine grafische Entwurfssprache. Mit umgangssprachlichen Formulierungen werden Daten und Kontrollstrukturen beschrieben. Für jede der Grundstrukturen der "Strukturierten Programmierung" gibt es in dieser halb formalisierten Planungssprache einen Ausdruck.

Der Pseudocode besteht aus:

- Schlüsselwörtern zur Ablaufsteuerung (IF, CASE, WHILE, DO, FOR)
- Texten in natürlicher Sprache für die Problemlösung

**Bewertung:** Der Pseudocode ist leicht erstellt, per Hand oder mit jedem beliebigen Texteditor. Er ist leicht erlernbar, verständlich und sehr flexibel. Eine Modifikation ist ohne Probleme möglich. Der Nachteil ist, dass ein Zwang zu einer disziplinierten Vorgehensweise aufgrund der geringen Normierung fehlt.

#### Übung

Der nachfolgend beschriebene Algorithmus soll ermitteln, wieviel Spinnen und wieviel Käfer sich in einer Schachtel befinden können, wenn insgesamt 64 Beine in der Schachtel sind und wenn Spinnen 8 Beine und Käfer 4 Beine haben. Wenn mehrere Möglichkeiten zutreffen können, soll das Programm alle möglichen Kombinationen ausgeben.

#### Lösungsbeschreibung im Pseudocode

For-Schleife: (für alle Spinnen, beginnend mit 1, bis maximal 8)  
 For-Schleife (für alle Käfer, beginnend mit 1, bis maximal 16)  
 überprüfen, ob die Summe der Beine = 64 ist

**Übung:** Bitte codieren Sie den oben als Pseudocode beschriebenen Algorithmus in Java.

#### Lösungsvorschlag:

```
public class Kaefer01 {
    public static void main(String[] args) throws Exception {
        for (int spinne=1; spinne<10; spinne++)
            for (int kaefer=1; kaefer<15; kaefer++)
                if ((spinne * 8) + (kaefer * 4) == 64)
                    System.out.printf("Kaefer = %d, Spinne =%d \n",
                                      kaefer, spinne);
    }
}
```



### 9.5.4 Entscheidungstabellen

Eine Entscheidungstabelle (ET) ist eine Matrix aus WENN-DANN-Beziehungen. Sie ist gut geeignet, wenn mehrere Aktionen, die von Bedingungen abhängen, analysiert, dargestellt und auf Vollständigkeit überprüft werden müssen.

Die Entscheidungsprozeduren werden in tabellarischer Form dargestellt. In der ersten Spalte der Tabelle werden zunächst alle **Bedingungen** aufgeführt, mit allen Kombinationen, die möglich sind. Der untere Teil, der **Aktionsteil**, enthält die Aufzählung aller Maßnahmen, die getroffen werden, wenn alle Bedingungen einer Spalte zutreffen. In den weiteren Spalten werden dann alle Regeln aufgeführt.

	Regel-1	Regel-2	Regel ...	Regel-n
Bedingung-1	J	J	...	N
Bedingung-2	J	N	...	N
Bedingung-3	J	N	...	N
Aktion-1	X	X		-
Aktion-2	-	X		-
Aktion-3	-	-		X

Abb.9.5: Aufbau einer Entscheidungstabelle

Die Bedingungsanzeiger beschreiben die jeweiligen Regeln, die dann spaltenweise zusammengefasst werden. Jede Regel repräsentiert eine Entscheidung, die zu einer oder mehreren Aktionen führt. Die einzelnen Bedingungen einer Regel sind logisch durch UND verbunden, die verschiedenen Regeln sind logisch durch ODER verbunden (oder anders gesagt: von oben nach unten gilt in einer ET die UND-Verknüpfung, von links nach rechts gilt die ODER-Verbindung). Dies demonstriert die folgende ET für den "Algorithmus" vor einer Verkehrsampel.

Ampel = rot	J	J	N	N
Ampel = gelb	N	J	J	N
Ampel = grün	N	N	N	J
Stoppen	X	-	X	-
Anfahren	-	X	-	-
Durchfahren	-	-	-	X

Abb.9.6: Entscheidungstabelle für den Autofahrer an einer Verkehrsampel

### Übung zum Thema Entscheidungstabelle: Bestellabwicklung

Die Bestellung eines Kunden wird maschinell überprüft. Wenn der Kunde kreditwürdig ist und wenn er nicht weniger als die Mindestmenge bestellt, so wird der Auftrag akzeptiert, andernfalls wird er zurückgewiesen. Zur Auslieferung muss der Lagerbestand abgefragt werden. Wenn dieser ausreicht, werden entweder die Versandpapiere geschrieben oder der Kunde bekommt eine Auftragsbestätigung,

Kunde kreditwürdig	J	J	-
Bestellung > Mindestmenge	J	J	-
Bestellung >= Lagerbestand	J	N	-
Versandpapiere schreiben	X	-	-
Auftragsbestätigung schreiben	-	X	-
Auftrag ablehnen	-	-	x

Abb. 9.7: Entscheidungstabelle für ein Programm zur Bestellabwicklung

### Bewertung

Die vollständige Form einer Entscheidungstabelle ist ein gutes Hilfsmittel, um alle möglichen Situationen auf ihre Konsequenzen zu prüfen. Sie kann mit formalen Methoden auf Vollständigkeit, Redundanz und Widerspruch überprüft werden.

Allerdings ist die Anzahl der möglichen Regeln sehr hoch, wenn es viele Bedingungen gibt:

Anzahl Kombinationen =  $2^{**}$  Anzahl Bedingungen (= kombinatorische Explosion).

So gibt es bei zwei Bedingungen maximal 4 Kombinationen, also 4 Regeln. Bei 5 Bedingungen sind es bereits  $2^5$ , also 32 Regeln usw. Deswegen gibt es so genannte "Dont-care"-Situationen, z.B. in der Regel 3 der obigen Entscheidungstabelle. Dadurch wird die Anzahl der dargestellten Regeln reduziert.

Ein weiterer **Vorteil** der Darstellung von komplexen Entscheidungssituationen mit Hilfe von Entscheidungstabellen besteht darin, dass die Umsetzung einer solchen Tabelle in Programmcode sehr einfach, evtl. auch maschinell durch ET-Generatoren, möglich ist. Häufig ist eine Codierung mit Fallunterscheidung (SWITCH...CASE) möglich, andernfalls führt jede Regel zu einem ELSE-Zweig.

Der **Nachteil** ist, dass Entscheidungstabellen lediglich für Teilprobleme eines Programms geeignet sind, nämlich nur für Alternativen und nicht z.B. für die Analyse und Darstellung von Schleifen.

## 9.6 Komplette Beispiel

### Aufgabenstellung: Kleinste von drei Zahlen ermitteln

Von den drei Zahlen  $z_1$ ,  $z_2$ ,  $z_3$  ist die kleinste auszuwählen. Bei Gleichheit entscheidet die höhere Priorität:  $z_1$  hat die höchste und  $z_3$  die niedrigste Priorität. Ist  $z_1$  ausgewählt, erfolgt die Verarbeitung in A, bei  $z_2$  in B und bei  $z_3$  in C.

#### Übung:

- Wie wird diese Übung im **Pseudocode** beschrieben?
- Wie sieht das **Struktogramm** aus?
- Wie sieht der **Programmablauf** aus?
- Wie sieht die **Entscheidungstabelle** aus?
- Wie würde man die Entscheidungstabelle in **Java** codieren?

#### Lösungsvorschlag für Pseudocode

```

if (z1 nicht größer z2) AND (z1 nicht größer z3)
    Ausführung A
else
    if (z2 nicht größer z3)
        Ausführung B
    else
        Ausführung C
    
```

#### Lösungsvorschlag für Entscheidungstabelle

a) Vollständige Entscheidungstabelle

$z_1 \leq z_2$	J	J	J	J	N	N	N	N
$z_1 \leq z_3$	J	J	N	N	J	J	N	N
$z_2 \leq z_3$	J	N	J	N	J	N	J	N
Block A	X	X	-			-		
Block B			-		X	-	X	
Block C			-	X		-		X

Abb. 9.8: Vollständige ET zur Ermittlung der kleinsten Zahl

b) Entscheidungstabelle mit "dont care"

<b>z1 &lt;= z2</b>	<b>J</b>	<b>N</b>	-
<b>z1 &lt;= z3</b>	<b>J</b>	-	-
<b>z2 &lt;= z3</b>	-	<b>J</b>	-
<b>Block A</b>	<b>X</b>		
<b>Block B</b>		<b>X</b>	
<b>Block C</b>			<b>X</b>

Abb. 9.9: ET zur Ermittlung der kleinsten Zahl (verkürzt)

Lösungsvorschlag für Struktogramm

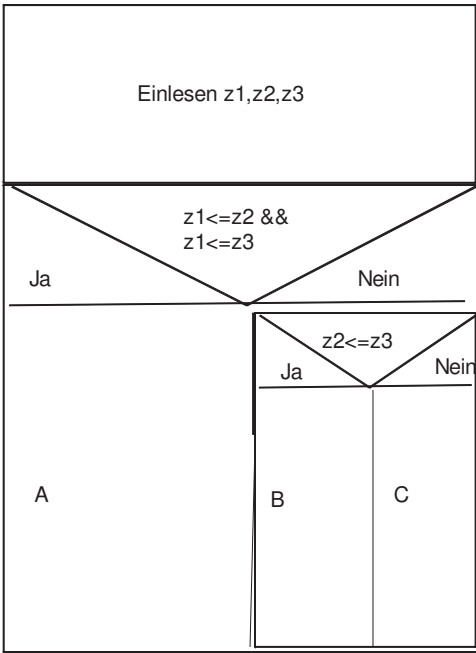


Abb. 9.10: Nassi-Shneidermann-Diagramm zur Ermittlung der kleinsten Zahl

### Lösungsvorschlag für Programmablaufplan

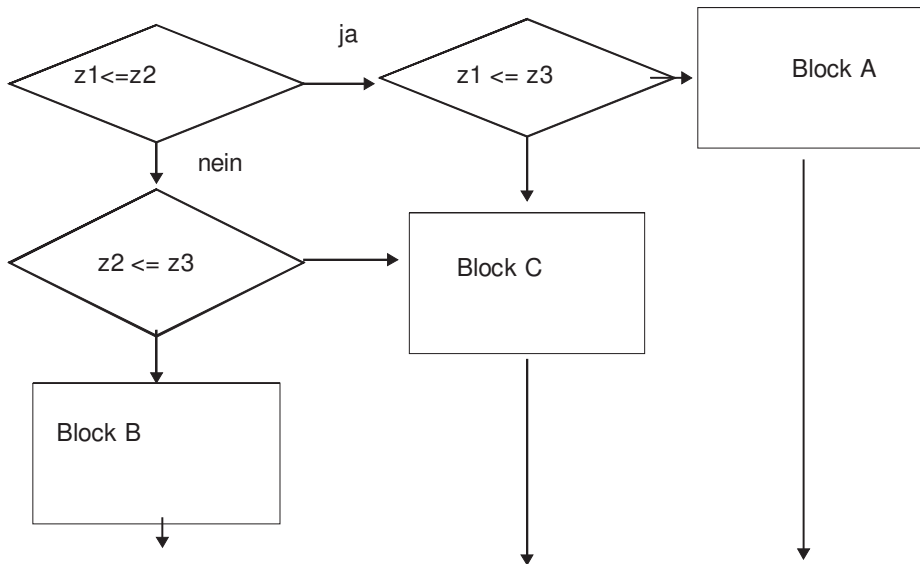


Abb.9.11: PAP zur Ermittlung der kleinsten Zahl

### Codierung in Java: Ermittlung der kleinsten Zahl von z1,z2 und z3

```
public class KleinsteZahl {
    public static void main(String[] args) {
        int z1, z2, z3;
        z1 = 20;
        z2 = 11;
        z3 = 10;

        if (z1 <= z2 && z1 <= z3)
            System.out.println("Block-A wird ausgeführt");
        else
            if (z2 <= z3)
                System.out.println("Block-B wird ausgeführt");
            else
                System.out.println("Block-C wird ausgeführt");
    }
}
```

---

# 10

## Methoden erklären, implementieren und benutzen

---

In Java kann ein Anweisungsblock einen Namen bekommen, der dann von anderen Anweisungen zur Ausführung aufgerufen wird. Dadurch entsteht eine Methode. Methoden werden in anderen Programmiersprachen auch als Unterprogramme, Prozeduren, Subroutinen oder Funktionen bezeichnet.

Sie lernen in diesem Kapitel,

- wie eine Methode deklariert wird,
- was man unter der Signatur einer Methode versteht,
- was formale und aktuelle Parameter sind und in welcher Form die Argumente übergeben werden,
- wieso die Übergabe bei Referenzvariablen ganz andere Konsequenzen hat als die Übergabe von einfachen Variablen,
- was bei der Rückgabe eines Ergebniswerts zu beachten ist und
- warum lokale Variablen ganz anders behandelt werden als Membervariablen.

Eine Möglichkeit, zwischen unterschiedlichen Arten von Methoden zu differenzieren, ist die Einteilung in Klassen-Methoden und Instanz-Methoden:

- **Instanz-Methoden** operieren auf Instanzen. Der Aufruf dieser Methoden setzt voraus, dass von ihrer Klasse ein Objekt erzeugt worden ist. Der Aufruf dieser Instanz-Methoden aus anderen Klassen heraus ist nur mit Hilfe des Objekts, mit dem sie arbeiten sollen, möglich.
- **Klassen-Methoden** dagegen operieren unabhängig von einer Instanz. Der Zugriff aus anderen Klassen erfolgt mit dem Klassennamen als Qualifizierer und nicht mit einem Instanznamen. Ein Nachteil dieser Klassen-Methoden ist, dass sie nicht auf die Instanzvariablen zugreifen können und auch keine Instanz-Methoden aufrufen können.

Obwohl die Klassen-Methoden eher einem konventionellen Programmieransatz und nicht dem objektorientierten Programmierstil entsprechen, werden Sie in diesem Kapitel häufig mit Klassenmethoden arbeiten. Das hat für den Lernenden den Vorteil, dass die Techniken der Methodenerstellung, des Methodenaufrufs und der Parameterübergabe erlernt werden können, ohne dass weitere Klassenbeschreibungen vorliegen müssen. Im folgenden Kapitel 11 wird dann, aufbauend auf den Erkenntnissen, die Sie in diesem Kapitel gewonnen haben, das Arbeiten mit Klassen und mit den dazugehörigen Instanz-Methoden ausführlich behandelt.

## 10.1 Was sind Methoden?

Eine Methode ist eine Befehlsgruppe (ein Programmblock), die einen Namen hat. Über diesen Namen wird sie zur Ausführung aufgerufen. Nach der Ausführung kehrt die Programmsteuerung wieder zurück an die Stelle des Aufrufs.

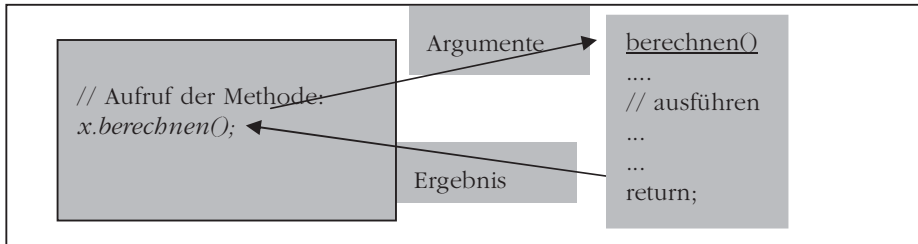


Abb. 10.1: Wie erfolgt die Programmsteuerung bei einem Methodenaufruf?

Bei dem Aufruf können Werte an die Methode übergeben werden. Mit diesen Daten kann innerhalb der Methode gearbeitet werden. Nach Ausführung der Anweisungen kann wahlweise auch ein Ergebnis an den Aufrufer zurück geliefert werden.

Eine Methode besteht aus einem Methodenkopf und einem Methodenblock:

```
methodenkopf {
    Methodenblock
}
```

### 10.1.1 Methodenkopf (header) und Methodenblock (body)

Im **Methodenkopf** stehen die Angaben zur Deklaration der Methode. Zur Deklaration gehört es, dass der Name festgelegt wird, die Eingangsdaten ("Parameter") beschrieben werden und der Datentyp des Ergebnisses festgelegt wird. Die Parameter werden unmittelbar hinter dem Namen der Methode, eingefasst in runden Klammern (...), als Liste von Deklarationen beschrieben.

In geschweiften Klammern eingefasst steht der **Methodenblock** (Rumpf). Er enthält die Implementierung der Methode. Dort stehen die Anweisungen. Jedes ausführbare Java-Programm enthält mindestens eine Methode, nämlich die Methode mit dem Namen *main*.

#### Programm *Methode01*: Die wichtigste Methode ist *main()*

```
public class Methode01 {
    public static void main (String[] args) {           // Kopf
        System.out.println(args[0]);                   // Block
    }
}
```

In der **ersten Zeile** wird die Klasse definiert. Ist die Klasse ein ausführbares Programm, so ist der Klassenname mit dem Programmnamen und auch mit dem Namen der Quelltextdatei identisch.

Die **zweite Zeile** des Programms enthält den **Kopf der Methode** (die "Deklaration"). Dort sind festgelegt:

- der Name der Methode (hier: *main*);
- Name und Typ der Eingangsdaten (Argumente, Parameter). Diese Methode erwartet Daten vom Datentyp *String* und speichert sie unter dem Identifier *args*;
- der Datentyp des Ergebnisses (hier: *void*, engl. nichts, nicht vorhanden);
- durch das Schlüsselwort *static*, dass es sich um eine Klassenmethode handelt, die aufgerufen werden kann, ohne dass vorher ein Objekt erzeugt worden ist;
- durch das Schlüsselwort *public*, dass es sich um eine öffentliche Methode handelt, die von überall her aufgerufen und genutzt werden kann. *Public* ist ein so genannter Access-Modifier, diese legen die "Sichtbarkeit" von Komponenten fest. Wir werden das Thema "Access-Modifier" im Kapitel 16 behandeln.

Die **dritte Zeile** enthält die Anweisungen des **Methodenblocks**, in diesem Beispiel ist das nur ein Statement. Die Ausführung eines Programmes startet mit dem Aufruf der *main*-Methode. Alles, was danach passiert, wird von dieser Hauptmethode aus gestartet: entweder werden weitere Methoden aus der eigenen Klasse aufgerufen oder es werden andere Methoden aus fremden Klassen aufgerufen.

### 10.1.2 Aufruf von Methoden durch das Senden von Messages

In objektorientierten Sprachen wie Java werden Methoden normalerweise immer aufgerufen für bestimmte Objekte. Ein Methodenaufruf wird auch bezeichnet als "das Senden von Nachrichten (messages) zu einem Objekt". Eine Message (Botschaft) fordert eine Dienstleistung an für ein ganz bestimmtes Objekt. Dabei können Argumente übergeben und ein Ergebnis zurückgeliefert werden. Generell haben Messages folgenden Aufbau:

```
objekt.methode(parameter);
```

#### Programm Methode02: Senden von Nachrichten an Objekte

```
class Methode02 {
    public static void main (String[] args) {
        String zeile = new String("Dies ist ein Satz");
        String wort = zeile.substring(9,12);
        System.out.println(wort);
    }
}
```



Für den Methodenaufruf sind also drei Angaben erforderlich:

- Name des **Objekts** (hier: *zeile*)
- Name der **Methode** (hier: *substring*)
- Wert der **Parameter** (hier: *9,12*).

Objektname und Methodenname werden durch einen Punkt getrennt ("**Punktnotation**"). Das Objekt wird auch Exemplar oder Instanz einer Klasse genannt. Es muss vorher erzeugt werden mit dem Schlüsselwort *new*. Beispiel:

```
String str = new String("Dies ist ein Satz");
```

Um Methoden einer anderen Klasse aufzurufen, muss der Identifier des Objekts dem Methodennamen vorangestellt werden. Innerhalb der Klasse genügt die Angabe des Methodenbezeichners, ohne weitere Qualifizierer.

## 10.2 Mitgelieferte Methoden benutzen

Wenn eine Methode aufgerufen werden soll, so muss deren "Schnittstelle" bekannt sein, d.h. der Aufrufer muss nicht nur den Namen kennen, er muss exakt wissen, welche Parameter die Methode erwartet und was sie als Ergebnis an ihn zurück liefert.

Die Standard-Bibliothek der J2SE enthält einige Tausend Klassen, die zum Basis-Sprachumfang von Java gehören. Jede davon kann eine Vielzahl von Methoden haben, die vom Java-Programmierer genutzt werden können.

Allein diese Fülle von Standard-Methoden zeigt, wo ein nicht zu unterschätzendes Problem der objektorientierten Programmierung liegt, nämlich in der Antwort auf die Frage, wie findet der Anwendungsprogrammierer die Methoden, die er benötigt?

Das [JDK von Sun](#) bietet eine umfangreiche Dokumentation aller Standardklassen. Sie werden in diesem Kapitel mit einigen Methoden dieser mitgelieferten Klassen arbeiten und dabei auch üben, die Dokumentation zu benutzen. Hilfreich ist es, wenn Sie sich ein Icon auf dem Desktop einrichten, um direkt die Dokumentation erreichen zu können.

**Arbeiten mit JOE:** Aus dem JOE-Editor heraus ist die Dokumentation direkt über einen Menüpunkt erreichbar: durch ? (Hilfe) und dann unter "JDK Dokumentation".

Die API-Spezifikation ist organisiert nach Packages. Entweder wählt man dann auf der linken Seite gezielt ein Paket aus und lässt sich die Klassen in diesem Paket anzeigen oder man bekommt alle Packages mit allen Klassen, alphabetisch sortiert, angezeigt. Darüber hinaus bieten die Browser über BEARBEITEN|SEITE DURCHSCHEN den gezielten Direktzugriff auf eine ganz bestimmte Klasse. Jede einzelne Klasse wird dokumentiert mit ihrem vollen Namen, ihrer Paketzugehörigkeit, ihrer Einordnung in die Vererbungshierarchie und mit allen Elementen, die diese Klasse enthält.

### 10.2.1 Arbeiten mit Methoden der mitgelieferten Klasse *Random*

Es gibt die Klasse *Random*, deren Dienstleistung darin besteht, Zufallszahlen zu erzeugen und Methoden bereit zu stellen, um damit zu arbeiten. Die API-Dokumentation enthält jeweils ausführliche Beschreibungen der Klassenelemente. Jedoch ist die komplette Dokumentation nur in englischer Sprache verfügbar.

So ist eine typische Methode der *Random*-Klasse in der API-Dokumentation wie folgt beschrieben:

```
int nextInt(int n)
    Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.
```

Die erste Zeile ist identisch mit dem Kopf der Methode, und die Zeilen danach erläutern verbal die Funktion dieser Methode. Aus dem Kopf kann man erkennen:

- Wie heißt die Methode? Antwort: *nextInt*
- Welche Eingangsparameter benötigt die Methode? Antwort: einen *int*-Wert
- Welchen Ergebnistyp liefert die Methode: Antwort: einen *int*-Wert

Die Erläuterungen dazu beschreiben, dass die Methode einen *Random*-Wert liefert, der zwischen 0 (inklusive) und dem spezifizierten Parameterwert liegt.

Aus der Sicht des Aufrufers ist eine Methode eine Black-Box, er weiß nicht (und ihn sollte auch nicht interessieren), wie die Methode intern implementiert ist. Für ihn ist nur wichtig, was die Methode leistet und wie die Schnittstelle aussieht.

#### Was versteht man unter Signatur einer Methode?

Zur Dokumentation einer Methode gehört die Beschreibung der Schnittstelle. Damit wird festgelegt, welche Nachricht der Empfänger akzeptiert. Gleichzeitig erkennt der Nutzer aus der Dokumentation, wie die Syntax für den Aufruf dieser Methode ist. Häufig spricht man auch von API (Application Programmer Interface) und meint damit eine Sammlung von Methodensignaturen.

Zur Signatur gehören:

- Name der Methode
- Art und Position der Parameter.

Diese Angaben sind die formale Festlegung, wie eine Methode aufgerufen wird, damit der Compiler überprüfen kann, ob der Methodenaufruf syntaktisch in Ordnung ist und ob er für das angegebene Objekt bzw. die Klasse erlaubt ist. Die Klasse *Random* ist Teil des Packages *java.util* (erkennbar in der ersten Zeile der Dokumentation dieser Klasse).

### Programm *Methode03*: Benutzen von Methoden der Klasse *Random*

```
public class Methode03 {  
    public static void main(String args[]) {  
        java.util.Random generator = new java.util.Random();  
        int zufallszahl = generator.nextInt(50);  
        System.out.println(zufallszahl);  
    }  
}
```

Eine Nachricht in diesem Programm (in Zeile 4) lautet:

```
generator.nextInt(50);
```

Auch sie enthält wieder die drei Bestandteile (Instanzname, Methodenname, Parameter). Die Aufgabe dieses Methodenaufrufs ist es, eine Zufallszahl zwischen 0 und 50 zu generieren. Diese Zahl wird nach Ausführung der Methode als Ergebnis zurück geliefert und "an die Stelle dieses Aufrufs" gesetzt, so dass danach die Wertezuweisung an die Variable *zufallszahl* erfolgen kann.

### Übung zum Programm *Methode03*

Ändern Sie das Programm so, dass mit dem *import*-Statement gearbeitet wird.

## 10.2.2 Arbeiten mit Methoden der mitgelieferten Klasse *PrintStream*

Das nachfolgende Beispiel *Methode04* enthält die bereits mehrfach genutzte Methode *println* (für die Ausgabe auf *Standard.out*). Diese Methode wird für eine Instanz aufgerufen, die der Programmierer nicht selbst erzeugt hat, sondern die in jedem Programm implizit zur Verfügung steht: *System.out*. Es handelt sich dabei um eine Klassenvariable mit dem Namen *out* aus der Klasse *System*. Sie enthält den Hinweis darauf, welches Gerät aktuell als Standard-Ausgabeeinheit festgelegt worden ist.

### Programm *Methode04*: Mitgeliefertes Objekt benutzen zum Methodenaufruf

```
public class Methode04 {  
    public static void main (String[] args)    {  
        System.out.println(args[0]);  
    }  
}
```

Im Programm *Methode04.java* sieht die Nachricht so aus:

```
System.out.println(args[0]);
```

Der Name des Zielobjekts dieser Nachricht steht in *System.out*; die Ausgabemethode hat den Identifier *println* und der aktuelle Parameter steht in der Variablen *args[0]*. Die Variable *args[]* ist ein Array (siehe Kapitel 13) - sie enthält die Aufrufparameter, die beim Start des Programms übergeben worden sind.

### Übung zum Programm *Methode04*

**Übung 1:** Klären Sie mit Hilfe der API-Dokumentation, welchen Datentyp das Feld *System.out* hat. Dazu ist die Beschreibung der Klasse *System* zu analysieren.

**Übung 2:** Das Feld hat den Datentyp *PrintStream*. Klären Sie mit Hilfe der API-Dokumentation, wieviel unterschiedliche *println*-Methoden es in dieser Klasse gibt und wodurch sich diese unterscheiden.

Die Aufgabe der *println*-Methode im Programm *Methode04* ist es, den Wert des ersten Commandline-Parameters auf der Standardausgabe-Einheit auszugeben. Wie aus der API-Dokumentation zu erkennen ist, gibt es etwa zehn verschiedene Methoden mit dem Namen *println*, sie unterscheiden sich lediglich durch unterschiedliche Parametertypen. Innerhalb einer Klasse kann es also mehrere Methoden geben mit demselben Identifier. Dann müssen sie sich allerdings unterscheiden durch den Typ oder durch die Anzahl der Parameter. Diese Technik nennt man **Überladen (Overloading)**.

Hier einige Hinweise zum Ausführen und Testen des Programms:

#### Testen des Programms *Methode04*

Die Übergabe der Parameter an die Methode *main* erfolgt dadurch, dass die String-Werte beim Programmstart als Aufrufparameter mitgegeben werden. Es gibt mehrere Möglichkeiten, wie dies geschehen kann.

##### Möglichkeit 1: Aufruf über Commandline

Beim Aufruf des Programms über die Befehlszeile eines Consolfensters können hinter dem Programmnamen die Werte angegeben werden. Beispiel:



Abb. 10.2: Parameterübergabe beim Aufruf des Programms über Commandline

##### Möglichkeit 2: Aufruf im JOE

Beim Arbeiten mit dem Java-Editor JOE erscheint nach dem Aufruf des Menüpunkts "JAVA|Starten mit Argumenten" ein Fenster für die Eingabe der Argumente.

Weitere Hinweise zum Arbeiten mit Argumenten beim Start eines Programms: Die Daten werden immer als String-Typen behandelt. Wenn im Programm mit Argumenten gearbeitet wird, dann müssen auch entsprechende Werte übergeben werden, andernfalls gibt es Fehler bei der Ausführung des Programms. Im Kapitel 13 (Arrays) gibt es ausführliche Beispiele zu diesem Thema.

### 10.2.3 Arbeiten mit Methoden der mitgelieferten Klasse *Math*

In der mitgelieferten Klasse *Math* gibt es eine Methode mit dem Identifier *max*. Sie ist wie folgt beschrieben:

```
static int max(int a, int b)  
    Returns the greater of two int values
```

Die erste Besonderheit ist das Schlüsselwort *static* im Kopf der Methode. Damit ist festgelegt, dass diese Methode eine so genannte Klassenmethode ist. Sie kann benutzt werden, ohne dass vorher eine Instanz erzeugt werden muss. Klassenmethoden sind an die Klasse (und nicht an einzelne Instanzen) gebunden.

Die Signatur dieser Methode beschreibt außerdem, dass sie zwei ganzzahlige Argumente erwartet. Nicht erkennbar ist die Semantik, also die Wirkung des Aufrufs. Diese ist verbal in der Dokumentation beschrieben und wird als "Spezifikation" bezeichnet. Der Aufruf für die Ausführung dieser Methode muss der Signatur entsprechen (das wird überprüft vom Compiler). Das Verhalten der Methode muss übereinstimmen mit der Spezifikation (das kann maschinell nicht überprüft werden).

Die Methode liefert einen *int*-Wert zurück.

#### Programm Math01: Arbeiten mit einer *static*-Methode

```
public class Math01 {  
    public static void main (String[] args) {  
        int ergebnis = Math.max(5,3);  
        System.out.println(ergebnis);  
    }  
}
```

#### Übung zum Programm *Math01*

Ändern Sie das Programm so, dass mit Hilfe einer Methode der Klasse *Math* ein *float*-Wert in eine ganze Zahl umgewandelt wird (mit Auf- bzw. Abrunden).

#### Hinweise zum *import*-Statement

Die Klasse *Math* ist Teil des Packages *java.lang*. Dieses Paket ist in jedem Java-Programm bekannt, ohne dass der Programmierer dies explizit angeben muss. Alle anderen Packages müssen ausdrücklich durch eine spezielle Anweisung, die *import*-Anweisung, im Programm bekannt gemacht werden, damit ein Zugriff auf die Klassen dieses Paketes möglich ist.

Eine andere Möglichkeit ist die voll-qualifizierte Namensangabe beim Aufruf von Klassen - dies ist jedoch schreibaufwändig.

Beispiele für den Einsatz der *import*-Anweisung folgen ab Abschnitt 10.3.

### 10.2.4 Arbeiten mit Methoden der mitgelieferten Klasse *System*

Eine häufig benutzte Klasse ist die Klasse *System*. Sie enthält verschiedene nützliche Felder und Methoden. Alle Member sind *static*, also Klassenmember, die benutzt werden können, ohne vorher eine Instanz zu erzeugen. Um diese Member zu referenzieren, muss die Variable oder die Methode qualifiziert werden mit dem Klassennamen, z.B. *System.out* oder *System.getSecurityManager()*.

Das folgende Programm benutzt die *System*-Klasse zweimal, nämlich beim Aufruf der Methode *getProperty*, um den Benutzernamen des Systems zu ermitteln und dann bei der Ausgabe mit *println*.

#### Programm System01: Arbeiten mit der *System*-Class

```
class System01 {
    public static void main(String[] args) throws Exception {
        String name;
        name = System.getProperty("user.name");
        System.out.println(name);
    }
}
```

#### Wozu können *static import*-Anweisungen genutzt werden?

Eine Variante ist der Einsatz von *static*-Import. Dadurch kann der Aufruf von statischen Membern einer Klasse vereinfacht werden.

#### Programm System02: *Static-Import* erlaubt "unqualifizierten" Zugriff

```
import static java.lang.System.*;
class System02 {
    public static void main(String[] args) throws Exception {
        String name;
        name = getProperty("user.name");
        out.println(name);
    }
}
```

Der Vorteil ist die kürzere Schreibweise beim Referenzieren von *static*-Membern.

### 10.2.5 Arbeiten mit Methoden der mitgelieferten Klasse *Integer*

Die Java-Standard-Bibliothek enthält für jeden primitiven Datentyp eine korrespondierende Klasse, z.B. für den *int*-Typ die Klasse *Integer* oder für den *boolean*-Typ die Klasse *Boolean*. Diese Klassen werden Wrapper-Klassen (Hüllenklassen) genannt, weil sie den primitiven Typ "einpacken" in einen Klassentyp.

Durch den Einsatz von Wrapper-Klassen erhält der Programmierer zusätzliche Verarbeitungsmöglichkeiten, denn die Klassen enthalten Methoden, die zusätzlich zu den eingebauten Operatoren auf diese Datentypen operieren. So enthält die Klasse *Integer* z.B. die folgende Methode:

static int	<code>parseInt(String s)</code> Parses the string argument as a signed decimal integer.
------------	--

### Programm *Integer01*: Arbeiten mit Wrapper-Klassen

```
public class Integer01 {
    public static void main (String[] args) {
        String str = "125";
        int ergebnis = Integer.parseInt(str);
        System.out.println(ergebnis);
    }
}
```

Weitere Hinweise zu Wrapper-Klassen finden Sie im Kapitel 15.5.

## 10.3 Methodenaufruf

Ein Fazit aus den bisherigen Übungen ist folgende Regel:

Zum Aufruf einer Methode muss deren Signatur (Name und Parameter) bekannt sein. Befindet sich die Methode nicht in derselben Klasse, so muss der Aufruf qualifiziert erfolgen, d.h. entweder mit dem Objektnamen (bei Instanzmethoden) oder mit dem Klassennamen (bei Klassenmethoden).

### Übung zum Programm *Integer01*

Ändern Sie das Programm so, dass mit Hilfe einer Methode der Klasse *Integer* der hexadezimale Wert der Variablen *ergebnis* ausgegeben wird.

### Lösungsvorschlag

```
public class Integer02 {
    public static void main (String[] args) {
        String str = "125";
        int ergebnis = Integer.parseInt(str);
        System.out.println(Integer.toHexString(ergebnis));
    }
}
```

### 10.3.1 Schachteln und Verkettung von Methodenaufrufen

Der vorherige Lösungsvorschlag zeigt, wie Methodenaufrufe geschachtelt werden können. Eine Schachtelung kann folgenden Aufbau haben:

```
object.method1(object.method2());
```

Das entspricht dem Ausdruck des letzten Statements. Dort ist die Methode *toHexString* Teil des Parameterausdrucks für die Methode *println*. Geklammerte Ausdrücke werden immer zuerst aufgelöst, d.h. es wird von innen nach außen gearbeitet - und dann werden die Aufrufe von links nach rechts abgearbeitet.

Ein Beispiel für die Verkettung von mehreren Methodenaufrufen, die **nacheinander** ausgeführt werden, enthält das folgende Beispiel.

#### Programm *BigDecimal01*: Verkettung von Methodenaufrufen

```
import java.math.*;
class BigDecimal01 {
    static public void main(String[] _) {
        BigDecimal d1 = new BigDecimal(15);
        BigDecimal d2 = new BigDecimal(3);
        System.out.println(d1.multiply(d2).divide(d1).multiply(d2));
    }
}
```

Hier hat die Verkettung in der letzten Zeile den folgenden Aufbau (innerhalb der Klammer):

```
object.method1().methode2().methode3();
```

Die Voraussetzung für diese Art der Verkettung von Methoden ist, dass Methode1 und Methode2 jeweils ein Objekt (eine Referenzvariable) als Ergebnis liefern.

#### Hinweise zum *import*-Statement

Die erste Zeile des Programms enthält eine *import*-Anweisung. Die Klasse *BigDecimal* ist Teil des *java.math*-Pakets. Der volle ("qualifizierte") Name der Klasse ist also: *java.math.BigDecimal*. Sie kann adressiert werden entweder mit diesem voll qualifizierten Namen oder durch eine Kurzschreibweise, indem das *import*-Statement benutzt und dann im Programm lediglich der Klassenname codiert wird. In diesem Programm enthält die erste Zeile eine *import*-Anweisung, deswegen kann im nachfolgenden Quelltext die Kurzschreibweise benutzt werden.

#### Übungen zum Programm *BigDecimal01*

**Übung 1:** Bitte klären Sie für sich, in welcher Reihenfolge die Methoden ausgeführt werden.

**Übung 2:** Bitte arbeiten Sie ohne die *import*-Anweisung. Lösungshinweis: Dann muss die Klasse *BigDecimal* voll qualifiziert werden.



### 10.3.2 Arbeiten mit Methoden der mitgelieferten Klasse *Properties*

Besonders im Package "*java.util*" sind einige Klassen enthalten, die als so genannte "Utilities" (Hilfsklassen) bezeichnet werden und dem Programmierer Dienste für häufig anfallende Arbeiten anbieten. So gibt es die Klasse *Properties*, die es mit einfachen Methodenaufrufen ermöglicht, im Speicher Daten mit Suchbegriffen zu kennzeichnen, so dass ein Direktzugriff darauf möglich ist. Die Daten werden in Form von zwei Strings gespeichert, wobei der erste String den Suchbegriff (Schlüssel, "key") und der zweite String den dazu gehörenden Wert ("value") enthält.

Diese Art der Speicherung ist für viele Aufgabenstellungen hilfreich, z.B. werden Parameter zwischen heterogenen Systemen häufig in der Key-Value-Form ausgetauscht oder Systemwerte in dieser Form gespeichert und abgefragt (z.B. in Environment-Variablen).

#### Programm *Properties01*: Arbeiten mit Key-/Value-Paaren

```
import java.util.*;
public class Properties01 {
    public static void main (String[] args) {
        Properties prop = new Properties();
        prop.setProperty("name", "Roman Merker");
        prop.setProperty("beruf", "Programmierer");
        prop.setProperty("ort", "Steinfurt");
        prop.list(System.out);
    }
}
```

#### Übung zum Programm *Properties01*

Die Klasse *Properties* bietet komfortable Möglichkeiten für den Direktzugriff auf einzelne Properties. Dabei muss lediglich der Schlüssel (key) als Suchbegriff angegeben werden. Bitte ändern Sie das Programm so, dass nur der Wohnort ausgegeben wird, möglichst durch einen geschachtelten Methodenaufruf. Lösungsvorschlag für die Ausgabe: `"System.out.println(prop.getProperty("ort"));`

Das folgende Programm zeigt, wie kompakt ein Ausdruck codiert werden kann, der einen geschachtelten Methodenaufruf enthält.

#### Programm *Properties02*: Geschachtelter Methodenaufruf

```
public class Properties02 {
    public static void main (String[] args) {
        String s = "os.name";
        System.out.println(System.getProperties().getProperty(s));
    }
}
```

Die Zeile mit dem *println*-Befehl enthält den Aufruf der statischen Methode *System.getProperties()*. Diese Methode liefert als Ergebnis ein Objekt mit einer Zusammenstellung aller Systemeigenschaften. Daraus wird eine ganz bestimmte Property gelesen: mit *getProperty()* wird der **Key** "os.name" gesucht und dessen **Value** mit *println()* ausgegeben.

Die Ausführung dieser geschachtelten Methoden erfolgt nacheinander, von links beginnend.

### 10.3.3 Arbeiten mit Methoden der mitgelieferten Klasse *DecimalFormat*

Das nächste Beispiel demonstriert verschiedene Möglichkeiten für das Formatieren und Ausgeben von Dezimalzahlen. Gezeigt wird, welche Möglichkeiten es gibt für primitive Typen und für Typen der Klasse *BigDecimal*.

#### Programm Zahlen01: Formatieren und Ausgeben von Dezimalzahlen

```
import java.text.DecimalFormat;
import java.math.*;
class Zahlen01 {
    public static void main(String[] args) {

        // Primitive Datentypen runden / aufbereiten
        double zahl1 = 0.15780003;
        DecimalFormat df = new DecimalFormat("##,##0.00");
        System.out.println(df.format(zahl1));

        // Komfortabler mit BigDecimal und MathContext
        BigDecimal zahl2 = new BigDecimal(0.15780003);
        int nachkomma = 3;
        MathContext mc = new MathContext(nachkomma,
                                         RoundingMode.HALF_UP);
        BigDecimal erg = zahl2.round(mc);
        System.out.println(erg);
    }
}
```

Für das Arbeiten mit der Variablen *zahl2* werden nacheinander drei Methoden aufgerufen: zunächst wird mit *new* eine Instanz der Klasse *MathContext* erstellt (und dafür die Konstruktormethode aufgerufen), und danach wird die Methode *round* aufgerufen. Für die Ausgabe ist dann die Methode *println* zuständig.

#### Übung mit Programm Zahlen01

Bitte versuchen Sie, die drei Methoden so weit wie möglich zu schachteln.

### Lösungsvorschlag: Methoden schachteln

```
import java.text.DecimalFormat;
import java.math.*;
class Zahlen02 {
    public static void main(String[] args) {
        BigDecimal zahl1 = new BigDecimal(0.15780003);
        System.out.println(zahl1.round(new MathContext(3, RoundingMode.HALF_UP)));
    }
}
```

Weil die drei Methoden jeweils in runden Klammern eingfasst sind, erfolgt die Ab-  
arbeitung von innen nach außen: zuerst *new*, danach *round* und dann *println*.

## 10.4 Eigene Methoden erstellen

Eine Methode ist immer Bestandteil einer Klasse. Um eine Methode zu deklarieren,  
wird eine Klasse erstellt und dann ihre Methoden (und Attribute) darin beschrieben.

Eine Methode hat zwei Bestandteile: die Methodendeklaration (den Kopf) und die  
Implementierung (den Rumpf):

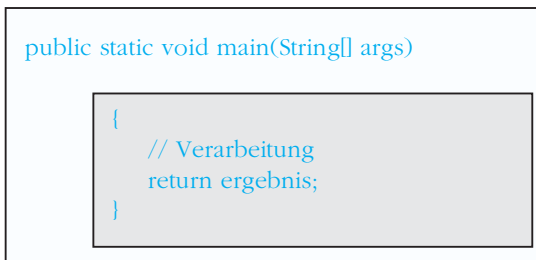


Abb. 10.3: Definition einer Methode

Die **Methodendeklaration** besteht im Minimum aus dem Namen und der Angabe  
des Returntyps. Der Methodenname kann (wie alle Identifier in Java) aus jedem Zei-  
chen des Unicodes bestehen. Aber es sollten folgende Empfehlungen eingehalten  
werden:

- nur ASCII-Zeichen benutzen
- nur Kleinbuchstaben benutzen
- wenn möglich, ein Verb benutzen.

Der **Kopf der Methode** kann weitere Informationen enthalten, die dem Aufrufer  
und dem Compiler die Methode näher spezifizieren. Wahlweise können auch Para-  
meter deklariert werden, die dann beim Aufruf mit aktuellen Werten gefüllt werden.  
Die Methodenimplementierung enthält die Sammlung von Statements, eingeschlos-

sen in geschweiften Klammern {...}. Wahlweise kann mit dem Schlüsselwort *return* ein Ergebniswert an den Aufrufer zurück gegeben werden.

#### 10.4.1 Neue Methode in Hauptklasse erstellen und benutzen

Das nächste Beispiel ergänzt eine ausführbare Klasse um eine weitere Methode. Die Klasse soll also neben der *main*-Methode noch eine zusätzliche Methode, nämlich die Methode *ausgeben()*, enthalten, ebenfalls eine *static*-Methode.

##### Programm Methode05: Mehrere Methoden in derselben Klasse

```
public class Methode05 {
    public static void main (String[] args) {
        for (int i=0; i<5; i++)
            ausgeben();
    }
    static void ausgeben() {
        System.out.println("Hallo Welt");
    }
}
```

Diese Art der Methodendeklaration und des Methodenaufrufs ist untypisch für objektorientiertes Programmieren, denn die Methode enthält das Schlüsselwort *static*. Dadurch ist der Aufruf dieser Methode möglich, ohne dass vorher eine Instanz erzeugt worden ist. Weil in diesem Beispiel die Methode aus einer anderen *static*-Methode aufgerufen wird und dort noch kein Objekt existiert, ist das Schlüsselwort *static* erforderlich.

##### Übungen zum Programm Methode05

**Übung 1:** Bitte versuchen Sie durch Programmänderung herauszufinden, ob es in Java möglich ist, dass Methodendeklarationen geschachtelt sein können, d.h. kann die Methode *ausgeben* auch Teil der Methode *main* sein, also innerhalb von *main* deklariert sein?

**Übung 2:** Nachdem wir geklärt haben, dass in Java keine geschachtelten Methodendeklarationen erlaubt sind, überprüfen Sie bitte durch eine weitere Programmänderung, welche Fehlermeldung die Umwandlung liefert, wenn Sie das Schlüsselwort *static* bei der Methode *ausgeben* entfernen.

Fazit: Klassenmethoden können keine Instanzmethoden aufrufen.

#### 10.4.2 Neue Methode in einer anderen Klasse erstellen und benutzen

Was ist nun zu beachten, wenn eine nicht-ausführbare Klasse mit neuen Methoden erstellt und benutzt wird? Wir zeigen einige Beispiele, bei denen eigenständige, abgeschlossene Aufgaben in neue Klassen ausgelagert werden, damit sie dann von diversen anderen Programmen genutzt werden können.

#### 10.4.2.1 Arbeiten mit *static*-Methoden (Klassen-Methoden)

Die Aufgabenstellung für folgende, neu zu erstellende Klasse lautet: Es sollen Daten von *System.in* eingelesen, als Werte vom Typ *double* interpretiert und an den Aufrufer der Methode übergeben werden.

##### Programm Lesen: Eine Hilfsklasse mit einer Lese-Methode

```
import java.io.*;
class Lesen {
    static String zeile;
    static InputStreamReader isr =
        new InputStreamReader(System.in);
    static BufferedReader bfr = new BufferedReader(isr);
    public static double liesDouble() throws Exception {
        System.out.println("Bitte double-Wert eingeben: ");
        zeile = bfr.readLine();
        double zahl = Double.parseDouble(zeile);
        return zahl;
    }
}
```

Diese Klasse hat keine *main*-Methode. Es handelt sich also nicht um ein ausführbares Programm, lediglich die Umwandlung ist möglich.

Im zweiten Schritt erstellen wir ein ausführbares Programm, das die Dienstleistung dieser Klasse *Lesen* nutzt (durch Aufruf der Methode *liesDouble*).

##### Programm LesenTest01: Einlesen und Ausgeben von Double-Werten

```
public class LesenTest01 {
    public static void main(String[] args) throws Exception {
        double d = Lesen.liesDouble();
        System.out.println(d);
    }
}
```

Beide Klassen können in *einer* Quelltextdatei stehen. Der Name der Datei *muss* dann lauten: *LesenTest01.java* (also so wie die ausführbare *public*-Klasse). Mögliche Fehlerquelle: Es darf keine weitere *public*-Klasse in dieser Datei geben.

### Übung

Erstellen Sie eine neue ausführbare Klasse *Methode06*. In der *main*-Methode soll die Methode *addieren(5, 3)* aufgerufen werden.

Diese *static*-Methode ist Teil einer weiteren, neu zu erstellenden Klasse A. Sie addiert die beiden Parameterwerte und gibt die Summe als Ergebnis zurück.

**Lösungsvorschlag (static-Methode aus einer anderen Klasse aufrufen)**

```
public class Methode06 {
    public static void main (String[] args) {
        System.out.println(A.addieren(5,3));
    }
}

class A {
    static int addieren(int a, int b) {
        return a + b;
    }
}
```

Beide Klassen können in einer Quellendatei stehen, damit werden sie gemeinsam umgewandelt, und es entstehen daraus zwei Class-Dateien. Diese Umwandlungseinheit muss aber den Namen des ausführbaren Programms haben, in diesem Fall den Dateinamen *Methode06.java*.

**10.4.2.2 Arbeiten mit non-static-Methoden (Instanz-Methoden)**

Die **typische Vorgehensweise** in objektorientierten Programmsystemen besteht in dem **Erzeugen von Objekten** und dem Austauschen von Nachrichten zwischen diesen Objekten. Wir modifizieren deshalb die beiden Einleseprogramme *Lesen.java* und *LesenTest01.java* so, dass die Methode *parseDouble* zu einer "normalen" Instanzmethode wird. Dazu muss an verschiedenen Stellen das Schlüsselwort *static* entfernt werden.

**Programm LesenTest02: Erstellen und Aufrufen von Non-Static-Methoden**

```
import java.io.*;

class Lesen02 {
    String zeile;
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader bfr = new BufferedReader(isr);

    public double liesDouble() throws Exception {
        System.out.println("Bitte double-Wert eingeben: ");
        zeile = bfr.readLine();
        double zahl = Double.parseDouble(zeile);
        return zahl;
    }
}

public class LesenTest02 {
    public static void main(String[] args) throws Exception {
        Lesen02 obj1 = new Lesen02();
    }
}
```

```
        double d = obj1.liesDouble();
        System.out.println(d);
    }
}
```

### Übung zum Programm Methode06 (!)

Ändern Sie das Programm *Methode06.java* so, dass die *addieren*-Methode zu einem ganz normalen Element der Klasse *A* wird (entfernen Sie also das Schlüsselwort *static*). Wie erfolgt jetzt der Aufruf der Methode? Was muss in jedem Fall vorher erzeugt werden?

### Lösungsvorschlag: Implementierung und Aufruf einer non-static Methode

```
public class Methode07 {
    public static void main (String[] args) {
        A a = new A();
        System.out.println(a.addieren(5,3));
    }
}
class A {
    int addieren(int a, int b) {
        return a + b;
    }
}
```

Beachten Sie die unterschiedliche Bedeutung von *a* und *A*.

Besonderheit: Bei der Deklaration der Methode *addieren* in der Klasse *A* wäre es formal möglich, das Schlüsselwort *static* zu benutzen, um dadurch die *addieren*-Methode zu einer Klassenmethode zu machen. Umwandlung und Ausführung würden auch fehlerfrei funktionieren, weil die Methode keine instanzabhängigen Variablen benutzt.

Obwohl Java also den Aufruf einer Klassenmethode über den Instanznamen erlaubt, empfehlen wir doch, in solchen Fällen immer den Klassennamen zu benutzen. Dadurch wird eindeutig signalisiert, dass es sich um eine Nachricht handelt, die klassenabhängige Arbeiten ausführt und die auch keine individuellen Datenwerte verarbeiten kann.

Das folgende Beispiel zeigt den falschen Einsatz einer *static*-Methode.

### Programm Methode08: Eine *static*-Methode - fehlerhaft eingesetzt

```
public class Methode08 {
    public static void main (String[] args) {
        A a = new A();
        System.out.println(A.addieren(5,3));
    }
}
```

```
    }  
}  
class A {  
    int x = 5;  
    static int addieren(int a, int b) {  
        return a + b + x;  
    }  
}
```

### Übung zum Programm *Method08*

Ändern Sie das Programm so, dass es fehlerfrei läuft. Hinweise: Die Methode darf nicht *static* sein (weil sie auf die Instanzvariable *x* zugreift). Die Methode muss aufgerufen werden mit einem Instanznamen (und nicht mit dem Klassennamen).

## 10.5 Methodenblock implementieren

Der Implementierungscode einer Methode steht im Methodenblock ("body").

### 10.5.1 Mit welchen Daten kann innerhalb einer Methode gearbeitet werden?

Zunächst einmal stehen einer Methode alle Variablen der eigenen Klasse zur Verfügung. Diese Variablen sind entweder Klassenvariablen oder Instanzvariablen. Abhängig davon stehen sie entweder jeder Methode zur Verfügung (die Instanzvariablen) oder nur den *static*-Methoden (die Klassenvariablen, denn die sind ebenfalls mit *static* deklariert). Dann können innerhalb der Methode neue Variablen definiert und mit Werten gefüllt werden. Diese werden lokale Variablen genannt.

Und - sehr wichtig - außerdem können einer Methode von außen (durch den Aufrufer) Daten übergeben werden. Diese werden Argumente (Parameter) genannt.

#### 10.5.1.1 Klassenvariable oder Instanzvariable?

#### Programm *Method09*: Arbeiten mit *static*-Methode (fehlerhaft)

```
public class Methode09 {  
    public static void main (String[] args) {  
        System.out.println(A.addieren(5,3));  
    }  
}  
class A {  
    int x = 5;  
    static int addieren(int a, int b) {  
        return a + b + x;  
    }  
}
```



### Übung zum Programm Methode09

Der Umwandlungsversuch endet mit einer Fehlermeldung ("non-static variable x cannot be referenced from a static context"). Bitte korrigieren Sie diesen Fehler. Versuchen Sie zwei unterschiedliche Lösungen. Lösungshinweis: Entweder wird die Variable x als *static* deklariert oder die Methode *addieren* wird ohne *static* deklariert. Versuchen Sie für sich selbst zu klären, was die Konsequenzen jeder Lösung sind.

Fazit: Eine *static*-Methode ist an die Klasse gebunden. Sie kann nicht mit den Instanzvariablen arbeiten.

#### 10.5.1.2 Lokale Variablen

Zusätzlich zu den Feldern ihrer Klasse können Methoden eigene Variable deklarieren und benutzen. Diese nennt man lokale Variablen, denn sie sind nur innerhalb dieser Methode ansprechbar. Sobald die Methode abgearbeitet ist und beendet wird, ist auch die lokale Variable nicht mehr verfügbar.

Eine lokale Variable wird nicht automatisch initialisiert, d.h. sie hat keinen Defaultwert wie die Felder einer Klasse. Deswegen muss der Programmierer dafür sorgen, dass die lokale Variable *vor* dem ersten Verarbeitungsbefehl einen korrekten Wert enthält.

#### Programm *MethodenTest10*: Arbeiten mit lokalen Variablen

```
public class MethodenTest10 {
    public static void main (String[] args)    {
        Methode10 m = new Methode10();
        for (int i=0; i<5; i++)
            m.ausgeben();
    }
}
class Methode10 {
    void ausgeben() {
        String text = "Hallo lokale Variable";
        System.out.println(text);
    }
    void aendern() {
        // text = "Neuer Inhalt für lokale Variable";
    }
}
```

#### Übung zum Programm *MethodenTest10*

**Übung 1:** Welche Fehlermeldung kommt bei der Umwandlung, wenn das letzte Statement aktiv wird (dazu muss der Kommentar in der drittletzten Zeile entfernt werden)? Warum kommt diese Meldung?

**Übung 2:** Machen Sie die Wertezuweisung in der Methode *ausgeben* zu einem Kommentar. Welche Fehlermeldung kommt bei der Umwandlung. Warum ist das so?

**Übung 3:** Fügen Sie der *main*-Methode als letztes Statement den folgenden Ausgabebefehl hinzu: `System.out.println("Die Variable i enthält: " + i);` Warum führt dieser Versuch zu einem Umwandlungsfehler? Wie kann der Fehler umgangen werden? Lösungshinweis: Die Deklaration der Variablen *i* darf nicht im Kopf der *For*-Schleife vorgenommen werden, weil diese dadurch zu einer lokalen Variablen für diesen Anweisungsblock wird.

Lokale Variable sind immer dann sinnvoll, wenn Zwischenergebnisse gespeichert oder temporäre Zustände festgehalten werden müssen, die nach Ablauf eines Blocks nicht mehr benötigt werden.

### 10.5.2 Überladen von Methoden (overload)

Die Signatur kennzeichnet eine Methode eindeutig. Sie besteht aus Methodennamen und Deklaration der Parameter. Wie Sie bereits bei der Erläuterung zum Programm *Methode04* gesehen haben, ist es durchaus möglich, innerhalb eines Programms denselben Methodennamen mehrfach zu verwenden, wenn die Methoden sich durch die Parameterliste unterscheiden. Beim Aufruf dieses Methodennamens sucht sich Java automatisch die passende Methode - und zwar abhängig von der Signatur.

#### Beispiel Methode11: Überladen von Methoden (Overloading)

```
public class Methodell {
    public static void main (String[] args) {
        A a = new A();
        System.out.println(a.addieren(5,3));
        System.out.println(a.addieren(15.3, 27.9));
    }
}

class A {
    int addieren(int a, int b) {
        System.out.println("Ganzzahlen addieren");
        return a + b;
    }
    double addieren(double a, double b) {
        System.out.println("Gleitkommazahlen addieren");
        return a + b;
    }
}
```

Was ist der Vorteil dieser Technik? Durch die Möglichkeit, Methoden zu überladen, kann der Programmierer gleichen Funktionalitäten denselben Namen geben, auch

wenn sie von verschiedenen Methoden ausgeführt werden. Dadurch wird das Programm lesbarer und Methodenaufrufe verständlicher.

### 10.5.3 Laufzeitfehler behandeln bzw. weiterreichen

Bei der Ausführung einer Methode kann es zu Laufzeitfehlern kommen. Je nach Art des Fehlers wird der Programmierer entweder gezwungen, die Reaktion darauf selbst zu programmieren, oder die Steuerung wird - wenn nichts anderes codiert ist - an die JVM übergeben, die dann entsprechende Aktionen durchführt.

#### Programm *StandardIn02*: Laufzeitfehler behandeln bzw. weiter reichen

```
class StandardIn02 {  
    public static void main(String[] args) throws Exception    {  
        int zeichen1 = System.in.read();  
        System.out.println(zeichen1);  
    }  
}
```

Durch die Angabe "throws Exception" bestimmt der Programmierer, dass eventuelle Fehler in dieser Methode nicht individuell behandelt werden sollen, sondern an die "nächsthöhere Ebene" weiter gereicht werden, in diesem Fall an die JVM.

#### Übung zum Programm *StandardIn02*

Prüfen Sie, welche Fehlermeldung der Compiler liefert, wenn die Klausel "throws Exception" im Quelltext fehlt.

Fazit: Im Kopf einer Methode können Informationen für den Aufrufer und für den Compiler codiert werden. Dazu gehört unter bestimmten Umständen die Information "throws Exception", die anzeigt, dass nicht auszuschließen ist, dass innerhalb einer Methode ein Laufzeitfehler auftreten kann, der dann aber nicht innerhalb dieser Methode abgefangen wird, sondern von dem Aufrufer behandelt werden muss.

## 10.6 Parameter übergeben und empfangen

Eine Methode wird aufgerufen, indem eine Nachricht an das Objekt gesendet wird. Dabei können Werte an die Methode übergeben werden. Diese werden als Parameter bezeichnet und sind die Eingangsvariablen für die Methodenausführung. Im Kopf der aufgerufenen Methoden müssen die Parameter deklariert sein.

### Formale Parameter

Bei der Methodendeklaration wird für jeden Parameter der Datentyp beschrieben und der interne Bezeichner dafür festgelegt. Die einzelnen Parameter werden durch Komma abgetrennt. Die Aufzählung in dieser Liste wird auch **formale Parameterliste** genannt.

Der Aufrufer muss die Schnittstelle für den Methodenaufruf kennen, d.h. er muss nicht nur die Datentypen, die erwartet werden, kennen, er muss auch wissen, an *welcher* Position in der Parameterliste *welcher* Typ erwartet wird.

### Aktuelle Parameter

Beim Aufruf der Methode muss für jeden formalen Parameter ein passendes Argument übergeben werden. Die Argumente werden **aktuelle Parameter** genannt. Sie müssen vom Typ her passen und werden positionsgenau, wie vom Empfänger gefordert, übergeben.

Die formalen Parameter wirken wie Platzhalter für die Werte der aktuellen Parameter. Innerhalb der Methode verhalten sich die formalen Parameter wie lokale Variablen, die initialisiert werden, indem der Wert der aktuellen Parameter in diese Speicherzellen kopiert werden.

### Programm Leerzeilen01: Methode, die einen ganzzahligen Wert erwartet

```
public class Leerzeilen01 {
    public static void main (String[] args)    {
        leerzeilen(5);
    }
    static void leerzeilen(int anzahl) {
        System.out.println("Nun werden " + anzahl
                           + "Leerzeilen ausgegeben");
        for (int i = 1; i<anzahl; i++) {
            System.out.println("\n");
        }
        System.out.println("Ausgabe beendet");
    }
}
```

Als aktuelle Parameter ("Argumente") können beim Aufruf angegeben werden: Variable, Literale oder Ausdrücke.

Als Datentyp sind primitive und auch Referenztypen möglich.

### Übung zum Programm Leerzeilen01

Bitte ändern Sie das Programm so ab, dass zwei Parameter übergeben werden: ein *char*-Wert und ein *int*-Wert. Dann soll die Methode den *char*-Wert so oft ausgeben, wie in der Ganzzahl angegeben.

Beispiel: Wird als Argumente ('a', 5) übergeben, so soll fünfmal der Buchstabe 'a' ausgegeben werden.

### 10.6.1 Wie werden Parameter übergeben?

Wir haben bereits geklärt, dass die Übergabe der aktuellen Parameter beim Methodenaufruf exakt in der Reihenfolge erfolgen muss, wie sie von der empfangenden Methode als formale Parameter deklariert wurden. Hierzu ein Beispiel:

**Deklaration** beim Empfänger:

```
void berechnen(int z1, int z2, String s, float z3);
```

**Aufruf** durch:

```
berechnen(zahl1, zahl2, text, zahl3);
```

Die Deklaration beschreibt vier formale Parameter. Sie haben die Datentypen *int*, *int*, *String* und *float*.

Die Namen dieser Parameter sind für den Aufrufer unwichtig, denn diese haben nur Bedeutung für den Block der Methode (also für die Implementierung).

Wichtig für den korrekten Aufruf ist allerdings, dass die vier Parameter in der Reihenfolge angeliefert werden, wie es vom Empfänger verlangt wird.

**Fazit:** Java arbeitet mit Positionsparametern, d.h. die Übergabe erfolgt exakt entsprechend ihrer Position (und nicht etwa als Namensparameter, bei denen die Reihenfolge keine Rolle spielt).

Der Compiler kann leider nur überprüfen, ob die Datentypen übereinstimmen. Wenn der Aufrufer aber die Bedeutung z.B. der beiden ersten *int*-Parameter verwechseln würde, führte das zu falschen Ergebnissen, ohne dass bei der Compilierung oder bei der Ausführung des Programms eine Fehlermeldung möglich wäre.

#### Eine wichtige Frage gilt es noch zu klären:

Werden Parameter so an die aufrufende Methode geliefert, dass die Originalinformationen dort ankommen (und dort auch gegebenenfalls geändert werden können) oder werden Kopien übergeben?

#### Java arbeitet immer mit "Call by Value"

Der Aufruf einer Methode erfolgt immer so, dass die Werte der Parameter kopiert und diese Kopien an den Aufrufer übertragen werden, d.h. in die Speicherzelle der formalen Parameter kopiert werden. Dieser Mechanismus wird "CALL BY VALUE" genannt.

Doch muss man dabei grundsätzlich zwischen den zwei Arten von Datentypen (primitive Typen und Referenztypen) unterscheiden.

### 10.6.2 Primitive Typen als Parameter

Das nachfolgende Programm demonstriert, wie die Parameterübergabe erfolgt, wenn es sich um primitive Datentypen handelt.

#### Programm Methode12: Primitiver Typ als Parameter

```
class Methode12 {
    public static void main (String[] args) {
        double zahl = 123.45;
        aendern(zahl);
        System.out.println(zahl);
    }
    static public void aendern(double zahl) {
        zahl = zahl + 200;
        System.out.println(zahl);
    }
}
```

Ausgegeben werden 323.45 und 123.45. Das zeigt, dass der Inhalt der Variablen *zahl* sich in der Hauptmethode *main* nicht geändert hat, weil nämlich die Methode *aendern* mit einer Kopie der Daten arbeitet.

Die Klasse *Punkt01.java* bietet folgenden Service: Sie kann die Werte für eine Position auf einer X- und einer Y-Achse speichern und überprüft bei Bedarf, ob beide Ganzzahlenwerte den Wert 0 enthalten. Dann wird eine Meldung am Consolbildschirm ausgegeben.

#### Programm Punkt01: Empfangen von *int*-Werten als Parameter

```
public class Punkt01 {
    private int x;
    private int y;

    void speichern(int a, int b) {
        x = a;
        y = b;
    }
    void pruef() {
        if (x < 0 || y < 0)
            System.out.println("Der Punkt liegt ausserhalb");
        else
            System.out.println("Der Punkt liegt im Fenster");
    }
}
```

Nun codieren wir einen Client, der eine Instanz erstellt und für dieses Objekt die beiden Methoden der Klasse ausführt.

**Programm *PunktTest01*: Testen der Klasse *Punkt01* und Parameterübergabe**

```
public class PunktTest01 {
    public static void main(String[] args) {
        int a = 5;
        int b = 3;
        Punkt01 p = new Punkt01();
        p.speichern(a, b);
        p.pruef();
    }
}
```

In dem vorletzten Statement wird die Methode *speichern* aufgerufen und dabei die aktuellen Parameter *a* und *b* übergeben. Die aktuellen Werte der beiden Variablen werden kopiert und dann diese Duplikate an Methode *speichern* übertragen.

**Übung zum *Punkt01***

Bitte ändern Sie die Methode *speichern* so ab, dass Sie die Variable *a* mit dem Wert 20 füllen (Wertezuweisung). Fügen Sie dann im Programm *PunktTest01* die folgende Zeile hinzu und überprüfen Sie das Ergebnis:

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

### 10.6.3 Referenztypen als Parameter

Natürlich können auch komplette Objekte als Parameter deklariert werden. Allerdings passiert dann etwas fundamental anderes als beim Arbeiten mit primitiven Datentypen. Um die unterschiedliche Wirkung zu demonstrieren, modifizieren wir die obigen Programme. Zunächst wird die Klasse *Punkt01* ergänzt um die Methode *addieren*. Diese Methode erwartet als Parameter ein Objekt der Klasse *Punkt02*.

**Programm *Punkt02*: Die Klasse enthält zusätzlich die Methode *addieren***

```
public class Punkt02 {
    private int x;
    private int y;

    void speichern(int a, int b) {
        x = a;
        y = b;
    }
    void pruef() {
        if (x < 0 || y < 0)
            System.out.println("Der Punkt liegt ausserhalb");
    }
}
```

```
        else
            System.out.println("Der Punkt liegt im Fenster");
    }
    void addieren(Punkt02 p2) {
        x = x + p2.x;
        y = y + p2.y;
    }
}
```

Danach ändern wir das Clientprogramm *PunktTest01* so, dass eine zweite Instanz erzeugt wird und diese als Parameter an die *additions*-Methode übergeben wird.

### Programm *PunktTest02*: Methode *addieren* benutzen

```
public class PunktTest02 {
    public static void main(String[] args) {
        int a = 5;
        int b = 3;
        Punkt02 p = new Punkt02();
        p.speichern(a, b);
        Punkt02 p2 = new Punkt02();
        p.addieren(p2);
    }
}
```

### Übung zum Programm *PunktTest02*

Ergänzen Sie die Methode *addieren* in der Klasse *Position02.java* um folgende Anweisung:

```
p2.x = -17;
```

Dadurch wird ein Element des *p2*-Objekts geändert. Dabei interessiert uns besonders die Frage, ob diese Änderung auch im Originalobjekt durchgeführt wurde.

Prüfen Sie deshalb im Programm *PositionTest02.java*, ob diese Änderung hier wirksam geworden ist (durch Aufruf der Methode *pruef* für dieses Objekt):

```
p2.pruef();
```

Es muss die Information kommen "Der Punkt liegt ausserhalb". Damit ist folgendes bewiesen: Bei der Übergabe von Instanzen als Parameter wird die Referenzvariable kopiert, und diese Kopie wird an die aufgerufene Methode übertragen. Damit hat die Methode die Originaladresse der Instanz. Jede Manipulation an diesem Objekt verändert somit das Originalobjekt.

**Fazit:** Bei der Übergabe von Referenzvariablen als Parameter ist die Wirkung dieselbe wie bei "Call by Reference": Der Empfänger arbeitet mit dem Original. Jede Änderung wirkt sich auch bei dem Aufrufer der Methode aus.



### Zusammenfassung der Regeln beim "Call by Value"

- Java arbeitet bei der Parameterübergabe grundsätzlich mit "Call by value". Das heißt, die Parameter werden kopiert und diese Kopie erhält dann die aufgerufene Methode.
- Allerdings ist die Wirkung dieses Mechanismus total unterschiedlich, je nachdem, ob primitive Variable oder Referenzvariable übertragen werden:
  - Bei primitiven Variablen haben Änderungen der Kopie keine Auswirkungen auf das Original.
  - bei Referenzvariablen verweisen die Kopie der Referenzvariablen und das Original auf ein und dasselbe Objekt. Werden also Änderungen vorgenommen, wird immer das Originalobjekt verändert (es existiert ja auch keine Kopie).

#### 10.6.4 Gibt es in Java Default-Parameter?

In Java gibt es nicht die Möglichkeit, einen Parameterwert mit einem Defaultwert zu belegen. Eine mögliche Lösung für diese Aufgabenstellung (durch Überladen der Methoden) zeigt das folgende Programm.

#### Programm *Leerzeilen02*: Methoden überladen, damit Defaultwert möglich

```
public class Leerzeilen02 {
    public static void main (String[] args)    {
        leerzeilen(05);
        leerzeilen();
    }
    static void leerzeilen(int anzahl) {
        for (int i = 1; i<anzahl; i++) {
            System.out.println("\n");
        }
    }
    static void leerzeilen() {
        int defaultanzahl= 2;
        for (int i = 1; i<defaultanzahl; i++) {
            System.out.println("\n");
        }
    }
}
```

Wenn beim Methodenaufruf kein Parameter übergeben wird, dann arbeitet die Methode *leerzeilen()* mit dem Default-Wert 2 (denn die JVM wählt automatisch die passende Methode aus).

### 10.6.5 Müssen Parametertypen beim Methodenaufruf exakt übereinstimmen?

Müssen die Datentypen der aktuellen Parameter exakt übereinstimmen mit den Typen der formalen Parameter. Nein, aber sie müssen "(zuweisungs-) kompatibel" sein, d.h. sie müssen ohne explizites Casting (siehe Kapitel 15) umwandelbar sein in den gewünschten Typ. Eine problemlose Umwandlung von Datentypen ist immer dann möglich, wenn dadurch kein Informationsverlust auftritt.

#### Programm Leerzeilen03: Übergabe von *short*-Typ, Empfang als *int*-Typ

```
public class Leerzeilen03 {
    public static void main (String[] args)    {
        short zahl = 5;
        leerzeilen(zahl);
    }
    static void leerzeilen(int anzahl) {
        for (int i = 1; i<anzahl; i++) {
            System.out.println("\n");
        }
    }
}
```

Der Aufruf der Methode *leerzeilen* erfolgt mit dem aktuellen Parameter *zahl*. Dieser ist zwar vom Typ *short* - und erwartet wird der Typ *int*. Da aber *short* zuweisungsverträglich ist, wird stillschweigend eine Typumwandlung vorgenommen und die Methode ausgeführt.

### 10.6.6 Aktuelle Parameter können auch in einem Ausdruck stehen

Beim Aufruf einer Methode können die aktuellen Parameter als Variable, als Literal oder als Ausdruck angegeben werden. Das nachfolgende Programm demonstriert, wie die Parameterübergabe als Ausdruck codiert werden kann.

#### Programm Leerzeilen04: Parameter in Form eines Ausdrucks

```
public class Leerzeilen04 {
    public static void main (String[] args)    {
        int zahl = 5;
        leerzeilen(zahl / 2 + 1);
    }
    static void leerzeilen(int anzahl) {
        for (int i = 1; i<anzahl; i++) {
            System.out.println("\n");
        }
    }
}
```

### 10.6.7 Variable Parameterliste

Nicht immer ist es möglich, bei der Deklaration einer Methode die genaue Anzahl der zu übergebenen Parameter anzugeben. Deshalb gibt es die Möglichkeit, eine variable Anzahl zu deklarieren. Dies geschieht durch Codieren von 3 Punkten ... zwischen dem Typ und dem Identifier.

#### Programm *VarArgs01*: Variable Anzahl Parameter empfangen

```
public class VarArgs01 {
    public static void main (String[] args)    {
        ausgabe("Roman", "Erwin");
    }
    static void ausgabe(String ... namen) {
        for (String n : namen) {
            System.out.println("Hallo " + n);
        }
    }
}
```

## 10.7 Rückgabewert

Beim Aufruf von Methoden ist zu unterscheiden zwischen Methoden, die einen Rückgabewert liefern, und Methoden, die lediglich eine Aufgabe erfüllen, aber kein Ergebnis an den Aufrufer zurückgeben. Manchmal werden diese beiden unterschiedlichen Arten von Unterprogrammen auch als Funktion (mit Rückgabewert) und Prozedur (ohne Rückgabewert) bezeichnet. In Java gibt es diese Unterscheidung nicht, beide Arten sind als Methoden zu implementieren.

Zunächst muss deutlich gesagt werden, dass maximal *ein* Ergebniswert an den Aufrufer übertragen werden kann. Mehr als ein Wert ist nicht möglich. Dies ist deswegen aber kein wirkliches Problem, weil doch mehrere Variablen zusammengefasst werden können, z.B. als Array, als Objekt oder als eine Collection von Instanzen.

Wenn die aufgerufene Methode ein Ergebnis an den Aufrufer liefert, so muss dies bei der Deklaration der Methode festgelegt werden. Im Kopf der Methode steht der Datentyp des Ergebnisses, und dann *muss* die Implementierung auch mit dem Schlüsselwort *return* das Ergebnis liefern. Gleichzeitig wird die Methode sofort verlassen, wenn *return* ausgeführt wird. Innerhalb einer Methode kann *return* mehrfach (alternativ) codiert werden

Der Aufruf einer Methode mit Rückgabewert erfolgt durch:

```
ergebnis = objekt.methode(parameter);
```

Hierzu ein Beispiel.

**Programm *Potenz01*: Ergebnisrückgabe**

```

public class Potenz01 {
    public static void main(String[] args) {
        int a = 5;
        int b = 3;
        System.out.println(potenzieren(a,b));
    }
    static int potenzieren(int z1, int z2) {
        int erg = z1;
        if (z2 == 0)
            return 1;
        for (; z2>1; z2--) {
            erg = erg * z1;
        }
        return erg;
    }
}

```

Dieses Programm hat eine *static*-Methode zum Errechnen der Potenz zweier Zahlen, wobei die erste Ganzzahl als Basis und die zweite als Exponent zur Basis 10 interpretiert wird. In der Kopfzeile der Methode *potenzieren* ist als Ergebnistyp *int* eingetragen. Das bedeutet, dass der Compiler überprüft, dass auch wirklich ein Ergebnis mit *return* geliefert wird und dass dieser Wert auch wirklich vom Typ *int* ist.

Es gibt auch typlose Methoden, das sind Methoden, die kein Ergebnis liefern. Diese werden bei der Deklaration durch das Schlüsselwort *void* gekennzeichnet anstelle des Rückgabetyps. Dies ist der Hinweis, dass der Aufrufer "nichts zu erwarten hat". Enthält die Deklaration einer Methode **nicht** das Schlüsselwort *void*, so **muss** mit *return* ein Ergebnis zurückgegeben werden. Auch dies wird vom Compiler überprüft.

**Fazit:** Hat eine Methode einen Rückgabetypp, so muss hinter Schlüsselwort *return* ein Wert dieses Typs stehen; enthält die Deklaration dagegen das Schlüsselwort *void*, so darf kein Returnwert vorhanden sein. Innerhalb einer Methode wird der Ausdruck hinter dem Schlüsselwort *return* kopiert und an den Aufrufer geliefert.

**Typanpassung bei dem Returnwert**

Für den Datentyp des Returnwerts gelten die gleichen Regeln wie beim Arbeiten mit Parametern:

- Es können sowohl primitive Datentypen als auch Referenztypen übergeben werden. Bei Referenztypen wird die Referenz an den Aufrufer übergeben, so dass dieser mit dem Originalobjekt arbeitet.
- Grundsätzlich muss der Wert dem definierten Typ entsprechen. Bei zuweisungsverträglichen Typen findet eine automatische Anpassung statt.

Das folgende Programm ermittelt aus zwei ganzzahligen Werten den größten gemeinsamen Teiler. Dabei weichen die Datentypen der formalen Parameter von den Typen der aktuellen Parameter ab, und auch der Returntyp ist anders deklariert als der Wert der Ergebnisvariablen. Wichtig dabei ist, dass die Typen zuweisungsverträglich sind.

### Programm Return01: Zuweisungsverträgliche Datentypen

```
public class Return01 {
    public static void main (String[] args)    {
        short zahl1 = 15;
        short zahl2 = 48;
        System.out.println(teiler(zahl1, zahl2));
    }
    static int teiler(int z1, int z2) {
        int rest;
        do {
            rest = z1 % z2;
            z1    = z2;
            z2 = rest;
        } while (rest > 0);
        return z1;
    }
}
```

Hinter *return* kann ein Ausdruck stehen. Der Typ dieses Ausdrucks bestimmt den Rückgabetyt der Methode.

### Programm Return02: Ein Ausdruck als Rückgabewert

```
public class Return02 {
    public static void main (String[] args)    {
        System.out.println(ausgabe("Erwin"));
    }
    static String ausgabe(String name) {
        return "Hallo " + name;
    }
}
```

Der Ausdruck hinter dem Schlüsselwort *return* kann auch in runde Klammern eingfasst werden. Außerdem ist es möglich, das Schlüsselwort *return* ohne irgendeinen Wert zu benutzen, dann wird die Steuerung an den Aufrufer zurückgegeben, ohne einen Ergebnistyp zu liefern. Voraussetzung dafür ist natürlich, dass im Kopf der Methode anstelle des Rückgabetyps *void* eingetragen ist.

## 10.8 Zusammenfassung

Methoden lösen eigenständige Teilaufgaben innerhalb einer Klasse. Sie haben einen Namen, damit sie unter diesem Namen aufgerufen werden können. Je nach Einsatz und Betrachtungsweise werden sie eingesetzt,

- um ein komplexes Programm zu zerlegen in überschaubare kleine Einheiten (= das ist die Sichtweise der Strukturierte Programmierung),
- um wiederkehrende Programmteile einmal zu programmieren und diese dann von verschiedenen Stellen aufzurufen (= Wiederverwendbarkeit von Code),
- um festzulegen, welche Operationen mit einem benutzerdefinierten Datentyp (also für die Felder einer Klasse) gemacht werden können (= das ist die objekt-orientierte Sichtweise),
- um die Sprache zu erweitern um einen neuen Command (= Sichtweise der prozeduralen Programmierung).

Eine Methode kann **Parameter** empfangen und/oder ein **Ergebnis** liefern. Die Argumente werden als Kopie des Wertes übergeben ("call by value"). Das bedeutet für einfache Variablen, dass der Datenwert transportiert wird; für Referenzvariablen bedeutet dies die Übergabe der Referenz. Wichtige Konsequenz dieses Mechanismus: Die aufgerufene Methode arbeitet bei einfachen Variablen mit der Datenkopie und bei Instanzen mit dem Originalobjekt.

Bei einer Methode, die **kein Ergebnis** liefert, muss anstelle des Returntyps das Schlüsselwort *void* angegeben werden. Fehlt dieses Schlüsselwort, so muss innerhalb der Methode mit *return* ein Wert geliefert werden.

Methoden sind immer als Teil einer Klasse definiert. Dabei werden Klassen-Methoden und Instanz-Methoden unterschieden. **Klassen-Methoden** (definiert mit dem Schlüsselwort *static*) sind allgemeine Dienstleistungen der Klasse, während **Instanz-Methoden** nur auf vorher erzeugte Instanzen dieser Klasse operieren. Durch das Senden von Messages zu diesem Objekt werden die Methoden aktiviert.

Der Aufbau einer Nachricht zum Methodenaufruf ist unterschiedlich. Innerhalb einer Klasse werden die Methoden aufgerufen mit ihrem Bezeichner. Außerhalb der Klasse erfolgt der Aufruf durch "Qualifizierung" dieses Bezeichners,

- bei Klassenmethoden durch: `klassenname.methodenname`
- bei Instanzmethoden durch: `objektname.methodenname`

In einer Klasse muss die **Signatur** einer Methode eindeutig ("unique") sein. Es darf keine zwei Methoden mit demselben Namen und derselben Parameterliste geben, weil dann die JVM kein Unterscheidungskriterium hat bei der Auswahl der richtigen Methode. Allerdings gibt es die Technik des **Überladens** ("**overload**"), bei der es in einer Klasse mehrere Methoden mit gleichem Namen gibt, die sich jedoch unterscheiden müssen durch die Anzahl oder Art der Parameter.

# 11

## Klassen beschreiben und benutzen

---

Objektorientierte Programmierung ist vor allem gekennzeichnet durch den Einsatz von Klassen als Schablonen für die Instanzerzeugung. In Klassen werden die Eigenschaften (Attribute) und Fähigkeiten (Methoden) von gleichartigen Objekten einmalig beschrieben und dann beliebig oft benutzt, um konkrete Einzelfälle zu bearbeiten. Dazu wird zunächst eine Instanz erzeugt (mit dem Schlüsselwort *new*) und dann können die Methoden dieser Klasse für diese Instanz aufgerufen werden.

In diesem Kapitel lernen Sie,

- wie eine Klassenbeschreibung aufgebaut ist und welche Unterschiede bestehen zwischen Klassenelementen und Instanzelementen;
- wie eine Klasse benutzt wird, um daraus Instanzen zu erzeugen und zu manipulieren;
- welche Bedeutung der Konstruktor hat und in welcher Reihenfolge die Initialisierung der Membervariablen erfolgt;
- dass es unterschiedliche Möglichkeiten gibt, wie Klassen sich gegenseitig benutzen und welche Bedeutung dabei die Vererbungstechnik hat;
- was man unter *override* von Methoden versteht;
- wann die Schlüsselwörter *this* und *super* benötigt werden;
- dass es weitere Sprachmittel gibt für die Beschreibung von Klassentypen, z.B. *interface* und *enum*.

### 11.1 Was steht in einer Klassenbeschreibung?

Eine Klasse enthält die Beschreibung für eine Menge von Objekten. Dazu werden die Daten (die "Felder") deklariert und der Ausführungscode zur Verarbeitung dieser Daten codiert. Eine Klasse hat in Java folgenden Aufbau:

```
class identifier {  
    // Class-Body, bestehend aus :  
        // Konstruktoren  
        // Instanz-/Klassen-Variablen und  
        // Instanz-/Klassen-Methoden  
}
```

Die Klasse besteht aus zwei Teilen, der Klassendeklaration und dem Klassenrumpf. Die Deklaration beginnt mit dem Schlüsselwort *class*. Danach folgt der Name der Klasse. Die Implementierung der Klasse, der Rumpf (body), ist in geschweiften Klammern zusammengefasst. Dort stehen die Elemente. Für die grafische Darstellung einer Klasse gibt es mehrere Möglichkeiten. Eine Variante ist die Darstellung als Rechteck (siehe hierzu auch Kapitel 12: UML Unified Modeling Language).



Abb. 11.1: Darstellung von Aufbau und Inhalt einer Klasse als UML-Diagramm

Das folgende Programm enthält eine Klassenbeschreibung für die Verarbeitung von Daten der Geschäftspartner einer Firma. Die Klasse ist sehr einfach und hat die Aufgabe, Daten zu speichern und zu verarbeiten, die sowohl für Kunden als auch Lieferanten benötigt werden.

Die Klasse enthält als **Attribute** (Felder, Variablen)

*name* (= für das Speichern des Partnernamens)  
*nr* (= für das Speichern der Identifikationsnummer).

Als Verarbeitungsmethoden für diese Daten sollen programmiert werden:

*neu* (= für das Anlegen eines neuen Geschäftspartners)  
*ausgeben* (= für die Anzeige der Daten eines bestimmten Geschäftspartners)  
*setName* (= für die Möglichkeit, den Namen zu ändern).

#### Programm *Partner01*: Klassenbeschreibung (ohne *main-Methode*)

```
class Partner01 {
    private int nr;
    private String name;

    void neu(int nr1, String name1) {
        nr = nr1;
        name = name1;
    }
    void ausgeben() {
```



```
        System.out.println(nr + " " + name);
    }
    void setName(String name1) {
        name = name1;
    }
}
```

Im Body werden die Elemente der Klasse (die "Member") definiert. Dazu gehören die Felder (Attribute) und die Methoden. Die Felder sind mit dem "access modifier" *private* deklariert, damit wird festgelegt, dass sie nur von Methoden dieser Klasse gelesen oder verändert werden können.

Die Reihenfolge der Elemente im Quelltext spielt keine Rolle. Die Felder könnten auch am Ende des Bodys aufgeführt werden, und auch die Reihenfolge der Methoden spielt keine Rolle - sie werden eh nur dann ausgeführt, wenn sie explizit aufgerufen werden.

### Übung zum Programm *Partner01*

Editieren und compilieren Sie das Programm. Die Umwandlung muss fehlerfrei möglich sein. Probieren Sie, ob dies Programm ausgeführt werden kann. Welche Fehlermeldung kommt beim Starten?

### Doughnut-Diagramm

Eine andere Art der Darstellung von Klassen, unabhängig von UML, ist das Kreisdiagramm (doughnut-diagram; donut, so heißen fettige englische, ringförmige Kuchenstücke). Die Klasse *Partner01* wird als Doughnut-Diagramm wie folgt dargestellt:

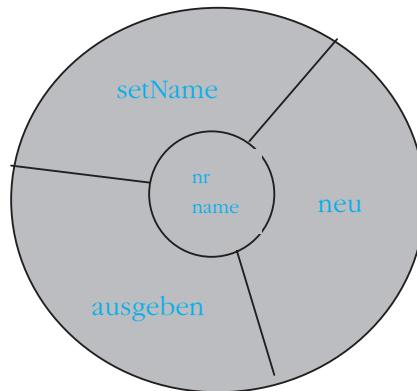


Abb. 11.2: Klasse *Partner01* als Doughnut-Diagramm

Diese Darstellung visualisiert die Kapselung der Daten: wie in einer Nuss sind die Attribute `nr` und `name` geschützt durch die Schale. Ein Zugriff darauf ist **nur** da-

durch möglich, dass ein Objekt erzeugt wird und dann eine Nachricht an das Objekt gesendet wird. Als mögliche Nachrichten sind vorgesehen: *ausgeben*, *neu* und *set-Name*. Ein direktes Lesen oder ein Ändern der Attribute ohne Aufruf dieser Methoden ist nicht möglich.

### Bewertung dieser Notation

Es demonstriert eindrucksvoll die Kapselung der privaten Attribute. Diese befinden sich innerhalb des Kekses und auf sie kann nur zugegriffen werden über die Methoden, die sie kapseln. Deshalb ist diese grafische Notation sehr gut geeignet für den Einstieg in die Denkweise der objektorientierten Programmierung. Nicht einsetzbar ist sie für die Darstellung von komplexen Zugriffsrechten oder für die Darstellung der Vererbungshierarchie.

## 11.2 Arbeiten mit Instanzen der Klassen

### Was ist eine Instanz?

Die objektorientierte Programmierung kennt als zentralen Begriff das "Objekt". Leider ist der Begriff nicht eindeutig definiert: meistens ist damit die konkrete Ausprägung einer Klasse (also eine "Instanz") gemeint; in manchen Zusammenhängen wird der Begriff aber auch als Synonym für eine Klasse benutzt. Deswegen ist es besser, entweder den Begriff "Klasse" zu benutzen, wenn die allgemeine Beschreibung einer Gruppe von Variablen und die damit verbundenen Methoden gemeint ist, oder den Begriff "Instanz" zu verwenden, wenn damit ein konkreter Einzelfall gemeint ist, der im Arbeitsspeicher zur Laufzeit eines Programms mit *new* erzeugt worden ist. Eine Instanz belegt Speicherplatz, der mit den individuellen Werten dieses Exemplars gefüllt ist.

Instanzen werden zur Laufzeit eines Programms erzeugt durch:

```
Klassenname instanzname;  
instanzname = new Klassenname();
```

Beispiel für das Erzeugen einer Instanz von der Klasse *Partner*:

```
Partner01 g1 = new Partner01();
```

Die so erzeugte Instanz *g1* kann benutzt werden, um mit den Attributen und den Methoden der Klasse zu arbeiten. Der Aufruf von Methoden geschieht durch das Senden von Nachrichten in der Form

```
instanzname.methodenname(parameter);
```

Beispiel für das Senden einer Nachricht:

```
g1.ausgeben();
```

Die nachfolgende Klasse enthält ein ausführbares Programm. Sie benutzt die Klasse *Partner01* als Schablone zum Erzeugen eines konkreten Objekts, um anschließend damit zu arbeiten.

### Programm *PartnerTest01*: Eine Instanz erzeugen und Nachrichten schicken

```
class PartnerTest01 {  
    public static void main(String[] args) {  
        Partner01 g1 = new Partner01();  
        g1.neu(4700, "Meyer");  
        g1.ausgeben();  
    }  
}
```

### Übung zum Programm *PartnerTest01*

Überprüfen Sie, welche Fehlermeldung bei der Umwandlung ausgegeben wird, wenn die Parameter (4700, Meyer) in umgekehrter Reihenfolge übergeben werden.

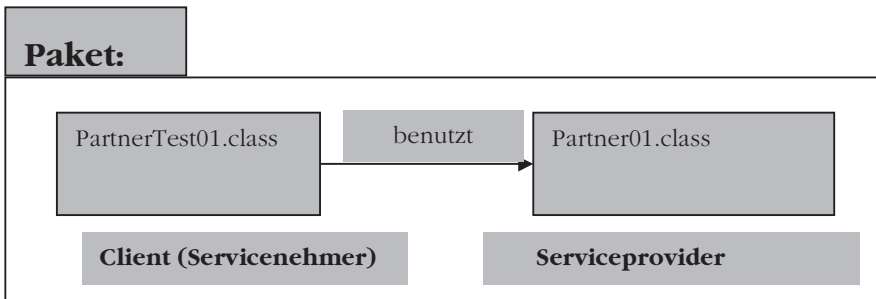


Abb. 11.3: Beziehung zwischen den beiden Klassen

Klassen sind das fundamentale Strukturelement in Java. Alle Methoden und Datendefinitionen stehen in Klassen. Sie sind die kleinste Einheit, die umgewandelt werden kann. Aber die Klassen werden organisiert in Paketen (siehe Kapitel 16). Beide Klassendateien unseres Beispiels gehören zu einem *Package*, und beide müssen im selben Verzeichnis stehen. Obwohl keine Paketzugehörigkeit explizit angegeben wurde, gehören sie zu *einem* (namenlosen) Default-Paket. Innerhalb eines Pakets ist jede Methode berechtigt, auf jede andere Methode oder auf jedes andere Datenfeld einer Klasse zuzugreifen.

Das folgende Beispielprogramm *Partner02.java* realisiert dieselbe Aufgabenstellung wie die Programme *Partner01* und *PartnerTest01*, nur dass jetzt die beiden Programme zusammen gefasst sind in einer Quelltextdatei (= eine Umwandlungseinheit). Das Programm *Partner02* enthält also nicht nur die Beschreibung der Klasse (die Schablone), sondern es ist auch gleichzeitig ein ausführbares Programm, das diese Schablone benutzt, denn es enthält die Methode *main*. In dieser *main*-Methode wird mit *new* eine Instanz erstellt von der eigenen Klasse. Die Instanz hat den Bezeichner *g1* - und mit Hilfe dieses Bezeichners kann das Objekt bearbeitet werden.

**Programm *Partner02*: Klasse mit Elementen und einer *main*-Methode**

```
public class Partner02 {
    public static void main(String[] args) {
        Partner02 g1 = new Partner02();
        g1.neu(4700, "Meyer");
        g1.ausgeben();
    }
    private int nr;
    private String name;
    void neu(int nr1, String name1) {
        nr = nr1;
        name = name1;
    }
    void ausgeben() {
        System.out.println(nr + " " + name);
    }
    void setName(String name1) {
        name = name1;
    }
}
```

**Übung zum Programm *Partner02***

Ergänzen Sie das Programm *Partner02* so, dass eine zweite Instanz erzeugt wird. Der Instanzname soll sein: g2. Die Werte sind 4001 für die Kundennummer und "Schulz" für den Namen. Danach geben Sie bitte zuerst die Instanz g2 und danach die Instanz g1 am Bildschirm aus.

## 11.3 Mitgelieferte Klasse benutzen

Das JDK von Sun (Java 2 Standard Edition) enthält eine Fülle von eingebauten Klassen. Diese sind ausführlich dokumentiert in der mitgelieferten API-Dokumentation.

**Arbeiten mit JOE:** Aus dem JOE-Editor heraus ist die Dokumentation direkt über einen Menüpunkt erreichbar: durch ? (Hilfe) und dann unter "JDK Dokumentation".

Die API-Spezifikation ist organisiert nach Packages. Entweder wählt man dann auf der linken Seite gezielt ein Paket aus und lässt sich alle Klassen in diesem Paket anzeigen oder man bekommt alle Packages mit allen Klassen, alphabetisch sortiert, angezeigt. Die Standard Edition 5.0 enthält etwa 5000 verschiedene Klassen, jede einzelne wird dokumentiert mit ihrem vollen Namen, ihrer Paketzugehörigkeit, ihrer Einordnung in die Vererbungshierarchie und mit allen Elementen, die diese Klasse enthält. Wir werden in den folgenden Abschnitten drei dieser mitgelieferten Klassen benutzen und dabei auch auf die Dokumentation zurückgreifen.

### 11.3.1 Beispiel 1: Arbeiten mit der Klasse *GregorianCalendar*

Allein für das Arbeiten mit Datum und Uhrzeit gibt es zahlreiche vorgefertigte Lösungen, abhängig von Zeitzonen und lokalen Besonderheiten. Sie sind Teil des Standard-API. Da es weltweit unterschiedliche Kalender gibt, die u.a. abhängig sind vom Beginn der Zeitrechnung, haben die Java-Entwickler eine Spezialisierung eingeführt, nämlich den *GregorianCalendar*. Die Dokumentation enthält dafür eine ausführliche Beschreibung. Hier ein Ausschnitt:

```
java.util Class GregorianCalendar
```

#### Field Summary

static int	<a href="#">AD</a>	Value of the <code>ERA</code> field indicating the common era (Anno Domini), also known as CE.
static int	<a href="#">BC</a>	Value of the <code>ERA</code> field indicating the period before the common era (before Christ), also known as BCE.

`GregorianCalendar` is a concrete subclass of `Calendar` and provides the standard calendar system used by most of the world.

#### Constructor Summary

[GregorianCalendar](#)() Constructs a default `GregorianCalendar` using the current time in the default time zone with the default locale.

[GregorianCalendar](#)(int year, int month, int dayOfMonth)  
Constructs a `GregorianCalendar` with the given date set in the default time zone with the default locale

#### Method Summary

void [add](#)(int field, int amount) Adds the specified (signed) amount of time to the given calendar field, based on the calendar's rules.

[Object](#) [clone](#)() Creates and returns a copy of this object.

protected void [computeFields](#)() Converts the time value (millisecond offset from the [Epoch](#)) to calendar field values.

**Methods inherited from class java.util.[Calendar](#):**

```
int get(int field)
    Returns the value of the given calendar field.
```

Abb.11.4: API-Dokumentation der Klasse *GregorianCalendar*

Die Elemente der Klasse werden in drei Abschnitten erläutert:

**Field Summary**

dokumentiert die Deklaration der Instanz- und Klassenvariablen, z.B. *int AD* oder *BC*. Die Bedeutung der Elemente kann man aus dem Kurztext erfahren oder - etwas ausführlicher - indem man auf den Namen klickt, im Abschnitt "Field Details".

**Konstruktor Summary**

dokumentiert die Konstruktoren dieser Klasse, indem der Methodenkopf beschrieben wird, z.B. *GregorianCalendar()*. Zusätzlich gibt es eine Kurzbeschreibung oder, durch Anklicken des Konstruktornamens, Details zu dem Konstruktor.

**Method Summary**

Der dritte wichtige Abschnitt enthält die Beschreibung der Methoden dieser Klasse. Zunächst wird jede Methode mit ihrem Methodenkopf und einer Kurzbeschreibung aufgeführt. Der Methodenkopf enthält die Signatur, die Parameternamen und den Datentyp des Rückgabewertes. Beispiel:

```
void add(int field, int amount)
```

Der Name dieser Methode ist *add*. Sie erwartet zwei Parameter vom Datentyp *int*. Die Werte dafür stehen innerhalb der Methode unter den Bezeichnern *field* und *amount* zur Verfügung. Der Aufrufer dieser Methode bekommt kein Ergebnis zurückgeliefert. Durch Anklicken des Methodennamens bekommt man Details zur Methode beschrieben.

**Programm *Gregorian01*: Anwendungsbeispiel (Client) für die Klasse *GregorianCalendar***

```
import java.util.*;
public class Gregorian01 {
    public static void main(String[] args) {
        GregorianCalendar heute = new GregorianCalendar();
        System.out.println(heute.get(Calendar.DAY_OF_MONTH));
        System.out.println(heute.get(Calendar.MONTH));
        System.out.println(heute.get(Calendar.YEAR));
    }
}
```

### Übung zum Programm Gregorian01

Ändern Sie das Programm so, dass auch die Uhrzeit ausgegeben wird. Dazu muss ebenfalls die Methode *get* aufgerufen werden, nun mit den folgenden eingebauten Konstanten als Parameter: `Calendar.HOUR`, `Calendar.MINUTE`, `Calendar.SECOND`.

#### 11.3.2 Beispiel 2: Klasse *Point*

Die Standard-Klasse *Point* ist Teil des mitgelieferten Pakets *java.awt*. Sie enthält die beiden Felder *x* und *y*, beide vom Datentyp *int*. Als Methoden werden von ihr u.a. zur Verfügung gestellt: *equals* und *toString*.

#### Programm *Punkt01*: Anwendungsbeispiel für die Klasse *java.awt.Point*

```
import java.awt.*;
public class Punkt01 {
    public static void main(String[] args) {
        Point p = new Point(5,3);
        System.out.println(p);
    }
}
```

Die Ausgabe sieht wie folgt aus: `java.awt.Point [x=5,y=3]`

Zunächst wird der volle Name der Klasse und dann Name und Inhalt der Felder ausgegeben. Wie kommt es zu dieser übersichtlichen Darstellung? Dafür sorgt ein eingebauter Mechanismus der Methode *println*. Diese ruft automatisch die Methode *toString* auf. Und genau die sorgt dafür, dass die Run-Time-Class der Instanz und der Inhalt der Membervariablen als String ausgegeben werden.

#### Übungen zum Programm *Punkt01*

**Übung 1:** Bitte prüfen Sie anhand der API-Dokumentation die Beschreibung der Klasse *Point*, und dort insbesondere die Methoden *equals* und *toString*.

**Übung 2:** Modifizieren Sie das Programm so, dass eine zweite Instanz von *Point* erstellt wird und dass durch Aufruf einer geeigneten Methode festgestellt wird, ob die beiden Punkte gleich sind oder nicht.

#### Lösungsvorschlag

```
import java.awt.*;
public class Punkt02 {
    public static void main(String[] args) {
        Point p1 = new Point(5,3);
        Point p2 = new Point(4,5);
        System.out.println(p1.equals(p2));
    }
}
```

### Übung zum Programm *Punkt02*

**Übung 1:** Verschieben Sie den Punkt *p1* auf *x* = 10 und *y* = 15. Benutzen Sie dazu die Methode *move*. Geben Sie die neuen Objektwerte aus. Benutzen Sie dazu die Methode *getLocation*.

**Übung 2:** Erhöhen Sie den *x*-Wert um 5 und den *y*-Wert um 20. Benutzen Sie dazu die Methode *translate*.

### 11.3.3 Beispiel 3: Klasse *Color*

Auch die Klasse *Color* ist Teil des *java.awt*-Packages. Sie kapselt Farbangaben nach dem RGB-Farbmodell.

Jede Farbe wird dabei aus einer Mischung von Rot, Grün und Blau beschrieben, wobei jede Farbe einen Anteil von 0 - 255 haben kann (die Mischung kann auch prozentual als float-Wert zwischen 0.0 und 1.0 angegeben werden). So ergeben z.B. die Werte 255 jeweils für Rot und Grün, kombiniert mit dem Wert 0 für Blau, die Farbe gelb.

Wir wollen ein Programm erstellen, das mehrere *Color*-Objekte mit unterschiedlichen verschiedenen Farbeinstellungen erstellt. Diese Objekte könnten dann beispielsweise für das Arbeiten mit grafischen Benutzeroberflächen genutzt werden.

#### Programm *Farben01*: Die Klasse *java.awt.Color*

```
import java.awt.*;
class Farben01 {
    public static void main (String[] args) {
        Color c1 = new Color(255,0,0);
        Color c2 = c1.darker();
        System.out.println("Farbe 1 " + c1);
        System.out.println("Farbe 2 " + c2);
        System.out.println(c1.getRed());
    }
}
```

Das erste Statement in der *main*-Methode erzeugt eine Instanz der Klasse *Color*. Dabei werden dem Konstruktor die drei Farbwerte für rot, grün und blau übergeben.

### Übungen zum Programm *Farben01*

**Übung 1:** Klären Sie anhand der API-Dokumentation die Arbeitsweise der Methoden *darker* und *getRed*.

**Übung 2:** Instanziiieren Sie ein drittes *Color*-Objekt für die Farbe gelb und geben Sie die Objektwerte mit *println* aus.



## 11.4 Eigene Klassen erstellen

Als Synonym für eine Klasse kann der Begriff "selbst erstellter Datentyp" hilfreich sein. Die Klasse beschreibt den Aufbau von Speicherplätzen (Felder) und die dafür möglichen Operationen (Methoden) - genau so wie es bei den eingebauten einfachen Datentypen auch der Fall ist. Wenn die in der Klasse deklarierten Speicherplätze im Arbeitsspeicher angelegt und mit Werten gefüllt werden sollen, dann geschieht dies durch das Erzeugen von Instanzen - mit dem Schlüsselwort *new*. Wir werden nun eigene Klassen beschreiben und damit arbeiten.

### 11.4.1 Beispiel für ein "Hallo Welt"- Programm als separate Klasse

#### Programm *Hallo01*: Eine ganz einfache Klasse

```
public class Hallo01 {  
    private String text = new String("Hallo Welt");  
    void ausgeben() {  
        System.out.print("Die Variable text enthaelt: ");  
        System.out.println(text);  
    }  
}
```

Diese Klasse kann eine *String*-Variable speichern und bei Bedarf (durch Aufruf der Methode *ausgeben*) am Konsolbildschirm wieder ausgeben. Nur zur Erinnerung: *String* ist ebenfalls eine Klasse. Sie wird als Teil des Standard-API mitgeliefert.

#### Übung zum Programm *Hallo01*

Wandeln Sie das Programm um. Dies muss fehlerfrei möglich sein. Versuchen Sie danach, das Programm zu starten. Es kommt die Meldung "NoSuchMethodError: main".

Um den Service der Klasse *Hallo01* nutzen zu können, benötigen wir ein ausführbares Programm, das eine Instanz erzeugt und eine Nachricht an diese Instanz schickt.

#### Programm *HalloClient01*: Benutzen der selbst erstellten Klasse *Hallo01*

```
public class HalloClient01 {  
    public static void main(String[] args) {  
        Hallo01 k = new Hallo01();  
        k.ausgeben();  
    }  
}
```

#### Übungen zum Programm *HalloClient01*

**Übung 1:** Wandeln Sie dieses Programm um. Eine mögliche Fehlerquelle ist, dass bei der Umwandlung die Klassenbeschreibung von *Hallo01* nicht im Zugriff ist. Stel-

len Sie deshalb sicher, dass sich beide Programme in demselben Verzeichnis befinden. (Hinweis für Fortgeschrittene: Stehen die Dateien in unterschiedlichen Ordnern, so muss über die *Classpath*-Variable der Suchpfad entsprechend ergänzt werden.)

**Übung 2:** Machen Sie folgenden Versuch: Löschen Sie die Class-Datei *Hallo01.class*, (nicht die Quelldatei *.java*!) und wandeln Sie das Programm *HalloClient01.java* erneut um. Beantworten Sie folgende Fragen:

- Ist die Umwandlung fehlerfrei möglich, obwohl die Klasse, die benutzt werden soll, nicht im Java-Byteformat vorliegt?
- Wenn ja, prüfen Sie, ob diese Class-File etwa automatisch neu erstellt worden ist.

**Übung 3:** Versuchen Sie danach, die beiden Quell-Programme (*Hallo01.java* und *HalloClient01.java*) in einer Quelltextdatei zusammen zu fassen. Eine Frage, die dabei geklärt werden muss, ist: Wie muss der Name der Quellendatei lauten?

### Lösungshinweise

- Wenn in einer Umwandlungseinheit mehrere Klassen stehen, so werden sie gemeinsam umgewandelt. Durch die Compilierung wird pro Klasse eine separate Class-File erzeugt; nur eine davon kann eine Java-Applikation sein, die zur Ausführung aufgerufen werden kann. Und das ist auch die Klasse, die den Namen der Umwandlungseinheit bestimmt.
- Wenn in einer Umwandlungseinheit mehr als eine Klasse stehen, darf nur eine davon das Schlüsselwort *public* haben.

**Fazit:** Eine Java-Applikation besteht aus einer Hauptklasse und beliebig vielen Nebenklassen. Die Hauptklasse muss die *main*-Methode enthalten und *public* sein. Zur Umwandlungszeit müssen alle Nebenklassen im Byte-Format im Zugriff sein, entweder im selben Verzeichnispfad oder über *Classpath*-Referenzen.

Natürlich müssen auch zur Ausführungszeit alle Klassen im Zugriff sein. Gestartet wird aber zunächst nur die Hauptklasse, alle anderen werden erst bei Bedarf in den Arbeitsspeicher übertragen.

### 11.4.2 Beispiel zum Prüfen einer Ganzzahl auf gerade/ungerade

Die Aufgabe für das folgende Programm lautet: Bitte erstellen Sie eine Klasse *Pruefen01.java*. Diese soll die Fähigkeit haben, einen Integer-Wert zu speichern, und sie soll die Methode *pruefInt* enthalten, in der geprüft wird, ob dieser Integerwert eine gerade oder eine ungerade Zahl ist. Die Methode liefert *true* oder *false* an den Aufrufer zurück.

Der Wert der *int*-Variablen wird als Initialwert beim Erzeugen der Instanz vergeben.

### Programm *Pruefen01*: Klasse, die einen Integerwert speichert und überprüft

```
class Pruefen01 {
    private int zahl = 5;
    boolean pruefInt() {
        if ((zahl % 2) == 0)
            return true;
        else
            return false;
    }
}
```

### Übung zum Programm *Pruefen01*

Erstellen Sie ein ausführbares Programm, das eine Instanz von *Pruefen01* erstellt und an diese Instanz eine Nachricht schickt. Abhängig vom Ergebnis dieser Nachricht wird der Bediener informiert darüber, ob die Zahl gerade oder ungerade ist.

### Lösungsvorschlag

```
public class PruefenClient01 {
    public static void main(String[] args) {
        Pruefen01 p = new Pruefen01();
        if (p.pruefInt())
            System.out.println("Die Zahl ist gerade");
        else
            System.out.println("Die Zahl ist ungerade");
    }
}
```

### Übung zum Programm *Pruefen01*

Ändern Sie die Klasse *Pruefen01.java* so ab, dass das Feld *zahl* nicht explizit initialisiert wird. Was liefert jetzt die Methode *pruefInt*? Die Erkenntnis wird sein, dass Membervariablen automatisch initialisiert werden. Wie lautet der Anfangswert einer *int*-Variablen, wenn nichts anderes vom Programmierer vorgegeben wird?

Fazit: Jedes Mal, wenn mit *new* eine Instanz im Speicher angelegt wird, erzeugt das Runtime-System einen komplett neuen Satz der Instanzvariablen dieser Klasse. Dabei werden diese Variablen mit Initialwerten vorbelegt: numerische Werte werden mit Nullen gefüllt, *char*- oder *String*-Variablen werden mit Blank gefüllt.

Zur Erinnerung: lokale Variable werden nicht automatisch initialisiert, dafür ist der Programmierer zuständig.

Es ist auch möglich, bei der Initialisierung nicht mit den Default-Werten zu arbeiten, sondern bei der Variablendefinition entsprechende Initialwerte anzugeben. In beiden Fällen arbeiten jedoch alle Instanzen mit denselben Anfangswerten.

Sollen die Membervariablen mit individuellen Werten belegt werden, so kann dies durch Codieren von entsprechenden **Konstruktoren** realisiert werden. Das werden wir im übernächsten Beispiel ("Zeitangaben") demonstrieren.

### 11.4.3 Beispiel zum Arbeiten mit Datumsangaben

Für die nachfolgenden Beispiele erstellen wir eine weitere komplett neue Klasse, die Klasse *Datum01*. Sie enthält die Felder *tag*, *monat* und *jahr*. Außerdem hat diese Klasse als weitere Elemente einige Methoden, um mit diesen Attributen zu arbeiten.

#### Programm *Datum01*: Datum speichern und verarbeiten

```
import java.util.*;
public class Datum01 {

    private int tag;
    private int monat;
    private int jahr;

    void erstellen() {
        Calendar cal = new GregorianCalendar();
        tag = cal.get(Calendar.DATE);
        monat = (cal.get(Calendar.MONTH) + 1);
        jahr = cal.get(Calendar.YEAR);
    }

    void ausgeben() {
        char c = '.';
        System.out.printf("%s%s%s%s",
                           tag, c, monat, c, jahr);
    }
}
```

#### Erläuterungen zur Klasse *Datum01*

Die Klasse kann Datumsangaben speichern, und sie enthält Methoden, um damit zu arbeiten. Zum Erstellen des aktuellen Tagesdatum greift sie zurück auf Dienstleistungen der Standardklassen *Calendar* und *GregorianCalendar*, indem sie eine Instanz der Klasse *GregorianCalendar* erzeugt und dann an diese Instanz die Nachricht *get()* schickt, um den aktuellen Tag, Monat und das Jahr zu lesen.

Als Parameter benutzt die *get*-Methode *static*-Felder, die lediglich sprechende Namen bilden für ganzzahlige Feldnummern, die für das Lesen der Datumsangaben benötigt werden. Weil die Monatsangabe bei 0 beginnt, muss sie für die Ausgabe um 1 erhöht werden.

Die Reihenfolge der Elementdefinition in einer Klasse ist ohne Bedeutung. Die Klasse wird eingelesen (aktiviert), dann stehen alle *static*-Felder zur Verfügung, egal, an welcher Stelle im Quelltext sie definiert sind. Alle anderen Felder stehen zur Verfügung nach der Instanzerzeugung. Und Methoden werden nur ausgeführt, wenn sie explizit aufgerufen werden.

### Die Klasse *Datum01* benutzen

Klassen stellen Services zur Verfügung, die von ausführbaren Programmen oder von anderen Klassen genutzt werden können. Noch fehlt dieses ausführbare Programm.

Das nachfolgende Programm *DatumClient01.java* soll die neu erstellte Klasse *Datum01* benutzen, um das aktuelle Tagesdatum von heute zu erzeugen und in einem Consolefenster auszugeben. Erstellen Sie bitte eine neue Umwandlungseinheit.

### Programm *DatumClient01*: Client-Programm für die *Datum*-Klasse

```
public class DatumClient01 {
    public static void main(String[] args) {
        Datum01 heute = new Datum01();
        heute.erstellen();
        heute.ausgeben();
    }
}
```

## 11.4.4 Beispiel zum Arbeiten mit Zeitangaben

Das nächste Beispiel beschreibt eine Klasse, die folgende Dienstleistungen erbringt:

- sie speichert die Stunden, Minuten und Sekunden einer bestimmten Uhrzeit,
- sie kann eine beliebige Stundenzahl addieren und
- sie kann die gespeicherte Zeit als *String* aufbereiten und diese Zeichenkette dann dem Aufrufer der entsprechenden Methode zur Verfügung stellen.

### Programm *Zeit*: Zeitangaben speichern und verarbeiten

```
public class Zeit {
    private int stunde;
    private int minute;
    private int sekunde;
    Zeit(int stunde, int minute) {
        this.stunde = stunde;
        this.minute = minute;
        this.sekunde = 0;
    }
    Zeit(int stunde, int minute, int sekunde) {
```

```
        this.stunde = stunde;
        this.minute = minute;
        this.sekunde = sekunde;
    }
    void addStunde(int st) {
        stunde = stunde + st;
        if (stunde > 24)
            stunde = stunde - 24;
    }
    public String toString() {
        String s1 = "Stunde: " + stunde + "\n";
        String s2 = "Minute: " + minute + "\n";
        String s3 = "Sekunde: " + sekunde + "\n";
        return s1 + s2 + s3;
    }
}
```

Die Klasse enthält einige Sprachmittel, die wir bisher nur am Rande erläutert haben. Es handelt sich um folgende Themen:

- Konstruktoren
- Overloading von Methoden und Konstruktoren
- Override von Methoden
- Arbeiten mit dem Schlüsselwort *this*

### Konstruktoren

Die ersten beiden Methoden dieser Klasse werden auch Konstruktoren genannt. Konstruktoren werden automatisch aufgerufen, wenn eine neue Instanz der Klasse erzeugt wird. Sie enthalten Statements zum Initialisieren der Felder. Formal unterscheiden sich diese Konstruktoren von "normalen" Methoden wie folgt: sie müssen denselben Namen wie die Klasse haben und sie dürfen keinen Rückgabebetyp deklarieren (auch das Schlüsselwort *void* darf bei Konstruktoren nicht angegeben werden).

### Überladen ("overloading") von Methoden

Die zweite Neuerung ist, dass der Konstruktorenname mehr als einmal verwendet wird. Es gibt einen Konstruktor, der zwei Parameter empfängt, und einen weiteren Konstruktor, der drei Parameter erwartet. Man bezeichnet diese Technik als Überladen von Konstruktoren. Dies ist auch bei "normalen" Methoden möglich. Methoden-namen und Konstruktornamen können also innerhalb einer Klassenhierarchie und auch sogar innerhalb einer einzigen Klasse mehrfach vorkommen. Dann müssen sie allerdings eine unterschiedliche Anzahl und/oder unterschiedliche Datentypen bei

den Parametern haben (also eine andere "Signatur" besitzen). Der Rückgabewert gehört nicht zur Signatur.

Der Einsatz dieser Technik ist immer dann sinnvoll, wenn vergleichbare Funktionalitäten mehrfach implementiert werden müssen, aber mit unterschiedlichen Parameterlisten. Beispiel: Verschiedene grafische Objekte sollen unterschiedlich gezeichnet. Dann können Sie in einer Klasse mehrere Methoden implementieren, alle mit demselben (aussagefähigen) Namen. Diese Zeichnen-Methoden müssen allerdings unterschiedliche Typen von Parametern definieren.

Die JVM entscheidet zur Ausführungszeit anhand der passenden Argumente, welche Version aufzurufen ist.

### Überschreiben ("override") von Methoden

Eine spezielle Form der generellen Technik des Overloading ist Override (überschreiben). Override (engl. für "sich hinwegsetzen") ist nur möglich in Verbindung mit der Vererbung, wenn in der Unterklasse eine geerbte Methode re-definiert, also neu geschrieben wird. Dabei kann dieselbe Signatur verwendet werden.

Die JVM entscheidet zur Ausführungszeit anhand des Objekttyps, welche Methode ausgeführt werden soll. Dieses Verfahren wird immer dann eingesetzt, wenn die Unterklasse eine Methode der Oberklasse ändern will.

In der objektorientierten Sprache Java hat jede Klasse eine Superklasse, von der sie erbt. Wenn der Programmierer nicht ausdrücklich den Namen einer Superklasse angibt, dann erbt die Klasse implizit von der Superklasse *Object*. Dies ist "die Mutter aller Klassen" - alle anderen Klassen sind direkt oder indirekt davon abgeleitet. Das Thema "Vererbung" wird auf den nächsten Seiten ausführlich besprochen.

In der API-Dokumentation für die Klasse *Object* ist beschrieben, dass sie eine Methode *toString()* zur Verfügung stellt, die dafür sorgt, dass einfache Datentypen in Texte (also in den Datentyp *String*) umgewandelt werden. Einige mitgelieferte Methoden rufen diese *toString*-Methode automatisch auf, ohne dass dies explizit vom Programmierer codiert werden muss (so arbeitet beispielsweise auch die *println*-Methode).

Diesen Mechanismus nutzt die Klasse *Zeit*. Sie enthält nämlich selbst eine Methode *toString* - und damit wird die gleichnamige Methode in der Superklasse überschrieben. Die selbst geschriebene Methode *toString* bereitet die einzelnen Felder der Klasse als Text auf und liefert diesen String als Ergebnis an den Aufrufer zurück.

**Merksatz:** Die JVM sucht die passende Methode in folgender Reihenfolge: Zunächst wird eine Methode mit der gleichen Signatur in der Klasse des Objekts gesucht (also in der eigenen Klasse). Wenn sie dort nicht fündig wird, geht die Suche in der übergeordneten Klasse weiter, bis hoch zur Klasse *Object*, also bis die gesamte Hierarchie durchsucht worden ist.

**this-Schlüsselwort**

Innerhalb einer Methode kann man direkt auf die Membervariablen der eigenen Klasse zugreifen. Adressiert werden sie über den einfachen Namen (Bezeichner). Wenn jedoch innerhalb der Methode derselbe Name bereits für lokale Variable benutzt wurde (z.B. durch formale Parameter, wie im Programm *Zeit*), kommt es zu Namenskonflikten. In Java werden dann die Membervariablen verdeckt, sie werden überlagert von den lokalen Variablen. Ein Zugriff auf die Membervariablen ist dann nur möglich, wenn diese mit dem Schlüsselwort *this* qualifiziert werden.

Mit *this* referenziert man innerhalb einer Klasse die aktuelle Instanz. Anders gesagt: hinter *this* steckt das Objekt, mit dem gerade gearbeitet wird.

**Übung zum Programm *Zeit***

Bitte klären Sie durch Selbsttest, welche Fehlermeldung bei der Umwandlung kommt, wenn das Schlüsselwort *public* bei der *toString*-Methode entfernt wird.

Lösungshinweis: Sinngemäß bedeutet die Meldung, dass versucht wird, eine Methode zu überschreiben und dabei die Zugriffsrechte einzuschränken. Das ist nicht erlaubt. Weil die Methode *toString* in der Klasse *Object* das Zugriffsrecht *public* hat, muss sie in den davon abgeleiteten Klassen auch *public* sein.

**Programm *ZeitTest*: Arbeiten mit der neu erstellten Klasse *Zeit***

```
import java.util.*;
public class ZeitTest {
    public static void main(String[] args) {
        // selbst erstellte Klasse benutzen
        Zeit z = new Zeit(7, 25);
        z.addStunde(2);
        System.out.println(z.toString());
        // mitgelieferte Klasse benutzen
        GregorianCalendar heute = new GregorianCalendar();
        int st = heute.get(Calendar.HOUR);
        int m = heute.get(Calendar.MINUTE);
        int se = heute.get(Calendar.SECOND);
        Zeit z2 = new Zeit(st, m, se);
        System.out.println(z2.toString());
    }
}
```

**11.5 Konstruktoren**

Instanzen werden mit Hilfe des Schlüsselwortes *new* erzeugt. Durch *new* wird auch eine so genannte Konstruktormethode automatisch aufgerufen. Das ist eine Methode, welche denselben Namen wie die Klasse selbst hat. Die Aufgabe dieser Kon-



struktormethode ist es, einmalige Arbeiten durchzuführen, die beim Erstellen eines Objekts sinnvoll bzw. notwendig sind. Ihre wichtigste Aufgabe ist es, die Felder der Klasse mit individuellen Anfangswerten zu versehen.

### 11.5.1 Standardkonstruktoren: selbst codieren?

Wenn der Programmierer keinen Konstruktor selbst codiert, so wird ein eingebauter Standard-Konstruktor ausgeführt. Der Standardkonstruktor ("Default-Konstruktor") ist dadurch gekennzeichnet, dass er keine Parameter empfängt.

#### Programm *Init01*: Arbeiten mit dem Default-Konstruktor

```
public class Init01 {
    int zahl1;
    char c;
    float zahl2;
    public static void main(String[] args) {
        Init01 instanz1 = new Init01();
        System.out.println(instanz1.zahl1);
        System.out.println(instanz1.zahl2);
    }
}
```

Bei der Ausführung dieses Programms wird der (unsichtbare) Standardkonstruktor ausgeführt. Im nächsten Beispiel ist ein Konstruktor ausdrücklich programmiert.

#### Programm *Init02*: Arbeiten mit einem selbst codierten Konstruktor

```
public class Init02 {
    int zahl1;
    char c;
    float zahl2;

    Init02(int z1, char c, int z2) {
        zahl1 = z1;
        this.c = c;
        zahl2 = z2;
    }

    public static void main(String[] args) {
        Init02 instanz1 = new Init02(15, 'a', 27);
        System.out.println(instanz1.zahl1);
        System.out.println(instanz1.c);
        System.out.println(instanz1.zahl2);
    }
}
```

Der Konstruktor muss den Namen der Klasse haben (*Init02*). Er kann beliebige Parameter empfangen, in diesem Beispiel werden Argumente für die drei Felder der Klasse entgegen genommen. Natürlich ist hier auch die Reihenfolge sehr entscheidend, denn Java arbeitet immer mit Positionsparameter: beim Aufruf des Konstruktors muss man wissen, an welcher Position welcher Wert erwartet wird. Die Namen sind nicht ausschlaggebend. Die Parameterangaben im Kopf des Konstruktors wirken wie die Definition von lokalen Parametern.

### Übung 1 zum Programm *Init02*

Versuchen Sie, in der *main*-Methode eine weitere Instanz zu erzeugen, diesmal allerdings ohne Parameterübergabe. Dadurch wird der Default-Konstruktor aufgerufen. Aber die Umwandlung erzeugt folgenden Fehler:

```
"... cannot find symbol ... constructor Init02() ... "
```

Der Grund dafür ist: Wenn vom Programmierer ein Konstruktor programmiert wird, dann wird der Default-Konstruktor nicht mehr automatisch erstellt, d.h. dann muss ein Konstruktor ohne Parameter auch vom Programmierer selbst codiert werden. Bitte ergänzen Sie das Programm entsprechend.

Regel: Entweder man definiert keinen Konstruktor oder man definiert explizit alle, also auch den Standardkonstruktor. Denn sobald eine Klasse (irgend-)einen Konstruktor enthält, wird der Standardkonstruktor auch nicht mehr von Java erzeugt.

Der Name des ersten Parameters ist in unserem Beispielprogramm *z1*, er steht für den Anfangswert der Membervariablen *zahl1*. Deswegen ist der erste Befehl im Rumpf des Konstruktors: *zahl1 = z1*. Bei dem zweiten Argument haben wir ein Problem zu lösen, denn der Name des Parameters ist mit *c* derselbe wie der Name der Membervariablen. Deswegen muss die Wertezuweisung mit einem qualifizierten Namen erfolgen - und das geschieht in diesem Fall mit dem Schlüsselwort *this*.

### Übung 2 zum Programm *Init02*

Überlegen Sie, warum es nicht sinnvoll ist, in dem Konstruktor des Programms die Referenzvariable *instanz1* anstelle des Schlüsselwortes *this* zu benutzen. Überprüfen Sie, ob dann überhaupt eine Umwandlung möglich ist.

## 11.5.2 Überladen von Konstruktoren

Im nächsten Beispiel wird noch einmal mit der bereits bekannten Klasse *Datum* gearbeitet. Diese Klasse ist eine Ergänzung zur Standardklasse *GregorianCalendar*. Diese Standardklasse ist ungeheuer umfangreich und leistungsfähig, wir benötigen aber nur einen kleinen Teil des gesamten Leistungsspektrums, der zudem noch auf unsere individuellen Bedürfnisse angepasst werden soll. Deswegen hatten wir eine neue Klasse *Datum01* eingeführt. Diese soll nachfolgend modifiziert und um Konstruktoren ergänzt werden.

### Programm *Datum*: Zwei Konstruktoren in einer Klasse

```
import java.util.*;
public class Datum {
    int tag;
    int monat;
    int jahr;

    Datum() {
        Calendar cal = new GregorianCalendar();
        tag = cal.get(Calendar.DATE);
        monat = (cal.get(Calendar.MONTH) + 1);
        jahr = cal.get(Calendar.YEAR);
    }
    Datum(int t, int m, int j) {
        tag = t;
        monat = m;
        jahr = j;
    }
    void ausgeben(char zeichen) {
        char c = zeichen;
        System.out.printf("%s %s %s %s %s",
                           tag, c, monat, c, jahr);
    }
}
```

Diese Klasse hat zwei Konstruktoren, der erste Konstruktor ist ein Standardkonstruktor, d.h. er erwartet keine Argumente. Für den Aufruf des zweiten Konstruktors sind drei *int*-Parameter notwendig. Java wählt zur Ausführungszeit den Konstruktor aus, der passend ist. Und passend ist der Konstruktor dann, wenn Anzahl und Typ der übergebenen Argumente übereinstimmt mit der deklarierten Anzahl und dem Typ. Man sagt, Java nimmt den Konstruktor, der die erwartete Signatur hat.

### Programm *DatumTest01*: Instanziiieren und Aufrufen des "richtigen" Konstruktors

```
public class DatumTest01 {
    public static void main(String[] args) {
        Datum d1 = new Datum(24, 12, 2005);
        d1.ausgeben('|');
    }
}
```

Beim Erzeugen der Instanz mit dem Schlüsselwort *new* werden drei Literale als Argumente an den Konstruktor übergeben.

### Übung zum Programm *DatumTest01*

Erstellen Sie eine zweite Instanz. Diesmal aber ohne Parameterübergabe an den Konstruktor. Klären Sie, welcher Konstruktor ausgeführt wird. Überprüfen Sie das Ergebnis dieses Konstruktoraufrufs.

#### 11.5.3 Beispiel: Klasse Rechteck

Das Programm *Rechteck.java* enthält eine Klassenbeschreibung für ein Rechteck. Die dafür notwendigen Daten sind zwei Positionsangaben: links oben und rechts unten, jeweils angegeben als x- und y-Wert. Außerdem enthält die Klasse zwei Konstruktoren, und sie überschreibt die Methode *toString* der Klasse *Object*.

##### Programm *Rechteck*: Override von Methoden, Overload des Konstruktors

```
import java.awt.*;
import java.util.*;
class Rechteck {
    private int x1, y1;
    private int x2, y2;

    Rechteck(int x1, int y1, int x2, int y2) {
        this.x1 = x1;
        this.y1 = y1;
        this.x2 = x2;
        this.y2 = y2;
    }
    Rechteck(Point linksoben, Point rechtsunten) {
        x1 = linksoben.x;
        y1 = linksoben.y;
        x2 = rechtsunten.x;
        y2 = rechtsunten.y;
    }
    public String toString() {
        return String.format("%d / %d / %d / %d",
                               x1, y1, x2, y2);
    }
    public static void main(String[] args) {
        Rechteck r1 = new Rechteck(5, 10, 20, 10);
        System.out.println(r1);
    }
}
```

Noch einmal der Hinweis auf die Methode *toString*: Sie ist zwar für jede Klasse verfügbar, weil sie Teil der Klasse *Object* ist, wird aber in diesem Programm überschrieben, um eine individuelle Lösung zu bekommen.

### Übung zum Programm Rechteck

Erstellen Sie ein zweites Objekt *r2*, dabei soll jedoch der zweite Konstruktor aufgerufen werden.

**Lösungshinweis:** Dazu sind zwei Instanzen der Klasse *Point* zu erstellen. Diese müssen dann bei der Instanziierung von *r2* als Parameter mitgegeben werden.

Im Paket *java.awt* gibt es diverse Klassen, um eine grafische Benutzeroberfläche zu implementieren. Dazu gehören auch die Klassen *Point*, *Dimension* und *Rectangle*. Die Klasse *Rectangle* bietet 7 verschiedene Konstruktoren an.

### Programm Rechteck02: Klassen benutzen sich gegenseitig

```
import java.awt.*;
public class Rechteck02 {
    public static void main(String[] args) {
        Point p1 = new Point(10,100);
        Dimension d1 = new Dimension(20,30);
        Rectangle r1 = new Rectangle(p1, d1);
        System.out.println(r1);
    }
}
```

## 11.6 Vererbung ("inheritance")

Die wichtigsten Motive für das Arbeiten mit Klassen sind

- die Kapselung von Daten, damit eine sichere und beweisbare Verwendung erfolgt,
- die Möglichkeit der Wiederverwendung von bereits vorliegendem Programmcode.

Die Wiederverwendung erfolgt dadurch, dass Klassen sich gegenseitig benutzen. Dabei kann man unterschiedliche Formen der Beziehungen zwischen Klassen unterscheiden:

- Die Klasse A benutzt eine andere Klasse B, indem sie mit *new* Instanzen davon erzeugt und damit arbeitet (**has - a - Beziehung**).
- Felder der Klasse A sind Referenztypen der Klasse B, und das bedeutet, dass diese Referenztypen von B automatisch instanziiert werden, wenn eine Instanz von A erzeugt wird.
- Die Klasse B ist eine spezialisierte Form der Klasse A. Dies wird realisiert durch die Vererbungsbeziehung (**is - a - Beziehung**).

Die ersten beiden Beziehungsarten haben wir in den bisherigen Programmen bereits mehrfach benutzt. In den nächsten Abschnitten werden wir uns mit der Vererbungstechnik befassen.

### 11.6.1 Was versteht man unter Vererbung ("inheritance")?

Durch den Einsatz der Vererbungstechnik ist es möglich, dass eine neue Klasse den Programmcode von einer bestehenden Klasse übernehmen und verwenden kann. Durch die Vererbungsbeziehung entsteht eine Klassenhierarchie:

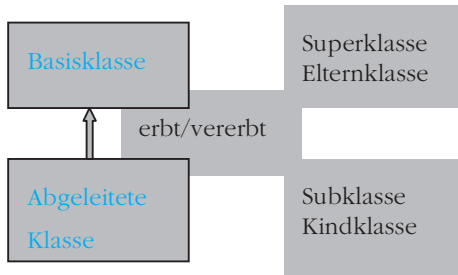


Abb. 11.4: Vererbungshierarchie (Super- und Subklasse)

Dabei ist es möglich, dass die Subklasse Daten und Methoden der Superklasse ändert, löscht oder auch neue hinzufügt. Die Klassenhierarchie kann theoretisch beliebig tief geschachtelt werden, praktisch besteht dann die Gefahr der Überfrachtung, weil die Objekte sich aus den Beschreibungen aller übergeordneten Klassen zusammensetzen.

### 11.6.2 Beispiel: Kunden und Lieferant als Subklassen der *Partner*-Klasse

Für die Beispiele zur Vererbungstechnik greifen wir auf ein bereits bekanntes Programm aus Abschnitt 11.1 zurück. Dort wurde die Klasse (Geschäfts-) *Partner01* definiert. Objekte können Kunden oder auch Lieferanten sein. Diese Klasse wollen wir als Basisklasse verwenden, sie sieht jetzt - leicht modifiziert - wie folgt aus:

#### Programm *Partner*: Beschreibung von Kunden- und Lieferantenobjekten

```
class Partner {
    private int nr;
    private String name;

    Partner(int nrl, String namel) {
        nr = nrl;
        name = namel;
    }
    void ausgeben() {
        System.out.println(nr + " " + name);
    }
    void setName(String namel) {
```

```
        name = name1;
    }
}
```

Gegenüber der ursprünglichen Version enthält diese Klasse einen Konstruktor, dafür ist die Methode *neu* entfallen.

### Erweiterung der Aufgabenstellung

Angenommen, wir wollen in einem Unternehmen **spezielle Informationen** über **Kunden** speichern und verarbeiten. Diese Informationen bestehen aus folgenden Feldern: Kundennummer, Name und Umsatz. Als Verarbeitungsmöglichkeiten für diese Daten sind vorgesehen:

- Erstellen von neuen Kundeninformationen (durch Aufruf des Konstruktors)
- Ausgeben der Kundeninformationen (durch Aufruf der Methode *ausgeben*)
- Ändern des Namens (durch Aufruf der Methode *setName*).

Große Teile dieser Aufgabenstellung sind bereits in der Klasse *Partner* realisiert (die Felder *nr* und *name* sind dort vorhanden). Lediglich die zusätzliche Information zum Umsatz gibt es dort nicht. Jetzt hilft uns folgende Erkenntnis: "Ein Kunde ist eine spezielle Form eines Geschäftspartners". Und solche Beziehungen können durch Einsatz der Vererbungstechnik in objektorientierten Sprachen elegant implementiert werden, indem für die spezielle Aufgabenstellung eine neue Klasse codiert wird, die sich auf die bereits vorhandene Lösung stützt und diese auch benutzen kann. Dies geschieht durch das Schlüsselwort "**extends**".

### Programm Kunde: Geerbt wird die vorhandene Lösung der Klasse *Partner* (1. Versuch, noch fehlerhaft)

```
class Kunde extends Partner {
    private double umsatz;
    void ausgeben() {
        super.ausgeben();
        System.out.println("Umsatz: " + umsatz);
    }
    void setUmsatz(double umsatz) {
        this.umsatz = umsatz;
    }
}
```

### Übung zum Programm *Kunde*

Ergänzen Sie die bestehende Quelltextdatei um die Klasse *Kunde*. Versuchen Sie eine Umwandlung. Dabei wird folgende Fehlermeldung ausgegeben:

... cannot find symbol ... constructor Partner()

Diese Fehlermeldung ist verwirrend und keineswegs selbsterklärend. Der Hintergrund ist: In der neuen Klasse *Kunde* ist kein Konstruktor definiert, also wird ein Default-Konstruktor eingefügt. Jeder Konstruktor ruft (unsichtbar) als ersten Befehl den Konstruktor seiner Superklasse auf, in diesem Fall also den Konstruktor *Partner()*. Weil jedoch in der Partner-Klasse ein "handgeschriebener" Konstruktor eingefügt wurde, wird der Default-Konstruktor nicht automatisch erstellt, sondern muss auch "handgeschrieben" werden. Also muss lediglich eine Zeile hinzugefügt werden.

**Die neue Klasse *Partner01a* sieht jetzt so aus:**

```
class Partner01a {
    private int nr;
    private String name;
    Partner01a() {}                                // neu
    Partner01a(int nr1, String name1) {
        nr = nr1;
        name = name1;
    }
    void ausgeben() {
        System.out.println(nr + " " + name);
    }
    void setName(String name1) {
        name = name1;
    }
}
```

### Übung zum Programm *Partner01a*

Ergänzen Sie die bestehende Quelltextdatei (mit den beiden Klassen *Partner01a* und *Kunde*) um eine neue Klasse *Lieferant*. Diese Klasse soll die Attribute und Methoden der Klasse *Partner01a* erben. Zusätzlich soll sie das Attribut *rabatt* vom Datentyp *float* bekommen und entsprechende Methoden zum Ausgeben und Updaten dieser Membervariablen.

### Lösungsvorschlag

```
class Lieferant extends Partner01a {
    private float rabatt;
    void ausgeben() {
        super.ausgeben();
        System.out.println("Rabatt: " + rabatt);
    }
    void setRabatt(float rabatt) {
        this.rabatt = rabatt;
    }
}
```



Jetzt haben wir **eine Umwandlungseinheit** mit den drei Klassen *Partner01a*, *Kunde* (Achtung: muss auch von *Partner01a* abgeleitet sein!), *Lieferant*. Die Beziehungen zwischen diesen Klassen beruhen auf Vererbungstechnik. Und dies kann grafisch dargestellt werden mit der UML-Notation (siehe Kapitel 12).

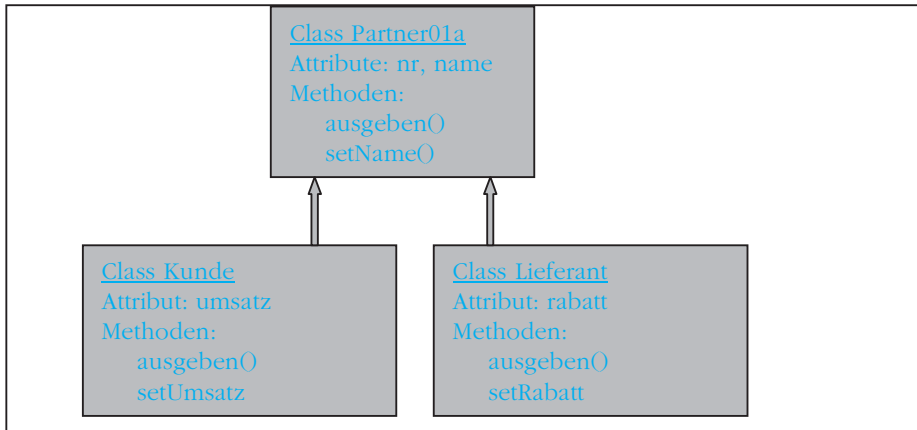


Abb.11.5: UML-Klassendiagramm

Einige Besonderheiten dieses Programmsystems müssen noch erläutert werden:

- Die Klasse *Partner01a* ist die Oberklasse (oder Superklasse oder Vaterklasse) für die beiden anderen Klassen.
- Die Klassen *Kunde* und *Lieferant* werden als Unterklasse oder Subklasse oder Kindklasse bezeichnet.

Eine Klasse ist definiert durch ihren Namen **und** ihre Oberklasse(n). Die Klasse erbt die Speicherstrukturbeschreibung (die Variablendeklarationen) und alle Operationen (die Methoden) der Oberklasse(n). Der Name der direkten Oberklasse kann fehlen, dann wird von der Klasse *Object* geerbt. Andernfalls wird explizit der Name der Superklasse hinter dem Schlüsselwort *extends* angegeben.

Die Unterklasse kann also die geerbten Elemente

- unverändert übernehmen (das geschieht automatisch, ohne dass der Programmierer etwas angeben muss),
- erweitern, d.h. ihnen etwas komplett Neues hinzufügen (das können Attribute oder auch Methoden sein),
- redefinieren, d.h. bestehende Methoden überschreiben, wenn diese sich ähnlich verhalten, jedoch mit kleinen Abweichungen)

Wenn von einem Interface (siehe Abschnitt 11.8) geerbt wird, so muss die Unterklasse das realisieren, was in dem Interface versprochen wurde.

### 11.6.3 Override und Schlüsselwörter *super* und *this*

Die Methode *ausgeben* gibt es sowohl in der Superklasse als auch in den beiden Subklassen - und zwar exakt mit derselben Signatur. Man sagt, sie wird von der Subklasse überlagert ("**override**"). Zur Ausführungszeit wird Java dafür sorgen, dass die richtige Methode ausgeführt wird, abhängig vom Datentyp der Instanz.

#### Schlüsselwort *super*

Für die Subklassen gilt: Innerhalb der Methoden *ausgeben* muss zusätzlich die Methode *ausgeben* der Superklasse ausgeführt werden, damit *nr* und *name* angezeigt werden. Dafür wird das Schlüsselwort ***super*** benutzt, denn dadurch wird die Superklasse referenziert. Das Keyword *super* kann auch benutzt werden, wenn es gleiche Feldnamen gibt in Sub- und Superklassen.

#### Schlüsselwort *this*

Außerdem sind innerhalb der Klassen die Namen für einige Attribute mehrfach vergeben worden. So gibt es z.B. in der Klasse *Kunde* den Identifier *umsatz* in der Klasse *Kunde* sowohl für die Membervariable als auch für eine lokale Variable (für den Parameter). Um diesen Namenskonflikt aufzulösen, wurde das Schlüsselwort *this* als Qualifizierer benutzt. Damit wird dem Compiler mitgeteilt, dass die Variable der aktuellen Instanz gemeint ist (und nicht etwa eine lokale Variable mit diesem Namen).

### 11.6.4 Client-Programm (Driver-Programm) erstellen, um Instanzen zu erzeugen

Das folgende Programm *PartnerTest02* soll drei Instanzen erstellen, jeweils von jeder der drei Klassen eine. Hier aber zunächst eine komplette Übersicht über die **endgültige Fassung** der drei Klassen, denn bisher fehlten noch 2 wichtige Konstruktoren:

#### Programm *Partner01a*: Mit den abgeleiteten Klassen *Kunde* und *Lieferant*

```
class Partner01a {
    private int nr;
    private String name;
    Partner01a() {}
    Partner01a(int nr1, String name1) {
        nr = nr1;
        name = name1;
    }
    void ausgeben() {
        System.out.println(nr + " " + name);
    }
    void setName(String name1) {
        name = name1;
    }
}
```

```
}  
class Lieferant extends Partner01a {  
    private float rabatt;  
    Lieferant(int nrl, String name1) {           // neu  
        super(nrl, name1);  
    }  
    void ausgeben() {  
        super.ausgeben();  
        System.out.println("Rabatt: " + rabatt);  
    }  
    void setRabatt(float rabatt) {  
        this.rabatt = rabatt;  
    }  
}  
class Kunde extends Partner01a {  
    private double umsatz;  
    Kunde(int nrl, String name1) {             // neu  
        super(nrl, name1);  
    }  
    void ausgeben() {  
        super.ausgeben();  
        System.out.println("Umsatz: " + umsatz);  
    }  
    void setUmsatz(double umsatz) {  
        this.umsatz = umsatz;  
    }  
}
```

### Programm *PartnerTest02*: Benutzen der Klassen

```
public class PartnerTest02 {  
    public static void main (String[] args) {  
        Partner01a p = new Partner01a(15, "Merker");  
        p.ausgeben();  
  
        Kunde k = new Kunde(12, "Schulz");  
        k.setUmsatz(50000);  
        k.ausgeben();  
  
        Lieferant l = new Lieferant(21, "Meyer");  
        l.setRabatt(15.0f);  
        l.ausgeben();  
    }  
}
```

Eine Instanz der Unterklasse *Kunde* oder *Lieferant* enthält neben den eigenen Attributen auch die Attribute, die von der Oberklasse geerbt werden, und sie kann auch auf die Methoden und Daten der Oberklasse zugreifen. Grafisch kann dies so dargestellt werden:

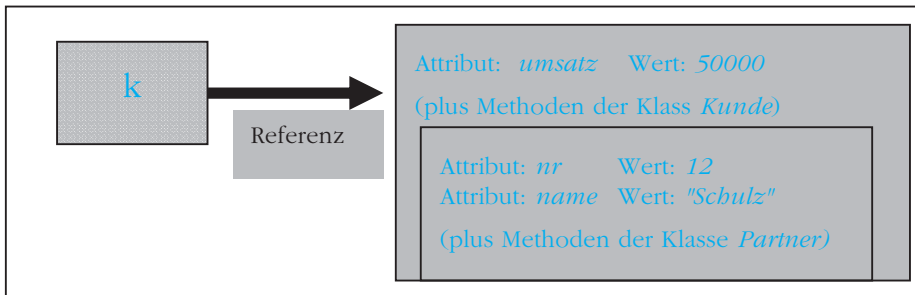


Abb. 11.7: Eine Instanz der Klasse *Kunde* enthält auch ein Objekt der Klasse *Partner*

Zur Ausführungszeit sucht die JVM die benötigten Elemente zunächst in der eigenen Klasse, danach wird weiter oben in Hierarchie weitergesucht, bis die Daten oder Methoden gefunden werden. Durch Verwendung des Zugriffsmodifiers *private* kann eine Superklasse verhindern, dass in abgeleiteten Klassen direkt auf dieses Element zugegriffen wird (siehe Kapitel 16).

#### Regeln:

Eine Unterklasse hat direkten Zugriff auf alle nicht-privaten Datenfelder und Methoden der Oberklasse(n).

Ein Objekt der Unterklasse ist eine spezialisierte Form eines Oberklassenobjekts. Es kann überall dort verwendet werden, wo eine Variable vom Typ der Superklasse erwartet wird (z.B. in einer Zuweisung). Das heißt, eine Referenzvariable vom Typ der Oberklasse kann auch auf Objekte aller Unterklassen zeigen.

### 11.6.5 Beispiel: Obst-Klassen

Mit den folgenden Klassen soll die Vererbungstechnik noch einmal an einem Komplettbeispiel demonstriert werden. Zunächst wird eine Klasse *Obst* erstellt, die als Oberklasse für weitere spezialisierte Klassen dient. Diese Unterklassen sind die Klassen *Apfel* und *Birne*, jeweils abgeleitet von der Klasse *Obst*.

#### Programm *Obst*: Superklasse für weitere Spezialisierungen

```
class Obst {
    private String name;
    private float gewicht;
```

```
Obst(String n, float g) {
    name = n;
    gewicht = g;
}
void print() {
    System.out.println("Bezeichnung: " + name);
    System.out.println("Gewicht:      " + gewicht);
}
}
```

Die Klasse *Apfel* soll abgeleitet sein von der Klasse *Obst*, sie erbt also die Merkmale *name* und *gewicht* sowie die Fähigkeit *print* zum Anzeigen der Daten (nicht geerbt wird der Konstruktor). Außerdem soll sie selbst ein zusätzliches Attribut verwalten, nämlich das Anbaugebiet.

### Programm *Apfel*: Unterklasse von *Obst*

```
class Apfel extends Obst {
    private String anbaugebiet;

    Apfel(String a, String n, float g) {
        super(n, g);
        anbaugebiet = a;
    }
    void print() {
        super.print();
        System.out.println("Anbaugebiet: " + anbaugebiet);
    }
}
```

### Übung

Erstellen Sie die zusätzliche Klasse *Birne*. Diese soll die Klasse *Obst* erweitern. Als zusätzliches Attribut enthält sie die *farbe*. Alle Attribute (die eigenen und die geerbten) sollen ausgegeben werden, wenn für eine Birnen-Instanz die Methode *print* aufgerufen wird.

### Lösungsvorschlag

```
class Birne extends Obst {
    private String farbe;
    Birne(String f, String n, float g) {
        super(n, g);
        farbe = f;
    }
    void print() {
```

```

        super.print();
        System.out.println("Farbe      : " + farbe);
    }
}

```

Abschließend benötigen wir noch eine Java-Applikation, die die Klassenbeschreibungen benutzt. Das Programm *ObstTest01.java* soll folgende Aufgabenstellung lösen:

- Es soll eine Instanz der Klasse *Apfel* mit den konkreten Werten *Boskop*, *120*, *Altes Land* erstellt werden.
- Es soll eine Instanz der Klasse *Birne* erstellt werden: *Williams Christbirne*, *140*, *Bodensee*.
- Die Daten der beiden Instanzen sollen auf dem Konsolbildschirm ausgegeben werden.

### Lösungsvorschlag

```

public class ObstTest01 {
    public static void main(String[] args) {
        Apfel a1 = new Apfel("Boskop", "Altes Land", 120);
        Birne b1 = new Birne("Williams Christ", "Bodensee", 140);
        a1.print();
        b1.print();
    }
}

```

## 11.6.6 Konstruktoren und Vererbungstechnik

**Fazit:** Die wichtigsten Regeln für das Arbeiten mit Konstruktoren im Zusammenhang mit der Vererbung sind: Konstruktoren werden nicht vererbt. Sie können innerhalb einer Klasse überladen werden, ein Überschreiben ist aber nicht möglich.

Aber: Ein gegenseitiges Aufrufen der Konstruktoren aus anderen Klassen ist möglich. So kann eine Unterklasse explizit den Konstruktor der Superklasse aufrufen. Dann **muss** dieser Aufruf **in der ersten Zeile** im Konstruktor der Subklasse stehen. Implizit (im Quelltext unsichtbar) wird immer zuerst der Defaultkonstruktor der Superklasse aufgerufen.

### Programm *Konstruktor01*: Reihenfolge der Konstruktoraufrufe

```

public class Konstruktor01 {
    public static void main(String[] args) {
        ClassB b = new ClassB();
    }
}

class ClassA {

```

```
ClassA () {  
    System.out.println("Hier ist Konstruktor A");  
}  
class ClassB extends ClassA {  
    ClassB() {  
        System.out.println("Hier ist Konstruktor B");  
    }  
}
```

### 11.7 Statische Elemente einer Klasse

Die in einer Klasse deklarierten Felder werden standardmäßig pro Instanz im Arbeitsspeicher angelegt und mit individuellen Werten gefüllt, d.h. es gibt keine gemeinsamen Datenbereiche für die Instanzen. Angenommen aber, wir benötigen bei der Verarbeitung einer Klasse ein Feld, das für **alle** Objekte zur Verfügung stehen soll, weil es - unabhängig von einzelnen Instanzen - allgemeine Informationen aufnehmen soll. Das könnte z.B. ein Summenfeld oder ein Zähler für die Anzahl der Instanzen sein. Die Lösung ist, dass dieses Element mit dem Schlüsselwort *static* gekennzeichnet wird.

#### Programm *Static01*: *static-Variable* (unabhängig von Instanzen)

```
public class Static01 {  
    static int zaehler = 0;  
    Static01() {  
        zaehler++;  
    }  
    public static void main(String[] args) {  
        Static01 z1 = new Static01();  
        Static01 z2 = new Static01();  
        System.out.println(zaehler);  
    }  
}
```

Die Variable *zaehler* wird angelegt, wenn die Klasse in den Arbeitsspeicher geladen wird. Sie steht dann sofort für die Nutzung zur Verfügung (im Gegensatz zu Instanzvariablen, die solange nicht genutzt werden können, wie kein Objekt erzeugt worden ist).

#### Übung zum Programm *Static01*

Ändern Sie das Programm so, dass die Variable *zaehler* eine ganz "normale" Mem-bervariable wird. Damit danach die Umwandlung fehlerfrei durchgeführt wird, muss

auch die Nachricht (message) an diese Variable geändert werden. Prüfen Sie dann, wie sich die Ausgabe des Programms verändert.

### Lösungsvorschlag

```
public class Static02 {
    int zaehler = 0;
    Static02() {
        zaehler++;
    }
    public static void main(String[] args) {
        Static02 z1 = new Static02();
        Static02 z2 = new Static02();
        System.out.println(z1.zaehler);
        System.out.println(z2.zaehler);
    }
}
```

Dadurch, dass die Variable *zaehler* zu einer "ganz normalen" Instanzvariablen gemacht wurde, wird sie auch pro Instanz im Arbeitsspeicher angelegt.

Das Schlüsselwort *static* kann auch für die Definition von Methoden angegeben werden. Dann sind dies Methoden, die als so genannte Klassenmethoden unabhängig von der Existenz eines Objekts aufgerufen werden können. Die Adressierung von *static*-Elementen erfolgt innerhalb derselben Klasse über den Namen, von außerhalb werden sie mit dem Klassennamen qualifiziert (und nicht wie die Instanzelemente über eine Referenzvariable).

### Programm Static03: Zugriff auf *static*-Elemente mit dem Klassennamen

```
class Static03 {
    public static void main(String[] args) {
        System.out.println(A.x);
        System.out.println(B.x);
    }
}
class A {
    static int x = 1;
}
class B {
    static int x = 2;
}
```

Die Abbildung 11.8 zeigt eine zusammenfassende Übersicht der Unterschiede zwischen *static*- und *non-static*-Elementen. Weitere Hinweise zu dem Schlüsselwort *static* gibt es im Kapitel 16.



	Instanz-Elemente	Klassen-Elemente
äußerlich erkennbar	Default, keine Schlüsselwort	Schlüsselwort <i>static</i>
wofür eingesetzt	individuelle Eigenschaften oder Methoden pro Objekt	allgemeingültige Attribute oder Methoden pro Klasse
wann angelegt	bei Instanzerzeugung	beim Laden der Klasse
wie oft angelegt	jeweils pro Instanz	nur einmal
existieren wie lange	solange Objekt aktiv	solange Klasse aktiv
wie erfolgt der Zugriff	objectreferenz.element <i>this</i> .element <i>super</i> .element	klassenname.element
haben Zugriff auf	alle Elemente (auch <i>static</i> )	nur auf <i>static</i> -Elemente

Abb.11.7: Gegenüberstellung Instanz- und Klassenelemente

## 11.8 Weitere Sprachmittel für Referenztypen (interface, enum)

Wie bereits mehrfach demonstriert, werden Objekte wie folgt deklariert:

```
datentyp bezeichner;
```

Als Datentyp kann ein primitiver Typ oder ein Referenztyp angegeben werden. Alles, was nicht ein primitiver Typ ist, ist ein Referenztyp. Bisher haben Sie als Referenztyp nur die Klassen kennen gelernt. Es gibt aber noch andere Referenztypen, z.B.

- Arraytypen (durch eckige Klammern [], siehe Kapitel 13)
- *interface*-Beschreibungen und
- *enum*-Typen.

### 11.8.1 Interface

Interfaces (Schnittstellen) enthalten lediglich Methodenbeschreibungen, d.h. der Kopf der Methode (mit ihrer Signatur) wird beschrieben, aber es gibt (noch) keine Implementierung, also keinen Rumpf. Syntaktisch erkennt man Interfaces daran, dass anstelle des Schlüsselworts *class* das Schlüsselwort *interface* benutzt wird.

#### Programm *Interface01*: Schnittstellenbeschreibung für zwei Methoden

```
interface Interface01 {
    void setZahl1(int z);
    int  getZahl1();
}
```

Diese Interface-Beschreibung deklariert die beiden Methoden: *setZahl1* und *getZahl1*. Genau wie Klassen auch, können Interfaces genutzt werden zum Vereinbaren von Referenzvariablen.

#### Programm *InterfaceTest01*: Interface als Datentyp bei der Deklaration

```
class InterfaceTest01 {
    public static void main(String[] args) {
        Interface01 schnittstelle01;
    }
}
```

Aber - nicht möglich ist das Erzeugen eines Objekts, d.h. das Anlegen von Speicherplatz für ein Interface, denn es fehlt noch die Implementierung. Implementiert wird das Interface durch eine Klassenbeschreibung, die sich auf dieses Interface bezieht. Dazu dient das Schlüsselwort "implements".

#### Programm *InterfaceTest02*: Implementierung eines Interfaces

```
public class InterfaceTest02 implements Interface01 {
    int zahl1;
    public void setZahl1(int z) {
        zahl1 = z;
    }
    public int getZahl1() {
        return zahl1;
    }
    public static void main(String[] args) {
        Interface01 schnittstelle01;
        schnittstelle01 = new InterfaceTest02();
        schnittstelle01.setZahl1(5);
        System.out.println(schnittstelle01.getZahl1());
    }
}
```

Durch das Schlüsselwort "*implements*" in der ersten Zeile verpflichtet sich die Klasse, alle im Interface vorgesehenen Methoden auch zu realisieren (zu "implementieren"). Die Zeilen 4 und 7 enthalten diese Implementierung. Und erst jetzt ist es auch möglich, in der *main*-Methode eine Instanz zu erzeugen und damit zu arbeiten. Die Instanz hat eine Referenzvariable vom Typ *Interface01*, obwohl das Objekt inhaltlich vom Typ *InterfaceTest02* ist.

Regel: Die Klasse, die das Interface implementiert, ist durch Vererbung mit dem Interface verbunden. Die Klasse ist ein Untertyp, die Schnittstelle ist die "Oberklasse". Eine Variable, deren Typ ein Interface ist, kann jedes Objekt referenzieren, dessen Klasse dieses Interface implementiert.

Im nächsten Beispiel soll eine Klasse das folgende Interface implementieren:

```
interface A {  
    int potenzieren(int z);  
}
```

Dieses Interface verlangt von einer Implementierungsklasse, die Methode *potenzieren* zu implementieren. Es soll die Zweierpotenz einer Ganzzahl errechnet und als Ergebnis dem Aufrufer zurückgegeben werden.

### Programm InterfaceTest03: Implementierung des Interface und Driver-Programm (allerdings noch fehlerhaft)

```
public class InterfaceTest03 {  
    public static void main(String[] args) {  
        A a = new B();  
        System.out.println(a.potenzieren(5));  
    }  
}  
  
class B implements A {  
    int z = 5;  
    int potenzieren() {  
        return z * z;  
    }  
}
```

### Übungen zum Programm InterfaceTest03

**Übung 1:** Die Umwandlung wird mit folgender Fehlermeldung abgebrochen: "B is not abstract and does not override abstract method potenzieren(int) in A". Klären Sie die Ursache dieses Fehlers. Hinweis: Die Signaturen in Interface und Klasse müssen exakt übereinstimmen.

**Übung 2:** Nach dieser Korrektur liefert die Umwandlung erneut einen Fehler: "potenzieren(int) in B cannot implement potenzieren(int) in A; attempting to assign weaker access privileges; was public". Die Ursache dieses Fehlers ist nicht so offensichtlich. Dazu muss man wissen, dass die Methoden im Interface immer *public* sind. Deswegen muss bei der Implementierung dieses Schlüsselwort angegeben werden.

Die Class B sieht also endgültig und fehlerfrei wie folgt aus:

```
class B implements A {  
    public int potenzieren(int x) {  
        return x * x;  
    }  
}
```

### Wozu braucht man Interfaces?

API-Spezifikationen bestehen zum Großteil aus Interface-Beschreibungen. Sie legen das "Protokoll" für den Aufruf von Dienstleistungen fest, indem die Methoden-Signaturen exakt beschrieben sowie der Rückgabotyp und zusätzliche Modifier festgelegt werden. Damit können unterschiedliche Ziele erreicht werden:

- Für den Aufrufer der Methoden reichen die Informationen über die Schnittstelle, um die Dienstleistung nutzen zu können. Die eigentliche Implementierung kann ihm verborgen bleiben (z.B. aus Sicherheits- oder aus Vereinfachungsgründen).
- Für die Klassen, die das Interface implementieren, wird dadurch der Kopf der Methoden exakt vorgeschrieben. Das ermöglicht eine Standardisierung von Anwendungen. Der korrekte Aufruf wird vom Compiler überprüft.
- Der Aufrufer kann sich darauf verlassen, dass die im Interface angebotenen Methoden leisten, was die Beschreibung verspricht (das allerdings kann von Java nicht überprüft und sichergestellt werden).
- Die Klasse, die das Interface implementiert, ist von dem Interface "abgeleitet", d.h. es gelten die Vererbungsregeln. Damit können z.B. Objekte der Subklasse überall da verwendet werden, wo das Interface erwartet wird.
- Weitere Einsatzmöglichkeiten von Interfaces ergeben sich bei verteilten Anwendungen (RMI oder EJB), beim Polymorphismus und beim dynamischen Laden von Klassen, alles Themen für erfahrene Java-Programmierer.

### 11.8.2 enum

Enum-Beschreibungen ("Enumerationen", engl. Aufzählung) enthalten die Aufzählung von Konstanten. Sie sind eine spezielle Form einer Klasse. Ihre Basisklasse ist nicht die Klasse *Object*, sondern (davorgeschaltet) die Klasse *Enum*. Man kann Variablen deklarieren und dabei als Typ eine *enum*-Beschreibung verwenden. Dadurch wird ein typsicheres Arbeiten mit den Konstanten der *enum* ermöglicht. Intern wird die Aufzählung der Konstanten mit einer Ganzzahl verknüpft.

#### Programm Enum01: enum als separate Einheit (unabhängig von einer Klasse)

```
enum Farbe {  
    rot, gruen, blau;  
}  
  
public class Enum01 {  
    public static void main(String[] args) {  
        for (Farbe f : Farbe.values())  
            System.out.println(f);  
    }  
}
```

In diesem Beispiel steht *enum* **auf gleicher Ebene** wie die Klasse (und könnte auch eine eigene Umwandlungseinheit sein). Syntaktisch erkennt man Enums daran, dass anstelle des Schlüsselworts *class* das Schlüsselwort *enum* benutzt wird.

Enums können auch **innerhalb einer Klasse** (gleichrangig mit Methoden, aber nicht innerhalb von Methoden) benutzt werden.

### Programm Enum02: *enum* innerhalb einer Klasse

```
public class Enum02 {
    enum Wochentage {
        sonntag, montag, dienstag, mittwoch,
        donnerstag, freitag, samstag;
    }
    public static void main(String[] args) {
        for (Wochentage w : Wochentage.values())
            System.out.println(w);
    }
}
```

Enum-Beschreibungen können wie normale Klassen gesehen werden. Sie erben Methoden von ihrer Basisklasse *Enum*, und sie können eigene Methoden enthalten.

Das folgende Beispiel besteht aus einer Umwandlungseinheit, die sowohl die *enum*-Beschreibung als auch ihre Benutzung enthält. Es demonstriert folgende Verarbeitungsmöglichkeiten:

- *enum*-Beschreibungen können als Typ einer Variablen genutzt werden,
- mit dem *if*-Statement wird der aktuelle Wert verglichen mit einem konstanten Wert aus der *enum*-Beschreibung,
- mit dem *switch*-Statement wird eine elegante Mehrfach-Abfrage realisiert,
- mit der *for*-Schleife wird der komplette Wertebereich des *enum*-Typs verarbeitet,
- Aufruf einer Methode aus der *enum*-Beschreibung.

### Programm Enum03: *enum*-Beschreibung für typsichere Aufzählungen

```
enum Tageszeit {
    morgens, mittag, abends;
    void anzeigen() {
        System.out.println(this);
    }
}

public class Enum03 {
    public static void main(String[] args) {
        Tageszeit t1; // Variable definieren
    }
}
```

```

t1 = Tageszeit.mittag;           // Wertezuweisung
if (t1 == Tageszeit.mittag)     // Variable abfragen
    System.out.println("Guten Tag");
switch(t1) {
    case morgens: System.out.println("Guten Morgen"); break;
    case mittag:  System.out.println("Guten Tag");    break;
    case abends:  System.out.println("Guten Abend");  break;
}
for (Tageszeit t2 : Tageszeit.values())
    System.out.println(t2);
t1.anzeigen();                  // Aufruf einer Methode
}
}

```

Wo liegt der Vorteil einer *enum*-Beschreibung im Vergleich zu *static final*-Variablen? Die Antwort gibt das nachfolgende Programm.

### Programm EnumTest02: Typsicherheit nicht gewährleistet

```

public class EnumTest02 {
    // Tageszeit
    static final int MORGENS = 0;
    static final int MITTAG  = 1;
    static final int ABENDS  = 2;

    // Farben
    static final int ROT      = 0;
    static final int BLAU     = 1;

    public static void main(String[] args) {
        int tageszeit = BLAU;           // FALSCH ! Keine Kontrolle
        if (tageszeit == MITTAG)
            System.out.println("Guten Tag");
    }
}

```

Das Programm gibt den Mittagsgruß aus, obwohl der Inhalt der Variablen *tageszeit* "blau" ist. Denn: Im Gegensatz zum *enum*-Typ gibt es keine Typsicherheit, der Wert der Variablen *tageszeit* kann ein beliebiger Integerwert sein.

Weitere [Vorteile](#) beim Einsatz von *enum* anstelle von *final*-Konstanten:

- Wenn eine weitere Konstante hinzukommt oder eine bestehende sich ändert, muss lediglich die *enum*-Beschreibung geändert werden (und nicht alle Klassen mit der Aufzählung von *final*-Feldern). Das erhöht die Wartungsfreundlichkeit.

- Es gibt eine allgemeine Basisklasse für alle Enumerationstypen, die Klasse *Enum*. Davon erbt jede *enum*-Beschreibung. Das heißt, es stehen weitere Methoden wie *equals*, *values* oder *toString* zur Verfügung, weil sie von dieser Klasse geerbt werden. Das erhöht die Vielseitigkeit und Leistungsfähigkeit.

## 11.9 Zusammenfassung

Eine Klasse ist eine Schablone (template, blueprint, Bauplan, Objektmuster), die Variablen und Methoden für eine Gruppe von Objekten definiert. Durch die Definition einer neuen Klasse erweitert der Programmierer die Sprache um einen neuen Datentyp.

### Die wichtigsten Elemente einer Klasse

```
class A {  
    int z;                                // Attribut/Feld/Variable  
    A() {  
        ...                               // Konstruktor  
    }  
    void verarbeiten(int a) {  
        ...                               // Methode  
    }  
}
```

Im **Kopf der Klasse** (unmittelbar vor dem Schlüsselwort *class*) können die Modifier *public* oder *private* angegeben werden, um die Zugriffsrechte auf diese Klasse festzulegen (falls sie von der Standardannahme abweichen, siehe hierzu Kapitel 16). Hinter dem Schlüsselwort *class* wird der Name der Klasse festgelegt. Bei einer Klasse mit dem Modifier *public* muss dieser Identifier mit dem Namen der Quelldatei übereinstimmen.

Der **Klassenbody** steht in geschweiften Klammern. Er enthält folgende Elemente ("member"):

- **Instanzvariablen und Instanzmethoden**: diese sind jeweils gebunden an eine ganz bestimmte Instanz, d.h. sie existieren erst, wenn eine Instanz erzeugt worden ist, und können auch erst dann genutzt werden.
- **Klassenvariablen und Klassenmethoden**, diese sind mit dem Schlüsselwort *static* gekennzeichnet und können genutzt werden, ohne dass eine Instanz erzeugt worden ist; sie existieren, sobald die Klasse in den Arbeitsspeicher geladen wird.

Außerdem enthält der Klassenbody die **Konstrukturen**. Das sind spezielle Methoden, mit abweichender Syntax und mit besonderer Semantik. Sie werden nicht vererbt.

Die Reihenfolge der Variablen- und Methodendeklarationen im Body ist beliebig und hat keinen Einfluss auf die Reihenfolge der Ausführung.

Dieselben Methodennamen können innerhalb einer Klasse mehrfach vorkommen. Allerdings müssen sie dann unterschiedliche Signaturen haben, d.h. sie müssen sich unterscheiden in der Anzahl oder beim Datentyp der Parameter. Dieser Mechanismus wird **Überladen (overload)** genannt.

Wird eine Methode einer Oberklasse in einer Unterklasse noch einmal definiert (mit derselben Signatur), so nennt man diesen Mechanismus **Überschreiben (override)**. Dadurch kann eine Methode der Superklasse individuell für die Subklasse modifiziert werden.

### Arbeiten mit der Klasse

Gearbeitet wird mit der Klasse, indem zur Ausführungszeit eines Programms konkrete Einzelexemplare ("Instanzen") dieser Klasse im Arbeitsspeicher erzeugt werden. Dabei wird der Klassenname wie ein eingebauter Datentyp bei der Deklaration der (Referenz-)Variablen eingesetzt. Danach kann mit dem Schlüsselwort *new* der Speicherplatz belegt und der Konstruktor der Klasse aufgerufen werden.

Beim Definieren eines Objekts ist es nicht so, dass der deklarierte Datentyp und der Typ des referenzierten Objekts in jedem Fall exakt übereinstimmen müssen, es können auch zwei verschiedene Typen auftreten (vorausgesetzt, sie sind "verwandt"):

```
Object obj = new String("Hallo");
```

Daraus ist ein gravierender Vorteil der Vererbungstechnik erkennbar: Untertypen können überall dort benutzt werden, wo ein Exemplar einer Oberklasse erwartet wird. Ein Beispiel zeigt die folgende Wertezuweisung:

```
Object obj;  
obj = new String("Hallo");
```

Innerhalb der Klasse können die Elemente (Attribute, Methoden) mit ihrem Namen angesprochen werden, von außerhalb muss die Punktnotation benutzt werden, d.h. die Elemente müssen mit dem Instanznamen (bzw. Klassennamen) qualifiziert werden.

Einerseits ist eine Klasse eine Beschreibung für einen Service, der dem Programmierer angeboten wird in Form von Datenbereichen und Verarbeitungsmethoden, andererseits ist eine Klasse auch ein Programm, das den Service von anderen Klassen nutzen kann. Ein typisches Java-Programm benutzt mehrere Klassen und erzeugt daraus viele Objekte, die miteinander kommunizieren, indem sie einander Nachrichten schicken.

Die Zusammenarbeit zwischen Klassen kann in zweierlei Form erfolgen:

- als Aggregation (*has-a*-Beziehung oder *part-of*-Beziehung).
- mit Vererbungstechnik (*is-a*-Beziehung), siehe Kapitel 12.



# 12

## Module entwerfen, kapseln und dokumentieren

---

Große Softwaresysteme sind komplex, der Entwicklungsprozess stellt große Anforderungen an das beteiligte Team. Aber nicht nur die Komplexität z.B. bei internationalen Großprojekten ist ein Problem - kritisch für die Beherrschbarkeit sind auch

- das grundsätzliche Verhalten diskreter Systeme. Kleine Änderungen können überraschende Auswirkungen haben, denn im Gegensatz zu kontinuierlichen Systemen genügt ein falsches Bit, um ein fehlerhaftes Verhalten zu erzeugen;
- die multiplikative Wirkung eines Einzelfehlers. Im Gegensatz zu vielen anderen Tätigkeiten, wo ein Fehler auch wirklich nur singuläre Wirkung hat, wird ein einzelner Programmfehler u.U. millionenfach vervielfältigt.

Ein Programm, das aus einzelnen, möglichst unabhängigen Modulen zusammengesetzt ist, ist leichter verständlich als ein monolithischer Programmblock mit Tausenden von Programmzeilen. Diese triviale Erkenntnis führte dazu, dass das wichtigste Prinzip bei der Entwicklung von EDV-Systemen die Modularisierung ist. Ein Programmierer baut seine Anwendung aus primitiven Grundbausteinen, aus Modulen, zusammen - vergleichbar der Plattenbauweise im Wohnungsbau oder der Modulbauweise bei technischen Geräten.

Objektbasierte Programmsysteme werden auf der Grundlage ihrer Datenstruktur modularisiert, unter Berücksichtigung aller Operationen, die mit dieser Datenstruktur ausgeführt werden können.

In diesem Kapitel bekommen Sie Antworten auf folgende Fragen:

- Was versteht man unter dem Schlagwort "Modul" (ist es eine Klasse, ein Objekt, ein Paket)?
- Welche Gründe sprechen für die Modularisierung?
- Warum ist es wichtig, die Prinzipien "encapsulation" und "information hiding" zu beachten?
- Wie ist die objektorientierte Vorgehensweise beim Realisieren von Softwaresystemen? Was ist OOA, OOD und OOP?
- Was ist UML (unified modeling language) und wann wird diese Notation benutzt?
- Was bedeuten die Fachbegriffe "Pattern" und "Framework"?

## 12.1 Was ist ein Modul?

Leider gibt es in der Informatik viele Begriffe, die unscharf definiert sind. Manche haben sogar eine mehrfache, unterschiedliche Bedeutung, andere sind unpräzise übersetzt aus dem Englischen. Hinzu kommt, dass Anglizismen und Abkürzungen benutzt werden, ohne zu beschreiben, was damit gemeint ist.

In diesem Kapitel werden wir es mit einigen typischen Beispielen dafür zu tun haben: Module, Komponenten und Frameworks. Zunächst der Begriff "Modul". In diesem Buch wollen wir diesen Begriff benutzen für Programm(-teile), die **folgende Eigenschaften** haben:

- Module fassen Operationen und Daten zu einem Programmteil zusammen.
- Module erfüllen eine abgeschlossene, sauber beschriebene Aufgabe.
- Die Kommunikation dieses Moduls mit der Außenwelt darf nur über eindeutig definierte Schnittstellen erfolgen.
- Zum Aufrufen und Benutzen dieses Moduls in einem Programmsystem sind keine Kenntnisse der internen Code-Implementierung notwendig.
- Module trennen also zwischen den Schnittstellen (interfaces) und der Implementierung. Die Implementierung enthält den eigentlichen Programmcode. Das Interface beschreibt die Aufrufmöglichkeiten und die Ein- und Ausgaben für das Modul (Parameter und Ergebnistyp).

Bezogen auf Java ist ein Modul ganz offensichtlich eine Klasse. Eine Klasse ist also in objektorientierten Sprachen *das Mittel zur Modularisierung von Softwaresystemen*.

In anderen Sprachen und Zusammenhängen wird der Begriff "Modul" auch mit Prozeduren oder Methoden, mit Komponenten oder mit Paketen gleichgesetzt, manchmal wird auch unterschieden zwischen den statischen Elementen einer Klasse und dem dynamischen Aspekt bei der Instanziierung, um den Begriff Modul zu definieren.

Wir bleiben bei der einfachen Definition: **ein Modul ist eine Klasse**.

Noch ein Hinweis zum Begriff "Komponente". Vielfach ist dies ein Synonym für den Modulbegriff. In der letzten Zeit allerdings ist mit dem Begriff "Komponententechnologie" immer häufiger die Aufteilung einer Anwendung auf mehrere Adressräume (also auf mehrere Maschinen bzw. bei Java auf mehrere JVM) verbunden. Beispiel: In der Enterprise-Technologie bezeichnet man die Enterprise Java Beans (EJB) als "Komponenten", sie können über eine bestimmte Technik von remoten (entfernten) Programmen genutzt werden (z.B. durch Aufruf von Methoden), als wären sie lokal in dem eigenen System vorhanden.

Eine Komponente in diesem Sinn ist also eine Weiterentwicklung eines Moduls, sie kann über Adressraumgrenzen (in Java also zwischen unterschiedlichen JVM) hinweg genutzt werden.

## 12.2 Motivation für Modulbildung

Das wichtigste Designprinzip bei der Programmentwicklung ist die Modularisierung des Gesamtkomplexes, d.h. ein Programmsystem soll aus einzelnen Modulen bestehen. Dadurch wird ein komplexes Problem in überschaubare Teilprobleme aufgeteilt ("teile und herrsche"). Eng verbunden mit der Modularisierung sind die beiden Prinzipien "Kapselung" und "Verstecken von Informationen".

### 12.2.1 Vorteile der Modulbildung

Wenn ein Modul ein abgeschlossener, unabhängiger Teil eines Softwareprogramms ist, der Daten und Verarbeitungsschritte zusammenfasst, so sind damit folgende Vorteile verbunden:

- Das gesamte System wird übersichtlicher, es wird gegliedert und strukturiert.
- Kleine, abgeschlossene Programmblöcke sind verständlicher als große.
- Mehrfachverwendung der Module ist möglich, sie sind beliebig kombinierbar.
- Testbarkeit wird vereinfacht durch inkrementelles Vorgehen.
- Änderbarkeit (Wartung, engl. maintenance) wird vereinfacht; wenn sich die Implementierung eines Moduls ändert, die Schnittstellen aber gleich bleiben, so berührt dies die Nutzerprogramme nicht, sie müssen nicht geändert werden. Beispiel: Ein Sortiermodul arbeitet nach dem Bubblesort-Algorithmus und wird ausgetauscht gegen Quicksort.
- Arbeitsteilung und Parallelentwicklung möglich (Verteilung auf mehrere Programmierer).

### 12.2.2 Vorteile der Kapselung (encapsulation)

Unter Kapselung (encapsulation) versteht man das "Privatisieren" der internen Elemente. Insbesondere die Datenelemente sollen von außen nicht zugreifbar sein. Weder das direkte Lesen und erst recht nicht das Verändern dieser Attribute sollen durch Methoden anderer Klassen möglich sein.

Manchmal werden Module dieser Art auch als "abstrakte Datentypen" bezeichnet, damit wird ausgedrückt, dass die Daten autonom implementiert sind und von ihnen nur Abstraktionen bekannt sind. Das heißt, die Realisierung bleibt verborgen, und der Zugriff auf die Datenfelder kann nur mittels zugelassener Operationen erfolgen.

Über Zugriffsmodifizier (siehe Kapitel 16) wird pro Element die Stufe der Kapselung (*private*, *protected* oder *public*) festgelegt.

- Durch Kennzeichnung der Daten als *private* sind sie vor unberechtigtem Zugriff geschützt, weil sie nur innerhalb der eigenen Klasse benutzt werden können.

- Der Zugriff erfolgt über *public*-Methoden. Diese haben exakt definierte Schnittstellen ("Signaturen"), und im besten Fall erlauben diese Schnittstellen minimalen Datenaustausch.
- Ein extremes Gegenbeispiel wären globale Daten (in Java mit *static public* definiert). Bei ihnen ist eine Kontrolle über die korrekte Verwendung fast unmöglich. Denn je mehr Benutzer direkt zugreifen können, umso fehleranfälliger und unüberschaubarer der Einsatz. Dagegen sind bei privaten Daten unbeabsichtigte, schwer nachvollziehbare "Seiteneffekte" nicht möglich.
- Dadurch erreicht man eine sichere und beweisbare Verwendung der Daten. Die Verantwortung für inhaltliche Richtigkeit ist dann klar definiert: sind die Daten fehlerhaft, so kann die Ursache nur innerhalb dieses Moduls gesucht werden - kein anderer darf direkt darauf zugreifen.

### 12.2.3 Vorteile des Information Hiding

Unter "information hiding" versteht man, dass die internen Details für die Benutzung eines Moduls versteckt werden. Dadurch bleiben dem Aufrufer die Implementierungsdetails verborgen. Bekannt sind von den einzelnen Modulen nur die dokumentierten Leistungsbeschreibungen und die Exportschnittstellen (wie werden die Methoden aufgerufen, um damit zu arbeiten?). Die interne Arbeitsweise, also wie ist die Dienstleistung realisiert, ist versteckt. Für den Klienten eines Moduls hat das folgende Vorteile:

- Information Hiding befreit ihn davon, die internen Abläufe eines Dienstleistungsmoduls verstehen zu müssen. Der Klient kennt von einer Klasse lediglich die Schnittstelle (das "interface"), damit er weiß, wie Methoden aufgerufen werden und welche Ergebnisse sie liefern.
- Die Übersicht über den Gesamtkomplex wird erleichtert und die Einarbeitung vereinfacht.

Woher bekommt aber der Nutzer einer Klasse die Informationen über die Schnittstellen? In Java erfolgt die Beschreibung der Interfaces einerseits im Kopf der Klasse bzw. im Kopf der einzelnen Methoden. Und andererseits gibt es sogar die Möglichkeit, ein **besonderes Sprachmittel für die Beschreibung von Schnittstellen** zu benutzen: **das *interface*** (weitere Informationen hierzu im Kapitel 11) Das ist eine spezielle Form einer Quelltextdatei, in der ausschließlich die Angaben zu den Exportschnittstellen (die Signaturen) stehen.

Natürlich muss der Nutzer einer Klasse bzw. der Aufrufer einer Methode auch die Leistungsbeschreibung des Moduls kennen. Für die Standardklassen steht die Beschreibung in der API-Dokumentation; für selbst codierte Klassen hat der Programmierer die Möglichkeit, mit Hilfe des Dokumentationsgenerators *javadoc* eine HTML-Datei mit der Leistungsbeschreibung zu erstellen (siehe Abschnitt 8.8.1).

## 12.3 Objektorientierte Systementwicklung

Der Entwickler einer neuen objektorientierten Anwendung hat zwei große Fragen zu klären: Gibt es bereits fertige Lösungen in Form von Klassen oder Klassenbibliotheken? Wenn nicht, wie muss ich vorgehen, um selbst neue Klassen zu entwickeln?

### Wie findet man die richtige Klasse für eine gegebene Aufgabenstellung?

Die Komplexität der Java-Sprache liegt nicht in der Sprachspezifikation (es gibt nur etwa 50 Schlüsselwörter), sondern in der Vielfalt der Klassenbibliotheken. Neben der Standard-Bibliothek, die zum Lieferumfang des JDK gehört und ohne die eine Java-Entwicklung nicht möglich ist, gibt es auf dem Markt eine schier unüberschaubare Fülle von Bibliotheken für jedes Sachgebiet: für Enterprise-Anwendungen, für die weitgehende Automatisierung der Prüfung von Benutzereingaben (Plausibilität und Validierung), für mathematische Methoden (Matrizen- und Differentialrechnung) für Statistikberechnung (Vorhersagen, Regression oder Optimierungen usw.).

### Wie muss man vorgehen, um neue Klassen zu entwerfen und zu codieren?

Bei der objektorientierten Softwareentwicklung gibt es zwei Vorgehensweisen, die sich unterscheiden durch die Richtung der Abstraktion: die Top-Down-Methode, bei der die grobe Lösungsidee schrittweise verfeinert wird, um so zu dem Modulentwurf zu kommen, und die Bottom-Up-Methode, bei der zunächst die einzelnen Bauteile (Module) entwickelt (oder auch: zusammengesucht) werden und daraus das Gesamtsystem zusammengestellt wird.

In der Praxis wird man üblicherweise eine Kombination beider Vorgehensweisen finden. Man kann dabei folgende Schritte unterscheiden (leider ist die Definition der folgenden Begriffe nicht allgemeingültig und sauber festgelegt):

- OOA (Objektorientierte Analyse)
- OOD (Objektorientiertes Design)
- OOP (Objektorientierte Programmierung).

### 12.3.1 Objektorientierte Analyse (OOA)

Kernaufgaben dieser Phase sind: Verstehen und Formulieren der Aufgabenstellung, Analyse der Kundenerwartungen, Lösungsalternativen entwickeln, Ziele und Zeitpläne festlegen.

Das Ergebnis sind Grobentwürfe für die Architektur des neuen Systems mit Beschreibungen der weiteren Vorgehensweise.

Bei kleineren Systemen kann es bereits in dieser Phase sinnvoll sein, eine fachliche Beschreibung in UML-Notation (siehe Abschnitt 12.4) zu erstellen oder Prototypen z.B. für die Benutzeroberfläche, einzusetzen.

### 12.3.2 Objektorientiertes Design (OOD)

Hier erfolgt die Festlegung der Systemstruktur. Der Schwerpunkt der objektorientierten Designmethode liegt darauf, das Projekt aufzuteilen in Klassen. Dazu gehört die detaillierte Festlegung, aus welchen Elementen die neuen Klassen bestehen, welche konkreten Eigenschaften und Aktionen sie kennen und wie die Klassen untereinander verbunden sind.

Dabei hilft es, zunächst zu generalisieren, bevor später dann Lösungen für Spezialfälle entwickelt werden. Die wichtigsten Entscheidungen, basierend auf den Erkenntnissen aus der Analysephase, sind: Wo sind gleichartige Objekte? Wo liegen die Gemeinsamkeiten innerhalb dieser Gruppe von Objekten. Wo sind die Unterschiede zu den anderen Objekten? Auf diese Weise werden Klassen identifiziert.

Klassen repräsentieren zunächst lediglich Konzepte, nicht die Anwendung selbst. Sie erweitern den Sprachumfang von Java, denn sie beschreiben neue (selbstdefinierte) Typen mit ihren Operationen. In den meisten Fällen kann man eine Klasse als Angebot einer Dienstleistung sehen, das vom Client der Klasse genutzt wird (durch das Senden einer Nachricht).

#### 12.3.2.1 Generelle Empfehlungen für das Klassendesign

Die Regeln zur Klassenbildung können wie folgt zusammengefasst werden:

- Ein Programmmodul soll **eine (und nur eine) Aufgabe** erfüllen. Die Dinge, die nicht zur Aufgabenstellung dieser Klasse gehören, sollten separiert und in einer eigenen Klasse untergebracht werden. Gut ist die Klassenbildung gelungen, wenn ihre Aufgabe mit einem Substantiv beschrieben werden kann.
- Die Klasse soll **so einfach wie möglich** sein. Kurze, einfache Klassen sind besser als große. Es sollte nicht zwanghaft versucht werden, alle denkbaren und möglichen Fälle abzudecken und in die Klasse aufzunehmen. Viele Projekte sind daran gescheitert, dass gleich am Anfang versucht wurde, zuviel auf einmal abzudecken. Konzentrieren Sie sich auf die Aufgaben, die anstehen. Nicht mehr - und nicht weniger.
- Die Dinge, die sich oft ändern, sollten **getrennt** werden von Aufgabenstellungen, die (relativ) konstant sind. Auch sollten plattformabhängige Lösungen getrennt werden von allgemein gültigen Lösungen. Damit erleichtert man die Wartung, und eine größere Anzahl von Klassen wird portabel.
- Vorteilhaft kann es sein, den **Standpunkt des Clients einzunehmen** (ist der Sinn der Klasse einleuchtend, ist die Klasse anwenderfreundlich, ist sie robust usw.).
- Der **Aspekt der Wiederverwendung** spielt in objektorientierten Systemen eine wichtige Rolle. Deswegen sollten Sie immer auch bedenken, ob universelle Bausteine entwickelt werden können, auch wenn der allgemeine Einsatz als wieder verwendbares Modul noch nicht unmittelbar zu erkennen ist.

### 12.3.2.2 Formale Prinzipien für die Bildung von Modulen (Klassen)

Die Entscheidung, welche Teile zu einer Klasse zusammengefasst oder welche auf mehrere Klassen aufgeteilt werden müssen und wie diese Klassen zusammenarbeiten, ist abhängig von folgenden Kriterien:

- Die **Modulbindung** soll hoch sein  
Das bedeutet, dass der innere Zusammenhalt möglichst eng sein soll (= große Kohäsion). Ein Modul sollte keine Bestandteile enthalten, die inhaltlich nichts miteinander zu tun haben ("eine Funktion = ein Modul"). Die Klasse sollte möglichst so atomar sein, dass sich ihre Aufgabe mit einem Wort beschreiben lässt.
- Die **Modulkopplung** soll niedrig sein  
Das bedeutet, dass die Abhängigkeit zwischen den verschiedenen Modulen möglichst gering sein soll. Die Verknüpfung untereinander sollte so schwach wie möglich sein: wenig Parametertausch, keine globalen Datenbereiche. Jedes Modul soll wohldefinierte öffentliche Schnittstellen besitzen, über die es mit seiner Umgebung kommuniziert.
- Berücksichtigung von **Design-Pattern**  
Profitieren Sie von den Erfahrungen anderer, benutzen Sie Entwurfsmuster. Pattern sind keine fertigen Lösungen, sondern sie bieten Lösungsmuster für häufig wiederkehrende Aufgabenstellungen. Beispiel für ein bekanntes Pattern: Trennung der Benutzeroberfläche von der fachlichen Verarbeitung und Einrichtung eines Steuerobjekts (Model-View-Control-Pattern, MVC).
- **Relationen** zwischen den Klassen **beschreiben**  
Wenn das Design der einzelnen Klassen feststeht, muss überlegt werden, wie die Beziehungen ("Relationen") zwischen den Klassen sind. Neben der Vererbung gibt es noch Beziehungen, die nicht auf Vererbung beruhen ("Assoziationen"). Als Assoziation wird zunächst einmal jede Form einer Beziehung zwischen zwei Klassen bezeichnet, die nicht auf Vererbung beruht. Nähere Angaben, z.B. über die Art der Realisierung, werden dabei nicht gemacht.
- Ergebnis des OOD **in UML dokumentieren**  
Die Spezifikationen der OOD müssen dokumentiert werden. Hier helfen die verschiedenen Diagramme der Unified Modeling Language (UML). Insbesondere sind Klassendiagramme hilfreich, Details siehe nächsten Abschnitt.

Es gibt zwei fundamentale Arten von Verbindungen (Beziehungen, Relationen, Assoziationen) zwischen Klassen:

- die **is-a-Beziehung** (Vererbung, z.B. Klasse B ist ein spezieller Typ der Klasse A)
- die **has-a-Beziehung** (Assoziation, z.B. Klasse A hat ein Objekt der Klasse B)

### 12.3.2.3 Is-a-Beziehung (Vererbung)

Die engste Beziehung zwischen Klassen ist die Vererbungsbeziehung. Eine Unterklasse ist ein spezieller Repräsentant der generellen Oberklasse. In Java wird dies durch das Schlüsselwort *extends* ausgedrückt:

```
class Unterklasse extends Oberklasse { ... }
```

Voraussetzung für das Bilden einer solchen Klassenhierarchie ist eine Beziehung, die beschrieben werden kann als Generalisierung (Verallgemeinerung) der Unterklasse bzw. Spezialisierung (Verfeinerung) der Oberklasse.

Ziel ist die Wiederverwendung von vorhandenem Code. Die Unterklasse erbt alle Fähigkeiten der Oberklasse. Dabei kann sie die Struktur oder das Verhalten ändern, ohne dass die Oberklasse davon berührt ist.

Ein weiterer Vorteil der Vererbungstechnik ergibt sich daraus, dass überall da, wo ein Objekt der generellen Klasse möglich ist, auch ein Objekt der speziellen Klasse stehen kann. Beispiel:

#### Programm Vererbung01: Zuweisungskompatibilität und Vererbungshierarchie

```
public class Vererbung01 {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        System.out.println(b.str);
        a = b;
        System.out.println(a.str);
    }
}

class A {
    String str = "KlasseA";
}
class B extends A {
}
```

Die Wertezuweisung `a = b` ist erlaubt, weil B eine Subklasse von A ist.

Beim Erzeugen einer Instanz der Subklasse werden **immer alle** Elemente der Superklasse automatisch eingebettet in das neu erzeugte Objekt. Diese enge Bindung hat jedoch nicht nur Vorteile, es besteht sehr leicht die Gefahr der Überfrachtung von Klassen. Die Vererbung wird in UML durch einen offenen Pfeil, ausgehend von der Subklasse in Richtung Superklasse, dargestellt.



### 12.3.2.4 has-a-Beziehungen (Assoziation)

Beziehungen, die nicht auf Vererbung beruhen, können als "hat-eine-Beziehung" beschrieben werden. Dabei werden in UML drei unterschiedlich enge Beziehungen unterschieden:

- die allgemeine **Assoziation**, bei der lediglich beschrieben ist, dass Klasse-A eine Instanz von Klasse-B benutzt;
- die **Aggregation**, bei der zwischen den Klassen eine Ganze-Teile-Beziehung besteht, z.B. wenn Klasse-B ein Teil von Klasse-A ist;
- die **Komposition**, eine stärkere Form der Aggregation, bei der die Klasse-B nicht nur in A enthalten ist, sondern sogar existentiell von ihr abhängig ist.

Während es in UML unterschiedliche Symbole gibt für diese drei Klassifikationen, unterscheidet man in Java nicht zwischen Assoziation, Aggregation und Komposition. Es gibt keine speziellen sprachlichen Konstrukte dafür, darüber hinaus ist die Unterscheidung nicht immer leicht und häufig auch nicht ganz eindeutig zu treffen.

#### Assoziation

Ist die Beziehung zwischen den Klassen lediglich in der allgemeinen Form von "A benutzt B", dann wird dies in UML durch eine einfache Linie dargestellt. Damit ist nur ausgesagt, dass eine Methode der Klasse A mit einem Objekt der Klasse B arbeitet. Dabei kann das Objekt B in einer Methode mit *new* erzeugt worden sein:

#### Programm *ClassA*: Instanz erzeugen und benutzen

```
public class ClassA {
    public static void main(String[] args) {
        new ClassB("Von wem wird dies ausgegeben?");
    }
}
class ClassB {
    ClassB(String text) {           // Konstruktor
        System.out.println(text);
    }
}
```

Diese Art der Verbindung zwischen Klassen ist immer dann sinnvoll, wenn Service in spezielle Module ausgelagert wird.

Das Programm *ClassA* hat (unabhängig vom Thema Assoziation) eine zusätzliche Besonderheit: es erzeugt mit *new* ein **anonymes Objekt** von der Klasse B. Das ist immer dann sinnvoll, wenn lediglich die Instanziierung (also das Durchlaufen des Konstruktors) ausgeführt werden soll, die lokale Variable für dieses Objekt danach aber nicht mehr benötigt wird.

Eine andere Variante der Assoziation demonstriert das folgende Programm. Hier empfängt eine Methode ein Objekt von B als Parameter.

#### Programm ClassA: Übergabe einer Referenz von A nach B

```
public class ClassA {
    String str = "Von wem wird dies ausgegeben?";
    public static void main(String[] args) {
        ClassA a = new ClassA();
        ClassB b = new ClassB(a);    // Hier ist meine Adresse
    }
}
class ClassB {
    ClassB(ClassA a) {
        System.out.println(a.str);
    }
}
```

Diese Art der Verbindung zwischen den Klassen wird praktiziert bei so genannten Callback-Methoden, bei denen sich ein Objekt bei einem anderen Objekt anmeldet, damit es dort verarbeitet wird.

Diese Beziehung kann auch als "benutzt-eine" beschrieben werden.

#### Aggregation

Eine konkretere Assoziation ist die Aggregation. Sie wird in UML durch eine offene Raute an der Linie dargestellt und bezeichnet eine Beziehung, bei der die beteiligten Klassen nicht gleichberechtigt sind, sondern wo eine Ganze-Teile-Beziehung besteht. Das kann die Zugehörigkeit einer Klasse zu einer Sammlung sein (z.B. ein Kundenobjekt als ein Element eines Arrayobjekts) oder auch darin bestehen, dass ein Objekt der Klasse-A sich zusammensetzt aus einem Objekt der Klasse-B, wobei B aber auch unabhängig existieren kann (z.B. ein Kunde hat ein String-Objekt als Attribut).

#### Komposition

Eine noch stärkere Form einer Beziehung zwischen Klassen ist die Komposition. Hier werden Klassen in andere Klasse eingefügt, indem Felder einer Klasse aus Objekten anderer Klassen bestehen, die **existentiell abhängig** sind von der Gesamtklasse. Wird das Gesamtobjekt zerstört, so sind auch die darin enthaltenen Objekte nicht mehr verfügbar.

Beispiel: Es gibt die Klasse *Auto*, die sich u.a. zusammensetzt aus Objekten der Klassen *Rad* (und *Sitz*, *Motor* usw.). Bei der Komposition der Klasse *Auto* ist das *Rad* also vollständig enthalten. Sobald eine Instanz von *Auto* erzeugt wird, wird auch das Datenmember vom Typ *Rad* erzeugt. Die Komposition wird in UML durch eine ausgefüllte Raute dargestellt.

Das folgende Programm demonstriert diese enge Beziehung zwischen den Klassen A und B. Hier sind Objekte von A **enthalten in** Klasse B, sie sind ein Attribut der Klasse B, und das bedeutet, wenn eine Instanz von B erzeugt wird, wird auch ein Objekt von A benötigt.

### Programm ClassTest: Die Klasse B enthält als Membervariable ein Objekt der Klasse A

```
public class ClassTest {
    public static void main(String[] args) {
        ClassA a = new ClassA("Von wem wird dies ausgegeben?");
        ClassB b = new ClassB(a);
        b.ausgeben();
    }
}

class ClassA {
    String str;
    ClassA(String s) {
        str = s;
    }
}

class ClassB {
    private ClassA a;
    ClassB(ClassA a) {
        this.a = a;
    }
    void ausgeben() {
        System.out.println(a.str);
    }
}
```

### Bewertung der unterschiedlichen Klassenbeziehung beim Bilden von neuen Klassen

Die Klassen sind das wichtigste Mittel für "Code-reuse" in objektorientierten Systemen. Java erlaubt es dem Programmierer nicht nur, neue Klasse zu erstellen, sondern bietet auch unterschiedliche Möglichkeiten, bestehende Klassen zu benutzen. Die einfachste Form der Nutzung ist die Instanziierung mit anschließendem Nachrichtenaustausch. Enger wird die Beziehung, wenn eine bestehende Klasse als Attribut in eine andere Klasse eingebettet wird.

Die engste Beziehung jedoch ist die Vererbungsbeziehung, denn bei der Instanziierung werden nicht nur implizit (automatisch) alle Objekte von Oberklassen eingebunden, sondern die Unterklasse ist auch von der Syntax und von der Semantik her "gefesselt" an der Oberklasse. Das hat nicht nur Vorteile:

- Auch Designfehler werden geerbt, d.h. diese pflanzen sich bis in alle Subklassen fort.
- Außerdem werden Modifikationen an Oberklassen umso schwieriger, je tiefer die Schachtelung der Klassenhierarchie ist.

Deswegen lautet eine Empfehlung beim Klassendesign: Entscheiden Sie sich für die Vererbung nur dann, wenn eindeutig eine Spezialisierung einer bestehenden Klasse benötigt wird, im Zweifel ist die Aggregation bzw. Komposition vorzuziehen, weil diese Klassenbeziehungen einfacher und flexibler sind.

### 12.3.3 Objektorientierte Implementierung (OOP)

Nach dem OOD folgt die Umsetzung des Designs in den Java-Quelltext, die Codierung. Dazu müssen die Feinstruktur der Klassen und die Algorithmen der Methoden entwickelt werden. Hier können Entwurfssprachen wie Pseudocode oder Ablaufpläne (siehe Kapitel 9) als Vorstufe zur eigentlichen Codierarbeit hilfreich sein.

#### Encapsulation und Information Hiding

Auch beim Implementieren der Klassen und Methoden sollten unbedingt die schon mehrfach besprochenen Prinzipien der Kapselung und des Information Hiding konsequent eingehalten werden. Die Vorteile zeigen sich spätestens in der Wartungsphase: die innere Struktur einer solchen Klasse kann geändert werden, ohne dass die Clients davon irgendwie berührt sind. Die Klasse (oder komplette Packages) werden ausgetauscht, so wie es aus dem technischen Bereich bekannt ist (z.B. Modul im TV-Gerät wird ausgetauscht, ohne dass andere Teile davon betroffen sind, solange die Schnittstellen gleich bleiben). Ein Beispiel für die Einhaltung dieses Prinzips ist (normalerweise) der Wechsel der JDK-Version: die Anwendungsprogramme müssen nicht geändert werden, obwohl die aufgerufenen Klassen u.U. intern erheblich modifiziert worden sind.

Die Felder einer Klasse sollten möglichst privat sein. Innerhalb einer Methode sollte - wann immer möglich - mit lokalen Variablen gearbeitet werden. Dadurch ist der *Scope* der Daten (siehe Kapitel 16) so schmal wie möglich, d.h. sowohl die Sichtbarkeit als auch die Lebensdauer der Elemente bleiben so kurz wie möglich. Ausnahmen sind zu dokumentieren. Für das Lesen und Verändern von Instanzvariablen werden speziellen Methoden geschrieben: Setter-Methoden zum Modifizieren und Getter-Methoden zum Lesen.

#### Trennung der Ausführungsklassen von Serviceklassen und Interfaces

In der Designphase sollte - wann immer möglich - getrennt werden nach Klassen, die einen Service anbieten, und Klassen, die diesen Service nutzen.

Durch die zusätzliche Erstellung von *interface*-Quelldateien wird die Syntax für den Aufruf eines Services, den eine Klasse bietet, extern beschrieben. Die Interfaces ent-

halten die *Signatures* der Methoden, also die Namen der Methoden und die Anzahl und Datentypen der Parameter. Außerdem enthält das Interface den Datentyp des Ergebnisses. Das sind die Informationen, die der Client benötigt, um den Aufruf syntaktisch korrekt zu codieren. Bei der Codierung dieser unterschiedlichen Sourcefiles kann eine personelle Trennung sinnvoll sein.

### Inkrementell entwickeln

Auch für die Implementierung gilt: Gehen Sie schrittweise vor. Codieren Sie den ersten Abschnitt, wandeln Sie ihn um (denn das bedeutet auch: lassen Sie den Compiler die Syntax prüfen) und testen Sie abschnittsweise. Verfeinern Sie Ihr Programm nach und nach. Ein Beispiel für diese Vorgehensweise enthält Kapitel 9.1.1.

### Überschaubare Größe und einfacher Codierstil

Die einzelnen Klassen und Methoden sollen eine überschaubare Größe haben. Jede Klasse ist einzeln compilierbar, d.h. sie kann eigenständig auf Syntaxfehler überprüft werden.

Methoden implementieren kurze, funktional abgegrenzte Aufgaben und sollten nur in Ausnahmefällen größer sein als 50 - 100 Zeilen

Vermeiden Sie Codiertricks. Solche Lösungen sind häufig so kunstvoll, dass sie schwer verständlich sind. Codieren sollte nicht als Kunst, sondern als eine Technik, die sehr viel Disziplin erfordert, angesehen werden.

### Musterverwendung

Nicht nur in der Designphase sollte auf Pattern zurückgegriffen werden. Auch in der Realisierungsphase gibt es für viele Aufgabenstellung bereits Musterlösungen (weitere Hinweise dazu folgen später).

### Strukturierte Programmierung

Die Reihenfolge, wie Methoden in einer Javaklasse aufgeführt sind, spielt für den Programmablauf keine Rolle, alle sind gleichberechtigt. Ausnahme ist die *main*-Methode. Jede Java-Applikation beginnt mit der *main*-Methode, von dort erfolgt der Aufruf aller anderen Klassen und Methoden dieses Prozesses. Allerdings sollten beim Codieren der *main*-Methode (und auch aller anderen Methoden) die Regeln der Strukturierten Programmierung beachtet werden. Diese sind im Kapitel 9 beschrieben.

### Integrierte Dokumentation (Arbeiten mit *javadoc*)

Kommentare helfen bei der Einarbeitung in den Quelltext. Nicht nur für den Autor sollten die Programme auch nach Monaten noch zu verstehen sein, auch Kollegen kommen in die Situation, fremde Sourcen modifizieren zu müssen. Sie werden dankbar sein für jede plausible Erläuterung.

In der JDK gibt es ein Hilfsprogramm ("tool"), das analysiert eine Quelltextdatei und extrahiert die Kommentare, um daraus eine HTML-Datei zu erstellen. Aufgerufen wird es mit:

```
javadoc *.java
```

nachdem in den Ordner gewechselt wurde, in dem sich die Programme befinden.

Das Tool erkennt nicht nur die eingefügten Kommentare in einem Programm (sofern sie in `/**` und `*/` eingeschlossen sind), sondern auch speziell markierte Absätze, die mit `@` beginnen.

## 12.4 Unified Modeling Language (UML)

Die Unified Modeling Language (UML) ist eine grafische Beschreibungssprache zur Darstellung von objektorientierten Softwaresystemen. Der UML-Standard umfasst mehrere unterschiedliche Arten von Diagrammen. Die wichtigste ist das Klassendiagramm, andere Arten sind Use-Case-Diagramme zur Darstellung von Anwendungsfällen oder Sequenzdiagramme zur Darstellung von Interaktionen.

Das Klassendiagramm beschreibt die Struktur von Klassen und ihre Beziehungen untereinander. Jede Klasse wird als Rechteck dargestellt. Ganz oben im Rechteck steht der Name der Klasse. Danach werden die Attribute und dann die Methoden aufgeführt.

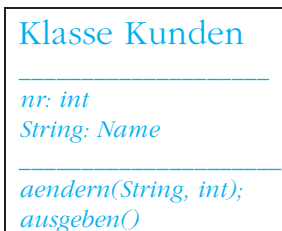


Abb. 12.1: Beispiel für UML-Klassendiagramm

Die Klassen eines Programms (oder Programmsystems) werden durch Linien und spezielle Symbole miteinander verbunden. Diese Symbole stellen die unterschiedlichen Beziehungen (Assoziationen) zwischen den Klassen dar:

- **Aggregation**, dargestellt durch eine Raute, eine "ist-Teil-von"-Beziehung,
- **Komposition**, dargestellt durch eine ausgefüllte Raute, eine physikalische "ist ein Teil von"-Beziehung, also stärker als die Aggregation,
- **Vererbung**, dargestellt durch einen Pfeil, stellt eine Verallgemeinerung bzw. Spezialisierung von Eigenschaften dar, sie wird auch als "ist-ein"-Beziehung bezeichnet.

Hier eine Übersicht über die wichtigsten Elemente eines Klassendiagramms:

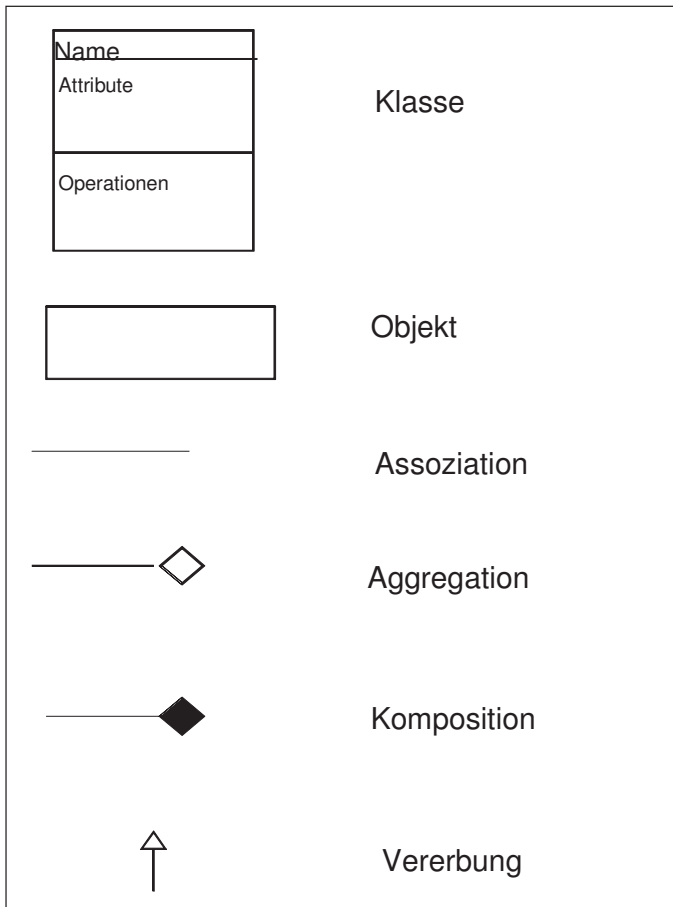


Abb. 12.2: Elemente eines UML-Klassendiagramms

Eine strenge visuelle Unterscheidung zwischen Objekt und Klasse ist in UML nicht vorgesehen, beide werden durch Rechtecke repräsentiert. Die Beschriftung der Kästchen ist weitgehend wahlfrei. Neben diesen Klassendiagrammen sind in UML auch Aktivitätsdiagramme möglich. Diese zeigen ganz allgemeine Abläufe und können auch benutzt werden, um Algorithmen grafisch darzustellen. Ähnlich wie in einem PAP wird durch Aktivitätsdiagramme der Kontrollfluss grafisch dargestellt.

**Vorteil der UML-Notation** im Vergleich zu PAP oder Nassi-Shneidermann: Sie erlaubt auch die Darstellung von speziellen OO-Strukturen wie z.B. Collections, Streams oder Exceptions.

## 12.5 Pattern und Frameworks

### 12.5.1 Design Pattern (Entwurfsmuster)

Objektorientierte Programmierung unterscheidet sich von strukturierter Programmierung nicht nur dadurch, dass Daten und Funktionen konsequent gekapselt und damit geschützt (privat) sind, sondern auch dadurch, dass die Softwaresysteme aus "unendlich" vielen kleinen Objekten bestehen, die miteinander Nachrichten austauschen. Es gibt eine ganze Reihe von Versuchen, den Aufbau der objektorientierten Anwendungen zu standardisieren und zu normieren. Dabei handelt es sich um Empfehlungen, die auf Erfahrungen aus der Praxis beruhen.

Beschrieben werden diese Empfehlungen in so genannten Design Pattern. Diese Entwurfsmuster beschreiben schematische Lösungen, nicht nur für Designprobleme, wie der Name suggeriert, sondern auch für die Codierung von Datenstrukturen und Algorithmen in einer bestimmten Programmiersprache. Sie beschreiben die Lösungsansätze für häufig wiederkehrende Standardprobleme, zusammengetragen und katalogisiert von erfahrenen Programmierern. Es gibt mittlerweile viele Hundert Pattern, veröffentlicht in einigen Dutzend Büchern.

Konkret handelt es sich dabei um in der Praxis bewährte Lösungen in Form von Empfehlungen. Diese können dann mit geringem Modifikationsaufwand so oder so ähnlich eingesetzt werden. Design Pattern basieren damit auf zwei bereits bekannten Konzepten der Softwareentwicklung: Abstraktion und Wiederverwendung.

#### Pattern und Abstraktion

Die Lösungsvorschläge sind unabhängig von konkreten Projekten oder von bestimmten Umgebungen (Hardware- oder Softwareplattform), oft sogar programmiersprachen-unabhängig. Allerdings entstammen die Lösungsvorschläge normalerweise den objektorientierten Architekturen. Konkret sind einige der Design Pattern sogar in die Standard-Klassen der Java-Sprache eingebaut, so z.B. das Factory-Design Pattern oder das Arbeiten mit einem Iterator.

#### Pattern und Wiederverwendung

Wiederverwendung ist natürlich das eigentliche Thema der Entwurfsmuster. Denn sie sind eine wiederverwertbare Vorlage für Problemlösungen, entstanden aus der Erfahrung von vielen Jahren Systementwicklung in vielen Projekten. Dies macht es den Anfängern etwas schwer, die Bedeutung der Design Pattern zu erkennen, denn bei den Vorschlägen handelt es sich meistens um Rezepte für kompliziertere Aufgabenstellungen, gedacht für erfahrene Programmierer, die komplexe Anwendungen realisieren müssen.

Ein Nebeneffekt der Beschäftigung mit Design Pattern ist, dass häufig wiederkehrende Probleme dadurch einen Namen bekommen. Dieser Name ist gleichbedeutend mit einer Standardlösung für dieses Problem. Zum Beispiel weiß jeder erfahrene



Programmierer, welche Lösungsansätze mit *Factory* oder *Singleton* gemeint sind (bei diesen Pattern handelt es sich um genau beschriebene Muster für das Erzeugen von Instanzen).

### 12.5.2 Beispiel 1 für ein Design Pattern: *Iterator*

Ein *Iterator* ist ein Objekt, das es ermöglicht, die Elemente einer *Collection* sequentiell zu durchlaufen. Eine *Collection* ist eine Sammlung von Objekten im Arbeitsspeicher. Eine einfache *Collection* ist in der Standardklasse *Vector* beschrieben. Ein *Vector*-Objekt ist vergleichbar mit einem Array (siehe Kapitel 13), allerdings hat *Vector* den Vorteil, dass seine Größe veränderlich ist. Das heißt, beim Erzeugen des Objekts muss keine Größe angegeben werden, und die Anzahl der Komponenten dieses Vektors kann sich zur Laufzeit verändern.

Das folgende Beispielprogramm erzeugt eine **Collection** aus der Klasse *Vector* und füllt diese Sammlung mit einigen Objekten. Danach soll die *Collection* sequentiell durchlaufen werden, um somit alle Komponenten, die gesammelt worden sind, am Bildschirm anzuzeigen. Für diese Aufgabenstellung ("Iteration durch eine *Collection*") gibt es drei mögliche Codierlösungen: entweder wird eine einfache *for*-Schleife mit einer Laufvariablen benutzt, oder der Programmierer benutzt das **Iterator-Pattern**, oder es wird die **erweiterte *for*-Schleife** codiert.

#### Programm *ArrayList01*: Lösungsvorschlag 1 - *Iterator-Pattern*

```
import java.util.*;
public class ArrayList01 {
    public static void main(String[] args) {
        ArrayList sammlung = new ArrayList();
        sammlung.add("Erstes Objekt");
        sammlung.add("Zweites Objekt");
        sammlung.add("Drittes Objekt");
        sammlung.add("Viertes Objekt");
        // Ausgeben mit Iterator
        Iterator it = sammlung.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

Diese Lösung zeigt, wie ein Objekt der Iteratorklasse eingesetzt werden kann, um eine Sammlung von Objekten, die im Arbeitsspeicher stehen, sequentiell zu verarbeiten. Sie ist unabhängig von der konkreten Implementierung der Objektsammlung, egal ob dies eine Liste oder ein Stack oder eine Queue ist, das *Iterator-Pattern* passt immer. Allerdings zeigt dieses Beispiel auch, dass ein Standard-Rezept nicht immer

die einzige (oder beste) Lösung ist: denn in diesem Fall wird ein erfahrener Java-Programmierer wahrscheinlich die erweiterte For-Schleife vorziehen.

### Programm ArrayList02: Lösungsvorschlag 2 - erweiterter For-Schleife

```
import java.util.*;
public class ArrayList02 {
    public static void main(String[] args) {
        ArrayList sammlung = new ArrayList();
        sammlung.add("Erstes Objekt");
        sammlung.add("Zweites Objekt");
        sammlung.add("Drittes Objekt");
        sammlung.add("Viertes Objekt");
        // Ausgeben mit erweiterter For-Schleife
        for (Object s : sammlung) {
            System.out.println(s);
        }
    }
}
```

Diese Variante, durch eine Collection zu iterieren, ist im Gegensatz zur Iterator-Pattern eine spezielle Java-Möglichkeit. Sie bietet sich an, wenn **alle** Elemente verarbeitet werden sollen und wenn die Verarbeitung kein Update der Collection-Inhalte erfordert.

Der Einsatz des Iterators dagegen ist ein universelles Design Pattern, unabhängig von einer speziellen Programmiersprache. Diese Lösung ist in Java immer dann einzusetzen, wenn die Collection nicht nur gelesen, sondern auch inhaltlich verändert werden soll.

### Übung zum Programm ArrayList02

Lösen Sie die Aufgabe mit einer einfachen *for*-Schleife.

Lösungshinweis: *for (int i=0; i<4; i++) System.out.println(sammlung.get(i));*

### 12.5.3 Beispiel 2 für Design Pattern: MVC (model-view-control)

Ein weiteres bekanntes Design-Pattern ist MVC (model-view-control). Damit wird beschrieben, wie eine Java-Anwendung in drei Teile aufgeteilt werden kann, die jeweils strikt getrennt voneinander als Module realisiert werden. Die Module haben folgende fest definierte Aufgaben:

- das "**model**"-Modul speichert und verarbeitet die Daten ("business-logic")
- die "**view**"-Modul ist zuständig für Anzeige der Daten ("presentation")
- das "**control**"-Modul steuert den Ablauf der gesamten Applikation.

Für die konkrete Umsetzung dieses Entwurfsmusters gibt es im Bereich der Unternehmensanwendungen ("enterprise application") mit Internet-Techniken sogar eigene Java-Programmtypen: für das Control-Modul werden *Servlets*, für das View-Modul *Java-Server-Pages* und für die Realisierung der Model-Aufgabe häufig *Javabeans* oder auch *EJB* (*Enterprise Java Beans*) eingesetzt.

### 12.5.4 Framework

Im Zusammenhang mit Design Pattern und Componententechnik wird häufig der Begriff "Framework" genannt. Leider ist auch diese Bezeichnung nicht einheitlich definiert. Nicht selten ist es nur ein Schlagwort aus dem Bereich der objektorientierten Softwareentwicklung. Wörtlich übersetzt bedeutet Framework Rahmenwerk, und man kann darunter verstehen:

- Ein Programmsystem, dass im Gegensatz zu einer reinen Klassenbibliothek auch eine Anwendungsarchitektur vorgibt. Häufig wird diese Bezeichnung auch als Marketingbezeichnung für objektorientierte Software-Produkte benutzt, die den **Lösungsrahmen für einen vorgegebenen Problembereich** bilden, oder anders gesagt: Sie sind Halbfabrikate, die in konkreten Kundensituationen angepasst und ergänzt werden müssen zu Endprodukten. In diesem Sinne gibt es auf dem Markt der Anwendungen Frameworks für jede denkbare Aufgabenstellung, z.B. für Buchhaltungssysteme, elektronische Warenhäuser im WWW, spezielle Lösungen für Clientanwendungen, es gibt 3D-Frameworks usw.
- Eine Paketsammlung, die für ein bestimmtes Anwendungsgebiet **neben der reinen API-Beschreibung in Form von Java-Interface auch die Implementierung** enthält. In diesem Sinne gibt es Framework-Beispiele sogar innerhalb der Standard-Library, integriert in die Klassenbibliothek von J2SE: das Collection-Framework oder das Swing-Framework.
- **Ergänzungen zu dem Java-Standard-API**, die (noch) kein offizieller Standard sind. So gibt es Frameworks, die eine optionale Erweiterung der J2SE-Plattform bieten, z.B. das JMF (Java Media Framework). Dabei handelt es sich um eine Java-Bibliothek mit Programmsystemen für das Arbeiten mit Audiosignalen (Mikrofon) und Videosignalen (Kamera).

Die allgemeinste und vielleicht treffendste Definition des Framework-Begriffs ist:

**Ein Framework enthält die Architektur einer Anwendung. Es definiert nicht nur die Bestandteile und die Struktur eines Programmsystems, sondern es implementiert auch Teile davon.**

Grundprinzipien der Frameworkarchitektur sind der Einsatz von Design Pattern und die Anwendung der Componententechnik. Beispiele für Frameworks sind JavaBeans, Struts und Webservices.

# 13

## Reihungen benutzen ("arrays")

---

Die meisten Programmiersprachen bieten Möglichkeiten, mehrere gleichartige Variablen im Arbeitsspeicher zu einer Einheit zusammen zu fassen. Dadurch wird die Verwaltung und Verarbeitung der einzelnen Komponenten vereinfacht. Diese Sammlungen haben - je nach Programmiersprache - unterschiedliche Bezeichnungen: sie werden Array, Tabelle oder Vektor genannt. Manchmal heißen sie auch einfach "Feld". Diese Bezeichnung ist im Java-Umfeld aber unpassend, denn damit sind üblicherweise die Attribute einer Klasse gemeint, also die Membervariablen.

Die in Java korrekte Bezeichnung ist Array (oder auf Deutsch: Reihung). Sie werden in diesem Kapitel lernen,

- was Arrays sind,
- welche Verarbeitungsmöglichkeiten es dafür gibt,
- worin die Vor- und Nachteile liegen im Vergleich etwa zu anderen Sammlungen, die in Java möglich sind,
- was zu beachten ist, wenn Arrays als Methodenparameter oder als Returnwert eingesetzt werden.

### 13.1 Erzeugen von Arrays

Ein Gruppe von Daten, die

- **gleichartig** sind (gleicher Datentyp, gleicher Name),
- **zusammenhängend** im Arbeitsspeicher stehen und nur
- durch eine "Platznummer" (**Index**) unterschieden werden,

nennt man Reihung (engl. array).

Arrays sind in Java echte Objekte, obwohl es keine Klasse gibt, von der sie erzeugt werden. Gleichwohl werden sie mit dem Schlüsselwort *new* erstellt. Äußerliches Merkmal dafür, dass es sich um den Datentyp *Array* handelt, ist die Verwendung von eckigen Klammern [ ] bei der Deklaration der Referenzvariable:

```
int[] zahlenreihe;
```

Mit dieser Deklaration wird die Referenz zu einem Arrayobjekt erstellt - oder besser gesagt, eine Variable, die die Fähigkeit hat, auf ein derartiges Objekt zu zeigen (denn es gibt ja noch kein Arrayobjekt). Der Identifier (hier: *zahlenreihe*) wird auch Arrayvariable genannt. Der Datentyp *int* gilt für jedes einzelne Mitglied der Gruppe

(Komponente, Element). Die eckige Klammer steht hinter dem Datentyp, sie kann auch hinter dem Identifier stehen, doch diese Schreibweise wird nicht empfohlen.

Das eigentliche Arrayobjekt kann - wie bei Objekten üblich - mit *new* erzeugt werden:

```
zahlenreihe = new int[5];
```

Mit dieser Anweisung werden dynamisch (zur Laufzeit) im Arbeitsspeicher 5 *int*-Elemente angelegt, alle unter dem Namen *zahlenreihe* ansprechbar. Sie werden unterschieden durch eine Platznummer (Index), z.B.

```
zahlenreihe[2] = 15;
```

Mit dieser Wertezuweisung wird die dritte(!) Komponente aus der Zahlenreihe mit dem Wert 15 gefüllt. Bitte beachten Sie: der Index muss ganzzahlig sein, und die 2 adressiert die 3. Zahl, weil die Nummerierung der Plätze bei 0 beginnt.

### Programm *Array01*: Ein Array erstellen und mit Werten füllen

```
public class Array01 {
    public static void main(String[] args) {
        int[] umsatz;
        umsatz = new int[3];
        umsatz[0] = 100;
        umsatz[1] = 200;
        umsatz[2] = 300;
    }
}
```

Durch die Deklaration `int[] umsatz;` wird eine Referenzvariable angelegt, die die Fähigkeit hat, auf eine Reihung von Integer-Werten zu verweisen. Durch das anschließende Statement `umsatz = new umsatz[3];` werden 3 Arbeitsspeicherplätze angelegt, jeder ist 4 Bytes lang. Jeder hat die Fähigkeit, Integerwerte aufzunehmen. Adressiert werden die einzelnen Plätze mit einem Index von 0 bis 2.



Abb.13.1: Array im Arbeitsspeicher

Fazit: Bei der Definition wird **die Anzahl** der Elemente angegeben und nicht etwa die höchste Platz-Nummer. Die **Platz-Nummern** (Indices) beginnen bei 0, deswegen ist der höchste Index für die Reihung *umsatz* die 2.

Die letzten Anweisungen sorgen dafür, dass die drei Plätze der Reihung mit Werten gefüllt werden.

Die Angabe der Komponentenanzahl kann fehlen, wenn das Array als formaler Parameter einer Methode definiert wird (ein typisches Beispiel dafür enthält die *main*-Methode, dort ist ein String-Array als Empfangsparameter definiert).

Aber es gibt in Java keine Möglichkeit, die Anzahl der Elemente zu ändern, wenn das Array einmal im Arbeitsspeicher angelegt ist. Man sagt, die **Größe eines Arrays ist statisch**. Werden dynamische Arrays benötigt, so muss die Standardklasse *Vector* benutzt werden (oder eine andere Klasse des Frameworks *Collection*).

### Übung

Erstellen Sie ein neues Programm mit einer Referenzvariablen. Diese Variable soll die Fähigkeit haben, auf ein Array mit *float*-Variablen zu referenzieren. Die Referenzvariable soll den Identifier *zahlen* haben. Danach soll das Array im Arbeitsspeicher angelegt werden. Benötigt werden 4 Elemente.

### Lösungsvorschlag

```
public class Array02 {  
    public static void main(String[] args) {  
        float[] zahlen;  
        zahlen = new float[4];  
    }  
}
```

## 13.2 Initialisieren von Arrays

Beim Anlegen des Speicherplatzes für das eigentliche Arrayobjekt ist es unbedingt erforderlich, dass Angaben zu der Größe des Arrays gemacht werden. Die Größe ergibt sich aus der Anzahl der Komponenten der Reihung, und die kann auf zwei Arten festgelegt werden.

Zunächst ist es möglich, dass im Deklarationsstatement eine **Ganzzahl** (oder ein Ausdruck, der eine Ganzzahl ergibt) steht:

```
int[] zahlenreihe = new zahlenreihe[a + 5];
```

Bei dieser Schreibweise werden die einzelnen Komponenten mit Defaultwerten initialisiert, abhängig vom Datentyp.

Eine andere Möglichkeit besteht darin, die Elemente bei ihrer Deklaration mit individuellen Anfangswerten zu füllen (zu "initialisieren"). In diesem Fall kann die Angabe der Anzahl entfallen, sie ergibt sich aus der Anzahl der Werte, die als **kommagetrennte Liste** in geschweiften Klammern steht:

```
int[] zahlenreihe = {5, 3, 4, 17, 21};
```

Bei dieser Kurzschreibweise entfällt auch das Schlüsselwort *new*, der Compiler ermittelt die Anzahl der Elemente aus der Liste selbst.

**Programm Array03: Deklaration und individuelle Initialisierung eines Arrays**

```
public class Array03 {
    public static void main(String[] args) {
        int[] umsatz = {100, 200, 300, 0, 0};
        umsatz[4] = 1100;
        System.out.println(umsatz[1]);
    }
}
```

In der Zeile 3 wird die Referenzvariable *umsatz* deklariert. Gleichzeitig wird das Arrayobjekt erzeugt und den fünf Komponenten die Werte aus der Liste der fünf *int*-Literele zugewiesen. Dadurch werden die Komponenten des Arrays initialisiert. Die darauf folgende einzelne Wertezuweisung adressiert das letzte Element in der Reihung. Und mit der letzten Anweisung wird der Wert des zweiten Elements ausgegeben.

**Übung**

Erstellen Sie ein neues Programm *Array04*. Deklarieren Sie darin ein Array, das aus 5 *char*-Elementen besteht. Die Elemente sollen initialisiert werden mit den Kleinbuchstaben von a bis e. Benutzen Sie für die Initialisierung die Kurzschreibweise, damit die explizite Angabe der Arraygröße ("Dimension") entfallen kann.

**Lösungsvorschlag**

```
public class Array04 {
    public static void main(String[] args) {
        char[] buchstaben = {'a', 'b', 'c', 'd', 'e'};
        System.out.println(buchstaben[3]);
    }
}
```

Wenn der Programmierer keine Anfangswerte vorgibt, werden die Komponenten eines Arrays mit Standardwerten initialisiert. Beispielsweise bekommen numerische Typen den Wert 0. Das folgende Programm definiert ein Array für die Aufnahme von **Referenzen auf Objekte** (siehe auch Abschnitt 13.4). Aber weil noch keine Objekte erzeugt worden sind, enthalten die Komponenten den Anfangswert *null*.

**Programm Array05: Die Komponenten werden mit Defaultwerten vorbelegt**

```
public class Array05 {
    public static void main (String args[]) {
        Object[] obj = new Object[3];
        System.out.println(obj[0]);
    }
}
```

### Übung zum Programm Array05

Überprüfen Sie durch Programmänderung, wie der Defaultwert lautet, wenn das Array aus *boolean*-Typen besteht (*false* oder *true*?).

Das folgende Programm kann nicht fehlerfrei umgewandelt werden.

### Programm Array06: Umwandlungsfehler " .. not have been initialized"

```
public class Array06 {
    public static void main (String args[]) {
        float[] zahlen;
        zahlen[0] = 15.4f;
    }
}
```

### Übung zum Programm Array06

Klären Sie die Ursache des Syntaxfehlers und korrigieren Sie den Fehler.

Lösungshinweis: Arrayobjekte müssen im Arbeitsspeicher mit *new* erzeugt werden.

Fazit aus den letzten beiden Übungen: Beim Initialisieren von Arrayobjekten muss sauber unterschieden werden zwischen der Initialisierung der Referenzvariablen (dies geschieht durch Aufruf von *new*) und der Belegung der einzelnen Komponenten mit Anfangswerten (dies geschieht dabei automatisch).

## 13.3 Zugriff auf die Array-Komponenten

Die einzelnen Variablen, aus denen sich das Array zusammensetzt, werden Komponenten (manchmal auch Elemente) genannt. Sie werden unterschieden über ihre Platz-Nummer innerhalb des Arrays, über den Index. Über diesen Index sind die einzelnen Elemente einer Reihung direkt ansprechbar - ohne Aufruf einer Methode, z.B.

```
    betraege[0] = 125.23;
```

Der Index muss eine Ganzzahl sein (entweder direkt als Literal oder als Ausdruck, der einen *int*-Wert ergibt). Spätestens zur Laufzeit des Programms wird sichergestellt, dass nur gültige Indices verarbeitet werden, andernfalls gibt es Fehlermeldungen ("Exceptions") durch die Java Virtuelle Maschine (JVM). Die Indices beginnen bei 0 zu zählen.

### 13.3.1 Vorteile beim Arbeiten mit Arrays

Durch den Einsatz eines Arrays wird die Deklaration, Verwaltung und Verarbeitung der einzelnen Komponenten erheblich vereinfacht. Sie brauchen nur einmal deklariert zu werden. Sie können einzeln oder als Ganzes verarbeitet werden. Man kann direkt auf einzelne Komponenten zugreifen oder man kann Schleifen formulieren, um sequentiell durch alle Elemente zu iterieren. Besonders elegant ist das Arbeiten mit der *for*-Schleife.



**Programm Array07: Iterieren durch ein Array und Zugriff auf jedes Element**

```
public class Array07 {  
    public static void main(String[] args) {  
        float[] zahlen;  
        zahlen = new float[4];  
        for (int i=0; i<4; i++)  
            zahlen[i] = i + 125.0f;  
    }  
}
```

Eine weitere Verarbeitungsvariante besteht darin, die Gruppe der Elemente als Ganzes zu verarbeiten. Dazu bietet die Klasse *java.util.Arrays* (siehe Abschnitt 13.5: Class *Arrays*) eine Fülle von Methoden, z.B. die Methode *fill*.

**Programm Array08: Das Array als Ganzes verarbeiten**

```
public class Array08 {  
    public static void main(String[] args) {  
        float[] zahlen;  
        zahlen = new float[4];  
        java.util.Arrays.fill(zahlen, 12.45f);  
        for (float zahl : zahlen)  
            System.out.println(zahl);  
    }  
}
```

**Übung zum Programm Array08**

Klären Sie anhand der API-Dokumentation, mit welcher Methode der Klasse *Arrays* ein *float*-Array in eine Stringrepräsentation umgewandelt werden kann. Ergänzen Sie das Programm um einen entsprechenden Ausgabebefehl.

**Lösungshinweis:**

```
System.out.println(java.util.Arrays.toString(zahlen));
```

**13.3.2 Prüfung des Index durch Run-Time-Umgebung**

Jeder Array-Zugriff wird von der Laufzeitumgebung überprüft. Der Versuch, einen Index zu verwenden, der kleiner als Null ist oder größer oder gleich der Anzahl der Elemente, führt zu einer Exception.

Es kann in Java nicht passieren, dass durch einen falschen Index Arbeitsspeicherbereiche adressiert (d.h. gelesen bzw. geändert) werden, die außerhalb des Arrays liegen. Dies wird von der JVM sichergestellt.

**Programm Array09: Falsche Adressierung einer Array-Komponente**

```
public class Array09 {  
    public static void main(String[] args) {  
        char[] buchstaben = {'a','b','c','d','e'};  
        for(int i=0; i<6; i++) {  
            System.out.println(buchstaben[i]);  
        }  
    }  
}
```

**Übung zum Programm Array09**

Bei der Ausführung gibt es die Fehlermeldung "ArrayIndexOutOfBoundsException: 5". Sinngemäß bedeutet das, dass der Index 5 einen Speicher außerhalb dieser Arraygrenzen adressiert. Korrigieren Sie das Programm, so dass es fehlerfrei ausgeführt werden kann.

**13.3.3 Member-Variable *length* enthält die Anzahl der Komponenten**

Zu jedem Array gibt es eine zusätzliche, eingebaute Variable, die als Wert die Anzahl der Array-Elemente enthält, die Variable *length*. Sie kann z.B. im Kopf einer *for*-Schleife benutzt werden, um die Durchlaufbedingung zu formulieren.

**Programm Array10: Arbeiten mit der Variablen *length***

```
public class Array10 {  
    public static void main(String[] args) {  
        char[] buchstaben = {'a','b','c'};  
        for(int i=0; i<buchstaben.length; i++) {  
            System.out.println(buchstaben[i]);  
        }  
    }  
}
```

Der Vorteil dieser Schreibweise liegt zum einen darin, dass der Programmierer nicht abzählen muss, wieviel Elemente das Array enthält. Zum anderen ist sie aussagefähiger: man erkennt sofort, dass alle Elemente verarbeitet werden.

**Übungen zum Programm Array10**

**Übung 1:** Ergänzen Sie das Programm so, dass der Wert der Variablen *length* auf der Konsole ausgegeben wird.

**Übung 2:** Bitte ändern Sie das Programm so, dass mit einer For-Each-Schleife das Array durchsucht wird nach dem Buchstaben b. Wenn dieser Wert gefunden wird, soll er ausgegeben und die Suche abgebrochen werden.

**Lösungsvorschlag**

```
public class Array11 {
    public static void main(String[] args) {
        char[] buchstaben = {'a', 'b', 'c'};
        for (char buchstabe: buchstaben) {
            if (buchstabe == 'b') {
                System.out.println(buchstabe);
                break;
            }
        }
    }
}
```

Die Variable *length* enthält nicht die Anzahl der **gefüllten** Komponenten, sondern die Gesamtanzahl aller Elemente, unabhängig davon, ob sie mit individuellen Werten gefüllt worden sind oder ob sie die Default-Initwerte enthalten.

Wenn also nur die gefüllten Elemente verarbeitet werden sollen, so muss die entsprechende Abfrage vom Programmierer codiert werden.

**Übung**

Erstellen Sie ein neues Programm *Array12*. Darin soll ein Array definiert werden, das die Zahlen 1 - 10 in seinen 10 Komponenten speichert. Die Werte der Komponenten sollen in einer *for*-Schleife durch Wertezuweisung vergeben werden.

Danach soll in einer zweiten *for*-Schleife das Array iteriert und die Inhalte aufaddiert werden. Das Ergebnis wird auf der Konsole ausgegeben.

**Lösungsvorschlag:**

```
public class Array12 {
    public static void main(String[] args) {
        int[] zahlen = new int[10];
        for (int i=0; i<10; i++) {
            zahlen[i] = i+1;
        }
        int summe = 0;
        for (int wert: zahlen) {
            summe = summe + wert;
        }
        System.out.println("Die Summe ist: " + summe);
    }
}
```

## 13.4 Objekte in Arrays sammeln

Arrays können nicht nur Werte von primitiven Datentypen enthalten, sondern sie können auch Objekte aufnehmen. Dann werden aber nicht die Objekte selbst im Array gespeichert, sondern lediglich die Referenzvariablen. Die Objekte müssen vom gleichen Typ sein, Verwandte sind erlaubt (d.h. es können auch Sub-Typen der definierten Klasse gespeichert werden).

### Programm *Array14*: Sammeln von Instanzen in einem Array

```
import java.util.*;
public class Array14 {
    public static void main(String[] args) {
        Calendar[] tage = new Calendar[3];
        for (int i=0; i<3; i++)
            tage[i] = Calendar.getInstance();
        System.out.println(tage[1]);
    }
}
```

Das Programm *Array14* erzeugt eine Referenzvariable *tage*. Diese hat die Fähigkeit, auf ein Array zu zeigen, das Objekte vom Typ *Calendar* sammelt. Mit *new* wird dieses Array erzeugt. Es enthält drei Komponenten, jede referenziert auf ein *Calendar*-Objekt.

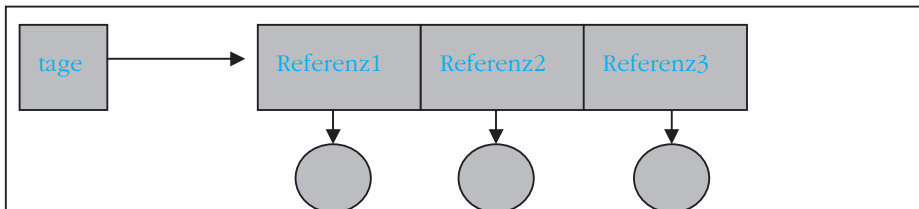


Abb.13.2: Array mit Referenzen auf 3 Objekte

Beim Sammeln und Verwalten von Objekten im Hauptspeicher ist es typisch, dass die Anzahl der Objekte, die gespeichert werden sollen, häufig stark variiert. Für diese Fälle ist ein Array nicht gut geeignet, weil dessen Größe statisch ist. Bessere Lösungen bietet das *Collection*-Framework von Java. Dort gibt es eine ganze Anzahl unterschiedlicher Klassen, die als Objekt-Sammlungen benutzt werden können, z.B. die *Vector*-Klasse. Der Programmierer kann damit nach der Erzeugung des Vectors beliebig viele Objekte eines Typs sammeln und verarbeiten. Java sorgt für die dynamische Anpassung der Länge.

### Übung zum Programm *Array14*

Prüfen Sie, ob anstelle des Referenztyps *Calendar* in Zeile 4 auch *Object* erlaubt ist. Wenn ja, erklären Sie warum (die Lösung steht im ersten Absatz dieses Abschnitts).

### 13.5 Methoden der Class Arrays

Für das komfortable Arbeiten mit kompletten Arrays gibt es die Standardklasse *Arrays* (Achtung: das anhängende s ist wichtig!). Sie enthält einige *static*-Methoden, die vom Programmierer genutzt werden können, um Arrays z.B.

- zu sortieren (*sort*)
- zu vergleichen (*equals*) oder
- binär zu durchsuchen (*binarySearch*).

#### Programm *Array15*: Sortieren mit der Class *Arrays*

```
public class Array15 {  
    public static void main(String[] args) {  
        int umsatz[] = {100,200,300,50,100,5,0,98,700,50,0,90};  
        java.util.Arrays.sort(umsatz);  
        for (int wert: umsatz) {  
            System.out.println(wert);  
        }  
    }  
}
```

Im Programm *Array15* wird die Klasse *Arrays* vollqualifiziert angesprochen. Dadurch kann auf eine *import*-Anweisung verzichtet werden.

#### Übung

Erstellen Sie ein neues Programm *Array16* mit einem Array, das aus 4 Integer-Elementen besteht. Initialisieren Sie das Array mit den Werten 11, 18, 3, 15. Geben Sie die (unsortierten) Werte am Bildschirm aus, sortieren Sie danach die Werte und geben Sie erneut den Array-Inhalt aus. Arbeiten Sie mit der *import*-Anweisung.

#### Lösungsvorschlag:

```
import java.util.*;  
public class Array16 {  
    public static void main(String[] args) {  
        int[] zahlen = {11, 23, 4, 15};  
        for (int zahl: zahlen)  
            System.out.format("%d ", zahl);  
        Arrays.sort(zahlen);  
        System.out.println('\n');  
        for (int zahl: zahlen) {  
            System.out.format("%d ", zahl);  
        }  
    }  
}
```

## 13.6 Mehrdimensionale Arrays

Ein Array kann Objekte von jedem anderen Datentyp enthalten, also auch vom Typ `Array`. Das heißt, in Java werden mehrdimensionale Arrays als Arrays definiert, deren Elemente wiederum Arrays sind (verschachtelte Arrays, "array of array"). Die Definition

```
int matrix[4][3];
```

vereinbart eine Matrix vom Typ `int`. Es werden 4 `int`-Elemente angelegt, die jeweils wieder aus 3 `int`-Elementen bestehen. Der Zugriff auf die einzelnen Elemente erfolgt unter Angabe der beiden Indizes, die jeweils in eckigen Klammern eingeschlossen sind:

```
matrix[2][1] = 8;
```

Damit wird das 3. Element adressiert, und innerhalb dieses 3. Elements das 2. Element.

Bei der Deklaration wird pro Schachtelungstiefe ein eckiges Klammernpaar benötigt. Und beim Zugriff werden soviel Klammernpaare angegeben, wie die Dimension des Arrays angibt. Für die Verarbeitung bieten sich geschachtelte *for*-Schleifen an.

### 13.6.1 Initialisierung von mehrdimensionalen Feldern

Mehrdimensionale Felder können bei der Deklaration ebenfalls mit Anfangswerten versehen werden. Die folgende Definition initialisiert eine `int`-Matrix gleichzeitig mit der Deklaration:

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

Gedanklich kann diese Definition so übersetzt werden: Erstelle ein Array aus zwei Zeilen mit jeweils drei Spalten. Das gleiche Ergebnis wird durch folgendes Programm erreicht:

#### Programm Array20: Zweidimensionales Array mit primitiven Typen

```
public class Array20 {
    public static void main(String[] args) {
        int[][] matrix;
        matrix = new int [2][3];
        matrix[0][0] = 1;
        matrix[0][1] = 2;
        matrix[0][2] = 3;
        matrix[1][0] = 4;
        matrix[1][1] = 5;
        matrix[1][2] = 6;
    }
}
```

### 13.6.2 Zugriff auf die Array-Elemente

Das folgende Programm erzeugt ein 2-dimensionales Array für die Speicherung von Strings. Es sollen 4 Sätze verarbeitet werden, die jeweils aus mehreren Wörtern bestehen. Die maximale Anzahl von Wörtern pro Satz ist 3.

#### Programm *Array21*: Zweidimensionales Array mit Objekten

```
public class Array21 {
    public static void main(String[] args) {
        String[][] woertermatrix = {
            {"Wort1", "Wort2", "Wort3"},
            {"Wort4", "Wort5", "Wort6"},
            {"Wort7", "Wort8", "Wort9"},
            {"Wort10", "Wort11", "Wort12"}
        };
        for(int i=0; i<4; i++) {
            for (int j=0; j<3; j++)
                System.out.println(woertermatrix[i][j]);
        }
    }
}
```

#### Übung

Bitte erstellen Sie ein Programm *Array22* mit einem 2-dimensionalen Array, das aus drei Elementen besteht. Jedes der drei Elemente soll wieder aus einem Array bestehen, nämlich jeweils aus einer Reihung von vier *char*-Werten. Im Programm soll durch Wertezuweisung in einer **geschachtelten *for*-Schleife** die ersten vier *char*-Variablen mit den Kleinbuchstaben a - d, die nächsten mit den Buchstaben e - h und die letzten vier Variablen mit den Zeichen i - l gefüllt werden.

Die Ausgabe soll wie folgt aussehen:

```
a b c d
e f g h
i j k l
```

#### Lösungsvorschlag

```
public class Array22 {
    public static void main(String[] args) {
        char[][] zeichen;
        zeichen = new char[3][4];
        char unicode = 97;
        // Fuellen mit Zeichen
        for(int i=0; i<3; i++) {
            for (int j=0; j<4; j++)
```

```
        zeichen[i][j] = unicode++;
    }
    // Ausgeben der Zeichen
    for(int i=0; i<3; i++) {
        for (int j=0; j<4; j++) {
            System.out.print(zeichen[i][j]);
        }
        System.out.println();
    }
}
```

### 13.7 Arrays als Parameter und Returnwert bei Methoden

Arrays sind keine einfachen Datentypen, sondern Referenztypen. Damit gilt für sie die Objektsemantik - und das bedeutet z.B. bei der Übergabe von Arrays als Parameter, dass der Empfänger mit dem Originalarray arbeitet und nicht mit einer Kopie, denn er bekommt die Referenz übergeben.

#### Programm *Array23*: Array als Argument beim Methodenaufruf

```
public class Array23 {
    public static void main(String[] args) {
        char[] buchstaben = {'a','b'};
        tauschen(buchstaben);
        for (char buchstabe: buchstaben)
            System.out.println(buchstabe);
    }
    static void tauschen(char[] b) {
        char hilf = b[0];
        b[0] = b[1];
        b[1] = hilf;
    }
}
```

#### Mehrere Returnwerte als Array zurückgeben

In dem nächsten Programm *Array24* wird ein Array als Möglichkeit benutzt, um mehr als ein Ergebnis an den Aufrufer einer Methode zurück zu geben. Das Programm enthält eine Methode, die eine Ganzzahl durch 3 dividiert und dieses Ergebnis zurückgibt. Gleichzeitig soll ein zweites Ergebnis, nämlich der ganzzahlige Rest, geliefert werden. Deshalb wird innerhalb der Methode eine lokale Array-Variable deklariert, mit den beiden Ergebnissen gefüllt und dann mit *return* die Referenz an den Aufrufer übertragen.



**Programm Array24: Array als Returnwert einer Methode**

```
public class Array24 {
    public static void main(String[] args) {
        int zahl = 125;
        int[] erg = dividieren(zahl);
        System.out.println("125 / 3 = " + erg[0]);
        System.out.println("Ganzzahliger Rest: " + erg[1]);
    }
    static int[] dividieren(int z) {
        int[] erg = {0,1};
        erg[0] = z / 3;
        erg[1] = z % 3;
        return erg;
    }
}
```

**13.8 Zusammenfassung****13.8.1 Arrays lernen in 21 Sekunden**

- Arrays sind Objekte. Sie werden mit dem Schlüsselwort *new* erzeugt.
- Aber: Arrays sind "klassenlose" Objekte, es gibt für sie keine explizite Klasse und von daher gibt es auch eine Subklassen von Arrays. Und dennoch: Arrays haben Membervariablen (*length*) und sie besitzen alle Methoden aus der Klasse *Object*.
- Für das Arbeiten mit Arrays gibt es eine besondere Syntax, z.B. beim Erzeugen und Initialisieren.
- Eine Arrayvariable ist eine Referenzvariable, erkennbar an der eckigen Klammer hinter dem Datentyp (oder hinter dem Namen).
- **Arrays haben eine feste Länge.** Die Kapazität (Größe) wird bei der Erzeugung des Objects mit *new* angegeben oder sie wird vom Compiler ermittelt anhand einer Liste von Initialwerten. Danach kann die Größe nicht mehr geändert werden.
- Individuelle Initialwerte können bei der Definition angegeben werden. Sie stehen in geschweiften Klammern, das Schlüsselwort *new* kann in diesem Fall entfallen.
- Arrays können primitive Datenwerte oder auch Referenzen auf Objekte enthalten. Allerdings müssen alle Elemente denselben Datentyp haben.
- Der Index zum Zugriff auf einzelne Elemente beginnt bei 0 zu zählen. Wenn der Index einen Speicherplatz außerhalb des Arraybereichs adressiert, gibt es Run-Time-Error ("Exception").

- Das Arbeiten mit Arrays ist **schnell**:
  - der Zugriff auf die einzelnen Komponenten erfolgt direkt, ohne Aufruf einer Methode,
  - die Zugriffsgeschwindigkeit für das Lesen, Schreiben, Löschen einer Komponente ist unabhängig von der Größe des Arrays.
- Das Arbeiten mit Arrays ist **sicher**:
  - es erfolgt eine Datentyp-Prüfung zur Umwandlungszeit und
  - es erfolgt eine Indexprüfung zur Laufzeit.

Wenn allerdings variabel große Sammlungen mit komfortablen Methoden benötigt werden, so reichen die Möglichkeiten des Arrays nicht. Dann müssen Klassen aus dem *Collection*-Framework (z.B. *Vector* oder *Map*) benutzt werden.

### 13.8.2 Unterschiedliche Codiermuster für das Kopieren von Arrays

Die folgenden Beispiele lösen **alle dieselbe Aufgabe**: es soll ein Array *z1* kopiert werden. Dabei werden noch einmal etliche der besprochenen Verarbeitungsmöglichkeiten demonstriert und auch gezeigt, worauf besonders zu achten ist.

#### Programm *Array30*: Es wird nur der Verweis kopiert

```
public class Array30 {
    public static void main(String[] args) {
        int[] z1 = {11, 23, 4, 15};
        int[] z2;
        z2 = z1;                // Referenz-Semantik !
        for (int z : z2)
            System.out.println(z);
    }
}
```

#### Übung zum Programm *Array30*

Bitte klären Sie durch Programmänderung, ob folgende zusätzliche Anweisung bewirkt, dass sich auch das Array *z2* ändert: *z1[2] = 100;*

#### Programm *Array31*: Per Schleife einzelne Werte kopieren

```
public class Array31 {
    public static void main(String[] args) {
        int[] z1 = {11, 23, 4, 15};
        int[] z2 = new int[z1.length];
        for (int i=0; i<z1.length; i++)
            z2[i] = z1[i];
    }
}
```

```
        for (int z : z2)
            System.out.println(z);
    }
}
```

### Übung zum Programm Array31

Bitte klären Sie durch Programmänderung, ob folgende zusätzliche Anweisung (eingefügt hinter Zeile 6) bewirkt, dass sich auch das Array `z2` ändert: `z1[2] = 100;`

Die `System`-Klasse enthält diverse nützliche *static*-Felder und mehr als 20 *static*-Methoden, u.a. auch die Methode *arraycopy*, die benutzt werden kann, um Arrays komplett oder teilweise zu kopieren.

### Programm Array32: Mit einer System-Methode kopieren

```
public class Array32 {
    public static void main(String[] args) {
        int[] z1 = {11, 23, 4, 15};
        int[] z2 = new int[z1.length];
        System.arraycopy(z1, 0, z2, 0, z1.length);
        for (int z : z2)
            System.out.println(z);
    }
}
```

Die Klasse *Object* ist die "Mutter aller Klassen", jede andere Klasse ist direkt oder indirekt davon abgeleitet. Das bedeutet, dass die Methoden dieser Klasse allen anderen Java-Klassen zur Verfügung stehen. Dazu gehören u.a. die Methoden *toString*, *equals* und auch die Methode *clone*.

### Programm Array33: Benutzen der geerbten Methode clone

```
public class Array33 {
    public static void main(String[] args) {
        int[] z1 = {11, 23, 4, 15};
        int[] z2 = z1.clone();
        for (int z : z2)
            System.out.println(z);
    }
}
```

### Übung zum Programm Array33

Bitte klären Sie auch hier durch Programmänderung, ob folgende zusätzliche Anweisung (eingefügt hinter Zeile 4) bewirkt, dass sich auch das Array `z2` ändert:

```
        z1[2] = 100;
```

# 14

## Zeichenketten anwenden ("strings")

---

Eine Zeichenkette besteht aus *mehreren* Zeichen. Dadurch unterscheidet sie sich von den eingebauten Datentypen wie *int* oder *char*, die aus nur *einer* Zahl oder aus nur *einem* einzelnen Zeichen bestehen.

Die wichtigste Klasse für das Arbeiten mit Zeichenketten (strings) ist die vordefinierte Klasse *String*. Sie bietet die Möglichkeit, einen beliebig langen String zu speichern. Für jedes Zeichen der Zeichenkette werden im Arbeitsspeicher 16 bit belegt, denn es wird der Unicode (UTF-16) benutzt. Die Klasse stellt Methoden für das Arbeiten mit diesem String zur Verfügung. Außerdem stellt die Klasse die Operatoren `+` und `+=` für die Verkettung von Strings zur Verfügung.

Sie haben bereits in vielen Übungen dieses Buches Zeichenketten benutzt. In diesem Kapitel werden wir dieses Thema vertiefen und folgende Fragen behandeln:

- Was sind die grundlegenden Operationen mit Strings?
- Wie können die verschiedenen Stringklassen *String*, *StringBuffer* und *StringBuilder* abgegrenzt werden?
- Welche Möglichkeiten gibt es, Texte in Dateien oder Zeilen am Bildschirm zu parsen (aufzuteilen)?
- Was sind "Reguläre Ausdrücke" und wie helfen sie bei der Textanalyse?

### 14.1 Erstellen von String-Objekten

Genauso wie Arrays sind auch Strings Objekte. Damit besteht auch dieser Datentyp aus zwei Teilen: zum einen aus der Referenzvariablen und zum anderen aus dem eigentlichen Stringobjekt. Die Referenzvariable enthält als Wert eine Referenz auf den aktuellen Wert des Stringobjekts.

Die Definition eines Strings erfolgt in zwei Schritten. Zunächst wird die Referenzvariable deklariert:

```
String str;
```

Dann muss der Speicherplatz für das eigentliche Stringobjekt angelegt und mit einem Wert gefüllt werden, z.B. durch ein Stringliteral:

```
str = new String("Dies ist eine Zeichenkette");
```

Beide Schritte können zusammengefasst werden durch

```
String str = new String("Dies ist eine Zeichenkette");
```

Eine Besonderheit der Stringklasse ist die Kurzform der Definition, ohne das Schlüsselwort *new*:

```
String str = "Dies ist eine Zeichenkette";
```

### Programm *String01*: Referenzvariable erzeugen (fehlerhaft)

```
public class String01 {  
    public static void main(String[] args) {  
        String vorname;  
        System.out.println(vorname);  
    }  
}
```

### Übung zum Programm *String01*

Das Programm erzeugt einen Umwandlungsfehler. Überlegen Sie, was der Grund dafür ist und korrigieren Sie das Programm, so dass es umgewandelt und ausgeführt werden kann.

#### Lösungshinweis

Die Fehlerursache ist, dass die Referenzvariable *vorname* im Ausgabebefehl benutzt wird, sie aber noch keinen Wert enthält. Für die Lösung der Übung gibt es zwei Möglichkeiten: entweder erzeugen Sie ein Objekt der Klasse *String* oder Sie sagen dem Compiler, dass (noch) kein Objekt benötigt wird und deswegen die Referenz leer bleiben soll.

Für die explizite Zuweisung einer leeren Referenz gibt es das Schlüsselwort *null*. Sinngemäß ist das die englische Bezeichnung für "Zeiger ins Nichts" - nicht zu verwechseln mit der deutschen Bezeichnung Null für die Zahl 0.

### Programm *String02*: Leere Referenz erzeugen (null-Referenz)

```
public class String02 {  
    public static void main(String[] args) {  
        String vorname = null;  
        System.out.println(vorname);  
    }  
}
```

### Übungen zum Programm *String02*

Ändern Sie die Wertzuweisung in der 3. Zeile wie folgt:

```
String vorname = "0";
```

Oder, wenn Sie wollen, auch so:

```
String vorname = "Null";
```

Wandeln Sie danach das Programm um und testen Sie es. Überlegen Sie, welche Arbeitsspeicherplätze im Originalprogramm bzw. nach dieser Änderung belegt sind.

Das folgende Programm macht den Unterschied zwischen *null* und einem leeren Objekt noch einmal deutlich.

#### Programm String03: *null* ist nicht *blank*

```
public class String03 {
    public static void main(String[] args) {
        String vorname = " ";
        System.out.println(vorname);
    }
}
```

Im Arbeitsspeicher werden durch das Programm folgende Plätze benötigt und angelegt:



Abb.14.1: Objekt im Speicher: Referenzvariable und Objektwert

Die Referenzvariable *vorname* enthält einen Verweis auf das eigentliche Stringobjekt. Dieses Stringobjekt enthält den Wert " " (blank bzw. space). Das ist laut Unicodetabelle der hexadezimale Wert 0020 (binär: 00000000 00100000). Wenn dagegen die Referenzvariable den Wert *null* enthält, so gibt es kein Stringobjekt, auf das sie weist.

### 14.1.2 Stringlitterale benutzen

Ein Stringliteral ist ein Text, der eingeschlossen ist in **doppelten** Hochkommas (nicht in einfachen Hochkommas wie *char*-Werte). So wie *char*-Litterale kann auch ein Stringliteral beliebige Escape-Sequenzen enthalten.

#### Programm String04: Stringlitterale und Escapesequenzen (mit Syntaxfehler)

```
public class String04 {
    public static void main(String[] args) {
        String str = 'Text \n mit Zeilenwechsel';
        System.out.println(str);
    }
}
```

#### Übung zum Programm String04

Das Programm enthält einen Syntaxfehler (Stringlitterale müssen in Anführungsstrichen stehen!). Bitte korrigieren Sie dies und führen Sie das Programm aus.

## 14.2 Methoden der Class *String*

### 14.2.1 Konstruktoren

Beim Erzeugen von neuen Objekten wird immer ein so genannter Konstruktor aufgerufen. Dies ist eine besondere Form einer Methode der jeweiligen Klasse, und er sorgt dafür, dass die Variablen, die zu dem Objekt gehören, initialisiert werden.

Für das Erstellen von neuen Stringobjekten gibt es in der Klasse *String* viele unterschiedliche Konstruktoren, z. B.

```
// Leeres String-Objekt erzeugen (das Argument ist blank)
String str1 = new String();
```

oder

```
// Kopie des Argument-Strings als Objekt erzeugen
String str2 = new String("Meyer");
```

Die Klasse *String* enthält mehr als 50 Methoden zum Arbeiten mit Zeichenketten, z.B. für

- |   |                    |
|---|--------------------|
| - das Verketteten von Strings             | <i>concat()</i>    |
| - das Extrahieren aus Teilstrings         | <i>substring()</i> |
| - das Positionieren innerhalb des Strings | <i>charAt()</i>    |
| - das Ersetzen von Zeichen                | <i>replace()</i>   |

Wir werden jetzt einige der Verarbeitungsmöglichkeiten für Strings besprechen.

### 14.2.2 Verkettung von Strings

Für das Aneinanderfügen von Zeichenketten ("Concatenation") gibt es zwei Möglichkeiten: den Verkettungsoperator `+` und die Methode *concat()*.

#### Programm *String05*: Zeichenketten aneinanderhängen

```
public class String05 {
    public static void main(String[] args) {
        String vorname = "Roman";
        String nachname = "Merker";
        String name1, name2;
        name1 = vorname + " " + nachname; // 1.Möglichkeit
        name2 = vorname.concat(" " + nachname); // 2.Möglichkeit
        System.out.println(name1);
        System.out.println(name2);
    }
}
```

### 14.2.3 Konvertierung zwischen primitiven Typen und Strings

Der Verkettungsoperator kann auch für Werte benutzt werden, die keine Strings sind. Dann konvertiert Java den Wert in einen String und konkateniert danach.

#### Programm *String06*: Automatische Umwandlung (implizit)

```
public class String06    {
    public static void main(String[] args)    {
        System.out.println("Bahnhofstr. " + 48);
    }
}
```

Regel: Wenn in einem Ausdruck der `+`-Operator benutzt wird, so kann dies unterschiedliche Aktionen zur Folge haben. Wenn alle Operanden numerische Typen sind, wird gerechnet, wenn auch nur ein Operand im Ausdruck ein String ist, so wird konkateniert.

Für die explizite Umwandlung von einfachen Datentypen in Strings bzw. umgekehrt die Konvertierung eines Strings in einen primitiven Datentyp enthält die Standard-Bibliothek vielfältige Möglichkeiten. Dieses Thema wird ausführlich im Kapitel 15 behandelt.

#### Programm *String07*: Explizites Konvertieren von Zahlen in Strings

```
public class String07    {
    public static void main(String[] args)    {
        float f1 = 3.57f;
        int z1 = 123;
        String s1, s2;
        // s = z1; // liefert Umwandlungsfehler
        s1 = String.valueOf(f1);
        s2 = String.valueOf(z1);
        System.out.println(s1 + s2);
        // System.out.println(f1 + z1);
    }
}
```

Die generellen Regeln bei Typumwandlungen mit String sind:

- String in primitiven Typ umwandeln: Der Zieltyp hat dafür eine Methode.
- Primitiven Typ in String umwandeln: Die Klasse *String* hat dafür Methoden.

#### Übungen zum Programm *String07*

**Übung 1:** Klären Sie die Frage, warum die 6. Zeile (`s=z1`) zu einem Umwandlungsfehler führt und deswegen "auskommentiert" werden musste. Notieren Sie auf einem Blatt Papier die interne Darstellung der Zeichen 123, wenn sie als Datentyp *int* co-



diert werden, und die interne Darstellung der Zeichen 123, wenn sie als *String* behandelt werden. Lösungshinweise siehe Kapitel 3.

**Übung 2:** Modifizieren Sie das Programm *String07.java* so, dass der Kommentar in der drittletzten Zeile entfernt wird. Überprüfen Sie die Ausgabe dieser Zeile und vergleichen Sie die Ergebnisse.

### Programm *String08*: Konvertieren von Strings in Arrays von Bytes

```
public class String08 {
    public static void main(String[] args) {
        String vorname = "Roman";
        byte[] zeichen = vorname.getBytes();
    }
}
```

Das obige Beispiel "encodiert" den Unicode-String in eine Sequenz von ASCII-Bytes, benutzt dabei den Default-CharacterSet der Plattform und speichert das Ergebnis in einem Array von Bytes.

### Übung zum Programm *String08*

Bitte ergänzen Sie das Programm um eine *for*-Schleife, in der die einzelnen Bytes des Arrays ausgegeben werden.

## 14.2.4 Vergleichen von Strings

Das Vergleichen von Strings auf inhaltliche Gleichheit ist ein heikles Thema. Grundsätzlich ist der Vergleichsoperator `==` **nicht** geeignet, um Strings zu vergleichen.

### Programm *String09*: Strings werden mit der Methode *equals* verglichen

```
public class String09 {
    public static void main(String[] args) {
        String s1 = "Merker";
        String s2 = new String("Merker");
        String s3 = "Merker";
        if (s1 == s2)
            System.out.println("==: Beide Strings sind gleich");
        if (s1.equals(s2))
            System.out.println("equals: Beide Strings sind gleich");
    }
}
```

### Übung zum Programm *String09*

Bitte experimentieren Sie mit dem Programm, um den Unterschied zwischen `==` und der Methode *equals* herauszufinden.

Nachdem Sie die verschiedenen Vergleiche mit den drei Strings durchgeführt haben, prüfen Sie besonders auch das Vergleichsergebnis zwischen s1 und s3. Auf den ersten Blick scheint es fast so, als seien die Vergleichsergebnisse vom Zufall abhängig. Wie kann dieses Phänomen erklärt werden? Denn natürlich gibt es auch hierfür logische Erklärungen.

### Erklärung

Der Compiler optimiert die Speicherung von Stringobjekten. Wenn Strings gleiche Inhalte haben, stehen sie unter Umständen nur einmal im Speicher (hier haben allerdings die unterschiedlichen JVMs auch individuelle Lösungen).

Dieses Beispiel hat hoffentlich eindrucksvoll demonstriert, dass beim Vergleich mit dem Vergleichsoperator `==` Vorsicht angebracht ist. Er prüft nämlich die **Referenzvariablen** auf inhaltliche Gleichheit. Nur beim Einsatz der Methode `equals` werden auch wirklich die Inhalte der **Stringobjekte** selbst verglichen.

**Fazit:** Verwenden Sie nie den Operator `==`, um Strings inhaltlich zu vergleichen. Arbeiten Sie immer mit der Methode `equals`. Diese Aussage gilt nicht nur für String, sondern grundsätzlich für den Vergleich von allen Objekten.

## 14.2.5 Einzelne Zeichen oder Teilstrings verarbeiten

Ein String kann für die Verarbeitung wie ein Array aus einzelnen Character behandelt werden. Das kann in vielen Fällen sinnvoll sein, z.B. wenn jedes einzelne Zeichen in der Zeichenkette abgefragt und eventuell verarbeitet werden soll.

### Programm *String10*: Den Buchstaben 'r' suchen

```
public class String10 {
    public static void main(String[] args) {
        String s1 = "Erwin Merker";
        for (int i=0; i<s1.length(); i++)
            if (s1.charAt(i) == 'r')
                System.out.println("Buchstabe r " +
                                   "steht auf Stelle " + i);
    }
}
```

### Übung:

Bitte schreiben Sie ein neues Programm, das im String "Vogelnest" das Wort "nest" extrahiert, in einer neuen String-Variablen speichert und am Bildschirm ausgibt.

**Lösungshinweis:** Es gibt die Methode `substring`; bitte klären Sie die Einsatzmöglichkeit dieser Methode anhand der API-Dokumentation.

### Lösungsvorschlag

```
public class String11 {
    public static void main(String[] args) {
        String s1 = "Vogelnest";
        String s2 = s1.substring(5, 9);
        System.out.println(s2);
    }
}
```

Der Methode *substring* können zwei Parameter übergeben werden, die Indices *von* - *bis*. Jedoch ist die Semantik dabei etwas eigenwillig, denn der Index *bis* bedeutet nicht einschließlich, sondern diese Position bezeichnet das erste Zeichen, das *nicht* kopiert werden soll. Vorteil dieser Zählweise: Die Länge des Substrings kann durch *bis* - *von* einfach errechnet werden.

## 14.2.6 Die Methoden der Stringklasse sind auch auf Stringlitterale anwendbar

### Programm *String12*: Länge feststellen und Zeichen umwandeln

```
public class String12 {
    public static void main(String[] args) {
        String name = "Merker";
        System.out.println("Steinfurt".length());
        System.out.println(name.toUpperCase());
    }
}
```

Ähnlich wie bei Arrays kann man auch für Strings die Länge feststellen. Bei Arrays kann man den Wert der Variablen *length* abfragen. Bei Strings sendet man die Nachricht *length()* an das Objekt. Der Unterschied in der Schreibweise (worin besteht er?) ist eine beliebte Fehlerquelle.

## 14.2.7 Was passiert, wenn ein Stringobjekt geändert wird?

Jetzt wollen wir einen bestehenden String inhaltlich verändern, um ihn an den chinesischen Markt anzupassen :-)

### Programm *String13*: Ersetzen der Buchstaben 'r' durch 'l'

```
public class String13 {
    public static void main(String[] args) {
        String s1 = "Merker";
        s1 = s1.replace('r', 'l');
        System.out.println(s1);
    }
}
```

Was bei dieser Änderung im Speicher genau passiert, werden wir nachfolgend näher analysieren.

### 14.2.8 Objekte der String-Klasse sind unveränderlich ("immutable")

Zum Einstieg in dieses Thema zunächst folgende Behauptung: Ein Objekt der Klasse *String* ist unveränderlich ("immutable"), d.h. man kann nur lesend darauf zugreifen. Wenn ein String-Objekt einmal mit einem Wert gefüllt ist, dann kann dieser Wert niemals geändert werden. Was bedeutet das - und was passiert z.B. bei Ausführung der Methode *replace* im Programm *String13*? Antwort: Es wird ein komplett neuer String erstellt und dann die Referenz in *s1* ersetzt durch die Speicheradresse des neuen Objekts. Das ist der Grund, warum alle Methoden, die ein String-Objekt modifizieren, als Rückgabebetyp ein Stringobjekt haben. Aber was passiert bei einer einfachen Wertezuweisung?

#### Programm *String14*: Auch die Referenzvariable ändert sich

```
public class String14 {
    public static void main(String[] args) {
        String s1 = new String("Heidi");
        String s2 = s1;
        System.out.println(s1 == s2);
        s2 = "Heidi";
        System.out.print(s1 == s2);
    }
}
```

Das Programm liefert zuerst *true*, dann aber *false* (zumindest bei der Sun-JRE unter Windows. Damit ist klar, dass die beiden Stringobjekte nach der Wertezuweisung zwar **inhaltlich gleich, aber nicht identisch** sind. Es wurde ein neues Objekt erzeugt.

Was ist der Grund für die auf den ersten Blick umständliche Arbeitsweise? Antwort: Strings werden in einem Pool gehalten. So kann man erreichen, dass Strings, die denselben Inhalt haben, nur einmal im Speicher stehen. Wenn also folgende Wertezuweisungen erfolgen:

```
s1 = "Heidi";
s2 = "Heidi";
```

dann können die Referenzen in *s1* und *s2* gleich sein. Verlassen kann man sich darauf aber nicht, denn dies kann von jeder Run-Time-Umgebung anders realisiert werden.

#### Übung zum Programm *String14*

Klären Sie, wie auf Ihrem System der Vergleich (*s1 == s2*) ausfällt, wenn Sie vorher auch der Variablen *s2* den Wert "Heidi" explizit zuweisen.

## 14.3 Methoden der Class *StringBuilder*

### 14.3.2 Warum gibt es drei String-Klassen?

Das oben beschriebene Verfahren ist relativ aufwändig. Die Entwickler sind offensichtlich davon ausgegangen, dass die Vorteile der gemeinsamen Nutzung überwiegen. Für häufig zu modifizierende Texte gibt es aber Stringklassen, die veränderbare Objekte erstellen: die class *StringBuffer* und die class *StringBuilder*.

Objekte dieser Klassen können während der Laufzeit des Programms verkürzt oder verlängert werden. Dadurch unterscheiden sich diese beiden Klassen von *String*. Für Einsteiger in die Programmierung mit Java gilt ganz einfach folgende Regel:

Wenn Stringvariable häufig geändert werden müssen, sollte als Datentyp *StringBuilder* und nicht *String* gewählt werden. Diese enthält (genau wie *StringBuf*) zahlreiche Methoden zum Verändern von Stringinhalten, z.B. *append()*, *insert()*, *delete()* oder *replace()*.

#### Programm String15: Arbeiten mit Class *StringBuilder*

```
public class String15    {
    public static void main(String[] args)    {
        StringBuilder sb1 = new StringBuilder("Vogel");
        sb1.append("nest");                    // Anhaengen
        System.out.println(sb1);
        sb1.replace(0,5,"Oster");              // Ersetzen
        System.out.println(sb1);
        sb1.delete(6,9);                       // Loeschen
        System.out.println(sb1);
        sb1.insert(6, " ist Urlaub");           // Einfuegen
        System.out.println(sb1);
    }
}
```

#### Programm String16: Wenn dieses Beispiel mit der Klasse *String* und der Methode *concat* codiert würde, würden 5 Objekte angelegt

```
public class String16    {
    public static void main(String[] args)    {
        StringBuilder str = new StringBuilder().
            append("Dieser ").append("Satz ").append("wird ").
            append("als ein ").append("Objekt angelegt");
        System.out.println(str);
    }
}
```

## 14.4 Strings als Commandline-Parameter

Strings sind beim **Datenaustausch** zwischen Anwendungen mit unterschiedlichen Programmiersprachen oder auf verschiedenen Betriebssystemen der wichtigste Datentyp. Bei Anwendungen im Internet erfolgt die Parameterübergabe zwischen Client- und Serverprogrammen generell als String, eventuell ergänzt um den Parameternamen ("Key-Value-Paare") oder eingepackt in Datenbeschreibungen nach dem XML-Standard.

Für den Datenaustausch zwischen Betriebssystem und einem Javaprogramm gibt es eine einfache Möglichkeit: beim Start eines Java-Programms können String-Werte an das Programm übergeben werden. Der empfangende Speicherbereich wird im Kopf der *main*-Methode definiert - und zwar als ein Array vom Datentyp *String*.

### Programm *String20*: Echo der Kommandozeilen-Parameter

```
public class String20 {  
    public static void main(String[] args) {  
        for (int i=0; i < args.length; i++)  
            System.out.println(args[i]);  
    }  
}
```

### Übung zum Programm *String20*

Testen Sie das Programm, indem Sie beim Starten die numerischen Werte 1, 5 und 7 als Parameter mitgeben (siehe hierzu Hinweise im Kapitel 10.2.2). Der Programmaufruf sieht in einer Commandline des Betriebssystems also wie folgt aus:

```
java String20 1 5 7
```

Es gibt eine Reihe von Standardaufgaben, die typisch sind, wenn String-Parameter von anderen Anwendungen eingelesen werden, nämlich prüfen, **ob** und **wieviele** Parameter übergeben worden sind, oder prüfen, von **welchem Datentyp** sie sind und welchen Inhalt sie haben.

### Programm *String21*: Prüfen, ob Eingabeparameter übergeben worden sind

```
public class String21 {  
    public static void main(String[] args) {  
        if (args.length == 0)  
            System.out.println("Eingabedaten fehlen");  
    }  
}
```

### Übung

Bitte codieren Sie ein neues Programm, das folgende Formalprüfung durchführt und gegebenenfalls eine entsprechende Fehlermeldung ausgibt: Es müssen zwischen 1

und max. 3 Parameter übergeben werden. Kein Parameter darf mehr als 5 Zeichen haben.

### Lösungsvorschlag: Wieviel und welche Parameter wurden übergeben?

```
public class String22 {
    public static void main(String[] args) {
        if (args.length == 0 || args.length > 3)
            System.out.println("Bitte 1 - 3 Parameter eingeben");
        for (int i=0; i < args.length; i++) {
            if (args[i].length() > 5) {
                System.out.print("Mehr als 5 Zeichen nicht erlaubt ");
                System.out.println(args[i]);
            }
        }
    }
}
```

Das folgende Programm *String23* überprüft, ob der Aufrufer dieses Programms einen Integer-Wert als Parameter mitgeliefert oder ob er eine Help-Funktion angefordert hat (durch Eingabe eines Fragezeichens ?). Andernfalls werden Fehlermeldungen ausgegeben.

### Programm *String23*: CMD-Line-Parameter abfragen und ggf. konvertieren

```
public class String23 {
    public static void main (String[] args) {
        int anzahl = 0;
        if (args.length == 1) {
            if (args[0].equals("Help")) {
                System.out.println("Aufruf: java String23 Anzahl");
                System.exit(1);
            }
        }
        try {
            anzahl = Integer.parseInt(args[0]);
        }
        catch (Exception e) {
            System.out.println("Es muss die Anzahl angegeben sein");
            System.exit(0);
        }
        System.out.println("Die Anzahl ist: " + anzahl);
    }
}
```

## 14.5 Zerlegen von Text

### 14.5.1 Methode *split()*

Die Methode *split* der Klasse *String* bietet eine Möglichkeit, mit Hilfe von "Regulären Ausdrücken" einen Satz zu zerlegen in einzelne Wörter. Das Thema "Reguläre Ausdrücke" wird im Abschnitt 14.6 ausführlich erläutert, hier zunächst ein einfaches Beispiel für das Arbeiten mit *split*.

#### Programm *Zerlegen01*: Aufsplitten eines kompletten Satzes

```
public class Zerlegen01 {
    public static void main(String[] args) {
        String s1 = "Dies ist ein Satz, der zerlegt wird";
        String[] ergebnis = s1.split(" ");
        for (int i=0; i<ergebnis.length; i++)
            System.out.println(ergebnis[i]);
    }
}
```

Das Standard-Trennzeichen ("delimiter") für das Splitten ist eine Leerstelle ("blank").

#### Übung zum Programm *Zerlegen01*

Ändern Sie das Literal, das in den String *s1* übertragen wird, so ab, dass mehrere Leerstellen zwischen den Wörtern *ist* und *ein* stehen. Prüfen Sie danach die Programmausgabe.

Hinweis: Zum Beschreiben von variablen Trennzeichen benötigt man "Reguläre Ausdrücke". Die Lösung gibt es im Abschnitt 14.6.

### 14.5.2 Class *Scanner*

Die Klasse *Scanner* bietet komfortable Möglichkeiten zum Lesen und Parsen von textbasierten Dateien und Zeilen. Auch sie erlaubt den Einsatz von "Regulären Ausdrücken". Sie ist die bessere Alternative zur veralteten Klasse *StringTokenizer*.

Bei der Beschäftigung mit diesem Thema stößt man immer wieder auf einige englische Fachbegriffe, die auch in der deutschen Literatur verbreitet sind: Ein *Scanner* teilt einen Eingabestring auf in einzelne *Token*. Dabei wird ein bestimmtes Muster zum Auftrennen benutzt (*delimiter pattern*). Das Default-Pattern zum Splitten ist ein *whitespace*. Unter *Whitespace* versteht man Zeichen, die von einem Texteditor am Bildschirm oder beim Ausdrucken nicht angezeigt werden, z.B. Leerzeichen oder Zeilenvorschub. Das Default-Pattern zum Splitten kann beliebig geändert werden.

Zum Beispiel bietet die *Scanner*-Klasse breitgefächerte Möglichkeiten, Daten von der Standardeingabe-Einheit *System.in* zu lesen und dabei einzelne Token nach bestimmten Regeln zu erkennen.



### Programm *Scanner01*: Einlesen und Auftrennen eines Satzes als *String*

```
import java.util.Scanner;
public class Scanner01 {
    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        eingabe.useDelimiter(",");
        String s = eingabe.next();
        System.out.println("Erster Teil: " + s);
        s = eingabe.next();
        System.out.println("Zweiter Teil: " + s);
    }
}
```

### Übung zum Programm *Scanner01*

Geben Sie zum Testen bitte folgenden Satz ein: "Um Objekte zu erzeugen, benutzt man das Schlüsselwort *new*". Wichtig ist das Komma im Satz. Das Ergebnis wird sein, dass die erste Satzhälfte als Echo wieder ausgegeben wird. Zum Anzeigen der zweiten Satzhälfte muss ein weiteres Trennzeichen (, Komma) eingegeben werden.

Die Klasse *Scanner* arbeitet mit einem "vorausschauenden Lesen" des Textes. Dazu gibt es die Methode *hasNext*. Diese Methode ist besonders hilfreich beim Formulieren von Leseschleifen - sie prüft, ob weitere Daten vorliegen, und dann kann zum eigentlichen Lesen dieser Daten die Methode *next* benutzt werden.

### Programm *Scanner02*: Lesen in Schleife mit *hasNext* und *next*

```
import java.util.Scanner;
class Scanner02 {
    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        String zeilenende = System.getProperty("line.separator");
        eingabe.useDelimiter(zeilenende);    // Delimiter aendern
        while (eingabe.hasNext())
            System.out.println(eingabe.next());
    }
}
```

Das obige Programm prüft mit *hasNext*, ob es Token im Eingabestrom gibt. Wenn nicht, bleibt es solange stehen, bis der Stream Daten enthält. Das kann ein zusätzliches Wort oder ein Delimiter sein. Dann wird der boolesche Wert *true* geliefert, der Schleifenrumpf ausgeführt, und *hasNext* startet neu. Beendet wird das Programm durch Eingabe eines Dateiende-Zeichens. In MS-Windows ist dies CTRL-C.

Die resultierenden Token können beim Lesen durch spezielle Methoden umgewandelt werden in eingebaute Datentypen, indem die entsprechenden Lesemethoden

benutzt werden, z.B. *nextInt* zum Einlesen von Ganzzahlen oder *nextFloat* zum Einlesen von Gleitkommazahlen.

### Programm *Scanner03*: Einlesen von eingebauten Datentypen

```
import java.util.Scanner;
class Scanner03 {
    public static void main(String[] args) {
        Scanner eingabe = new Scanner(System.in);
        eingabe.useDelimiter(",");           // Delimiter aendern
        String str    = eingabe.next();       // Komplettes Wort lesen
        int zahl1     = eingabe.nextInt();    // Ganzzahl lesen
        double zahl2  = eingabe.nextDouble(); // E-Format z.B. 5e3 lesen
        System.out.printf("%s | %d | %f", str, zahl1, zahl2);
    }
}
```

#### Tipps zum Testen des Programms *Scanner03*

- Achten Sie bei der Eingabe von Wörtern oder Zeichen darauf, dass diese durch Komma getrennt werden. Leerstellen sind nicht erlaubt. Beispiel einer korrekten Eingabe: Wort,2000,5e3,
- Wenn das abschließende Komma vergessen wird, erwartet das Programm weitere Eingaben und bleibt stehen - solange, bis das Komma eingegeben wird.
- Bei einer fehlerhaften Eingabe bricht das Programm ab mit einer System-Fehlermeldung

Diese Art des Einlesens von mehreren Token ist relativ primitiv, unhandlich und fehleranfällig. Komfortabler ist das Arbeiten mit Regulären Ausdrücken.

## 14.6 Reguläre Ausdrücke

"Reguläre Ausdrücke" werden für das komfortable Durchsuchen von Texten eingesetzt. Es wird ein Suchbegriff formuliert; danach überprüft Java, ob dieses Muster im Text gefunden wird, und stellt den gefundenen String für die weitere Verarbeitung zur Verfügung. In Java werden reguläre Ausdrücke durch das Paket *java.util.regex* unterstützt. Dieses Paket enthält nur die Klassen *Pattern* und *Matcher*. Objekte dieser beiden Klassen arbeiten Hand in Hand. Die *Pattern*-Objekte halten die regulären Ausdrücke bereit, und *Matcher*-Objekte führen den Vergleich durch und verwalten die Ergebnis-Menge.

Um ein Muster zu beschreiben, werden besondere Zeichen verwendet, die als Platzhalter dienen oder auch Regeln für die Suche beschreiben. Die wesentlichen Sonderzeichen und Suchregeln werden hier kurz vorgestellt:

.	der Punkt ist Platzhalter für jedes Zeichen, außer dem Zeilenvorschub
*	Wiederholung vorausgehender Zeichen, auch keinmal ist möglich
+	mindestens einmal, aber auch beliebig oft
\$	steht für das Zeilenende, z.B. "\n", "\r\n", oder "\r"
\n	Zeilenvorschub (' \u000A)
\s	Whitespace (nicht darstellbare Zeichen, wie Tabulator oder Zeilenvorschub)
[]	Bereich, z.B. '[0-9]' für den Zahlenbereich oder [abc] für a, b oder c
\	Backslash
^	Negation, z.B. '^abc' für jedes Zeichen außer a, b oder c
	Oder-Operator, z.B. '(A B)' bedeutet A oder B

Ein Muster wie „Se.“ wird sowohl „See“ als auch „Sea“ finden. Der reguläre Ausdruck „Schil[f]\*ahrt“ findet Schifahrt, Schiffahrt usw., aber eben auch Schiahrt. Mit dem Ausdruck Schil[f]+ahrt wird mindestens ein „f“ erzwungen. Eine Negierung eines Zeichens oder Bereiches erfolgt mit ^, also lässt [^0-9] keine Zahl zu.

Alle anderen Zeichen, die kein Sonderzeichen sind, müssen genau so wie angegeben im Text vorkommen, damit die Suche erfolgreich ist. Um ein Zeichen, das auch als Sonderzeichen definiert wurde, im Text zu finden, ist der Backslash „\“ als Fluchtzeichen (Escape-Sequenz), anzugeben, z.B. für das Dollarzeichen also „\\$“.

### 14.6.1 Die Klasse *Pattern*

Mit Objekten der Klasse *Pattern* werden die regulären Ausdrücke intern in kompilierter Form verwaltet. Die Instanz bekommt mit der Methode *compile* einen String mit dem regulären Ausdruck und lässt sich dann an verschiedenen Stellen im Programm performant verwenden. Ein direktes Verwenden eines regulären Ausdrucks wird auch unterstützt, nur bei mehrfacher Verwendung eines Ausdrucks sollte ein entsprechendes Objekt genutzt werden, damit die Analyse des Ausdrucks nicht immer wieder durchgeführt werden muss. Der Vergleich wird mit der Methode *matches* durchgeführt.

#### Programm RegEx01: Substrings mit der Methode *matches* suchen

```
import java.util.regex.*;
public class RegEx01 {
    public static void main(String[] args) {
        boolean b = Pattern.matches(".*pfel.*", "Birne Apfel Banane");
        if (b)
            System.out.println("Gefunden");
        else
            System.out.println("Nicht gefunden");
    }
}
```

Eine häufige Aufgabenstellung in der EDV ist das Zerlegen eines Strings in seine Bestandteile anhand von Trennzeichen. Häufig tauschen Programme Daten über Import-/Export-Textdateien aus. In diesen steht die Information dann zeilenweise einfach durch Semikolon oder Komma getrennt ("CSV-Files"). Um diese Daten nach dem Einlesen wieder zu zerlegen, bietet die Pattern-Klasse die sehr nützliche Methode *split*. Diese Methode zerlegt einen String anhand eines regulären Ausdrucks und stellt die Ergebnisse in ein Array.

### Programm RegEx03: Zerlegen von Strings (Methode *split* der Klasse *Pattern*)

```
import java.util.regex.*;
public class RegEx03 {
    public static void main(String[] args) {
        Pattern p = Pattern.compile(",;");
        String[] erg = p.split("Andre;Maier;Hauptweg.12;55131 Mainz");
        for (int i=0; i<erg.length; i++)
            System.out.println(erg[i]);
    }
}
```

### Übung zum Programm RegEx03

Ändern Sie das Programm so ab, das als Trennzeichen sowohl Semikolon, Komma und Whitespace verwendet werden. Der zu untersuchende String ist „eins zwei; drei,vier“.

#### Lösungsvorschlag:

```
import java.util.regex.*;
public class RegEx04 {
    public static void main(String[] args) {
        Pattern p = Pattern.compile(";|,|\\s");
        String[] erg = p.split("eins zwei; drei,vier");
        for (int i=0; i<erg.length; i++)
            System.out.println(erg[i] + '\n');
    }
}
```

### Übung zum Programm Zerlegen01 (aus Abschnitt 14.5.1):

Das Programm arbeitet mit der *split*-Methode der Klasse *String*. Leider ist der Splitvorgang in der bisherigen Variante wenig komfortabel (z.B. werden mehrere Blanks zwischen den Wörtern nicht richtig verarbeitet). Aber jetzt kennen Sie "Reguläre Ausdrücke" - und die können auch in der *split*-Methode der Klasse *String* eingesetzt werden.

Ändern Sie deshalb das Programm so ab, dass im String auch **mehrere** Blanks zwischen den einzelnen Wörtern richtig erkannt werden.

### Lösungsvorschlag

```
public class Zerlegen02 {
    public static void main(String[] args) {
        String sl = "Dies ist ein Satz, der zerlegt werden soll";
        String[] ergebnis = sl.split(" ");
        for (int i=0; i<ergebnis.length; i++)
            System.out.println(ergebnis[i]);
    }
}
```

### 14.6.2 Die Klasse *Matcher*

Diese Klasse vergleicht einen String mit dem regulären Ausdruck und verwaltet die Ergebnisse des Vergleiches. Objekte der Klasse *Matcher* werden von der Factory der *Pattern*-Klasse erzeugt, da ein *Matcher* nur im Zusammenhang mit einem *Pattern*-Objekt Sinn macht. Die Methode *find* sucht nach dem ersten bzw. dem nächsten Auftreten des regulären Ausdrucks in einem String. Die *Matcher*-Klasse stellt zwei unterschiedliche Methoden zur Verfügung. Die Methode *matches* vergleicht den Eingabe-String mit dem regulären Ausdruck und gibt eine Ergebnismenge zurück.

### Programm RegEx05: Methode *find* der Klasse *Matcher*

```
import java.util.regex.*;
public class RegEx05 {
    public static void main(String[] args) {
        // Pattern anlegen
        Pattern p = Pattern.compile("rot");
        // Factory zum Anlegen des Matcher
        Matcher m = p.matcher("Suche: rotes Auto, rotes Rad");
        boolean b = m.find();
        int anz = 0;
        while(b) {
            anz++;
            b = m.find();
        }
        System.out.println("Die Anzahl ist: " + anz);
    }
}
```

### Übung zum Programm RegEx02 (!)

Sehen Sie sich bitte das Programm *RegEx02* noch einmal an und ändern Sie es so, dass ein Objekt der Klasse *Pattern* angelegt wird. Der Vergleich soll dann von einem Objekt der Klasse *Matcher* durchgeführt werden.

**Lösungsvorschlag (Methode *matches* der Klasse *Matcher*):**

```
import java.util.regex.*;
public class RegEx06 {
    public static void main(String[] args) {
        Pattern p = Pattern.compile(".*pfel.*");
        Matcher m = p.matcher("Birne Apfel Banane");
        Boolean b = m.matches();
        if (b)
            System.out.println("Gefunden");
        else
            System.out.println("Nicht gefunden");
    }
}
```

**14.7 Strings und Unicode**

Intern, das heißt, innerhalb der Java Virtuellen Maschine, werden Werte der *String*-Klasse und Zeichen vom Typ *char* durch Unicode-Verschlüsselungen repräsentiert. Jedes Zeichen belegt 16 bits. Für die Kompatibilität mit bestehenden Dateisystemen, Datenbanken und Peripheriegeräten erfolgt eine Umwandlung, meistens in die ASCII-Byte-Darstellung. Also: Beim Einlesen der Daten vom Bildschirm oder von einer Festplatten-Datei werden aus den ASCII-Zeichen entsprechende Unicode-Verschlüsselungen gemacht. Noch einmal der Hinweis: Dies gilt nicht für die anderen Java-internen Datentypen wie *int* oder *float*, sondern nur für *char* und *String*.

Für die Umwandlung wird das Standard-Encoding der Plattform benutzt. Natürlich kann auch mit einer anderen Codierung gearbeitet werden, z.B. mit UTF-8.

**UTF8**

Um die größtmögliche Kompatibilität über alle Hardware-Plattformen und Betriebssysteme zu gewährleisten, gibt es die UTF8-Codierung.

**Programm *UTF08*: Komprimiertes Sichern von Unicode-Daten und kompatibler Austausch von Unicode-Daten über Plattformgrenzen durch UTF-8**

```
public class Utf08 {
    public static void main(String[] args) throws Exception {
        String str1 = "A\u0001\u0093";

        // Konvertieren String in UTF
        byte[] b1 = str1.getBytes("UTF8");
        for (int i=0; i<b1.length; i++)
            System.out.println(b1[i]);

        // Rekonstruieren von String aus Byte-Array
```

```
String str2 = new String(b1, "UTF8");
System.out.println(str2);
}
}
```

### Zusammenfassung

Zeichenketten sind Java-Objekte. Für das Arbeiten mit Strings gibt es in der Standardbibliothek von Java drei wichtige Klassen:

- **Class `String`:** Unveränderliches ("immutable") Zeichenkettenobjekt mit viel Methodenkomfort
- **Class `StringBuffer`:** Veränderliches Stringobjekt, kann bei Bedarf während der Programmlaufzeit vergrößert oder verkleinert werden
- **Class `StringBuilder`:** Genau wie `StringBuffer`, aber nicht threadsafe. Sollte immer dann verwendet werden, wenn Stringobjekte sich häufig ändern und wenn man "threadsafty" nicht braucht.

Der Unterschied zwischen den beiden letztgenannten Klassen ist ein Thema für Fortgeschrittene: *StringBuffer* ist threadsafe; *StringBuilder* ist zwar API-kompatibel zu *StringBuffer*, bietet aber keine Synchronisation bei Mehrfachzugriff.

**Grundlegende Operationen** für das Arbeiten mit Strings sind:

- Verketteten (*concat*)
- Aufteilen (*split*, *substring*)
- Umwandeln von Groß-/Kleinbuchstaben (*toLowerCase*, *toUpperCase*)
- Formatieren (*format*, *trim*).

Die veränderlichen Klassen *StringBuffer* und *StringBuilder* bieten darüber hinaus eine Fülle von Methoden zum Manipulieren von String:

- löschen (*delete*)
- einfügen (*insert*)
- anhängen (*append*).

**Vorsicht ist geboten beim Vergleichen von Zeichenketten.** Ein inhaltlicher Vergleich sollte nur erfolgen mit Hilfe der Methode *equals* (und nicht mit dem Vergleichsoperator `==`, denn dadurch wird lediglich ein Identitätsvergleich durchgeführt).

Auch für das wichtige Thema "Konvertieren von einfachen Datentypen in Strings" enthält die Stringklasse die entsprechenden Methoden (*valueOf*).

---

# 15

## Typumwandlungen verstehen ("casting")

---

Java ist eine typ-sensitive Sprache. Das heißt, die Sprache unterscheidet nach Datentypen und

- stellt sicher, dass **nur die erlaubten Typen** in den Variablen gespeichert werden können;
- überprüft, dass **nur die Operationen** ausgeführt werden können, die für die jeweiligen Typen auch erlaubt sind.

Die Festlegung des Datentyps erfolgt bei der Definition des Speicherplatzes für eine Variable. Dadurch ist es in den meisten Fällen bereits dem Compiler möglich, entsprechende Prüfungen vorzunehmen. In Ausnahmefällen kann aber erst zur Laufzeit des Programms geklärt werden, ob der Operator oder die Methode für diese Variable gültig ist.

Wir werden in diesem Kapitel klären, ob der Datentyp einer Variablen wirklich einmalig festgelegt wird und danach unveränderlich bis zum Programmende gilt oder ob es Möglichkeiten gibt, Typanpassungen vorzunehmen.

Dabei werden wir sehen, dass in manchen Fällen der Datentyp automatisch geändert (angepasst) wird. Und es ist in anderen Fällen möglich, die Typumwandlung explizit (aber dann auf eigenes Risiko!) durchzuführen.

Im Einzelnen erfahren Sie in diesem Kapitel

- wie primitive Datentypen in andere primitive Typen konvertiert werden;
- wie Referenztypen in andere Referenztypen konvertiert werden;
- welche Regeln gelten für das "Verpacken" (boxing) von einfachen Typen in Referenztypen bzw. umgekehrt für das "Auspacken" (unboxing) der Referenztypen in einfache Typen;
- welche speziellen Methoden genutzt werden können, um zwischen einfachen Typen und Stringobjekten zu konvertieren.

Die Umwandlung von einem Datentyp in einen anderen kann erfolgen:

- bei der Wertezuweisung ("assignment"), abhängig vom Typ des Empfängers,
- innerhalb eines Ausdrucks, um alle Operanden "gleichnamig" zu machen oder
- beim Methodenaufruf, abhängig davon, welche Parametertypen der Empfänger akzeptiert.



## 15.1 Erweiternde Konvertierung bei einfachen Typen

Typanpassung ist die Konvertierung von einem Datentyp (oder Klasse) in einen anderen Typ (oder Klasse). Zwischen einfachen Typen ist eine Konvertierung nur möglich, wenn die Typen gleichartig sind. Gleichartig sind alle numerischen Typen - und das sind außer *boolean* alle anderen sieben eingebauten Typen. Nicht möglich ist also lediglich die Umwandlung eines *boolean*-Typs in einen numerischen Typ.

Überhaupt keine Probleme ergeben sich, wenn der ursprüngliche Datentyp so interpretiert wird, dass eine erweiternde Umwandlung ("widening conversion") erfolgt, wenn also der neue Typ mehr Speicherplatz bzw. einen größeren Wertebereich hat als der bisherige.

Man kann eine Hierarchie der primitiven Typen erstellen, abhängig vom Speicherplatz, den sie zur Verfügung haben:

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

Problemlos ist eine Typanpassung in Pfeilrichtung, also von links nach rechts. So ist z.B. ein *int*-Wert kompatibel mit einem *long*-Typ (aber nicht umgekehrt).

### 15.1.1 Typanpassung bei Wertezuweisung

#### Programm *Konversion01: short auf long erweitern (automatisch)*

```
public class Konversion01 {  
    public static void main(String[] args) {  
        short a = 123;  
        long b = a;  
    }  
}
```

Bei der Umwandlung von *short* in *long* gehen keinerlei Informationen verloren, denn der neue Typ hat einen größeren Wertebereich als der ursprüngliche. Diese Art der Datentypumwandlung erfolgt automatisch, ohne Warnung, denn sie ist gefahrlos. Auch die Konvertierung von *char*-Typen (2 Bytes lang) in *int*-Typen (4 Bytes) erfolgt problem- und lautlos.

Wenn in einer Wertezuweisung ("assignment") ein Literal benutzt wird, so wird zunächst der Datentyp des Literals vom Compiler festgelegt, abhängig von der Schreibweise des Literals.

Danach wird in einem zweiten Schritt dieser Datentyp der empfangenden Variablen angepasst, eventuell ist eine Konvertierung notwendig. Wenn in einer Wertezuweisung ein Ausdruck aus mehreren Literalen und Variablen benutzt wird, so müssen zunächst die Datentypen der Operanden vereinheitlicht werden. Daraus bestimmt sich der Typ des Ausdrucks.

**Programm *Konversion02*: Datentyp eines Literals automatisch bestimmen**

```
public class Konversion02 {
    public static void main(String[] args) {
        byte b = 66;
        System.out.println(b);
    }
}
```

Das Literal 66 wird interpretiert als Platznummer im Unicode und nicht als Zahl vom *int*-Datentyp. Der Beweis kann dadurch geführt werden, dass wir das Programm zwingen, den Inhalt der Variablen b als Zeichen (Typ *char*) auszugeben:

**Programm *Konversion03*: Die korrekte Interpretation eines Typs erzwingen**

```
public class Konversion03 {
    public static void main(String[] args) {
        byte b = 66;
        System.out.println((char)b);
    }
}
```

Dies war ein Beispiel für das explizite Casting. Im folgenden Programm werden gleich mehrere Typanpassungen erforderlich: zunächst werden die Datentypen der Literale ermittelt und dann das Ergebnis angepasst an den Empfänger der Zuweisung.

**Programm *Konversion05*: Implizites Casting bei Zuweisung**

```
public class Konversion05 {
    public static void main(String[] args) {
        char z1 = 75 + 2;
        long z2 = 0x4b + 1;
        System.out.printf("%c %d ", z1, z2);
    }
}
```

**15.1.2 Numeric Promotion bei Ausdrücken**

Wenn in einem Ausdruck mit unterschiedlichen Typen gerechnet werden soll, so werden diese auf den höchsten Datentyp konvertiert (mindestens jedoch auf *int*-Typ). Dieser Vorgang wird "numeric promotion" genannt. Das folgende Beispiel zeigt diese automatische Anpassung aller Datentypen in einem Ausdruck.

**Programm *Konversion06*: Anpassung der unterschiedlichen Operanden**

```
public class Konversion06 {
```

```
public static void main(String[] args) {
    short a    = 110;
    float b    = 456789.1f;
    float erg  = a * b;
    System.out.println(erg);
}
```

Im Programm *Konversion06.java* werden zunächst die unterschiedlichen Operanden "gleichnamig" gemacht, bevor gerechnet wird, d.h. der *short*-Typ wird zu einem *float*-Typ konvertiert. Danach erfolgt die Wertezuweisung, und auch dabei kann eine zusätzliche Typanpassung erforderlich werden.

### Übungen zum Programm *Konversion06*

**Übung 1:** Überprüfen Sie das Rechenergebnis. Ist es mathematisch exakt? Oder gibt es Ungenauigkeiten (die begründet sind durch den Wechsel des Zahlensystems von Dezimalwertigkeit auf Dualstellensystem)?

**Übung 2:** Ändern Sie den Datentyp der empfangenden Variablen von *float* auf *double*. Ist das ohne weiteres möglich oder ändert das etwas am Ergebnis?

### 15.1.3 Typanpassung bei Parameterübergabe

Das nächste Beispiel demonstriert, dass eine automatische Typanpassung auch bei der Übergabe von Argumenten erfolgt. Der aktuelle Parameter wird angepasst an den Typ des formalen Parameters, der vom Empfänger erwartet wird.

#### Programm *Konversion07*: Typanpassung bei einer Parameterübergabe

```
public class Konversion07 {
    public static void main (String[] args) {
        ausgeben(25);
    }
    static void ausgeben(double zahl) {
        System.out.println(zahl);
    }
}
```

#### Übung zum Programm *Konversion07*

Machen Sie folgendes Experiment: Ändern Sie den Typ des formalen Parameters von *double* auf *int*. Ändern Sie außerdem den Methodenaufruf so, dass als aktueller Parameter ein Gleitkommawert übergeben wird.

Ist eine fehlerfreie Umwandlung möglich? Wenn nein, finden Sie in dem nächsten Abschnitt die Lösung für diese Aufgabe.

## 15.2      **Einschränkende Konvertierung bei einfachen Typen**

Problematisch ist eine Umwandlung von einem breiteren Datentyp in einen schmalere Typ, denn dabei können Informationen verloren gehen. Die vorherige Übung enthielt ein Beispiel für diese Situation. Das gleiche Problem kann auch bei einer Wertezuweisung entstehen oder bei der "numeric promotion" von arithmetischen Ausdrücken. Zunächst ein Beispiel für eine Zuweisung einer "langen" Variablen in eine Variable mit einem schmaleren Wertebereich.

### **Programm *Konversion08*: Ein Versuch, von *long* auf *short* verkürzen**

```
public class Konversion08 {
    public static void main(String[] args) {
        long a = 1234567;
        short b = a;
    }
}
```

Diese Codierung führt zu einem Umwandlungsfehler mit der Meldung: "Possible loss of precision". Um dennoch die Konvertierung zu erzwingen, muss dies vom Programmierer ausdrücklich codiert werden - und zwar durch Hinschreiben des Zieldatentyps, in runden Klammern.

### **Programm *Konversion09*: Erzwingen der Typumwandlung durch Casting**

```
public class Konversion09 {
    public static void main(String[] args) {
        long a = 123;
        short b = (short)a;
        System.out.println(b);
    }
}
```

### **Übung mit Programm *Konversion09***

Bitte ändern Sie das Literal in Zeile 3 von 123 auf 1234567 und prüfen Sie das Ergebnis. Kann fehlerfrei umgewandelt werden? Ist das Ergebnis sinnvoll?

Die Konvertierung eines großen primitiven Datentyps in einen kleinen primitiven Datentyp nennt man "einschränkende" Konversion (einengende Typumwandlung, "narrowing conversion"). Weil damit eventuell der Verlust von Information verbunden ist, erfolgt die Konversion nicht automatisch, sondern nur auf ausdrücklichen Wunsch des Programmierers. Diesen Vorgang nennt man "**Casting**" (engl. für "in Form gießen" oder auch "eine Rolle besetzen"). Erforderlich ist die explizite Angabe eines Cast-Operators: das ist Zieldatentyp (in Klammern).

Durch Einsatz eines Cast-Operators kann auch das Programm *Konversion07* formalfehlerfrei erstellt werden.

### Lösung zur Übung *Konversion07*

```
public class Konversion10 {
    public static void main (String[] args) {
        ausgeben((int)25.3);
    }
    static void ausgeben(int zahl) {
        System.out.println(zahl);
    }
}
```

Durch die explizite Angabe des Zieltyps wird eine Umwandlung des Datentyps vom Programmierer erzwungen. Damit übernimmt er auch die Verantwortung dafür, ob das Ergebnis sinnvoll ist oder nicht.

### Programm *Konversion11*: Typumwandlung bei Gleitkomma-Zahlen

```
public class Konversion11 {
    public static void main(String[] args) {
        double a = 12345678E300;
        float b = a;           // Fehler !
        System.out.println(a);
    }
}
```

### Übung mit Programm *Konversion11*

Das Programm enthält einen formalen Fehler, der bei der Umwandlung festgestellt wird. Bitte korrigieren Sie den Quelltext.

### Programm *Konversion12*: Explizites Casting mit Datenverlust

```
public class Konversion12 {
    public static void main(String[] args) {
        double a;
        long b = 12345678912348999L;
        a = b;                               // korrekt
        b = (long)a;                         // Datenverlust
        System.out.println(b);
    }
}
```

Die Bezeichnung "Typumwandlung" könnte suggerieren, dass die Verkürzung des Datentyps dazu führt, dass der neue Wert der Variablen nach einem ausgefeilten Verfahren ermittelt wird. In Wirklichkeit ist es ganz einfach so, dass nicht umgewandelt, sondern lediglich abgeschnitten (bzw. aufgefüllt) wird.

Jetzt folgt noch ein Beispiel, das zeigt, dass auch eine erweiternde Konvertierung manchmal erzwungen werden muss, damit überhaupt ein korrektes Ergebnis entstehen kann.

#### Programm *Konversion13*: Erweiterndes Casting erzwingen

```
public class Konversion13 {
    public static void main(String[] args) {
        double erg;
        int x = 5;
        int y = 3;
        erg = x / y;                // Abschneiden
        System.out.println(erg);
        erg = (double)x / y;        // Korrekt
        System.out.println(erg);
    }
}
```

### 15.3 Verallgemeinernde Konvertierung bei Referenztypen

Generell ist eine Konvertierung bei Referenztypen nur möglich, wenn eine Verwandtschaft besteht zwischen den Klassen. Es muss sich also um eine Ober- und Unterklasse handeln, d.h. sie müssen in einer Vererbungsbeziehung stehen (durch das Schlüsselwort *extends*).

Keine Probleme gibt es, wenn die Referenz einer Unterklasse umgedeutet wird auf eine Referenz der Oberklasse. Jede Unterklasse enthält alle Elemente der Oberklasse, so dass diese Umdeutung zwar zu Informationsverlust führen kann, aber technisch und formal keine Probleme verursacht, denn die Subklasse kann und hat mindestens das alles, was die Oberklasse enthält.

#### Programm *Konversion20*: Implizite Typanpassung bei Verallgemeinerung

```
public class Konversion20 {
    public static void main(String[] args) {
        A referenzA = new A();
        B referenzB = new B();
        referenzA = referenzB;
    }
}
class A {}
class B extends A {}
```

Die Wertezuweisung von B nach A ist ohne Probleme möglich, dabei findet eine automatische Konvertierung des Typs von unten (B) nach oben (A) statt. Dabei bleiben intern die speziellen Informationen über die Subklasse erhalten.

## 15.4 Spezialisierende Konvertierung bei Referenztypen

Ganz anders ist die Situation bei einer Referenz auf die Oberklasse, die nun als Referenz interpretiert werden soll, die auf ein davon abgeleitetes Objekt zeigt. Die Unterklasse enthält mehr Informationen (Attribute und/oder Methoden) als die Superklasse. Deswegen wird dieser Versuch vom Compiler abgelehnt.

### Programm *Konversion21*: Referenztyp spezialisieren

```
public class Konversion21 {
    public static void main(String[] args) {
        A referenzA = new A();
        B referenzB = new B();
        referenzB = referenzA;    // Fehler, inkompatibel
    }
}
class A {}
class B extends A {}
```

Der Programmierer kann aber auch hier das Casting erzwingen - genau wie bei den primitiven Datentypen. Die Konvertierung wird auch bei Referenztypen erzwungen durch Angabe des Cast-Operators. Der Cast-Operator besteht aus dem Namen der Zielklasse. Wir machen einen neuen Versuch, indem wir ein explizites Casting erzwingen.

### Programm *Konversion22*: Cast-Operator bei Referenztypen

```
public class Konversion22 {
    public static void main(String[] args) {
        A referenzA = new A();
        B referenzB = new B();
        referenzB = (B)referenzA;
    }
}
class A {}
class B extends A {}
```

Jetzt ist eine fehlerfreie Umwandlung möglich. Aber: Beim Versuch, das Programm auszuführen, gibt es einen Abbruchfehler (Run-Time-Error: "CastException").

Der Grund für diesen Fehler: Wir haben versucht, aus einem Referenztyp der allgemeinen Klasse A einen spezialisierten Typ der Klasse B zu machen. Das geht nicht - oder besser gesagt, das geht nur, wenn es sich auch wirklich um diesen spezialisierten Typ handelt. Mehr Schein als Sein lässt die JVM nicht zu.

Wichtige Regel: Nur wenn der Programmierer sich ganz sicher ist, dass die Referenz in Wahrheit eine Referenz auf die Subklasse ist, darf das Casting erzwungen werden, andernfalls führt der explizite Cast zu einem Run-Time-Fehler.

### Wie kann sichergestellt werden, dass keine Laufzeitfehler entstehen?

Es gibt einen Operator, mit dessen Hilfe zur Laufzeit überprüft werden kann, auf welchen Datentyp eine Referenz verweist: *instanceof*. Dies ist der dringend empfohlene Weg, um sicher zu stellen, dass kein falsches Casting durchgeführt wird.

#### Programm *Konversion23*: Laufzeittypprüfung durch *instanceof*

```
public class Konversion23 {
    public static void main(String[] args) {
        A referenzA = new A();
        B referenzB = new B();
        if (referenzA instanceof B)
            referenzB = (B)referenzA;
        else
            System.out.println("Kein Casting möglich");
    }
}
class A {}
class B extends A {}
```

Das nächste Beispiel demonstriert beide Arten der Konvertierung von Referenzen.

#### Programm *Konversion24*: Verallgemeinerung und Spezialisierung durch Casting

```
import java.awt.Point;
public class Konversion24 {
    public static void main(String[] args) {
        Point p1, p2;
        Object object;

        p1 = new Point(100,200);
        p2 = new Point(300,400);
        object = new Object();
        object = p1;           // generalisierende Konvertierung
        p2 = (Point)object;    // spezialisierende Konvertierung
        System.out.println(p2);
    }
}
```



## 15.5 Typ-Umwandlung zwischen einfachen und Referenztypen

In Java ist es nicht erlaubt, einfache Typen in Referenztypen zu casten oder umgekehrt. Es ist also z.B. nicht möglich, mit Hilfe eines Cast-Operators aus einem einfachen *float*-Typ einen Klassentyp *Double* zu machen.

Doch es gibt andere Möglichkeiten, um

- aus einfachen Typen die entsprechenden Referenztypen zu erzeugen bzw.
- aus Referenztypen die entsprechenden primitiven Typen herauszuziehen.

Damit dies möglich ist, gibt es in der Standardbibliothek für jeden primitiven Datentyp eine entsprechende Klasse, die einen Wert dieses Typs speichern und verarbeiten kann. Diese Klassen werden Wrapper-Klassen genannt. Es gibt für jeden der acht primitiven Datentypen eine entsprechende Wrapper-Klasse, also für *int* die Klasse *Integer* und für *float* die Klasse *Float*. Diese "hüllen" die primitiven Typen ein in Objekte und bieten zusätzliche Methoden zum Arbeiten mit diesen Referenztypen.

### Wozu braucht man Wrapper-Klassen?

Notwendig ist der Einsatz dieser Wrapper-Klassen z.B. immer dann, wenn eine Methode ausschließlich Objekte (also Referenztypen) als Parameter akzeptiert und keine primitiven Datentypen zulässt. In den meisten Fällen erfolgt dann das "Einpacken" (wrapping, boxing) automatisch, d.h. aus dem einfachen Datentyp wird ohne besondere Vorkehrung durch den Programmierer ein Referenztyp der Wrapperklasse erzeugt. Dieser Vorgang wird Autoboxing genannt. Umgekehrt erfolgt auch das Auspacken (unboxing) in den meisten Fällen automatisch.



Abb. 15.1: Beispiel für das Wrappen einer primitiven int-Variablen

In manchen Situationen muss das Arbeiten mit den Wrapper-Klassen aber ausdrücklich codiert werden. Damit zunächst einmal praktisch geübt werden kann, wie man mit Wrapperklassen arbeitet, zeigt das Programm *Wrapper01*, wie eine ganze Zahl als Objekt erzeugt und verarbeitet wird.

### Programm *Wrapper01*: Ganzzahl als einfacher Datentyp und als Referenztyp

```
class Wrapper01 {  
    public static void main(String[] args) {  
        int zahl1 = 15;  
        // Integer zahl1 = new Integer(15);  
        System.out.println(zahl1);  
    }  
}
```

### Übung zum Programm *Wrapper01*

Ersetzen Sie die Definition des einfachen Typs in der Zeile 3 durch die gleichwertige Definition in Zeile 4 und testen Sie das Ergebnis. Ändert sich das Ergebnis?

Zusammenfassung: Eine Konvertierung zwischen Referenztyp und einfachem Typ ist in Java nicht möglich. Wohl möglich ist das Einpacken (boxing) von einfachen Typen in **den entsprechenden Typ** der dazugehörigen Wrapperklasse bzw. das Auspacken (unboxing) von Referenztypen, um wieder einfache Typen zu bekommen.

#### 15.5.1 Autoboxing/Unboxing

Das Ein- oder Auspacken erfolgt immer dann automatisch, wenn Java erkennt, dass dies notwendig und möglich ist. Beispiel: Wenn primitive Typen an die Methode *printf* übergeben werden, dann werden diese einfachen Typen in Objekte der jeweiligen Klasse umgewandelt (sie werden "eingepackt in eine Box", auf englisch: "autoboxing" oder "autowrapping"). Auch der umgekehrte Vorgang, das Entpacken (oder "unboxing") kann erfolgen, ohne dass der Programmierer dafür besondere Vorkehrungen treffen muss.

#### Programm *Boxing01*: Automatisches Boxing/Unboxing

```
class Boxing01 {
    public static void main(String[] args) {
        int zahl1 = 15;
        Integer zahl2 = new Integer(25);
        System.out.println(zahl1 + zahl2);    // Autoboxing
        zahl1 = zahl2;                        // Auto-Unboxing
        System.out.println(zahl1);
    }
}
```

#### 15.5.2 Konvertieren mit speziellen Methoden

Wie bereits mehrfach betont, bietet die Java-Sprache keine Möglichkeit, um mit einem Casting-Operator zwischen einfachen und Referenztypen zu konvertieren. Aber es gibt Methoden zum Umwandeln von Stringobjekten in primitive Typen und auch Methoden, um einfache Typen in Stringobjekte zu konvertieren.

#### Umwandeln von einfachen Typen in Stringobjekte

Die Wrapperklassen enthalten eine *toString*-Methode, die eine String-Repräsentation für einen einfachen Datentyp liefert. Wir wissen auch bereits, dass *toString* von bestimmten Methoden automatisch aufgerufen wird - so z.B. von der *println*-Methode.

### Programm *KonvertMethod01*: String erzeugen durch Methodenaufruf

```
class KonvertMethod01 {
    public static void main(String[] args) {
        float zahl1 = 15E3f;
        System.out.println(zahl1);
    }
}
```

### Umwandeln von *String* in einfache Typen

Für den umgekehrten Weg, nämlich das Zurückführen eines Stringobjekts in einen primitiven Datentyp, gibt es mehrere Möglichkeiten:

- entweder wird eine Methode der entsprechenden Wrapperklasse aufgerufen,
- oder es wird eine Methode einer Stringklasse (*String*, *StringBuf* oder *StringBuilder*)

aufgerufen.

### Programm *KonvertMethod02*: *int* erzeugen aus einem *String* (1.Möglichkeit)

```
class KonvertMethod02 {
    public static void main(String[] args) {
        String str = "153";
        Integer zahl = Integer.valueOf(str);
        System.out.println(zahl);
    }
}
```

### Programm *KonvertMethod03*: *int* erzeugen aus einem *String* (2.Möglichkeit)

```
class KonvertMethod03 {
    public static void main(String[] args) {
        String str = "153";
        int zahl = Integer.parseInt(str);
        System.out.println(zahl);
    }
}
```

Jede Wrapperklasse enthält die entsprechende *parseXXX*-Methode zum Parsen des Strings, um den jeweiligen primitiven Typ daraus zu erzeugen.

### Übung zum Programm *KonvertMethod03*

Ändern Sie das Programm so ab, dass der *String* in einen einfachen *float*-Typ umgewandelt wird.

### Lösungsvorschlag

```
class KonvertMethod04 {
    public static void main(String[] args) {
        String str = "153E5";
        float zahl = Float.parseFloat(str);
        System.out.println(zahl);
    }
}
```

### Zusammenfassung

Prinzipiell ist beim Arbeiten mit Variablen der vom Programmierer in der Deklaration festgelegte Datentyp streng einzuhalten. Dies wird vom Compiler überprüft und sichergestellt. Prinzipiell können Zuweisungen oder Parameterübergaben nur durchgeführt werden, wenn die Datentypen exakt übereinstimmen. Stimmen Datentyp von Sender und Empfänger nicht überein, so kann in bestimmten Fällen eine automatische Datentyp-Anpassung erfolgen. Das geschieht aber nur, wenn die Konversion ohne Gefahren von Informationsverlust oder -fälschung möglich ist. Und dafür gibt es strenge [Regeln](#):

- bei einfachen Typen: Umwandlung nur von klein nach groß,
- bei Referenztypen: Umwandlung nur von speziell auf allgemein.

Diese strenge Typisierung in Java kann allerdings mit einigen Sprachmitteln aufgeweicht werden. So gibt es den [Typ-Cast](#), damit kann der Programmierer das Casting durch explizite Angabe eine Uminterpretation eines Datenwertes erzwingen. In den meisten Fällen wird es sich dabei um einschränkende bzw. spezialisierende Konversionen handeln.

Weil es dabei zu Run-Time-Fehlern kommen kann, wird dringend empfohlen, damit vorsichtig umzugehen und vor dieser Art der Umwandlung **immer** eine Typprüfung vorzunehmen (mit *instanceof*). Die Verantwortung liegt allein beim Programmierer, der Compiler kann dann nicht mehr helfen. Deswegen muss streng unterschieden werden zwischen impliziter Typkonversion, die automatisch erfolgt, weil sie problemlos ist, und dem expliziten Casting, bei dem der Programmierer in Kauf nimmt, dass Daten verfälscht werden können.

Darüber hinaus gibt es spezielle Methoden, wenn [zwischen einfachen Typen und Stringobjekten](#) konvertiert werden soll.

Eine besondere Art der Umwandlung ist das [Auto-Boxing](#) bzw. [Auto-Unboxing](#). Darunter versteht man die automatische Umwandlung eines primitiven Typs in einen Referenztyp bzw. das Auspacken eines Objekts in einen Basistyp.

# 16

## Modifizier richtig einsetzen ("access control")

---

Variablen sollten einen möglichst eingegrenzten Benutzerkreis haben. Dies erhöht die Sicherheit und erleichtert die Wartung. Dieses Kapitel beschreibt die Regeln für die "Sichtbarkeit" von Variablen, d.h. Sie werden lernen, wo die Identifier (Bezeichner) der Variablen bekannt und zugreifbar sind, wie der Schutz der Daten vor unerwünschtem Zugriff funktioniert und wie der Programmierer dies beeinflussen kann.

Im Einzelnen werden in diesem Kapitel folgende Themen besprochen:

- Wer darf *innerhalb* einer Klasse auf welche Variablen zugreifen?
- Wer darf von *außerhalb* einer Klasse auf welche Klassenelemente zugreifen?
- Welche Rolle spielt dabei das *package* und die *import*-Anweisung?
- Wie wird von *innerhalb* und *außerhalb* einer Klasse auf Elemente zugegriffen?
- Welche Schlüsselwörter gibt es, um Datenkapselung zu erreichen?
- Wie lange existieren die Variablen im Arbeitsspeicher?
- Wann sollten Elemente einer Klasse mit dem *static*-Modifizier versehen werden?

Es gibt zwei unterschiedliche Sprachmittel, die mit diesen Themen verbunden sind:

- zum einen ist es wichtig, wo (an welcher Stelle innerhalb einer Klasse) die Variablen deklariert werden und
- zum anderen gibt es die Zugriffsmodifizier *private*, *public* und *protected*, die bei der Deklaration benutzt werden können.

Wir werden uns zuerst mit der Frage beschäftigen, welchen Einfluss die Position der Deklaration auf die Sichtbarkeit hat, und danach die unterschiedlichen Zugriffsmodifizier besprechen.

### 16.1 Lokale Variable und Member-Variable

Die Antwort auf alle aufgeworfenen Fragen hängt wesentlich davon ab, ob die Variablen lokale oder Member-Variablen sind.

Eine **lokale Variable** wird innerhalb einer Methode deklariert, entweder durch ein Deklarationsstatement oder als formaler Parameter im Kopf der Methode. Wir haben bereits erläutert, dass innerhalb einer Methode ein Anweisungsblock codiert werden

kann (das sind Befehle, die in geschweiften Klammern zusammengefasst werden). Auch innerhalb eines Blocks kann eine lokale Variable deklariert werden.

Eine **Member-Variable** ist ein Element einer Klasse, deklariert außerhalb jeder Methode. Typischerweise werden sie ganz am Klassenanfang beschrieben (dies ist aber nicht zwingend).

Das folgende Bild soll verdeutlichen, wo die Unterschiede zwischen diesen beiden Arten liegt:

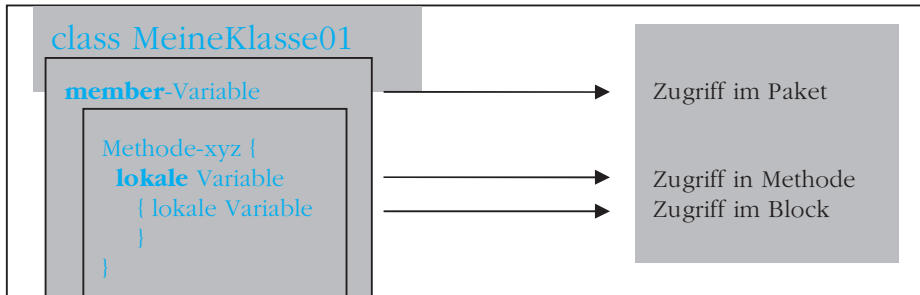


Abb. 16.1 Lokale und Member-Variable

### Wo liegt die Bedeutung dieser Unterscheidung?

Die Auswirkungen dieser unterschiedlichen Positionierung sind gravierend. Sie hat Folgen

- für die **Sichtbarkeit** ("visibility") bzw. **Gültigkeit** ("scope"): Lokale Variable sind nur bekannt und zugreifbar innerhalb der Methode (bzw. des Blocks); Member-Variablen dagegen sind in der ganzen Klasse bekannt und sogar darüber hinaus, nämlich in dem gesamten Paket (das Thema *package* wird in den nächsten Abschnitten detailliert erläutert);
- für die **Initialisierung** der Variablen: Member-Variablen werden von Java automatisch mit Anfangswerten versehen; lokale Variablen werden nicht automatisch initialisiert;
- für die **Lebensdauer** (duration/lifetime) der Variablen: lokale Variable existieren nur für die Dauer der Methodenausführung. Sie werden erzeugt beim Methodenaufruf, und der Speicherplatz ist wieder verfügbar, wenn die Methode beendet wird. Bei Member-Variablen kann die Frage der Lebensdauer nur entschieden werden, wenn man weiß, ob es sich um *static*-Member oder um Objekt-member handelt (auch dazu später mehr);
- für die Möglichkeit, so genannte **Access Modifier** zu benutzen, um die Sichtbarkeit einzugrenzen oder auszuweiten (dies ist nur für Member-Variablen möglich, lokale Variable bleiben immer nur lokal sichtbar).

## 16.2 Sichtbarkeit und Gültigkeit von Variablen

Der Scope (Gültigkeitsbereich) einer Variablen ist der Bereich, in dem die Variable referenziert werden kann mit ihrem simplen Namen. Außerdem bestimmt die Gültigkeit, wann das System die Variable erstellt und wieder zerstört. Bei Member-Variablen muss außerdem noch über die Visibility (Sichtbarkeit) entschieden werden. Dadurch wird festgelegt, wer auf diese Variable von außerhalb der Klasse zugreifen darf.

### Programm *Scope01*: Membervariable und lokale Variable

```
class Scope01 {
    int zahl1;                                // Membervariable
    void setZahl1() {
        zahl1 = 5;
        int zahl2 = 15;                      // lokale Variable
    }
    void ausgeben() {
        System.out.println(zahl1);
        // System.out.println(zahl2);        // Fehler, lokale V.
    }
}

public class ScopeTest01 {
    public static void main(String[] a) {
        Scope01 scope = new Scope01();
        scope.setZahl1();
        scope.ausgeben();
    }
}
```

Die Quelltext-Datei für dieses Programm muss *ScopeTest01.java* heißen!

Die Variable *zahl1* ist außerhalb jeder Methode deklariert. Sie ist damit ein Element ("member") der Klasse und in der gesamten Klasse gültig. Sie kann in jeder Methode dieser Klasse benutzt werden. In diesem Programm wird die Variable in der Methode *ausgeben* angesprochen. Innerhalb einer Klasse werden die Member mit ihrem simplen Namen referenziert (d.h. sie müssen nicht explizit qualifiziert werden mit dem Instanz- oder Klassennamen, der ist implizit vorhanden).

Dieselbe Methode versucht, die Variable *zahl2* zu benutzen. Diese Variable wurde innerhalb der Methode *setZahl1* deklariert. Damit ist sie eine lokale Variable. Und diese sind nur sichtbar innerhalb der Methode (oder des Blocks), in dem sie deklariert worden sind.

### Übung zum Programm *Scope01*

Entfernen Sie bitte die Kommentarzeichen. Wie lautet dann die Fehlermeldung bei der Umwandlung?

Eine Besonderheit im Leben einer Variablen ist eine Situation, wo sie temporär verdeckt wird von einer anderen, gleichnamigen Variablen. Trotzdem kann sie angesprochen und verarbeitet werden.

### Programm *Shadow01*: Überdecken einer Variablen

```
class Shadow01 {  
    static int z1 = 100;           // Member-V.  
    public static void main(String[] a) {  
        int z1 = 2;              // Lokale V.  
        System.out.println(z1);  
    }  
}
```

### Übung zum Programm *Shadow01*

**Übung 1:** Bitte überlegen Sie, wie in der *main*-Methode auf die Membervariable *z1* zugegriffen werden kann und testen Sie die Lösung. Lösungshinweis: Da es sich um eine *static*-Variable handelt, wird sie qualifiziert mit dem Klassennamen.

**Übung 2:** Bitte prüfen Sie durch Programmänderung, ob derselbe Name *innerhalb* einer Methode mehrfach definiert werden kann, wenn dies in unterschiedlichen Blöcken geschieht.

Fazit: Eine Variable ist nur innerhalb eines Scope benutzbar, am Ende des Scope ist sie nicht mehr verfügbar. Dabei können für Membervariablen und lokale Variablen dieselben Identifier benutzt werden.

## 16.3 Welchen Anfangswert haben die Variablen?

Grundsätzlich werden alle Member-Variablen von Java mit einem "passenden" Anfangswert vorbelegt. Dieser Initialwert ist abhängig vom Datentyp: z.B. werden numerische Variablen mit Nullen vorbelegt und boolesche Variablen mit *false*.

### Programm *Init01*: Defaultwert, wenn Objekte fehlen: null

```
public class Init01 {  
    public static void main(String[] args) {  
        KlasseA a = new KlasseA();  
        System.out.println(a.str);  
    }  
}  
  
class KlasseA {  
    String str;  
}
```



### 16.3.1 Ausnahme: lokale Variablen

Von dieser grundsätzlichen Regel zum Initialisieren gibt es jedoch eine Ausnahme: Lokale Variablen werden nicht automatisch initialisiert, das ist Aufgabe des Programmierers. Dies ist auch sinnvoll, um fehlerhafte Vorbelegungen zu vermeiden, denn lokale Variablen werden häufig temporär als Zwischenspeicher benötigt.

#### Programm *Init02*: Lokale Variablen müssen "per Hand" initialisiert werden

```
public class Init02 {
    public static void main(String[] args) {
        String str;
        System.out.println(str);
    }
}
```

#### Übung zum Programm *Init02*

Das Programm kann nicht fehlerfrei umgewandelt werden. Korrigieren Sie diesen Fehler.

## 16.4 Lebensdauer von Variablen

Wann legt das System Speicherplatz für die Variablen an und wann wird dieser wieder frei gegeben? Anders gefragt, wie ist die Lebensdauer ("lifetime", "duration") der Variablen?

### 16.4.1 Lebensdauer von lokalen Variablen

Am einfachsten ist diese Frage bei lokalen Variablen zu beantworten: Lokale Variablen werden im Arbeitsspeicher angelegt, wenn eine Deklarationsanweisung ausgeführt wird. Methodenparameter belegen Speicherplatz, sobald die Methode aufgerufen wird. Danach kann jede Anweisung innerhalb dieses Blocks bzw. der Methode darauf zugreifen. Die Existenz der Variablen endet, wenn der Block oder die Methode, in der sie deklariert worden ist, verlassen wird.

#### Programm *Lokal01*: Versuch, eine lokale Variable nach Ende der Lebensdauer auszugeben

```
public class Lokal01 {
    public static void main(String[] a) {
        for (int i = 0; i < 3; i++) {
            System.out.println(i);
        }
        System.out.println(i);
    }
}
```

### Übung zum Programm *Lokal01*

Das Programm kann nicht fehlerfrei umgewandelt werden. Bitte beheben Sie den Fehler.

#### Lösungsvorschlag

```
public class Lokal02 {  
    public static void main(String[] a) {  
        int i;  
        for (i = 0; i < 3; i++) {  
            System.out.println(i);  
        }  
        System.out.println(i);  
    }  
}
```

Die Lebensdauer von Methodenparametern ähnelt der von lokalen Parametern. Auch sie existieren nur für kurze Zeit. Sie werden im Kopf der Methode deklariert und sind nur innerhalb der Methode sichtbar. Nach Ausführung der Methode "stirbt" auch diese Variable, und sie wird bei einem erneuten Aufruf auch **neu** wieder im Arbeitsspeicher **angelegt**.

### Programm *MethodParm01*: Wie lange "lebt" ein Parameter?

```
public class MethodParm01 {  
    public static void main(String[] a) {  
        MethodParm01 m = new MethodParm01();  
        m.methodA(17);  
        // System.out.println(zahl);    // existiert nicht mehr!  
    }  
    void methodA(int zahl) {  
        System.out.println(zahl++);  
    }  
}
```

Im Programm *MethodParm01* wird die Variable *zahl* als formaler Parameter von der Methode *methodA* deklariert. Der Versuch, von außerhalb dieser Methode auf die Variable *zahl* zuzugreifen, führt zu einem Umwandlungsfehler. Deswegen ist die Zeile 5 als Kommentar geschrieben worden.

### Übung zum Programm *MethodParm01*

Bitte korrigieren Sie das Programm so, dass der gewünschte Effekt erzielt wird. (Lösungshinweis: Der Wert des Parameters muss als Ergebnis von der aufgerufenen Methode zurückgegeben werden und beim Aufruf empfangen werden.)

## 16.4.2 Lebensdauer von Membervariablen

Etwas differenzierter muss die Frage nach der Lebensdauer bei Membervariablen beantwortet werden. [Member-Variablen werden unterteilt in](#)

- [Instanz-Elemente \(Objekt-Variablen\)](#) und
- [Klassen-Elemente \(static-Variablen\)](#).

Normalerweise sind Member-Variablen auch Objekt-Variablen. Sie sind existenziell verknüpft mit einem Objekt, d.h. sie werden im Arbeitsspeicher angelegt, sobald eine Instanz erzeugt wird. Jede neue Instanz erstellt einen neuen Satz von Variablen dieser Klasse. Angesprochen (adressiert) werden sie mit Hilfe der Referenz-Variablen.

### 16.4.2.1 static-Variablen (Klassenelemente)

Es gibt auch Situationen, wo Informationen gespeichert werden müssen, die unabhängig sind von jeder Instanz. Das sind Variablen oder Konstanten, die für die gesamte Klasse gelten und deswegen auch nur einmal pro Klasse existieren. Solche Memberelemente werden mit dem Schlüsselwort *static* gekennzeichnet.

Mit dem Schlüsselwort *static* für Felder einer Klasse wird gleichzeitig festgelegt, dass diese Elemente sofort genutzt werden können, sobald die Klasse in den Speicher geladen wurde. In dem Fall ist es also nicht erforderlich, dass vorher Instanzen dieser Klasse erzeugt werden, um darauf zuzugreifen.

Noch ein Hinweis zum Begriff *static*: Das Schlüsselwort *static* wird manchmal gleichgesetzt mit konstant. Das passt in diesem Fall aber nicht, denn mit *static* ist das Gegenteil von dynamic gemeint, weil diese Variablen nicht zur Laufzeit erzeugt werden, sondern beim Laden der Klasse und dann (statisch) bis zum Ende der Programmlaufzeit zur Verfügung stehen.

### Programm StaticTest01: Static-Variable verarbeiten

```
class Static01 {
    static int zahl;
    static String text;
}
class StaticTest01 {
    public static void main(String[] a) {
        System.out.println(Static01.zahl + Static01.text);
    }
}
```

### Übung zum Programm StaticTest01

Bitte klären Sie durch Programmänderung die Frage, ob die *main*-Methode auch innerhalb der Klasse *Static01* stehen könnte. Wie ändert sich dann die Art der Adressierung von *zahl* und *text*?

### **static-Modifizier für Methoden**

Nicht nur Felder können das Schlüsselwort *static* bekommen, auch bei der Deklaration von Methoden kann *static* angegeben werden. Dadurch wird diese Methode zu einer Klassenmethode.

Ähnlich wie Klassenvariablen unabhängig von Instanzen angelegt werden und sichtbar sind, kann auch auf Methoden, die das Schlüsselwort *static* haben, zugegriffen werden, ohne dass eine Instanz vorliegt. Prominentes Beispiel einer solchen Methode: die Methode *main*. Sie wird vom Laufzeitsystem gestartet, ohne dass vorher eine Instanz dieser Klasse erzeugt wird.

#### **16.4.2.2 Instanzelemente (Objektvariablen)**

Am wichtigsten sind die Instanzelemente (z.B. Variablen), diese werden für jede Instanz angelegt, die erzeugt wird (normalerweise mit dem Schlüsselwort *new*). Die Verarbeitung kann aus allen "normalen" Methoden dieser Klasse heraus erfolgen durch Benutzung des einfachen Namens.

#### **Programm *InstanzTest01*: Instanzen erstellen und bearbeiten**

```
class Instanz01 {
    int zahl;
    String text;
    Instanz01() {};
    Instanz01(int z, String t) {
        zahl = z;
        text = t;
    }
}

class InstanzTest01 {
    public static void main(String[] a) {
        Instanz01 instanz1 = new Instanz01(17, "ABC");
        Instanz01 instanz2 = new Instanz01(25, "XYZ");
        System.out.println(instanz1.zahl + instanz1.text);
    }
}
```

Die Umwandlungseinheit (Quelltextdatei) muss *InstanzTest01.java* heißen. Ein Test des Programms demonstriert, dass ein Satz Membervariablen pro Objekt angelegt wird. Außerdem zeigt es, wie die einzelnen Variablen referenziert werden.

#### **Übung zum Programm *InstanzTest01***

Bitte ändern Sie das Programm so, dass die erste Instanz die Werte 4750 und "Beispieltext" enthält und für die zweite Instanz der Default-Konstruktor aufgerufen wird.

## 16.5 Zugriffsrechte von außerhalb einer Klasse ("access control")

Dieser Abschnitt befasst sich mit der Frage: Wer darf von außen auf Elemente einer Klasse zugreifen? Wer darf also die Felder benutzen, d.h. lesen oder verändern, und wer darf die Methoden aufrufen? Innerhalb einer Klasse kann jede Methode jedes Feld und jede andere Methode benutzen, aber wie ist es, wenn die aufrufende Methode zu einer anderen Klasse gehört?

### Was sind die "Elemente" einer Klasse?

Zu den Elementen einer Klasse gehören die Felder (d.h. die Variablen und die Konstanten) sowie die Methoden. Zur Erinnerung: Elemente einer Klasse können vererbt werden, deswegen gehören die Konstruktoren auch nicht zu den Elementen, denn sie werden nicht vererbt.

### Wer darf auf die Elemente einer Klasse zugreifen?

Die Standardregel für den Zugriff auf Elemente einer Klasse lautet:

Alle zu einem **Paket** gehörenden Klassen können auf alle Elemente der anderen Klassen in diesem Paket zugreifen.

### Wie wird auf die Elemente zugegriffen?

Innerhalb einer Klasse wird durch den einfachen Namen referenziert, außerhalb der Klasse muss das Element qualifiziert angesprochen werden.

### Was bedeutet Qualifikation?

Beim Zugriff auf Methoden oder Variablen reicht oft die Angabe des Bezeichners nicht aus, weil er nicht eindeutig ist. Der Grund dafür kann sein, dass mehrere Exemplare desselben Bezeichners existieren (z.B. beim Erzeugen von mehreren Instanzen sind die Instanzvariablen auch mehrfach im Arbeitsspeicher), es kann aber auch sein, dass der gleiche Bezeichner für unterschiedliche Variablenarten (z.B. für lokale Variablen und für Membervariablen) benutzt worden ist.

Um diese Namenskonflikte aufzulösen, müssen die Bezeichner "qualifiziert" werden. Dies geschieht durch die Punktnotation: Vor den Namen werden Qualifier gestellt, die den Bezeichner näher spezifizieren:

- bei **Instanzelementen** (Variablen oder Methoden) wird die Referenzvariable als Qualifier benutzt, z.B. `instanz1.methodeA()` ;
- bei **Klassenelementen** wird der Klassenname als Qualifier benutzt, z.B. `klasseA.memberX`;
- bei **Klassennamen** wird der Paketname als Qualifier benutzt, z.B. `java.util.Date`

Für die qualifizierende Adressierung von Membervariablen wird also entweder der Name der Instanz oder der Name der Klasse vorangestellt - abhängig davon, ob das Element eine Objektvariable oder eine *static*-Variable ist.

**Programm ZugriffTest01: Qualifikation durch Referenzvariable**

```

class Zugriff01 {
    int zahl1;                                // Nicht private!
    void setZahl1(int z3) {
        zahl1 = z3;
    }
    void ausgeben() {
        System.out.println(zahl1);
    }
}

public class ZugriffTest01 {
    public static void main(String[] a) {
        Zugriff01 z01 = new Zugriff01();
        z01.setZahl1(15);
        z01.ausgeben();
        System.out.println(z01.zahl1);        // Direktaufruf
    }
}

```

Die Membervariable *zahl1* ist ein Element der Klasse *Zugriff01*. Sie ist **nicht** ausdrücklich als *private* deklariert. Deswegen gilt die Standardregel: Jede andere Klasse in diesem Paket kann auf dieses Element zugreifen (und es auch verändern). Der Zugriff erfolgt anders als innerhalb einer Klasse nun "qualifiziert", d. h. mit Hilfe des Punkt-Operators (*instanzname.feldname*).

**Übungen zum Programm Zugriff01**

**Übung 1:** Codieren Sie für die Membervariable *zahl1* den Zugriffsmodifizier *private* (das *e* nicht vergessen, denn *private* ist kein deutsches Wort). Versuchen Sie eine Umwandlung der Umwandlungseinheit *ZugriffTest01.java*. Ergebnis: es wird ein Umwandlungsfehler ausgegeben. Löschen Sie danach den Modifizier *private* wieder.

**Übung 2:** Ergänzen Sie die Klasse *ZugriffTest01* um folgende Aufgabe: Es soll eine zweite Instanz *z02* der Klasse *Zugriff01* erzeugt werden. Die Membervariable *zahl1* soll den Wert 27 bekommen. Danach soll die Methode *println* aufgerufen werden, die direkt die Membervariable *zahl1* referenziert, allerdings für die Instanz *z02*.

Diese Übung hat u.a. gezeigt, dass von der Standardregel für das Zugriffsrecht abgewichen werden kann, indem so genannte Zugriffsmodifizier (z.B. *private*) benutzt werden.

Bevor wir jedoch detailliert die einzelnen Schlüsselwörter für die Zugriffsrechte besprechen, müssen wir die Frage klären, was unter einem Paket zu verstehen ist und welche Rolle ein Package für das Zugriffsrecht spielt.

## 16.6 Bedeutung der Package-Namen

Ein wichtiges Ziel der objektorientierten Programmierung ist es, den Zugriff auf Daten und Methoden so stark wie möglich einzuschränken. Dadurch werden mögliche Fehlerquellen reduziert. Bei lokalen Variablen oder bei den Parameter-Variablen von Methoden ist die Begrenzung in die Sprache eingebaut: sie sind nur innerhalb eines Blocks bzw. einer Methode ansprechbar. Anders verhält es sich mit Member-Variablen. Die Membervariablen können nicht nur von allen Methoden der eigenen Klasse, sondern auch von Methoden abgeleiteter und sogar von total fremden Klassen angesprochen (gelesen, geschrieben und benutzt) werden, wenn diese zum gleichen Paket gehören. Einschränkungen dieser Zugriffsrechte sind - wie schon angedeutet - nur mit speziellen Modifiern möglich.

Jede Klasse in Java gehört zu einem Paket. Und die Zugriffsrechte werden abhängig von der Paketzugehörigkeit vergeben.

### Paketzugehörigkeit festlegen

Die Zugehörigkeit zu einem Paket wird in Java durch das Schlüsselwort *package* festgelegt. Diese Anweisung muss ganz oben in der Quelldatei stehen, sie gilt für die gesamte Umwandlungseinheit, also für alle Klassen dieser Quelldatei. Verbunden ist mit der Paketnamensvergabe auch, dass die Class-Dateien in einem Ordner mit diesem Namen stehen müssen.

Wenn die Angabe eines expliziten Packagenamens fehlt, gilt ein (unsichtbarer) Default-Packagename. Wenn eine Klasse aber die *package*-Anweisung enthält, dann wird der dort vergebene Bezeichner zum Namensbestandteil der Klasse. Das bedeutet, dass Elemente dieser Klasse von außerhalb nur angesprochen werden können mit dem vollen ("qualifizierten") Namen, der sich zusammensetzt aus Package- und Klassennamen.

### Bedeutung von Paketen

Die Bedeutung von Paketen liegt vor allem darin,

- dass jedes Paket einen eigenen Namensraum bildet, d.h. die Identifier müssen innerhalb eines Paketes eindeutig sein. Wenn in anderen Paketen dieselben Namen noch einmal vorkommen, werden sie durch Voranstellen des Paketnamens unterschieden.
- Die zweite wichtige Bedeutung der Paketbildung besteht darin, dass alle Elemente, die nicht ausdrücklich durch spezielle Modifizier gekennzeichnet sind, innerhalb dieses Pakets sichtbar sind. Auf sie kann von allen Klassen dieses Pakets zugegriffen werden.

Speichern Sie das folgende Programm *Zugriff02* in einem eigenen Ordner innerhalb Ihrer Rootdirectory ab. Der Packagename muss identisch sein mit dem Namen dieser Subdirectory. Also könnte diese Quelltext-Datei z.B. abgespeichert werden in folgendem Ordner: *c:/...root.../merker/Zugriff02.java*.

### Programm *Zugriff02*: Arbeiten mit Packagenamen

```
package merker;
class Zugriff02 {
    int zahl1;
    void setZahl1(int z3) {
        zahl1 = z3;
    }
    void ausgeben() {
        System.out.println(zahl1);
    }
}
```

Durch die Packageangabe in der ersten Zeile dieser Datei wird die Klasse zum Bestandteil eines Pakets, nämlich zum Element des Pakets *merker*. Der Package-Name wird Teil des Klassennamens, also voll qualifiziert heißt diese Klasse *merker.Zugriff02*.

### Übung zum Programm *Zugriff02*

Öffnen Sie einen Konsolbildschirm und verzweigen Sie in die Directory, in der sich das Quellenprogramm befindet. Wandeln Sie das Programm *Zugriff02* mit dem Aufruf des Compilers aus dieser Commandline um (und nicht aus einer Entwicklungsumgebung). Überprüfen Sie, in welchem Ordner sich die erzeugte Classdatei mit dem Bytecode befindet.

Speichern Sie die nachfolgende Datei in der Root-Directory des Ordners *merker* ab, also nicht in demselben Ordner, wie die Class-Datei *Zugriff02.java*, sondern in dem übergeordneten Folder.

### Programm *ZugriffTest02*: Benutzen der Klasse eines Pakets

```
import merker.*;
public class ZugriffTest02 {
    public static void main(String[] a) {
        Zugriff02 z1 = new Zugriff02();
        z1.setZahl1(15);
        z1.ausgeben();
    }
}
```

Durch die Angabe in der ersten Zeile dieser Datei kann dieses Programm auf die Class-File *merker.Zugriff02* zugreifen, ohne immer wieder den Packagenamen mit anzugeben. Alternativ kann diese Angabe auch entfallen, dann müsste die Erzeugung der Instanz in der vierten Zeile lauten:

```
merker.Zugriff02 z1 = new merker.Zugriff02();
```



### Übung zum Programm *ZugriffTest02*

Wandeln Sie danach das Programm *ZugriffTest02* um. Achten Sie darauf, dass dies aus der Commandline geschieht und dass Sie vorher in den Ordner verzweigen, in dem sich dieses Programm befindet. Die Umwandlung schlägt fehl, es kommt die Fehlermeldung:

```
"merker.Zugriff02 is not public in merker; cannot be accessed
from outside package".
```

Wie dieser Fehler zu beheben ist, werden wir im Abschnitt 16.7.1 besprechen. Es fehlt lediglich ein Schlüsselwort.

### Weitere Informationen zu Packages

Für die Vergabe von Package-Namen gilt: sie werden üblicherweise in Kleinbuchstaben geschrieben. Eine gebräuchliche Bezeichnung für Packages ist auch der Begriff "Library". Die gesamte mitgelieferte Klassenbibliothek der JDK ist organisiert in Paketen. Die API-Dokumentation sowohl für die Standard-Edition als auch für die Enterprise- und Micro-Edition ist nach Paketen unterteilt. Das folgende Beispielprogramm ermittelt die Anzahl der Pakete.

### Programm *PackageList01*: Wieviel Pakete gibt es in der J2SE?

```
public class PackageList01 {
    public static void main(String[] a) {
        java.lang.Package[] all = java.lang.Package.getPackages();
        System.out.println("Es gibt " + all.length + " Pakete");
    }
}
```

### Übung zum Programm *PackageList01*

Ändern Sie das Programm *PackageList01.java* so ab, dass auch die Namen der einzelnen Pakete - und nicht nur die Gesamtanzahl - ausgegeben werden.

### Lösungsvorschlag

```
public class PackageList02 {
    public static void main(String[] a) {
        java.lang.Package[] all = java.lang.Package.getPackages();
        for (int i=0; i<all.length; i++)
            System.out.println(all[i]);
    }
}
```

## 16.7 Zugriffsmodifizier *private*, *public*, *protected*

### 16.7.1 Zugriffsmodifizier für Felder

Die Vorgabe (default) ist, dass auf Klassen und ihre Elemente von allen anderen Klassen aus demselben Paket zugegriffen werden können.

Wenn eine Klasse aber versucht, auf Klassen oder deren Elemente zuzugreifen, die außerhalb des eigenen Packages sind, ist dies nicht möglich. Es sei denn, der Programmierer hat dies mit den entsprechenden Schlüsselwörtern ausdrücklich erlaubt. Und dieses Schlüsselwort ist *public*.

Das ist die Erklärung für den Umwandlungsfehler in dem Programm *Zugriff02* aus dem vorherigen Abschnitt. Die Lösung lautet also: Die Klasse muss *public* sein!

### Übung zu den Programmen *Zugriff02*/*ZugriffTest02*

Fügen Sie im Programm *Zugriff02.java* für die Klasse und für die beiden Methoden das Schlüsselwort *public* ein und wandeln Sie das Programm neu um.

#### Lösungsvorschlag

```
package merker;
public class Zugriff02 {
    int zahl1;
    public void setZahl1(int z3) {
        zahl1 = z3;
    }
    public void ausgeben() {
        System.out.println(zahl1);
    }
}
```

Vergessen Sie nicht, eventuell auch das Programm *ZugriffTest02* neu umzuwandeln. Danach müsste die Programmausführung fehlerfrei möglich sein.

### Welche Zugriffsmodifizier gibt es?

Es gibt drei Schlüsselwörter, um den Zugriff zu beschränken bzw. zu erweitern:

- *private*: nur innerhalb der eigenen Klasse zugreifbar,
- *public*: alle dürfen zugreifen,
- *protected*: nur abgeleitete Klassen (und die eigene natürlich).

Wenn keines dieser Schlüsselwörter angegeben ist, gilt der "**default access**":

- innerhalb eines Pakets ist alles auch von anderen Klassen nutzbar.

### Zugriffsmodifier und das "information hiding"

Es gilt also: Wenn der Programmierer nichts anderes ausdrücklich bestimmt, dann kann eine Variable von allen Programmen dieses Pakets verändert werden, d.h. sie ist nicht geschützt vor dem Zugriff aus anderen Klassen. Dies entspricht nicht der Idee der Datenkapselung, und die Empfehlung ist deshalb, Membervariablen mit dem Schlüsselwort *private* zu deklarieren, damit der Zugriff so eingegrenzt wird, dass von außerhalb der eigenen Klasse kein direktes Lesen und kein direktes Verändern (Update) möglich ist.

#### Programm *Private01*: Privat - deswegen Zugriff verboten

```
class Private01 {
    private int zahl;
}
class PrivateTest01 {
    public static void main(String[] a) {
        Private01 p1 = new Private01();
        System.out.println(p1.zahl);    // illegal
    }
}
```

Ein Zugriff auf die privaten Membervariablen ist in solchen Fällen nur indirekt möglich, durch Setter- und Getter-Methoden. Dies entspricht der empfohlenen Vorgehensweise. Wenn eine Einschränkung als *private*-Variable nicht möglich ist, so sollte geprüft werden, ob die Variable mit dem Modifier *protected* gekennzeichnet werden kann. Dies hat zur Folge, dass nur von abgeleiteten Klassen zugegriffen werden kann (der Zugriff "bleibt innerhalb der Verwandtschaft").

#### Programm *Protected01*: Zugriff nur für "Verwandte" und innerhalb des Pakets

```
class Protected01 {
    protected int zahl;
}
class ProtectedTest01 {
    public static void main(String[] a) {
        Protected01 p1 = new Protected01();
        System.out.println(p1.zahl);    // legal
    }
}
```

#### Übung zum Programm *ProtectedTest01*

**Übung 1:** Wandeln Sie das Programm um und testen Sie es. Sowohl die Umwandlung als auch die Ausführung sind ohne Probleme möglich. Nach welcher Regel ist der Zugriff auf die Variable *zahl* möglich?

**Lösungshinweis:** *protected* erlaubt den Zugriff entweder von allen abgeleiteten Klassen **oder aus demselben Package** heraus. Beide Klassen stehen in demselben Default-Package. Deswegen gibt es keine Probleme.

**Übung 2:** Bitte ordnen Sie zunächst nur die class *Protected01* einem (beliebigen) Package zu. Testen Sie. Danach ordnen Sie auch die class *ProtectedTest01* diesem Package zu. Dann müsste sowohl Umwandlung als auch Ausführung funktionieren.

Es gibt noch ein drittes Schlüsselwort für Variablen, um die Zugriffsrechte explizit zu ändern. Und das ist der Modifizier *public*. Damit wird **jedem** erlaubt, die so gekennzeichnete Variable zu lesen und/oder zu verändern. Dringende Empfehlung: dieses Schlüsselwort sollte nur in wohl begründeten Ausnahmefällen eingesetzt werden.

### 16.7.2 Zugriffsmodifizier für Methoden

Ein Zugriff auf eine Methode einer Klasse ist selbstverständlich auch von jeder Methode der eigenen Klasse möglich. Dies kann auch nicht eingegrenzt werden.

Aber weil auch hier die Vorgabe lautet: nur innerhalb eines Paketes kann diese Methode benutzt werden, ist es häufig notwendig, diese Rechte zu erweitern durch den Modifizier *public*. Damit wird es auch den Nutzern von außerhalb des Paketes möglich, diese Methode aufzurufen.

Eine Eingrenzung der Zugriffsrechte ist ebenfalls möglich: durch den Modifizier *private*. Dann darf diese Methode nur von Methoden der eigenen Klasse aufgerufen werden.

### 16.7.3 Zugriffsmodifizier für Klassen

Eine Klasse sollte mit dem Modifizier *public* versehen sein, wenn sie eine *main*-Methode enthält. Das heißt, eine Java-Applikation, die durch Aufruf des Interpreters gestartet wird, muss nicht nur in einer Umwandlungseinheit mit dem Namen dieser ausführbaren Klasse stehen, sondern diese Klasse sollte auch *public* sein. Und sie muss die einzige Klasse in dieser Umwandlungseinheit mit diesem Modifizier sein.

#### Programm *PublicTest01*: Wann dürfen Klassen *public* sein?

```
public class PublicTest01 {
    public static void main(String[] a) {
        Public01 p = new Public01();
    }
}
public class Public01 {
    private int zahl1;

    void setZahl1(int z3) {
        zahl1 = z3;
    }
}
```

```

    }
    void ausgeben() {
        System.out.println(zahl1);
    }
}

```

### Übungen mit Programm *PublicTest01*

**Übung 1:** Speichern Sie die obige Umwandlungseinheit in einer Quelldatei mit dem Namen *PublicTest01.java*. Bei der Umwandlung werden Fehlermeldungen ausgegeben. Bitte korrigieren Sie diesen Fehler. Lösungshinweis: Nur die ausführbare Klasse darf *public* sein.

**Übung 2:** Wenn Übung 1 erfolgreich war, speichern Sie die geänderte, fehlerfreie Umwandlungseinheit in einer Quelldatei mit dem Namen *Public01.java*. Versuchen Sie die Umwandlung dieser Datei. Warum funktioniert das nicht?

### Zusammenfassung

Für die Sicherheit und Wartbarkeit von Programmen sollten folgende Prinzipien beachtet werden:

**Datenkapselung:** Die Felder einer Klasse sollten, soweit möglich, mit dem Schlüsselwort *private* versehen werden. Dadurch wird ein Zugriff von außerhalb der Klasse strikt unterbunden. Für das Lesen und Schreiben können Getter- und Settermethoden eingesetzt werden. Diese Methoden müssen selbstverständlich von außerhalb zugreifbar sein, deswegen benötigen sie den Modifier *public*.

**Information Hiding:** Das Arbeiten innerhalb von Methoden (oder Programmblöcken) soll für den Nutzer dieser Programmteile möglich sein, ohne die Details der internen Implementierung zu kennen. Die Schnittstellen zum Aufrufer sollen möglichst einfach sein. Gemeinsame Datenbereiche sind - so weit möglich - zu vermeiden. Deswegen sollten die Variablen dieses Blocks möglichst *lokale Variable* sein; der Modifier *static* ist nur in begründeten Ausnahmefällen zu verwenden.

### Überblick der Access Level:

Modifier	class	Subclass	Package	Weltweit
<b>private</b>	x			
<b>protected</b>	x	x	x	
<b>public</b>	x	x	x	x
<b>(default)</b>	x		x	

Abb. 16.2: Welchen Zugriff erlauben die einzelnen Access-Modifier?

## A

**Installationshinweise J2SE SDK 5.0**

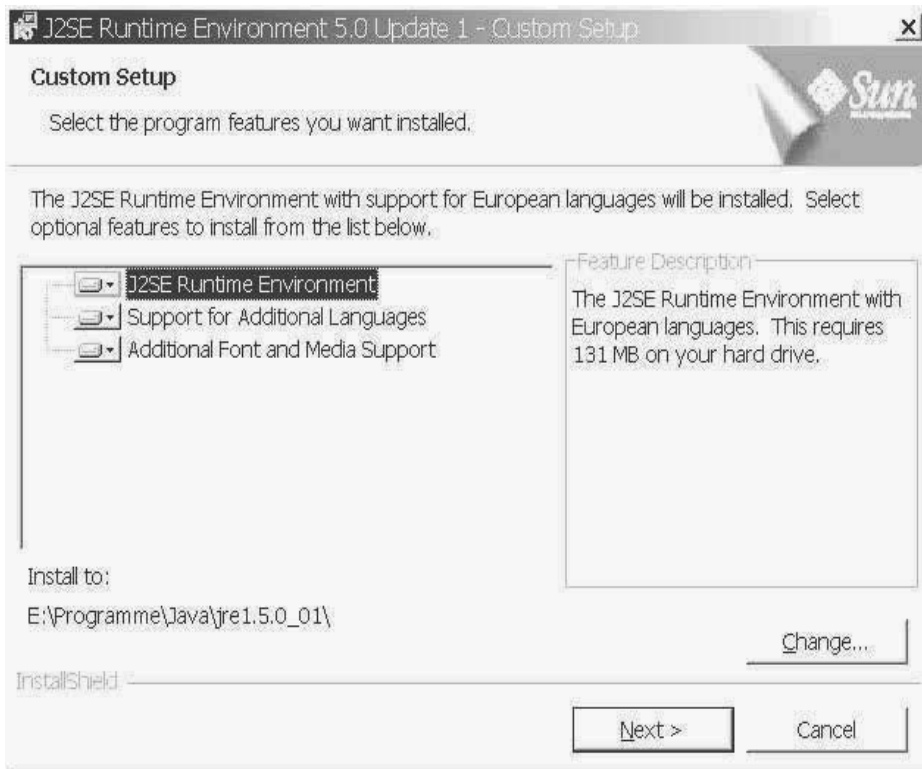
Für die Installation der J2SE SDK 5.0 unter MS-Windows sind folgende Schritte notwendig:

- Es werden Verwaltungsrechte benötigt.
- Gestartet wird die Datei, die von der Web-Site der Firma Sun geholt wurde, z.B. *jdk-1\_5\_0\_05-windows-i586-p.exe*
- Nach der Bestätigung der Lizenzvereinbarungen erscheint das folgende Fenster:

**A Installationhinweise J2SE SDK 5.0**

- Empfehlung: Alle Angaben unverändert lassen, eventuell kann der Installationsordner geändert werden.
- Den Button *Next* auswählen

- Das folgende Fenster bietet weitere Setup-Angaben an:



- Das Fenster bietet die Möglichkeit, die Unterstützung für nicht-europäische Sprachen zu installieren.
- Empfehlung: Alle Angaben unverändert lassen (lediglich die Installationsordner könnten geändert werden).
- Den Button *Next* auswählen
- Dann prüft das Installationsprogramm, welche Browser benutzt werden, und bietet für alle gefundenen Browser eine Registrierung des Java-Plug-Ins an.
- Folgendes Bild erscheint:



- Empfehlung: Alle Angaben unverändert lassen, damit das Plug-In für die Browser installiert wird.
- Das Java-Plug-In verknüpft den Browser mit der neu installierten Java-Plattform. Dadurch wird das Arbeiten mit Applets auf diesem System möglich.

In diesem Java-Lehrbuch wird das Thema "Applets" nicht behandelt. Die Gründe: Um einfache Java-Applets zu schreiben (oder auch nur zu verstehen), sind Kenntnisse in den Internet-Standards wie HTML und HTTP notwendig. Darüber hinaus muss man mit der Programmierung von grafischen Oberflächen vertraut sein. Selbstverständlich ist ein tiefes Verständnis der umfangreichen Klassenbibliotheken notwendig. Und sinnvoll ist der Einsatz von komplexen Entwicklungsumgebungen wie z.B. Eclipse. All dies macht es dem Anfänger zu Beginn relativ schwer, mit Applets zu arbeiten.

- Den Button *Next* auswählen
- Zum Abschluss der Installation kommt folgendes Fenster:

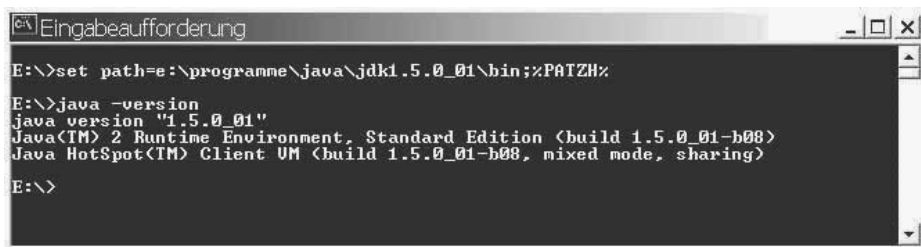




- Herzlichen Glückwunsch: das J2SE Developer Kit 5.0 ist auf Ihrem Rechner verfügbar.

### Änderung der *path*-Environment-Variable

Damit die notwendigen Programme zum Umwandeln und Ausführen aus einer CMD-Shell ("Eingabeaufforderung" unter Windows) beim Aufruf auch gefunden werden, muss nun abschließend die PATH-Environment-Variable ergänzt werden um den *bin*-Ordner der Installations-Directory. Dies geschieht entweder temporär durch Eingabe des folgenden Commands:



oder über das ICON "Arbeitsplatz": Bei Windows 2000 durch Aufruf mit der rechten Maustaste , danach: *Eigenschaften*|*Erweitert*|*Umgebungsvariablen*.

### Hinweise zum *path*-Suchpfad:

- Diese Änderungen werden erst wirksam nach dem Neustart des DOS-Fensters.
- Die Angaben in der PATH-Environment-Variablen beschreiben für das Programm "command.com" die Suchreihenfolge der ausführbaren Programme (.exe-Dateien). Wenn also z.B. das Java-Programm "progr1" ausgeführt werden soll, wird der Befehl "java progr1" eingegeben, und dann sucht Windows die *java.exe*-Datei nach den Pfad-Angaben in der **path**-Environment-Variablen. (Das Programm *progr1* ist ein Java-Programm vom Dateityp .class; dieses wird gesucht anhand des **classpath**-Suchpfads, falls es nicht im Arbeitsordner steht.)
- Die einzelnen Suchpfade werden im *Classpath* durch Semikolon abgetrennt.
- Durch die Angabe *%path%* wird der derzeitige Stand eingefügt.
- Natürlich kann dieser Befehl auch in einer Batch-Datei stehen (.bat).

### Änderung der *classpath*-Environment-Variable

Für die Umwandlung und Ausführung der Beispiele in diesem Buch sind **keine** Änderungen in der *classpath*-Variablen erforderlich.

Der *classpath* beschreibt die Suchreihenfolge für die Suche nach Class-Dateien, also für Dateien mit der Endung .class. Sie enthält die Directories, die durchsucht werden. Benötigt werden diese Angaben für die Umwandlung und bei der Ausführung. Die Standardannahme für die Suche einer .class-Datei ist, dass diese im aktuellen Ordner steht. Wenn also der Aufruf aus dem Ordner erfolgt, wo die Datei steht, ist keine besondere *classpath*-Angabe notwendig.

Sollten Sie eine *classpath*-Environmentvariable einrichten oder ändern (z.B. weil class-Dateien referenziert werden, die in einem anderen Ordner stehen), so achten Sie darauf, **dass die Environment-Variable auch einen Punkt enthält**.

Der Punkt hat eine wichtige Bedeutung für die Suche, denn er bestimmt, dass auch der aktuelle Pfad bei der Suche einbezogen wird. Beispiel:

```
set classpath=%classpath%;d:\sun\lib\j2ee.jar; .;
```

Eine weitere Möglichkeit, den Classpath zu setzen, besteht beim Aufruf des Java-Compilers und des Java-Interpreters. Beide Programme erlauben eine *-classpath* Option beim Starten.

```
E:\>java -cp e:\merker Test01
```

Natürlich wirkt die Option *-cp* nur für die Suche nach CLASS-Dateien (nicht für die Suche von Quelldateien (.java-Dateien)).

## B

## Meta-Sprachen zur Syntaxbeschreibung

Die Syntax einer Sprache kann mit einer speziellen Meta-Sprache ("eine Sprache, die eine andere Sprache erklärt") beschrieben werden. Beispiele für Metasprachen sind die Backus-Naur-Form (BNF) oder spezielle Syntaxdiagramme.

## 1. Syntaxdiagramme

Grafische Darstellung der Syntax. Sie besteht aus Rechtecken, die Erläuterungen oder Hinweise auf andere Syntaxdiagramme enthalten, und Kreisen (oder Ovale), die Symbole enthalten. Syntaxdiagramme werden von links nach rechts, den Linien folgend, interpretiert. Beispiel:

Dezimalzahl:

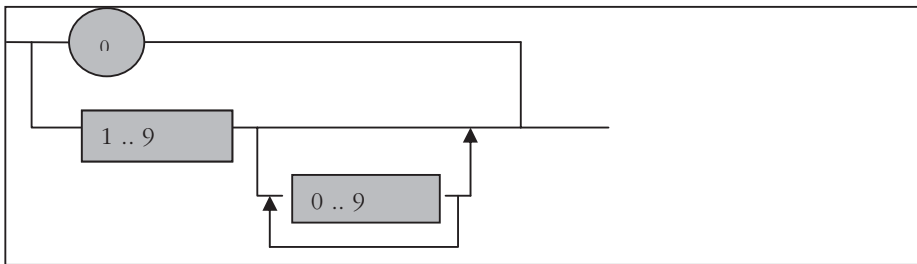


Abb. B1: Syntaxdiagramm (Was ist eine Dezimalzahl in Java?)

## 2. Backus-Naur-Form (BNF)

Ein anderes Beispiel ist die textuelle Darstellung der Sprachsyntax. Angelehnt ist dies meistens an die seit den 60er Jahren bekannte Backus-Naur-Form (BNF), bei der durch festgelegte Sonderzeichen ("Metazeichen") beschrieben wird, wie die formal richtige Codierung des Quelltexts zu erfolgen hat.

Es gibt zahlreiche Varianten. Hier die von Sun benutzten Symbole:

- |       |  |
|-------|--|
| a: z  | a ist der zu erklärende Begriff, z enthält die Beschreibung (a wird durch z definiert) |
| x   y | trennt Alternativen (entweder x oder y)  |
| [ ]   | wahlweises Vorkommen (0-mal oder 1-mal), optional                                      |
| { }   | Auswahl, Vorkommen ist 1-mal oder beliebig oft   |
| +     | Element kann wiederholt werden   |

Das nachfolgende Beispiel ist angelehnt an die *Language Specification* von Sun (siehe <http://java.sun.com/docs/books/jls/>).

### BNF-Beispiel - Variablendeklaration:

```
VariableDeclarators:
    VariableDeclarator { ,    VariableDeclarator }

VariableDeclarator:
    Identifier VariableDeclaratorRest

VariableDeclaratorRest:
    BracketsOpt [ =    VariableInitializer]
```

### Kurzbeschreibung für Control-Statements:

```
if (Expression) Statement [else Statement]

for ( ForInitOpt ; [Expression] ; ForUpdateOpt ) Statement

while (Expression) Statement

do Statement while (Expression) ;
```

### Ausführliche Beschreibung (Switch):

```
switch (Expression) { SwitchBlockStatementGroups }
SwitchBlockStatementGroups:
    { SwitchBlockStatementGroup }
SwitchBlockStatementGroup:
    SwitchLabel BlockStatements
SwitchLabel:
    case ConstantExpression    :
    default:
BlockStatements:
    { BlockStatement }
BlockStatement :
    LocalVariableDeclarationStatement
    ClassOrInterfaceDeclaration
    [Identifier :] Statement
```

C

Die ersten 256 Unicode-Zeichen (0000-ffff)

Code pint( dez.)	hexa- dez.	Sym- bol <b>Unicode-Namen</b> (Bezeichnung)	<b>CP 1252</b> Windows	<b>CP 850</b> D O S
0	0x0000	NULL*		
1	0x0001	START OF HEADING*		
2	0x0002	START OF TEXT*		
3	0x0003	END OF TEXT*		
4	0x0004	END OF TRANSMISSION*		
5	0x0005	ENQUIRY*		
6	0x0006	ACKNOWLEDGE*		
7	0x0007	BELL*		
8	0x0008	BACKSPACE*		
9	0x0009	CHARACTER TABULATION*		
10	0x000A	LINE FEED (LF) *		
11	0x000B	LINE TABULATION*		
12	0x000C	FORM FEED (FF) *		
13	0x000D	CARRIAGE RETURN (CR) *		
14	0x000E	SHIFT OUT*		
15	0x000F	SHIFT IN*		
16	0x0010	DATA LINK ESCAPE*		
17	0x0011	DEVICE CONTROL ONE*		
18	0x0012	DEVICE CONTROL TWO*		
19	0x0013	DEVICE CONTROL THREE*		
20	0x0014	DEVICE CONTROL FOUR*		
21	0x0015	NEGATIVE ACKNOWLEDGE*		
22	0x0016	SYNCHRONOUS IDLE*		
23	0x0017	END OF TRANSMISSION BLOCK*		
24	0x0018	CANCEL*		
25	0x0019	END OF MEDIUM*		
26	0x001A	SUBSTITUTE*		
27	0x001B	ESCAPE*		
28	0x001C	INFORMATION SEPARATOR FOUR*		
29	0x001D	INFORMATION SEPARATOR THREE*		
30	0x001E	INFORMATION SEPARATOR TWO*		
31	0x001F	INFORMATION SEPARATOR ONE*		

Teil 1 von 8: Codepoint 0000-0031: Steuerzeichen, identisch mit US-ASCII

## C Die ersten 256 Unicode-Zeichen (0000-ffff)

Code- point dezim.	he- xade z.	Sym- bol	Unicode-Namen (Bezeichnung)	CP 1252 Windows	CP 850 D O S
32	0x0020		SPACE		
33	0x0021	!	EXCLAMATION MARK		
34	0x0022	"	QUOTATION MARK		
35	0x0023	#	NUMBER SIGN		
36	0x0024	\$	DOLLAR SIGN		
37	0x0025	%	PERCENT SIGN		
38	0x0026	&	AMPERSAND		
39	0x0027	'	APOSTROPHE		
40	0x0028	(	LEFT PARENTHESIS		
41	0x0029	)	RIGHT PARENTHESIS		
42	0x002A	*	ASTERISK		
43	0x002B	+	PLUS SIGN		
44	0x002C	,	COMMA		
45	0x002D	–	HYPHEN-MINUS		
46	0x002E	.	FULL STOP		
47	0x002F	/	SOLIDUS		
48	0x0030	0	DIGIT ZERO		
49	0x0031	1	DIGIT ONE		
50	0x0032	2	DIGIT TWO		
51	0x0033	3	DIGIT THREE		
52	0x0034	4	DIGIT FOUR		
53	0x0035	5	DIGIT FIVE		
54	0x0036	6	DIGIT SIX		
55	0x0037	7	DIGIT SEVEN		
56	0x0038	8	DIGIT EIGHT		
57	0x0039	9	DIGIT NINE		
58	0x003A	:	COLON		
59	0x003B	;	SEMICOLON		
60	0x003C	<	LESS-THAN SIGN		
61	0x003D	=	EQUALS SIGN		
62	0x003E	>	GREATER-THAN SIGN		
63	0x003F	?	QUESTION MARK		

Teil 2 von 8: Codepoint 0032-0063: Ziffern und Sonderzeichen, identisch mit US-ASCII

Code-point dez.	hexa- dez.	Sym- bol	Unicode-Namen (Bezeichnung)	CP 1252 Windows	CP 850 D O S
64	0x0040	@	COMMERCIAL AT		
65	0x0041	A	LATIN CAPITAL LETTER A		
66	0x0042	B	LATIN CAPITAL LETTER B		
67	0x0043	C	LATIN CAPITAL LETTER C		
68	0x0044	D	LATIN CAPITAL LETTER D		
69	0x0045	E	LATIN CAPITAL LETTER E		
70	0x0046	F	LATIN CAPITAL LETTER F		
71	0x0047	G	LATIN CAPITAL LETTER G		
72	0x0048	H	LATIN CAPITAL LETTER H		
73	0x0049	I	LATIN CAPITAL LETTER I		
74	0x004A	J	LATIN CAPITAL LETTER J		
75	0x004B	K	LATIN CAPITAL LETTER K		
76	0x004C	L	LATIN CAPITAL LETTER L		
77	0x004D	M	LATIN CAPITAL LETTER M		
78	0x004E	N	LATIN CAPITAL LETTER N		
79	0x004F	O	LATIN CAPITAL LETTER O		
80	0x0050	P	LATIN CAPITAL LETTER P		
81	0x0051	Q	LATIN CAPITAL LETTER Q		
82	0x0052	R	LATIN CAPITAL LETTER R		
83	0x0053	S	LATIN CAPITAL LETTER S		
84	0x0054	T	LATIN CAPITAL LETTER T		
85	0x0055	U	LATIN CAPITAL LETTER U		
86	0x0056	V	LATIN CAPITAL LETTER V		
87	0x0057	W	LATIN CAPITAL LETTER W		
88	0x0058	X	LATIN CAPITAL LETTER X		
89	0x0059	Y	LATIN CAPITAL LETTER Y		
90	0x005A	Z	LATIN CAPITAL LETTER Z		
91	0x005B	[	LEFT SQUARE BRACKET		
92	0x005C	\	REVERSE SOLIDUS		
93	0x005D	]	RIGHT SQUARE BRACKET		
94	0x005E	^	CIRCUMFLEX ACCENT		
95	0x005F	_	LOW LINE		

Teil 3 von 8: Codepoint 0064-0095: Großbuchstaben und Sonderzeichen, identisch mit US-ASCII

Code-point dezim.	hexa- dez.	Sym- bol	Unicode-Namen (Bezeichnung)	CP 1252 Windows	CP 850 D O S
96	0x0060	`	GRAVE ACCENT		
97	0x0061	a	LATIN SMALL LETTER A		
98	0x0062	b	LATIN SMALL LETTER B		
99	0x0063	c	LATIN SMALL LETTER C		
100	0x0064	d	LATIN SMALL LETTER D		
101	0x0065	e	LATIN SMALL LETTER E		
102	0x0066	f	LATIN SMALL LETTER F		
103	0x0067	g	LATIN SMALL LETTER G		
104	0x0068	h	LATIN SMALL LETTER H		
105	0x0069	i	LATIN SMALL LETTER I		
106	0x006A	j	LATIN SMALL LETTER J		
107	0x006B	k	LATIN SMALL LETTER K		
108	0x006C	l	LATIN SMALL LETTER L		
109	0x006D	m	LATIN SMALL LETTER M		
110	0x006E	n	LATIN SMALL LETTER N		
111	0x006F	o	LATIN SMALL LETTER O		
112	0x0070	p	LATIN SMALL LETTER P		
113	0x0071	q	LATIN SMALL LETTER Q		
114	0x0072	r	LATIN SMALL LETTER R		
115	0x0073	s	LATIN SMALL LETTER S		
116	0x0074	t	LATIN SMALL LETTER T		
117	0x0075	u	LATIN SMALL LETTER U		
118	0x0076	v	LATIN SMALL LETTER V		
119	0x0077	w	LATIN SMALL LETTER W		
120	0x0078	x	LATIN SMALL LETTER X		
121	0x0079	y	LATIN SMALL LETTER Y		
122	0x007A	z	LATIN SMALL LETTER Z		
123	0x007B	{	LEFT CURLY BRACKET		
124	0x007C		VERTICAL LINE		
125	0x007D	}	RIGHT CURLY BRACKET		
126	0x007E	~	TILDE		
127	0x007F		DELETE*		

Teil 4 von 8: Codepoint 0096-0127: Kleinbuchstaben und Sonderzeichen, identisch mit US-ASCII



Code-point dez.	hexa- dez.	Sym- bol	Unicode-Namen (Bezeichnung)	CP 1252 Windows	CP 850 D O S
128	0x0080	*		80 €	80 Ç
129	0x0081	*			81 Û
130	0x0082		BREAK PERMITTED HERE*	82 ,	82 é
131	0x0083		NO BREAK HERE*	83 f	83 â
132	0x0084	*		84 ”	84 ä
133	0x0085		NEXT LINE (NEL) *	85 ...	85 à
134	0x0086		START OF SELECTED AREA*	86 †	86 å
135	0x0087		END OF SELECTED AREA*	87 ‡	87 ç
136	0x0088		TABULATION SET*	88 ^	88 ê
137	0x0089		TABULATION WITH JUST.*	89 ‰	89 ë
138	0x008A		LINE TABULATION SET*	8A Š	8A è
139	0x008B		PARTIAL LINE FORWARD*	8B ‹	8B ĩ
140	0x008C		PARTIAL LINE BACKWARD*	8C Œ	8C î
141	0x008D		REVERSE LINE FEED*		8D ì
142	0x008E		SINGLE SHIFT TWO*	8E Ž	8E Ä
143	0x008F		SINGLE SHIFT THREE*		8F Å
144	0x0090		DEVICE CONTROL STRING*		90 É
145	0x0091		PRIVATE USE ONE*	91 ‘	91 æ
146	0x0092		PRIVATE USE TWO*	92 ’	92 Æ
147	0x0093		SET TRANSMIT STATE*	93 “	93 ô
148	0x0094		CANCEL CHARACTER*	94 ”	94 õ
149	0x0095		MESSAGE WAITING*	95 •	95 ò
150	0x0096		START OF GUARDED AREA*	96 —	96 ù
151	0x0097		END OF GUARDED AREA*	97 —	97 ù
152	0x0098		START OF STRING*	98 ~	98 ÿ
153	0x0099	*		99 ™	99 Ö
154	0x009A		SINGLE CHARACTER INTR.*	9A š	9A Ü
155	0x009B		CONTROL SEQUENCE INTR.*	9B ›	9B ø
156	0x009C		STRING TERMINATOR*	9C œ	9C £
157	0x009D		OPERATING SYSTEM CM*		9D Ø
158	0x009E		PRIVACY MESSAGE*	9E ž	9E ×
159	0x009F		APPLICATION PR. CMD*	9F Ÿ	9F f

Teil 5 von 8: Codepoint 0128-0159: Im Unicode als Steuerzeichen belegt  
Unterschiedlich belegt bei CP1252, CP850 oder 8859-1

Code-point dez..	hexa- dez.	Sym- bol	Unicode-Namen (Bezeichnung)	CP 1252 Windows	CP 850 D O S
160	0x00A0		NO-BREAK SPACE	A0	A0 á
161	0x00A1	¡	INVERTED EXCLAM. MARK	A1 ¡	A1 í
162	0x00A2	¢	CENT SIGN	A2 ¢	A2 ó
163	0x00A3	£	POUND SIGN	A3 £	A3 ú
164	0x00A4	¤	CURRENCY SIGN	A4 ¤	A4 ñ
165	0x00A5	¥	YEN SIGN	A5 ¥	A5 Ñ
166	0x00A6	¦	BROKEN BAR	A6 ¦	A6 ¢
167	0x00A7	§	SECTION SIGN	A7 §	A7 ¢
168	0x00A8	¨	DIAERESIS	A8 ¨	A8 ¿
169	0x00A9	©	COPYRIGHT SIGN	A9 ©	A9 ®
170	0x00AA	ª	FEMININE ORDINAL INDICATOR	AA ¢	AA ¬
171	0x00AB	«	LEFT-POINTING DOUBLE ANGLE	AB «	AB ½
172	0x00AC	¬	NOT SIGN	AC ¬	AC ¼
173	0x00AD		SOFT HYPHEN	AD	AD ¡
174	0x00AE	®	REGISTERED SIGN	AE ®	AE «
175	0x00AF	ˉ	MACRON	AF ¯	AF »
176	0x00B0	°	DEGREE SIGN	B0 °	B0 ☐
177	0x00B1	±	PLUS-MINUS SIGN	B1 ±	B1 ☐
178	0x00B2	²	SUPERSCRPT TWO	B2 ²	B2 ☐
179	0x00B3	³	SUPERSCRPT THREE	B3 ³	B3
180	0x00B4	´	ACUTE ACCENT	B4 ´	B4
181	0x00B5	µ	MICRO SIGN	B5 µ	B5 Á
182	0x00B6	¶	PILCROW SIGN	B6 ¶	B6 Â
183	0x00B7	·	MIDDLE DOT	B7 ·	B7 À
184	0x00B8	¸	CEDILLA	B8 ¸	B8 ©
185	0x00B9	¹	SUPERSCRPT ONE	B9 ¹	B9 ¶
186	0x00BA	º	MASCULINE ORDINAL INDICATOR	BA ¢	BA
187	0x00BB	»	RIGHT-POINTING DOUBLE ANGLE	BB »	BB ¶
188	0x00BC	¼	ONE QUARTER	BC ¼	BC ¶
189	0x00BD	½	ONE HALF	BD ½	BD ¢
190	0x00BE	¾	THREE QUARTERS	BE ¾	BE ¥
191	0x00BF	¿	INVERTED QUESTION MARK	BF ¿	BF 7

Teil 6 von 8: Codepoint 0160-0191: Unicode, CP1252 und 8859-1 identisch.  
CP850 hat abweichende Belegung

Code-point dezim.	hexa- dez.	Sym- bol	Unicode-Namen (Bezeichnung)	CP 1252 Windows	CP 850 D O S
192	0x00C0	À	A WITH GRAVE	C0 À	C0 L
193	0x00C1	Á	A WITH ACUTE	C1 Á	C1 Ł
194	0x00C2	Â	A WITH CIRCUMFLEX	C2 Â	C2 T
195	0x00C3	Ã	A WITH TILDE	C3 Ã	C3
196	0x00C4	Ä	A WITH DIAERESIS	C4 Ä	C4 -
197	0x00C5	Å	A WITH RING ABOVE	C5 Å	C5 †
198	0x00C6	Æ	LETTER AE	C6 Æ	C6 ā
199	0x00C7	Ç	C WITH CEDILLA	C7 Ç	C7 Â
200	0x00C8	È	E WITH GRAVE	C8 È	C8 Ł
201	0x00C9	É	E WITH ACUTE	C9 É	C9 ₣
202	0x00CA	Ê	E WITH CIRCUMFLEX	CA Ê	CA Ł
203	0x00CB	Ë	WITH DIAERESIS	CB Ë	CB ₣
204	0x00CC	Ì	I WITH GRAVE	CC Ì	CC ₣
205	0x00CD	Í	I WITH ACUTE	CD Í	CD =
206	0x00CE	Î	I WITH CIRCUMFLEX	CE Î	CE ₣
207	0x00CF	Ï	I WITH DIAERESIS	CF Ï	CF □
208	0x00D0	Ð	LETTER ETH	D0 Ð	D0 ð
209	0x00D1	Ñ	N WITH TILDE	D1 Ñ	D1 Ð
210	0x00D2	Ò	O WITH GRAVE	D2 Ò	D2 Ê
211	0x00D3	Ó	O WITH ACUTE	D3 Ó	D3 Ë
212	0x00D4	Ô	O WITH CIRCUMFLEX	D4 Ô	D4 È
213	0x00D5	Õ	O WITH TILDE	D5 Õ	D5 ,
214	0x00D6	Ö	O WITH DIAERESIS	D6 Ö	D6 Í
215	0x00D7	×	MULTIPLICATION SIGN	D7 ×	D7 Î
216	0x00D8	Ø	O WITH STROKE	D8 Ø	D8 Ï
217	0x00D9	Ù	U WITH GRAVE	D9 Ù	D9 J
218	0x00DA	Ú	U WITH ACUTE	DA Ú	DA r
219	0x00DB	Û	U WITH CIRCUMFLEX	DB Û	DB ■
220	0x00DC	Ü	U WITH DIAERESIS	DC Ü	DC ■
221	0x00DD	Ý	Y WITH ACUTE	DD Ý	DD ¡
222	0x00DE	Þ	LETTER THORN	DE Þ	DE ì
223	0x00DF	ß	SMALL LETTER SHARP S	DF ß	DF ■

Teil 7 von 8: Codepoint 0192-0223: Unicode, CP1252 und 8859-1 identisch  
CP850 hat abweichende Belegung

Code-point dezim.	hexa- dez.	Sym- bol	Unicode-Namen (Bezeichnung)	CP 1252 Windows	CP 850 D O S
224	0x00E0	à	a WITH GRAVE	E0 à	E0 Ó
225	0x00E1	á	a WITH ACUTE	E1 á	E1 ß
226	0x00E2	â	a WITH CIRCUMFLEX	E2 â	E2 Ô
227	0x00E3	ã	a WITH TILDE	E3 ã	E3 Ò
228	0x00E4	ä	a WITH DIAERESIS	E4 ä	E4 õ
229	0x00E5	å	a WITH RING ABOVE	E5 å	E5 Ö
230	0x00E6	æ	LETTER AE	E6 æ	E6 μ
231	0x00E7	ç	c WITH CEDILLA	E7 ç	E7 þ
232	0x00E8	è	e WITH GRAVE	E8 è	E8 þ
233	0x00E9	é	e WITH ACUTE	E9 é	E9 Û
234	0x00EA	ê	e WITH CIRCUMFLEX	EA ê	EA Û
235	0x00EB	ë	e WITH DIAERESIS	EB ë	EB Û
236	0x00EC	ì	i WITH GRAVE	EC ì	EC ý
237	0x00ED	í	i WITH ACUTE	ED í	ED Ý
238	0x00EE	î	i WITH CIRCUMFLEX	EE î	EE -
239	0x00EF	ï	i WITH DIAERESIS	EF ï	EF '
240	0x00F0	ð	LETTER ETH	F0 ð	F0
241	0x00F1	ñ	n WITH TILDE	F1 ñ	F1 ±
242	0x00F2	ò	o WITH GRAVE	F2 ò	F2 =
243	0x00F3	ó	o WITH ACUTE	F3 ó	F3 ¾
244	0x00F4	ô	o WITH CIRCUMFLEX	F4 ô	F4 ¶
245	0x00F5	õ	o WITH TILDE	F5 õ	F5 §
246	0x00F6	ö	o WITH DIAERESIS	F6 ö	F6 ÷
247	0x00F7	÷	DIVISION SIGN	F7 ÷	F7 „
248	0x00F8	ø	o WITH STROKE	F8 ø	F8 °
249	0x00F9	ù	u WITH GRAVE	F9 ù	F9 "
250	0x00FA	ú	u WITH ACUTE	FA ú	FA •
251	0x00FB	û	u WITH CIRCUMFLEX	FB û	FB ¹
252	0x00FC	ü	u WITH DIAERESIS	FC ü	FC º
253	0x00FD	ý	y WITH ACUTE	FD ý	FD º
254	0x00FE	þ	LETTER THORN	FE þ	FE ■
255	0x00FF	ÿ	y WITH DIAERESIS	FF ÿ	FF

Teil 8 von 8: Codepoint 0224-0255: Unicode, CP1252 und 8859-1 identisch  
CP850 hat abweichende Belegung

# D

## Komplettbeispiel einer verteilten Application

---

Das Kapitel 6.1.3 (Streams) enthält folgendes Beispielprogramm zum Schreiben und Lesen über eine TCP/IP-Leitung.

### Programm Stream04: Server für Kommunikation über TCP/IP-Verbindung

```
import java.net.*;
import java.io.*;
class Stream04 {
    public static void main(String[] args) throws Exception {
        ServerSocket ss = new ServerSocket(1500);
        Socket s = ss.accept();

        DataInputStream ein = new
            DataInputStream(s.getInputStream());

        int zahl = ein.read();
        System.out.println(zahl);
    }
}
```

Zum völligen Verständnis sind einige Kenntnisse des TCP/IP-Protokolls erforderlich. Das Programm erwartet über die Port-Nr. 1500 ein Zeichen, das gelesen und danach am Consolebildschirm ausgegeben wird. Hier ist das entsprechende Senderprogramm.

### Programm Stream04a: Client für Kommunikation über TCP/IP-Verbindung

```
import java.net.*;
import java.io.*;
class Stream04a {
    public static void main(String[] args) throws Exception {
        Socket s = new Socket("localhost", 1500);
        DataOutputStream aus = new
            DataOutputStream(s.getOutputStream());
        aus.write('A');
        aus.close();
    }
}
```

### Hinweise zum Testen der Anwendung:

Es werden **zwei** DOS-Boxen benötigt. Achtung: Diese DOS-Boxen dürfen nicht über JOE geöffnet werden, weil dann Environment-Variable geändert werden. Es ist wie folgt vorzugehen:

- START|Programme|Zubehör|Eingabeaufforderung (zweimal).
- Danach jeweils in die Ordner verzweigen, in denen die Class-Dateien stehen.
- Dann zunächst das Serverprogramm starten: *java Stream04*
- Und jetzt das Clientprogramm starten: *java Stream04a*.

Das Ergebnis der wundervollen Zusammenarbeit dieser beiden Programme ist, dass auf dem Consolebildschirm des Serverprogramms der Codepoint des Buchstabens A ausgegeben wird.

### Übung zum Programm Stream04

**Übung 1:** Ändern Sie das Programm so, dass nicht der Codepoint 65, sondern der Buchstabe A vom Serverprogramm ausgegeben wird. (...println(**((char)**zahl);

**Übung 2:** Ändern Sie die beiden Kommunikationsprogramme so ab, dass nicht einzelne Zeichen gesendet und gelesen werden, sondern komplette Zeichenketten (Strings) mit den Methoden writeUTF und readUTF ausgetauscht werden.

### Lösungsvorschlag

```
import java.net.*;
import java.io.*;
class Stream05 {
    public static void main(String[] args) throws Exception {
        ServerSocket ss = new ServerSocket(1500);
        Socket s = ss.accept();
        DataInputStream ein = new DataInputStream(s.getInputStream());
        String text = ein.readUTF();
        System.out.println(text);
    }
}
import java.net.*;
import java.io.*;
class Stream05a {
    public static void main(String[] args) throws Exception {
        Socket s = new Socket("localhost",1500);
        DataOutputStream aus = new
            DataOutputStream(s.getOutputStream());
        aus.writeUTF("Hallo");
        aus.close();
    }
}
```

# E

## Glossar

---

### **Abstraktion**

Allgemeine Vorgehensweise, um Komplexität zu verringern. Dabei wird versucht, die für eine bestimmte Aufgabenstellung wichtigen Teile zu erkennen und das Unwesentliche zu vernachlässigen. Wird in der Programmierung z.B. beim Design von Klassen eingesetzt, um die wesentlichen Merkmale eines Gegenstandes oder Begriffs herauszusondern.

**access modifier**, siehe Modifizier

### **Algorithmus**

Beschreibung für die Lösung eines bestimmten Problems, konkrete Anleitung, Handlungsvorschrift; kann die Form eines Computerprogramms haben.

### **Aggregation**

beschreibt, wie einzelne Teile vereinigt werden zu etwas Ganzem. In der objektorientierten Programmierung die Beschreibung, wie Klassen miteinander verbunden werden, nämlich in eine Ganze-Teile-Hierarchie ("is-part-of"). In Java erfolgt Aggregation dadurch, dass ein Objekt eine Referenz auf ein anderes Objekt erzeugt. Siehe auch *Komposition*.

### **API** Application Programming Interface

bezeichnet eine Sammlung von Schnittstellenspezifikationen, die einem Anwendungsprogramm zur Verfügung stehen, um eine bestimmte vorprogrammierte Funktion zu nutzen, z.B. den Zugriff auf Datenbanken oder das Arbeiten mit XML. Ein API besteht aus der Beschreibung der Methoden, die vom Programmierer für die Anwendungsentwicklung verwendet werden können (in Java häufig in Form von *Interfaces*).

### **Application** (Java-Anwendung)

In Java unterscheidet man unterschiedliche Programmformen. Eine Application ist ein vollständiges, eigenständiges Javaprogramm ("stand alone-program"), das eine *static* Main-Methode enthält und per Betriebssystem-Befehl zur Ausführung aufgerufen werden kann. Andere Programmformen sind Applets, Servlets, Java Server Pages. Diese erfordern zusätzlich so genannte Container für die Ausführung, z.B. Webserver oder Browser.

**Arbeitsspeicher**, siehe Hauptspeicher

**Architektur**

In der Anwendungsentwicklung die Spezifikation der grundlegenden Struktur eines Systems.

**Argument**, siehe Parameter

**ASCII** American Standard Code for Information Interchange

eine Zuordnung der gebräuchlichsten 128 Zeichen (Buchstaben und Ziffern) zu einem 7-bit-Code. Der ASCII-Standard hat sich in den 60er Jahren zur Zeit der Datenübertragung mittels Telex entwickelt. Mit 7 Bit erfasst der ASCII-Code zunächst aber nur die Buchstaben des lateinischen Alphabets, die Ziffern und einige Steuerungs-codes, die z.B. für die Druckersteuerung benötigt werden. Das 8. Bit des Bytes war ein Prüfbit. Später hat man auf die Prüfung jedes Einzelbytes verzichtet, dadurch standen zusätzliche 128 Zahlenkombinationen zur Verfügung. Diese wurde unterschiedlich genutzt - es entstanden länderspezifische Ergänzungen des ASCII-Codes, z.B. ISO-8859-1 für Westeuropa, in dem auch die deutschen Sonderzeichen (Umlaute und ß) enthalten sind.

**Assoziation**

beschreibt eine Beziehung (Relation), die zwischen Klassen besteht. Dabei kann es sich um ganz unterschiedliche Arten der Relation handeln, die jedoch nicht näher spezifiziert sein müssen.

**Attribut**, siehe Feld

**Auswahl (Selektion)**

Konstrukt der Ablaufsteuerung, beschreibt die Verzweigung innerhalb eines Programms aufgrund von Bedingungen. In Java realisiert durch *if* und *switch*-statements.

**Backslash**

Der Slash "/", der "auf dem Rücken liegt", also Rückwärtsstrich "\". Im Windowsdateisystem zur Trennung der Ordner eines Pfades eingesetzt. In Java auch benutzt für die Angabe von Escape-Sequenzen (siehe dort).

**Basisklasse**, siehe Superklasse

**BCD-Code** (binary coded decimal)

Verschlüsselungsverfahren für Zahlen. Dabei wird nicht das rein binäre Zahlensystem wie z.B. bei dem *int*-Typ verwendet, sondern die Stellenwertigkeit des Dezimalsystems beibehalten, damit Ungenauigkeiten durch den Wechsel des Stellenwertsystems vermieden werden. Jede Dezimalziffer wird in 4 bits (ein Halbbyte) codiert. Wird in Java von der Klasse *BigInteger* benutzt und insbesondere für kommerzielle Anwendungen eingesetzt.

**blank** (space)

Leerzeichen, im Unicode "\u0020" (nicht zu verwechseln mit "null"); das wichtigste Whitespace-Character (siehe dort).



### **Boolesche Algebra**

Von Boole (1815 - 1864) entwickelte Logik, die mit dem Dualzahlensystem als Basis algebraische Operationen durchführt. Diese auch als Schaltalgebra bezeichnete Logik wird bei der Entwicklung von digitalen Schaltungen benutzt und ist die Grundlage für die logischen Operatoren in den Programmiersprachen. Die wichtigsten Operationen sind UND, ODER, NICHT.

### **Bytecode**

Maschinenunabhängiger (Zwischen-)Code, der vom Java-Compiler erstellt und von dem Java-Interpreter (JVM) ausgeführt wird.

### **Casting**

Explizites Konvertieren von Objekten oder von primitiven Datentypen mit Hilfe eines Operators, dem "Castoperator". Voraussetzung ist eine Typverträglichkeit. Die Typumwandlung kann auch erfolgen: mit speziellen Methoden, implizit (automatisch) ohne Castoperator oder durch Autoboxing.

### **Character**

engl. Zeichen, in Java verschlüsselt nach dem Unicode in 16 bit (UTF-16).

### **Codepoint**

bezeichnet den (dezimalen oder hexadezimalen) Wert, den ein Zeichen im Unicode hat. Die Codepoints des ASCII-Zeichensatz umfassen dezimal 0 - 127 (hexadezimal 00-7F). Beispiel: Das Multiplikationszeichen \* hat den dezimalen Codepoint 215.

### **Codierung**

- a) Phase der Softwareentwicklung, in der die Übertragung in die Programmiersprache erfolgt; das Schreiben des Quelltextes
- b) verschlüsseln, z.B. Zeichen in dem Unicode.

**compilation unit**, siehe Umwandlungseinheit

### **Compiler**

Programm, das die Quelltext-Anweisungen aus einer symbolischen Programmiersprache, z.B. C++ oder Pascal, prüft und in den Maschinencode übersetzt. In Java wird zunächst ein maschinenunabhängiger Zwischencode erstellt (Bytecode). Das Programm, das Java-Quelltext in den Bytecode übersetzt, wird ebenfalls als Compiler bezeichnet und heißt *java.exe*.

### **Datenflussplan**

Symbolische Darstellung des Datenflusses innerhalb eines Informationssystems. Enthält grafische Symbole für die Datenträger und für die Programme sowie Ablauflinien für den Datenfluss. Genormt nach DIN 66001.

### **Debugging**

Englischer Ausdruck für das Lokalisieren und Beseitigen von Programmfehlern.

**Default**

engl. für Standard, Voreinstellung; z.B. Default-Konstruktor oder Default-Wert bei der Initialisierung.

**Definition**

Ein Statement, das Speicherplatz reserviert für eine Variable.

**Deklaration (Vereinbarung)**

Ein Statement, das dem Compiler einen Identifier und seinen Datentyp bekannt macht. Eine Deklaration reserviert keinen Speicherplatz für eine Instanz.

**Design-Pattern**, siehe Entwurfsmuster

**Dimension**

Festlegung der Größe eines Arrays durch Angabe eines ganzzahligen Wertes. Java-beispiel: der Wert 5 legt fest, dass das Array aus 5 Komponenten besteht und damit der Zugriff auf die einzelnen Komponenten dieses Arrays über die Indices von 0 bis 4 erfolgt.

**Encapsulation** siehe Kapselung

**Entwurf (Design)**

Phase der Softwareentwicklung, in der aus der (groben) Leistungsbeschreibung die technische Systemarchitektur entwickelt wird.

**Entwurfsmuster (Design Pattern)**

Vorschläge für den Entwurf von objektorientierten Softwaresystemen. Sie sind eine Art "Rezept" für gutes und effizientes Programmieren und beschreiben eine generalisierte Lösungsidee zu immer wiederkehrenden Entwurfsproblemen. Sie sind oft das Ergebnis von jahrelanger Erfahrung in komplexen Projekten und beschreiben bewährte Lösungsansätze. Sie enthalten aber keine fertig codierten Lösungen.

**Escape-Sequenz**, siehe Steuerzeichen

**Exception**

Ausnahmesituation bei der Ausführung eines Java-Programms ("run-time-error"). Tritt eine Ausnahme auf, so wird ein Objekt "geworfen", d.h. vom Anwendungsprogramm an die JVM übergeben. Die JVM sucht dann nach einem Exception-Handler, d.h. nach einer Methode, die in der Lage ist, auf diese Exception zu reagieren.

**Expression**

engl. für Ausdruck; Teil eines Statements, besteht aus Operanden und Operatoren bzw. Methodenaufrufen. Eine Expression wird ausgewertet und liefert *ein* Ergebnis. Der Datentyp des Ergebnisses bestimmt den Datentyp des Gesamtausdrucks.

**Feld (field)**

Unterschiedliche Bedeutung. Manchmal wird ein *Array* als Feld bezeichnet. In Java meistens im Sinne von: ein Datenelement einer Klasse. Beispiel: Die Klasse *Kunden* enthält ein Feld *name* und ein Feld *adresse*. Ein Feld kann selbst wiederum ein Objekt sein.

### **Framework**

Ein Satz von kooperierenden Klassen mit einer engen Verbindung untereinander. Beispiel: *Swing*, ein Framework für das Erstellen einer grafischen Java-Anwendung mit GUI-Oberfläche.

### **Garbage Collection**

Der Mechanismus, der in Java dafür sorgt, dass nicht mehr benötigter Speicherplatz ("garbage") wieder frei gegeben wird für die Benutzung durch andere Programme.

### **Grafisches User-Interface (GUI)**

Die Möglichkeit, ein Programm mit Tastatur und Maus (oder ähnlichen Geräten) zu bedienen und Informationen nicht nur zeilenweise, sondern pixelweise in Bildschirmfenstern darzustellen.

### **Hauptspeicher (auch: Arbeitsspeicher)**

RAM (Random Access Memory), interner Speicher des Computers, enthält neben dem Betriebssystem auch die auszuführende Anwendung. Das Anwendungsprogramm arbeitet mit Daten, die als Variablen (oder auch als Konstanten oder Literale) im Hauptspeicher stehen müssen. Daten, die außerhalb des Arbeitsspeichers stehen, können nicht direkt verarbeitet werden, sondern müssen zunächst eingelesen werden. Sie sind dort solange verfügbar, wie das Programm aktiv ist. Sollen die Daten auch außerhalb des Programms existieren, so müssen sie entweder als "Stream-Objekte" an eine andere Java Virtuelle Maschine transferiert werden oder als "persistente Objekte" z.B. auf einen externen Datenträger wie Magnetplatte oder Speicherstick, ausgelagert werden.

### **IDE Integrated Development Environment**

Entwicklungsumgebung, die komfortable Möglichkeiten bietet zum Editieren des Source-Codes und zum Compilieren. Enthalten sind häufig Dokumentationstools, Build-Tools, Werkzeuge zur Projektverwaltung, Generierung von Programmteilen z.B. für GUI-oder Help-Programme, integrierte Debugger. Produkte sind z.B. Eclipse, Netbeans von Sun, Borlands JBuilder oder Websphere Studio von IBM.

### **Implementierung**

a) Phase der Softwareentwicklung, in der aus der Leistungsbeschreibung mit Entwurfshilfsmitteln wie Programmaufbauplan oder Struktogrammen der Quellcode der Programme erstellt wird (siehe auch Codierung).

b) Bei verteilten Systemen die Phase, in der die fertig ausgetesteten Programme auf den Run-Time-Systemen installiert werden (auch Deployen genannt).

### **Information Hiding (Geheimnisprinzip)**

Dieses Konzept beschreibt eine wichtige Technik, um "gute" Software zu erstellen. Denn dadurch wird es unmöglich gemacht, dass Objekte den internen Zustand anderer Objekte in unerwarteter Weise lesen oder ändern (siehe auch "Kapselung"). Für den Anwender eines Moduls (in Java: Klasse) bedeutet "information hiding", dass nicht nur die Datenstrukturen, sondern auch die Algorithmen verborgen sind, er muss sich mit den internen Details nicht beschäftigen.

**Initialisieren**

Einen Startwert (Anfangswert) für eine Variable vergeben, entweder automatisch oder durch Anweisungen des Programmierers.

**Inkrement**

Erhöhung, Zuwachs; gewöhnlich angegeben als Differenz zwischen zwei Arbeitsschritten. Inkrementieren in Java: erhöhen des Variableninhalts um einen festen Wert, z.B. eine Laufvariable in einer For-Schleife mit `i++`;

**Instanz (instance)**

Synonym für Objekt. Ein konkret im Speicher vorhandener Einzelfall einer Klasse. Benutzt die Klassenbeschreibung als Schablone für das Anlegen von Arbeitsspeicherplatz ("**instanziieren**") und erlaubt es dem Programmierer, die Methoden der Klassen mit Hilfe des Instanznamens aufzurufen. Jede Instanz hat denselben Satz von Attributen wie die anderen Instanzen dieser Klasse, aber sie hat ihre individuellen Werte, die diesen Attributen zugewiesen worden sind.

**Instanz-Methode**

Eine Methode, die auf Instanzen einer Klasse operiert. Für den Aufruf einer Instanzmethode muss vorher ein Objekt erzeugt worden sein. Dann kann an diese Instanz eine Nachricht geschickt werden in der Form: *object.methode()*;

**Instanz-Variable**

Eine Variable, die als Attribute eines Objektes definiert wurde. Die Klasse definiert den Typ und den Identifier einer Variablen, aber die Instanz belegt Speicherplatz und füllt ihn mit Werten.

**Interface**, siehe Schnittstelle

**Interpreter**

Ein Programm, das einen Source-Code Zeile für Zeile interpretiert und ausführt. Eine klassische Interpretersprache ist JavaScript. In Java wird nicht der Quelltext interpretiert, sondern der Bytecode. Die Run-Time-Umgebung, in der der Bytecode ausgeführt wird, enthält den Interpreter. Er wird aufgerufen durch *java <programmname>*.

**Iteration**, siehe Schleife

**JavaScript**

Bezeichnung für eine von Netscape entwickelte Script-Sprache für HTML. JavaScript dient hauptsächlich zur Erweiterung von HTML, es ist eine betriebssystemunabhängige Scriptsprache, die in die WWW-Browser wie den Firefox integriert ist. JavaScript ist nicht Java. Beispiele für die Unterschiede: JavaScript ist eine spezialisierte Sprache für Web-Client-Anwendungen, Java ist eine umfangreiche Technologie, auch und gerade für Server-Anwendungen. JavaScript wird zur Laufzeit interpretiert, Java wird (vor-)compiliert. JavaScript ist objektbasierend und kennt keine Vererbung, Java ist voll objektorientiert. JavaScript ist allenfalls eine Alternative zu Java-Applets.

### **Java Development Kit (JDK)**

Software-Entwicklungsumgebung für das Erstellen und Übersetzen von Java-Applicationen und Java-Applets, definiert und herausgegeben von SUN Microsystems. Kann auch von anderen Herstellern angeboten werden, wenn die Lizenz vorhanden ist. Jedes JDK enthält mindestens folgende Werkzeuge (tools): Java Compiler, Java Virtual Machine, Java Class Libraries, Java Applet Viewer, Java Debugger.

### **Java Runtime Environment (JRE)**

Ein Subset des JDK; enthält nur die JVM und die Java Core Classes. Wird benötigt, wenn lediglich Java-Programme ausgeführt werden sollen, also fertig compilierte Class-Files vorhanden sind.

### **Java Virtual Machine (JVM)**

Eine Software-Implementierung der virtuellen CPU (Central Procssing Unit), die compilierten Javacode (Bytecode) interpretieren und ausführen kann. Ist Teil der Java-Laufzeitumgebung (JRE, Java Runtime Environment).

### **Kapselung (encapsulation)**

Daten und ihre darauf operierenden Methoden sind nicht getrennt, sondern werden als Einheit betrachtet. Daten und Methoden sind in einer Kapsel, wobei die Daten innen liegen und im Idealfall ein Zugriff nur möglich ist über die öffentlichen Methoden dieser Kapsel (dieses "Moduls"). In Java ist das Modul eine Klasse. Das Ziel des "encapsulation" ist es, die Sicherheit der Programme zu erhöhen, weil die Möglichkeiten und die Verantwortung, Daten zu manipulieren, genau festgelegt sind. Eng mit diesem Prinzip verknüpft ist das Prinzip des "information hiding".

### **Keyword**

engl. für Schlüsselwort; die Schlüsselwörter bilden den Kern einer Programmiersprache. Java kennt etwa 50 keywords, jedes ist ein vordefiniertes Wort mit festgelegter Bedeutung. Ein Keyword kann nicht benutzt werden als Bezeichner (identifier).

### **Klasse**

Ein zentraler Begriff in der OOP. Klassen sind Programmmodule. Jede Klasse ist eine Programmeinheit, die eine Beschreibung für eine Menge von Objekten enthält. Sie beschreibt Daten und die dazu gehörenden Methoden. Dann kann sie als Schablone genutzt werden, um Instanzen zu erzeugen.

### **Klassenbibliothek**

eine Sammlung von Klassen: in Java entspricht dies einem Paket. Das ist eine organisatorische Einheit, die Klassen und Interfaces zusammenfasst in einem Ordner und die einen Namensraum bildet. Physikalisch kann die Bibliothek auch als Archiv gepackt vorliegen (.jar-File). Das J2SE-API ist die Klassenbibliothek der JDK.

### **Klassendiagramm (UML)**

Ein Klassendiagramm zeigt eine Menge von Klassen und ihre Beziehungen.

**Klassenmethode**

Eine Methode, die mit dem Schlüsselwort *static* deklariert wurde. Gehört zur Klasse als Ganzes und nicht zur Instanz. Sie wird nicht mit dem Instanznamen, sondern mit dem Klassennamen aufgerufen (wird auch *static method* genannt).

**Klassenvariable**

Eine Variable, die mit dem Schlüsselwort *static* deklariert wurde. Sie existiert pro Klasse nur einmal im Arbeitsspeicher, unabhängig von der Anzahl der Instanzen (wird auch *static field* genannt).

**Komponente**

a) Ein ausführbares Softwaremodul mit klar definierten Schnittstellen; häufig wird dieser Begriff benutzt in Verbindung mit dem remoten Aufruf von Methoden (RPC, in Java: RMI oder EJB).

b) Ein Element einer Reihung (arrays), direkter Zugriff über Index (Platznummer).

**Komposition**

In der OOP eine besondere Form der Beziehung zwischen Klassen: Ein Objekt ("Großobjekt") besitzt als Datenfeld ein anderes Objekt ("Kleinobjekt"). Beide sind untrennbar miteinander verbunden, die Lebensdauer beider Objekte ist identisch.

**Konsole**

Ein zeilenorientiertes Fenster für Kommandozeileneingabe. Wird in Java als Standard-Eingabegerät (*System.in*) und als Standard-Ausgabegerät (*System.out*) angesprochen.

**Konstruktor**

Eine besondere Art einer Methode, die automatisch aufgerufen wird, wenn eine neue Instanz einer Klasse mit dem Schlüsselwort *new* erzeugt wird. Ein Konstruktor hat denselben Namen wie die Klasse selbst und kann z.B. benutzt werden, um die Datenfelder des Objekts auf individuelle Anfangswerte zu setzen (zu initialisieren).

**Literal**

In der Java-Spezifikation definierte Zeichenfolge zur Darstellung von Werten der Basistypen und von Strings, z.B. bedeutet 'a' ein Wert des *char*-Datentyps und "a" der Wert eines Strings.

**Lokale Variable**

Eine Variable, die innerhalb einer Methode bzw. eines Blocks deklariert wird. Sie kann nur benutzt werden innerhalb dieser Methode bzw. innerhalb dieses Blocks.

**Loop**, siehe Schleife

**Member (einer Klasse)**

Elemente einer Klasse, dazu gehören die Felder und die Methoden.

**Methode**

allgemein: eine Handlungsvorschrift, die beschreibt, wie ein Ziel bzw. Ergebnis unter gegebenen Bedingungen erreicht werden kann. In Java: ein Codeblock innerhalb

einer Klasse, der eine bestimmte Aufgabe ausführen kann. Die Methode implementiert also eine Operation und wird mit einem Namen aufgerufen. Dabei können Parameter übergeben und ein Ergebnis zurückgeliefert werden. Vergleichbar mit einem Unterprogramm, einer Subroutine, Funktion oder einer Prozedur in anderen Programmiersprachen.

**MIME** (Multipurpose Internet Mail Extension)

beschreibt den Datentyp einer kompletten Datei, z.B. einer aus dem Netz geladenen Ressource. Ursprünglich lediglich für die Beschreibung von *e-mail*-Attachment genutzt, weil innerhalb der *e-mail* nur einfacher ASCII-Text möglich ist. Mime wird heute von Web-Browsern und Java-Programmen zur Identifikation von Dateiformaten, auch für reine Binärdateien, verwendet. Beispiel: "*text/plain*".

**Modifier**

Schlüsselwörter bei der Deklaration einer Variablen, Methode oder Klasse zur verfeinerten Festlegung von Eigenschaften, z.B. durch den access modifier *public*.

**Modul**

Abgeschlossener Programmbaustein mit eigenem Namen und mit eigenen (gekapselten) Variablen. Vorteile des modularen Aufbaus: saubere Zuordnung der Verantwortung, parallele Programmentwicklung möglich, Aufruf ohne Kenntnis der inneren Struktur möglich, Wartbarkeit wird erleichtert, Wiederverwendbarkeit möglich.

**MVC Model/View/Control**

beschreibt das modulare Design von GUI-Anwendungen, um überschaubare Sourcen zu codieren. Die Anwendung wird nicht als monolithisches Programm codiert, sondern sie besteht aus einzelnen Modulen mit klar abgegrenzten Aufgaben: *Model* enthält die Logik für die eigentliche Anwendung, *View* enthält das Benutzer-Interface und *Control* beschreibt das Steuermodul für den Gesamtablauf der Application.

**Nachricht** (message)

ist der Mechanismus, wie Objekte untereinander kommunizieren. Eine Nachricht besteht üblicherweise aus drei Teilen: Instanz, Methodenname und Parameter.

**NaN** (not-a-number)

Ein besonderer Wert der Datentypen *double* und *float*, der ein undefinierbares Ergebnis einer mathematischen Operation (z.B. Null geteilt durch Null) repräsentiert.

**Nassi-Shneidermann-Diagramm**

Darstellungs- und Entwurfsmittel, das die Struktur des Programms dokumentiert, auch Struktogramm genannt. Die Notation ist genormt durch DIN 66261.

**Nebeneffekt**

Wenn ein Ausdruck nicht nur ausgewertet ("evaluiert") wird, sondern nebenbei auch den Wert einer Variablen ändert, so nennt man dies Nebeneffekt (nicht zu verwechseln mit Seiteneffekt).

**null**

engl. für "Nichts" (umgangssprachlich: "null and void" bedeutet: null und nichtig). In Java der Nullzeiger, der ins Nichts zeigt, wenn eine Referenzvariable noch keinen Wert enthält. Nicht zu verwechseln mit der Ziffer 0, die im Unicode eine feste Bitcodierung hat.

**Oberklasse**, siehe Superklasse

**Objekt**, siehe Instanz

**Objektidentität**

Dadurch unterscheidet sich ein Objekt im Speicher von allen anderen. In Java ist es die Referenzvariable, die auf einen bestimmten Speicherbereich verweist und dadurch die Instanz identifiziert, auch wenn möglicherweise ein anderes Objekt mit gleichen Attributwerten existiert.

**OOP Objektorientierte Programmierung**

Eine Programmiermethode, die auf den Konzepten Vererbung und Datenabstraktion beruht. Steht im Gegensatz zur prozeduralen Programmierung.

**Operation**

Ein Service, der auf Anforderung für ein bestimmtes Objekt aufgerufen werden kann. In Java wird eine Operation als Methode einer Klasse implementiert.

**Overloading**, siehe Überladen

**Overriding**, siehe Überschreiben

**Paket (package)**

Eine Sammlung von mehreren Klassen und/oder Interfaces, die organisatorisch und logisch zusammengehören. In Java identisch mit dem Namen des Ordners, in dem die Klassen aufbewahrt werden.

**Parameter**

Daten für den Austausch von Informationen beim Methodenaufruf. Man unterscheidet die *formalen* Parameter, für die bei der Definition einer Methode ihr Datentyp und ihr Identifier angegeben werden, von den *aktuellen* Parametern ("Argumenten"), das sind die Werte, die beim Aufruf einer Methode angegeben werden.

**Parsen**

Allgemeine Bezeichnung aus der Sprachanalyse für das Analysieren und Zerlegen von Texten. In der Informatik kommt Parsing in vielen Bereichen vor. Wird in Java z.B. benutzt, um eine Zeile, Seite oder Datei auf bestimmte Begriffe ("Token") zu überprüfen, um diese abzutrennen oder durch andere Begriffe zu ersetzen.

**Performanz**

engl. performance, bezeichnet die Effizienz der Programmausführung. Oft ist damit lediglich die Ausführungsdauer, also die Antwortzeit für den Nutzer gemeint. Doch umfasst "Performanz" den gesamten Ressourcenverbrauch, also z.B. auch die verbrauchte CPU-Zeit und die Größe des Hauptspeicherplatzes, der belegt wird.



### **Phasenmodell**

Vorgehensplan, der den Softwareentwicklungsprozess inhaltlich und zeitlich in abgeschlossene Phasen zerlegt.

### **Programm**

Arbeitsanweisung an den Computer, besteht im Quellcode aus Symbolen ("Token") der Programmiersprache und im Ausführungscode aus den Maschinenbefehlen. In Java gibt es noch einen Zwischencode (Bytecode), der vom Compiler erzeugt und vom Interpreter ausgeführt wird.

### **Programmablaufplan (PAP)**

Darstellungs- und Entwurfsmittel, das mit genormten Symbolen den Ablauf eines Programms zeigt. Die einzelnen Verarbeitungsschritte werden mit Pfeilen verbunden. Die Notation ist genormt durch DIN 66001.

### **Prototyp**

Ein vorläufiges, unvollständiges Produkt. Meistens wird es als temporäres Muster mit unvollständigen Eigenschaften des endgültigen Produktes erstellt, damit gemeinsam mit dem Kunden (Endbenutzer) die Funktion experimentell überprüft und weiterentwickelt werden kann.

### **Prozess**

Verwaltungseinheit für das Betriebssystem. Der Prozess ist der Eigentümer der Programmressourcen wie Arbeitsspeicherplatz (Adressraum), Dateien, Drucker usw. In Java kann ein Prozess aus einem oder mehreren Threads bestehen; einer dieser Threads muss die Methode *main* haben (Hauptthread).

### **Pseudocode**

Darstellungs- und Entwurfsmittel, das mit Hilfe einer halb-standardisierten natürlichen Sprache die Funktion und den Ablauf eines Programmes beschreibt.

### **Qualifier**

Bei Verwenden von Namen für Variablen, Methoden, Klassen oder Paketen wird ein vom Programmierer vergebener Identifier (Name) benutzt. In manchen Situationen genügt der einfache Name (z.B. *zahl*), in anderen Fällen muss jedoch ein Qualifier vorangestellt werden, um den Identifier näher zu spezifizieren (z.B. *objekt1.zahl*). Grundsätzliche Regel in Java: Innerhalb einer Klasse kann ein Member mit dem einfachen Namen angesprochen werden, von außerhalb muss entweder der Klassenname oder ein Instanzname vorangestellt werden, abgetrennt durch einen Punkt.

### **Quelltext** (Quellenprogramm, Sourcecode)

Textform eines Programms; wird vom Programmierer mit einem Editor geschrieben, ist nur dann unmittelbar lauffähig, wenn es von einem Interpreter ausgeführt wird. In den meisten Programmiersprachen müssen die Quellenprogramme umgewandelt (compiliert) werden in das Maschinenprogramm. In Java steht der Quelltext von einer oder mehreren Klassen im ASCII-Code in einer Umwandlungseinheit (*.java*-File), der vom Java-Compiler in den Bytecode (*.class*-Files) umgewandelt wird.

**RAM** (Random Access Memory), siehe Arbeitsspeicher

**Referenz** (reference)

Eine (RAM-)Speicheradresse für ein Objekt. In Java werden Objekte und Reihungen (*arrays*) immer über Referenzen angesprochen. Außerdem werden Objekte beim Methodenaufruf als Referenz übergeben (und nicht als Kopie der Werte).

**Rückgabewert** (return value)

Ergebnis eines Methodenaufrufs. Der Datentyp dieses Ergebnisses wird im Kopf der aufgerufenen Methode definiert (Ergebnistyp).

**Schleife** (Iteration, Loop, Wiederholung)

Konstrukt der Ablaufsteuerung, sorgt dafür, dass eine oder mehrere Anweisungen wiederholt ausgeführt werden, gesteuert von einer Bedingung. In Java realisiert durch die Schlüsselwörter *while*, *do* und *for*.

**Schlüsselwort**, siehe keyword

**Schnittstelle** (interface)

Allgemein: Techniken, die beschreiben, wie Klienten einen Service nutzen können. Dabei werden Programmschnittstellen (API) von Benutzerschnittstellen (GUI oder Kommandozeilen) unterschieden. In Java beschreiben *interfaces* den extern sichtbaren Teil eines Moduls; manchmal beschrieben in einer eigenen Quelldatei, die umwandelbar ist. Sie enthält eine Liste der Signaturen der nicht-privaten Methoden, ohne Implementierungscode. Ein Interface ermöglicht es einer Klasse, sich auf dieses Interface zu beziehen durch das Schlüsselwort *implements*. Dann ist die Klasse verpflichtet, alle Methoden der Schnittstelle zu implementieren.

**Scope** (Zuständigkeitsbereich)

Gültigkeitsbereich und Lebensdauer von Variablen. Die Gültigkeit bestimmt, wo innerhalb eines Programms ein Identifier genutzt werden kann. Die Lebensdauer beschreibt, ab wann und wie lange eine Variable im Arbeitsspeicher existiert. In Java können Felder und Methoden einer Klasse standardmäßig in der gesamten Klasse angesprochen werden. Ausnahme: Lokale Variablen, die nur innerhalb des Blocks, in dem sie deklariert worden sind, genutzt werden können.

**Script-Sprachen**

Spezialisierte Programmiersprachen (im Gegensatz zu den universellen höheren Programmiersprachen), ursprünglich gedacht für die Automatisierung von System-Administrator-Aufgaben. Ein Script ist ein Programm, das wie jedes andere Programm aus einer Folge von Befehlen besteht. Ein Script, z.B. eine DOS-Batch-Datei, wird zeilenweise gelesen und interpretiert. Andere Script-Sprachen sind z.B. Shell-Scripts in Unix, JavaScript, JScript, Perl, PHP, Tcl, Python, Rexx. Der Vorteil von Script-Sprachen liegt in der einfachen Erstellung und Pflege. Jede Änderung ist sofort lauffähig. Nachteile sind geringere Ausführungsgeschwindigkeit und eingeschränkte, spezialisierte Funktionen.

**Seiteneffekt**

damit werden Veränderungen an nicht lokalen Variablen durch ein gerufenes Unterprogramm bezeichnet. Dies ist in vielen Situationen nicht erwünscht, weil es zu heimtückischen Fehlern führen kann. Nicht zu verwechseln mit Nebeneffekt.

**Selektion**, siehe Auswahl

**Sequenz**

"normale" Form der Ablaufsteuerung, wenn der Programmierer nichts anderes bestimmt; bezeichnet eine Folge von Anweisungen, die nacheinander (sequentiell, seriell) ausgeführt werden.

**Signatur**

kennzeichnet eine Methode eindeutig, nämlich durch den Methodennamen sowie die Typen und die Reihenfolge der formalen Parameter. Die Parameternamen und der Rückgabetyt gehören nicht zur Signatur.

**Software-Engineering**

Sammelbezeichnung für den Einsatz anerkannter Prinzipien, Methoden und Techniken bei der Planung und Durchführung von EDV-Projekten.

**Spezialisierung**

Gegenteil der Generalisierung. Bei einer "top-down"-Entwicklung geht man von generellen Strukturen (Klassifikationen) aus, die dann schrittweise verfeinert werden. Bei der Klassenbildung ist es das Vorgehen, das von allgemeinen Klassen die Vererbungshierarchien ableitet.

**Static Field**, siehe Klassenvariable

**Static Methode**, siehe Klassenmethode

**Steuerzeichen**

häufig auch bezeichnet als "escape-sequence"; Zeichen, die nicht angezeigt oder gedruckt werden, sondern als Kommando z.B. für den Compiler zur Darstellung von Sonderzeichen oder für die Steuerung eines Peripheriegeräts erkannt werden. In Java können diese Steuerzeichen in Literalen enthalten sein. Sie beginnen mit dem Backslash (Rückwärts-Schrägstrich \), z.B. '\t' für die Tabulator-Funktion.

**Strukturiertes Design**

Methode zum Entwurf modularer Systeme. Qualität des Entwurfs wird an der Modulkopplung und der Modulfestigkeit gemessen

**Strukturierte Programmierung**

Vorgehensweise beim Entwurf und Implementierung eines Einzelprogrammes. Hat das Ziel, einen übersichtlichen Quellcode zu erstellen. Ursprünglich im Wesentlichen darauf ausgerichtet, die Ablaufkonstrukte zu beschränken auf Sequenz, Selektion und Iteration (d.h. Vermeidung von GOTO).

**Subklasse** (Unterklasse, Kindklasse)

Eine Klasse, die abgeleitet worden ist von einer bestehenden Klasse und dadurch alle Methoden und Variablen dieser Superklasse erbt. Die Unterklasse ist eine Spezialisierung einer Oberklasse, in der die allgemeinen Eigenschaften und Methoden beschrieben sind (siehe auch Vererbung). Die Superklasse kann wiederum Subklasse einer anderen Klasse in der Hierarchie sein.

**Superklasse** (Oberklasse, Basisklasse, Elternklasse)

Die Superklasse ist die Klasse, von der eine andere abgeleitet wird und somit all deren Eigenschaften erbt. Die Superklasse beschreibt den allgemeinen Fall, von dem dann mittels Vererbung Unterklassen als Spezialfälle gebildet werden können.

**Syntax** (Grammatik)

die Regeln, wie die Wörter, Zahlen und Sonderzeichen in einem Source-Programm geschrieben werden müssen, damit sie vom Compiler richtig verstanden und richtig übersetzt werden können.

**Token**

Zeichenmuster; Symbole, die in der Syntax eine Bedeutung haben. In einem Java-Programm alle Zeichen, die kein Kommentar sind oder kein Whitespace (siehe dort).

**Top-Down-Design**

Vorgehen, bei dem man ein Problem in Teilprobleme zerlegt, diese wieder in Teilprobleme usw., bis man überschaubare Aufgaben erhält.

**Typkonvertierung**, siehe Casting**Überladen** (overloading)

Die Fähigkeit, mehrere unterschiedliche Methoden mit demselben Identifier, aber mit einer unterschiedlichen Anzahl oder mit unterschiedlichen Datentypen der Argumente zu haben. Auch Konstruktoren können überladen werden, ebenfalls durch Verwendung des gleichen Namens, aber mit verschiedener Parameterliste.

**Überschreiben** (override)

Wenn in einer Subklasse eine Methode mit gleicher Signatur wie in der Superklasse definiert wird und damit die Methode in der Superklasse ersetzt, so wird dies als Override (engl. überfahren, sich hinwegsetzen) bezeichnet.

**UML** Unified Modeling Language

eine grafische Beschreibung von objektorientierten Software-Systemen. Ist durch die Object Management Group (OMG) international standardisiert.

**Umwandlungseinheit** (compilation unit)

Quellendatei, die eine oder mehrere Klassen enthält. Sie hat die Dateiendung *.java*. Nach der Umwandlung durch den Compiler entsteht für jede Klasse eine eigene Datei mit der Dateiendung *.class*. Eine Compilation-Unit besteht aus drei Teilen: *package*-Deklaration, *import*-Deklaration und Klassen-Deklarationen.

### **Unicode**

Ein 16/32-bit-Code für Text-Zeichen in verschiedenen Sprachen, definiert durch ISO 10646 (siehe auch ASCII-Code). ASCII- und Latin-1-Zeichen sind mit den ersten 256 Unicodezeichen identisch. Java verwendet den 16-bit-Unicode (UTF-16) - und zwar für die Datentypen *char* und *String*. Unicode definiert den numerischen Wert aller bekannten Zeichen, aber es gibt keine Angaben darüber, wie das Zeichen darzustellen ist. Dazu sind die Schriftarten (fonts) verantwortlich. Standardisiert wird der Unicode von dem Unicode-Konsortium und der ISO (siehe auch [www.unicode.org](http://www.unicode.org)).

### **unique**

(engl. für eindeutig, einzigartig). Wichtige Voraussetzung für die Namensvergabe im Quelltext, insbesondere für Package-Namen in Java. Sind diese nicht unique, so kann es schwer aufzulösende Namenskonflikte geben.

**Unterklasse**, siehe Subklasse

### **Unterprogramm**

Teilprogramm, das mit eigenem Namen angesprochen wird (siehe auch Modul, Methode)

### **Variable**

Ein Identifier, der Daten im Arbeitsspeicher repräsentiert. Die Werte einer Variablen können zur Laufzeit des Programms geändert werden. Der Wertebereich der Variablen ist festgelegt durch den Datentyp.

### **Vererbung (inheritance)**

Ein Mechanismus, bei dem Klassen die Attribute und Methoden benutzen können von bereits bestehenden Klassen, die ähnliche, aber weniger konkrete Lösungen enthalten. Diese Technik ist die Basis der objektorientierten Programmierung. Sie erlaubt es den Subklassen, existierende Klassen als Grundlage zum Erstellen von neuen Klassen zu nehmen, um den bereits vorhandenen Code mitbenutzen zu können. Vererbung implementiert eine Beziehung, die eine Generalisierung und Spezialisierung ausdrückt. Alternative: Aggregation.

Die Subklasse kann auf Basis der bereits vorhandenen Superklasse neue Bestandteile hinzufügen oder bereits vorhandene überlagern (überschreiben).

### **Verteilte Anwendungen**

Anwendungen, die in unterschiedlichen Adressräumen laufen und miteinander kommunizieren, d.h. entweder Daten austauschen oder/und sich gegenseitig aufrufen bzw. Methoden aus dem "remoten" Adressraum aufrufen. In Java hat jede JVM einen eigenen Adressraum.

### **voller Name (qualified identifier)**

Bezeichner, der um ein Präfix ergänzt wird. Dieser Präfix kennzeichnet den Namensraum des Bezeichners, z.B. werden in Java volle Klassennamen durch das Voranstellen der Packagenamens, abgetrennt durch Punkte, gebildet ("java.util.Scanner").

**Wert (value)**

beliebige Zeichen des Unicode oder auch Zahlen. Kann sein:

- Inhalt einer Variablen,
- Ergebnis eines Ausdrucks,
- Ergebnis eines Methodenaufrufs.

**Whitespace**

Zeichen im Unicode, die in einem Texteditor nicht sichtbar sind (z.B. Leerraum, Zwischenraum). Sie dienen zur Textformatierung oder dazu, Wörter voneinander abzugrenzen. Beispiele: Space (blank, Leertaste auf Tastatur, ' \u0020' ), Tabulator (tab), Newline (nl), Wagenrücklauf (cr), Zeilenvorschub (lf).

**Wiederholung**, siehe Schleife

**Wildcard**

Wildcards ersetzen bei der Suche von Begriffen ein Zeichen, das für den oder die gesuchten Begriffe bedeutungslos ist. Andere Bezeichnung für Joker. Ein Platzhalter, der einzelne Zeichen oder ganze Zeichenfolgen ersetzt.

**Wrapper-Class**

(engl. to wrap für einpacken, einhüllen); "Hüllenklasse", gibt es für jeden einfachen Datentyp, z.B. die Klasse *Integer*, die numerische Ganzzahlen verpackt in eine Klassendefinition.

**Zeichensatz** (charset)

Zeichensätze decken einzelne Schriftkulturen und damit verbundene Sprachen oder Sprachfamilien ab. So definiert z.B. der Bytewert 252 im Zeichensatz 8859-1 den deutschen Umlaut «ü». Mehrsprachige Dokumente unterschiedlicher Kulturen lassen sich mit *einem* Zeichensatz häufig nicht darstellen. Um das Problem zu lösen, wurde der Unicode eingeführt. Die gewählte Schriftart und die Schriftgröße realisieren die Darstellung.

## Sachwortverzeichnis

---

### A

Abstraktion 206  
access control *Siehe* Zugriffsrechte  
Aggregation 303  
Algorithmus 154  
Anweisung 147  
Anweisungsblock 148  
Application 15  
Arbeitsspeicher *Siehe* RAM  
Array 313  
Arrays (mehrdimensional) 323  
ASCII-Code 31  
ASCII-Code (erweitert) 34  
Assoziation 302  
Assoziativität 146  
Attribute 253  
Ausdruck 114  
Ausnahmebehandlung *Siehe* Exception  
Aussagenlogik 135  
Auswertungsreihenfolge 115  
Autoboxing 359

### B

BCD-Code 127  
Bedingungsoperator 161  
Bezeichner 20, 44  
Big Endian 41, 103  
binär (rein binär) 35  
Binäre Darstellung 31  
Bits 27  
Bitweise Operatoren 142  
Boolean-Literal 79  
Boolean-Typ 58  
Boolesche Algebra 136  
Break-Anweisung 179

Brückenklassen 106  
Bytecode 6, 18  
Bytes 27  
byte-Typ 54

### C

Call By Value 242  
Casting 351, 356  
Characterset *Siehe* Zeichensatz  
char-Literal 81  
char-Typ 59  
Codepage *Siehe* Zeichensatz  
Codepoint 37, 104  
Collection 310  
Commandline-Parameter 225, 339  
Compilation Unit *Siehe*  
    Umwandlungseinheit  
Compile-Time-Error *Siehe*  
    Umwandlungsfehler  
Compilieren *Siehe* Umwandeln  
concatenation *Siehe* Verkettung  
Console, einlesen von 89  
Continue-Anweisung 179  
control statement *Siehe* Steueranweisung

### D

dangling else 160  
Dateiverarbeitung 102  
Datentyp 45  
Datentyp, primitiver 47  
Deklaration 44  
Dekrement 120  
Delimiter 97  
Design Pattern 309  
Do-Statement 171  
double-Typ 57

---

**E**

Editor 4  
Eingabeaufforderung 7  
encapsulation *Siehe* Kapselung  
Encoding 105, 106  
Entscheidungstabellen 214  
Entwurfsmuster *Siehe* Design Pattern  
Entwurfssprachen 208  
*enum*-Typ 166, 289  
Escapesequenz 82  
Exception 59, 97  
Expression 113  
extends 276, 278

---

**F**

Fehlerbehandlung 96  
Felder (einer Klasse) 254  
final 74  
Floating-Point *Siehe* Gleitkommazahlen  
float-Typ 57  
For-Each-Schleife 177  
For-Statement 173  
Framework 312

---

**G**

Ganzzahlen 54  
Garbage Collector 18  
Geheimnisprinzip 196, 400  
Geschachtelt (If-Anweisung) 159  
Gleitkomma-Literal 80  
Gleitkommazahlen 56  
Gültigkeit *Siehe* Scope

---

**H**

Hexadezimal 31

---

**I**

Identifizier *Siehe* Bezeichner  
if-Anweisung 155  
immutable 337  
implements 287  
import 52, 229  
Index (Array) 313  
Information Hiding 196, 297  
inheritance *Siehe* Vererbung  
Initialisierung 65  
Inkrement 120  
Instanz 255  
Instanziiieren 69  
Instanzvariable 237  
Integertypen 54  
Interface 286  
int-Typ 54  
ISO-8859 39  
Iteration *Siehe* Schleife

---

**J**

javadoc 189, 306

---

**K**

Kapselung 296  
Keyword *Siehe* Schlüsselwort  
Klasse 252  
Klasse Arrays 322  
Klasse BigInteger 125  
Klasse Formatter 100  
Klasse InputStreamReader 110  
Klasse Integer 227  
Klasse OutputStreamWriter 110  
Klasse Point 70  
Klasse Properties 230  
Klasse Scanner 95, 341  
Klassentyp *Siehe* Referenztyp  
Klassenvariable 237  
Kommentar 10, 306



Komplement 144  
Konstante 74  
Konstante, eingebaute 75  
Konstruktor 267, 269, 283  
Konvertierung 350  
Kurzschlussauswertung 141

---

## **L**

Label 182  
Laufvariable 173  
Laufzeitfehler 23, 240  
Lebensdauer (von Variablen) 366, 368  
Literele 76  
Little Endian 41, 103  
logische Operatoren 134  
lokale Variable 238, 362  
long-Typ 54

---

## **M**

Member-Variable 362  
Message *Siehe* Nachricht  
Methoden 220  
Methodenaufruf 228  
Modifizier *Siehe* Zugriffsmodifizier  
Modifizier (static) 369  
Modul 295  
Modulbildung 204  
Modulo-Operator 116

---

## **N**

Nachricht 221  
Namensvergabe (Regeln) 25, 190  
Nebeneffekt 114, 122  
new 48  
null 83, 330  
numeric promotion 119, 351

---

## **O**

Objekt 49  
Objekt erzeugen 69  
Objektvariablen 369  
Operand 114  
Operator 114  
Overload *Siehe* Überladen  
Override *Siehe* Überschreiben

---

## **P**

package 372  
Parameter 240  
Parameter (variable Anzahl) 248  
Pattern *Siehe* Design Pattern  
Postfix 120  
Präzedenz 145  
Prefix 120  
primitive Datentypen 46  
primitive Variablen 85  
print/println-Methode 99  
Priorität (der Auswertung) 145  
Programmablaufplan (PAP) 209  
Programmieren im Großen 198  
Programmieren im Kleinen 193  
Programmierstil 188  
Prototyping 203  
Pseudocode 213

---

## **Q**

qualifizierter Name 370  
Quelltext 13, 17

---

## **R**

RAM Random Access Memory 63  
Rechnen, Probleme beim 122  
Referenztyp 47  
Referenzvariablen 68

Reguläre Ausdrücke 343  
 Reihung *Siehe* Array  
 relationale Operatoren 131  
 return 184  
 return-value *Siehe* Rückgabewert  
 Rückgabewert 248  
 Runden (bei Arithmetik) 128  
 Runtime-Error *Siehe* Laufzeitfehler

## S

Schachteln (von Methodenaufrufen) 229  
 Schachtelung (von Schleifen) 178  
 Schleife (Loop) 167  
 Schlüsselwort 19  
 Scope 364  
 Selektion *Siehe* Verzweigung  
 Short-Circuit-Evaluation *Siehe*  
     Kurzschlussauswertung  
 short-Typ 54  
 Sichtbarkeit *Siehe* Scope  
 Signatur 223  
 Sourcecode *Siehe* Quelltext  
 Spiralmodell 201  
 Sprung-Anweisung 179  
 Standard-In 94  
 Standard-Out 98  
 Statement *Siehe* Anweisung  
 static import 227  
 Static-Elemente 284  
 Steueranweisung 149, 154  
 Stream, byteorientiert 90  
 Stream, characterorientiert 90  
 Stream-Konzept 87  
 String-Klasse 69  
 String-Literal 83  
 Strings 329  
 Strings vergleichen 334  
 Struktogramm *e* 156, 210  
 Strukturierte Programmierung 196  
 super 279  
 Switch-Anweisung 162  
 Syntax 19

## T

TCP/IP, lesen von 92  
 Testen 24, 191  
 Text zerlegen 341  
 this 269, 279  
 throws Exception 59, 240  
 Try-Catch *Siehe* Fehlerbehandlung  
 Typanpassung 350  
 Typanpassung (bei Parametern) 352  
 Typanpassung (bei Referenztypen) 355  
 Typkonvertierung 333  
 Typkonzept (Vorteile) *e* 61

## U

Überladen 267  
 Überladen (von Konstruktoren) 271  
 Überladen (von Methoden) 239  
 Überlauf-Probleme 124  
 Überschreiben (von Methoden) 268  
 UML 278, 307  
 Umwandeln 6  
 Umwandlungseinheit 14  
 Umwandlungsfehler 22  
 Unicode 36  
 UTF 103  
 UTF-16 38  
 UTF-8 38, 111

## V

Variable Parameterliste 248  
 Variablen 63  
 Vererbung 274, 301  
 Vergleich (von Objekten) 133  
 Vergleichsoperatoren 131  
 Verkettung (von Strings) 332  
 Verzweigung 155  
 void 221

---

## **W**

Wasserfallmodell 201  
Wertebereich 46  
Wertezuweisung 66, 150  
while-Statement 167  
Wrapper-Klasse 60, 358

---

## **Z**

Zählschleife 173  
Zeichenketten *Siehe* Strings  
Zeichensatz 34, 106  
Zeichentyp *Siehe* char-Typ  
Zugriffsmodifizier 375  
Zugriffsrechte 370  
Zuweisungskompatibilität 66  
Zweierkomplement 126