PROF. BALZERT
STIFTUNG
Helmut Balzert
5. Auflage
Java: Der Einstieg
in die Programmierung
Strukturiert & prozedural programmieren
Mit Einführung in C und Processing

Wissensgebiete Informatik | Programmierung | Java
Zielgruppen Alle, die einen systematischen & fundierten Einstieg in die Programmierung suchen | Studierende im 1. Semster Informatik (Haupt- & Nebenfach) | Quereinsteiger in die Informatik | Fachinformatiker | Schüler
Voraussetzungen Bedienung eines PCs mit Windows oder Linux

Charakteristika dieses Buches

- Mehr als ein Java-Buch - Eine grundlegende Einführung in die Programmierung.
- Sorgfältig durchdachte Didaktik, die das Lernen erleichtert.
- Der 1. Schritt auf dem Weg zum Junior-Programmierer.
- Vermittlung der Grundlagen und Konzepte, die fast allen Programmiersprachen zugrunde liegen.
- Die Konzepte werden Schritt für Schritt aufeinander aufbauend erklärt.
- Neben Wissen und Kenntnissen werden Fähigkeiten erworben, selbst Programme zu entwickeln und
- zu testen.
- Besonderer Wert wurde auf vollständige Beispiele gelegt.
- Eine Fallstudie OptiTravel wird schrittweise entwickelt und zeigt, wie eine systematische Softwareentwicklung abläuft.
- Themenschwerpunkte: Basiskonzepte, Kontroll-strukturen, Felder, Methoden, Testen, Verifikation.
- Neueste Java-Version und Nutzung der UML 2 (Unified Modeling Language).
- Einsatz der Entwicklungsumgebung BlueJ.
- Einführung in die Sprache C.
- Einführung in die Sprache Processing
- 4 Kapitel »Vom Problem zur Lösung«
- 3 Kreuzworträtsel
- Merkeboxen für andere Perspektiven auf Themen
- 187 Abbildungen, 95 Glossarbegriffe, 100 Programme

Univ.-Prof. Dr. -Ing. habil.
Helmut Balzert
Softwaretechnik
Ruhr-Universität Bochum

»Didaktik ist unsere Stärke«

PROF. BALZERT
STIFTUNG

Inhaltsübersicht

1 Aufbau und Gliederung

2 Der Schnelleinstieg

3 Einfache Txpen, ihre Werte und Operationen

4 Kontrollstrukturen

5 Felder

6 Prozeduren, Funktionen und Methoden

7 Das Wichtigste zum Testen

8 Die Grundideen der Verifikation

9 Die Programmiersprache C

10 Die Programmiersprache »Processing«

Java: Der Einstieg in die Programmierung

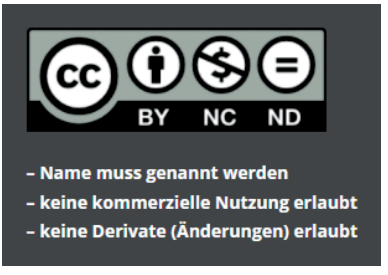
PROF. BALZERT STIFTUNG

Die gemeinnützige **Prof. Balzert- Stiftung** wurde von Prof. Dr. Helmut Balzert und Prof. Dr. Heide Balzert im Jahr 2021 g egründet, um Wissenschaft und Kultur zu fördern.

Im Bereich der Wissenschaft vergibt die Stiftung zusammen mit der Gesellschaft für Informatik e.V. jedes Jahr einen Preis in Höhe von 10.000 € für einen herausragenden Beitrag zur digitalen Didaktik in der Informatik.

Um die Verbreitung von didaktisch gelungenen Lehrbüchern zu fördern, stellt die Stiftung u.a. über die Digital Library der Gesellschaft für Informatik e.V. (www.dl.gi.de) Lehrbücher zum kostenlosen Herunterladen unter der Creative Commons Lizenz mit folgenden Einschränkungen (by-nc-nd, siehe Abb.) zur Verfügung:

1. Das Nutzungsrecht bleibt bei der Prof. Balzert-Stiftung.
2. Die Namen der ursprünglichen Urheber müssen genannt werden.
3. Die Buchtexte dürfen nicht kommerziell genutzt werden.
4. Die Buchtexte dürfen nicht geändert werden.



„Dieses Werk ist unter einer Creative Commons Lizenz vom Typ Namensnennung - Nicht kommerziell - Keine Bearbeitungen 4.0 International zugänglich. Um eine Kopie dieser Lizenz einzusehen, konsultieren Sie http://creativecommons.org/licenses/by-nc-nd/4.0/ oder wenden Sie sich brieflich an Creative Commons, Postfach 1866, Mountain View, California, 94042, USA.“

Wenn Sie selbst ein didaktisch gutes Lehrbuch unter der angegebenen Creative Commons Lizenz über die Prof. Balzert-Stiftung veröffentlichen wollen, schreiben Sie bitte an stiftung@balzert.de.

Im Bereich der Kultur vergibt die Stiftung außerdem zusammen mit den Ballettfreunden Dortmund e.V. jedes Jahr einen Preis in Höhe von 10.000 € für analog-digitale Ballettchoreografie an das Ballett des Theaters Dortmund.

Helmut Balzert

Java: Der Einstieg in die Programmierung

Strukturiert & prozedural programmieren

Mit einer Einführung in die Sprachen C und Processing

5. Auflage

Unter Mitwirkung von
Ulrich Breymann

Prof. Balzert-Stiftung | Dortmund

Autoren:
Prof. Dr. Helmut Balzert
E-Mail: stiftung@balzert.de
Prof. Dr. Ulrich Breymann
E-Mail: breymann@hs-bremen.de

Der Verlag und der Autor haben alle Sorgfalt walten lassen, um vollständige und akkurate Informationen in diesem Buch und den Programmen zu publizieren. Der Verlag übernimmt weder Garantie noch die juristische Verantwortung oder irgendeine Haftung für die Nutzung dieser Informationen, für deren Wirtschaftlichkeit oder fehlerfreie Funktion für einen bestimmten Zweck. Ferner kann der Verlag für Schäden, die auf einer Fehlfunktion von Programmen oder Ähnliches zurückzuführensind, nicht haftbar gemacht werden. Auch nicht für die Verletzung von Patentund anderen Rechten Dritter, die daraus resultieren. Eine telefonische oder schriftliche Beratung durch den Verlag über den Einsatz der Programme ist nicht möglich. Der Verlag übernimmt keine Gewähr dafür, dass die beschriebenen Verfahren, Programme usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen,Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Bildnachweis: Umschlaggrafik: »Abstract colorful background Copyright Helen Stock, 2013, mit Genehmigung von Shutterstock.com«

© 2022 Prof. Balzert-Stiftung | Dortmund | DOI 10.18420/LB-JavaEinstieg

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt.

1. Auflage: Juni 2005
2. Auflage: August 2008
1. korrigierter Nachdruck: Juni 2010
3. Auflage: Oktober 2010
4. Auflage: Oktober 2013
5. Auflage: Juli 2022

Für dieses Buch gibt es alle Programme zum Herunterladen, die Sie unter der `DOI 10.18420/LB-JavaEinstieg` abrufen können.

Für dieses Buch gilt die Creative Common Lizenz vom Typ Namensnennung, nicht kommerziell, keine Bearbeitungen.

Gesamtgestaltung: Prof. Dr. Heide Balzert, Dortmund

Satz: Der Satz erfolgte aus der Lucida, Lucida sans und Lucida casual.

Vorwort zur 5. Auflage

Faszination pur

Programmieren ist eine faszinierende Tätigkeit. Einem Computersystem vorzuschreiben, was es tun soll und anschließend zu sehen, dass es genau das tut, was man sich vorgestellt hat, ist ein tolles Gefühl!

Zur Umschlaggrafik

Die Grafik auf der Umschlagseite habe ich ausgewählt, weil für mich die bunten Rechtecke die Bausteine eines Programms symbolisieren und die gesamte Grafik die Dynamik eines Programmablaufs »veranschaulicht«.

Bei aller Begeisterung für das Programmieren darf jedoch nicht übersehen werden, dass Programmieren eine äußerst anstrengende Tätigkeit ist, die viel **Disziplin, Sorgfalt, Ausdauer** und **Systematik** erfordert. Ohne diese Fähigkeiten kommt man über kleine »Spiel-Programme« für die eigene Benutzung *nicht* hinaus.

Programmieren ist eine **konstruktive Tätigkeit**. Bevor man die Konstruktion jedoch beherrscht, muss man die jeweiligen Programmierkonzepte gut verstanden haben, sonst artet die Programmierung in »Versuch und Irrtum« aus.

Konzepte der Programmierung

Ziel dieses Buches ist es daher, Ihnen die **grundlegenden Konzepte der Programmierung** zu vermitteln. Diese grundlegenden Konzepte gelten für fast alle Programmiersprachen.

Theorie vs. Praxis

Konzepte kann man theoretisch verstehen. Richtig begreifen und verinnerlichen kann man sie jedoch nur durch die **praktische Erfahrung**. Für die Programmierung bedeutet dies, dass die Konzepte in Programmen einer konkreten Programmiersprache angewandt und am Computersystem ausprobiert werden.

Java

Als konkrete Programmiersprache wird in diesem Buch die am meisten verbreitete objektorientierte Programmiersprache Java in ihrer neuesten Version vorgestellt und eingesetzt.

C

Gerade ein Programmieranfänger neigt dazu, die Syntax und Semantik der ersten erlernten Programmiersprache als den »Standard« zu verinnerlichen. Daher habe ich dieses Buches um eine Einführung in die Sprache C ergänzt, damit Sie als Einsteiger in die Programmierung einen erweiterten Horizont bekommen. Ich danke meinem Kollegen Prof. Dr. Ulrich Breymann von der Hochschule Bremen, dass er bereit war, die Einführung in C zu konzipieren und zu schreiben.

Processing

Zusätzlich gibt es eine Einführung in die Programmiersprache Processing. Durch diese Sprache erhalten Sie einen Einstieg in die zweidimensionale Grafikprogram-mierung, in die Programmierung von Animationen und in Maus- und Tastaturereignisse. Diese Kenntnisse erleichtern den späteren Einstieg in die objektorientierte Programmierung (siehe

mein Buch »Java: Objektorientiert programmieren«) und in die Programmierung von grafischen Benutzungsoberflächen (siehe mein Buch »Java: Anwendungen programmieren«).

Vom Problem zur Lösung

Beim Programmieren geht es immer darum, zu einem gegebenen Problem eine Lösung zu finden. Programmieren kann man daher gleichsetzen mit **Problemlösen**. Um Ihnen als Anfänger den Weg vom Problem zur Lösung zu erleichtern, habe ich in dieses Buches vier Kapitel eingefügt, die Hinweise zum systematischen Problemlösen anhand von Beispielen geben. Auf der hinteren Buchinnenseite finden Sie eine Zusammenfassung »In 10 Schritten vom Problem zur Lösung«.

Voraussetzungen

Als Einführungsbuch in die Programmierung werden fast keine Voraussetzungen verlangt. Alle wichtigen Begriffe und grundlegendes Wissen werden behandelt. Sie sollten jedoch einen PC mit dem Betriebssystem Windows oder dem Betriebssystem Linux in den Grundzügen bedienen können und einen Zugriff aufs Internet besitzen. .

Neue Didaktik

Um Ihnen als Leser das Lernen zu erleichtern, wurde für die Lehrbücher der Prof. Balzert-Stiftung eine neue Didaktik entwickelt. Der Buchaufbau und die didaktischen Elemente sind im Anschluss an dieses Vorwort beschrieben (Hinweise der Stiftung).

Alle Programme, die im Buch behandelt werden, können Sie auf Ihr Computersystem herunterladen, um die Programme auszuprobieren und weiter zu entwickeln (DOI 10.18420/LB-WissArbeiten).

Dank

Mein besonderer Dank gilt allen Studierenden in meinen Vorlesungen und Online-Kursen, die durch Hinweise und Kommentare dazu beigetragen haben, dieses Buch schrittweise zu verbessern. Mein besonderer Dank gilt meinem ehemaligem wissenschaftlichen Mitarbeiter Dr. Michael Goll. Anja Schartl hat mit viel Geduld die zahlreichen Grafiken gezeichnet. Danke dafür.

Damit Sie prüfen können, ob Sie wichtige Begriffe der Programmierung kennen, finden Sie im Buch mehrere **Kreuzworträtsel** um Ihr Wissen zu überprüfen. Die Lösungen finden Sie im Buchanhang.

Hinweise zur Didaktik

Über 25 Jahre Erfahrung beim Unterrichten von klassischen Programmiersprachen haben mir gezeigt, dass es verschiedene Komplexitätsstufen beim Verständnis vom Programmierkonzepten gibt. Diese Stufen lassen sich wie folgt benennen:

- Strukturiertes Programmieren
- Prozedurales Programmieren
- Objektorientiertes Programmieren
- Programmierung von grafischen Benutzungsoberflächen (GUIs)

Die **Komplexität** nimmt von Stufe zu Stufe zu. Ein Programmieranfänger hat daher in der Regel Probleme, wenn direkt in das objektorientierte Programmieren eingestiegen wird. Ich habe dies mehrere Jahre hinweg ausprobiert und festgestellt, dass ohne fundiertes prozedurales Programmieren beim objektorientierten Programmieren für den Anfänger vieles »im Nebel« und Ungewissen bleibt. Insbesondere sind fundierte Kenntnisse über den Parametermechanismus erforderlich, um das Methodenkonzept der objektorientierten Programmierung zu verstehen. Ähnlich verhält es sich mit dem Ereignisverarbeitungskonzept von grafischen Benutzungsoberflächen.

Aufgrund dieser Erfahrungen habe ich mich dazu entschlossen, die Komplexitätsstufen schrittweise zu vermitteln, beginnend beim strukturierten Programmieren. In diesem Buch finden Sie daher die Programmierkonzepte zum strukturierten und prozeduralen Programmieren. Die objektorientierten Konzepte werden in dem Buch »Java: Objektorientiert Programmieren« vermittelt, die Programmierung von grafischen Benutzungsoberflächen in dem Buch »Java: Anwendungen programmieren«.

Ans Werk

Starten Sie jetzt mit Ihrem Einstieg in die faszinierende Welt der Programmierung – das erste Kapitel liegt vor Ihnen. Viel Spaß und Erfolg!

Ihr

Entstehungsgeschichte des Buches

Dieses Buch wurde für die W3L-Akademie (W3L = Web Life Long Learning) der Firma W3L AG in Dortmund entwickelt.

Im Jahr 2016 wurde die W3L-Akademie von der Springer Nature Campus GmbH übernommen.

Die urheberrechtlichen Nutzungsrechte wurden freundlicherweise von der Springer Nature Campus GmbH an die Autoren zurückgegeben.

Die Autoren wiederum haben der Prof. Balzert-Stiftung Nutzungsrechte für die Bereitstellung der Bücher unter der Creative Commons Lizenz übertragen.

Hinweise der Prof. Balzert-Stiftung

Zum Aufbau des Buches

Dieses Buch besteht aus **Kapiteln** und **Unterkapiteln**. Jedes Unterkapitel ist im **Zeitungsstil** geschrieben. Am Anfang steht die Essenz, d. h. das Wesentliche. Es kann Ihnen zur Orientierung dienen – aber auch zur Wiederholung. Anschließend kommen die Details. Die **Essenz** ist grau hervorgehoben.

Sternesystem

Jedes Kapitel und Unterkapitel ist nach einem **Sternesystem** gekennzeichnet:
* = Grundlagenwissen
** = Vertiefungswissen
*** = Spezialwissen
**** = Expertenwissen
Dieses Sternesystem hilft Ihnen, sich am Anfang auf die wesentlichen Inhalte zu konzentrieren (1 und 2 Sterne) und sich vielleicht erst später mit speziellen Themen (3 und 4 Sterne) zu befassen.

Übungen ermöglichen eine Selbstkontrolle und Vertiefung des Stoffes. Sie sind durch ein Piktogramm in der Marginalspalte gekennzeichnet. Tests einschließlich automatischer Korrekturen finden Sie in dem zugehörigen (kostenpflichtigen) E-Learning-Zertifikatskurs.

Beispiel

Beispiele helfen Sachverhalte zu verdeutlichen. Sie sind in der Marginalspalte mit »Beispiel« gekennzeichnet. Der Beispieltext ist mit einem Grauraster unterlegt.

Tipps/ Hinweise

Hilfreiche **Tipps**, **Empfehlungen** und **Hinweise** sind durch eine graue Linie vom restlichen Text getrennt.

Glossar

Glossarbegriffe sind fett gesetzt, **wichtige Begriffe grau** hervorgehoben. Ein vollständiges Glossarverzeichnis finden Sie am Buchende.

Dieses Piktogramm zeigt an, dass wichtige Inhalte nochmals in einer so genannten Merkebox zusammengefasst wiederholt werden – oft unter einer anderen Perspektive, um den Lerneffekt zu erhöhen.

Frage & Antwort

In den meisten Lehrbüchern wird »die Welt« so erklärt, wie sie ist – ohne dem Leser vorher die Möglichkeit gegeben zu haben, über »die Welt« nachzudenken. In einigen Kapiteln werden Ihnen Fragen gestellt. Diese Fragen sollen Sie dazu anregen, über ein Thema nachzudenken. Erst nach dem Nachdenken sollten Sie

weiter lesen. (Vielleicht sollten Sie die Antwort nach der Frage zunächst durch ein Papier abdecken).

Englische Begriffe *kursiv*

Für viele Begriffe – insbesondere in Spezialgebieten – gibt es keine oder noch keine geeigneten oder üblichen deutschen Begriffe. Gibt es noch keinen eingebürgerten deutschen Begriff, dann wird der englische Originalbegriff verwendet. Englische Bezeichnungen sind immer *kursiv* gesetzt, sodass sie sofort ins Auge fallen.

Querverweise

Damit Sie referenzierte Seiten schnell finden, enthalten alle Querverweise absolute Seitenzahlen.

Dieses Piktogramm »Unter der Lupe« weist darauf hin, dass jetzt ein Sachverhalt für den interessierten Leser detailliert vorgestellt wird.

Syntaxnotation für Java

Neben Syntaxdiagrammen wird die Java-Syntax in folgender Notation dargestellt (alle Elemente, die zur Beschreibung der Syntax dienen, sind *kursiv* dargestellt):

- *A ::= B* : Das zu definierende nicht-terminale Symbol *A* (*Platzhalter*) steht auf der linken Seite, durch *::=* von seiner Definition *B* auf der rechten Seite getrennt.
- *[ ]* : Eckige Klammern *[ ]* schließen optionale Elemente ein, d. h. Elemente, die auch fehlen dürfen.
- *|* : Ein kursiver senkrechter Strich *|* trennt alternative Elemente, d. h. von den aufgeführten Elementen ist ein Element auszuwählen.
- *{}* : Aus den aufgeführten Elementen ist ein Element auszuwählen.
- *+* : Ein *+* gibt an, dass das Element wiederholt werden kann.
- *...* : Drei Punkte kennzeichnen eine Liste von Elementen, wobei die Elemente durch ein Komma (,) getrennt werden.
- `terminale Symbole`: Die Zeichen, die im Quellcode des Programms stehen müssen, sind in normaler Schrift dargestellt.
- `Schlüsselwörter`: Schlüsselwörter sind mit einem Grauraster unterlegt oder fett dargestellt.
- *Syntaxsymbole*: Symbole, die zur Beschreibung der Syntax verwendet werden, sind kursiv dargestellt, z. B. *{ }*, sonst handelt es sich um terminale Symbole.

Viel Freude beim Lesen und viel Erfolg bei Ihrem Einstieg in die Welt der Programmierung wünscht Ihnen

die Prof. Balzert-Stiftung

Inhalt

1	**Aufbau und Gliederung** *	1
2	**Der Schnelleinstieg** *	7
2.1	Programmieren – Programme – Compiler *	8
2.2	Skriptsprachen, Zwischensprachen und ihre Interpreter *.	13
2.3	Die Programmiersprache Java *	17
2.4	Das erste Java-Programm *	18
2.4.1	»Hello World« mit Java *	19
2.4.2	Zum Aufbau eines Java-Programms *	24
2.5	Grundlegende Konzepte der Programmierung: das Wichtigste *	27
2.5.1	Variablen, Konstanten und Typen *	28
2.5.2	Zuweisung und Ausdrücke *	32
2.5.3	Java-Programm mit lokalen Variablen und einfachen Anweisungen *	36
2.5.4	Java-Programme mit Konsoleneingabe *	39
2.5.5	Java-Pakete anlegen und benutzen: das Wichtigste *	43
2.6	Java-Entwicklungsumgebungen *	50
2.7	OptiTravel: Gespräch Auftraggeber – Auftragnehmer *	51
3	**Einfache Typen, ihre Werte und Operationen** *	55
3.1	Java: Syntaxnotation *	56
3.2	Der Typ `boolean` *	61
3.3	Ganzzahlige Typen *	64
3.4	Gleitpunkt-Typen *	70
3.5	Darstellung von Gleitpunkt-Zahlen **	73
3.6	Rechengenauigkeit mit Gleitpunkt-Zahlen **	77
3.7	Eingeschränkte Mathematikgesetze ***	81
3.8	Der Zeichentyp `char` *	85
3.9	Operatorprioritäten *	89
3.10	Typumwandlungen *	91
3.11	Vom Problem zur Lösung: Teil 1 **	95
3.12	Box: Kreuzworträtsel 1 **	99
4	**Kontrollstrukturen** *	101
4.1	Die Sequenz *	106
4.2	Die ein- und zweiseitige Auswahl *	109
4.3	Die Mehrfachauswahl *	117
4.4	Bedingte Wiederholung und n + 1/2-Schleife *	122
4.5	Die Zählschleife und die Endlosschleife *	131
4.6	Termination von Schleifen *	134
4.7	Der Aufruf *	137
4.8	Geschachtelte Kontrollstrukturen *	140
4.9	OptiTravel: Zeitvergleich *	145
4.10	OptiTravel: Funktionsauswahl *	147
4.11	Anordnung von Auswahlanweisungen *	149
4.12	Auswahl von Kontrollstrukturen *	152
4.13	Strukturierte Programmierung ***	153
4.14	Behandlung von Ausnahmen *	157

4.15	Zusicherungen in Java **	162
4.16	Vom Problem zur Lösung: Teil 2 **	164
4.17	Box: Kreuzworträtsel 2 **	170
5	**Felder ***	173
5.1	Eindimensionale Felder *	174
5.2	OptiTravel: Balkendiagramm *	179
5.3	Mehrdimensionale Felder *	180
5.4	Sonderformen von Feldern **	187
5.5	OptiTravel: Tabellen *	189
5.6	Einfaches Sortieren *	195
5.7	Iteration über Felder: Die erweiterte `for`-Schleife **	198
5.8	Aufzählungen mit `enum` ***	200
5.9	Vom Problem zur Lösung: Teil 3 **	202
6	**Prozeduren, Funktionen und Methoden ***	207
6.1	Parameterlose Prozeduren *	208
6.2	Prozeduren mit Eingabeparametern *	213
6.3	Felder als Eingabeparameter *	219
6.4	Funktionen und Ausgabeparameter *	223
6.5	Java-Funktionen nutzen *	225
6.6	Felder als Ergebnisparameter *	230
6.7	Variable Parameterlisten ***	232
6.8	Überladen von Methoden **	233
6.9	UML-Sequenzdiagramme **	236
6.10	Rekursion *	239
6.11	Rekursion: Türme von Hanoi ***	244
6.12	Rekursion: direkt vs. indirekt **	250
6.13	Datenabstraktion: Gemeinsame Daten *	252
6.14	OptiTravel: Gesamtlösung *	258
6.15	Vom Problem zur Lösung: Teil 4 **	267
7	**Das Wichtigste zum Testen ***	279
7.1	Einfaches Testen *	279
7.2	Regressionstest *	281
7.3	Stapelverarbeitungsprogramme: `.bat`-Dateien **	287
7.4	Zur Auswahl von Testdaten **	292
8	**Die Grundideen der Verifikation ***	297
8.1	Intuitive Einführung ***	297
8.2	Zusicherungen ***	301
8.3	Spezifizieren mit Anfangs- und Endebedingung ***	303
8.4	Verifikationsregeln ****	306
8.5	Termination von Schleifen ****	313
8.6	Entwickeln von Schleifen ****	315
8.7	Vor- und Nachteile ***	320
8.8	Box: Kreuzworträtsel 3 **	320
9	**Die Programmiersprache C ***	323
9.1	»Hello World« in C *	324
9.2	Einfache Datentypen *	326
9.3	Einfache Ein- und Ausgabe *	328

9.4 Kontrollstrukturen und Zusicherungen * 331
9.5 Zeiger und Adressen * 334
9.6 Felder * 337
9.7 C-Zeichenketten ** 339
9.8 Strukturen ** 340
9.9 Dynamische Daten ** 342
9.10 Modularität * 343

10 Die Programmiersprache »Processing« * 349
10.1 Einstieg in die zweidimensionale Grafikprogrammierung * 350
10.2 Animationen * 355
10.3 Maus- und Tastaturereignisse * 359
10.4 Ausblick *** 364

Anhang A Kreuzworträtsel 1: Lösung ** 369

Anhang B Kreuzworträtsel 2: Lösung ** 371

Anhang C Kreuzworträtsel 3: Lösung ** 373

Glossar 375

Literatur 385

Sachindex 387

Weitere Bücher der Prof. Balzert-Stiftung unter der Creative Commons Lizenz

Helmut Balzert, Marion Schröder, Christian Schäfer
Wissenschaftliches Arbeiten

Marion Schröder
Heureka, ich hab's gefunden!

Helmut Balzert
Wie schreibt man...
erfolgreiche Lehrbücher und
E-Learning-Kurse?

Klaus Mentzel
BWL für Manager

Helmut Balzert
Java: Objektorientiert programmieren

Heide Balzert
UML 2 in 5 Tagen

1 Aufbau und Gliederung *

Ohne Vorkenntnisse

Eine Programmiersprache können Sie auf verschiedene Arten lernen. Wie leicht Sie das Programmieren lernen, hängt auch davon ab, welche Vorkenntnisse Sie bereits besitzen. Dieses Buch geht davon aus, dass Sie noch *keine* oder nur geringe Vorkenntnisse besitzen und Anfänger in der Programmierung und der Programmiersprache Java sind.

Schrittweise, aufeinander aufbauend

Um Ihnen den Einstieg in die Welt der Programmierung und der Programmiersprache Java so einfach wie möglich zu machen, werden schrittweise und aufeinander aufbauend die **grundlegenden Konzepte der Programmierung** erklärt und veranschaulicht. Die Abb. 1.0-1 zeigt, dass zunächst die **Basiskonzepte** behandelt werden, die jeder Programmiersprache zugrunde liegen.

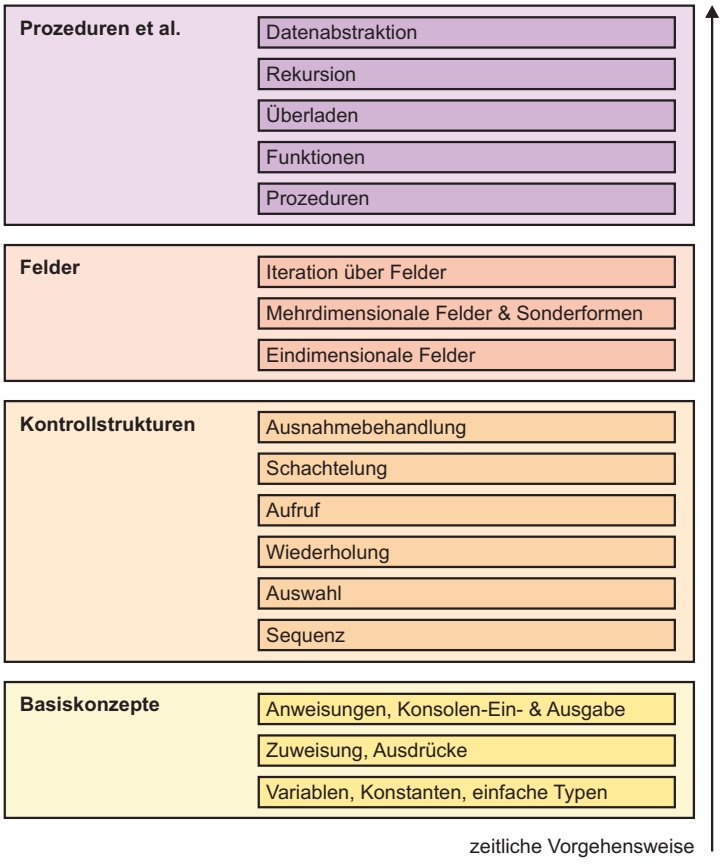


Abb. 1.0-1: Die Basiskonzepte der Programmierung.

Anschließend wird auf die sogenannten **Kontrollstrukturen** – auch Steueranweisungen genannt – eingegangen. Im dritten Block werden **Felder** – auch Reihungen genannt – eingeführt. Den Abschluss bilden die Konzepte der **Prozeduren** und **Funktionen** – auch Methoden genannt.

Ziel

Die hier vermittelten Konzepte sind in den meisten der heutigen Programmiersprachen zu finden. Nach dem Durcharbeiten dieses Buches können Sie einfache Programme nicht nur in der Programmiersprache Java erstellen, sondern – nach einer kurzen Einarbeitung – auch in anderen Programmiersprachen.

Programme werden in der Regel in Textform dargestellt. Zur Veranschaulichung und zur Modellierung eignen sich aber auch grafische Notationen. Industriestandard ist inzwischen die grafische Modellierungssprache **UML** *(Unified Modeling Language)*, die in diesem Buch an vielen Stellen eingeführt und verwendet wird.

Praktische Arbeit mit dem Computer nötig *learning by doing*

Programmieren können Sie *nicht* allein theoretisch erlernen. Sie müssen immer an Ihrem Computersystem überprüfen, ob Ihr Programm das tut, was Sie sich gedacht haben. Dazu ist es erforderlich, dass Sie die Arbeit mit Ihrem Computersystem und den notwendigen Hilfsmitteln für die Programmierung beherrschen.

Hinweis

Noch ein Hinweis vorneweg: Programmieren lernen ist *nicht* einfach, sondern eher schwer. Wissen und Verstehen allein reichen *nicht* aus, sondern Programmieren ist eine konstruktive Tätigkeit. Vergleicht man diese Tätigkeit mit dem Bauingenieurwesen, dann handelt sich um die Konstruktion und das Errichten von Bauwerken. Verglichen mit dem Bauingenieurwesen werden in diesem Buch zunächst eine Hundehütte konzipiert und realisiert und am Ende vielleicht ein Gartenhaus. Für ein Einfamilien- oder Mehrfamilienhaus oder gar ein Hochhaus reichen die Kenntnisse in diesem Buch noch *nicht* aus.

Schnelleinstieg

Am Anfang dieses Buches steht ein sogenannter Schnelleinstieg, der Sie mit den wichtigsten Rahmenbedingungen vertraut macht, bevor Sie mit dem Programmieren beginnen können. Dazu gehört ein Überblick über Programme und ihre Übersetzer (Compiler), über Skriptsprachen und ihre Interpreter sowie über die Programmiersprache Java. Damit Sie Ihre Programme auf Ihrem Computersystem »laufen lassen« können, müssen Sie auf Ihrem Computersystem mindestens eine Java-Entwicklungsumgebung installiert haben. Wie dies geschieht, wird in einem kostenlosen E-Learning-Kurs erklärt, der zu diesem Buch gehört. Da sich die Installationsanleitungen und Versionen der Entwicklungsumgebungen oft ändern, wurde auf einen Abdruck im Buch verzichtet. Im E-Learning-Kurs finden Sie immer die aktuellen Informa-

tionen. Am Ende des Schnelleinstiegs haben Sie Ihr erstes Programm geschrieben und auf Ihrem Computersystem zum Laufen gebracht.

Qualität Ihrer Programme

Sie haben sich sicher schon oft geärgert, wenn Programme, die Sie benutzen, nicht das tun, was sie erwartet haben, oder zu Fehlern führen oder einfach »abstürzen«. Wenn Sie nun selbst Programme schreiben, werden Sie feststellen, dass es gar nicht so einfach ist, fehlerfreie Programme zu entwickeln. Daher werden parallel zu den Programmierkonzepten Konzepte vermittelt, um Fehler von vornherein zu vermeiden und gemachte Fehler schnell zu finden. Neben der Erstellung gut lesbarer und kommentierter Programme gehören dazu das systematische Testen von Programmen sowie das defensive Programmieren. Ein Ausblick vermittelt ein Gefühl für die sogenannte Verifikation von Programmen, die es ermöglicht, die Korrektheit von Programmen mathematisch zu beweisen.

learning by example

Beispiele erleichtern das Lernen. Daher finden Sie in diesem Buch in der Regel vollständige, ablauffähige Programme, die Sie auf Ihr Computersystem herunterladen und ausprobieren können. Fangen Sie anschließend an, Änderungen an diesen Programmen vorzunehmen, um die Wirkung einzelner Konzepte selbst zu »erleben«.

learning by analogy

Der nächste Schritt kann darin bestehen, ein vorhandenes Programm als Beispiel zu nehmen und durch Analogieschluss ein verwandtes Problem zu lösen. Beispielsweise nehmen Sie ein Programm, das Celsius-Grade in Fahrenheit-Grade umrechnet, als Vorbild und schreiben ein neues Programm, das Kilometer in Meilen umrechnet.

learning by exploration

Nicht alle möglichen Programmierkonzepte und insbesondere Java-Konzepte können vorgestellt werden. Sie werden jedoch in diesem Buch in die Lage versetzt, selbst im Internet zu recherchieren und sich die notwendigen Informationen zu besorgen.

Fallstudie

In der Praxis werden heute umfangreiche Programme geschrieben. Damit Sie ein Gefühl für größere und realistische Programme erhalten, wird in diesem Buch eine Fallstudie OptiTravel präsentiert, die schrittweise zu einem umfangreichen Programm führt.

Nach dem jeweiligen Kennenlernen eines neuen Konzepts müssen Sie durch Übungen selbst feststellen, ob Sie alles richtig verstanden haben.

Umfassendes didaktisches Konzept

Dieses Buch bildet die **Basis** für eine **umfassende Ausbildung im Programmieren** und der Programmiersprache **Java**. Es werden die Basiskonzepte, das strukturierte und das prozedurale Programmieren vermittelt. Diese Konzepte sind in fast allen Programmiersprachen vorhanden, sodass Sie das hier erworbene Wissen leicht auf andere Programmier- und Skriptsprachen übertragen können.

Weitere Sprachen

Um Ihnen bereits einen Einblick in andere Programmiersprachen zu vermitteln, finden Sie am Ende des Buches Einführungen in zwei weitere Programmiersprachen.

Sprache C

Eine Einführung stellt die auch heute noch weit verbreitete Programmiersprache C vor. Diese Programmiersprache wird insbesondere für hardwarenahe Programmieraufgaben eingesetzt und bildet die Grundlage für die objektorientierte Programmiersprache C++.

Sprache Processing

Die junge Sprache Processing wurde 2005 entwickelt und basiert auf Java. Sie erlaubt es auf einfache Art und Weise Grafiken und Animationen zu programmieren sowie Maus- und Tastaturereignisse abzufragen. Diese Kenntnisse erleichtern Ihnen den späteren Einstieg in die objektorientierte Programmierung und in die Programmierung von grafischen Benutzungsoberflächen.

Objekt-orientiert programmieren

Aufbauend auf diesen Basiskonzepten werden in dem Buch »Java: Objektorientiert programmieren – Vom objektorientierten Analysemodell bis zum objektorientierten Programm« die objektorientierten Konzepte vermittelt. Diese Konzepte bieten neue Möglichkeiten bei der Programmierung, erhöhen aber auch die Komplexität.

SQL & RDBS

Für die langfristige Speicherung umfangreicher Daten werden heute relationale Datenbanksysteme (RDBS) eingesetzt, die mit der deklarativen Programmiersprache SQL programmiert werden. In den folgenden Büchern wird dieses Wissen vermittelt:

- SQL: Der Einstieg in die deklarative Programmierung
- Datenbankanwendungen entwerfen & programmieren

Nebenläufig & verteilt

Softwaresysteme werden heute von vielen Benutzern gleichzeitig verwendet. Außerdem sind diese Softwaresysteme auf verschiedene Computersysteme verteilt. In dem Buch »Java: Nebenläufige & verteilte Programmierung« werden die dafür notwendigen Programmierkonzepte vermittelt.

Anwendungen programmieren

In dem Buch »Java: Anwendungen programmieren – Von der GUI-Programmierung bis zur Datenbank-Anbindung« lernen Sie, wie

Sie grafische Benutzungsoberflächen programmieren und relationale Datenbanken anschließen, sodass Sie vollständige, einsatzfähige Anwendungen erstellen können.

Algorithmen & Datenstrukturen

Neben Programmierkonzepten gibt es auch allgemeine Konzepte für Algorithmen und sogenannte Datenstrukturen, z. B. Sortieralgorithmen. Diese muss man kennen und beherrschen, um optimale Programme schreiben zu können. In dem Buch »Java: Algorithmen und Datenstrukturen« lernen Sie die Grundlagen dazu.

Mobile Computing

Immer mehr Benutzer wollen Programme mobil verwenden. In dem Buch »Mobile Computing« lernen Sie, wie Sie mobile Anwendungen programmieren.

Didaktisches Schichtenmodell

Der hier geschilderten Vorgehensweise liegt ein didaktisches Schichtenmodell zugrunde, das die Abb. 1.0-2 zeigt.

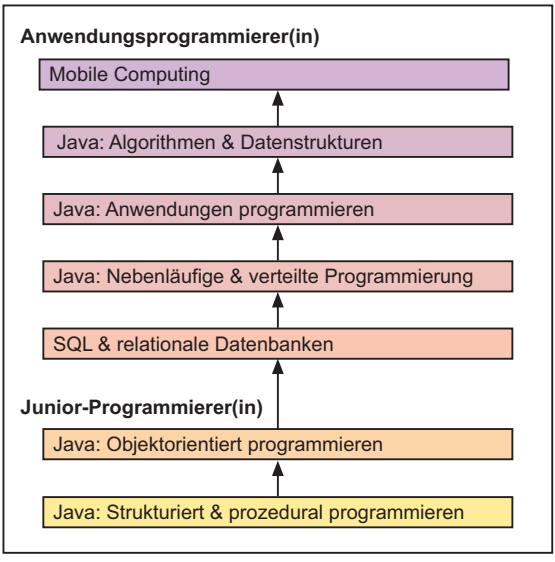


Abb. 1.0-2: Vom Juniorprogrammierer bis zum Anwendungsprogrammierer.

An den Start

Und nun legen Sie los mit dem ersten Kapitel auf dem Weg zum Junior-Programmierer bzw. zur Junior-Programmiererin.

6

1

2 Der Schnelleinstieg *

Wie lernen?

Eine Programmiersprache können Sie auf verschiedene Art und Weise lernen. Sie können sich zuerst mit den Konzepten befassen und anschließend mit der Praxis oder umgekehrt. Damit Sie zunächst ein »Gefühl« für das Programmieren bekommen, hat es sich bewährt, zunächst einige praktische Schritte zu tun, bevor die Konzepte erläutert werden.

Zuerst einige Grundlagen

Vorher ist es jedoch erforderlich, zu wissen, was eine problemorientierte Programmiersprache und eine maschinennahe Sprache sind und was in einem Computersystem abläuft:

- »Programmieren – Programme – Compiler«, S. 8

Compiler vs. Interpreter

Programme können auf unterschiedliche Art und Weise ausgeführt werden. Die Unterschiede zwischen compilierten und interpretierten Programmen lernen Sie hier kennen:

- »Skriptsprachen, Zwischensprachen und ihre Interpreter«, S. 13

Java

Einiges Wissenswertes zur Programmiersprache Java finden Sie hier:

- »Die Programmiersprache Java«, S. 17

1. Java-Programm

Nach diesen Grundlagen kann es jetzt losgehen. Sie können das erste Java-Programm auf Ihrem Computersystem »zum Laufen bringen«:

- »Das erste Java-Programm«, S. 18

Programmiergrundlagen

Die meisten Programmiersprachen besitzen ähnliche Grundkonzepte. Einige dieser Grundkonzepte werden zunächst vermittelt und dann ihre Realisierung in Java gezeigt:

- »Grundlegende Konzepte der Programmierung: das Wichtigste«, S. 27

Programmierumgebungen

Um komfortabel programmieren zu können, wird eine Programmierumgebung benötigt. Einen Überblick und den ersten Einsatz einer Java-Umgebung finden sie hier:

- » Java-Entwicklungsumgebungen«, S. 50

Ziel

Nach dem Durcharbeiten aller dieser Kapitel haben Sie einen ersten Eindruck von der Programmierung in Java, haben Ihre ersten Erfahrungen mit dem Java-Compiler und einer Java-Entwicklungsumgebung gemacht und können einige kleinere Programme bereits selbst schreiben.

2.1 Programmieren – Programme – Compiler *

Ein Computersystem besteht im Kern aus einem Prozessor und einem Arbeitsspeicher. Programme und Daten befinden sich im Arbeitsspeicher. Der Prozessor führt schrittweise die Befehle eines Programms im Arbeitsspeicher aus und manipuliert dabei die Daten. Die von einem Prozessor ausführbaren Programme sind in einer Maschinensprache geschrieben, die auf die Eigenschaften des jeweiligen Prozessors zugeschnitten ist. Programmierer verwenden dagegen problemorientierte Programmiersprachen, die dem Abstraktionsniveau von Menschen angepasst sind. Übersetzer – genauer ausgedrückt Compiler – übersetzen problemorientierte Programme in Maschinenprogramme.

Programmieren ist faszinierend. Einem **Computersystem** zu sagen, was es zu tun hat, ist eine tolle Sache! Vinton G. Cerf, Miterfinder des Internets, hat dies einmal so ausgedrückt: »*Programming is like playing God. Within the scope of the program you can do anything.*«

Programm

Damit ein Computersystem das tut, was der Programmierer will, muss er ein Programm schreiben. Ein **Programm** ist dabei nichts anderes als eine Handlungsanleitung für ein Computersystem. Damit ein Computersystem ein Programm »versteht« – genauer gesagt, es Schritt für Schritt ausführen kann – muss es bestimmte Eigenschaften besitzen:

Ein Programm muss in einer genau festgelegten Schreibweise dem Computersystem eindeutig und detailliert durch Anweisungen vorschreiben, was es Schritt für Schritt tun soll.

Syntax & Semantik

Die Syntax, d. h. die Schreibweise der einzelnen Anweisungen, und die Semantik, d. h. die Bedeutung der einzelnen Anweisungen, wird für Programme in Programmiersprachen festgelegt.

Problemorientiert vs. maschinennah

Es gibt problemorientierte Programmiersprachen (auch benutzernahe oder höhere Programmiersprachen genannt), maschinennahe Programmiersprachen und Maschinensprachen.

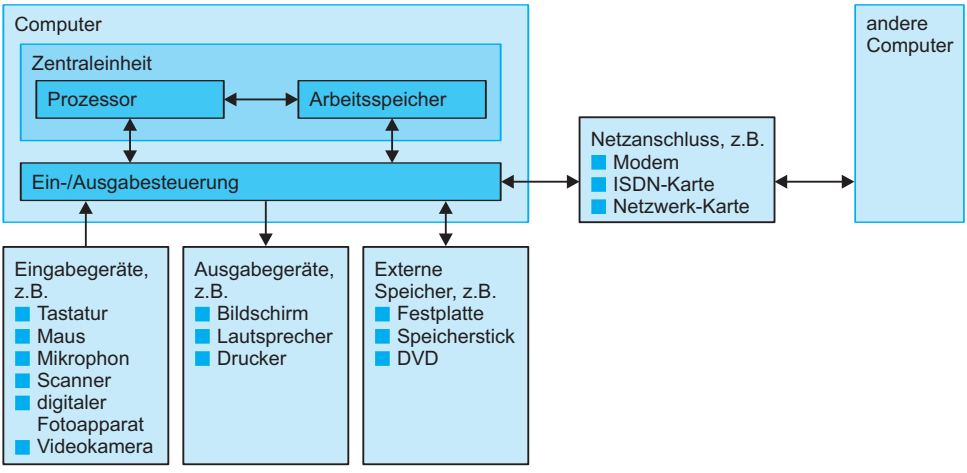
Problemorientierte Programmiersprachen wurden so konzipiert, dass es Menschen möglichst einfach haben, Programme zu schreiben.

Maschinennahe Programmiersprachen sind dagegen so aufgebaut, dass ihre Programme nahezu unverändert in Maschinensprachen transformiert werden können. Programme in einer Maschinensprache können direkt von Computersystemen ausgeführt werden.

Prozessor & Arbeitsspeicher

Ein Computersystem besteht – vereinfacht ausgedrückt – aus einem **Prozessor** und einem **Arbeitsspeicher**, beides zusammen

wird als **Zentraleinheit** bezeichnet. Die Zentraleinheit kommuniziert über eine Ein-/Ausgabesteuerung mit Eingabegeräten – wie Tastatur und Maus – Ausgabegeräten – wie Bildschirm, Lautsprecher und Drucker – externen Speichern – wie Festplatten, Speichersticks, DVDs – und über Netze mit anderen Computersystemen (Abb. 2.1-1).



Peripheriegeräte

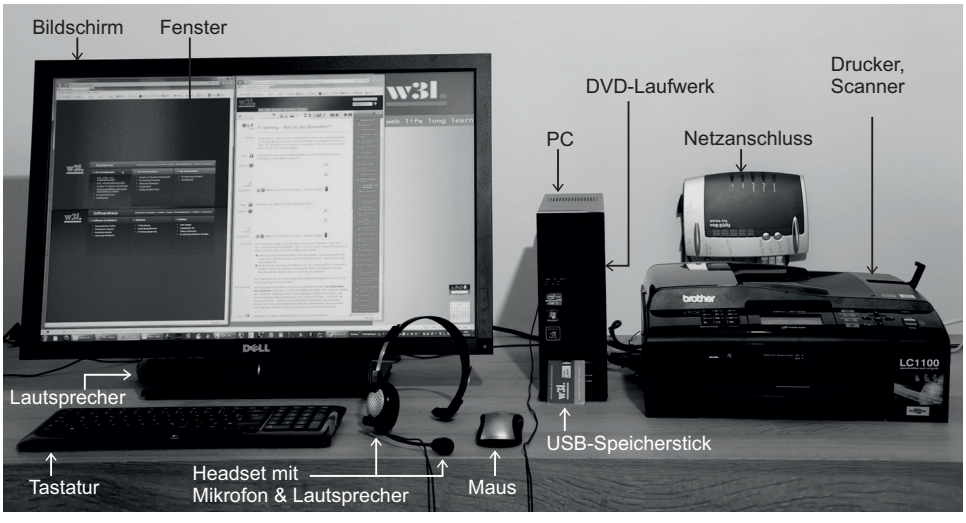


Abb. 2.1-1: So sieht der grundsätzliche Aufbau eines Computers mit seinen Komponenten aus.

Abarbeitung eines Programms

Ein Maschinenprogramm – auch Objekt-Programm genannt – wird in den Arbeitsspeicher transportiert (geladen). Der Prozessor liest jede Anweisung – Befehl genannt – und führt ihn aus. Daten werden ebenfalls im Arbeitsspeicher aufbewahrt. Der Pro-

zessor kann die Daten lesen, speichern und über die Ein-/Ausgabesteuerung einlesen und ausgeben.

Beispiel 1

Die Berechnung eines Warenwertes aus Einzelpreis mal Menge kann folgendermaßen aussehen:

In einer problemorientierten Programmiersprache sehen die Anweisungen zur Berechnung wie folgt aus (Quell-Programm genannt):

```
read(Menge);
read(PreisNetto);
read(MWST);
WarenwertNetto = Menge * PreisNetto;
WarenwertBrutto = WarenwertNetto * (MWST + 100) / 100.0;
print(WarenwertBrutto);
```

In einer maschinennahen Programmiersprache sehen die Befehle zur Berechnung z. B. so aus:

```
INPUT Menge
INPUT PreisNetto
INPUT MWST
LOAD Menge
MUL   PreisNetto
STORE WarenwertNetto
LOAD MWST
ADD 100.0
DIV 100.0
MUL WarenwertNetto
STORE WarenwertBrutto
OUTPUT WarenwertBrutto
```

Vor der Ausführung im Prozessor wird ein maschinennahes Programm noch in ein Maschinenprogramm transformiert. Ein Maschinenprogramm wird in einem **Binärcode** dargestellt. Ein Binärcode kennt nur die beiden Zeichen `0` und `1` – Binärzeichen genannt. Das Wort Binärzeichen wird als **Bit** bezeichnet, von englisch *binary digit*.

Plattform

Maschinennahe Programmiersprachen und Maschinensprachen sind im Gegensatz zu problemorientierten Programmiersprachen immer auf eine Computer-**Plattform** zugeschnitten. Eine Plattform ist gekennzeichnet durch den jeweils verwendeten Prozessortyp – z. B. Intel Core i9 – und das eingesetzte **Betriebssystem**. Unterschiedliche Plattformen besitzen auch unterschiedliche maschinennahe Sprachen und Maschinensprachen.

Übersetzer

Damit ein Programmierer in einer problemorientierten Sprache programmieren kann, gibt es Übersetzer. Allgemein ist es die Aufgabe eines **Übersetzers** in der Informatik, alle Sätze einer Quellsprache, d. h. die Quell-Programme, in *gleichbedeutende* Sätze einer Zielsprache, die Ziel-Programme, zu transformieren.

Ist das Quellprogramm in einer problemorientierten Programmiersprache geschrieben, dann bezeichnet man den Übersetzer als **Compiler**. Bei der Zielsprache kann es sich dabei um eine maschinennahe, eine »maschinennähere« Sprache oder eine Maschinensprache handeln. »Maschinennähere« Sprache bedeutet, dass die Sprache sich stärker am Aufbau eines Computersystems orientiert als die Sprache, aus der übersetzt wird.

Compiler
to compile = zusammenstellen, sammeln

Abb. 2.1-2 veranschaulicht den Übersetzungsvorgang.

Übersetzungsvorgang

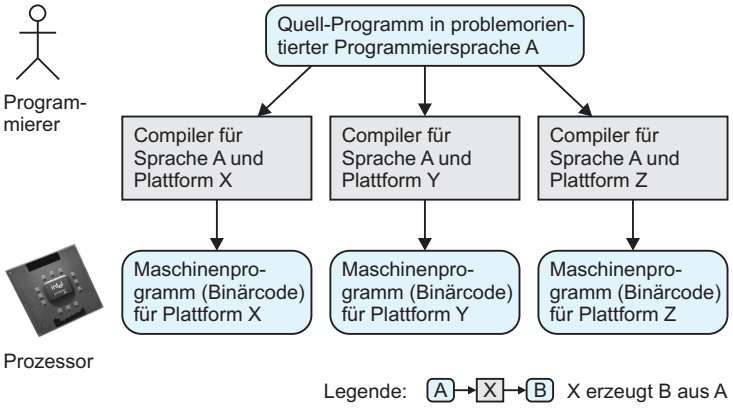


Abb. 2.1-2: Der Mensch schreibt Programme in einer problemorientierten Programmiersprache. Ein solches Programm wird von einem Compiler in ein maschinennahes Programm übersetzt. Nach der Transformation in ein Maschinenprogramm wird es vom jeweiligen Prozessor ausgeführt.

Ein Compiler übersetzt also die problemorientierte Programmdarstellung im Beispiel 1 in die darunter angegebene maschinennahe Form.

Beispiel 2

Der Einsatz eines Compilers hat folgende Vor- und Nachteile:

+ Die übersetzten Programme nutzen die jeweiligen Prozessor- bzw. Plattformeigenschaften optimal aus und erreichen dadurch eine hohe Abarbeitungsgeschwindigkeit.

Vorteil

− Für jeden Prozessortyp muss das Programm mit einem anderen Compiler neu übersetzt werden. Genau genommen unterscheiden sich die Compiler nicht nur bezüglich des Prozessortyps, sondern bezüglich der jeweiligen Computer-Plattform.

− Das übersetzte Programm läuft nur auf dem jeweiligen Prozessortyp bzw. der jeweiligen Plattform, d. h. das übersetzte Programm ist *nicht* plattformunabhängig. (Wenn das Programm nur auf einer Plattform benötigt wird, dann gilt dieser allgemeine Nachteil natürlich nicht).

Nachteile

- Oft gibt es für verschiedene Prozessortypen bzw. Plattformen nur Compiler unterschiedlicher Hersteller, die sich teilweise unterschiedlich verhalten, z. B. kann der übersetzte Code verschieden sein.
- Programmiersprachen sind oft *nicht* plattformunabhängig definiert, sodass pro Prozessortyp bzw. Plattform die übersetzten Programme voneinander abweichen (Wenn das Programm immer nur auf einer Plattform laufen soll, dann gilt dieser allgemeine Nachteil natürlich nicht).

Vom Problem zum Programm

Programme sind in der Regel sehr umfangreich und komplex. Liegt fest, welches Problem durch ein Programm gelöst werden soll, dann werden Sie meist nicht sofort das fertige Programm hinschreiben. Oft wird die Problemlösung zunächst in Form eines **Algorithmus** hingeschrieben. Ein Algorithmus ist semiformal, d. h. er ist im Gegensatz zur Umgangssprache genauer beschrieben, aber noch nicht vollständig detailliert, wie dies eine Programmiersprache erfordert (Abb. 2.1-3).

Beispiel 3

Umgangssprachlich in Form eines Algorithmus würde das Beispiel 1 folgendermaßen aussehen:

```
1. Lese Menge, PreisNetto und MWST ein
2. Berechne: WarenwertNetto = Menge * PreisNetto
3. Berechne: WarenwertBrutto =
                          WarenwertNetto * (MWST + 100) / 100
4. Gebe WarenwertBrutto aus
```

Das Wort Algorithmus

Die Bezeichnung Algorithmus geht zurück auf den arabischen Schriftsteller Abu Dshafar Muhammed Ibn Musa **al-Khwarizmi**. Er lebte um 825 n. Chr. in der Stadt Khiva im heutigen Usbekistan, die damals Khwarizm hieß und als Teil des Namens verwendet wurde. Er beschrieb die Erbschaftsverhältnisse, die sich ergaben, wenn ein wohlhabender Araber starb, der bis zu vier Frauen in unterschiedlichem Stand und eine Vielzahl von Kindern besaß. Dazu verwendete er algebraische Methoden und schrieb ein Lehrbuch mit dem Titel »Kitab al jabr w'almuqabalah« (Regeln zur Wiederherstellung und zur Reduktion), wobei die Übertragung von Gliedern einer Gleichung von einer zur anderen Seite des Gleichheitszeichens gemeint ist. Der Begriff *Algebra* leitete sich aus dem Titel des Lehrbuchs ab. Aus dem Namen des Schriftstellers wurde **algorism** und daraus **Algorithmus**.

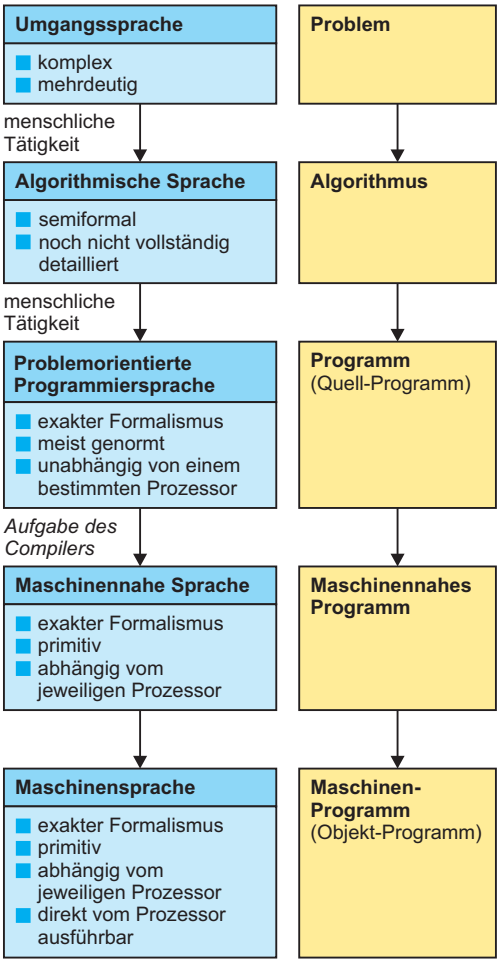


Abb. 2.1-3: Ausgangspunkt für ein Programm ist ein zu lösendes Problem bzw. eine zu lösende Aufgabe. Als Erstes wird die Problemlösung umgangssprachlich als Algorithmus formuliert. Anschließend erfolgt die Umsetzung in den strengen Formalismus (Syntax) einer Programmiersprache. Ein Compiler übersetzt dann ein solches problemorientiertes Programm in ein Maschinenprogramm.

2.2 Skriptsprachen, Zwischensprachen und ihre Interpreter *

Ein Programm, geschrieben in einer problemorientierten Programmiersprache, kann compiliert und/oder interpretiert werden. Ein Compiler kann ein Programm in eine maschinennahe Programmiersprache übersetzen oder in eine Zwischensprache, die dann interpretiert wird. Ein Programm kann auch direkt durch einen Interpreter ausgeführt werden.

Programmiersprachen lassen sich in drei Gruppen gliedern, in Abhängigkeit von der Art und Weise, wie ihre Programme ausgeführt werden:

Compilierte Programme

- Programmiersprachen, deren Programme durch einen **Compiler** in eine maschinennahe Sprache bzw. eine Maschinensprache übersetzt und dann ausgeführt werden. Zu dieser Gruppe gehört z. B. die Sprache **C++** .

Interpretierte Programme

- Programmiersprachen, deren Programme **Anweisung** für Anweisung interpretiert und – wenn die jeweilige Anweisung syntaktisch korrekt ist – sofort ausgeführt werden. Es erfolgt *keine* Übersetzung in eine maschinennahe Sprache oder eine Maschinensprache. Solche Sprachen heißen **Skriptsprachen**. **Interpreter** analysieren die Anweisungen und führen sie aus. Zu dieser Gruppe gehören Sprachen wie **JavaScript**, **JScript**, **VBScript**, Basic und Groovy (javabasierte Skriptsprache).

Compilierte & interpretierte Programme

- Programmiersprachen, deren Programme in einen Zwischencode *(intermediate language)* übersetzt werden. Die Anweisungen dieses Zwischencodes werden dann von einem Interpreter analysiert und ausgeführt, oder von einem *Just-in-Time*-Compiler (siehe unten) schrittweise übersetzt. Zu dieser Gruppe gehören die Sprachen **Java** und **C#**.
  Der von einem Java-Programm erzeugte Zwischencode heißt **Bytecode**. Der Interpreter, der den Bytecode analysiert und ausführt, heißt **JVM** *(Java virtual machine)* – kurz auch VM genannt. Der Interpreter verdeckt also die Eigenschaften der jeweiligen Plattform und bietet eine höhere »Abstraktionsschicht«. Da man sich diese Abstraktionsschicht als eine gedachte Plattform bzw. als einen gedachten Prozessor vorstellen kann, spricht man von einer virtuellen Maschine.
  Der von einem C#-Programm erzeugte Zwischencode heißt **MSIL** *(Microsoft Intermediate Language)* – kurz IL genannt. Für die Ausführung von Programmen ist die **.Net**-Laufzeitumgebung (.Net gesprochen Dot-Net), die sogenannte *Common Language Runtime* (CLR) zuständig, die als virtuelle Maschine läuft. Sie ist vergleichbar mit der JVM.
  Diese Gruppe von Programmiersprachen steht zwischen den beiden anderen Gruppen, nutzt deren Vorteile und reduziert deren Nachteile. Die erste Programmiersprache, die diese Technik benutzt hat, war die Sprache Pascal (1971), der Zwischencode heißt P-Code.

Die Abb. 2.2-1 veranschaulicht die Funktionsweise dieser drei Gruppen.

Compiler

Ein Prozessor kann nur Programme ausführen, die in seiner Maschinensprache vorliegen. Compiler übersetzen Programme in

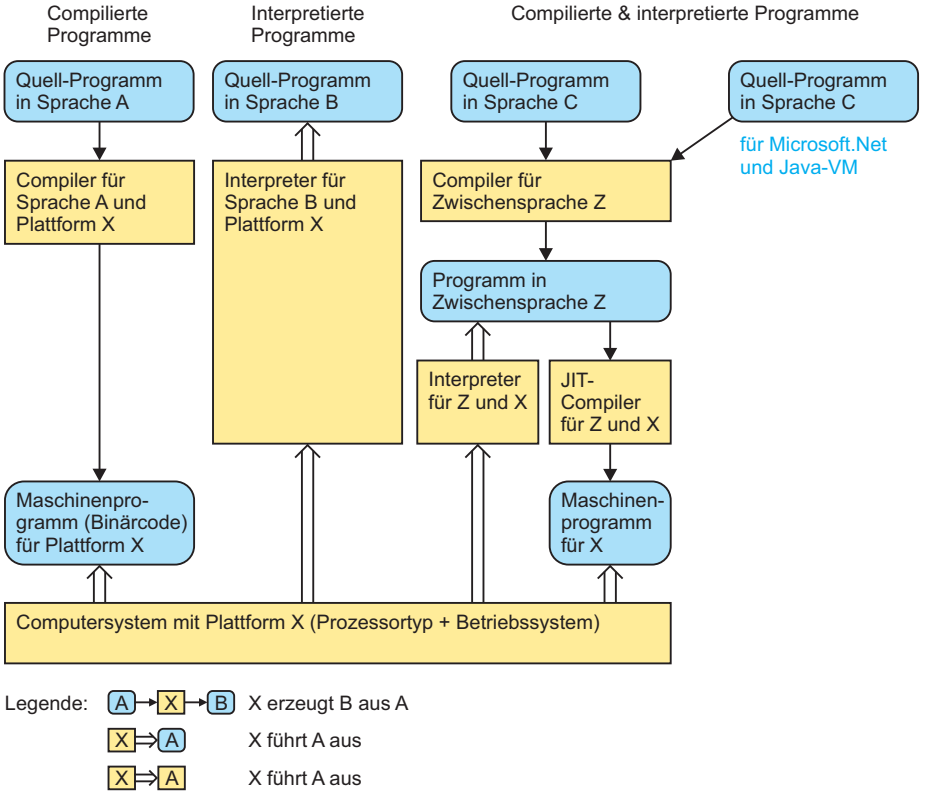


Abb. 2.2-1: Interpreter sind sozusagen »virtuelle« Prozessoren auf einer höheren Abstraktionsebene, die den realen Prozessor gegenüber der Programmiersprache »verdecken«.

der Regel in die jeweilige Maschinensprache, sodass sie direkt vom Prozessor ausgeführt werden können.

Interpreter

Interpreter sind selbst Programme, die in der Maschinensprache des jeweiligen Prozessors vorliegen und von ihm ausgeführt werden. Ein Interpreter analysiert jeweils eine **Anweisung** *(statement)* eines Programms, prüft sie auf korrekte Syntax und führt diese Anweisung dann aus. Anschließend wird die nächste Anweisung entsprechend verarbeitet.

Zwischensprache

Um Programme einer mächtigen Programmiersprache interpretieren zu können, ohne den Interpreter zu komplex werden zu lassen, werden Sprachen wie Java und C# durch einen Compiler zunächst in eine Zwischensprache übersetzt, die maschinennäher ist. Anschließend interpretiert ein »schlanker« Interpreter diese Zwischensprache. Ein Vorteil dieser Technik besteht darin, dass es einfacher ist, Interpreter für verschiedene Prozessorty-

pen zu entwickeln als Compiler für verschiedene Prozessortypen zu schreiben.

Java

In Java wird diese Technik dazu benutzt, die Sprache **plattformunabhängig** zu machen. Der Bytecode, d. h. die verwendete Zwischensprache, ist für alle Plattformen gleich. Nur die Interpreter müssen für jede Plattform angepasst werden.

JIT-Compiler

Da das Interpretieren eines Programms länger dauert als das Ausführen eines einmal übersetzten Programms gibt es noch die Variante, dass der Zwischencode *nicht* interpretiert, sondern durch einen weiteren sogenannten ***Just-in-Time-Compiler*** (JIT-Compiler) in die Maschinensprache des jeweiligen Prozessors übersetzt wird. Diese JIT-Compiler unterscheiden sich von normalen Compilern dadurch, dass sie die Zwischensprache schrittweise übersetzen. Es wird jeweils der Teil übersetzt, der für die Abarbeitung gerade benötigt wird und *nicht* das gesamte Programm wie bei einem normalen Compiler. Dadurch kann mit der Ausführung des Programms sofort begonnen werden. Bei der Ausführung muss dann jedoch u. U. auf noch nicht übersetzte Teile gewartet werden.

HotSpot

Um diesen Nachteil zu beheben, identifiziert die JVM während der Laufzeit häufig aufgerufene Stellen im Programm *(Hotspots)*, und übersetzt *nur* diese Stellen. Da somit nur ein geringer Teil des Programms übersetzt werden muss, verbleibt dem HotSpot-Compiler Zeit, diese Stellen besonders optimiert zu übersetzen. Diese Technik nennt sich »Adaptive Optimierung«.

Der Vorteil des JIT-Compilers gegenüber einem Interpreter besteht also darin, dass der Zwischencode bei jedem Programmlauf nicht jedes Mal neu übersetzt werden muss. Der bereits übersetzte Zwischencode wird in einem Zwischenspeicher *(cache)* aufbewahrt.

Microsoft

Microsoft verwendet die Zwischensprache – neben dem Ziel der Plattformunabhängigkeit – noch zu einem anderen Zweck. Programme der Programmiersprachen C++.Net, C# und Visual Basic.Net werden durch Compiler in dieselbe Zwischensprache MSIL übersetzt. Anschließend erfolgt mit einem JIT-Compiler eine schrittweise Übersetzung der Zwischensprache in den Maschinencode. Durch diese Technik ist es möglich, für eine zu programmierende Aufgabe verschiedene Programmiersprachen einzusetzen.

Java

Neben Java gibt es inzwischen auch weitere Programmiersprachen, die in den Bytecode, d. h. die Zwischensprache von Java, übersetzt werden und damit die JVM als Laufzeitumgebung nutzen. Dies gilt beispielsweise für die Sprachen Processing, Clojure, JRuby, Jython, Rhino (Javascript), Groovy und Scala.

2.3 Die Programmiersprache Java *

Die Programmiersprache Java ist eine objektorientierte Programmiersprache, die es heute ermöglicht, ein breites Anwendungsspektrum zu programmieren – vom Handy bis zur unternehmenskritischen Anwendung. Java ist plattformunabhängig, d. h. Programme, geschrieben in Java, können auf allen Plattformen ausgeführt werden, die über einen Java-Interpreter verfügen. Mit Java können Java-Anwendungen, Java-Applets, Java-Servlets und *JavaServer Pages* geschrieben werden.

Java ist eine **problemorientierte** und **objektorientierte** Programmiersprache. Problemorientiert bedeutet, dass die Sprachkonstrukte so gewählt wurden, dass einem Programmierer das Programmieren möglichst einfach gemacht wird. Objektorientiert bedeutet, dass die Konzepte der objektorientierten Programmierung wie Klassen, Objekte und Vererbung unterstützt werden.

Zur Historie

Das Fundament für die Programmiersprache Java wurde 1990 gelegt. Bei der Firma Sun untersuchte ein Entwicklungsteam mit den Innovatoren Patrick Naughton, James Gosling und Mike Sheridan den Konsumentenmarkt. Es erkannte, dass zunehmend Mikroprozessoren in alle elektronischen Konsumgeräte integriert wurden, sowohl in Videorekorder als auch in Telefone und Waschmaschinen.

Ziel des Teams war es daher, ein einfaches, herstellerunabhängiges Betriebssystem zu entwickeln. James Gosling erfand unter dem Namen »Oak« eine dafür geeignete, plattformunabhängige, robuste und sichere objektorientierte Programmiersprache. Im August 1992 stellte das Team den Projektstatus dem Vorsitzenden von Sun vor.

Für die Präsentation setzten sie eine Zeichentrickfigur – Duke genannt – ein, die später zum »Maskottchen« für Java wurde. Die Ideen wurden von Sun großartig aufgenommen. Es wurde eine unabhängige Firma »First Person« gegründet, um mit Herstellern von Konsumelektronik-Geräten zu verhandeln. Der Markt war für diese Ideen aber noch *nicht* reif, sodass alle Verhandlungen scheiterten. »First Person« wurde 1994 aufgelöst.

Sprache für das Internet

Zu diesem Zeitpunkt erlebte das Internet eine rasante Entwicklung. Sun erkannte das Potenzial der sicheren, plattformunabhängigen Programmiersprache für das Internet. Im Januar 1995 wurde »Oak« in »Java« umbenannt. Dies hatte vor allem markenzeichenrechtliche Gründe. Auf den Namen Java kam das Team in der Cafeteria – in den USA wird für Kaffee der Name Java verwendet.

Die Sprache war inzwischen verbessert und weitere Sicherheitskomponenten waren hinzugefügt worden. Sun stellte die Sprache für das Internet bereit. Diese historische Entwicklung macht einige Besonderheiten von Java – verglichen mit anderen Programmiersprachen – verständlich.

In Java kann man mehrere Arten von Programmen schreiben:

Java-Anwendungen

- **Java-Anwendungen** *(applications)*, die selbstständig auf einem Computersystem laufen – dies ist die übliche Form bei anderen Programmiersprachen.

Java-Applets

- **Java-Applets** *(applets)*, die über das Internet von einem Web-Server geladen und in einem Web-Browser ausgeführt werden – dies unterscheidet Java von anderen Programmiersprachen. Applets können – im Unterschied zu Anwendungen – auf lokale Daten des Computersystems, auf das sie vom Netz geladen wurden, in der Regel nicht zugreifen. Sie können auch keine Daten auf der Festplatte des lokalen Computersystems speichern. Der Grund für diese Einschränkungen liegt in dem Sicherheitskonzept der Java-Applets. Es ist durch dieses Konzept sichergestellt, dass ein über das Netz geladenes Applet keinen Schaden auf dem Client-Computersystem anrichten kann.

Java-Servlets & JSPs

- **Java-Servlets** *(servlets)* werden auf einem Web-Server ausgeführt, in der Regel angestoßen über Befehle in einem HTML-Dokument. Diese Technik wurde 1996 entwickelt – als Gegenstück zu den Java-Applets. Um die Programmierung von Web-Anwendungen zu erleichtern, wurden 1999 **JSPs** *(Java-Server Pages)* erfunden, die auf den Java-Servlets aufsetzen. JSPs werden automatisch in Java-Servlets transformiert.

Vom Handyprogramm bis zur Unternehmensanwendung

Java unterstützt sowohl die Programmierung von Geräten, die nur eingeschränkte Ressourcen besitzen, z. B. Handys, als auch von umfangreichen Unternehmensanwendungen, die besondere Anforderungen erfüllen müssen.

Sowohl Java-Anwendungen als auch Java-Applets und Java-Servlets sind übersetzte und lauffähige Java-Programme – jeweils in der dafür vorgesehenen Umgebung.

Oracle

Die Programmiersprache Java wurde ursprünglich von der Firma Sun Microsystems entwickelt. 2009 übernahm die Firma Oracle die Firma Sun Microsystems.

2.4 Das erste Java-Programm *

Bevor Sie das erste Java-Programm schreiben, übersetzen und ausführen können, müssen Sie das notwendige (Hand-)Werkzeug auf Ihrem Computersystem installiert haben.

Texteditor

Im einfachsten Fall benötigen Sie einen Texteditor oder ein Textverarbeitungssystem, um ein Programm »einzutippen«.

Tipp

Als reinen Text speichern
Wenn Sie ein Textverarbeitungssystem, wie beispielsweise Microsoft Word, verwenden, dann speichern Sie den Text als Dateityp `Nur Text` (in Word: `txt`), um keine Auszeichnungsformate im Text zu erhalten, die der Compiler nicht verarbeiten kann.

Dateiendung `.java`

Das eingetippte Quellprogramm müssen Sie anschließend als Datei abspeichern. Bei einem Java-Programm muss die Datei den Dateisuffix `.java` erhalten, damit der Compiler prüfen kann, ob die angegebene Datei ein geeignetes Quellprogramm enthält.

Java-Compiler

Wenn Sie **Java**-Programme übersetzen wollen, dann benötigen Sie als Minimum einen Java-Compiler einschließlich einer **JVM** *(Java virtual machine)* für Ihre Computer-**Plattform**.

JDK

Die Firma Oracle stellt ein kostenloses **JDK** *(Java Development Kit)* zur Verfügung. Installieren Sie das JDK (siehe Internet) und machen sich dem Konsolenfenster und seinen Grundbefehlen vertraut (suchen Sie im Web nach Konsolenfenster & cmd-Befehle).

Nachdem Sie **das JDK installiert** haben, machen Sie sich mit der **Bedienung des Konsolenfensters** vertraut! Wichtige Befehle: `cd`, dir. Aufruf: Lupe, Eingabe: `cmd`, als Admin ausführen.

1. Java-Programm

Nach der Installation des JDK können Sie Ihr erstes Programm übersetzen und ausführen:

- »»Hello World« mit Java«, S. 19

Aufbau

Jedes Java-Programm ist nach demselben Schema aufgebaut:

- »Zum Aufbau eines Java-Programms«, S. 24

2.4.1 »Hello World« mit Java *

Ein Java-Programm kann mit einem Texteditor erfasst werden. Die Datei muss die Endung `.java` erhalten. Der Dateiname und der Klassenname des Programms müssen identisch sein. Mit dem JDK *(Java Development Kit)* von Oracle wird das Programm durch den Befehl `javac Dateiname.java` übersetzt und durch den Befehl `java Dateiname` ausgeführt.

Tradition

Traditionell ist das erste Programm, das ein Programmierer schreibt, wenn er eine neue Sprache lernt, das Programm »*Hello World*«. Tradition ist es ebenfalls, das fertige Programm, wie es im Folgenden vorgegeben wird, mechanisch in einen **Texteditor** einzutippen, es zu übersetzen und dann zu sehen, wie es läuft, bevor erklärt wird, was die einzelnen Programmzeilen bewirken.

Das folgende Beispiel zeigt eine einfache Java-Anwendung, die den Text »Hello World!« als Zeichenfolge in einem zeichenorientierten Bildschirmfenster ausgibt.

Beispiel: 1. Java-Anwendungs-Programm

```
/*
Dies ist ein Kommentar
*/
// Und dies ist ein Zeilenkommentar
public class HelloWorld  //HelloWorld ist ein Klassenname
//Der Dateiname muss HelloWorld.java heißen!
{
 //Dies ist eine Operation bzw. Methode
 public static void main (String args[])
 {
   //Dies ist eine Ausgabeanweisung
   System.out.println("Hello World!");
 }
}
```

Schritt 1: Programm erfassen

Erfassen Sie das im Beispiel angegebene Programm in einem neuen Dokument in einem Texteditor. Speichern Sie diese sogenannte Quell-Datei *(source file)* unter dem Namen `HelloWorld.java` in ein Verzeichnis, in das Sie Ihre Java-Programme speichern wollen, z. B. `C:\Java`. Hier in diesem Beispiel wird ein Unterverzeichnis `HelloWorld` für das Programm angelegt. Die Datei `HelloWorld.java` wird somit im Verzeichnis `C:\Java\HelloWorld` abgelegt.

Hinweis 1

Auf korrekte Groß- und Kleinschreibung achten
In Java wird zwischen Groß- und Kleinbuchstaben unterschieden, d. h. `Dateiname` und `DateiName` sind unterschiedliche Dateibezeichnungen.

Hinweis 2

Dateiname = Klassenname
Der Programmname, der in Java-Programmen hinter `class` angegeben ist, sollte identisch mit dem Namen der Quell-Datei sein. Sonst wird eine `.class`-Datei mit dem Klassennamen erzeugt. Das Programm muss dann mit dem `.class`-Dateinamen gestartet werden.

Schritt 2: Compiler starten

Starten Sie als Nächstes den Java-Compiler:

Wenn Sie eine Windows-Plattform benutzen, dann öffnen Sie das Lupensymbol und geben in das Eingabefeld `cmd` (bei Windows 11) ein. Es öffnet sich ein Konsolenfenster (Standardeinstellung: Schwarzer Hintergrund, Weiße Schrift).

Konsolenfenster

Wechseln Sie in das Verzeichnis, wo Ihr Programm ist, z.B. durch Eingabe von `cd Java`, wenn Ihr Verzeichnis `C:\Java` ist (Abb. 2.4-1). In diesem Beispiel wird in das Verzeichnis `C:\Java\HelloWorld` gewechselt.

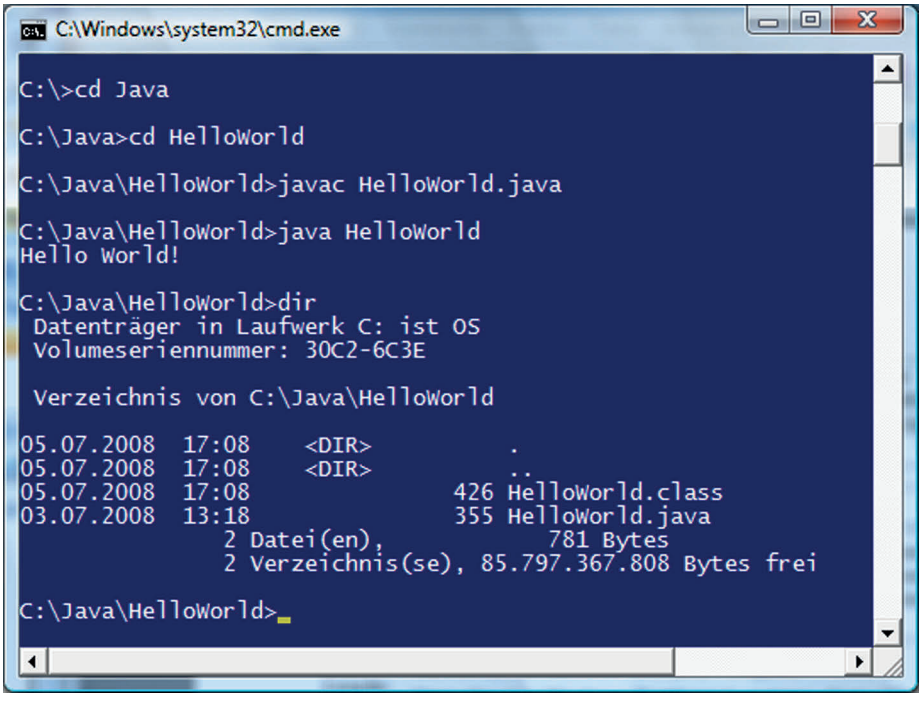


Abb. 2.4-1: Übersetzen und Ausführen der Java-Anwendung `Hello World` im Konsolenfenster mit dem JDK.

Geben Sie den Befehl `javac HelloWorld.java` ein.

Hinweis 3

Compilerstart durch Befehl javac

Mit dem Befehl `javac` (steht für *Java C*ompiler) gefolgt von dem Dateinamen rufen Sie den Java-Compiler auf, der dann die angegebene Quell-Datei in eine Bytecode-Datei mit der Endung `.class` übersetzt. Voraussetzung für die Verwendung des `javac`-Befehls ist, dass Sie die sogenannten Path- und CLASSPATH-Einstellungen vorgenommen haben. Damit wird dem JDK mitgeteilt, wo sich der Java-Compiler und das auszuführende Programm auf Ihrem Computer befinden. Informieren Sie sich dazu im Web unter »Oracle, Path, CLASSPATH« (siehe auch Abb. 2.5-8 auf Seite 50).

Wenn die Eingabeaufforderung, z. B. `C:\>`, ohne Fehlermeldung erscheint, haben Sie Ihr Programm erfolgreich übersetzt. Der Compiler hat die Bytecode-Datei `HelloWorld.class` erzeugt.

Fehlerquellen

Mögliche Fehler

Sollten Sie eine den folgenden Meldungen ähnliche Fehlermeldung erhalten:

```
Bad command or file name
```

oder

```
The name specified is not recognized as
an internal or external command,
operable program or batch file
```

dann kann Windows den Java-Compiler `javac` nicht finden. Überprüfen Sie, ob der sogenannte *classpath* richtig eingestellt ist.

Schritt 3: Übersetztes Programm starten

Starten Sie Ihr übersetztes Java-Programm, indem Sie im *gleichen* Verzeichnis den Befehl `java` gefolgt von dem Dateinamen (ohne Endung) eingeben: `java HelloWorld`. An der Eingabeaufforderung erscheint als Ergebnis des Programmlaufs der Text `Hello World!`. Das war nicht spektakulär, aber dennoch das Ergebnis Ihres ersten Java-Programms!

Hinweis 4

Start des übersetzten Programms durch java

Mit dem Befehl `java` gefolgt von dem Dateinamen ohne Dateinamensendung starten Sie ein übersetztes Java-Programm, d. h. die `.class`-Datei.

Ersetzen Sie den Text `Hello World` in der Zeile

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

durch einen eigenen Text. Übersetzen Sie das Programm neu und starten Sie es. Welches Ergebnis erhalten Sie?

Lassen Sie in Ihrem Programm am Ende der Zeile

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

das Semikolon weg und übersetzen Sie das Programm erneut. Was stellen Sie fest?

Sie erhalten vom Compiler eine Fehlermeldung, dass Ihr Programm nicht korrekt ist. Java-Programme müssen exakt der Java-Syntax entsprechen, sonst meldet der Compiler einen Fehler und Sie können Ihr Programm nicht ausführen.

Probieren Sie aus, was passiert, wenn Sie `System.out.println("");` und `System.out.println();` verwenden.

Erstellen, übersetzen, ausführen

Die Erstellung, Übersetzung und Ausführung einer Java-Anwendung verdeutlicht die Abb. 2.4-2.

Den vom Compiler erzeugten Bytecode (siehe auch »Skriptsprachen, Zwischensprachen und ihre Interpreter«, S. 13) können Sie sich mit geeigneten Werkzeugen ansehen. Recherchieren Sie im Web nach »Bytecode Viewer«.

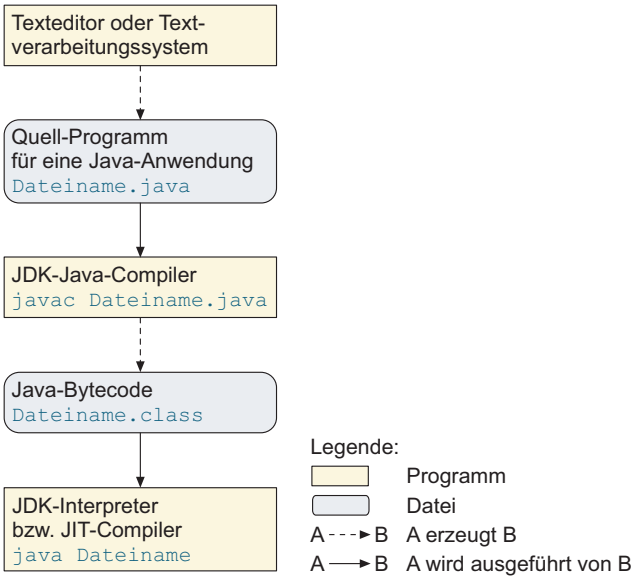


Abb. 2.4-2: Erstellung, Übersetzung und Ausführung eines Java-Programms mit dem JDK.

Beispiel

Das folgende Beispielprogramm führt zu dem Bytecode der Abb. 2.4-3:

```
public class DemoByteCode
{
  public static void main (String args[])
  {
      int Menge = 50;
      int PreisNetto = 25;
      double MWST = 19.0;
      double WarenwertNetto = Menge * PreisNetto;
      double WarenwertBrutto =
          WarenwertNetto * (MWST + 100) / 100.0;
      System.out.println(WarenwertBrutto);
   }
}
```

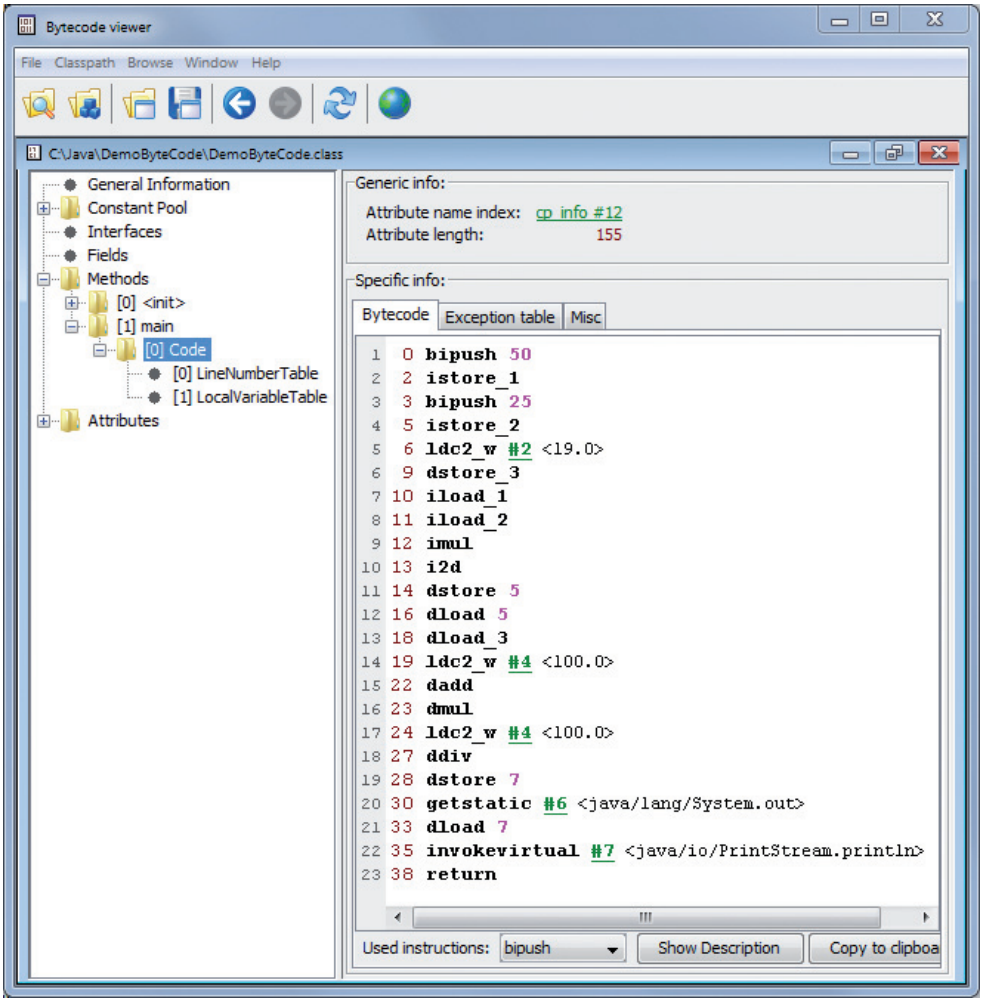


Abb. 2.4-3: Bytecode des Java-Programms DemoByteCode.

2.4.2 Zum Aufbau eines Java-Programms *

Ein Java-Programm besteht aus einer oder mehreren Klassen: `class Klassenname {...}`. Innerhalb einer Klasse kann es eine oder mehrere Methoden geben. Die Methode, die mit `public static void main(String args [])` beginnt, wird beim Programmstart zuerst ausgeführt. Es gibt drei verschiedene Kommentararten.

Klasse

Jedes Java-Programm besteht aus *mindestens* einer oder mehreren sogenannten **Klassen**. Eine Klasse wird in Java durch das Schlüsselwort `class` gekennzeichnet. Steht vor dem Klassenname das Schlüsselwort `public`, dann ist das Programm von anderen Programmen aus sichtbar, sonst nur in einem bestimmten Kon-

text. Hinter dem Schlüsselwort `class` folgt der Klassenname, der die Klasse kennzeichnet. Alles, was anschließend folgt und zur Klasse gehört, wird in geschweifte Klammern `{...}` eingeschlossen.

Dies ist ein grundlegendes Prinzip in Java. Immer wenn man etwas Zusammengehöriges zusammenfassen will, klammert man es in geschweifte Klammern. Das bedeutet auch, dass Klammern immer paarweise auftreten. Fehlt eine Klammer, dann meldet der Compiler einen Fehler.

{...}

Beispiel 1

```
/*
Dies ist ein Kommentar
*/
// Und dies ist ein Zeilenkommentar
public class HelloWorld  //HelloWorld ist ein Klassenname
//Der Dateiname muss HelloWorld.java heißen!
{
 //Dies ist eine Operation bzw. Methode
 public static void main (String args[])
 {
  //Dies ist eine Ausgabeanweisung
  System.out.println("Hello World!");
 }
}
```

Operationen bzw. Methoden

Eine Klasse kann wiederum mehrere Operationen – in Java **Methoden** genannt – enthalten, im Beispiel heißt die Operation `main`. Alles, was anschließend folgt und zur Operation gehört, wird wiederum in geschweifte Klammern eingeschlossen. Kann eine Operation von anderen Programmen genutzt werden, dann steht das Schlüsselwort `public` davor.

Hinweis

Auf die Bedeutung von `public static void main (String args[])` wird an dieser Stelle nicht näher eingegangen, da dazu erst weitere Konzepte behandelt werden müssen (siehe »Felder als Eingabeparameter«, S. 219).

Richtlinien

Formatierung von Programmen

Folgende Richtlinien sollten eingehalten werden, um gut lesbare Programme zu erhalten:

- Paarweise zusammengehörende geschweifte Klammern (eine öffnend, eine schließend) stehen in der Regel immer in derselben Spalte untereinander.
- In der Zeile, in der eine geschweifte Klammer steht, steht in der Regel sonst nichts mehr.
- Alle Zeilen innerhalb eines Klammerpaars sind jeweils um eine feste Anzahl Zeichen nach rechts eingerückt, z. B. zwei Zeichen.

Ausgabeanweisung println()

Innerhalb der Operation `main` steht in dem Programm `HelloWorld` die Ausgabeanweisung

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

Diese Anweisung gibt den Text, der in `" "` eingeschlossen ist, in einem Bildschirmfenster aus. Ein so gekennzeichneter Text ist in Java eine Zeichenkette (`String`). Ändert man den Text, übersetzt das Programm neu und startet es, dann wird der geänderte Text angezeigt.

Duplizieren Sie die Ausgabeanweisung und fügen Sie sie mehrfach hinter der bisherigen Ausgabeanweisung ein. Ändern Sie in jeder Zeile den Text, der in Anführungszeichen steht. Übersetzen Sie das Programm und führen Sie es aus. Was sehen Sie als Ergebnis?

Diese Anweisung gibt also einen Text zeichenweise in einer Zeile in einem Bildschirmfenster aus. Am Ende der Zeile wird auf eine neue Zeile positioniert (`println` = *print line* = drucke Zeile).

print()

Die Ausgabeanweisung `System.out.print("Java");` gibt den Text `Java` aus, ohne auf eine neue Zeile zu positionieren, d. h. ohne einen Zeilenvorschub vorzunehmen. Eine weitere Anweisung `System.out.print(" ist toll!");` würde folgenden Text in eine Zeile schreiben: `Java ist toll!`

Semikolon

Anweisungen werden *immer* durch ein Semikolon abgeschlossen!

main

Jede Java-Anwendung muss übrigens in *mindestens* einer Klasse eine Operation mit dem Namen `main` besitzen, da diese Operation beim Start der Anwendung zuerst ausgeführt wird.

Schlüsselwörter, Wortsymbole

Jede Programmiersprache besitzt eine Reihe von **Schlüsselworten** bzw. **Wortsymbolen**. Dabei handelt es sich um Worte wie `class`, `public` und `void`, die eine festgelegte Bedeutung in der Sprache besitzen und *nicht* für andere Zwecke benutzt werden dürfen.

Kommentare

Ein Programm wird heute in der Regel nicht nur vom Autor des Programms gelesen, sondern auch von anderen Personen, z. B. Kollegen, der Qualitätssicherung usw. Es ist daher nötig, dass ein Programm gut dokumentiert ist. Eine gute Dokumentierung erhält man u. a. durch die geeignete Verwendung von Kommentaren in einem Programm. Jede Programmiersprache erlaubt es, Kommentare in Programme einzufügen. Kommentare werden vom Compiler überlesen. Dem menschlichen Leser erleichtern passende Kommentare das Verständnis des Programms erheblich.

Java-Kommentare

In Java werden drei verschiedene Arten von Kommentaren unterschieden (siehe Beispiel 1):

- Traditioneller Kommentar: `/*Kommentar*/`
  Alle Zeichen zwischen `/*` und `*/` werden vom Compiler überlesen (dies ist auch die übliche Kommentarart in den Sprachen C und C++).
- Einzeilenkommentar: `//Kommentar`
  Alle Zeichen nach `//` bis zum Zeilenende werden überlesen (übliche Kommentarart in der Sprache C++).
- Dokumentationskommentar: `/**Kommentar*/`
  Wie der traditionelle Kommentar, jedoch kann dieser Kommentar von dem Java-Programm `Javadoc` sowie einigen Programmierumgebungen ausgewertet werden, um eine automatische Dokumentation im **HTML**-Format zu erstellen.

Gehen Sie im Weiteren immer von dem in der Abb. 2.4-4 dargestellten **Programmschema** aus.

Programmschema

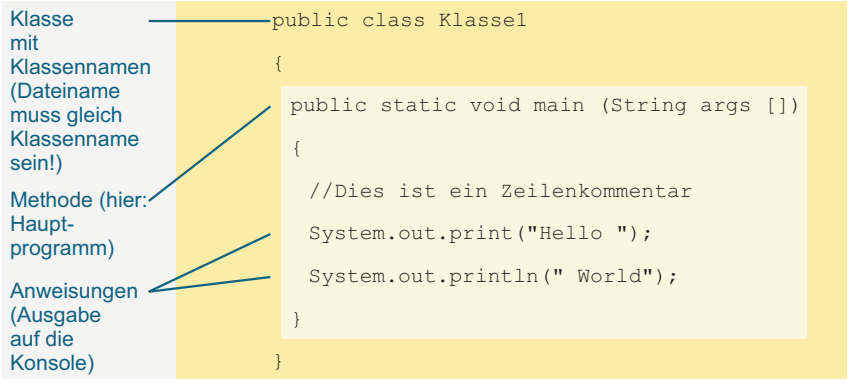


Abb. 2.4-4: Grundaufbau eines Java-Programms: Klasse und eingebettetes Hauptprogramm.

2.5 Grundlegende Konzepte der Programmierung: das Wichtigste *

Die wichtigsten Konzepte fast aller Programmiersprachen sind gleich. Die Syntax und Semantik ist in den einzelnen Programmiersprachen jedoch – meist geringfügig – verschieden.

Hinweis

Wenn Sie bereits eine Programmiersprache kennen, dann können Sie die folgenden allgemeinen Konzepte überspringen und direkt die Realisierung in Java betrachten:

- »Java-Programm mit lokalen Variablen und einfachen Anweisungen«, S. 36

Damit Sie alle Programme, die hier vorgestellt und beschrieben werden, nachvollziehen, erweitern und als Grundlage für eige-

ne Programme nehmen können, laden Sie alle Programme auf Ihr Computersystem, legen Sie ein Verzeichnis an, z. B. `Java`, und entpacken Sie in dieses Verzeichnis die heruntergeladene Datei.

Sie finden alle Programme zum Herunterladen unter der DOI `10.18420/LB-JavaEinstieg`.

Grundlegend für jede Programmiersprache sind die Konzepte **Variable** und **Typ**, die hier beschrieben werden:

Grundkonzepte

- »Variablen, Konstanten und Typen«, S. 28

Die elementare Operation in einer Programmiersprache ist die **Zuweisung**. **Ausdrücke** legen fest, wie Werte von Variablen miteinander verknüpft werden:

- »Zuweisung und Ausdrücke«, S. 32

Realisierung in Java

Die Anwendung dieser Grundkonzepte in Java sieht folgendermaßen aus:

- »Java-Programm mit lokalen Variablen und einfachen Anweisungen«, S. 36

In Java ist es einfach, **Ausgaben** in das Konsolenfenster vorzunehmen. Wie **Eingaben** über das Konsolenfenster erfolgen können, wird hier beschrieben:

- »Java-Programme mit Konsoleneingabe«, S. 39

Ziel

Die Beherrschung dieser Grundkonzepte und ihre Realisierung in Java ermöglicht es Ihnen, bereits kleinere Programme selbst zu schreiben.

2.5.1 Variablen, Konstanten und Typen *

Das Variablenkonzept ist grundlegend für jede Programmierung. Eine Variable besteht aus einem Bezeichner, der einen »logischen« Speicherplatz adressiert, und dem zugehörigen Speicher-Inhalt, Wert genannt. Der (Daten-)Typ der Variablen gibt an, welche Werte für die Variable erlaubt bzw. zulässig sind. Kann auf eine Variable nach der Initialisierung nur lesend zugegriffen werden, dann handelt es sich um eine Konstante. Die Lebensdauer gibt an, wie lange eine Variable mit ihrem Wert existiert, d. h. vorhanden ist. Der Sichtbarkeits- bzw. Gültigkeitsbereich legt fest, von welchen anderen Programmteilen aus auf eine Variable zugegriffen werden kann.

Bezeichner & Wert

Ein Basiskonzept jeder Programmierung ist das Konzept der Variablen. Eine **Variable** besteht aus einem Namen bzw. einem **Bezeichner** *(identifier)* und einem **Wert** *(value)*. Technisch betrachtet gibt der Bezeichner der Variablen einen Speicherplatz im **Arbeitsspeicher** eines Computers an. In diesem Speicher-

platz können nacheinander verschiedene Werte gespeichert werden. Man sagt: Der Variablen wird ein Wert zugewiesen.

Literal

Der Wert einer Variablen wird durch ein **Literal** *(literal)* dargestellt. Beispielsweise wird in den meisten Programmiersprachen eine Zeichenkette in doppelte Anführungszeichen eingeschlossen wie `"Dies ist eine Zeichenkette"`. Die Abb. 2.5-1 veranschaulicht das Variablenkonzept.

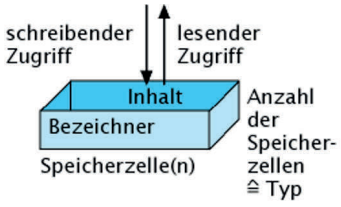


Abb. 2.5-1: Eine Variable kennzeichnet mit ihrem Bezeichner einen Speicherplatz im Arbeitsspeicher, in dem ein Wert gespeichert, d. h. aufbewahrt werden kann.

Hinweis

In jeder Programmiersprache gibt es Vorschriften, wie die Bezeichner aufgebaut sein dürfen und wie die Werte dargestellt werden (Syntax). Manche Programmiersprachen unterscheiden die Klein- und Großschreibung, andere nicht. Wird groß und klein unterschieden, dann sind `Gewicht` und `gewicht` unterschiedliche Variablen. Bei den Werten werden Nachkommastellen *nicht* durch ein Komma, sondern durch einen Punkt von den Vorkommastellen getrennt, d. h. `65.70` statt `65,70`. Maßeinheiten wie `kg` werden *nicht* angegeben (Tab. 2.5-1).

Fachlicher Name	Mögliche Bezeichner	Mögliche Werte
Körpergewicht	`koerpergewicht`, `koerper_gewicht`, `koerperGewicht`, `gewicht`, `weight`	80, 65.70
Körpergröße	`koerpergroesse`, `koerper_groesse`, `koerperGroesse`, `groesse`, `height`	1.60, 1.80
Kontonummer	`kontonummer`, `kontonr`, `nr`, `number`	8067788, 121239

Tab. 2.5-1: Beispiele für Bezeichner.

Konvention

In objektorientierten Programmiersprachen ist es üblich, Bezeichner **mit einem Kleinbuchstaben** zu beginnen. Für diejenigen, die sich schon etwas auskennen: Dies tut man, um Verwechslungen mit Klassennamen zu vermeiden, die immer mit einem Großbuchstaben beginnen.

Englisch vs. Deutsch

Als Bezeichner können Sie englische oder deutsche Begriffe wählen (oder eine andere Sprache). Empfehlenswert ist es, dass Sie

sich auf eine Sprache festlegen und *nicht* ständig zwischen verschiedenen Sprachen wechseln.

Auf Umlaute verzichten

Manche Programmiersprachen – auch Java – erlauben es, Umlaute in Bezeichnern zu verwenden. Um Programme international austauschen zu können, sollten Sie darauf aber verzichten.

Empfehlung

Sie sollten konsequent eine **Namenssystematik** anwenden, um Schreibfehler zu minimieren, z. B. `geburts_gewicht` (alles klein mit Unterstrich) oder `geburtsGewicht` (Jedes neue Wort eines Bezeichners beginnt mit einem Großbuchstaben, außer dem ersten Buchstaben. Das ist die sogenannte Kamelhöcker-Notation).

Verbalisierung

Die Bezeichner sollen möglichst detailliert über den Verwendungszweck der Variablen Auskunft geben. In dem obigen Beispiel ist daher der problemorientierte Bezeichner `koerperGewicht` dem Bezeichner `gewicht` vorzuziehen, wenn es um das Körpergewicht in dem Programm geht. Eine solche bewusste Wahl der Bezeichner bezeichnet man als **Verbalisierung**.

Bezeichner in Java

In Java besteht ein Bezeichner aus einer beliebig langen Sequenz von Java-Buchstaben *(Java letters)* und Java-Ziffern *(Java digits)*, wobei das erste Zeichen ein Java-Buchstabe sein muss (siehe Java Language Specification (`https://docs.oracle.com/javase/specs/jls/se18/jls18.pdf`, S. 26). Ein Java-Buchstabe ist einer der Buchstaben A bis Z oder a bis z. Aus historischen Gründen sind auch der Unterstrich (_) sowie das $-Zeichen erlaubt. Das $-Zeichen sollte aber nur für automatisch generierten Quellcode verwendet werden. Die Java-Ziffern umfassen die Ziffern 0 - 9.

Typ

Für jede Variable muss ein **Typ** *(type)* festgelegt werden. Ein Typ legt fest, welche Werte eine Variable annehmen kann, und welche Operationen auf diesen Werten ausgeführt werden können. Durch den Typ wird auch festgelegt, wie viel Speicherplatz, d. h. wie viele Speicherzellen für die Werte benötigt werden. Für ein Zeichen *(character)* sind zwei Speicherzellen, für eine Zeichenkette *(string)*, bestehend aus 20 Zeichen, sind 40 Speicherzellen erforderlich, wenn sie im sogenannten **Unicode** vorliegen.

In vielen Programmiersprachen werden Typen durch Schlüsselwörter gekennzeichnet, z. B. `int` für ganze Zahlen, `double` für Gleitkommazahlen.

Beispiele

Die Variablen `koerperGroesse` und `koerperGewicht` haben den Typ »Gleitkommazahl« bzw. »Gleitpunktzahl«, d. h. sie besitzen Nachkommastellen.

Eine Variable `kontonummer` besitzt den Typ »Ganze Zahl«, d. h. sie hat keine Nachkommastellen.

Zwei Operationen können mit Speicherzellen ausgeführt werden:

- Schreibender Zugriff auf eine Speicherzelle:
  Es wird eine Information bzw. ein Wert oder Inhalt in der Speicherzelle abgelegt. Vorher in der Speicherzelle vorhandene Informationen werden überschrieben, d. h. gelöscht.

Schreibender Zugriff

- Lesender Zugriff auf eine Speicherzelle:
  Es wird die in der Speicherzelle gespeicherte Information gelesen. Die gespeicherte Information bleibt dabei *unverändert*. Beim Lesen wird nur eine Kopie der gespeicherten Information erzeugt.

Lesender Zugriff

Dieses technische Speicherkonzept spiegelt sich in problemorientierten Programmiersprachen in dem Konzept der Variablen und Konstanten wider.

Einer **Variablen** können nacheinander verschiedene Werte bzw. Inhalte zugewiesen werden, d. h. auf eine Variable kann sowohl lesend als auch schreibend zugegriffen werden. Vor dem ersten Lesezugriff muss immer ein Schreibzugriff erfolgt sein, damit der Variablen ein definierter Wert zugewiesen ist.

Variable

Einer **Konstanten** kann nur einmal ein Wert zugewiesen werden, der dann unveränderbar ist, d. h. auf eine Konstante kann nach der Initialisierung nur lesend zugegriffen werden. Eine Konstante ist sozusagen ein **Sonderfall einer Variablen**. In den meisten Programmiersprachen werden Konstanten besonders gekennzeichnet, um sie von Variablen zu unterscheiden.

Konstante

Variablen müssen, bevor auf sie zugegriffen werden darf, deklariert bzw. vereinbart werden (Variablen-Deklaration). Diese Regel gilt in den meisten Programmiersprachen. Der Compiler verwendet diese Informationen, um Speicherplatz zu reservieren und Konsistenzprüfungen vorzunehmen.

Variablen-Deklaration

Für jede Variable und jede Konstante wird außerdem festgelegt, ob und für wen sie sichtbar sind, d. h., welche anderen Programmteile auf sie Zugriff haben. Anstelle von Sichtbarkeit spricht man auch von Gültigkeit. Die Sichtbarkeit wird entweder explizit oder implizit festgelegt. Implizit wird die Sichtbarkeit durch den Ort der Deklaration festgelegt (siehe »Die Sequenz«, S. 106, und »Parameterlose Prozeduren«, S. 208).

Sichtbarkeitsbereich

Die Lebensdauer einer Variablen gibt an, wie lange sie im Speicher existiert bzw. ob sie auf einem externen Speicher langfristig aufbewahrt werden soll. Damit der Wert einer Variablen auf einem externen Speicher gespeichert wird, sind besondere Maßnahmen erforderlich, die von der verwendeten Programmiersprache abhängen.

Lebensdauer

Variablen und Konstanten ...
- bestehen aus einem Typ, einem Bezeichner und einem Wert (dargestellt durch ein Literal).
- müssen zuerst initialisiert werden, danach kann beliebig oft lesend (Konstante und Variable) und schreibend (Variable) darauf zugegriffen werden.
- besitzen einen Sichtbarkeits- bzw. Gültigkeitsbereich und eine Lebensdauer.

2.5.2 Zuweisung und Ausdrücke *

Einer Variablen wird durch eine Zuweisung *(assignment)* einmalig (Konstante) oder mehrmals ein Wert zugeordnet, d. h. dieser Wert wird in die entsprechende Speicherzelle eingetragen. Auf der rechten Seite einer Zuweisung kann ein Ausdruck *(expression)* stehen, in dem der Wert – unter Berücksichtigung der Operatorprioritäten – berechnet wird, der anschließend der Variablen auf der linken Seite zugewiesen wird. Im Gegensatz zur mathematischen Formelschreibweise werden Ausdrücke in linearer Form hingeschrieben, so dass sie auf eine Schreibzeile passen.

Zuweisung

Einer Variablen wird durch eine sogenannte **Zuweisung** *(assignment)* ein **Wert** zugewiesen, d. h. in den zugehörigen Speicherplatz wird der Wert eingetragen. Man spricht bisweilen auch von einer Zuweisungsoperation, Wertzuweisung, Ersetzung oder Substitution.

:= oder =

In einigen Programmiersprachen wird als Zuweisungszeichen ein := verwendet, um zu verdeutlichen, dass es sich um keine Gleichsetzung handelt, wie es ein =-Zeichen nahelegt. Um den Schreibaufwand zu reduzieren, verwenden Sprachen wie Java und C++ nur das Gleichheitszeichen. Hier müssen Sie jedoch darauf achten, dass Sie nicht = (Zuweisung) und == (Vergleich auf Gleichheit) verwechseln!

Schreibender Zugriff

Ein schreibender Zugriff wird durch eine Zuweisung angegeben.

Beispiel

Die Abb. 2.5-2 zeigt, wie durch eine sogenannte Zuweisung in die Speicherzelle `dienstjahre` der ganzzahlige Wert `15` eingetragen wird. Eine solche Zuweisung wird folgendermaßen gelesen:
»`dienstjahre` ergibt sich zu `15`« oder
»`dienstjahre` wird `15` zugewiesen« oder
»`dienstjahre` sei `15`«.

Befand sich in der Speicherzelle `dienstjahre` bereits ein Wert, dann wird er durch eine Zuweisung automatisch gelöscht, d. h. der alte Wert wird durch den neuen überschrieben.

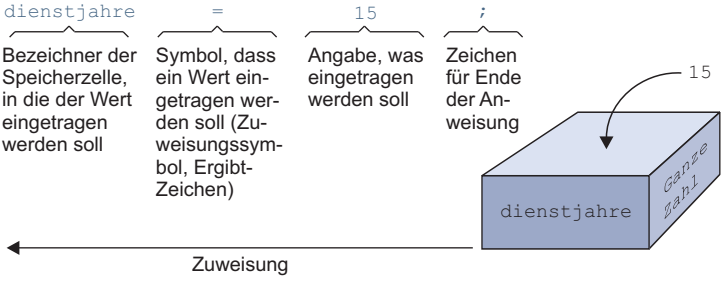


Abb. 2.5-2: Bei einer Zuweisung wird ein Wert in die Speicherzelle eingetragen, deren Bezeichner auf der linken Seite des Zuweisungszeichens angegeben ist.

Schreibender & lesender Zugriff

Auf der rechten Seite einer Zuweisung kann nicht nur ein Wert, sondern auch ein (mathematischer) **Ausdruck** stehen, der Werte miteinander verknüpft. Die Werte werden aus den Speicherzellen der aufgeführten Bezeichner gelesen und die angegebenen Operationen ausgeführt. Das Ergebnis des Ausdrucks wird dann dem Bezeichner, genauer gesagt der zugeordneten Speicherzelle, auf der linken Seite der Zuweisung zugeordnet. Ein Wert in einem Ausdruck kann auch ein **Literal** sein.

Beispiel

Die Abb. 2.5-3 zeigt auf der rechten Seite der Zuweisung einen Ausdruck, der aus drei Operanden und zwei Operationen besteht.

Auf die Speicherzellen `dienstjahre` und `grundpraemie` wird jeweils einmal lesend zugegriffen. Der Wert 3 steht als Literal in der Formel. Im Prozessor des Computersystems werden die gelesenen Werte entsprechend den Operatoren (eine Multiplikation, eine Addition) miteinander verknüpft und dann das Ergebnis in die Speicherzelle `praemie` eingetragen.

Auf alle Variablen, die rechts vom Zuweisungssymbol (=) stehen, wird immer nur lesend zugegriffen, auf die links vom Zuweisungssymbol stehende Variable immer schreibend. Bevor das erste Mal lesend auf eine Variable zugegriffen wird, muss sichergestellt sein, dass der Variablen bereits ein Wert zugewiesen worden ist. In Java ist es auch erlaubt, auf der rechten Seite einer Zuweisung eine weitere Zuweisung einzubetten, z. B. `a = (b = b + 1);` oder abgekürzt `a = ++b;`. In solchen Fällen werden zuerst die Zuweisungen auf der rechten Seite ausgeführt und dann auf die Ergebnisse lesend zugegriffen.

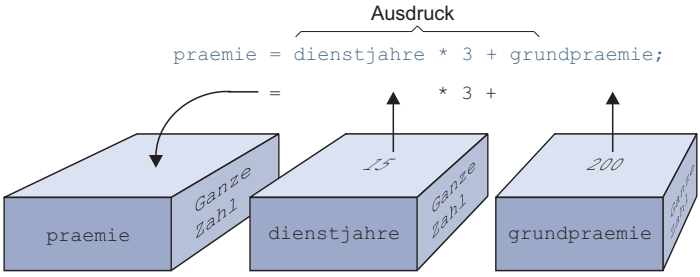


Abb. 2.5-3: Die auf der rechten Seite der Zuweisung angegebenen Variablenwerte werden aus den Speicherzellen ausgelesen und entsprechend den angegebenen Operationen * und + miteinander verknüpft. Der Wert 3 ist direkt als Literal im Ausdruck angegeben.

Lesen & schreiben auf dieselbe Variable

In der Programmierpraxis ist es oft erforderlich, den Wert einer Variablen herauf- oder herunterzuzählen.

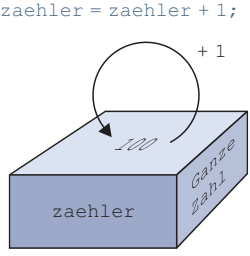


Abb. 2.5-4: Eine Zählervariable wird um den Wert Eins erhöht, indem der vorhandene Wert gelesen, um Eins erhöht und das Ergebnis wieder in die Speicherzelle geschrieben wird.

Beispiel

Die Abb. 2.5-4 zeigt, wie die Variable `zaehler` durch eine Zuweisung mit Ausdruck auf der rechten Seite um `1` erhöht wird. Zuerst wird der Wert 100 aus der Speicherzelle `zaehler` gelesen, dann eine Eins hinzu addiert und der neue Wert in die Speicherzelle `zaehler` zurückgeschrieben.

Zuweisungen mit oder ohne Ausdrücke bezeichnet man als einfache oder elementare **Anweisungen** *(statements)*.

Ausdrücke

In den bisherigen Beispielen wurden folgende Anweisungen verwendet:

```
1 dienstjahre = 15;
2 praemie = dienstjahre * 3 + grundpraemie;
3 zaehler = zaehler + 1;
```

Bevor die Zuweisung (=) ausgeführt werden kann, müssen in den Fällen 2 und 3 zunächst die Operationen ausgeführt werden (hier: * und +).

Die Zuweisung unterscheidet sich von den anderen Operationen dadurch, dass einer Variablen ein Wert zugewiesen wird, während bei den anderen Operationen jeweils zwei Werte verknüpft werden.

Rechts des Zuweisungszeichens steht ein Ausdruck, der abgearbeitet werden muss, bevor die Zuweisung ausgeführt werden kann. Der ausgewertete Ausdruck liefert als Ergebnis einen Wert, der dann durch die Zuweisung einer Variablen zugeordnet wird. Ein Ausdruck ist also nichts anderes als eine Verarbeitungsvorschrift zum Ermitteln eines Wertes. Ein Ausdruck setzt sich aus Operanden, d. h. Variablen und Konstanten, und Operatoren zusammen. Jeder Operand kann selbst wieder ein Ausdruck sein. Kommt in einem Ausdruck mehr als ein Operator vor, so muss die Reihenfolge der Ausführung definiert sein.

Prioritäten

Dies geschieht durch festgelegte Vorrangregeln – Prioritäten – für die Ausführungsreihenfolge der Operatoren. In den meisten Programmiersprachen gelten die in der Mathematik üblichen Prioritäten, d. h. Punkt vor Strich: Zuerst Multiplikation und Division, dann Addition und Subtraktion.

Um eine andere Ausführungsreihenfolge zu erhalten, können **Ausdrücke in Klammern** eingeschlossen werden. Geklammerte Ausdrücke werden immer zuerst ausgewertet.

Schachtelung von Ausdrücken

Ausdrücke darf man beliebig ineinander schachteln. Dadurch können sehr komplizierte Ausdrücke entstehen. Die Schachtelungsstruktur wird durch runde Klammern gekennzeichnet.

Lineare Notation

Alle Ausdrücke werden in **linearer Notation** geschrieben, d. h. sie werden in eine Form gebracht, sodass sie in einer Schreibzeile dargestellt werden können.

Beispiele

Die Tab. 2.5-2 zeigt einige Beispiele in mathematischer Schreibweise und die äquivalente lineare Schreibweise.

Mathematische Schreibweise	**Lineare Schreibweise**
$\frac{k \cdot t \cdot p}{100 \cdot 360}$	k * t * p / (100 * 360)
$\frac{a \cdot f + c \cdot d}{a \cdot e - b \cdot d}$	(a * f + c * d) / (a * e – b * d)
$a + \frac{b}{d + \frac{e}{f + \frac{g}{h}}}$	a + b / (d + e / (f + g / h))
$b_0 \left(1 - n \frac{p}{100}\right)$	b0 * (1 – n * p / 100)

Tab. 2.5-2: Mathematische vs. lineare Schreibweise.

Durch die angegebenen Prioritätsregeln (Punkt vor Strich) können Klammern eingespart werden (siehe auch »Operatorprioritä-

ten«, S. 89). Wenn Sie unsicher sind, dann setzen Sie lieber eine Klammer zu viel als eine Klammer zu wenig.

Beispiele

Die Tab. 2.5-3 zeigt einige Ausdrücke mit Klammern und die äquivalenten Ausdrücke ohne Klammern.

Ausdrücke mit Klammern	Äquivalente Ausdrücke, Klammern eingespart
(8 * 5) + 3	8 * 5 + 3
a + (b / c)	a + b / c
((a / b) / (c / d))	(a / b) /(c/ d)
(a + b) / (c + d)	(a + b) / (c + d)
a + (b / c) + d	a + b / c + d

Tab. 2.5-3: Ausdrücke mit und ohne Klammern.

Regel

Zuerst schreiben, dann lesen: Vor dem ersten lesenden Zugriff muss eine Variable immer initialisiert werden.

Einfache Anweisungen *(statements)* und Ausdrücke *(expressions)*

- Syntax einer einfachen Anweisung:
  `Variablenbezeichner = Ausdruck;`
- Ein Ausdruck wird linear aufgeschrieben.
- Ein Ausdruck wird von links nach rechts unter Beachtung der Prioritäten berechnet; Prioritäten können durch Klammerung () gesteuert werden.
- Das Ergebnis wird in der Variablen auf der linken Seite gespeichert.

2.5.3 Java-Programm mit lokalen Variablen und einfachen Anweisungen *

In Java müssen alle Variablen vor der ersten Verwendung deklariert werden *(field declaration)*. In der Deklaration muss vor dem Variablennamen der Typ angegeben werden. Außerdem kann eine Initialisierung vorgenommen werden. Konstante werden durch das Schlüsselwort `final` vor der Typangabe gekennzeichnet. Wichtige Typen sind `String`, `int`, `char` und `float`. Mit der Konkatenationsoperation `+` können Zeichenketten zusammengefügt werden. Mit der Ausgabeanweisung `System.out.println(...)` können Texte und Variableninhalte in das Konsolenfenster ausgegeben werden – auch in Kombination.

Deklaration von Variablen

In Java müssen alle **Variablen** vor ihrer *ersten* Verwendung deklariert bzw. vereinbart werden – in Java *field declaration* genannt. Durch die Deklaration ist der Compiler in der Lage zu prüfen, ob die jeweilige Variable in Anweisungen richtig verwendet wird.

Variablen & Typen

Für jede Variable muss in der Deklaration ein **Typ** angegeben werden. In Java sind beispielsweise folgende Typen vordefiniert:

- `String`: Zeichenketten (Achtung: großes S), Schreibweise eines **Literals**, z. B. `"Text"`
- `char`: einzelne Zeichen, Schreibweise eines Literals, z. B. `'A'`
- `int`: ganze Zahlen, Schreibweise eines Literals, z. B. `123`
- `double`: doppeltgenaue Gleitkommazahlen, Schreibweise eines Literals, z. B. `123.58`

Zuweisung

Als **Zuweisungs**-Symbol wird in Java das Gleichheitszeichen `=` verwendet, für Abfragen auf Gleichheit das doppelte Gleichheitszeichen `==`.

Konkatenation

Mit Hilfe der **Konkatenations**-Operation `+` können in Java zwei Zeichenketten zu einer Zeichenkette zusammengefügt werden.

Beispiel

```
String text1 = "Java";
String text2 = "ist toll!";
String werbung;
werbung = text1 + " " + text2;
// 1 Leerzeichen wird zwischen die Texte durch " " eingefügt
```

Syntax

Die Typangaben stehen vor dem Variablenbezeichner. Jede Variablendeklaration wird durch ein Semikolon abgeschlossen. Besitzen mehrere Variable denselben Typ, dann können als Kurzschreibweise mehrere Variablenbezeichner, getrennt durch Kommata, hinter der Typangabe aufgeführt werden. Außerdem ist es möglich, jeder Variablen einen Voreinstellungs- bzw. Initialisierungswert zuzuweisen. Der jeweilige Wert steht hinter der Variablen, getrennt durch ein Zuweisungszeichen `=`. **Konstanten** werden durch das Schlüsselwort `final` gekennzeichnet, das vor dem Typ angegeben wird. Die Bezeichner von Konstanten können innerhalb von Methoden in Klein- *oder* Großbuchstaben geschrieben werden. Außerhalb von Methoden (siehe »Datenabstraktion: Gemeinsame Daten«, S. 252) sind sie in Großbuchstaben zu schreiben (Java-Konvention), mehrere Worte durch Unterstreichstriche getrennt.

Beispiele

```
// Konstante mit Initialisierung:
final int PLZ_UNI_BOCHUM = 44780;
// Variable mit Initialisierung:
int grundpraemie = 200;
// 2 Variablen vom selben Typ, Kurzschreibweise:
double gewicht, groesse;
```

```
// 2 Variablen mit unterschiedlichen Typen:
int zaehler; String zeichenkette;
// Variable mit Initialisierung:
String text = "Dies ist eine Zeichenkette";
```


Ausgabe in das Konsolenfenster

Mit der Ausgabeanweisung `System.out.println(...);` kann eine Textzeile in das Konsolenfenster ausgegeben werden. Nach jeder Ausgabe erfolgt ein Zeilenvorschub in die nächste Zeile. Zwischen den runden Klammern kann eine Zeichenkette, eingeschlossen in "...", stehen. Es kann aber auch der Inhalt, d. h. der Wert, einer Variablen ausgegeben werden. Um dies zu erreichen, gibt man den Variablenbezeichner an. Mit Hilfe der Konkatenationsoperation + ist es möglich, Texte und Variableninhalte miteinander zu kombinieren, z. B.

```
System.out.println("Dieser Artikel kostet: "
        + buchPreis + " Euro");
```

Beispiel BMI

Das folgende Java-Programm berechnet für fest einprogrammierte Werte von Gewicht und Größe den **BMI** *(Body Mass Index)*. Der BMI ist zurzeit die anerkannte Methode, um Über- oder Untergewicht festzustellen. Er wird berechnet nach der Formel: (*Gewicht*/kg) / (*Größe*/m)^2. Der BMI gilt gleichermaßen für Frauen und Männer. Das Idealgewicht liegt bei einem BMI zwischen 20 und 24 vor. Ein BMI zwischen 25 und 30 zeigt ein leichtes Übergewicht, ein BMI über 30 zeigt Fettsucht an. Ein BMI unter 18 gilt als Untergewicht.

BMI1

```
/***********************************************
Programm zur Berechnung des BMI (Body Mass Index)
***********************************************/
public class BMI1
{
 public static void main (String args[])
 {
   double koerperGewicht, koerperGroesse, bmi; //lokale Vari.
   koerperGewicht = 61;   //Zuweisung
   koerperGroesse = 1.68; //Zuweisung
   //Ausdruck
   bmi = koerperGewicht / (koerperGroesse * koerperGroesse);
   System.out.println
      ("Ihr Gewicht von " + koerperGewicht + " kg");
   System.out.println
      ("und Ihre Groesse von " + koerperGroesse + " m");
   System.out.println("ergeben einen BMI von " + bmi);
   System.out.println
     ("Die hoechste Lebenserwartung haben Menschen");
   System.out.println("mit einem BMI zwischen 20 und 24");
 }
}
```

Die Abb. 2.5-5 zeigt die Übersetzung und Ausführung dieses Programms im Konsolenfenster.

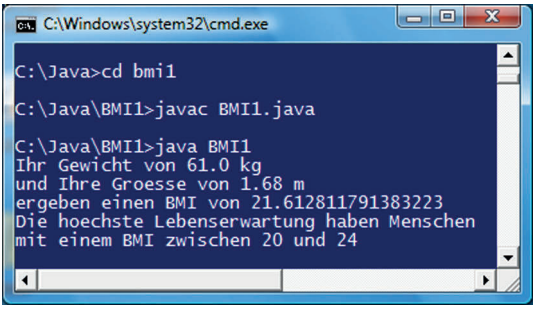


Abb. 2.5-5: So sieht die Übersetzung und Ausführung des Java-Programms BMI1 in einem Konsolenfenster aus.

Tragen Sie bei Gewicht und Größe Ihre Daten ein, erfassen Sie das Programm mit einem Texteditor, übersetzen Sie es und führen Sie es aus.

2.5.4 Java-Programme mit Konsoleneingabe *

Eine Ausgabe in das Konsolenfenster kann in Java über die Anweisung `System.out.println()` erfolgen. Die Eingabe von Informationen über das Konsolenfenster ist nicht so einfach, da die Eingabe vom jeweiligen Eingabetyp abhängt. Außerdem müssen Überprüfungen auf fehlerhafte Eingaben vorgenommen werden. Durch das Hinzufügen von Methoden zum eigenen Programm können Texte, Gleitkommazahlen und ganze Zahlen eingelesen werden.

In vielen Fällen soll ein Programm nicht nur Informationen an den Benutzer ausgeben, sondern das Programm erwartet auch Eingaben vom Benutzer.

Konsolenfenster Ausgabe

In Java ist es mit der Anweisung `System.out.println();` einfach, Texte und Variablenwerte in ein Konsolenfenster auszugeben.

Konsolenfenster Eingabe

Die Eingabe gestaltet sich *nicht* so einfach, da es davon abhängt, von welchem Typ der einzulesende Wert ist. Außerdem müssen fehlerhafte Eingaben, z. B. Texteingabe statt Zahleingabe, abgefangen werden. Um die Eingabe möglichst einfach handhaben zu können, verkapselt man die Eingabe durch eine oder mehrere Methoden.

Eingabe von Gleitpunktzahlen

Die Anweisung `gleitpunktzahl = readDoubleComma();` liest eine Gleitpunktzahl von der Konsole und weist sie der Variablen `gleitpunktzahl` zu, wobei `gleitpunktzahl` vom Typ `double` sein muss. Bei der Eingabe müssen die Nachkommastellen durch ein Komma getrennt werden (deutsche Notation).

Wollen Sie diese Anweisung nutzen, dann müssen Sie folgende Methode in Ihr Programm einfügen – wobei die Wirkungsweise dieser Methode hier nicht erklärt wird (nehmen Sie sie als gegeben hin):

```
/**Liest eine Gleitpunktzahl vom Typ double von der Konsole
  * Deutsche Notation: Trennung der
  * Nachkommastellen durch Komma
  * @return Gleitpunktzahl vom Typ double
  * @exception InputMismatchException:
  *      Die Eingabe entspricht nicht dem Typ.
  * @exception NoSuchElementException:
  *      Es wurde keine Eingabezeile gefunden.
  * @exception IllegalStateException:
  *      Die verwendete Methode ist nicht geöffnet.
  */
 public static double readDoubleComma() throws
   InputMismatchException, NoSuchElementException,
   IllegalStateException
 {
    Locale.setDefault(Locale.GERMAN);
    return new Scanner(System.in).nextDouble();
 }
```

Zusätzlich müssen Sie vor Ihre Klasse noch folgenden Import-Befehl schreiben:

```
import java.util.*; //Importieren der Bibliothek util
```

Hinweis

Die obige Methode nimmt selbst keine Überprüfungen der Benutzereingabe vor. Dies ist die Aufgabe des nutzenden Programms. Wie dies geschieht wird im Kapitel »Behandlung von Ausnahmen«, S. 157, gezeigt.

Beispiel BMI2

Im folgenden Beispiel wird der **BMI** berechnet. Die notwendigen Eingabewerte werden über die Konsole eingegeben. Das Programm sieht folgendermaßen aus, wobei die wichtigen Teile fett hervorgehoben sind:

```
/************************************************
Programm zur Berechnung des BMI (Body Mass Index)
Eingabewerte werden ueber die Konsole eingelesen
************************************************/
import java.util.*; //Importieren der Bibliothek util

public class BMI2
{
// Hilfsmethode zur Eingabe ////////////////////////////////
  /**Liest eine Gleitpunktzahl vom Typ double von der Konsole
  * Deutsche Notation: Trennung der
  * Nachkommastellen durch Komma
  * ... (wie oben)
  */
 public static double readDoubleComma() throws
   InputMismatchException, NoSuchElementException,
```

```
    IllegalStateException
  {
     Locale.setDefault(Locale.GERMAN);
     return new Scanner(System.in).nextDouble();
  }
  // Ende Hilfsmethode zur Eingabe ////////////////////////////

  //Berechnet den BMI
  public static void main (String args[])
  {
    double bmi; //lokale Variablen
    double koerperGewicht = 0.0;
    double koerperGroesse = 0.0;
    System.out.println
     ("Geben Sie bitte Ihr Gewicht in kg ein:");
    //Aufruf der Methode readDoubleComma()
    koerperGewicht = readDoubleComma();
    System.out.println
     ("Geben Sie bitte Ihre Groesse in m ein:");
    //Aufruf der Methode readDoubleComma()
    koerperGroesse = readDoubleComma();

    bmi = koerperGewicht / (koerperGroesse * koerperGroesse);
    System.out.println
       ("Ihr Gewicht von " + koerperGewicht + " kg");
    System.out.println
       ("und Ihre Groesse von " + koerperGroesse + " m");
    System.out.println("ergeben einen BMI von " + bmi);
    System.out.println
       ("Die hoechste Lebenserwartung haben Menschen");
    System.out.println("mit einem BMI zwischen 20 und 24");
  }
}
```

Die Abb. 2.5-6 zeigt die Übersetzung und beispielhafte Ausführung des Programms.

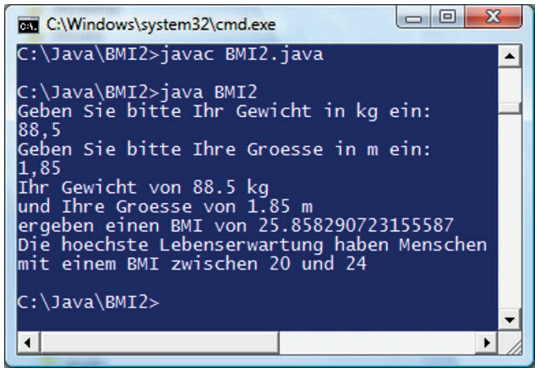


Abb. 2.5-6: So sieht die Übersetzung und die Ausführung des Programms BMI2 mit einer Eingabemethode aus.

Ersetzen sie in dem Programm BMI2 die Zeile
`Locale.setDefault(Locale.GERMAN);` durch folgende Zeile
`Locale.setDefault(Locale.ENGLISH);`
Übersetzen Sie das Programm und führen Sie es erneut aus. Was stellen Sie fest?

2

Beispiel MWST

MWST

Aus einem eingegebenen Bruttobetrag soll der Nettobetrag und die MWST berechnet werden. Das Java-Programm sieht folgendermaßen aus:

```
/***********************************************
Programm zur Berechnung der MWST
Der Bruttowert wird ueber die Konsole eingelesen
Ausgegeben werden der Nettowert und die MWST
***********************************************/
import java.util.*; //Importieren der Bibliothek util
public class MWST
{
 // Hilfsmethode zur Eingabe ////////////////////
 // analog wie im Programm BMI2
 // Ende Hilfsmethode zur Eingabe /////////////////

// Berechnet die MWST
 public static void main (String args[])
 {
   final double VOLLE_MWST = 19.0;
   double netto, brutto, mwstBetrag;
   System.out.println
    ("Geben Sie bitte den Bruttobetrag ein:");
   brutto = readDoubleComma(); //Einlesen
   netto = brutto * 100.0 / (VOLLE_MWST + 100.0);
   mwstBetrag = brutto * VOLLE_MWST / (VOLLE_MWST + 100.0);
   System.out.println("Brutto " + brutto + " Euro");
   System.out.println("Netto " + netto + " Euro");
   System.out.println("MWST " + mwstBetrag + " Euro");
 }
}
```

Die Abb. 2.5-7 zeigt die Ein- und Ausgabe im Konsolenfenster für ein Beispiel.

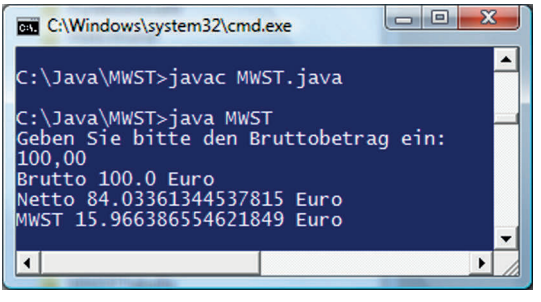


Abb. 2.5-7: So sieht die Ein- und Ausgabe des Java-Programms MWST aus.

Geldbeträge ganzzahlig verarbeiten Hinweis

Wird mit Gleitpunktzahlen gerechnet, dann können wegen dem beschränkten Speicherplatz sogenannte Rundungsfehler auftreten (siehe »Rechengenauigkeit mit Gleitpunkt-Zahlen«, S. 77). Um solche Fehler zu vermeiden, werden Geldbeträge in der Praxis mit ganzen Zahlen berechnet, d. h. statt mit Euro wird mit Cent gerechnet. Für noch genauere Berechnungen gibt es in Java die Klasse `BigDecimal`.

2.5.5 Java-Pakete anlegen und benutzen: das Wichtigste *

Bereits vorhandene Java-Programme (Klassen) können in Paketen *(packages)* zusammengefasst werden. *Ein* Programm in einem Paket kann dann mit der Anweisung `import paketname.programmname;` in ein eigenes Programm importiert und benutzt werden. *Alle* Programme eines Paketes werden durch die Anweisung `import paketname.*;` importiert. Pakete können ineinander geschachtelt werden. Sollen eigene Programme in Paketen zur Verfügung gestellt werden, dann wird vor das Programm die Anweisung `package paketname;` geschrieben. Der Name des Ordners, in dem die Programmdateien abgelegt werden, muss dabei mit dem Paketnamen identisch sein.

Ein grundlegendes Prinzip in der Programmierung besteht darin, Programme, die es bereits gibt, zu benutzen und *nicht* neu selbst zu schreiben. In Java ist jedes Programm eine sogenannte Klasse.

Klassen, die in anderen Programmen benutzt werden sollen, werden in Java zu sogenannten **Paketen** *(package)* zusammengefasst. Solche Pakete können dann in eigene Programme importiert werden. Pakete

Um Programme nutzen zu können, die sich in Paketen befinden, müssen am Anfang eines Java-Programms ein oder mehrere sogenannte `import`-Anweisungen stehen. Hinter dem Schlüsselwort `import` steht der Paketname, in der Regel gefolgt durch einen Punkt und einem Stern. Der Stern gibt an, dass *alle* Programme, genauer gesagt alle Klassen des Pakets, benutzt werden sollen. `import`-Anweisung

In Java ist es erlaubt und sogar der Standardfall, dass Pakete ineinander geschachtelt werden. Zu Java gehören eine Vielzahl vordefinierter Pakete, die mit dem **JDK** ausgeliefert werden. Das umfassendste Paket heißt `java`. Darin befinden sich dann die anderen Pakete. Wollen Sie die Programme eines dieser Pakete benutzen, dann müssen Sie es importieren. Geschachtelte Pakete

Beispiele

```
import java.util.*; //Importieren des Java-Pakets util
//Das Paket util befindet sich innerhalb des Pakets java

//Importieren des selbst geschriebenen Pakets inout
import inout.*;
```


Konvention

Paketnamen enthalten ausschließlich Kleinbuchstaben, z. B. `util`, `math`, `inout`.

Eigenes Paket

Wenn Sie eigene Programme haben, die Sie anderen zur Verfügung stellen wollen oder die Sie selbst öfters benötigen, dann können Sie diese Programme in ein selbst definiertes Paket legen. Dazu ist Folgendes zu tun:

1. Schritt

Sie legen einen Ordner mit dem gewünschten Dateinamen an, z. B. `inout`.

2. Schritt
Paketname = Ordnername!

Sie legen die Programme, die Sie in diesem Paket zur Verfügung stellen wollen, in diesen Ordner. Bei den Programmen muss es sich um Klassen handeln. Als erste Anweisung schreiben Sie in die Klasse das Schlüsselwort `package` gefolgt von den von Ihnen gewählten Paketnamen. Der Paketname muss dabei mit dem Ordnernamen übereinstimmen. Sie übersetzen jede Klasse in diesem Ordner.

3. Schritt

Sie legen diesen Ordner z.B. in einen Ordner, der alle Ihre Pakete enthält, z.B. `JavaPakete`. Sie haben jetzt folgende Ordnerhierarchie: `JavaPakete` ↳ `inout`. Setzen Sie den CLASSPATH auf den Ordner `JavaPakete`.

4. Schritt

Sie schreiben in Ihr Programm, das das Paket verwenden soll, eine `import`-Anweisung mit dem Paketnamen. Jetzt können Sie innerhalb Ihres Programms die Klassen des Pakets verwenden.

Beispiel

Ein Freund hat Ihnen mehrere Methoden für die Eingabe von Text und Zahlen von der Konsole geschrieben. Da Sie diese Methoden öfters benötigen, wollen Sie diese Methoden *nicht* jedes Mal in Ihr Programm einbetten, sondern in ein Paket auslagern. Sie legen einen Ordner mit dem Namen `inout` an. Die Methoden Ihres Freundes legen Sie in eine Klasse mit dem Namen `Console`, wobei Sie an den Anfang der Klasse `package inout;` schreiben:

```
package inout;

import java.util.Scanner;
import java.util.Calendar;
import java.util.Locale;
import java.util.Date;
import java.util.NoSuchElementException;
import java.util.InputMismatchException;
import java.util.regex.Pattern;
```

```
import java.text.ParseException;
import java.text.DateFormat;

/** Diese Klasse stellt Methoden zur Verfuegung, <br/
>* um Texte und einfache Typen von der Konsole einzulesen.<br/>
 * Die Ausnahmebehandlung ist Aufgabe des Aufrufers.<hr/>
 * @author Helmut Balzert
 * @version 2.2 / 1.06.2022
*/
public class Console
{
private static Scanner sc;

 //Unterdrückung des default-Konstruktor,
 //um eine Objekterzeugung zu verhindern
 private Console()
 {
  //Dieser Konstruktor wird nie aufgerufen
 }

 /**Liest eine Zeile von der Konsole
  * @return Eingelesene Zeile vom Typ String.
  * @exception NoSuchElementException:
  * Es wurde keine Eingabezeile gefunden.
  * @exception IllegalStateException:
  * Die verwendete Methode ist nicht geoeffnet.
 */
 public static String readString()
 throws NoSuchElementException, IllegalStateException
 {
    Scanner sc = new Scanner(System.in);
    return sc.nextLine();
 }

 /**Liest eine Zeile von der Konsole
  * @return Eingelesene Zeile vom Typ char[].
  * @exception NoSuchElementException:
  *    Es wurde keine Eingabezeile gefunden.
  * @exception IllegalStateException:
  *    Die verwendete Methode ist nicht geöffnet.
  */
 public static char[] readCharArray()
  throws NoSuchElementException, IllegalStateException
 {
    sc = new Scanner(System.in);
    String text = sc.nextLine();
    return text.toCharArray();
 }

 /**Liest einen booleschen Wert von der Konsole
  * @return Boolescher Wert true oder false.
  * @exception NoSuchElementException:
  * Es wurde keine Eingabezeile gefunden.
  * @exception IllegalStateException:
  * Die verwendete Methode ist nicht geoeffnet.
```

```
 * @exception InputMismatchException:
 *    Die Eingabe entspricht nicht dem Typ.
 */
public static boolean readBoolean() throws
  InputMismatchException, NoSuchElementException,
  IllegalStateException
{
   sc = new Scanner(System.in);
   return sc.nextBoolean();
}

/**Liest eine ganze Zahl vom Typ int von der Konsole
 * @return Ganze Zahl vom Typ int.
 * @exception InputMismatchException:
 * Die Eingabe entspricht nicht dem Typ.
 * @exception NoSuchElementException:
 * Es wurde keine Eingabezeile gefunden.
 * @exception IllegalStateException:
 * Die verwendete Methode ist nicht geoeffnet. */
public static int readInt() throws

  InputMismatchException, NoSuchElementException,
  IllegalStateException
{
   return new Scanner(System.in).nextInt();
}

/**Liest eine ganze Zahl vom Typ long von der Konsole
 * @return Ganze Zahl vom Typ long
 * @exception InputMismatchException:
 * Die Eingabe entspricht nicht dem Typ.
 * @exception NoSuchElementException:
 * Es wurde keine Eingabezeile gefunden.
 * @exception IllegalStateException:
 * Die verwendete Methode ist nicht geoeffnet. */
public static long readLong() throws

  InputMismatchException, NoSuchElementException,
  IllegalStateException
{
   return new Scanner(System.in).nextLong();
}

/**Liest eine Gleitpunktzahl vom Typ float von der Konsole
 * Englische Notation: Trennung der
 * Nachkommastellen durch Punkt
 * @return Gleitpunktzahl vom Typ float
 * @exception InputMismatchException:
 * Die Eingabe entspricht nicht dem Typ.
 * @exception NoSuchElementException:
 * Es wurde keine Eingabezeile gefunden.
 * @exception IllegalStateException:
 * Die verwendete Methode ist nicht geoeffnet. */
 public static float readFloatPoint() throws
```

```
  InputMismatchException, NoSuchElementException,
  IllegalStateException
{
   Locale.setDefault(Locale.ENGLISH);
   return new Scanner(System.in).nextFloat();
}

/**Liest eine Gleitpunktzahl vom Typ float von der Konsole
 * Deutsche Notation: Trennung der
 * Nachkommastellen durch Komma
 * @return Gleitpunktzahl vom Typ float
 * @exception InputMismatchException:
 * Die Eingabe entspricht nicht dem Typ.
 * @exception NoSuchElementException:
 * Es wurde keine Eingabezeile gefunden.
 * @exception IllegalStateException:
 * Die verwendete Methode ist nicht geoeffnet. */
public static float readFloatComma() throws

  InputMismatchException, NoSuchElementException,
  IllegalStateException
{
   Locale.setDefault(Locale.GERMAN);
   return new Scanner(System.in).nextFloat();
}

/**Liest eine Gleitpunktzahl vom Typ double von der Konsole
 * Englische Notation: Trennung der
 * Nachkommastellen durch Punkt
 * @return Gleitpunktzahl vom Typ double
 * @exception InputMismatchException:
 * Die Eingabe entspricht nicht dem Typ.
 * @exception NoSuchElementException:
 * Es wurde keine Eingabezeile gefunden.
 * @exception IllegalStateException:
 * Die verwendete Methode ist nicht geoeffnet. */
public static double readDoublePoint() throws

  InputMismatchException, NoSuchElementException,
  IllegalStateException
{
   Locale.setDefault(Locale.ENGLISH);
   return new Scanner(System.in).nextDouble();
}

/**Liest eine Gleitpunktzahl vom Typ double von der Konsole
 * Deutsche Notation: Trennung der
 * Nachkommastellen durch Komma
 * @return Gleitpunktzahl vom Typ double
 * @exception InputMismatchException:
 * Die Eingabe entspricht nicht dem Typ.
 * @exception NoSuchElementException:
 * Es wurde keine Eingabezeile gefunden.
 * @exception IllegalStateException:
 * Die verwendete Methode ist nicht geoeffnet.
```

```
  */
 public static double readDoubleComma() throws
   InputMismatchException, NoSuchElementException,
   IllegalStateException
 {
    Locale.setDefault(Locale.GERMAN);
    return new Scanner(System.in).nextDouble();
 }

/**Liest ein Zeichen vom Typ char von der Konsole
 * @return Erstes eingegebene Zeichen vom Typ char.
 * @exception NoSuchElementException:
 *    Es wurde keine Eingabezeile gefunden.
 */
 public static char readChar() throws
   NoSuchElementException,IllegalStateException
 {
    String s = new Scanner(System.in).next();
    return s.charAt(0);
 }

}
```

Legen Sie für diese Klasse eine Datei `Console.java` in dem Ordner `inout` an und übersetzen Sie diese. In Ihr Programm `BMI3` importieren Sie nun dieses Paket:

BMI3

```
/************************************************
Programm zur Berechnung des BMI (Body Mass Index)
Eingabewerte werden über die Konsole eingelesen
************************************************/

//Importieren des Pakets inout mit der Klasse Console
import inout.Console;

public class BMI3
{
 //Berechnet den BMI
 public static void main (String args[])
 {
   double bmi; //lokale Variablen
   double koerperGewicht = 0.0;
   double koerperGroesse = 0.0;
   System.out.println
    ("Geben Sie bitte Ihr Gewicht in kg ein:");
   //Aufruf der Methode readDoubleComma()
   koerperGewicht = Console.readDoubleComma();
   System.out.println
    ("Geben Sie bitte Ihre Groesse in m ein:");
   //Aufruf der Methode readDoubleComma()
   koerperGroesse = Console.readDoubleComma();

   bmi = koerperGewicht / (koerperGroesse * koerperGroesse);
   System.out.println
     ("Ihr Gewicht von " + koerperGewicht + " kg");
   System.out.println
```

```
      ("und Ihre Groesse von " + koerperGroesse + " m");
    System.out.println("ergeben einen BMI von " + bmi);
    System.out.println
      ("Die hoechste Lebenserwartung haben Menschen");
    System.out.println("mit einem BMI zwischen 20 und 24");
  }
}
```

Sie können nun alle Methoden der Klasse `Console` in Ihrem Programm benutzen. In diesem Beispiel muss vor die Methodennamen noch der Klassenname `Console`, getrennt durch einen Punkt, stehen.

Beispiel

Sie haben das Paket `inout` in den Ordner `JavaPakete` gelegt. Sie müssen dann folgenden `CLASSPATH` setzen:

```
C:\JavaPakete
```

wenn sich der Ordner `JavaPakete` auf der obersten Ebene im Laufwerk `C` befindet.

```
C:\JavaPakete>cd inout

C:\JavaPakete\inout>javac Console.java
```

```
C:\Java\BMI3>javac BMI3.java

C:\Java\BMI3>java BMI3
Geben Sie bitte Ihr Gewicht in kg ein:
90,5
Geben Sie bitte Ihre Groesse in m ein:
1,80
Ihr Gewicht von 90.5 kg
und Ihre Groesse von 1.8 m
ergeben einen BMI von 27.932098765432098
Die hoechste Lebenserwartung haben Menschen
mit einem BMI zwischen 20 und 24
```

Die Zusammenhänge zwischen Path, CLASSPATH und Paketen veranschaulicht die Abb. 2.5-8.

Eigene Pakete mehrfach nutzen:

- Vor eigene Klassen schreiben: `package eigenerPaketname;`
- Ordner mit Paketnamen anlegen.
- Klassen dort ablegen.
- Jede Klasse übersetzen.
- CLASSPATH auf den Ordner setzen.

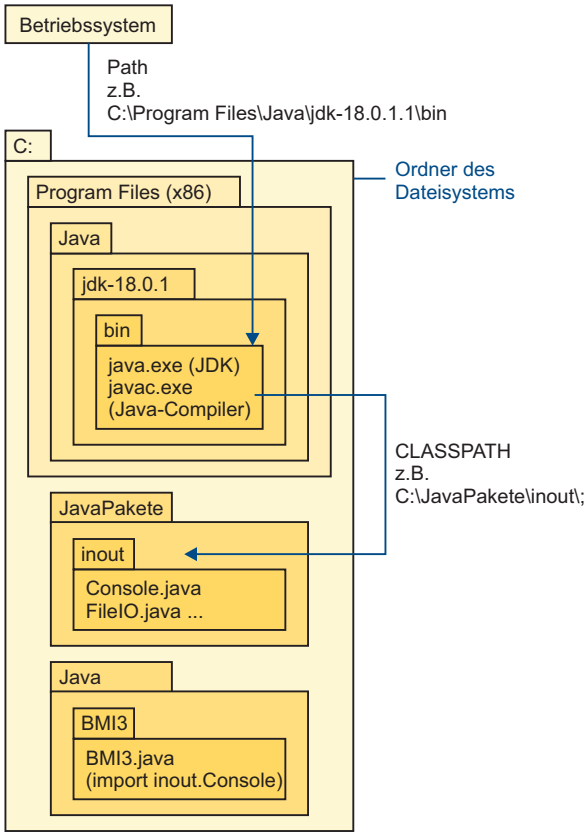


Abb. 2.5-8: Veranschaulichung von Path und CLASSPATH.

2.6 Java-Entwicklungsumgebungen *

IDEs

Mit dem **JDK** *(Java Developer Kit)* von Oracle können Sie Java-Programme entwickeln. Das JDK stellt Ihnen einen Java-Compiler und eine **JVM** *(Java virtual machine)* zur Verfügung. Für eine professionelle Softwareentwicklung bietet das JDK jedoch zu wenig Komfort. Es gibt daher von verschiedenen Herstellern sogenannte Java-Entwicklungsumgebungen – auch Java-IDEs *(Integrated Development Environments)* genannt – die den Entwickler auf vielfältige Art und Weise unterstützen.

Zum Lernen

Zum Erlernen von Java eignet sich besonders gut die Entwicklungsumgebung BlueJ, die kostenlos verfügbar ist.

Installieren Sie BlueJ auf Ihrem Computersystem und machen Sie sich mit dieser Java-Entwicklungsumgebung vertraut.

2.7 OptiTravel: Gespräch Auftraggeber – Auftragnehmer *

Die Firma ProManagement hat das Geschäftsziel, Manager bei ihren Tätigkeiten durch Serviceleistungen zu unterstützen. Da ProManagement bereits gute Erfahrungen mit der Firma WebSoft gemacht hat (siehe Box), möchte sie der Firma WebSoft einen neuen Auftrag erteilen.

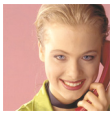
Abb. 2.7-1: Firma WebSoft

Die Firma WebSoft ist ein junges, innovatives Unternehmen, das sich auf die Erstellung von Websites spezialisiert hat, d. h. auf Software, die über das Web zu bedienen ist. Aufträge werden durch ein interdisziplinäres Team bearbeitet. Jedes Teammitglied ist auf ein Fachgebiet spezialisiert. Die Abbildung zeigt einen Teil der WebSoft-Mannschaft.

Zu einem ersten Gespräch treffen sich Herr Froh von der Firma ProManagement sowie der Projektleiter, Herr Pilot, und die Systemanalytikerin, Frau Sonnenschein, von der Firma WebSoft. Zusätzlich dabei ist Frau Jung, die als Junior-Programmiererin gerade bei der Firma WebSoft angefangen hat.

1. Gespräch

Versetzen Sie sich in die Rolle der Junior-Programmiererin Frau Jung. Dies ist Ihre Rolle!



Herr Pilot erklärt ihr vor dem Gespräch, dass es Ziel des ersten Gesprächs sei, die **Anforderungen des Kunden** anzuhören, Rückfragen zu stellen und anhand dieser Informationen an-

schließend ein sogenanntes **Pflichtenheft** zu erstellen und dem Kunden vorzulegen. Diese Aufgaben gehören zur Rolle eines **Systemanalytikers**.

Erste Anforderungen

Herr Froh erzählt, dass die Firma WebSoft eine Software **OptiTravel** entwickeln soll, die es Geschäftsreisenden ermöglicht, ihre Reisewege und Transportmittel nach Zeit- und/oder Kostengesichtspunkten optimal zu wählen. Folgende Fragen sollen einem Geschäftsreisenden beantwortet werden:

- Welche Transportmittel gibt es von einem Ort A zu einem Ort B (Auto, Zug, Flugzeug)?
- Wie viel Zeit wird für jedes Transportmittel benötigt, um von A nach B zu gelangen?
- Bei Autofahrten werden drei Varianten unterschieden: Zügige Fahrt, normale Fahrt (+ 20 % Aufschlag auf die Zeit), gemäßigte Fahrt (+ 40 % Aufschlag auf die Zeit gegenüber der zügigen Zeit).
- Bei den Straßenangaben zwischen zwei Orten ist zu vermerken, wie viel Kilometer der Strecke Autobahn ohne Geschwindigkeitsbegrenzung, Autobahn mit 130 km/h Geschwindigkeitsbeschränkung, Bundesstraße, Landstraße und Stadtverkehr sind. Da diese Angaben von der Firma ProManagement erst ermittelt werden, sind für das zu entwickelnde Programm Testwerte anzunehmen.
- Entfernungsangaben soll zusätzlich auch in Meilen angegeben werden.
- Für die Benutzung des Autos sollen zusätzlich die Kosten angegeben werden. Dafür müssen folgende Angaben gemacht werden:
  - Spritverbrauch pro 100 Kilometer
  - Benzin-/Dieselpreis pro Liter

Einschränkungen

- Um zunächst die Kundenakzeptanz bei Geschäftsreisenden zu überprüfen, soll das Software-System sich zunächst auf die zehn größten deutschen Städte (nach Einwohnerzahl) beschränken.
- Da noch nicht entschieden ist, über welches Medium (PC, Handy, Smartphone) OptiTravel dem Geschäftsreisenden angeboten werden soll, ist zunächst *keine* Benutzungsoberfläche für die Software zu konzipieren.

Erste Aufgaben

Nach dem Gespräch bittet Frau Sonnenschein die Junior-Programmiererin Frau Jung als Erstes folgende Aufgaben zu erledigen:

- Ermitteln der zehn größten deutschen Städte.
- Schreiben eines Programms zur Umrechnung von Liter in Gallonen und umgekehrt. Dabei ist das Paket `inout` für die Eingabe zu verwenden.

Frau Jung freut sich auf Ihre ersten Aufgaben und recherchiert im Internet.

10 Städte

Die zehn größten Städte nach Einwohnerzahl sind: (1) Berlin, (2) Hamburg, (3) München, (4) Köln, (5) Frankfurt am Main, (6) Dortmund, (7) Stuttgart, (8) Essen, (9) Düsseldorf, (10) Bremen.

Frau Jung druckt eine Deutschlandkarte aus und markiert diese Städte, um eine visuelle Vorstellung von den Entfernungen zu erhalten (Abb. 2.7-2).

Abb. 2.7-2: Deutschlandkarte mit den 10 größten Städten (nach Einwohnerzahl).

Liter vs. Gallone

Für die Umrechnung von Litern in Gallonen ermittelt Frau Jung folgende Umrechnungsformel:

1 US-Gallone = 3,785411784 Liter bzw. 1 Liter = 0,264172052 US-Gallonen

Sie erstellt folgendes Programm:

```
/***********************************************
Programm zur Umrechnung von Litern in US-Gallonen
und umgekehrt
Eingabewerte werden über die Konsole eingelesen
***********************************************/

import inout.Console; //Importieren des Pakets inout

public class Liter
```

```
{
 public static void main (String args[])
 {
  double liter;
  double gallonen;
  System.out.println("Geben Sie bitte die Literanzahl ein:");
  //Aufruf der Methode readDoubleComma()
  liter = Console.readDoubleComma();
  //gallonen = 0.264172052 * liter;
  gallonen = 1.0/3.785411784 * liter;
  System.out.println
    (liter + " Liter ergeben " + gallonen + " US-Gallonen");
  System.out.println("Geben Sie bitte die US-Gallonenanzahl ein:");

  gallonen = Console.readDoubleComma();
  liter = 3.785411784  * gallonen;
  System.out.println
   (gallonen + " US-Gallonen ergeben " + liter + " Liter");
 }
}
```

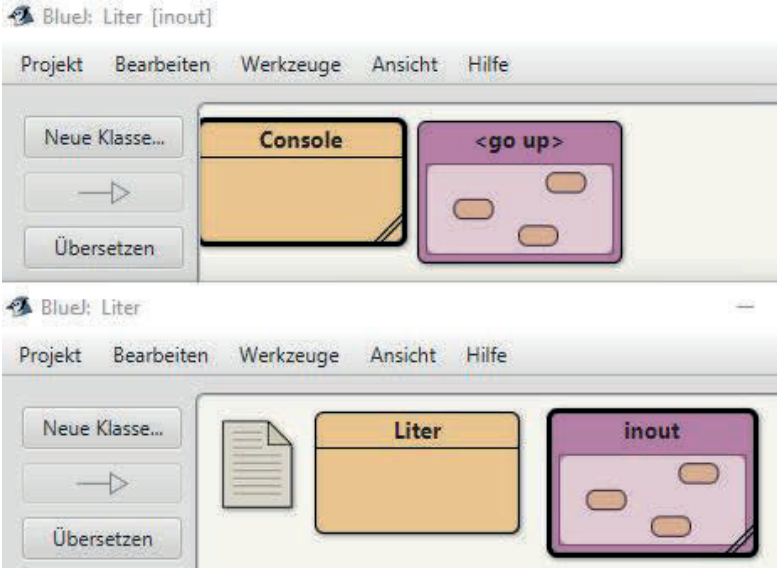
Eine beispielhafte Ausführung ergibt folgende Ausgaben:

```
Geben Sie bitte die Literanzahl ein:
15
15.0 Liter ergeben 3.962580785372227 US-Gallonen
Geben Sie bitte die US-Gallonenanzahl ein:
3,962580785372227
3.962580785372227 US-Gallonen ergeben 15.000000000000002 Liter
```

Damit hat Frau Jung Ihre ersten Aufgaben erfolgreich erledigt und wartet auf die nächsten Aufgaben.

Hinweis

Auf die Rundungsfehler, die sich bei der Berechnung ergeben, wird im Kapitel »Rechengenauigkeit mit Gleitpunkt-Zahlen«, S. 77, eingegangen.



3 Einfache Typen, ihre Werte und Operationen *

Ein grundlegendes Konzept von Programmiersprachen ist das **Typ-Konzept**. Bevor jedoch auf dieses Konzept näher eingegangen wird, ist es notwendig, sich anzusehen, wie Konzepte in einer Sprache dargestellt werden.

Syntax

Zur korrekten Beschreibung der Syntax einer Programmiersprache gibt es verschiedene Notationen. Die in diesem Buch verwendeten Notationen finden Sie hier beschrieben:

- »Java: Syntaxnotation«, S. 56

Typ

Jeder Variablen und jeder Konstanten muss in Java ein Typ zugeordnet werden. Der **Typ** legt fest, welche Werte die jeweilige Variable bzw. Konstante annehmen kann und welche Operationen auf diesen Werten ausgeführt werden können. Durch den Typ wird außerdem festgelegt, wie viel Speicherplatz, d. h. wie viele Speicherzellen, für die Werte benötigt werden.

Einfache Typen

In Java werden **einfache Typen** *(primitive types)* und Referenztypen *(reference types)* unterschieden. Hier werden die einfachen Typen behandelt.

Einfache Typen gehören in Java zum Sprachumfang und besitzen festgelegte Schlüsselwörter.

boolean

Der Typ `boolean` ist in Java – wie in den meisten anderen Programmiersprachen auch – vordefiniert. Eine Variable vom Typ `boolean` kann jeweils einen der Wahrheitswerte `true` oder `false` annehmen:

- »Der Typ boolean«, S. 61

Ganzzahlige Typen

Die Wertebereiche der ganzzahligen Typen *(integral types)* umfassen nur Teilbereiche der ganzen Zahlen ($\mathbb{Z}$). Ihre Wertebereiche und Operationen sind hier zusammengestellt:

- »Ganzzahlige Typen«, S. 64

Gleitpunkttypen

In vielen Anwendungsbereichen – insbesondere in der Mathematik – wird jedoch ein Typ benötigt, der reelle Zahlen ($\mathbb{R}$) näherungsweise wiedergibt. In Java gibt es für diese Zwecke die Gleitpunkttypen `float` und `double`, die numerisch-reelle Zahlen als Wertebereich umfassen. Die Wertebereiche und Operationen sind hier aufgeführt:

- »Gleitpunkt-Typen«, S. 70

Darstellung von Gleitpunktzahlen

Durch den beschränkten Speicherplatz können Zahlen mit Nachkommastellen oft *nicht* exakt in einem Computersystem dargestellt werden. Daher sind bei Gleitpunktzahlen eine Reihe von Besonderheiten zu beachten:

■ »Darstellung von Gleitpunkt-Zahlen«, S. 73

Rechengenauigkeit

Bei Rechenoperationen mit Gleitpunktzahlen können Rundungsfehler auftreten, die zu ungenauen Ergebnissen führen können:

■ »Rechengenauigkeit mit Gleitpunkt-Zahlen«, S. 77

Eingeschränkte Mathematikgesetze

Auch das Assoziativ- und das Distributivgesetz der Mathematik gelten auf einem Computersystem nur noch eingeschränkt:

■ »Eingeschränkte Mathematikgesetze«, S. 81

char

Der Java-Typ `char` erlaubt es, einzelne Zeichen zu speichern und zu manipulieren:

■ »Der Zeichentyp char«, S. 85

Prioritäten

Zwischen den einzelnen Operationen gibt es in Java – ähnlich wie in der Mathematik – Verarbeitungsprioritäten:

■ »Operatorprioritäten«, S. 89

Typumwandlungen

Unterschiedliche Typen bei Variablen und Konstanten in Ausdrücken können *nicht* beliebig miteinander kombiniert werden, sondern es gibt Regeln für das Zusammenspiel verschiedener Typen:

■ »Typumwandlungen«, S. 91

3.1 Java: Syntaxnotation *

Zur Beschreibung von Programmiersprachen werden Syntaxnotationen verwendet. Beliebt sind grafische Syntaxdiagramme. Kompakter ist die textuelle EBNF-Notation. In den Notationen wird zwischen nicht-terminalen Symbolen und terminalen Symbolen unterschieden.

In der Informatik haben sich verschiedene Beschreibungsformalismen für die Syntax von Programmiersprachen eingebürgert. Mithilfe solcher Syntaxbeschreibungen ist es einfacher, eine Sprache zu definieren und ein Programm zu analysieren. Insbesondere dient die Darstellung der einzelnen Sprachkonstrukte auch dazu, dem Programmierer beim Nachschlagen, wie die korrekte Syntax aussieht, zu helfen.

Syntaxdiagramm

Eine beliebte und anschauliche grafische Darstellung bieten Syntaxdiagramme, die durch die Verwendung bei der Programmiersprache Pascal populär wurden. Ein **Syntaxdiagramm** besteht aus zwei Teilen. Links oben wird in einem Rechteck der Name des Syntaxdiagramms angegeben. Dieser Name gibt an, welche Sprachkonstruktion das Syntaxdiagramm beschreibt. Man bezeichnet eine solche Sprachkonstruktion als **nicht-terminales Symbol**. Das eigentliche Diagramm besteht aus Ovalen, Kreisen und Rechtecken, verbunden durch gerichtete Pfeile. Syntaxdia-

gramme werden von links nach rechts gelesen, indem man der Richtung der Pfeile folgt. Ein Rechteck steht für ein Sprachkonstrukt, d. h. ein nicht-terminales Symbol, das in einem anderen Syntaxdiagramm definiert ist. Ein Kreis oder ein Oval enthält Zeichen, die exakt so in dem entsprechenden Programmteil stehen müssen. Man bezeichnet diese Zeichen als **terminale Symbole**.

Beispiel

Die Abb. 3.1-1 zeigt vier Syntaxdiagramme, die die Java-Syntax für einfache Typen definieren. Es werden die vier Sprachkonstrukte (nicht-terminale Symbole) `PrimitiveType`, `NumericType`, `IntegralType` und `FloatingPointType` spezifiziert. Der `PrimitiveType` besteht aus einem `NumericType` oder dem Schlüsselwort `boolean`.

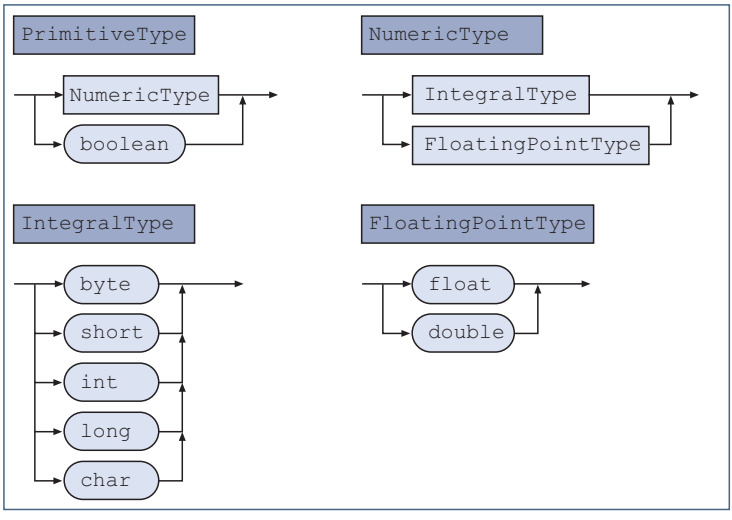
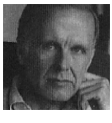


Abb. 3.1-1: So sieht die Syntax für einfache Typen in Java aus.

BNF & EBNF

Nachteilig bei Syntaxdiagrammen ist, dass sie manuell aufwendig zu erstellen sind und viel Platz benötigen. Daher wird in diesem Buch in der Regel eine kompaktere, textuelle Notation verwendet, die sich an die sogenannte **EBNF** *(Extended Backus-Naur-Form)* anlehnt. Sie wird hier als Pseudo-EBNF bezeichnet. Die so genannte Backus-Naur-Form (kurz BNF) wurde von den Wissenschaftlern John Warner Backus und Peter Naur 1960 zur Beschreibung der Syntax der Programmiersprache Algol 60 entwickelt und später erweitert *(extended)*.



Tipp

Es gibt eine ganze Reihe von Zeichenwerkzeugen, die aus einer EBNF-Notation ein Syntaxdiagramm erzeugen. Suchen Sie im Internet mit den Stichworten: `EBNF Syntaxdiagram Tool`.

Verwendete textuelle Syntaxnotation

- *A ::= B* : Das zu definierende nicht-terminale Symbol *A* (*Platzhalter*) steht auf der linken Seite, durch *::=* von seiner Definition *B* auf der rechten Seite getrennt.
- *[ ]* : Eckige Klammern *[ ]* schließen optionale Elemente ein, d. h. Elemente, die auch fehlen dürfen.
- *|* : Ein kursiver senkrechter Strich *|* trennt alternative Elemente, d.h von den aufgeführten Elementen ist ein Element auszuwählen.
- *{}* : Aus den aufgeführten Elementen ist ein Element auszuwählen.
- *+* : Ein *+* gibt an, dass das Element wiederholt werden kann.
- *...* : Drei Punkte kennzeichnen eine Liste von Elementen, wobei die Elemente durch ein Komma (,) getrennt werden.
- `terminale Symbole`: Die Zeichen, die im Quellcode des Programms stehen müssen, sind in normaler Schrift dargestellt.
- `Schlüsselwörter`: Schlüsselwörter sind mit einem Grauraster unterlegt oder fett dargestellt.
- *Syntaxsymbole*: Symbole, die zur Beschreibung der Syntax verwendet werden, sind kursiv dargestellt, z. B. *{ }*, sonst handelt es sich um terminale Symbole.

Beispiel

Das in der Abb. 3.1-1 angegebene Beispiel sieht in der textuellen Notation folgendermaßen aus:

PrimitiveType ::= { NumericType | `boolean` *}*
NumericType ::= { IntegralType | FloatingPointType }
IntegralType ::= { `byte` | `short` | `int` | `long` | `char` *}*
FloatingPointType ::= { `float` | `double` *}*

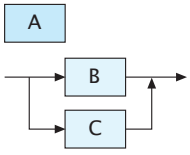
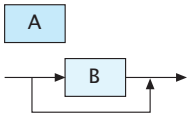
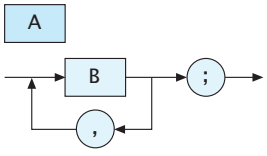
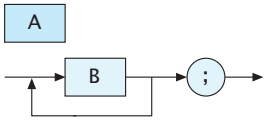
Eine Gegenüberstellung beider Notationen zeigt die Tab. 3.1-1. Die Definition einer Syntaxregel zeigt die Tab. 3.1-2.

Beispiel

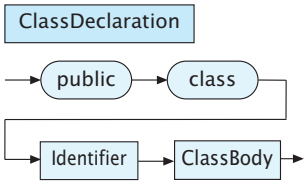
Die Syntax einer Variablen- bzw. Konstantendeklaration in Java sieht in der Pseudo-EBNF wie folgt aus:

FieldDeclaration ::=
{ `private` | `public` | `protected` | `final` | `static` | *}+ Type*
{ Identifier | VariableDeclaratorId [ ]
[= { Expression | ArrayInitializer }] }...;

Das entsprechende Syntaxdiagramm zeigt die Abb. 3.1-2. Zu beachten ist, dass von den Schlüsselwörtern `private`, `public` und `protected` nur jeweils ein Schlüsselwort verwendet werden darf. Außerdem dürfen die Schlüsselwörter `final` und `static` nur jeweils einmal auftreten. Dies Beispiel zeigt, dass die Bedeutung einer Sprachkonstruktion, d. h. die Semantik, durch die Syntax oft nicht oder nur unvollständig beschrieben werden kann.

Notation	**Syntaxdiagramm**	**Pseudo-EBNF**
Nicht-terminales Symbol		*A* (in kursiver Schrift)
Terminales Symbol		**class** (in normaler Schrift, Schlüsselwörter grau hinterlegt, hier in der Tabelle fett dargestellt)
Alternative		*A ::= B | C*
Option		*A ::= [B]*
Wiederholung (Liste)		*A ::= B...* ;
Wiederholung		*A ::= B+* ;

Tab. 3.1-1: Syntaxnotationen im Vergleich.

Notation:	**Syntaxdiagramm**	**Pseudo-EBNF**
Definition einer Syntaxregel (Beispiel)		*ClassDeclaration ::=* **public class** *Identifier Classbody* (Schlüsselwörter grau hinterlegt, hier in der Tabelle fett dargestellt)

Tab. 3.1-2: Definition einer Syntaxregel.

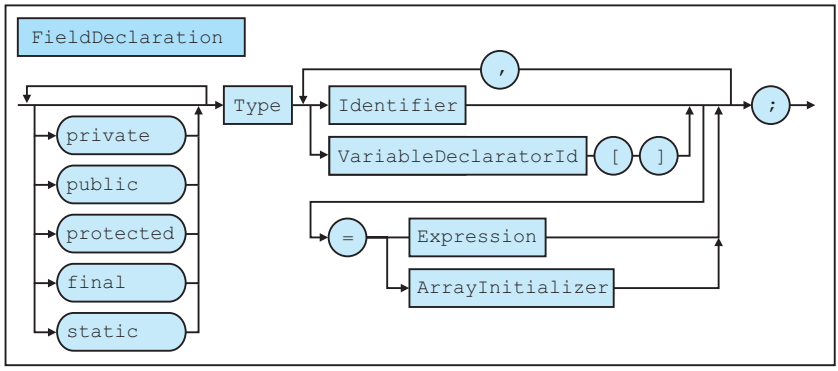


Abb. 3.1-2: So sieht eine Variablendeklaration in Java aus (Ausschnitt).

Beispiel

Die Syntax für Zuweisungen zeigt die folgende Pseudo-EBNF:

*Assignment ::= LeftHandSide { = | += | -= | *= | /= | %= | <<= | >>= | >>>= | &= | |= | ^= } AssignmentExpression*

LeftHandSide ::= ExpressionName | FieldAccess | ArrayAccess

AssignmentExpression ::= Assignment | ConditionalExpression

Das entsprechende Syntaxdiagramm zeigt die Abb. 3.1-3.

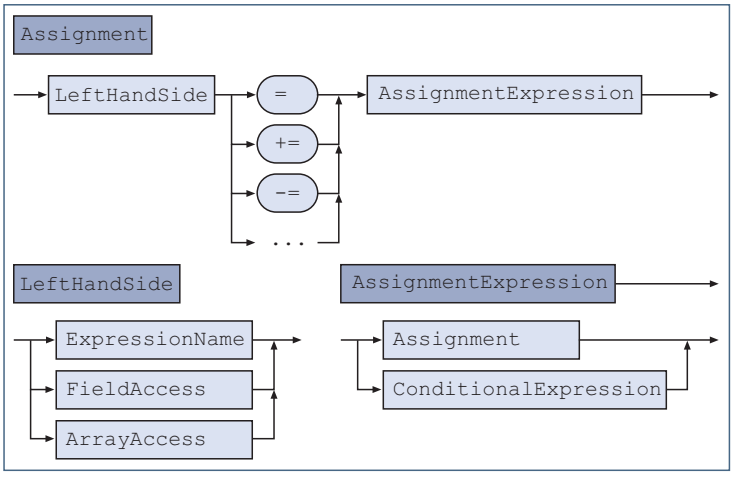
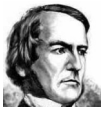


Abb. 3.1-3: So sieht die Syntax für eine Zuweisung in Java aus.

Die Spezifikation der Sprache Java einschließlich der Syntaxbeschreibung finden Sie auf der Webseite Java Language Specification:
`https://docs.oracle.com/javase/specs/jls/se18/jls18.pdf`

3.2 Der Typ boolean *

Variablen und Konstanten vom Typ `boolean` besitzen entweder den Wahrheitswert bzw. logischen Wert `true` oder `false`. Folgende Operationen sind mit booleschen Werten möglich: Abfrage auf Gleichheit (`==`) und Ungleichheit (`!=`), logisches UND (`&`), logisches ODER (`|`), logisches XODER (`^`) sowie die boolesche Negation (`!`). Standardmäßig erfolgt eine vollständige Auswertung von booleschen Ausdrücken, auch wenn das Ergebnis nach Auswertung eines Teilausdrucks schon feststeht. Eine verkürzte Auswertung wird durch die Operationen `&&` und `||` erreicht. Der `?`-Operator erlaubt es, in Abhängigkeit vom Ergebnis eines booleschen Ausdrucks zwei verschiedene andere Ausdrücke auszuwerten.



Eine Variable oder Konstante vom Typ `boolean` kann zu einem Zeitpunkt genau einen von zwei sogenannten Wahrheitswerten `true` (wahr) oder `false` (falsch) annehmen. Solche Wahrheitswerte werden als **logische Werte** oder **Boolesche Größen** bezeichnet – nach dem englischen Mathematiker George Boole (1815–1864).

Ein sinnvoller **Einsatz** dieses Typs ergibt sich, wenn man sich in einem Programm merken muss, ob ein Ereignis eingetreten ist oder nicht. Variablen vom Typ `boolean` stellen praktisch Schalter dar, die jeweils genau einen von zwei Zuständen annehmen können. Außerdem ergeben Vergleiche wie `3 > 10` als Ergebnis stets einen booleschen Wert, hier `false`.

Eigenschaften

Die wichtigsten Eigenschaften des Typs `boolean` sind:

Wertebereich

Der Wertebereich ist plattformunabhängig und umfasst die Werte `true` und `false`. Beides sind Schlüsselwörter und werden als boolesche Literale bezeichnet.

Binäre Operationen

Die Tab. 3.2-1 zeigt, welche binären Operationen (Operand1 Operator Operand2) möglich sind.

Operator	Erklärung	Beispiel	Symbol in der Mathematik
==	Gleichheit	`true == false` ergibt `false`	=
!=	Ungleich	`true!= true` ergibt `false`	≠
&	logisches UND	`true & true` ergibt `true`	∧
|	logisches ODER	`false | false` ergibt `false`	∨
^	logisches XODER	`false ^ true` ergibt `true`	⊻

Tab. 3.2-1: Binäre Operationen.

Unäre Operation

Die mögliche unäre Operation (Operator Operand) zeigt die Tab. 3.2-2.

Operator	Erklärung	Beispiel	Symbol in der Mathematik
!	boolesche Negation	!true ergibt false	¬

Tab. 3.2-2: Unäre Operation.

Die Operatorprioritäten werden im Kapitel »Operatorprioritäten«, S. 89, behandelt.

Da boolesche Variablen nur zwei Werte annehmen können, gibt es für eine Kombination von zwei solchen Werten nur vier verschiedene Möglichkeiten, die eintreten können. Die Ergebnisse der Operationen &, |, ^ und ! zeigt die Tab. 3.2-3.

x & y	true	false	x | y	true	false	x ^ y	true	false	x	! x
true	true	false	true	true	true	true	false	true	true	false
false	false	false	false	true	false	false	true	true	false	true

Tab. 3.2-3: Die Operationen &, |, ^ und !.

Vollständige Auswertung

In Java werden boolesche Ausdrücke mit & und | immer *vollständig* ausgewertet. Bei einer &-Operation kann man aber bereits aufhören, wenn der erste Operand falsch ist. Der gesamte Ausdruck kann nur noch falsch sein.

Verkürzte Auswertung

Will man eine solche verkürzte Auswertung, dann muss man die Operation && benutzen. Bei einer |-Operation kann man die Auswertung beenden, wenn der erste Operand wahr ist. Eine verkürzte Auswertung erreicht man durch die Operation ||.

Beispiele

Bei dem zusammengesetzten Ausdruck (7 == 10) & (8!= 9) genügt bereits die Auswertung des ersten Operanden (7 == 10), um festzustellen, dass der gesamte Ausdruck falsch ist. Der Teilausdruck (8!= 9) müsste daher nicht mehr ausgewertet werden. Dies erreicht man durch eine verkürzte Auswertung: (7 == 10) && (8!= 9). Da es in Java jedoch erlaubt ist, in einem Ausdruck eine Zuweisung vorzunehmen, kann in solchen Fällen eine vollständige Auswertung nötig sein:

```
//Deklaration boolescher Variablen
boolean ungleich = false, wert;
wert = (7 == 10) & (ungleich = 8 != 9);
```

Bei der vollständigen Auswertung erhält ungleich den Wert true. Bei einer verkürzten Auswertung wird nach (7 == 10) abgebrochen und ungleich behält den Voreinstellungswert false.

Empfehlung

Verwenden Sie in der Regel die verkürzte Auswertung.

Gesetze

Aus der Definition der Operatoren ergeben sich Gesetze, die in der Praxis oft nützlich sind (Tab. 3.2-4).

Boolesche Gesetze	Name des Gesetzes
`x | y = y | x` `x & y = y & x`	Kommutativgesetze
`(x | y) | z = x | (y | z)` `(x & y) & z = x & (y & z)`	Assoziativgesetze
`(x & y) | z = (x | z) & (y | z)` `(x | y) & z = (x & z) | (y & z)`	Distributivgesetze
`!(x | y) =!x &!y` `!(x & y) =!x |!y`	de Morgansche Gesetze

Tab. 3.2-4: Gesetze mit booleschen Operatoren.

Beispiel

DemoBoolean

Das folgende Programm liest boolesche Werte ein und verknüpft sie miteinander. Es verwendet die Methode `readBoolean()` zum Einlesen eines booleschen Wertes (siehe »Java-Pakete anlegen und benutzen: das Wichtigste«, S. 43):

```
// Das Programm demonstriert die
// booleschen Operationen

import inout.Console;

public class DemoBoolean
{
//Führt boolesche Operationen aus
 public static void main (String args[])
 {
   boolean schalter1 = false;
   boolean schalter2 = true;
   System.out.println
    ("Boolsche Operationen");
   System.out.println("Schalter 1 (true oder false eingeben):");
   schalter1 = Console.readBoolean(); //Einlesen
   System.out.println("Schalter 2 (true oder false eingeben):");
   schalter2 = Console.readBoolean(); //Einlesen
   boolean und = schalter1 & schalter2;
   boolean oder = schalter1 | schalter2;
   System.out.println("Schalter 1 = " + schalter1);
   System.out.println("Schalter 2 = " + schalter2);
   System.out.println("Und = " + und);
   System.out.println("Oder = " + oder);
 }
}
```

Führen Sie das Programm `DemoBoolean` auf Ihrem Computersystem aus und geben Sie alle Kombinationen für `schalter1` und `schalter2` ein.

Erweitern Sie anschließend das Programm um die oben angegebenen Gesetze und prüfen Sie die Formeln mit einigen Werten.

?:-Operator

In Java gibt es einen speziellen ternären Operator `?:`. Die Syntax sieht folgendermaßen aus:

Ausdruck1 `?` *Ausdruck2* `:` *Ausdruck3*

Ausdruck1 kann ein beliebiger Ausdruck sein, dessen Ergebnis `true` oder `false` ist. Wenn der *Ausdruck1* `true` ergibt, dann wird der *Ausdruck2* ausgewertet. Ist der *Ausdruck1* `false`, dann wird der *Ausdruck3* ausgewertet.

Beispiel

Bei der Berechnung von Zinstagen werden pro Monat max. 28 Tage zugrunde gelegt. Werden die Tage 29, 30 oder 31 eingelesen, dann müssen sie auf 28 gesetzt werden.

```
int zinstage;
tage = Console.readInt();
zinstage = tage > 28 ? 28 : tage;
```

Wenn `tage > 28` ist, dann wird `zinstage = 28` zugewiesen, sonst wird `zinstage = tage` gesetzt.

Hinweis

Da der Operator `?:` je nach Wert des *Ausdrucks1* entweder den *Ausdruck2* oder den *Ausdruck3* auswertet, zählt dieser Operator auch zu den Kontrollstrukturen (siehe »Die ein- und zweiseitige Auswahl«, S. 109).

3.3 Ganzzahlige Typen *

Für ganze Zahlen stellt Java die Typen `int`, `long`, `short` und `byte` mit jeweils unterschiedlichen Wertebereichen zur Verfügung. Ganzzahlige Literale werden als Ziffernfolge, in Oktaldarstellung mit vorangestellter Null oder in Hexadezimaldarstellung mit vorangestelltem `0x` dargestellt. Literale vom Typ `long` werden durch ein nachgestelltes `L` bzw. `l` gekennzeichnet. Folgende binäre Operationen sind möglich: `+`, `-`, `*`, `/`, `%`, `<`, `<=`, `>`, `>=`, `==`, `!=`. Folgende unäre Operationen sind erlaubt: `-`, `~`, `++`, `--`. Tritt bei der Ausführung der Operationen *kein* Überlauf *(overflow)* über den jeweiligen Wertebereich auf, dann sind die Ergebnisse exakt.

Ganze Zahlen *(integer)*, d. h. Zahlen ohne Nachkommastellen, können beliebig groß sein. Wegen der begrenzten Größe einer Speicherzelle im Arbeitsspeicher eines Computersystems kann aber nur eine **Teilmenge der ganzen Zahlen** dargestellt wer-

den. In Abhängigkeit von dem zur Verfügung stehenden Speicherplatz unterscheidet Java vier verschiedene ganzzahlige Typen (Tab. 3.3-1). Der Wertebereich der Typen ist plattformunabhängig.

Typ	Kleinster Wert	Größter Wert	Speicherplatz	Voreinstell.
`byte`	-128 (-2^7)	$+127$ ($+2^7-1$)	1 Byte	0
`short`	-32.768 (-2^{15})	32.767($+2^{15}-1$)	2 Bytes	0
`int`	$-2.147.483.648$ (-2^{31})	$2.147.483.647$ ($+2^{31}-1$)	4 Bytes	0
`long`	$-9.223.372.036.854.775.808$ (-2^{63})	$9.223.372.036.854.775.807$ ($+2^{63}-1$)	8 Bytes	0L

Tab. 3.3-1: Ganzzahlige Typen in Java.

Literaldarstellung

Ganzzahlige Werte werden als `IntegerLiteral` dargestellt. Vier Darstellungsarten sind möglich:

- Dezimaldarstellung: Ziffernfolge, z. B. `110`.
- Binärdarstellung: Ziffernfolge mit vorangestelltem `0B` bzw. `0b`, z. B. `0b1101110` (ab Java 7).
- Oktaldarstellung: Ziffernfolge mit vorangestellter Null, z. B. `0156`.
- Hexadezimaldarstellung: Ziffernfolge mit vorangestelltem `0X` bzw. `0x`, z. B. `0x6E`.

Zahlen vom Typ `long` werden wie folgt dargestellt:

- Literal vom Typ `long`: Ziffernfolge mit nachgestelltem `L` bzw. `l`, z. B. `110L`.
- Binär-, Oktal- und Hexadezimaldarstellung sind ebenfalls möglich, z. B. `0x6EL`.

Strukturierung

Seit Java 7 ist es möglich, numerische Literale durch Unterstreichstriche (_, *underscore*) zu strukturieren. Am Anfang und am Ende eines Literals darf kein Unterstreichstrich stehen.

Beispiele

```
int zwoelfMillionen = 12_000_000;
System.out.println("Betrag: " + zwoelfMillionen);
//Aufeinanderfolgende Unterstreichstriche sind erlaubt
int eineMilliarde = 1_000__000___000;
long langeZahl = 1_23_45_67_89_123L;
int hex = 0Xab_cd;
```

Alle Werte besitzen ein Vorzeichen Minus (–) oder Plus (+), wobei das Pluszeichen weggelassen werden kann.

Einsatz

In der Regel werden die Typen `int` und `long` verwendet. Der Typ `int` wird insbesondere zum Zählen und für ganzzahlige mathematische Operationen verwendet. Der Typ `long` wird benutzt,

wenn der Typ `int` nicht ausreicht, um den gewünschten Wert aufzunehmen. Die Typen `byte` und `short` werden für spezielle Anwendungsfälle eingesetzt. Beispielsweise wird der Typ `byte` für das Speichern von Daten auf externen Datenträgern und für die Übertragung von Daten über das Netz benutzt.

Operationen

Die Tab. 3.3-2 zeigt, welche binären Operationen mit ganzen Zahlen möglich sind. Binäre Operation bedeutet dabei, dass links und rechts vom Operator jeweils ein Operand steht (`Operand1 Operator Operand2`).

Operator	Erklärung	Beispiele
Arithmetische Operationen	Typ des Ergebnisses: `int` oder `long`	
+	Addition	`5 + 6` ergibt 11
–	Subtraktion	`9 - 3` ergibt 6
*	Multiplikation	`10 * 15` ergibt 150
/	Division	`13 / 3` ergibt 4
%	Modulo (Rest der Division)	`20% 7` ergibt 6
Vergleichs-Operationen	Typ des Ergebnisses: `boolean`	
<	Kleiner	`3 < 5` ergibt `true`
<=	Kleiner gleich	`3 <= 3` ergibt `true`
>	Größer	`2 > 10` ergibt `false`
>=	Größer gleich	`15 >= 16` ergibt `false`
==	Gleich	`3 == 3` ergibt `true`
!=	Ungleich	`5!= 5` ergibt `false`

Tab. 3.3-2: Ganzzahlige binäre Operationen.

Ganzzahlige Division

Die Division bedeutet reellwertiges Dividieren und nachfolgendes Abschneiden hinter dem Dezimalpunkt. Die Rundung erfolgt also stets in Richtung der Null auf der Zahlengeraden.

Restberechnung

Die Modulo-Operation dient zur Restberechnung bei einer ganzzahligen Division. Sie erfüllt die Gleichung

```
A = (A / B) * B + (A % B)
```

Verständlicher wird die Formel, wenn man die geklammerten Ausdrücke durch Variablen ersetzt.

```
C = A / B: Ganzzahlenquotient = Dividend / Divisor
R = A % B: Rest = Divident % Divisor
```

Die Formel sieht dann wie folgt aus: `A = C * B + R`

```
A = 20; B = 3; C = 20/3 = 6; R = 20 % 3 = 2;
A = 6 * 3 + 2 = 20
```

Beispiel

Die Tab. 3.3-3 zeigt, welche unären Operationen mit ganzen Zahlen möglich sind. Unäre Operation bedeutet dabei, dass rechts vom Operator genau ein Operand steht (`Operator Operand`).

Operator	Erklärung	Beispiele
-	Unäre Negation	`- 3`
~	Bitweises Komplement	`~ 101` ergibt 010
++	Inkrement	`++ A` steht für `A = A + 1`
−−	Dekrement	`-- A` steht für `A = A - 1`

Tab. 3.3-3: Ganzzahlige unäre Operationen.

- In einer Anweisung bedeutet `A = ++ B`, dass `B + 1` vor der Zuweisung ausgeführt wird: `A = (B = B + 1);`
- In einer Anweisung bedeutet `A = B ++`, dass `B + 1` erst nach der Zuweisung von B an A erhöht wird. Analog gelten diese Regeln für --.

Die Operatorprioritäten werden im Kapitel »Operatorprioritäten«, S. 89, behandelt.

Zusätzlich gibt es noch die **bitweisen Operatoren** `&`, `|`, `^`, `<<`, `>>`, `>>>`, `~`, `<<=`, `>>=`, `>>> =`, `&=`, `|=` und `^=`, die hier aber *nicht* behandelt werden.

Hinweis

Überlauf

Da der Wertebereich zur Darstellung von ganzen Zahlen – in Abhängigkeit vom gewählten Typ – begrenzt ist, gelten die meisten **Gesetze der Arithmetik** auf einem Computersystem *nur* eingeschränkt.

Beispiel

Die Addition auf einem Computersystem (hier gekennzeichnet durch `++`) stimmt nur dann mit der normalen Addition überein, d. h. `x ++ y = y + x`, wenn `|x| < MAX`, `|y| < MAX` und `|x + y| < MAX` ist. `MAX` ist dabei die Zahl (ohne Vorzeichen), die im Wertebereich gerade *nicht mehr* dargestellt werden kann. Hinweis: Die senkrechten Striche kennzeichnen den Absolutbetrag, d. h. den Wert ohne Vorzeichen. Beispiel: |−300 + 100| = |−200| = 200.

Tritt bei der Ausführung eines Programms *kein* **Überlauf** *(overflow)* auf, dann sind die Ergebnisse aller Operationen mit ganzzahligen Größen exakt.

Wird in einem Java-Programm bei der Berechnung eines Ausdrucks der ganzzahlige Zahlenbereich überschritten, dann erfolgt *keine* Fehlermeldung.

Beispiel 1a

```
public class DemoOverflow
{
   public static void main(String[] args)
   {
       int a = 3, b = 5, c = Integer.MAX_VALUE;
       //Integer.MAX_VALUE liefert den größten
       //ganzzahligen Wert
       System.out.println("c = " + c);
       System.out.println("a + c = " + (a + c));
       System.out.println("a + b = " + (a + b));
       //Mit Fehlermeldung
       System.out.println("Mit Overflow-Check: " +
          SecuredArithmetics.checkedIADD(a,c));
    }
}
```

Dieses Programm liefert folgendes Ergebnis:

```
c = 2147483647
a + c = -2147483646
a + b = 8
Fehler: Overflow
Mit Overflow-Check: -2147483646
```

In dem Artikel »*Signalling Integer Overflows in Java*« von [BaKi08] wird beschrieben, wie durch eine zusätzliche Abfrage eine Fehlermeldung im Falle der Zahlenbereichsüberschreitung erzeugt werden kann.

Beispiel 1b

Zusätzliche Klasse mit der Methode `checkedIADD()` zur Überprüfung des Zahlenbereichs:

```
public class SecuredArithmetics
// Quelle: [BaKi08, S. 55]
{
    static int checkedIADD(int a, int b)
    {
        int r = a + b;
        if (( (a^r) & (b^r) ) < 0)
          System.out.println("Fehler: Overflow");
        return r;
    }
}
```

Diese Methode bewirkt die oben angegebene Fehlermeldung im Beispiel 1a.

- Der Sage nach war der indische König Shehram vom **Schachspiel** so begeistert, dass er dem Erfinder Sessa Ebn Daher jeden Wunsch zu erfüllen versprach. Um so überraschter war er, als der Erfinder die *bescheidene* Bitte äußerte, der König

möge ihm so viele Weizenkörner schenken, wie sich zusammen ergeben, wenn man auf das erste der 64 Felder des Schachbretts ein Weizenkorn legt, auf das zweite zwei, auf das dritte vier, auf das vierte acht und so fort, d. h. auf das nächste Feld immer die doppelte Anzahl von Körnern wie auf das vorhergehende (Abb. 3.3-1). Wie viele Körner kommen dabei zusammen und welcher Java-Typ kann diese Zahl noch darstellen?

Abb. 3.3-1: Schachbrett mit jeweils verdoppelter Reiskörneranzahl.

- Schreiben Sie folgende Deklaration und übersetzen Sie sie:
  ```
  int a = 2147483648;
  ```
- Was erhalten Sie als Ergebnis beim folgenden Programmausschnitt:
  ```
  int a = 2147483647 , b = 2147483647;
  System.out.println(a + b);
  ```
  Modifizieren Sie die Werte, so dass `|a + b| > MAX` ist. Welche Ergebnisse erhalten Sie?

Hinweis

In Java können Zuweisungen (=) mit Operationen verknüpft werden, um eine kompaktere Schreibweise zu erreichen. Beispielsweise kann anstelle von `a = a + b;` geschrieben werden:

```
a += b;
```

U. a. sind folgende Zuweisungen mit Operationen möglich: `+=, -=, *=, /=, %=`.

3.4 Gleitpunkt-Typen *

Die Typen `double` und `float` erlauben in Java die näherungsweise Darstellung von Zahlen mit Nachkommastellen. Ein `float`-Literal muss den Suffix `f` bzw. `F` besitzen, ein `double`-Literal kann den Suffix `d` bzw. `D` haben. Literale werden in halblogarithmischer Darstellung geschrieben, z. B. `123E-3`. Es sind die Operationen `+, -, *, /, %, <, <=, >, >=, ==, !=` (binär) und `-, ++, --` (unär) erlaubt.

Zur näherungsweisen Darstellung von reellen Zahlen stellt Java die Typen `float` und `double` zur Verfügung. Bei `float`- und `double`-Variablen bzw. Konstanten handelt es sich um gebrochene Zahlen, d. h. um Zahlen, die Stellen hinter dem Komma haben können (z. B. 3,141592...).

Punkt statt Komma

In der Informatik wird anstelle des Dezimalkommas ein Dezimalpunkt verwendet (z. B. 3.141592..., angelsächsische Schreibweise). Daher spricht man auch von Gleit*punkt*-Typen anstelle von Gleit*komma*-Typen.

Eigenschaften

Die wichtigsten Eigenschaften der Typen `float` und `double` sind:

Wertebereich

Der Wertebereich ist plattformunabhängig und entspricht dem ANSI/IEEE Standard 754 von 1985. Die Tab. 3.4-1 gibt die Wertebereiche an.

Typ	Wertebereich (ungefähr)	Speicherplatz	Voreinstellung
`float`	$-3.4*10^{-38}$ bis $+3.4*10^{+38}$ (ergibt ungefähr 6 bis 7 signifikante Dezimalziffern Genauigkeit)	4 Bytes (32 Bits)	0.0f
`double`	$-1.7*10^{-308}$ bis $+1.7*10^{+308}$ (ergibt ungefähr 15 signifikante Dezimalziffern Genauigkeit)	8 Bytes (64 Bits)	0.0d

Tab. 3.4-1: Eigenschaften von `float` und `double`.

Einsatz Normalfall: `double`

Der Typ `double` ermöglicht Zahldarstellungen, die gegenüber dem Typ `float` die doppelte Genauigkeit besitzen. **In der Regel wird der Typ `double` verwendet!**

Hinweis

In Java werden `float`- und `double`-Ausdrücke ungefähr gleich schnell berechnet. Ein Vorteil für `float` liegt nur dann vor, wenn Speicherplatz gespart werden muss. Die Voreinstellung

in Java ist `double`, das sieht man an den Literalen, die ohne Suffix immer vom Typ `double` sind.

Der Typ `float` ist ungenau bei sehr großen und bei sehr kleinen Werten. Er ist geeignet, wenn ein Bruchteil benötigt wird, aber kein größerer Präzisionsgrad gefordert ist.

Suffix ohne Suffix = double

Gleitpunkt-Literale unterscheiden sich durch ihren Suffix. Ein `float`-Literal besitzt am Ende ein kleines oder großes `f`, z. B. `123f`, ein `double`-Literal ein nachgestelltes kleines oder großes `d`, z. B. `123D`. Fehlt der Suffix, dann liegt ein `double`-Literal vor!

Potenzschreibweise

Um einen großen Zahlenbereich überstreichen zu können, ist oft eine Zahlendarstellung in Potenzschreibweise vorteilhaft.

Beispiele

$123\ 000\ 000 = 123 * 10^6 = 1.23 * 10^8$

Bei der Potenzdarstellung von Dezimalzahlen können auch negative Exponenten verwendet werden:

$123\ 000\ 000\ 000 * 10^{-3} = 123\ 000\ 000\ 000\ (1/10^3) =$
$123\ 000\ 000\ 000\ (1/1000) = 123\ 000\ 000.$

Negative Exponenten werden vorteilhaft verwendet, um sehr kleine Zahlen übersichtlich darzustellen:

$0.000\ 000\ 789 = 0.789 * 10^{-6} = 7{,}89 * 10^{-7} = 789 * 10^{-9}.$

Halblogarithmisch

Da im Allgemeinen mit Dezimalzahlen gearbeitet wird, lässt man die Basis 10 weg und kommt zu der in der Abb. 3.4-1 gezeigten **halblogarithmischen Darstellung** einer Komma- bzw. Punktzahl (E steht für Exponent). Der Vorteil dieser Schreibweise liegt darin, dass man Gleitpunktzahlen auch in Potenzdarstellung in *einer* Zeile schreiben kann.

Beispiele

$5.67E8 = 5.67 * 10^8$
$123e{-}9 = 123 * 10^{-9}$
$1E{-}2 = 0.01$
$0.7e00 = 0.7$

Strukturierung

Seit Java 7 ist möglich, numerische Literale durch Unterstreichstriche zu strukturieren (siehe »Ganzzahlige Typen«, S. 64). Unterstreichstriche können auch nach dem Dezimalpunkt verwendet werden, z. B. `double demo = 123_456.8_9;`, aber nicht direkt vor oder hinter dem Dezimalpunkt.

Operationen

Es sind alle binären Operationen analog wie bei ganzzahligen Typen möglich:

- Arithmetische Operationen: `+`, `-`, `*`, `/`, `%` (Der Modulo-Operator liefert den Dezimal-Divisionsrest).

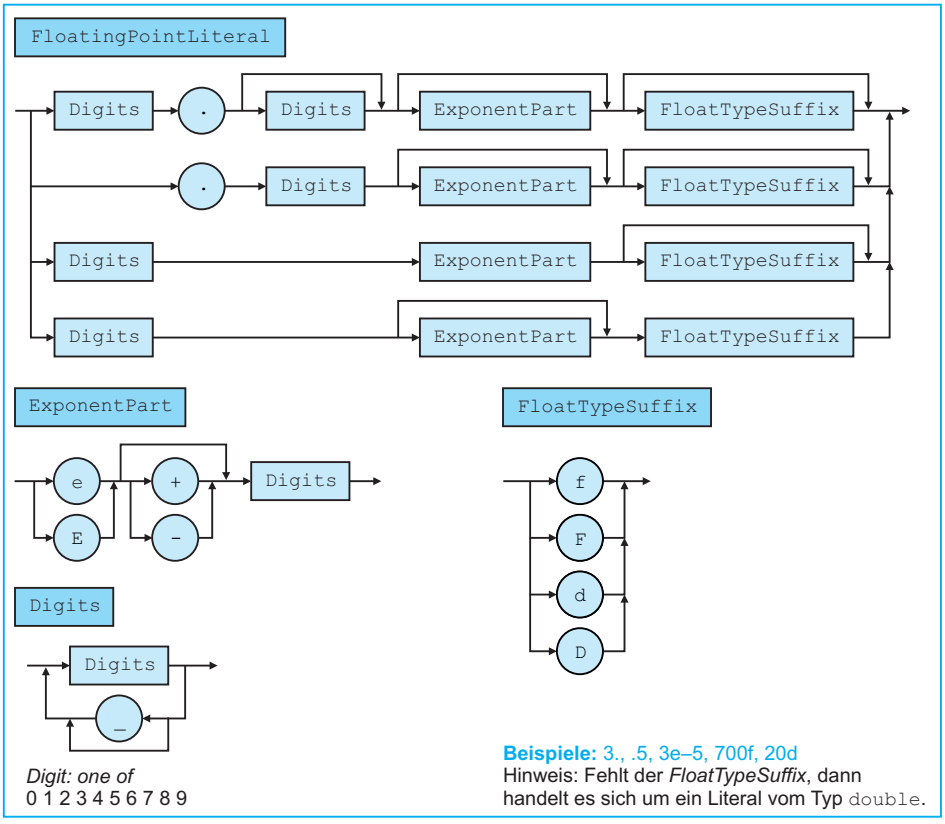


Abb. 3.4-1: Syntax für Gleitpunkt-Literale in Java.

- Sind bei den arithmetischen Operationen beide Operanden vom Typ `float`, dann ist der Ergebnistyp `float`, in allen anderen Fällen `double`
- Vergleichsoperationen: `<, <=, >, >=, ==, !=`

Folgende unäre Operationen sind – analog wie bei ganzzahligen Typen – erlaubt:

- `-, ++, --`

Programmierregel

Die Typen `double` und `float` sollen nur verwendet werden, wenn die benötigten Wertebereiche von der Problemstellung her Gleitpunkt-Zahlen erfordern. Ist von vornherein bekannt, dass eine Variable ganzzahlig ist, dann ist auch ein ganzzahliger Typ zu verwenden. Die Wertemenge, die von der Problemstellung her gegeben ist, sollte so eng wie möglich vereinbart werden. Ist beispielsweise ein Problem von Natur aus ganzzahlig (z. B. Berechnung der Fakultät), dann sollte auch ein

ganzzahliger Typ vereinbart werden. Das Programm wird dadurch klarer und seine Wirkungsweise leichter verständlich. Die Berechnung der Fakultät ist in der Mathematik beispielsweise nur für ganze Zahlen definiert.

In der internationalen Norm IEEE 754 (ANSI/IEEE Std 754–1985; IEC-60559:1989) sind Standarddarstellungen für binäre Gleitpunktzahlen in 32 Bit- *(single precision)* und in 64 Bit-Genauigkeit *(double precision)* festgelegt, die auch in Java eingehalten werden. Weitere IEEE 754-Darstellungen sind *single extended* (> 42 Bit) und *double extended* (> 78 Bit), die es in Java aber nicht als eigenständige Typen gibt. Bei Berechnungen werden in Java aber diese Darstellungen benutzt, wenn es die Hardware »hergibt«. Dadurch erhält man eventuell auf unterschiedlichen Plattformen unterschiedliche Ergebnisse. Um dies zu verhindern kann man das Schlüsselwort `strictfp` vor Klassen oder Methoden setzen, um auf allen Plattformen das gleiche Ergebnis zu erhalten. In Java tritt beim Überschreiten eines gültigen Bereichs *kein* Laufzeitfehler auf, obwohl dies in der IEEE 754 gefordert wird.

3.5 Darstellung von Gleitpunkt-Zahlen **

Durch die begrenzte Stellenanzahl für eine Gleitpunktzahl in einem Computersystem werden Gleitpunktzahlen in normalisierter, halblogarithmischer Form gespeichert. Reelle Zahlen bilden ein Kontinuum, das im Computersystem auf eine endliche Menge von Repräsentanten abgebildet wird. Durch die halblogarithmische Darstellung ist die Dichte der Repräsentanten unterschiedlich. Dadurch ist der absolute Fehler, der durch den Verlust von Stellen entsteht, unterschiedlich.

Während das Rechnen mit ganzen Zahlen zu exakten Ergebnissen führt, solange die Resultate nicht im Überlaufbereich liegen, gilt dies für arithmetische Operationen mit Gleitpunkt-Zahlen in Computersystemen *nicht* mehr. Das hat verschiedene Gründe.

Aufbau Gleitpunkt-Zahl

Wie bei ganzen Zahlen, so kann auch bei Gleitpunkt-Zahlen nur eine endliche Ziffernzahl gespeichert werden. Eine Gleitpunkt-Zahl besitzt allgemein folgende Form:

$a = \pm m * b^{\pm e}$ (z. B. $16.875 * 10^{-1}$)

m wird als Mantissenteil (auch Bruch genannt), b als Basis und e als Exponent bezeichnet. Diese Form wird **Gleitpunkt-Darstellung** *(floating point)* oder **halblogarithmische Darstellung** genannt.

Die Bezeichnung Gleitpunkt kommt daher, dass durch die Wahl des Exponenten die Lage des Punktes bzw. Kommas festgelegt

wird und durch Änderung des Exponenten verschoben werden kann.

Beispiel

$1.6875 = 1.6875 * 10^0 = 0.16875 * 10^1 = 0.016875 * 10^2 = 16.875 * 10^{-1}$

Man läßt meist die Basis 10 weg und schreibt nur den Exponenten hin, z. B.

1.6875E0 = 0.16875E1 = 0.016875E2 = 16.875E-1.

Da nur ein ganzzahliger Exponent herausgezogen wird, der Mantissenteil aber nicht logarithmisch dargestellt wird, heißt dies halblogarithmische Darstellung.

Normalisierte Darstellung

Wie das Beispiel zeigt, gibt es bei der Gleitpunkt-Darstellung für eine Zahl verschiedene Darstellungsformen. Um eine eindeutige Darstellung zu erhalten, wird in vielen Computersystemen der Mantissenteil immer so gewählt, dass `1 / b <= m < 1` gilt.

Ist die Basis `b = 10`, dann muss `0.1 <= m < 1` erfüllt sein. Alle Mantissen sind dann kleiner 1, d. h. sie haben einen gedachten Punkt vor ihrer höchsten Stelle (Sie fangen alle mit `0.` an). Außerdem sind alle Mantissen `>= 1 / b` (Hinter `0.` folgt direkt keine `0`). So dargestellte reelle Zahlen nennt man **normalisiert**. Die normalisierte Darstellung der Zahl 1.6875 lautet also $0.16875 * 10^1$. Der Vorteil dieser Darstellungsform liegt darin, dass überflüssige Nullen durch Punktverschiebung beseitigt werden, d. h. die verfügbare Stellenzahl wird optimal genutzt.

Beschränkter Speicherplatz

Für den Exponenten e und die Mantisse m stehen in Computersystemen nur eine beschränkte Stellenzahl zur Verfügung (`e < E`, `m < M`). Die Werte von b, E und M variieren dabei von Computer-Plattform zu Computer-Plattform. Die Basis b ist meistens nicht 10 (dezimale Gleitpunktzahl), sondern 2 oder eine Potenz von 2.

Abschneiden oder Runden

Bei den folgenden Beispielen wird von b = 10 ausgegangen, d. h. von Dezimalzahlen. Durch die beschränkte Stellenzahl, insbesondere des Mantissenteils, können Gleitpunktzahlen, wenn sie eine größere Stellenzahl besitzen, nicht mehr exakt in einem Computersystem dargestellt werden. Die Ziffern, die außerhalb des Darstellungsbereichs liegen, müssen entweder **abgeschnitten** *(truncated)* werden, oder es erfolgt ein **Auf-** bzw. **Abrunden** *(rounding)* der letzten darstellbaren Ziffer in Abhängigkeit von dem Rest (z. B. `Rest >= 0.5`, dann aufrunden).

Beispiele

Für die folgenden Beispiele gilt folgende vereinfachende Annahme: Für den Mantissenteil stehen drei Stellen nach dem Punkt zur Verfügung. Der Exponent sei maximal zweistellig.

Die Wirkung von Abschneiden und Runden bei beschränkter Stellenzahl zeigt die Tab. 3.5-1.

Reelle Zahl	Normalisierte abgeschnittene Gleitkommazahl	Normalisierte gerundete Gleitkommazahl
1/3 = 0.3333...	0.333E0	0.333E0
2/3 = 0.6666...	0.666E0	0.667E0
π = 3.141592...	0.314E1	0.314E1
$\sqrt{5}$ = 2.236067...	0.223E1	0.224E1

Tab. 3.5-1: Wirkung von Abschneiden und Runden.

Durch die begrenzte Stellenzahl für den Mantissenteil gehen Stellen verloren, d. h. an Stelle der exakten Werte wird mit Näherungswerten gerechnet. Da nur mit Näherungswerten gerechnet wird, folgt natürlich, dass es sich bei den Ergebnissen auch nur um Näherungswerte handeln kann. Die Abschätzung der Fehler, die sich aus den Operationen mit Zahlen ergeben, ist in den meisten Fällen sehr schwierig. Mit diesem Gebiet befasst sich die Numerische Mathematik. Die meisten Computersysteme wandeln Dezimalzahlen in das Zweiersystem um. Auch einfach erscheinende Zahlen wie 0.1 werden dabei gerundet und deshalb ungenau. Ohne Rundungsfehler wird aber z. B. 0.125 als 2^{-3} dargestellt.

Frage

Überlegen Sie sich für folgende Zahlen jeweils die normalisierte Gleitkommadarstellung. Der Rest ist abzuschneiden. Es gelten die oben gemachten Annahmen:

```
1.   0.1 = ?          2.  4.0099E-1 = ?    3.  50.0E-2 = ?
4.   0.50055 = ?      5.  1.0 = ?          6.  10000. = ?
7.   40099. = ?       8.  5.0E+4 = ?        9.  50055. = ?
10.  100000. = ?
```

Antwort

```
1. 0.100E0  2.  0.400E0 3.  0.500E0
4. 0.500E0  5.  0.100E1 6.  0.100E5
7. 0.400E5  8.  0.500E5 9.  0.500E5
10. 0.100E6
```

Diese Übungsbeispiele zeigen noch weitere Auswirkungen, die sich durch die normalisierte Darstellung ergeben.

Kontinuum & Repräsentanten

Im Gegensatz zu ganzen Zahlen bilden reelle Zahlen ein **Kontinuum**, d. h. in jedem noch so kleinen Intervall (Bereich) der reellen Zahlenachse liegen unendlich viele reelle Zahlen (zwischen 0 und 1 liegen unendlich viele reelle Zahlen, aber auch zwischen 0

und 0.1 usw.). In einem Computersystem kann aber nur eine endliche Menge von **Repräsentanten** des Kontinuums dargestellt werden. Die Abbildung des Kontinuums auf die Repräsentanten geschieht durch Runden oder Abschneiden (0.50055 wird z. B. auf den Repräsentanten 0.500 abgebildet). Durch die Verwendung der halblogarithmischen Darstellung ist außerdem noch die Dichte der Repräsentanten unterschiedlich. Der Abschnitt 0.1 bis 1 (Frage: 1. bis 3.) enthält gleich viele Repräsentanten wie der Abschnitt 10 000 bis 100 000 (Frage: 6. bis 10.).

Die Abb. 3.5-1 veranschaulicht die abnehmende Dichte der Repräsentanten bei wachsenden Zahlen (annähernd exponentieller Verlauf). Annahme: 3-stellige Genauigkeit des Computersystems.

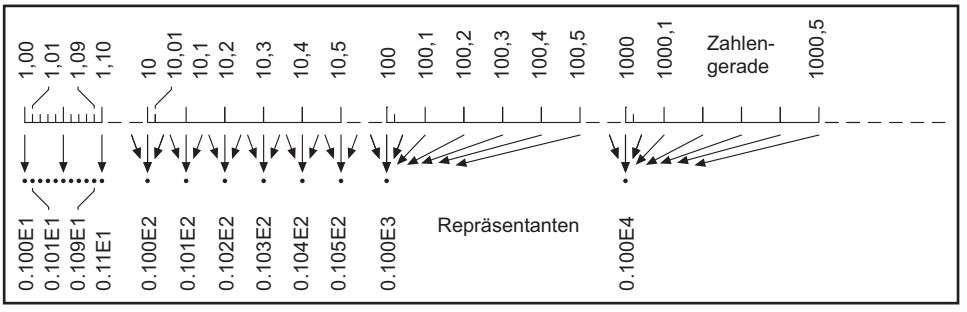


Abb. 3.5-1: Abnehmende Dichte der Repräsentanten bei wachsenden Zahlen.

Je größer die Zahlen werden, desto größer ist das Intervall, das auf einen Repräsentanten abgebildet werden muss.

Beispiel

Alle Zahlen, die im Intervall von 1.00 bis 1.01 liegen (Intervalllänge 0.01) werden auf die zwei Repräsentanten 0.100E1 und 0.101E1 abgebildet. Alle Zahlen, die im Intervall von 10 bis 10.1 liegen (Intervalllänge 0.1) werden auf 0.100E2 und 0.101E2, alle Zahlen im Intervall von 100 bis 101 (Intervalllänge 1) auf 0.100E3 und 0.101E3 abgebildet.

Relative Genauigkeit vs. absolute Fehler

Durch die ungleiche Verteilung der Repräsentanten ist die **relative Genauigkeit** überall gleich, aber der **absolute Fehler**, der durch den Verlust von Stellen entsteht, ist unterschiedlich.

Beispiel

Absoluter Fehler bei 4.: 0.50055–0.500 = **0.00055**
Relativer Fehler: 0.11 %

Absoluter Fehler bei 9.: 50055 – 50000 = **55**
Relativer Fehler: 0.11 %

Hinweis: relativer Fehler = absoluter Fehler / exakter Wert

In der Norm IEEE 754 werden zusätzlich ein positives und ein negatives Unendlich *(infinity)* definiert, die Zahlen repräsentieren, deren Betrag zu groß ist, um dargestellt zu werden. Zum Beispiel ergibt der Ausdruck 1.0/0.0 per Definition den Wert +Unendlich.

Außerdem werden mehrere Zahlen definiert, die eigentlich gar keine sind. Es handelt sich hierbei um ungültige (oder nicht definierte) Ergebnisse, NaN genannt – für *Not-a-Number*. Sie werden als Fehlerindikator für das Ergebnis von ungültigen oder nicht definierten Rechenoperationen benutzt, z. B. 0.0 / 0.0. In verschiedenen Anwendungsbereichen werden NaNs benutzt, um »Unbekannter Wert« oder »Kein Wert« darzustellen. Ein Test auf NaN geht in Java wie folgt: `if (Double.isNaN(d))`

3.6 Rechengenauigkeit mit Gleitpunkt-Zahlen **

Durch die begrenzte Stellenanzahl für eine Gleitpunktzahl in einem Computersystem – insbesondere beim Mantissenteil – sind die Addition von Zahlen sehr *unterschiedlicher* Größe und die Subtraktion von *dicht benachbarten* Zahlen (Auslöschung) zu vermeiden. Wegen der Überlaufgefahr soll *keine* Division durch Werte in der Nähe von Null vorgenommen werden. Abfragen auf Gleichheit (==) von zwei Gleitpunktzahlen sind verboten. Stattdessen ist auf einen tolerierbaren Differenzbetrag ε abzufragen (`Math.abs(x - y) < Epsilon`).

Die Darstellung von Gleitpunktzahlen in Computersystemen (siehe »Darstellung von Gleitpunkt-Zahlen«, S. 73) hat Auswirkungen auf die Ergebnisse arithmetischer Operationen. Welche Auswirkungen dies sind, sollen folgende Ausführungen anhand von Beispielen verdeutlichen (nicht dargestellte Ziffern werden abgeschnitten). Eine Verdoppelung der Operator-Zeichen (z. B. ++ statt +) in den folgenden Beispielen gibt an, dass die Operation auf einem Computersystem ausgeführt wird.

Addition

Nach den Regeln der Potenzrechnung können die Mantissenteile zweier Zahlen nur addiert werden, wenn die Exponenten gleich sind. Der Mantissenteil des Operanden mit dem kleineren Exponenten muss daher um so viele Stellen nach rechts geschoben werden wie der Betrag der Differenz angibt. Dann können die Mantissenteile der beiden Operanden addiert werden.

Beispiel 1

```
    54,3          0.543E2
+ 4560       ++ 0.456E4
= 4614,3
```

Da der Exponent des 1. Operanden kleiner als der des 2. ist, wird dieser Mantissenteil um 4–2 = 2 Stellen nach rechts geschoben:

```
    50            0.005E4
+ 4560          ++ 0.456E4
= 4610           = 0.461E4
```

Jetzt können die Mantissenteile addiert werden.

Werden Zahlen *sehr unterschiedlicher* Größe addiert, dann gehen die Ziffern der kleineren Zahl durch das Hinausschieben verloren.

Beispiel 2

```
  10000            0.100E5
+     0,851     ++ 0.851E0
```

Angleichung der Exponenten:

```
  10000            0.100E5
+     0          ++ 0.000E5
= 10000           = 0.100E5
```

Nach der Addition kann der Mantissenteil des Ergebnisses auch größer als 1 werden. Es erfolgt dann eine Normalisierung, wobei der Mantissenteil um eine Stelle nach rechts verschoben und der Exponent um 1 erhöht wird.

Beispiel 3

```
    54,3            0.543E2
+   72,3          ++ 0.723E2
=  126,6           = 1.266E2
                   = 0.126E3
```

Wie die Beispiele zeigen, werden bei der Angleichung der Exponenten und bei der Normalisierung *rechts Ziffern abgeworfen*. In diesen Fällen wird vom Computersystem automatisch eine Rundung vorgenommen.

Subtraktion

Bei der Subtraktion sind – wie bei der Addition – zuerst die Exponenten anzugleichen, bevor die Mantissenteile voneinander subtrahiert werden.

Beispiel 4

```
  0.000456          0.456E-3
- 0,00000543     -- 0.543E-5
```

Der kleinere Exponent E-5 wird angeglichen.

Beispiel 5

```
  0.000456          0.456E-3
- 0,000005       -- 0.005E-3
= 0,000451         = 0.451E-3
```

Beispiel 6

```
  653              0.653E3
- 651            -- 0.651E3
=   2             = 0.002E3
                  = 0.200E1
```

Sind beide Operanden nur wenig voneinander verschieden, so heben sich bei der Subtraktion die *signifikantesten Ziffern auf* (in Beispiel 6 sind dies die Ziffern 6 und 5). Die Differenz verliert dadurch an Signifikanz (Anzahl bedeutungsvoller Stellen).

Auslöschung

Diese Erscheinung bezeichnet man als **Auslöschung** führender Ziffern *(cancellation)*. Durch die anschließende Normalisierung (Verschieben der Mantisse nach links und Verkleinerung des Exponenten) erscheinen Nullen, die eine nicht vorhandene **Genauigkeit** vortäuschen.

Führen Sie folgende Subtraktionen aus:

```
1.     0.126E12          2. - 0.521E-2
     - 0.375E10             - 0.972E-3
```

Multiplikation

Gleitkommazahlen werden multipliziert, indem die Mantissenteile multipliziert und die Exponenten addiert werden. Das Produkt ist gegebenenfalls zu normalisieren (Linksverschiebung des Mantissenteils und Verkleinerung des Exponenten).

Beispiel

```
1300 * 0,028 = 36,4

0.130E4 ** 0.280E-1 = ?
Addition der Exponenten:            4 + (-1) = 3
Multiplikation der Mantissenteile:  0.130 ** 0.280 = 0.0364
Normalisierung:                     Aus 0.0364 wird 0.364
                                    Aus 3 wird 2
Ergebnis:                           0.364E2 = 36.4
```

Multiplizieren Sie folgende Zahlen:

```
1.   0.355E-4 * -0.532E-12        2.   -0,128E2 * -0.672E0
```

Neben Rundungsfehlern können auch Zahlenbereichsüberschreitungen bei Exponenten auftreten.

Beispiele

Multiplikationsüberlauf: `0.842E40 ** 0.200E62 = 0.168E102`
Multiplikationsunterlauf: `0.842E-40 ** 0.200E-62 = 0.168E-102`

Division

Zwei Gleitkommazahlen werden dividiert, indem die Mantissenteile dividiert und die Exponenten subtrahiert werden. Der Quotient ist gegebenenfalls zu normalisieren.

Beispiel

```
0,0234 / 1300 = 0,000018

0.234E-1 // 0.130E4 = ?
Subtraktion der Exponenten:  (-1) - 4 = - 5
Division der Mantissenteile: 0.234 // 0.130 = 1.800
Normalisierung:              Aus 1.800 wird 0.180
                             Aus -5 wird -4
Ergebnis:                    0.180E-4 = 0.000018
```

Neben Rundungsfehlern können auch Zahlenbereichsüberschreitungen bei Exponenten auftreten.

Beispiele

Divisionsüberlauf: `0.843E96 // 0.200E-4 = 0.421E100`
Divisionsunterlauf: `0.842E-96 // 0.200E4 = 0.421E-100`

Division durch kleine Werte

Insbesondere ist die Division durch *kleine* Werte gefährlich, da das Resultat leicht im Überlaufbereich liegen kann.

Bei der Durchführung der Grundoperationen mit Gleitpunktzahlen ist also an folgende Regeln zu denken:

- Addition von Gleitpunktzahlen sehr unterschiedlicher Größe vermeiden (Stellen der kleinen Zahlen gehen verloren)!
- Subtraktion von dicht benachbarten Gleitpunktzahlen vermeiden (Auslöschung)!
- Division durch Werte in der Nähe von Null vermeiden (Überlaufgefahr)!

Beispiel

Die mangelnde Genauigkeit beim Rechnen mit Gleitpunktzahlen demonstriert das folgende Beispiel. Es wird die Zahl `1` einmal als Produkt von `0.1 * 10` berechnet und einmal als Summe von `10` Werten à `0.1`:

Rechen genauigkeit

```
public class Rechengenauigkeit
{
 public static void main (String args[])
 {
  double summe = (0.1 + 0.1 + 0.1 + 0.1 + 0.1 +
    0.1 + 0.1 + 0.1 + 0.1 + 0.1); //sollte 1 ergeben
  double produkt = 10.0 * 0.1; //sollte 1 ergeben

  System.out.println("Summe: " + summe);
  System.out.println("Produkt: " + produkt);
  System.out.println
    ("Produkt - Summe: " + (produkt - summe));
  System.out.println("Produkt - Summe < Eps: "
   + (Math.abs(produkt - summe) < 1e-10));
 }
}
```

Das Ergebnis sieht wie folgt aus:

```
Summe: 0.9999999999999999
Produkt: 1.0
```

```
Produkt - Summe: 1.1102230246251565E-16
Produkt - Summe < Eps: true
```

Wie die Ausgabe zeigt, ist die Differenz extrem gering, aber vorhanden.

Abfrage auf Gleichheit

Wegen der *nicht* exakten Zahlendarstellung von reellen Zahlen ist in einem Programm die Abfrage auf Gleichheit von zwei Gleitpunktzahlen zu vermeiden (z. B. `wenn A == B dann`), da diese Gleichheit wegen der Rundungsfehler u. U. nie eintritt.

Eine bessere Abfrage hat z. B. die Form »wenn | A – B | < ε dann...«, wobei ε eine passende gewählte Zahl ist. Als ein Maß für die Genauigkeit der Gleitpunkt-Arithmetik kann eine Größe ε angenommen werden. ε ist die kleinste positive Zahl, bei der 1 und (1 ++ ε) verschieden sind. Gestattet ein Computersystem eine Darstellung von Werten des Typs `float` mit einer Genauigkeit von n dezimalen Stellen, dann ist $\varepsilon = 10^{-(n-1)}$.

Beispiel

Annahme: Genauigkeit = 3 dezimale Stellen.
Dann ist $\varepsilon = 10^{-2} = 1 / 100$.

```
Die Zahl 1          = 0.100E1
und 1 ++ Eps        = 0.100E1
                   ++ 0.001E1
                    = 0.101E1
```

Programmierregel

Da bei Gleitpunktzahlen Rundungsfehler auftreten können, dürfen zwei Gleitpunktzahlen *niemals* auf Gleichheit überprüft werden, d. h. `x == y` ist verboten. Stattdessen wird die Differenz gebildet, davon der Absolutwert genommen (mit der Operation `Math.abs()`) und auf einen tolerierbare Differenzbetrag, oft Epsilon genannt, abgeprüft: `Math.abs(x - y) < Epsilon`.

3.7 Eingeschränkte Mathematikgesetze ***

Das Assoziativ- und das Distributivgesetz sind bei Gleitpunktzahlen *nicht* mehr gültig. Bei Operationen mit Gleitpunktzahlen in Computersystemen muss man jedoch von folgenden vorhandenen Eigenschaften ausgehen: Kommutativität der Addition und Multiplikation, Symmetrie bzgl. 0, Monotonie der Grundoperationen.

Durch die beschränkte Stellenzahl sind das **Assoziativgesetz** und das **Distributivgesetz** der Arithmetik beim Rechnen mit Gleitpunktzahlen *nicht* mehr gültig.

Assoziativgesetz

Das Assoziativgesetz der Addition (a + b) + c = a + (b + c) besagt, dass die Reihenfolge der Auswertung der Terme *keine* Rolle spielt.

Beispiel

Das Beispiel der Abb. 3.7-1 zeigt jedoch, dass (a ++ b) ++ c ≠ a ++ (b ++ c) sein kann. Wie man sieht, wird der Fehler dann gravierend, wenn die Exponentendifferenz die Mantissenlänge erreicht. Hinweis: Eine Verdopplung der Operator-Zeichen gibt an, dass die Operation auf einem Computersystem ausgeführt wird.

```
(a ++ b) ++ c     =   (0.789E6 ++ 0.312E3) ++ 0.792E3
                  =    0.789E6
                      ++ 0.000E6
                  =    0.789E6
                      ++ 0.000E6
                  =    0.789E6

a ++ (b ++ c)     =    0.789E6 ++ (0.312E3 ++ 0.792E3)
                       0.789E6 ++ (0.312E3
                                  ++  0.792E3)
                                =   0.110E4
                  =    0.789E6
                      ++ 0.001E6
                  =    0.790E6
```

Abb. 3.7-1: Das Beispiel zeigt, dass das Assoziativgesetz bei Gleitpunktzahlen nicht mehr gelten kann.

Distributivgesetz

Das Distributivgesetz lautet: a * (b + c) = (a * b) + (a * c)

Beispiel

Das Beispiel der Abb. 3.7-2 zeigt, dass für das Rechnen mit numerischen Werten a ** (b ++ c) ≠ (a ** b) ++ (a ** c) sein kann.

```
a ** (b ++ c)      =   0.200E1 ** (0.501E5 ++ (-0.500E5))
                   =   0.200E1 ** (0.501E5
                                  ++ -0.500E5
                                     0.001E5
                   =   0.200E3

(a ** b) ++ (a ** c)   =   (0.200E1 ** 0.501E5) ++ (0.200E1 ** (-0.500E5))
                       =    0.100E6 ++ (-0.100E6)
                       =    0.000E6
```

Abb. 3.7-2: Das Beispiel zeigt, dass bei Gleitpunktzahlen das Distributivgesetz verletzt sein kann.

Die Ungültigkeit wichtiger arithmetischer Gesetze zeigt, dass in Programmen die Gültigkeit von mathematischen Gesetzen und Aussagen *nicht* ohne Weiteres vorausgesetzt werden dürfen.

Beispiel

Annahme: In einem Computersystem können Gleitpunktzahlen mit 6-stelligem Mantissenteil und einem Exponentenbereich von $-99 \leq e \leq +99$ dargestellt werden.

Die größte darstellbare Zahl ist damit 0.999 999 E99. Da dies eine ganze Zahl ist, müsste man mit einem Programm, das beginnend bei einer Summe 1 fortlaufend auf die jeweilige Summe eine 1 addiert, die größte Zahl erreichen. Dies ist jedoch nicht der Fall. Denn ist die Summe

```
    0.999 999 E 6
```

erreicht und wird jetzt eine 1 addiert,

```
    0.999 999 E 6
++ 0.100 000 E 1
```

dann ergibt sich

```
    0.999 999 E 6
++ 0.000 001 E 6
 = 0.100 000 E 7.
```

Bei der nächsten Addition erhält man jedoch:

```
    0.100 000 E 7
++ 0.100 000 E 1
```

daraus wird:

```
    0.100 000 E 7
++ 0.000 000 E 7
 = 0.100 000 E 7
```

An dieser Stelle wird das Computersystem weiter addieren, ohne dass sich am Ergebnis etwas ändert, da die Mantissenlänge nicht mehr ausreicht.

Automatische Normalisierung

Die aufgeführten Beispiele sollten zeigen, welche Effekte bei Gleitpunktoperationen auftreten können und wie Gleitpunktzahlen im Computersystem dargestellt werden. Eine Normalisierung wird vom Compiler oder dem Computersystem vorgenommen. Der Vorgang wurde hier nur erklärt, um die Wirkung zu veranschaulichen.

Ausblick

Obwohl die elementaren Rechenoperationen mit Gleitpunktzahlen nur schwer exakt charakterisiert werden können, so ist es doch einfach möglich, **Minimalbedingungen** anzugeben, die als gültig angenommen werden dürfen.

9 Axiome

Im Folgenden werden neun **Axiome** aufgeführt[1], die jeweils anschließend durch Beispiele veranschaulicht werden.

[1] Die Aufzählung dieser Axiome orientiert sich an N. Wirth »Systematisches Programmieren«, S. 55 f., Stuttgart 1972

A 1 Der numerisch-reelle Wertebereich eines Computersystems (bezeichnet mit R) ist eine endliche Untermenge der reellen Zahlenmenge $\mathbb{R}$: $R \subset \mathbb{R}$

Beispiel
$0.999E99 \in R$
$0.999E{-}99 \in R$

A 2 Jeder Zahl $x \in \mathbb{R}$ ist eindeutig ein Repräsentant $x' \in R$ zugeordnet.

Beispiel $x = 1.2345 \Rightarrow x' = 0.123E1$

A 3 Jedes $x' \in R$ repräsentiert mehrere x, aber der von x' repräsentierte Bereich ist zusammenhängend. Außerdem gilt: $x \in R \Rightarrow x' = x$. Insbesondere sollen 0 und 1 exakt repräsentiert sein, d. h. $0 \in R$, $1 \in R$ und daher $0' = 0$ und $1' = 1$.

Beispiel $x = 0.123 \in R \Rightarrow x' = 0.123E0 = x$

A 4 Es gibt einen größten Wert MAX, so dass $x' = MAX$ für alle $|x| > MAX$. Alle Zahlen $|x| > MAX$ befinden sich im Überlaufbereich U. $\mathbb{R} \setminus U$ ist zusammenhängend. Daraus folgt (wenn $x, y \in \mathbb{R} \setminus U$):
$x < y \Rightarrow x' \leq y'$
$x = y \Rightarrow x' = y'$
$x > y \Rightarrow x' \geq y'$

Beispiele
$1.11111 < 2.22222 \Rightarrow 0.111E1 \leq 0.222E1$
$1.12345 = 1.12345 \Rightarrow 0.112E1 = 0.112E1$
$1.1234 > 1.1233 \Rightarrow 0.112E1 \geq 0.112E1$

A 5 R soll symmetrisch bezüglich 0 sein, d. h. $(-x)' = -(x')$

Folgende Eigenschaften sind für eine **brauchbare Computer-Arithmetik** erforderlich (Die Operatoren werden mit ++, $--$, ** und // bezeichnet). Es wird $x, y \in R$ vorausgesetzt.

A 6 **Kommutativität** der Addition und Multiplikation:
$x \mathbin{++} y = y \mathbin{++} x$, $x \mathbin{**} y = y \mathbin{**} x$

A 7 $(x \geq 0) \Rightarrow (x \mathbin{--} y) \mathbin{++} y = x$

A 8 **Symmetrie** der Grundoperationen bezüglich 0:
$x \mathbin{--} y = x \mathbin{++} (-y) = -(y \mathbin{--} x)$
$(-x) \mathbin{**} y = x \mathbin{**} (-y) = -(x \mathbin{**} y)$
$(-x) \mathbin{//} y = x \mathbin{//} (-y) = -(x \mathbin{//} y)$

A 9 **Monotonie** der Grundoperationen
Wenn $0 \leq x \leq a$ und $0 \leq y \leq b$, dann gelte stets auch:
$x \mathbin{++} y \leq a \mathbin{++} b$
$x \mathbin{**} y \leq a \mathbin{**} b$
$x \mathbin{--} b \leq a \mathbin{--} y$
$x \mathbin{//} b \leq a \mathbin{//} y$

Für gewisse $0 \leq x < a$ und $0 \leq y < b$ sei
$x \mathbin{++} y \leq a \mathbin{++} b$ oder $x \mathbin{**} y \leq a \mathbin{**} b$, jedoch *niemals*
$x \mathbin{++} y > a \mathbin{++} b$ oder $x \mathbin{**} y > a \mathbin{**} b$ möglich.

Aus den Axiomen lassen sich folgende **wichtige Gesetze** herleiten:

- $y \geq 0 \Rightarrow x \mathbin{++} y \geq x$
- $x \geq y \Rightarrow x \mathbin{--} y \geq 0$
- $(x \geq 0)$ und $(0 \leq y \leq 1) \Rightarrow x \mathbin{**} y \leq x$
- $0 < x \leq y \Rightarrow x \mathbin{//} y \leq 1$
- $x \mathbin{--} x = 0$
- $x \mathbin{++} 0 = x \mathbin{--} 0 = x$
- $x \mathbin{**} 0 = 0$
- $x \mathbin{**} 1 = x \mathbin{//} 1 = x$
- $x \mathbin{//} x = 1$

3.8 Der Zeichentyp char *

Der Zeichentyp `char` umfasst in Java den Wertebereich von 0 bis 65535 und erlaubt es damit alle 16 Bit-Unicode-Zeichen zu speichern. Zeichen werden in einfachen Anführungszeichen, z. B. `char Zchn = 'X'`, oder als ASCII-Dezimalwert, z. B. `Zchn = 88`, oder als Unicode-Hexadezimalzeichen, z. B. `Zchn = '\u0058'`, dargestellt. Spezielle Zeichen beginnen mit einem Rückwärtsschrägstrich *(backslash)*, z. B. `'\n'` für einen Zeilenvorschub.

Unicode

Der Typ `char` erlaubt es, ein sogenanntes Unicode-Zeichen in der so deklarierten Variablen oder Konstanten zu speichern. Der **Unicode** (UCS) (siehe Unicode Homepage (`http://www.unicode.org/`)) wurde seit 1987 von den Firmen Apple und Xerox entwickelt, um die Schriftzeichen aller Verkehrssprachen der Welt aufzunehmen. Sowohl alphabetische als auch syllabische und logographische Schriften sollten darstellbar sein. Der Unicode beruht ursprünglich auf 16 Bits (2 Byte) und konnte damit 65536 Positionen zur Verfügung stellen. Im Juni 1992 wurde er zur internationalen Norm (ISO[2]/IEC 10646–1). In den folgenden Jahren wurde das Unicode-System weiterentwickelt. Im April 2008 erschien die Version 5.1.0, die 100.000 Zeichen codiert. Der Unicode gliedert sich in Codebereiche zu jeweils 16 Bits, sogenannte Ebenen *(planes)*. Die erste Ebene (*plane 0* genannt) ist die wichtigste Ebene *(basic multilingual plane)*.

UTF-16

Der Unicode wird in unterschiedlichen Formaten (*encodings* genannt) gespeichert und übertragen. Java verwendet UTF-16 (*Unicode Transformation Format,* 16 Bits). Will man ein Unicode-Zeichen in Java darstellen, das mehr als 16 Bits umfasst, dann

[2] ISO = International Organization for Standardization, Genf

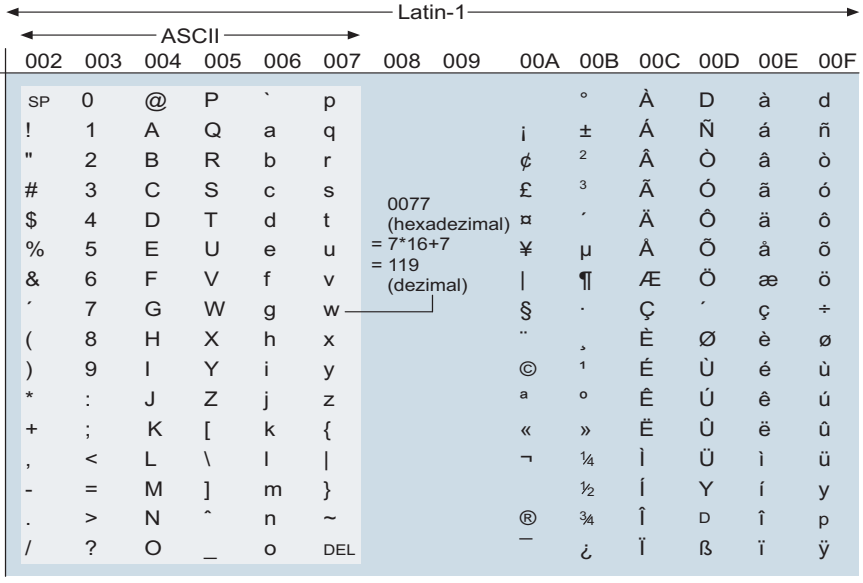
muss man sogenannte *Supplementary Characters* verwenden (siehe Java Supplementary Characters im Web). Um dies zu ermöglichen wurden neue Funktionen und Methoden für den Zugriff auf Zei-chenwerte geschaffen, die mit `int`-Werten (`signed 32bit`) arbeiten.

ASCII Eine Teilmenge des Unicode ist der bekannte ASCII-Code *(American Standard Code for Information Interchange)*. Beim ASCII-Code (genauer gesagt beim USASCII-Code) handelt es sich um den 1963 genormten amerikanischen 7-Bit-Zeichensatz für den Fernschreibverkehr und den Datenaustausch zwischen Computersystemen. Von den 128 Positionen benutzt er nur 100. Er hatte nur Großbuchstaben. Er umfaßt also – grob gesagt – alle Zeichen, die sich auf einer englischen PC-Tastatur befinden, d. h. es gibt keine länderspezifischen Sonderzeichen wie Umlaute oder ß.

ECMA-6 1965 beschloss die ECMA[3] eine Normzeichentabelle, ECMA-6. Sie entsprach im Wesentlichen ASCII, fügte aber auf den unbesetzten Positionen die 26 Kleinbuchstaben hinzu. 10 Positionen sollten nationalen Sonderzeichen zur Verfügung stehen (siehe Abb. 3.8-1). Diese Tabelle wurde 1968 vom ANSI[4] als X3.4 übernommen und zum amerikanischen Normzeichensatz. 1974 wurde sie internationale Norm (ISO 646, revidiert 1983 und 1991) und gleichzeitig deutsche Norm (DIN 66003).

Latin Durch Hinzunahme eines achten Bits zu einem 7-Bit-Code wie ASCII ergeben sich 128 weitere Positionen in der Zeichentabelle. Diese werden heute auf verschiedene Weise genutzt. 1981 führte IBM für DOS-Computersysteme die Codepage 437 ein, die die hinzugewonnenen Plätze für den amerikanischen Datenverkehr nutzte, aber auch die großen westeuropäischen Verkehrssprachen berücksichtigte. Im März 1985 beschloss ECMA einen 8-Bit-Code mit 256 Zeichen, der alle westeuropäischen Sprachen abdeckte (ECMA-94). Er wurde 1986 von der ISO zur Weltnorm gemacht (ISO 8859–1, genannt **Latin-1**, siehe Abb. 3.8-1) und ab 1987 nach und nach durch neun weitere Zeichensätze, zum Teil auch für nichtlateinische Alphabete, ergänzt (ISO 8859–1 bis 10) – die Nummer 2 (Latin-2) deckt die mittelosteuropäischen ab. Die IBM-Codepage 819 für DOS ist identisch mit Latin-1. Meist wird mit DOS dagegen Codepage 850 verwendet, die sich mit Latin-1 im Zeichenvorrat deckt, die Zeichen aber in anderer Reihenfolge anordnet. Das Betriebssystem Windows enthält sämtliche Zeichen aus Latin-1 und noch einige mehr. Die deutsche Entsprechung zu ISO 8859–1 ist DIN 66303 (2000).

[3] ECMA = European Computer Manufacturers Association, Genf
[4] ANSI = American National Standards Institute, New York



	002	003	004	005	006	007	008	009	00A	00B	00C	00D	00E	00F
0	SP	0	@	P	`	p				°	À	D	à	d
1	!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ
2	"	2	B	R	b	r			¢	²	Â	Ò	â	ò
3	#	3	C	S	c	s			£	³	Ã	Ó	ã	ó
4	$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô
5	%	5	E	U	e	u			¥	µ	Å	Õ	å	õ
6	&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö
7	´	7	G	W	g	w			§	·	Ç	´	ç	÷
8	(	8	H	X	h	x			¨	¸	È	Ø	è	ø
9	)	9	I	Y	i	y			©	¹	É	Ù	é	ù
A	*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú
B	+	;	K	[	k	{			«	»	Ë	Û	ë	û
C	,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü
D	-	=	M	]	m	}				½	Í	Y	í	y
E	.	>	N	^	n	~			®	¾	Î	D	î	p
F	/	?	O	_	o	DEL			¯	¿	Ï	ß	ï	ÿ

SP: Space (Leerschritt, Zwischenraum) DEL: Delete (Löschen)
Linke Tabellenhälfte: Die darstellbaren Zeichen aus der Zeichentabelle »ASCII« (ISO 646).
Beide Hälften: Die darstellbaren Zeichen aus der Zeichentabelle »Latin-1« (ISO 8859-1).

Abb. 3.8-1: Die linke Tabellenhälfte zeigt den ASCII-Code. Die gesamte Tabelle zeigt den Latin-1-Zeichensatz.

Lesen der Code-Tabelle

Die Abb. 3.8-1 enthält die Zeichen und Sonderzeichen des Latin-1-Codes, jedoch keine Steuerzeichen. Um den Hexadezimalcode eines Zeichens zu erhalten, gehen Sie in die gewünschte Spalte und lesen oben in der Tabelle die ersten drei Hexadezimalziffern ab. Die letzte Hexadezimalziffer finden Sie am linken Rand in der Zeile, in der das Zeichen steht. Das Zeichen w finden Sie z. B. in der 7. Spalte mit den Hexadezimalzeichen 007 und in der 7. Zeile mit der Hexadezimalziffer 7, ergibt zusammen 0077 für w. Die letzten beiden Ziffern bilden den ASCII-Wert, also 77 (hexadezimal), das ergibt den Wert 7 * 16 + 7 = 119 (dezimal).

In den Positionen 0020 (hexadezimal) = 32 (dezimal) bis 007E (hexadezimal) = 126 (dezimal) stimmen Unicode und ASCII überein.

Die Tab. 3.8-1 zeigt den plattformunabhängigen Wertebereich von char. Es gibt keine negativen Zeichen.

Typ	Kleinster Wert	Größter Wert	Speicherplatz	Voreinstellung
char	\u0000 (0)	\uffff (65535)	2 Bytes	\u0000

Tab. 3.8-1: Der Zeichentyp char in Java.

Beispiel

Das €-Zeichen hat den Unicode-Wert `'\u20AC'`.

Darstellungsarten

Folgende Darstellungsarten für den Typ `char` sind in Java möglich:

- Darstellung in einfachen Anführungszeichen, z. B. `'A'`.
- Darstellung in Hexadezimalform in der Syntax `'\uXXXX'`, wobei `\u` der Präfix für einen Unicode-Wert und `X` ein Hexadezimalzeichen ist, z. B. `'\u00A9'` repräsentiert das Copyright-Zeichen.
- Spezielle Zeichen werden mit vorangestellten `\` geschrieben, z. B. `'\n'` (neue Zeile) (Tab. 3.8-2).

Fluchtzeichen *(escape sequence)*	**Bezeichnung**	**Unicode-Wert**
`\b`	Rücktaste *(backspace)*	`\u0008`
`\t`	Tabulator	`\u0009`
`\n`	Zeilenvorschub *(linefeed)*	`\u000a`
`\r`	Wagenrücklauf *(carriage return)*	`\u000d`
`\"`	doppeltes Anführungszeichen *(double quote)*	`\u0022`
`\'`	einfaches Anführungszeichen *(single quote)*	`\u0027`
`\\`	Rückwärtsschrägstrich (von links oben nach rechts unten) *(backslash)*	`\u005c`

Tab. 3.8-2: Spezielle Zeichen und ihre Darstellung.

Hinweis

Theoretisch können Sie alle Unicode-Werte verwenden. Welche angezeigt werden, hängt jedoch vom Web-Browser (bei Java-Applets) und vom verwendeten Betriebssystem oder von beiden ab.

Operationen

Obwohl `char`-Werte keine `int`-Werte sind, kann in vielen Fällen mit ihnen gearbeitet werden, als wären sie ganze Zahlen. Beispielsweise können zwei Zeichen addiert oder ein Zeichen inkrementiert werden. Außerdem können Zeichen mithilfe von Vergleichsoperationen miteinander verglichen werden.

char vs. String

Verwechseln Sie nicht `char`-Zeichen und `String`-Zeichenketten! Der Typ `String` ist in Java *kein* einfacher Typ. Er dient zur Aufnahme von Zeichenketten, die in doppelte Anführungszeichen eingeschlossen werden, während Zeichen durch einfache Anführungszeichen repräsentiert werden.

Beispiel

DemoChar

```
//Beispiele für den Typ char in Java
import inout.Console;
public class DemoChar
{
 public static void main (String args[])
 {
   char zchn1, zchn2, zchn3, tab;
   zchn1 = '\u0057'; //hexadezimal
   zchn3 = 88; //dezimal
   tab = '\t'; //Tabulatorzeichen
   System.out.println
    ("Zeichen zu Uni-Code 0057 (hexadezimal):" + tab + zchn1);
   System.out.println
    ("Zeichen zu ASCII-Code 88 (dezimal):" + tab + zchn3);
   System.out.println("Bitte ein Zeichen eingeben:");
   zchn2 = Console.readChar();
   System.out.println
     ("Eingegebenes Zeichen: " + tab + tab + tab + zchn2);
   //Rechnen mit ASCII-Werten
   zchn2++;
   System.out.println
    ("Nächstes Zeichen in der Code-Tabelle: "+ tab + zchn2);
 }
}
```

Folgende Ergebnisse werden ausgegeben:

```
Zeichen zu Uni-Code 0077 (hexadezimal): w
Zeichen zu ASCII-Code 88 (dezimal): X
Bitte ein Zeichen eingeben: 4
Eingegebenes Zeichen:           4
Nächstes Zeichen in der Code-Tabelle:     5
```

Modifizieren Sie das Programm `DemoChar` durch Zuweisung eigener Werte. Probieren Sie die Wirkung der speziellen Zeichen aus.

3.9 Operatorprioritäten *

Klammern können in Ausdrücken eingespart werden, wenn die Operatorprioritäten berücksichtigt werden. In Java gilt die Reihenfolge (von hoher zu niedriger Priorität): 1. Inkrement- und Dekrement-Operationen, 2. Arithmetische Operationen, 3. Vergleichsoperationen, 4. Boolesche Operationen, 5. Zuweisungsoperationen.

Ein Ausdruck setzt sich aus Operanden, d. h. Variablen und Konstanten, und Operatoren zusammen. Jeder Operand kann selbst wieder ein Ausdruck sein. Kommt in einem Ausdruck mehr als ein Operator vor, so muss die Reihenfolge der Ausführung definiert sein.

Prioritäten

Dies geschieht durch festgelegte Vorrangregeln – Prioritäten – für die Ausführungsreihenfolge der Operatoren. In Java gelten folgende allgemeine Regeln:

1 Inkrement- und Dekrement-Operationen
2 Arithmetische Operationen
3 Vergleichsoperationen
4 Boolesche Operationen
5 Zuweisungsoperationen

Die Tab. 3.9-1 gibt die exakte Prioritätenfolge an.

Operator	Bemerkungen
`++ --! ~ + -`	Unäre Operatoren
`(type)expression`	Dient der Typwandlung *(casting)*
`* / %`	Multiplikation, Division, Modulo
`+ -`	Addition, Subtraktion
`<< >> >>>`	Bitweises Verschieben nach links und rechts
`< <= > >=`	Vergleich
`== !=`	Gleichheit, Ungleichheit
`&`	logisches UND
`^`	logisches XODER
`|`	logisches ODER
`&&`	logisches UND (verkürzt)
`||`	logisches ODER (verkürzt)
`?:`	Tenärer?-Operator
`= += -= *= /= %=`	Zuweisungen, z. B. `a += b;` entspricht `a = a + b;`

Tab. 3.9-1: Operatorprioritäten in Java (oben = höchste Priorität).

Abarbeitungsreihenfolge

Operatoren auf derselben hierarchischen Stufe werden von links nach rechts abgearbeitet. Ausnahmen sind die unären Operationen und die Zuweisungen. Sie werden von rechts nach links ausgeführt.

Klammern

Um eine andere Ausführungsreihenfolge zu erhalten, können Ausdrücke in Klammern eingeschlossen werden. Geklammerte Ausdrücke werden immer zuerst ausgewertet.

Boolesche Operationen werden hauptsächlich auf logische Ausdrücke angewandt, die sich durch die Vergleichsoperatoren aus arithmetischen Ausdrücken ergeben.

Beispiel

DemoBrief

```
// Ein Standardbrief der Post hat ein Gewicht bis 20 g,
// eine Länge zwischen 14 und 23,5 cm, eine Breite
// zwischen 9 und 12,5 cm und eine Höhe bis 0,5 cm.
//...
boolean standardsendung = false;
```

```
double gewicht = 0.0, laenge = 0.0, breite = 0.0, hoehe = 0.0;
//Anweisung mit booleschen Operationen
standardsendung = gewicht <= 20.0 && 14.0 >= laenge
    && laenge <= 23.5 && 9.0 <= breite
    && breite <= 12.5 && hoehe <= 0.5;
// ...
```

Da die Vergleichsoperationen `<=` und `>=` eine höhere Priorität als der Operator `&&` haben, werden zuerst die Vergleichsoperationen ausgeführt. Die Zuweisung `=` hat die geringste Priorität und wird erst nach der Operation `&&` ausgeführt.

3.10 Typumwandlungen *

Bei Zuweisungen und in Ausdrücken müssen die Typen der Variablen, Konstanten und Literalen miteinander kompatibel und der Wertebereich des Zieltyps muss größer sein als der Quelltyp. Ist dies der Fall, dann wird von Java eine automatische Typausweitung vorgenommen. Ist dies *nicht* der Fall, dann muss der Programmierer eine explizite Typeinengung *(casting)* vornehmen durch `(Zieltyp) Quelltyp`.

Typ bestimmt Wertebereich

Ein Kennzeichen einer Variablen und einer Konstanten ist sein Typ. Ein Typ lässt sich charakterisieren durch die Werte oder Wertebereiche, die einer Variablen bzw. Konstanten dieses Typs zugewiesen werden können, und durch die Operationen, die auf die Werte dieses Typs angewandt werden können.

Das bedeutet, dass einer Variablen bzw. einer Konstanten auch nur Werte zugewiesen werden können, die entsprechend ihrem Typ möglich sind. Bildlich kann man sich das dadurch vorstellen, dass für jeden Typ eine charakteristische Speicherzellenform und Speicherzellengröße vorhanden sind.

Beispiel

```
int a; float b = 10.5f;
a = b; // Fehler wegen unterschiedlicher Typen
```

Dieses Beispiel zeigt deutlich, dass eine Zuweisung eines `float`-Wertes an eine `int`-Variable *nicht* sinnvoll ist. Es ist unklar, ob die Stellen hinter dem Komma abgeschnitten werden sollen, oder ob eventuell gerundet werden soll (Abb. 3.10-1).

Prinzipiell ist es daher zunächst *nicht* möglich, dass einer Variablen Werte zugewiesen werden, die zu einem anderen Typ gehören.

Java

Java ist eine streng typisierte Sprache. Daher können in Ausdrücken und bei Zuweisungen nur Variablen und Konstanten miteinander verknüpft werden, die vom gleichen Typ sind.

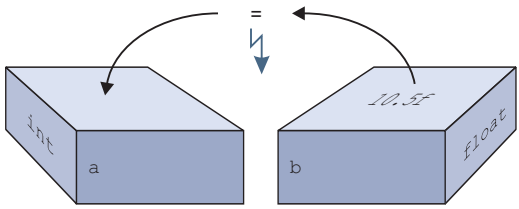


Abb. 3.10-1: Ein Wert vom Typ `float` kann nicht einer Variablen vom Typ `int` zugewiesen werden, da sonst ein Genauigkeitsverlust eintritt.

Typumwandlung

Befinden sich in einem Ausdruck Variablen bzw. Konstanten *unterschiedlichen* Typs, dann müssen die Werte der Variablen bzw. Konstanten in einen einheitlichen Typ umgewandelt werden. Dies geschieht in Java durch **Typumwandlungen** *(conversions)*. Es werden automatische Typausweitungen und explizite Typeinengungen unterschieden.

Automatische Typausweitung

Folgende Typausweitungen werden in Java automatisch vorgenommen:

- Von `byte` nach `short`, `int`, `long`, `float` oder `double`
- Von `short` nach `int`, `long`, `float` oder `double`
- Von `char` nach `int`, `long`, `float` oder `double`
- Von `int` nach `long`, `float` oder `double`
- Von `long` nach `float` oder `double`
- Von `float` nach `double`

Bei jeder dieser Typausweitungen geht *keine* Information über die Größe eines numerischen Werts verloren. Die Umwandlung eines Werts vom Typ `int` oder `long` nach `float` oder die eines Werts vom Typ `long` nach `double` kann jedoch zu einem Genauigkeitsverlust führen. Die Typen `char` und `boolean` sind *nicht* miteinander kompatibel.

Beispiel

```
double a, b;
float c;
a = b + c + 2.785f;
// Der Typ float der Variablen c und des Literals 2.785f
// werden für die Berechnung des Ausdrucks automatisch
// in double konvertiert (Typ der Variablen a und b)
```

Regeln

Java verwendet folgende Regeln bei der Auswertung von Ausdrücken:

- Enthält ein Ausdruck `byte`- und/oder `short`-Typen, dann werden die `byte`- und `short`-Typen vor der Berechnung in `int`-Typen gewandelt. Die Berechnung findet also mit `int`-Typen statt.

- Ist ein Operand vom Typ `long`, dann wird der gesamte Ausdruck mit `long`-Typen berechnet.
- Ist ein Operand vom Typ `float`, dann wird der gesamte Ausdruck mit `float`-Typen berechnet.
- Ist ein Operand vom Typ `double`, dann wird der gesamte Ausdruck mit `double`-Typen berechnet.

Innerhalb eines Ausdrucks erfolgt zwar eine automatische Typausweitung, aber *nicht* bezogen auf die Zuweisung.

Beispiel

Sie arbeiten mit `int`-Variablen in einem arithmetischen Ausdruck. Das Ergebnis soll einer `double`-Variablen zugewiesen werden.

Demo Typausweitung

```
public class DemoTypausweitung
{
 public static void main (String args[])
 {
  int wertInt1 = 1235, wertInt2 = 524;
  double wertDouble1, wertDouble2;
  wertDouble1 = wertInt1 / wertInt2; //Fall1
  System.out.println("WertDouble1: " + wertDouble1);
  wertDouble2 = (double)wertInt1 / (double)wertInt2; //Fall2
  System.out.println("WertDouble2: " + wertDouble2);
 }
}
```

Die Ergebnisse sehen folgendermaßen aus:

```
WertDouble1: 2.0
WertDouble2: 2.3568702290076335
```

Im ersten Fall erfolgt eine Ganzzahldivision, d. h. als Ergebnis wird nur der ganzzahlige Anteil verwendet und dann der Variablen `wertDouble1` zugewiesen.

Im zweiten Fall erfolgt durch die Angabe `(double)` vor der jeweiligen Variablen eine explizite Typausweitung auf einen `double`-Wert (ohne den Wert der `int`-Variablen selbst zu verändern), dann erfolgt eine Gleitpunkt-Division und dann die Zuweisung des Gleitpunkt-Ergebnisses an die `double`-Variable.

Frage

Erweitern Sie das Programm `DemoTypausweitung` um folgende Anweisung und führen Sie das Programm aus:

```
wertDouble3 = ((double)wertInt1) / wertInt2;
```

Welches Ergebnis erhalten Sie und warum?

Prüfen Sie folgende zunächst verblüffende Aussage: Enthält ein Ausdruck `byte`- und/oder `short`-Typen, dann werden die `byte`- und `short`-Typen vor der Berechnung in `int`-Typen gewandelt. Die Berechnung findet also mit `int`-Typen statt (siehe oben: Regeln). Addieren Sie zwei `short`-Werte und weisen das Ergebnis wieder einer `short`-Variablen zu. Was sagt der Compiler?

Antwort

Wenn Sie folgenden Programmcode schreiben

```
short s1, s2 = 10, s3 = 20; s1 = s2 + s3;
```

dann meldet der Compiler »*possible loss of precision*«. Erst wenn sie das Ergebnis durch `(short)` wieder in einen `short`-Typ wandeln, wird der Ausdruck durch den Compiler übersetzt:

```
s1 = (short)(s2 + s3);
```

Explizite Typeinengung *(casting)*

Eine Typeinengung erfolgt, wenn der gewünschte Typ, eingeschlossen in Klammern und gefolgt von dem Variablen- bzw. Konstantennamen, im Ausdruck hingeschrieben wird:

```
( Zieltyp )  Wert
```

Beispiel

```
int a ; double b = 10.52;
a = (int) b;
a = (int) (b / 3.3 + 5.73);
```

Durch diese Typeinengung wird der `double`-Wert `10.52` in den `int`-Wert `10` gewandelt. Der Bruchanteil wird abgeschnitten. In der zweiten Anweisung erfolgt die Typanpassung erst nach der Ausführung der Berechnung `b / 3.3 + 5.73`.

Bei Typeinengungen tritt in der Regel ein Informationsverlust ein. Um Fehler im Programm möglichst frühzeitig zu entdecken, soll durch die explizite Typkonversion der Programmierer daran erinnert werden, mit welchen Typen er arbeitet. Der Java-Compiler kann Typverletzungen erkennen und als Fehler melden. Eine Typeinengung zwischen Werten vom Typ `boolean` und numerischen Typen ist *nicht* möglich.

Beispiel

```
// Es sollen Zinsen für ein Kapital k mit dem Prozentsatz
// p für t Tage berechnet werden:
double z, k, p; int t;
z = k * p * (double) t / 100.0 / 360.0;
```

Wichtig ist, dass die Werte `100` und `360` als Gleitpunkt-Literale hingeschrieben werden. Da in diesem Fall für die Variable `t` eine Typausweitung erfolgt, ist eine explizite Typumwandlung *nicht* erforderlich, aber erlaubt.

Runden statt Abschneiden

Soll bei einer Typeinengung der Bruchanteil einer Gleitpunkt-Zahl *nicht* abgeschnitten, sondern gerundet werden, dann kann dazu die Operation `round` aus der Java-Mathematik-Klasse verwendet werden.

Wenn der Typ `int` auf den Typ `byte` eingeengt wird, dann wird der Modulo-Wert der Variablen vom Typ `byte` zugewiesen, wenn der `int`-Wert größer als der `byte`-Wertebereich ist. Analog gilt dies auch bei der Typeinengung anderer Typen auf `byte`.

Beispiel

Typwandlung


```
//Beispiele für Typumwandlungen
public class Typwandlung
{
 public static void main (String args[])
 {
  double zahl1 = 10.5;
  double zahl2 = 10.49;
  int zahl1int = (int)zahl1;
  int zahl1intgerundet = (int)Math.round(zahl1);
  int zahl2int = (int)zahl2;
  int zahl2intgerundet = (int)Math.round(zahl2);
  System.out.println("Zahl1: " + zahl1);
  System.out.println("Zahl1 abgeschnitten: " + zahl1int);
  System.out.println("Zahl1 gerundet: " + zahl1intgerundet);
  System.out.println("Zahl2: " + zahl2);
  System.out.println("Zahl2 abgeschnitten: " + zahl2int);
  System.out.println("Zahl2 gerundet: " + zahl2intgerundet);
  byte b; //Wertebereich 0 bis 255
  int i = 260;
  b = (byte)i;
    //führt zu einer Modulo-256-Operation (Rest = 4)
  System.out.println("i: " + i + " b: " + b);
 }
}
```

Es werden folgende Ergebnisse ausgegeben:

```
Zahl1: 10.5
Zahl1 abgeschnitten: 10
Zahl1 gerundet: 11
Zahl2: 10.49
Zahl2 abgeschnitten: 10
Zahl2 gerundet: 10
i: 260 b: 4
```

3.11 Vom Problem zur Lösung: Teil 1 **

Ein Programm zu entwickeln ist *nicht* einfach. Beim Programmieren geht es immer darum, zu einem gegebenen Problem eine Lösung zu finden. Programmieren kann man daher gleichsetzen mit Problemlösen.

Ein Problem lässt sich durch folgende Aspekte charakterisieren:

- einen Anfangszustand,
- einen Zielzustand und
- einen Problemlöseraum.

Der Problemlöseraum bestimmt,

- ob es eine Lösung, mehrere Lösungen oder keine Lösung gibt, um den Anfangszustand in den Zielzustand zu überführen,
- welche Mittel für eine Lösung zur Verfügung stehen,
- welche Einschränkungen zu berücksichtigen sind.

»Ein Problem löst man, indem man einen Weg zwischen bestehendem Ausgangs- und gewünschtem Zielzustand findet« [FuZu06, S. 217].

Beim Problemlösen müssen

- Lösungswege gesucht und
- notwendige Entscheidungen getroffen werden sowie
- die Entscheidungen in der richtigen Reihenfolge erfolgen.

Die Abb. 3.11-1 veranschaulicht den Weg vom Problem zur Lösung.

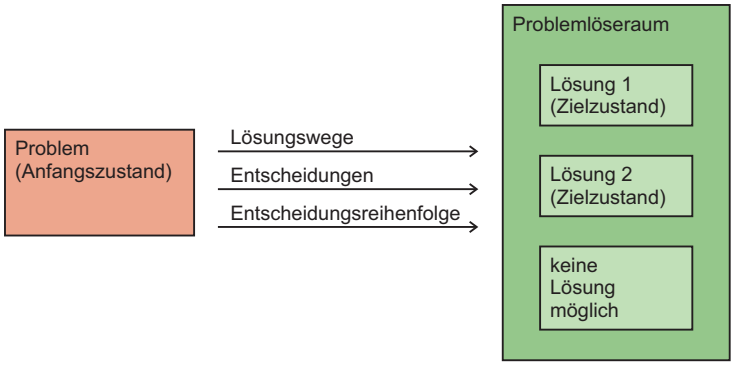


Abb. 3.11-1: Vom Problem zur Lösung.

Es gibt gut strukturierte (einfache) Probleme und schlecht strukturierte (komplexe) Probleme.

Für gut strukturierte Probleme gibt es oft Lösungsschablonen, Lösungsmuster oder Algorithmen, die den Lösungsweg vorgeben.

Beispiel 1

Sie sollen folgendes Problem lösen:

Eine Rechnung über 200 € enthält die Mehrwertsteuer von 19 %. Wie hoch ist der Betrag ohne Mehrwertsteuer?

Sie stellen folgende Gleichungen auf:

200 € entspricht 119 %
x € entspricht 100 %

Sie wandeln die Gleichungen in eine Verhältnisgleichung um:

200 : x = 119 : 100

Es ergibt sich:

200 / x = 119 / 100

Aufgelöst nach x:

x = 200 * 100 / 119 = 168,07

Gegenprobe: 168,07 * 119 / 100 = 200 €.

Sie erinnern sich an Ihre Schulzeit, und wissen, dass es sich um einen Dreisatz handelt. Nach diesem Schema können Sie nun ähnliche Aufgaben schrittweise lösen.

In der Regel lassen sich Programmierprobleme *nicht* schematisch lösen.

Hinweis

Die theoretische Informatik hat übrigens nachgewiesen, dass es *keinen* Algorithmus gibt, der für ein beliebiges Problem ein Programm erstellt.

Das folgende Beispiel zeigt, wie Sie systematisch vorgehen können, um ein Programmierproblem zu lösen. Es wird bei diesem Beispiel davon ausgegangen, dass Sie mit Ihrem bisherigen Programmierwissen dieses Problem lösen können.

Beispiel 2

Problem: Lesen Sie 2 Variablenwerte von der Konsole ein.

Ordnen Sie der 1. Variablen den Wert der 2. Variablen zu und umgekehrt.

Geben Sie die neuen Variablenwerte auf der Konsole aus.

Mögliche Lösungsschritte zeigt die Tab. 3.11-1.

Schritt	Semiformale Lösung	Formale Java-Lösung
1.	Variable 1 einlesen	`var1 = Console.readInt();`
2.	Variable 2 einlesen	`var2 = Console.readInt()`
3.	Variablenwerte vertauschen	Wie?
4.	Neuen Variablenwert 1 ausgeben	`System.out.println("Variablenwert 1 = ", var1);`
5.	Neuen Variablenwert 2 ausgeben	`System.out.println("Variablenwert 2 = ", var2);`

Tab. 3.11-1: Mögliche Lösungsschritte: Variablenwerte vertauschen.

Bei der Erarbeitung einer Lösung werden Sie folgende Probleme feststellen:

- Bei der Problemstellung wird *keine* Aussage über die Typen der Variablen gemacht. Bei der semiformalen Lösung spielt es auch keine Rolle, jedoch bei der formalen Lösung in Java. Sie stellen fest, dass die Lösung unabhängig vom Typ der Variablen ist und entscheiden sich bei der Realisierung für den Datentyp `int`.

○ Das eigentlich zu lösende Problem lautet: »Wie werden Variablenwerte vertauscht?« Dazu müssen Sie das Variablenkonzept einer Programmiersprache kennen und verstanden haben. Dann kommen Sie auf die Lösungsidee, dass Sie eine weitere Variable als Hilfsvariable benötigen, um den Wert einer Variablen zwischenzuspeichern, während Sie den Wert dieser Variablen anschließend überschreiben. Diese Lösungsidee führt zu folgender formalen Lösung:

```
hilfsvar = var1;
var1 = var2;
var2 = hilfsvar;
```

Außerdem müssen Sie alle Variablen noch deklarieren:

```
int var1, var2, hilfsvar;
```

Dieses Beispiel verdeutlicht folgendes:

- Sie müssen die Problemstellung verstanden haben bzw. die Problemstellung muss exakt genug spezifiziert sein.
- Die gewünschte Problemlösung muss exakt genug spezifiziert sein.

Diese beiden Voraussetzungen für eine Problemlösung sind hier gegeben. Sie müssen folgende Entscheidungen treffen:

- Wie viele Variablen werden benötigt?
- Wie viele und welche Anweisungen werden benötigt?
- Von welchen Typen sind die Variablen?
- Müssen die Variablen initialisiert werden und wenn ja, wie?
- Wie sieht die Reihenfolge der Anweisungen aus?
- Gibt es Lösungsalternativen?

Hinweis

Im Gegensatz zu anderen Fachgebieten, insbesondere der Mathematik, spielt bei der Programmierung immer die **Dynamik** eine wesentliche Rolle, d. h. es ist immer zu beachten, dass bei der Abarbeitung eines Programms die Reihenfolge der Anweisungen richtig gewählt ist.

Wie werden Sie nun vom Programmieranfänger zum Programmierexperten?

Zitat

»Es zeigt sich [..], dass verschiedene Kompetenzen und Fertigkeiten eine Rolle spielen: Zum einen verfügen Experten nicht nur über ein umfangreiches Faktenwissen, sondern auch über Wissen zu Zusammenhängen in einem Bereich – darauf kann beim Lösen von Problemen zurückgegriffen werden. Zum anderen können diese Menschen Besonderheiten einer (Problem-)Situation sehr schnell erkennen; das wiederum erlauben ihnen die bereichsspezifischen Schemata, als komplexe Wissensgefüge, auf die bei Bedarf zugegriffen werden kann« [FuZu06, S. 209].

3.12 Box: Kreuzworträtsel 1 **

Um programmieren zu können, müssen Sie Programmierkonzepte kennen, verstanden haben und auf eigene Probleme anwenden können.

Zusätzlich müssen Sie wichtige Begriffe der Informatik und Programmierung kennen, d. h. Sie müssen auch in der Lage sein, den Beschreibungen die richtigen Begriffe zuzuordnen. Anhand eines Kreuzworträtsels können Sie diese Fähigkeiten überprüfen.

Lösen Sie das Kreuzworträtsel (Abb. 3.12-1). Die Musterlösung dazu finden Sie im Anhang.

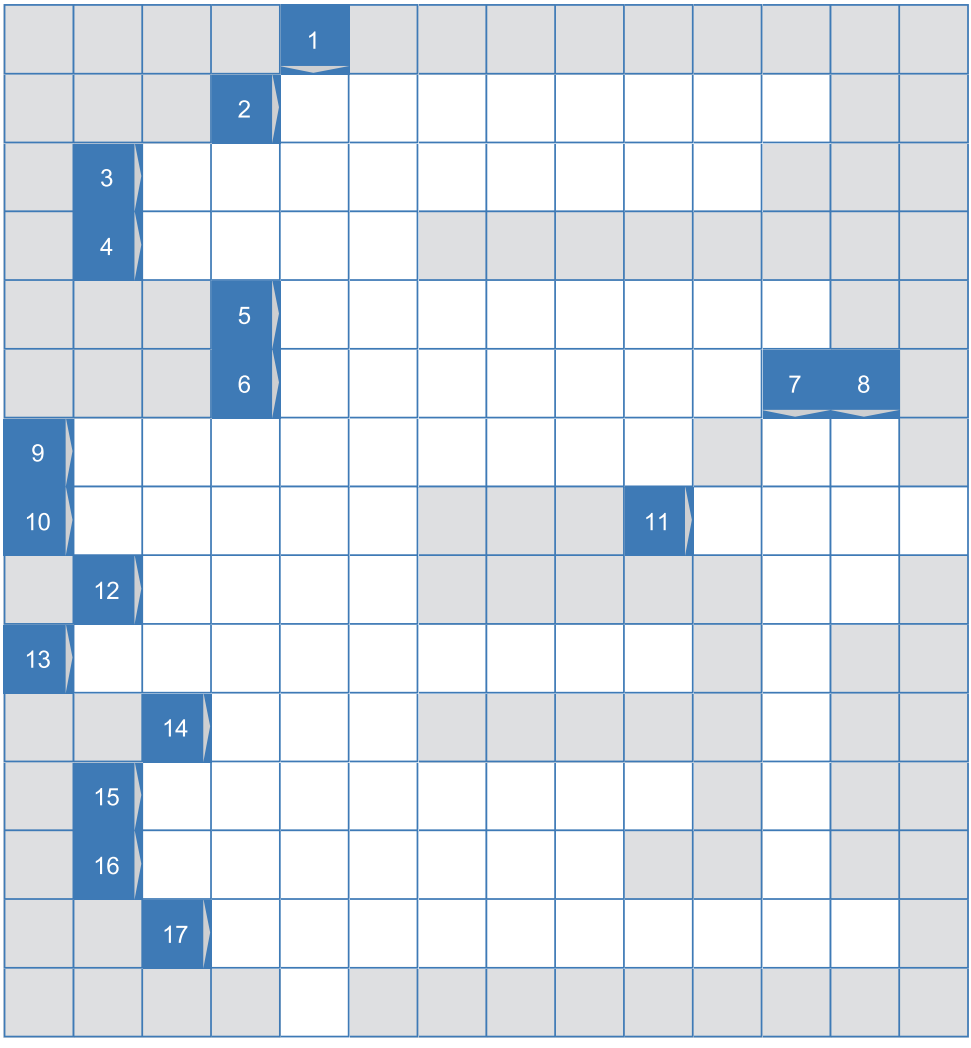


Abb. 3.12-1: Kreuzworträtsel zu Basiskonzepten der Programmierung.

Gesuchte Wörter:

1 Hardware und Software als Einheit.
2 Sorgt dafür, dass problemorientierte Programme (Quellprogramme) in Objektprogramme transformiert werden.
3 Der Teil eines Computers, in dem die Programme Schritt für Schritt verarbeitet werden.
4 Kurzform für: hypertext markup language.
5 Handlungsvorschrift für einen Computer.
6 Genormter 16-Bit-Zeichensatz (65.469 Positionen), der die Schriftzeichen aller Verkehrssprachen der Welt aufnehmen soll.
7 Gegenteil von Konstante.
8 Interpreter, der den Java-Bytecode analysiert und ausführt (Kurzform).
9 Kombination aus Prozessortyp und verwendetem Betriebssystem.
10 Wird in der Softwaretechnik nicht durch die Post ausgeliefert.
11 Programmiersprache, die ursprünglich für die Programmierung elektronischer Konsumgeräte entwickelt wurde, ihren Siegeszug aber erst im Web antrat.
12 Inhalt einer Variablen.
13 Variable, die nach der Initialisierung nicht mehr verändert werden kann.
14 Dieses Mädchen ist nicht mein ... Bei Programmiersprachen beschwert sich der Compiler.
15 Verarbeitungsvorschrift zur Ermittlung eines Wertes.
16 Darstellung des Werts einer Variablen oder Konstanten.
17 Namen für Variable, Konstante, Typen, Funktionen, Prozeduren, Klassen, Objekte usw. in Programmiersprachen, um sie eindeutig identifizieren zu können.

4 Kontrollstrukturen *

In der Programmierung werden prinzipiell zwei Arten von **Anweisungen** *(statements)* unterschieden:

- einfache Anweisungen und
- Steueranweisungen – Kontrollstrukturen genannt.

Einfache Anweisungen sind Zuweisungen mit oder ohne Ausdrücke.

Beispiel

```
radius = durchmesser / 2.0; //einfache Anweisung
```

Kontrollstrukturen

Kontrollstrukturen *(control structures)* steuern die Ausführung von Anweisungen, d. h. sie geben an, in welcher Reihenfolge, ob bzw. wie oft Anweisungen ausgeführt werden sollen.

Ziele

Kontrollstrukturen sollen

- es ermöglichen, Problemlösungen in problemangepasster, natürlicher Form zu beschreiben,
- so beschaffen sein, dass sich die Problemstruktur im Programm widerspiegelt,
- leicht lesbar und verständlich sein,
- eine leichte Zuordnung zwischen statischem Programmtext und dynamischem Programmzustand erlauben,
- mit minimalen, orthogonalen Konzepten ein breites Anwendungsspektrum abdecken,
- Korrektheitsbeweise von Algorithmen erleichtern.

Eine problemadäquate Umsetzung von Problemlösungen in Kontrollstrukturen wird durch folgende fünf semantisch unterschiedliche Kontrollstrukturen ermöglicht (Abb. 4.0-1):

- Sequenz,
- Auswahl,
- Wiederholung,
- Aufruf anderer Programme,
- Nebenläufigkeit.

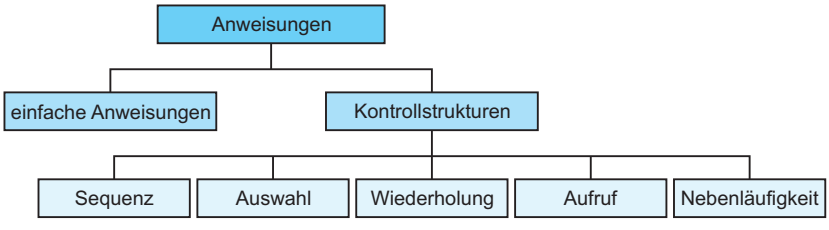


Abb. 4.0-1: Gliederung von Anweisungen.

Strukturierte Programmierung

Seit der Entwicklung der strukturierten Programmierung Anfang der 70er Jahre gilt es als »Stand der Technik«, nur diese Kon-

trollstrukturen zu verwenden (siehe »Strukturierte Programmierung«, S. 153).

Die Semantik der ersten vier Kontrollstrukturen wird in den folgenden Kapiteln skizziert:

- »Die Sequenz«, S. 106
- »Die ein- und zweiseitige Auswahl«, S. 109
- »Die Mehrfachauswahl«, S. 117
- »Bedingte Wiederholung und n + 1/2-Schleife«, S. 122
- »Die Zählschleife und die Endlosschleife«, S. 131
- »Termination von Schleifen«, S. 313
- »Der Aufruf«, S. 137

3 Notationen

Gleichzeitig werden sie in drei Notationen angegeben:

- Struktogramm-Notation,
- Aktivitätsdiagramm in der UML-Notation und
- Java-Syntax.

Struktogramme

Die **Struktogramm-Notation** beruht auf einem Vorschlag von [NaSh73], daher auch **Nassi-Shneiderman-Diagramm** genannt, und ermöglicht eine grafische Darstellung von Kontrollstrukturen. Die Notation ist in [DIN66261] genormt.

Aktivitäts-diagramm

Eine moderne grafische Darstellungsform von Programmen und Modellen ermöglicht die sogenannte **UML** *(unified modeling language)*, die verschiedene Diagrammformen unterscheidet. Das **Aktivitätsdiagramm** der UML ermöglicht es, Kontrollstrukturen in Fluss-Notation zu beschreiben. Es benutzt Rechtecke mit abgerundeten Ecken zur grafischen Darstellung von Aktionen *(actions)* sowie kleine Rauten *(diamonds)* zur Darstellung von Verzweigungen *(decision nodes)* und Zusammenführungen *(merge nodes)*. Die Symbole werden durch Pfeile miteinander verbunden, die den möglichen Kontrollfluss angeben. Alternativ gibt es noch eine kompakte Knoten-Notation.

Tipp

> Zum Zeichnen von UML-Diagrammen kann das kostenlose Programm Modelio verwendet werden.

Hinweis

> Aktuelle Informationen über die UML finden Sie auf der Website UML-Diagramme (`http://www.uml-diagrams.org/`).

PAP

Die Aktivitätsdiagramme können als Nachfolger der **Programmablaufpläne** (PAPs) angesehen werden, bei denen ebenfalls grafische Symbole verwendet werden, die durch Linien miteinander verbunden sind. PAPs – auch **Flussdiagramme** genannt – sind bereits seit 1969 in Gebrauch und genormt [DIN66001]. Da die UML-Notation heute Industrie-Standard ist und durch ent-

sprechende Werkzeuge unterstützt wird, verlieren PAPs immer mehr an Bedeutung. Sowohl Aktivitätsdiagramme in Fluss-Notation als auch PAPs sind aus Sicht der strukturierten Programmierung (siehe »Strukturierte Programmierung«, S. 153) für die Konzeption von Programmen *nicht* gut geeignet. Die Aktivitätsdiagramme in Knoten-Notation unterstützen dagegen die strukturierte Programmierung. Einen Quervergleich grafischer Notationen für Kontrollstrukturen enthält [DIN EN28631].

Java

Ein Java-Programm besteht aus einer oder mehreren **Klassen**. Jede Klasse wiederum besteht aus einer oder mehreren Operationen – in Java **Methoden** genannt. Eine Methode löst eine eigenständige Teilaufgabe innerhalb einer Klasse (siehe »Prozeduren, Funktionen und Methoden«, S. 207).

Methoden-Deklaration

Die Beschreibung einer Methode wird **Methoden-Deklaration** *(method declaration)* genannt – analog wie die Beschreibung einer Variablen als Variablen-Deklaration bezeichnet wird. Eine Methoden-Deklaration wiederum besteht aus einem **Methoden-Kopf** *(method header)* und einem **Methoden-Rumpf** *(method body)*.

Beispiel

Im einfachsten Fall besteht eine Klasse in Java aus einem Hauptprogramm, der sogenannten `main`-Methode:

```
public class Klasse
{
 //main-Methode
 public static void main (String args[]) //Methoden-Kopf
 {
   //Hier steht der Methodenrumpf
   int zahl; //Lokale Variablendeklaration
   zahl = zahl + 1; //Einfache Anweisung
 }
}
```

In Java besteht der Rumpf einer Methode aus lokalen Variablen- und Konstantendeklarationen *(local declaration statements)* sowie aus Anweisungen *(statements)* (Abb. 4.0-2).

Syntax

Einen Überblick über den vollständigen Aufbau eines Java-Methoden-Rumpfes gibt die Abb. 4.0-2. Auf die einzelnen Syntaxkonstrukte wird in den jeweiligen Kapiteln eingegangen.

Einen zusammenfassenden Überblick über die Kontrollstrukturen gibt die Abb. 4.0-3.

Kontrollstrukturen können beliebig ineinander geschachtelt werden. Dadurch ist es möglich, komplexe Problemstellungen zu programmieren:

Schachtelung

- »Geschachtelte Kontrollstrukturen«, S. 140

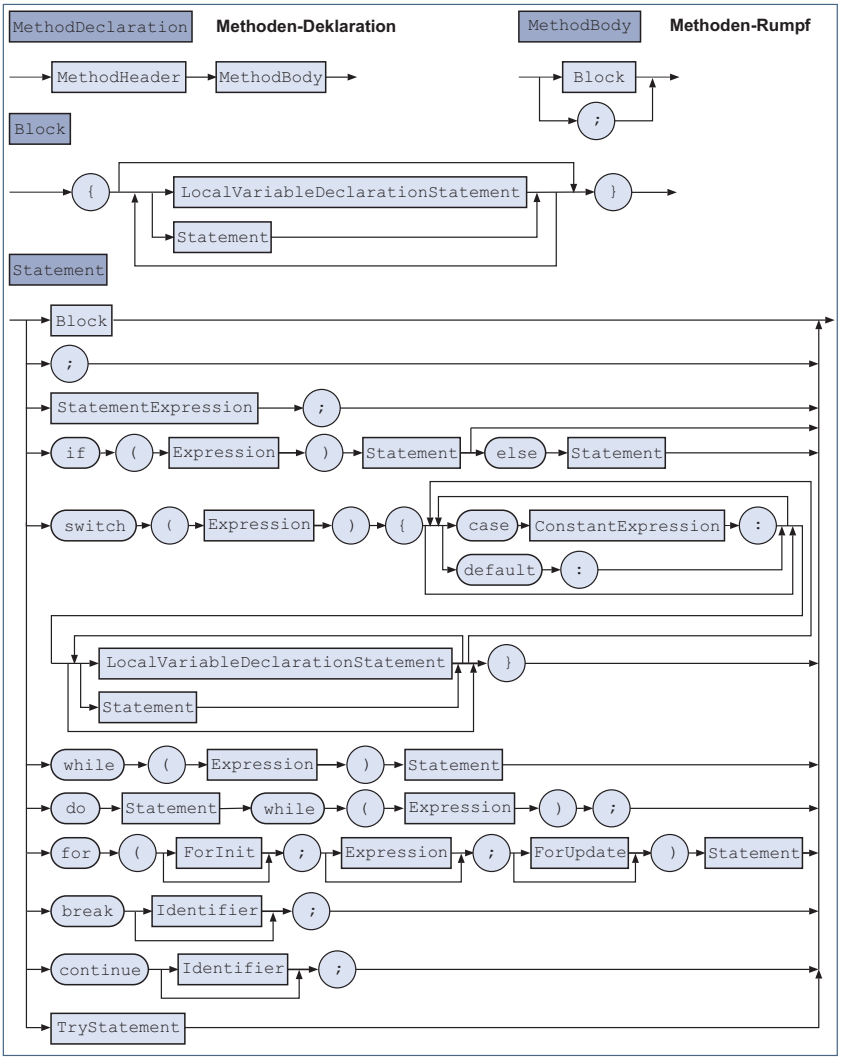


Abb. 4.0-2: So sieht die Syntax für eine Methodendeklaration in Java aus.

Fallstudie

Mithilfe geschachtelter Kontrollstrukturen können auch ein Zeitvergleich und eine Funktionsauswahl für die Fallstudie OptiTravel programmiert werden:

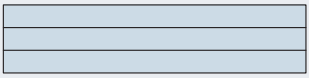
- »OptiTravel: Zeitvergleich«, S. 145
- »OptiTravel: Funktionsauswahl«, S. 147

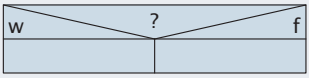
Richtige Anordnung

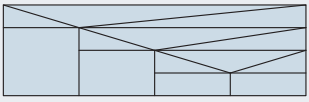
In Abhängigkeit von der Problemstellung muss eine effiziente Anordnung von Auswahlanweisungen ermittelt werden:

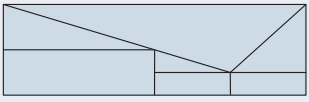
- »Anordnung von Auswahlanweisungen«, S. 149

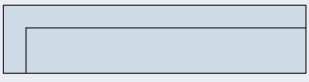
Sprachkonzept	Struktogramm	Java

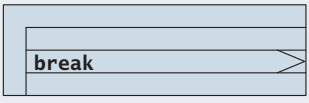
Sequenz		`Anweisung1;`
		`Anweisung2;`
		`Anweisung3;`

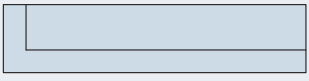
Ein- und zweiseitige Auswahl		`if(expression)`
		`  {...}`
		`[else {...}]`

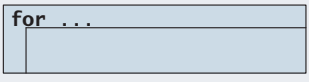
Auswahlketten		`if(expression)`
		`  {...}`
		`else if(expression)`
		`  {...}`
		`//kein eigenes Konstrukt`

Mehrfachauswahl		`switch(expression)`
		`{  case expression: ...; break;`
		`  case...:...; break;`
		`  default:...; break;`
		`}`

while-Schleife		`while(expression)`
		`{...`
		`}`

n+1/2-Schleife		`while(true)`
		`{...`
		`  if(expression) break;`
		`}`
		`//kein eigenes Konstrukt`

Schleife mit Bedingung am Ende		`do`
		`{...`
		`}`
		`while(expression);`

Zählschleife		`for(i=1; i<=10; i=i+1)`
		`{...`
		`}`

Aufruf		`Methodenname(Parameter1,` `  Parameter2, ...);`

Legende: [ ] = optional

Abb. 4.0-3: Kontrollstrukturen im Überblick.

Richtige Wahl

Wichtig für die richtige Wahl einer Kontrollstruktur ist besonders die grundlegende Unterscheidung zwischen einer Auswahl und einer Wiederholung:

»Auswahl von Kontrollstrukturen«, S. 152

Strukturiert

In »alten« Programmiersprachen und in maschinennahen Sprachen gibt es nur die Möglichkeit, mithilfe der Sprachkonstruktion `goto` (gehe nach) an eine beliebige Stelle im Programm zu springen. Eine Unterscheidung zwischen einer Auswahl und einer Wiederholung gibt es *nicht*.

Da diese Sprachkonstruktion zu vielen Programmierfehlern geführt hat, wurde die strukturierte Programmierung eingeführt:

■ »Strukturierte Programmierung«, S. 153

Ausnahmen

Ein im Einsatz befindliches Programm darf durch Eingabefehler des Benutzers oder andere Fehler, z. B. Division durch Null, *nicht* »abstürzen«, d. h. darf das Computersystem weder zum Stillstand bringen noch seine Arbeit willkürlich beenden. Egal was passiert, der Programmierer muss dafür sorgen, dass ein Programm immer definiert zum Abschluss kommt. In modernen Programmiersprachen steht dem Programmierer dafür das Konzept der Ausnahmebehandlung zur Verfügung:

■ »Behandlung von Ausnahmen«, S. 157

Zusicherungen

Um Fehler während der Programmentwicklung zu finden, gibt es die Möglichkeit, in das Programm sogenannte Zusicherungen »einzubauen«. Eine Zusicherung überprüft, ob die in der Zusicherung angegebenen Werte von Variablen zutreffen. Wenn nein, dann wird ein Fehler ausgelöst und das Programm abgebrochen:

■ »Zusicherungen in Java«, S. 162

4.1 Die Sequenz *

Durch Semikolon getrennte Anweisungen werden als Sequenz bezeichnet und von links nach rechts und von oben nach unten ausgeführt. Mehrere Anweisungen können zu Blöcken zusammengefasst werden (eingeschlossen in `{...}`), die geschachtelt werden können. Innerhalb eines Blocks können lokale Variable deklariert werden, die nur in diesem Block und allen eingeschachtelten Blöcken existieren.

Erfordert eine Problemlösung, dass mehrere Anweisungen hintereinander auszuführen sind, dann formuliert man eine **Sequenz** bzw. eine Aneinanderreihung von Anweisungen. Einen Überblick über die Darstellung der Sequenz in den verschiedenen Notationen gibt die Abb. 4.1-1.

Java-Syntax

Alle Anweisungen, die sequenziell, d. h. hintereinander ausgeführt werden sollen, werden durch ein **Semikolon** (;) voneinander getrennt (Abb. 4.1-2). Dies ist in fast allen Programmiersprachen – und auch in Java – so. Bei der Sequenz erfolgt die Abarbeitung, d. h. die Ausführung der Anweisungen, immer *von oben nach unten* und *von links nach rechts* (falls in Java mehr als eine Anweisung in einer Zeile steht).

Sequenz	allgemein	Erläuterung
Struktogramm	Anweisung1 Anweisung2 Anweisung3	Ein beliebig groß gewähltes Viereck wird nach jeder Anweisung mit einer horizontalen Linie abgeschlossen.

Java	`Anweisung1;`
	`Anweisung2;`
	`Anweisung3;`
		Die einzelnen Anweisungen werden jeweils durch ein Semikolon (;) voneinander getrennt.

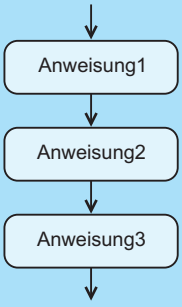
UML		Einfache Anweisungen werden durch Rechtecke mit abgerundeten Ecken (Aktionen) dargestellt, die wiederum durch Pfeile verbunden werden, die den Kontrollfluss angeben.

Abb. 4.1-1: Notationen für die Darstellung einer Sequenz.

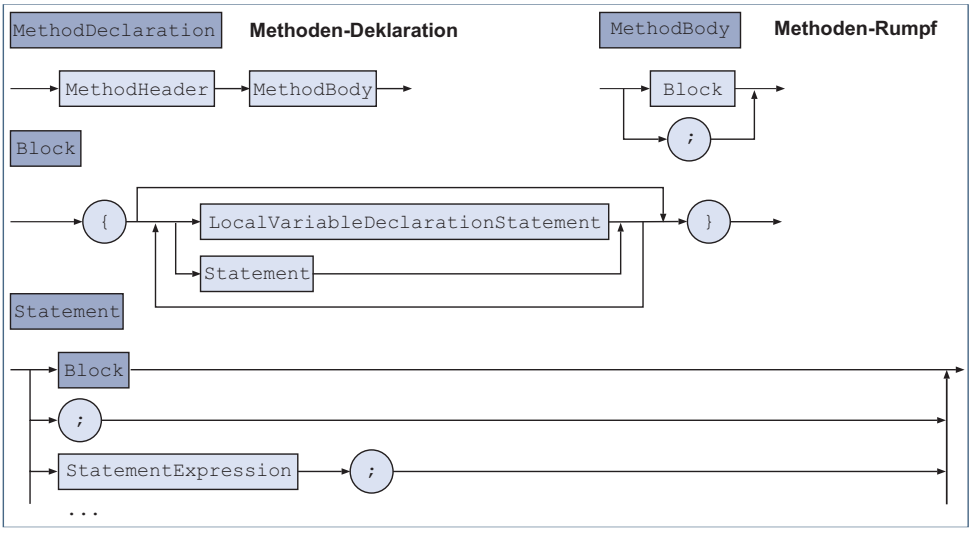


Abb. 4.1-2: Java-Syntax für sequenziell auszuführende Anweisungen.

Beispiel

DemoSequenz

```
public class DemoSequenz
{
 public static void main (String args[])
 {
    int a,b,c,d;
    a = 10; b = 20; //Zuerst a = 10, dann b = 20
    c = a + b; //dann diese Anweisung
    d = c * a; //dann diese Anweisung
 }
}
```

Blöcke

Bestehen Methoden-Rümpfe aus vielen Anweisungen, dann ist es sinnvoll, eine Strukturierung durch **Blöcke** *(blocks)* vorzunehmen. In Java bestehen Blöcke aus einem geschweiften Klammerpaar `{...}` (Abb. 4.1-2). Neben Anweisungen können in einem solchen Block auch **lokale Variable** deklariert werden, die dann nur bis zum Ende des Blocks sichtbar und existent bzw. gültig sind. **Lokale Variable** sind immer dann sinnvoll, wenn z. B. Zwischenergebnisse innerhalb eines Blocks zwischengespeichert werden müssen, die aber außerhalb des Blocks *nicht* mehr benötigt werden.

Beispiel

DemoBlock

```
//Beispiel für Blöcke
public class DemoBlock
{
 public static void main (String args[])
 {
   //lokale Variable in einem Methoden-Rumpf
   int merkeZahl = 0;
   merkeZahl = merkeZahl + 10;
   System.out.println("Merkezahl: " + merkeZahl);
   {  //Block1
     int merkeZahl2 = 20; //lokale Variable in Block1
     merkeZahl = merkeZahl + 20;
     System.out.println("Merkezahl in Block1: " + merkeZahl);
      //merkeZahl ist in Block1 sichtbar
     merkeZahl2 = merkeZahl;
     System.out.println
         ("Merkezahl2 in Block1: " + merkeZahl2);
   }  //Ende Block1
   //ab hier ist MerkeZahl2 nicht mehr sichtbar
   merkeZahl = merkeZahl + 50;
   System.out.println("Merkezahl: " + merkeZahl);
 }// Ende des Methoden-Rumpfs
}
```

Das Ergebnis sieht folgendermaßen aus:

```
Merkezahl: 10
Merkezahl in Block1: 30
Merkezahl2 in Block1: 30
Merkezahl: 80
```

Schachtelung

Blöcke können beliebig ineinander **geschachtelt** werden.

Blöcke ...
- werden in Methoden-Rümpfen eingesetzt.
- fassen Anweisungen zu einer Einheit zusammen.
- werden durch `{..}` geklammert.
- können ineinander geschachtelt werden.
- können lokale Variable enthalten.
- bilden für lokale Variable einen eigenen Lebensdauer- bzw. Gültigkeitsbereich.

4.2 Die ein- und zweiseitige Auswahl *

Eine Auswahl-Anweisung ermöglicht die Auswahl einer bestimmten Anweisung oder Anweisungsfolge (Block) während der Ausführung eines Programms. Bei der ein- und zweiseitigen Auswahl wird eine Anweisung oder Anweisungsfolge in Abhängigkeit von einer Bedingung ausgewählt. Bei einer zweifachen Auswahl sollte im `else`-Teil als Kommentar angegeben werden, welche Bedingungen gelten.

Sollen Anweisungen nur in Abhängigkeit von bestimmten Bedingungen ausgeführt werden, dann verwendet man das Konzept der **Auswahl** – auch **Verzweigung** oder **Fallunterscheidung** genannt.

3 Konzepte

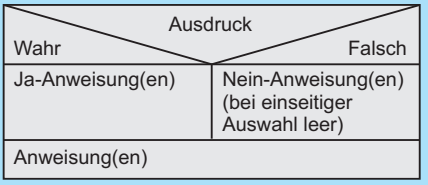
Es gibt drei verschiedene Auswahl-Konzepte, die jeweils für bestimmte Problemlösungen geeignet sind:

- einseitige Auswahl (Abb. 4.2-1),
- zweiseitige Auswahl (Abb. 4.2-1),
- Mehrfachauswahl (siehe »Die Mehrfachauswahl«, S. 117).

In der UML gibt es zwei Möglichkeiten, eine ein- und zweiseitige Auswahl darzustellen. Die linke Darstellung (siehe Abb. 4.2-1) zeigt durch die Pfeile detailliert den Kontrollfluss (Fluss-Notation). In der kompakten rechten Darstellung wird die Auswahl durch einen sogenannten **Entscheidungsknoten** *(conditional node)* repräsentiert (Knoten-Notation). Der Entscheidungsknoten ist ein Sonderfall des strukturierten Knotens der UML *(structured activity node)*. Der Entscheidungsknoten wird durch ein gestricheltes Rechteck mit abgerundeten Ecken dargestellt, das aus mehreren Bereichen besteht. Der `if`-Bereich enthält den Ausdruck, der über den weiteren Kontrollfluss entscheidet. Ist der Ausdruck wahr, dann werden die Anweisungen im `then`-Bereich ausgeführt, sonst die Anweisungen im `else`-Bereich. Der `else`-Bereich kann auch fehlen. In diesem Fall liegt eine einfache Auswahl vor.

Die Java-Syntax sieht folgendermaßen aus:

Java-Syntax

Auswahl (ein- und zweiseitig)	allgemein	Erläuterung
Struktogramm		Ist der Ausdruck wahr, dann werden die Ja-Anweisungen, sonst die Nein-Anweisungen ausgeführt.

Java	`if (Ausdruck)`
	`  Ja-Anweisung;`
	`else`
	`  Nein-Anweisung;`
	`Anweisung(en);`
		Semantik analog zum Struktogramm. Bei der einseitigen Auswahl fehlt »`else` Anweisung«. Das Ergebnis des Ausdrucks muss vom Typ `boolean` sein. Anstelle von `Anweisung;` kann auch ein Block stehen: `{Anweisungen;}`

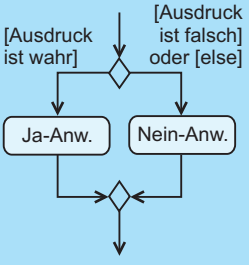
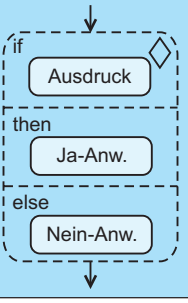
UML	Fluss-Notation:  Knoten-Notation: 	Semantik analog zu Java. Bei der einseitigen Auswahl entfällt das abgerundete Rechteck mit Nein-Anweisungen. In eckigen Klammern steht an den Pfeilen ein Wächter *(guard)*. Der Ausdruck muss wahr sein, damit der Kontrollfluss entlang des jeweiligen Pfeiles verlaufen kann.

Abb. 4.2-1: Darstellungsformen für die ein- und zweiseitige Auswahl.

if-Statement ::=
`if` (*Expression*) *Statement* [`else` *Statement*]

Zweiseitige Auswahl

Durch die Bedingung – genauer gesagt durch das Ergebnis der Auswertung eines Ausdrucks *(expression)* – wird eine Auswahl der auszuführenden Anweisungen vorgenommen. Ist die Bedingung erfüllt bzw. wahr (`true`), dann werden die Ja-Anweisungen ausgeführt, sonst die Nein-Anweisungen (`false`).

Beispiel

DemoAuswahl

```
// Programm, das feststellt,
// ob eine Ziffer eingegeben wurde

import inout.Console;

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

```
  char zchn;
  String text;
  System.out.println("Geben Sie bitte ein Zeichen ein");
  zchn = Console.readChar();

  if (zchn >= '0' && zchn <= '9')
   text = "eine Ziffer";
  else
   text = "keine Ziffer";

  System.out.println
    ("Das Zeichen " + zchn + " ist " + text);
 }
}
```

Die zweimalige Ausführung des Programms ergibt folgende Ausgaben:

```
Geben Sie bitte ein Zeichen ein
2
Das Zeichen 2 ist eine Ziffer
Geben Sie bitte ein Zeichen ein
a
Das Zeichen a ist keine Ziffer
```

Das Struktogramm und die UML-Notationen zeigt die Abb. 4.2-2.

Hinweis

Obwohl Struktogramme ursprünglich nur zur Darstellung von Anweisungen entwickelt wurden, sollte man sich angewöhnen, die Variablen- und Konstanten-Deklarationen ebenfalls anzugeben. Dies kann dadurch geschehen, dass vor den Anweisungsteil des Struktogramms immer ein Viereck gesetzt wird (durch eine doppelte Linie getrennt), das die Variablen- und Konstanten-Deklarationen enthält.

Konvention

Bei Java beginnen die Ja-Anweisungen hinter dem geklammerten Ausdruck und die Nein-Anweisungen hinter dem Schlüsselwort `else`. Die Ja- und die Nein-Anweisungen sollten textuell eingerückt unter dem Ausdruck bzw. dem `else`-Schlüsselwort in jeweils einer neuen Zeile beginnen.

Einseitige Auswahl

Bei der einseitigen Auswahl handelt es sich um einen Sonderfall der zweiseitigen Auswahl. Im `else`-Zweig steht *keine* Anweisung. Bei Programmiersprachen fehlt bei der einseitigen Auswahl der `else`-Zweig, ebenso in der Knoten-Notation der UML.

Beispiel

Versandkosten

```
// Programm, das die Versandkosten ermittelt

import inout.Console;
public class Versandkosten
{
 public static void main (String args[])
```

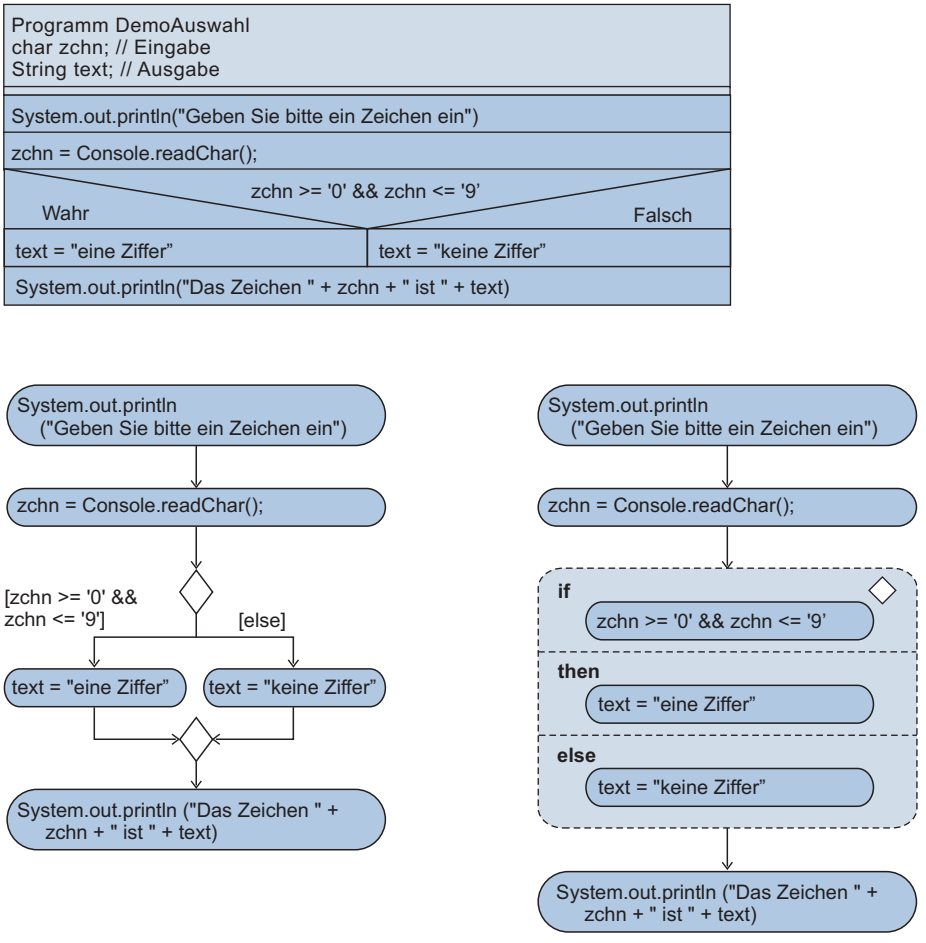


Abb. 4.2-2: Struktogramm und UML-Aktivitätsdiagramm in Fluss- und Knotennotation des Programms `DemoAuswahl`.

```
 {
  double warenwert, versandkosten = 0.0, gesamtkosten;
  System.out.println("Geben Sie bitte den Warenwert ein");
  warenwert = Console.readDoubleComma();
  if (warenwert < 20)
     versandkosten = 3;
  gesamtkosten = warenwert + versandkosten;

  System.out.println("Warenwert " + warenwert);
  System.out.println("Versandkosten " + versandkosten);
  System.out.println("Gesamtkosten " + gesamtkosten);
 }
}
```

Ein Programmlauf liefert folgendes Ergebnis:

```
Geben Sie bitte den Warenwert ein
18,22
Warenwert 18.22
Versandkosten 3.0
Gesamtkosten 21.22
```

Zeichnen Sie das zugehörige Struktogramm sowie die zwei UML-Notationen.

Gehören zum Ja-Teil und/oder zum Nein-Teil mehrere Anweisungen, dann müssen sie in Java als **Block** (eingeschlossen in geschweifte Klammern) geschrieben werden.

Innerhalb einer Auswahl kann wieder eine Auswahl stehen usw. Es liegen dann geschachtelte Auswahlanweisungen vor.

Schachtelung

Beispiel

Die Mitarbeiter einer Firma erhalten am Jahresende eine Prämie. Ist die Anzahl der Dienstjahre kleiner oder gleich zwei Jahre, dann bekommen sie keine Prämie. Liegen die Dienstjahre unter fünf Jahren oder betragen sie genau fünf Jahre, dann beträgt die Prämie € 200.-. Ist der Mitarbeiter länger als fünf Jahre bei der Firma, dann bekommt er € 100.- und für jedes Dienstjahr € 20.- zusätzlich. Ist der Mitarbeiter länger als fünf Jahre bei der Firma und außerdem älter als 50 Jahre, dann erhält er noch eine Zulage von € 100.-.

Praemie

```
// Prämienberechnung in Abhängigkeit von den
// Dienstjahren und dem Alter

import inout.Console;
public class Praemie
{
 public static void main (String args[])
 {
  final int GRUNDPRAEMIE1 = 200, GRUNDPRAEMIE2 = 100,
  ZULAGE1 = 20, ZULAGE2 = 100; //Konstanten
  int dienstjahre, alter; //Eingabe
  int praemie = 0; //Ausgabe
  System.out.println("Dienstjahre?");
  dienstjahre = Console.readInt();
  System.out.println("Alter?");
  alter = Console.readInt();
  if (dienstjahre > 5)
    { //Block
        praemie = GRUNDPRAEMIE2 + ZULAGE1 * dienstjahre;
        if (alter > 50) //eingeschachtelt
            praemie = praemie + ZULAGE2;
    }//Ende Block
  else //Dienstjahre <= 5
    if (dienstjahre > 2) //eingeschachtelt
        praemie = GRUNDPRAEMIE1;
  System.out.println("Dienstjahre:\t" + dienstjahre);
  System.out.println("Alter:\t\t" + alter);
  System.out.println("Prämie:\t\t" + praemie);
```

```
 }
}
```

Ein Programmlauf führt zu folgenden Ergebnissen:

```
Dienstjahre?
25
Alter?
55
Dienstjahre:     25
Alter:       55
Prämie:         700
```

Das zugehörige Struktogramm zeigt die Abb. 4.2-3, das UML in der Fluss-Notation die Abb. 4.2-4 und in der Knoten-Notation die Abb. 4.2-5.

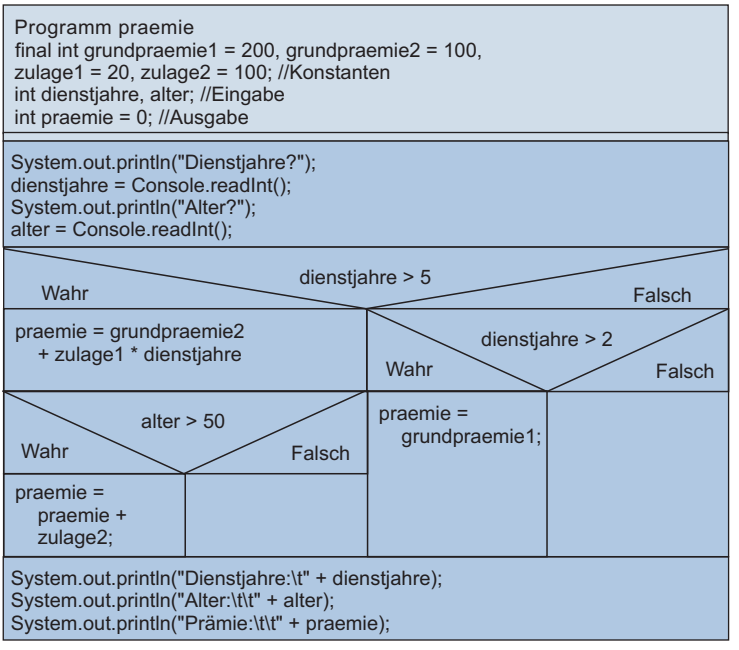


Abb. 4.2-3: Struktogramm des Programms `Praemie`.

Hinweis

Da mehrere hintereinander angeordnete Anweisungen von der Programmkonzeption her unkritisch sind, können Sie im Struktogramm in *ein* Viereck gezeichnet werden (Abb. 4.2-3).

Tipp

else-Teil mit Kommentar

Bei einer zweifachen Auswahl sollten Sie im `else`-Teil als Kommentar immer angeben, welche Bedingungen im `else`-Teil gelten. Die Verneinung der Bedingung, die im `then`-Teil gilt, ist

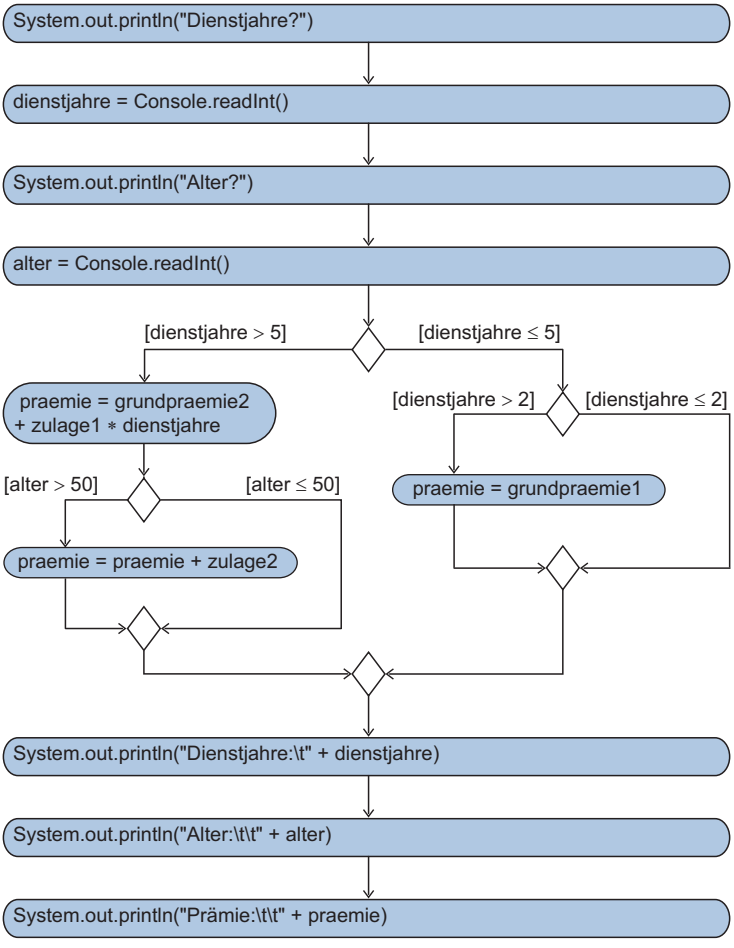


Abb. 4.2-4: UML-Aktivitätsdiagramm in Flussnotation des Programms `Praemie`.

bei kombinierten Bedingungen oft schwer zu überblicken. Die Fluss-Notation der UML eignet sich dafür besonders gut.

Beispiel

Das Struktogramm zur Lösung von zwei linearen Gleichungen mit zwei Variablen: `ax + by = c; dx + ey = f` zeigt die Abb. 4.2-6.

Wandeln Sie das Struktogramm zur Lösung von zwei linearen Gleichungen in die zwei UML-Notationen und erstellen Sie ein entsprechendes Java-Programm. Zu dem Absolutwert `abs` und dem Differenzbetrag `Eps` siehe »Rechengenauigkeit mit Gleitpunkt-Zahlen«, S. 77 (letztes Beispiel und Programmierregel).

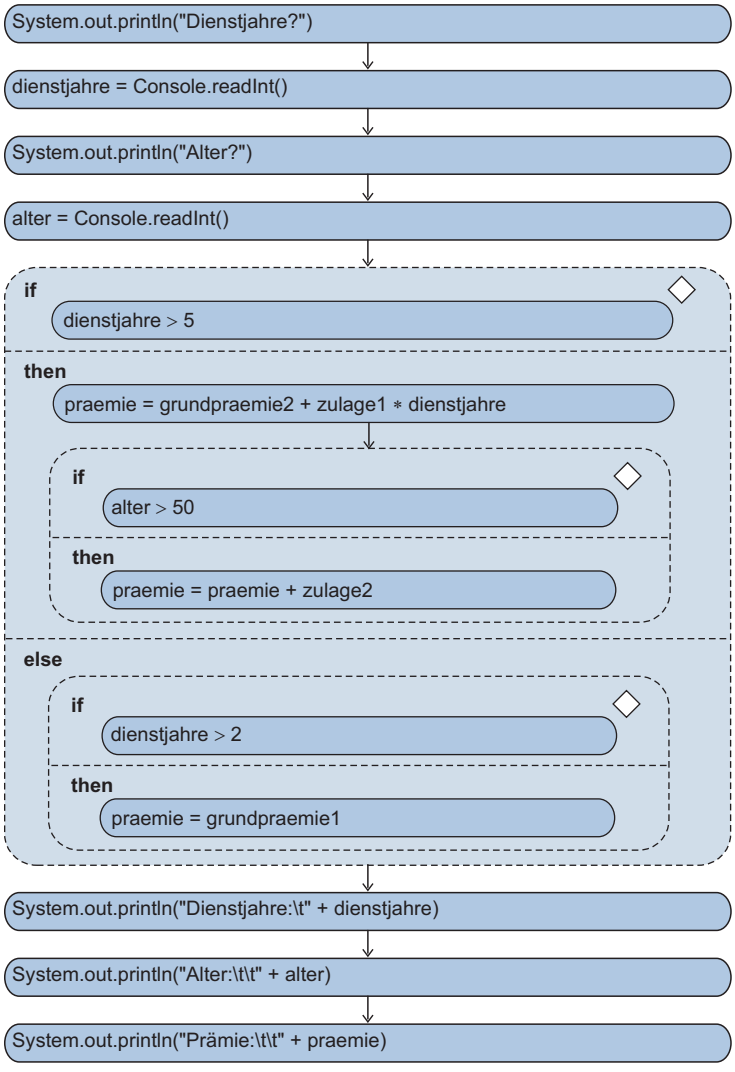


Abb. 4.2-5: UML-Aktivitätsdiagramm in Knotennotation des Programms `Prämie`.

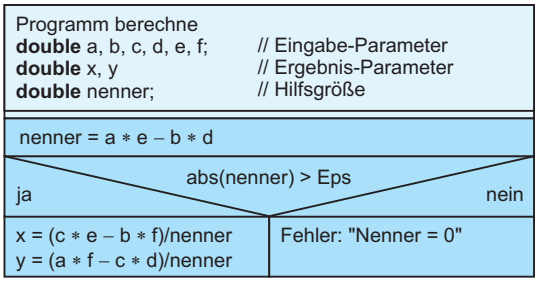


Abb. 4.2-6: Struktogramm zur Lösung von zwei linearen Gleichungen.

4.3 Die Mehrfachauswahl *

Eine Auswahl-Anweisung ermöglicht die Auswahl einer bestimmten Anweisung oder Anweisungsfolge (Block) während der Ausführung eines Programms. Bei der Mehrfach-Auswahl steuert ein Selektor die Auswahl einer markierten Anweisung oder Anweisungsfolge aus mehreren disjunkten Alternativen. Die Mehrfachauswahl ist immer mit einem Fehlerausgang (`default`) zu versehen.

Mehrfach-auswahl

Muss zwischen *mehr* als zwei Möglichkeiten gewählt werden, dann wird die Mehrfachauswahl verwendet (Abb. 4.3-1).

Die UML-Notationen kennen keine spezielle Darstellung für die Mehrfachauswahl. In der Knoten-Notation werden in den Entscheidungsknoten mehrere `if`-Bereiche eingetragen, für jeden Fall ein `if`-Bereich.

Java

In Java übergibt die `switch`-Anweisung in Abhängigkeit vom Wert des Ausdrucks hinter `switch` die Kontrolle an eine von mehreren Anweisungen. Der Typ des Ausdrucks muss `char`, `byte`, `short`, `int`, `String` (ab Java 7) oder ein Aufzählungstyp (siehe »Aufzählungen mit enum«, S. 200) sein. Der Rumpf einer `switch`-Anweisung muss ein Block sein. Jede im Block enthaltene Anweisung kann mit einer oder mehreren `case`-Markierungen und maximal einer `default`-Markierung markiert sein. Jeder konstante Ausdruck einer `case`-Markierung muss typverträglich mit dem Ausdruck hinter `switch` sein. Die Werte der konstanten `case`-Ausdrücke müssen *disjunkt* sein.

Java-Syntax

Die Java-Syntax der Mehrfachauswahl zeigt die Abb. 4.3-2.

Beispiel

Fracht-berechnung

```
// Programm, das die Frachtkosten in Abhängigkeit
// von der Tarifzone ermittelt

import inout.Console;

public class Frachtberechnung
{
 public static void main (String args[])
 {
  int tarifzone;
  double frachtkostenanteil = 0.0;
  boolean ok = true;
  System.out.println("Geben Sie bitte die Tarifzone ein");

  tarifzone = Console.readInt();
  switch (tarifzone)
  { //Block
      case 1: frachtkostenanteil = 15.0; break;
      case 2: frachtkostenanteil = 25.5; break;
      case 3: frachtkostenanteil = 35.5; break;
      case 4: frachtkostenanteil = 40.0; break;
```

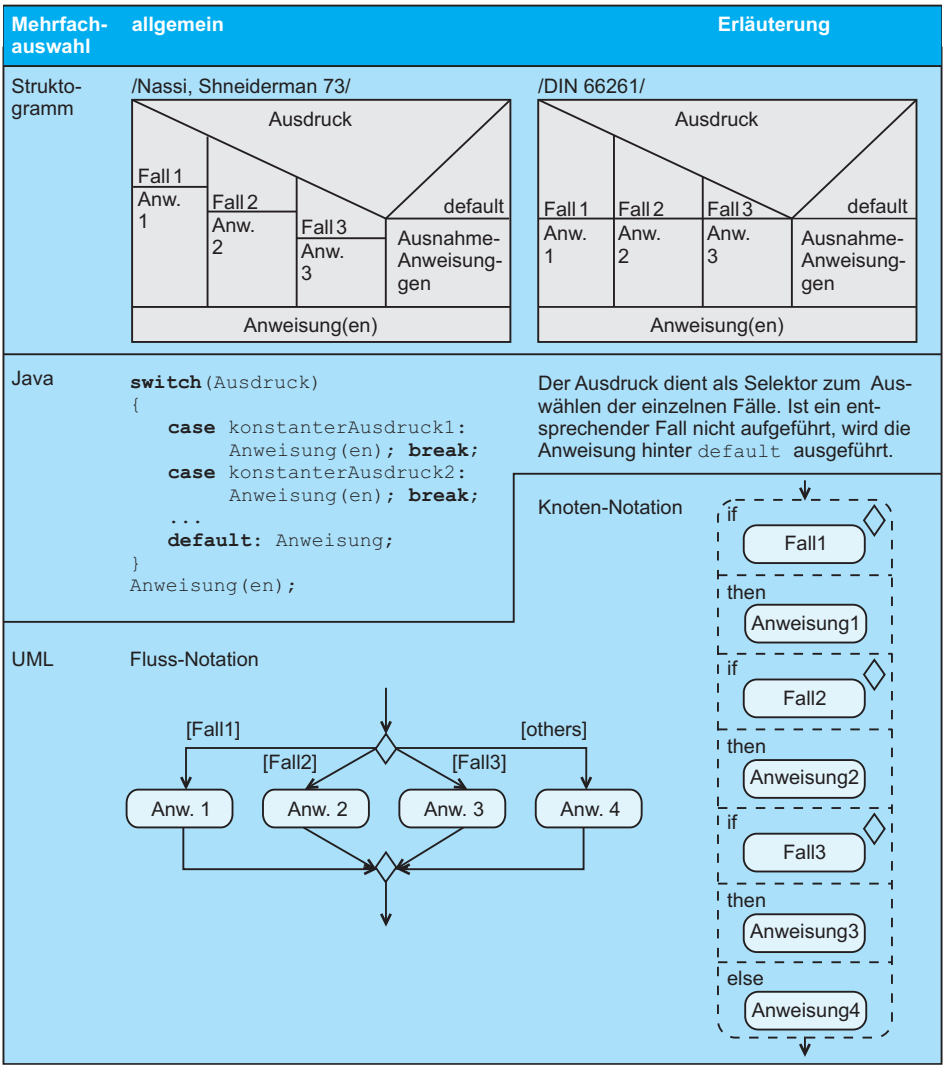


Abb. 4.3-1: Darstellungsformen für die Mehrfach-Auswahl.

```
      default: ok = false;
  }
  if (ok)
    System.out.println("Frachtkostenanteil: "
     + frachtkostenanteil);
  else
    System.out.println("Keine gültige Tarifzone: "
     + tarifzone);
 }
}
```

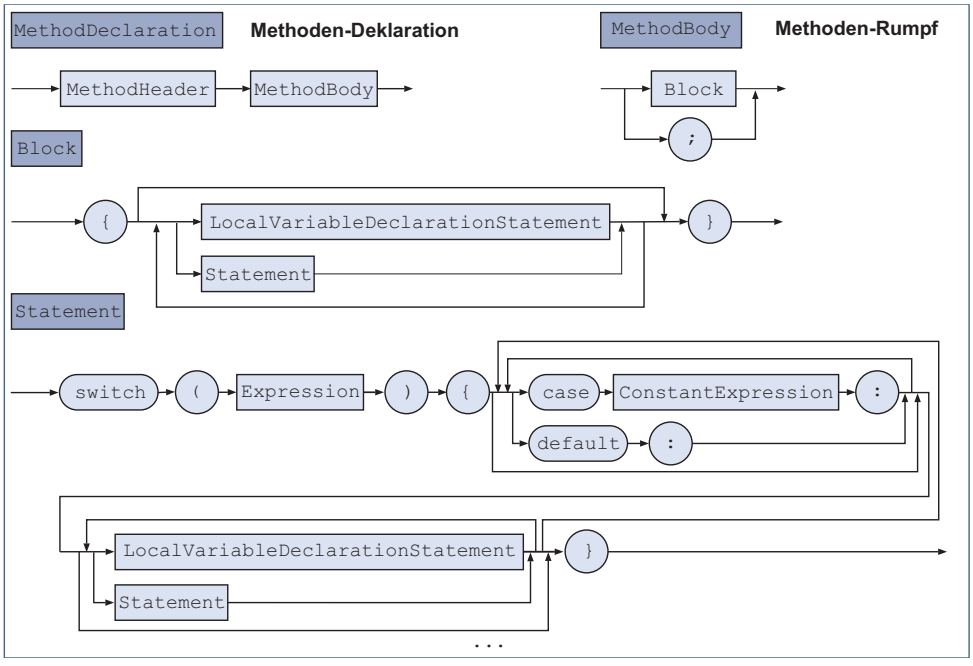


Abb. 4.3-2: Java-Syntax für eine Mehrfachauswahl-Anweisung.

Die zweimalige Ausführung führt zu folgenden Ergebnissen:

```
Geben Sie bitte die Tarifzone ein
3
Frachtkostenanteil: 35.5
Geben Sie bitte die Tarifzone ein
6
Keine gültige Tarifzone: 6
```

Eine `switch`-Anweisung wird folgendermaßen abgearbeitet:

1. Der Ausdruck hinter `switch` wird ausgewertet.
2. Ist eine der `case`-Konstanten gleich dem Wert des Ausdrucks, dann tritt der entsprechende Fall ein, und alle Anweisungen hinter dieser `case`-Markierung werden der Reihe nach ausgeführt. Die optionale `break`-Anweisung sorgt für einen Sprung an das Ende der `switch`-Anweisung. Fehlt die `break`-Anweisung, dann werden alle Anweisungen bis zum nächsten `break` oder dem Ende der `switch`-Anweisung durchgeführt.
3. Trifft kein Fall zu und gibt es eine `default`-Anweisung, dann werden alle Anweisungen hinter `default` ausgeführt.
4. Trifft kein Fall zu und gibt es keine `default`-Anweisung, dann wird nichts unternommen, und es wird hinter der `switch`-Anweisung fortgefahren.

Das Struktogramm zur Frachtberechnung zeigt die Abb. 4.3-3. Die UML-Notationen zeigen die Abb. 4.3-4 und die Abb. 4.3-5.

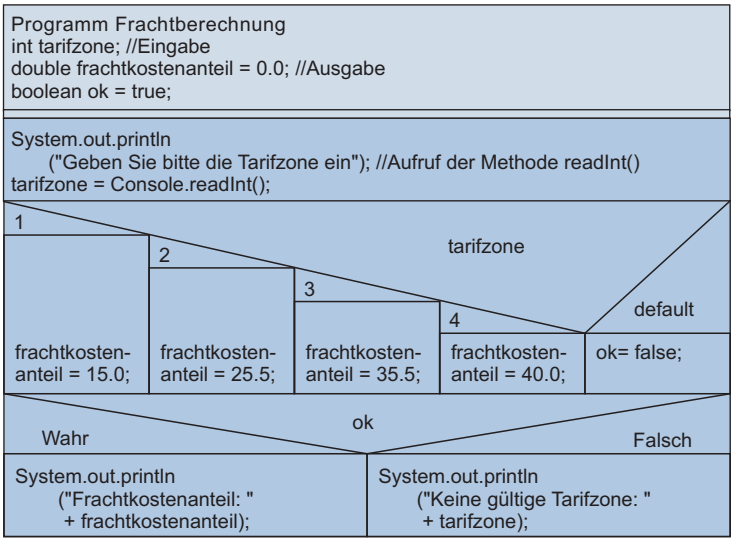


Abb. 4.3-3: Struktogramm des Programms `Frachtberechnung`.

Entfernen Sie aus dem Programm `Frachtberechnung` die `break`-Anweisungen und führen Sie das Programm dann mit mehreren Tarifzonen aus. Was stellen Sie fest?

Beispiel

KfzKenn zeichen

```
//Mehrfachauswahl mit String (ab Java 7)

import inout.Console;

public class KfzKennzeichen
{
  public static void main(String args[])
  {
     String kfzKennzeichen ="", verwaltungsbezirk ="";
     System.out.println("Kfz-Kennzeichen bitte eingeben:");
     kfzKennzeichen = Console.readString();
      switch (kfzKennzeichen)
      {
         case "A":
             verwaltungsbezirk = "Augsburg (Bay)";break;
          case "AA":
              verwaltungsbezirk = "Aalen (BaWü)";break;
          case "AB":
              verwaltungsbezirk = "Aschaffenburg (Bay)";break;
          // usw.
          default:
              verwaltungsbezirk =
               "Dieses Kennzeichen gibt es nicht";
      }
```

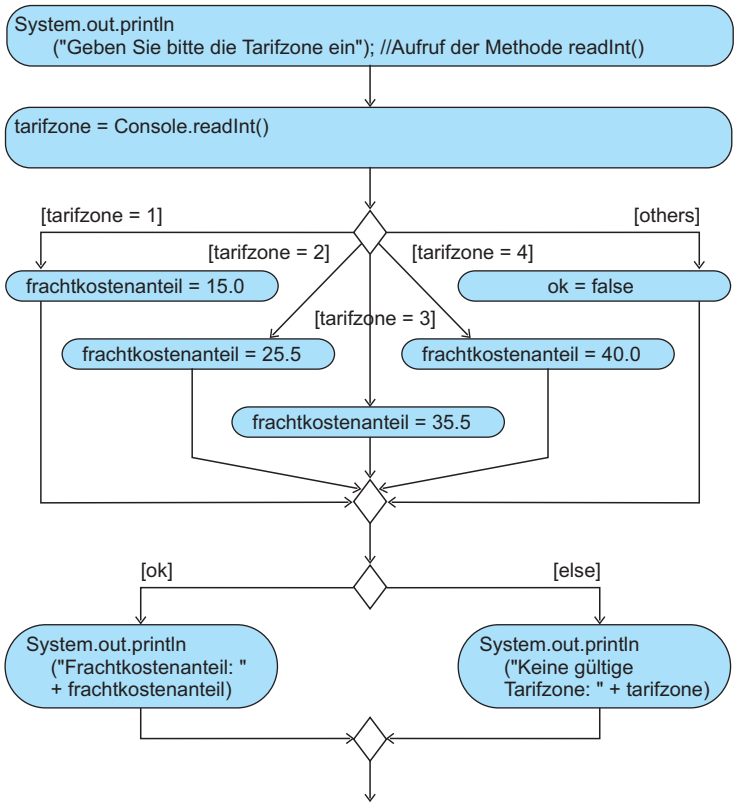


Abb. 4.3-4: UML-Aktivitätsdiagramm in Flussnotation des Programms `Frachtberechnung`.

```
        System.out.println(kfzKennzeichen + ": "
        + verwaltungsbezirk);
    }
}
```

Regeln

Aus Gründen einer sicheren Programmierung sollten Sie bei der `switch`-Anweisung immer folgende Regeln einhalten:

- Immer einen `default`-Fall vorsehen.
- Den `default`-Fall als letzten Fall hinschreiben.
- Jeden Fall mit `break` abschließen, sonst wird *nicht* an das `switch`-Ende gesprungen, sondern die nächste `case`-Anweisung ausgeführt (»Fall durch die Markierung«). Der Java-Interpreter wertet nur einmal zu Beginn der `switch`-Anweisung den Ausdruck aus und springt dann zu der entsprechenden Marke. Ab da sind für ihn alle Marken uninteressant.

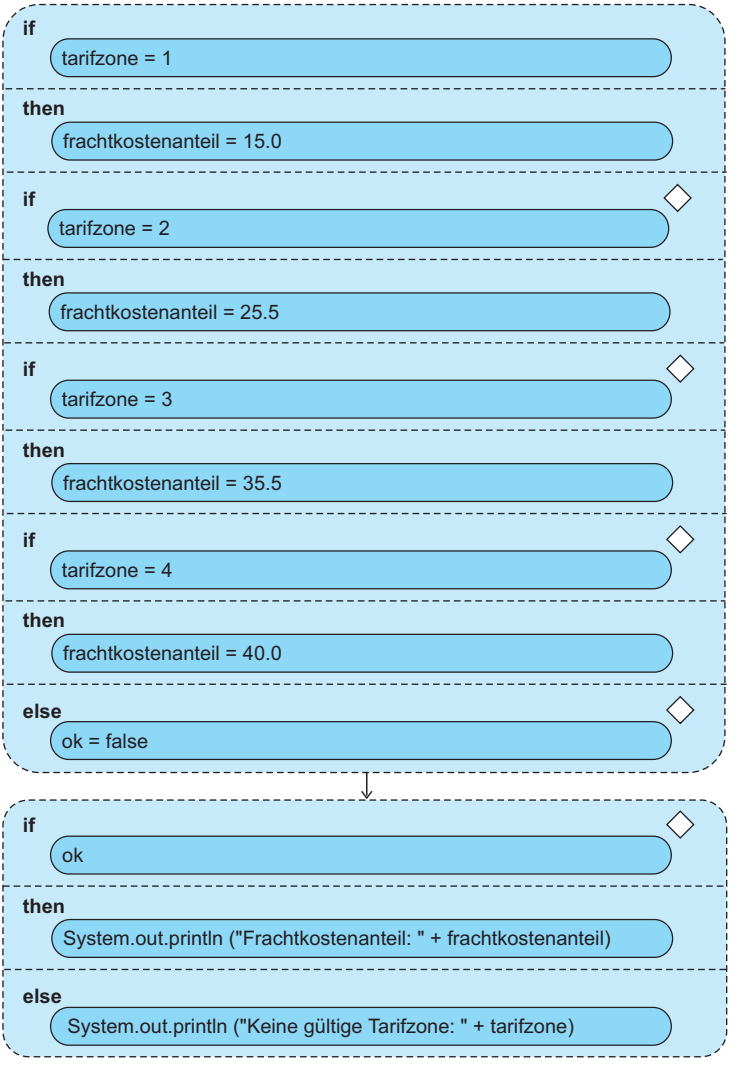


Abb. 4.3-5: UML-Aktivitätsdiagramm in Knotennotation des Programms Frachtberechnung (Ausschnitt).

4.4 Bedingte Wiederholung und n + 1/2-Schleife *

Bedingte Wiederholungen erlauben es, Anweisungen solange zu wiederholen, solange eine Bedingung erfüllt ist. Bei der abweisenden Wiederholung (`while ...`) wird vor der ersten Wiederholung die Bedingung überprüft. Bei der akzeptierenden Wiederholung (`do ...`) wird jeweils nach der Wiederholung die Bedingung überprüft. Mit `break` kann innerhalb einer Wiederholung

die weitere Ausführung der Wiederholung abgebrochen werden (z. B. in einem Fehlerfall). Mit `continue` kann die laufende Wiederholung abgebrochen und die Bedingung erneut überprüft werden (n + 1/2-Schleife).

Sollen eine oder mehrere Anweisungen (Blöcke) in Abhängigkeit von einer Bedingung wiederholt durchlaufen werden, dann ist das Konzept der **Wiederholung** bzw. **Schleife** zu verwenden. Zwei bedingte Wiederholungskonstrukte werden unterschieden (Abb. 4.4-1):

2 bedingte Wiederholungskonstrukte

- Wiederholung mit Abfrage vor jedem Wiederholungsdurchlauf (abweisende Wiederholung),
- Wiederholung mit Abfrage nach jedem Wiederholungsdurchlauf (akzeptierende Wiederholung).

Zusätzlich gibt es noch den Sonderfall, dass eine Wiederholung abgebrochen werden muss, z. B., wenn ein Fehler aufgetreten ist. Diesen Sonderfall bezeichnet man als n + 1/2-Schleife.

Wie die Abb. 4.4-1 zeigt, gibt es bei der UML sowohl eine Fluss-Notation als auch eine Knoten-Notation zur Darstellung der bedingten Wiederholung. Bei der Knoten-Notation wird eine Wiederholung durch einen sogenannten **Schleifenknoten** *(loop node)* dargestellt. Er wird durch ein Rechteck mit abgerundeten Ecken und gestrichelten Linien gezeichnet, das in zwei Bereiche aufgeteilt ist. Der `while`-Bereich enthält die Bedingung. Ist diese Bedingung wahr, dann wird der Schleifenrumpf erneut durchlaufen. Andernfalls wird die nächste Anweisung nach der Wiederholung ausgeführt. Die Java-Syntax zeigt die Abb. 4.4-2.

Abfrage vor Wiederholung

Bei der Wiederholung mit Abfrage *vor* jedem Wiederholungsdurchlauf wird solange wiederholt, wie die Bedingung erfüllt ist. Dann wird hinter der zu wiederholenden Anweisung bzw. Anweisungsfolge fortgefahren. Ist die Bedingung bereits am Anfang *nicht* erfüllt, dann wird die Wiederholungsanweisung *keinmal* ausgeführt. Die Bedingung muss daher am Anfang der Wiederholung bereits einen eindeutigen Wert besitzen.

Beispiel

Parkschein automat

```
// Programm, das die Eingabe von Münzen zählt

import inout.Console;

public class Parkscheinautomat
{
 public static void main (String args[])
 {
  int gebuehr = 350; //in Cent
  int einzahlung = 0;
```

Wiederholung	allgemein	Erläuterung
Struktogramm	Ausdruck Wiederholunganweisung(en) Anweisung(en)	Wiederholung mit Abfrage vor jedem Wiederholungsdurchlauf (Abweisende Schleife)
	Wiederholunganweisung(en) Ausdruck Anweisung(en)	Wiederholung mit Abfrage nach jedem Wiederholungsdurchlauf (Akzeptierende Schleife)

Java	`while (Ausdruck)`
	`{`
	`  Wiederholungsanweisungen;`
	`}`
	`Anweisung(en);`
		Wiederholung mit Abfrage vor jedem Wiederholungsdurchlauf

	`do`
	`{`
	`  Wiederholungsanweisungen;`
	`}`
	`while (Ausdruck);`
	`Anweisung(en);`
		Wiederholung mit Abfrage nach jedem Wiederholungsdurchlauf

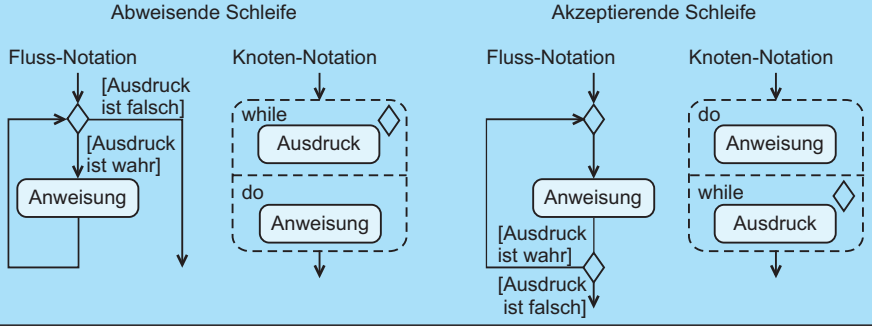
UML		

Abb. 4.4-1: Notationen für bedingte Wiederholungen.

```
int muenze;
System.out.println
 ("Die Gebühr beträgt (in Cent):" + gebuehr);
System.out.println
 ("Bitte verwenden Sie nur 10, 20, 50 Cent & 1 Euro-Münzen");
System.out.println
 ("Achtung: Überzahlungen werden nicht erstattet!");
while (einzahlung < gebuehr )
{
  System.out.println
   ("Bitte Münzwert eingeben (1 Euro als 100 Cent)");
  muenze = Console.readInt();
  switch (muenze)
  {
```

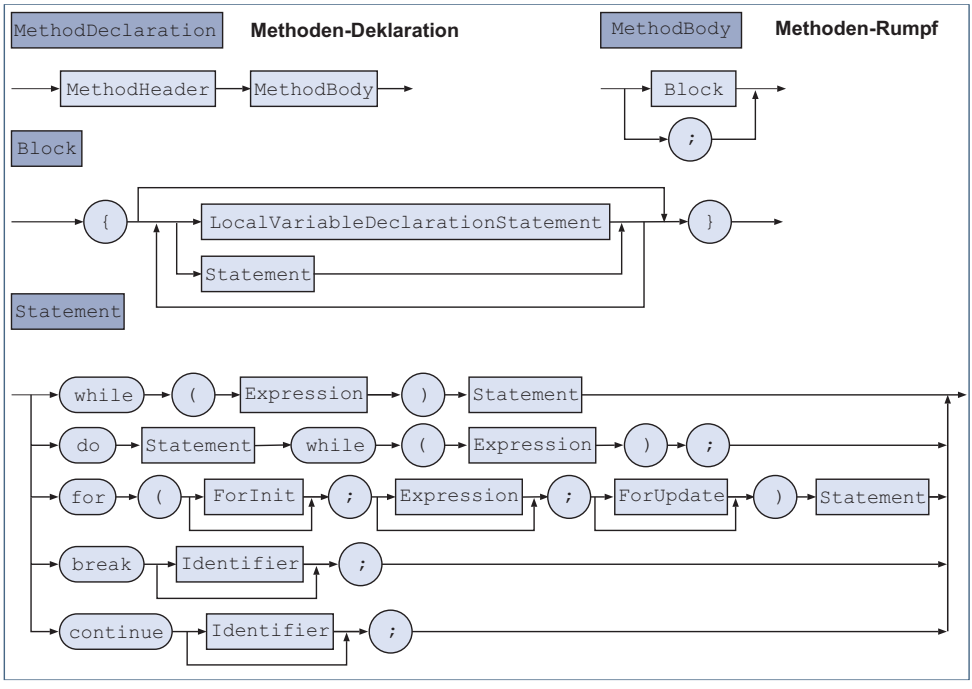


Abb. 4.4-2: Java-Syntax für Wiederholungs-Anweisungen.

```
      case 10:
      case 20:
      case 50:
      case 100:
      einzahlung = einzahlung + muenze;
      System.out.println
       ("Es fehlen noch:" + (gebuehr - einzahlung) + " Cent");
      break;
      default: System.out.println
                 ("Keine zulässige Münzart!");
     }
  } // Ende while
  System.out.println("Sie haben die Gebühr bezahlt! Danke");
  System.out.println
   ("Gebühr: " + gebuehr + " Einzahlung: " + einzahlung);
 }
}
```

Ein Programmlauf sieht folgendermaßen aus:

```
Die Gebühr beträgt (in Cent):350
Bitte verwenden Sie nur 10, 20, 50 Cent und 1 Euro-Münzen
Achtung: Überzahlungen werden nicht erstattet!
Bitte Münzwert eingeben (1 Euro als 100 Cent)
100
Es fehlen noch: 250 Cent
```

```
Bitte Münzwert eingeben (1 Euro als 100 Cent)
50
Es fehlen noch: 200 Cent
Bitte Münzwert eingeben (1 Euro als 100 Cent)
30
Keine zulässige Münzart!
Bitte Münzwert eingeben (1 Euro als 100 Cent)
20
Es fehlen noch: 180 Cent
Bitte Münzwert eingeben (1 Euro als 100 Cent)
100
Es fehlen noch: 80 Cent
Bitte Münzwert eingeben (1 Euro als 100 Cent)
50
Es fehlen noch: 30 Cent
Bitte Münzwert eingeben (1 Euro als 100 Cent)
50
Es fehlen noch: -20 Cent
Sie haben die Gebühr bezahlt! Danke
Gebühr: 350 Einzahlung: 370
```

Dieses Beispiel zeigt, dass es in Sonderfällen sinnvoll sein kann, die Mehrfachauswahl ohne `breaks` zu verwenden (Durchfallen durch die Fälle!, siehe »Die Mehrfachauswahl«, S. 117). Die Struktogramm-Notation zeigt die Abb. 4.4-3, die UML-Knoten-Notation die Abb. 4.4-4.

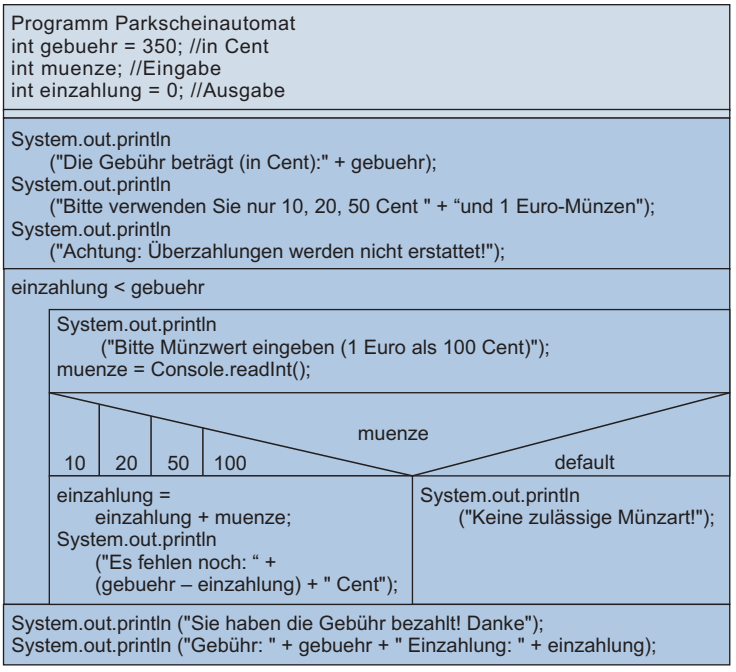


Abb. 4.4-3: Struktogramm des Programms `Parkscheinautomat`.

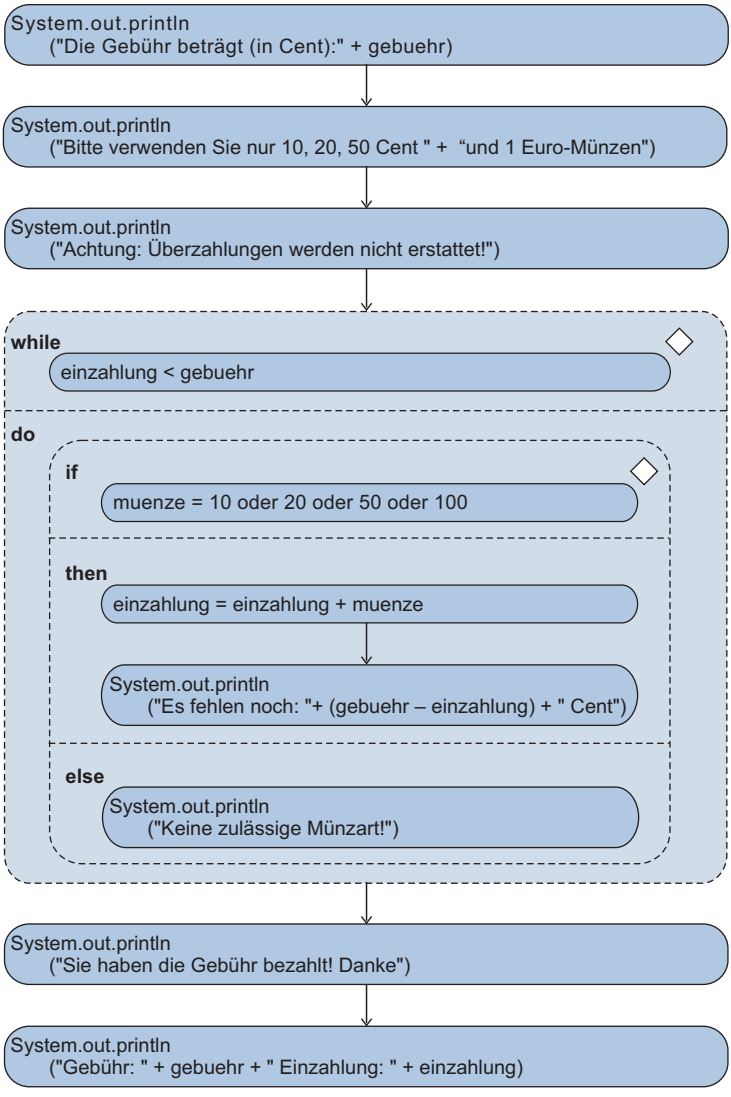


Abb. 4.4-4: UML-Aktivitätsdiagramm des Programms `Parkscheinautomat`.

Erweitern Sie das Programm `Parkscheinautomat` so, dass der Benutzer eine Parkzeit wählt und anhand der Parkzeit die Gebühr angezeigt bekommt.

Abfrage nach Wiederholung

Bei der Wiederholung mit Abfrage *nach* jedem Wiederholungsdurchlauf wird solange wiederholt, wie die Bedingung erfüllt bzw. der Ausdruck wahr ist. Die zu wiederholenden Anweisungen werden also in jedem Fall einmal ausgeführt, da die Bedin-

gung erst am Ende abgefragt wird. Eine Wiederholung mit Abfrage nach jedem Durchlauf lässt sich auf eine Wiederholung mit Abfrage vor jedem Durchlauf zurückführen, wenn die Schleife so initialisiert wird, dass die Bedingung am Anfang erfüllt ist.

Beispiel

Für die Kassen in einem Supermarkt soll ein Programm geschrieben werden, das die Anzahl der eingegebenen Beträge, den minimalen und den maximalen Betrag sowie die Gesamtsumme berechnet:

Kasse

```
// Kassensumme mit Min und Max berechnen

import inout.Console;

public class Kasse
{
 public static void main (String args[])
 {
  int wert = 0; //in Cent, Maximaler Wert: 100000
  int summe = 0, min = 100000, max = 0, anzahlPos = -1;
  final int ende = -999;
  System.out.println("Bitte Werte eingeben ");
  System.out.println("Ende bei Eingabe von -999:");

  do
  {
    summe = summe + wert;
    anzahlPos = anzahlPos + 1;
    if (wert < min && wert > 0)
      min = wert;
    if (wert > max)
      max = wert;
    wert = Console.readInt();
  } while (wert != ende);

  if (anzahlPos > 0)
  {
   System.out.println("Summe: " + summe);
   System.out.println("Anzahl Positionen: " + anzahlPos);
   System.out.println("Maximaler Betrag: " + max);
   System.out.println("Minimaler Betrag: " + min);
  }
  else
   System.out.println("Kein Wert eingegeben");
 }
}
```

Zwei Programmläufe sehen folgendermaßen aus:

```
Bitte Werte eingeben
Ende bei Eingabe von -999:
-999
Kein Wert eingegeben
Bitte Werte eingeben
Ende bei Eingabe von -999:
23
```

```
76
42
4
56
-999
Summe: 201
Anzahl Positionen: 5
Maximaler Betrag: 76
Minimaler Betrag: 4
```

Den Unterschied im Ablauf zwischen einer `while`- und einer `do`-Wiederholung veranschaulicht nochmals die Abb. 4.4-5.

`while` vs. `do`

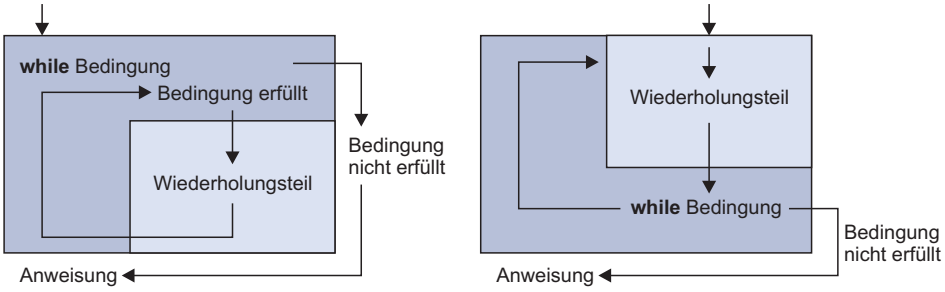


Abb. 4.4-5: Unterschied zwischen einer `while-do` und einer `do-while`-Wiederholung.

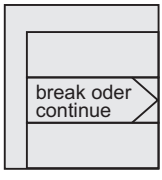
Zeichnen Sie ein Struktogramm des Programms `Kasse` sowie ein UML-Diagramm in der Knoten-Notation.

n + 1/2 -Schleife

Es gibt Fälle, in denen es notwendig ist, innerhalb der Wiederholungsanweisungen die laufende Wiederholung abzubrechen. Dies ist insbesondere dann sinnvoll, wenn z. B. bei einer Berechnung innerhalb einer Wiederholung Fehler auftreten, die eine weitere Verarbeitung der Wiederholungsanweisungen überflüssig machen.

Die Abbildung in der Marginalspalte zeigt eine mögliche Darstellung in einem Struktogramm. Diese Darstellung wird aber *nicht* einheitlich verwendet.

Struktogramm-Darstellung (nicht einheitlich verwendet)



Konzeptionell hat man die Kontrollstruktur Wiederholung so verallgemeinert, dass

- innerhalb des Wiederholungsteils ein oder mehrere Unterbrechungen *(breaks)* oder Aussprünge programmiert werden können, die bewirken, dass aus dem Wiederholungsteil hinter das Ende der Wiederholung verzweigt bzw. gesprungen wird,

- die aktuelle Wiederholung abgebrochen wird und sofort eine neue Wiederholung beginnt *(continue)*, d. h. es wird zur jeweiligen Wiederholungsbedingung verzweigt, und
- die Methode, in der sich die Wiederholung befindet, durch ein `return` beendet wird (In Java ist dies nicht nur in einer Funktion, sondern auch in einer Prozedur durch `return;` möglich, siehe »Prozeduren, Funktionen und Methoden«, S. 207).

Wird eine Wiederholung auf eine dieser beiden Arten verlassen, dann spricht man von einer **n + 1/2 -Schleife**.

Beispiel

Wetter

Für die Auswertung von Wetterdaten wird ein Programm benötigt, das die Anzahl der Frostwerte und die durchschnittliche Frosttemperatur berechnet.

```
// Auswertung von Wetterdaten

import inout.Console;

public class Wetter
{
 public static void main (String args[])
 {
  double temperatur = 0.0, summeMinus = 0.0;
  int anzahlTemp = -1, anzahlTempMinus = 0;
  final int ende = 999;
  System.out.println("Bitte Temperaturen eingeben ");
  System.out.println("Ende bei Eingabe von 999:");

  do
  {
    anzahlTemp = anzahlTemp + 1;
    temperatur = Console.readInt();
    if (temperatur >= 0)
     continue;
    anzahlTempMinus ++;
    summeMinus = summeMinus + temperatur;
   } while (temperatur != ende);

  if (anzahlTemp > 0)
  {
   System.out.println
    ("Durchschnittlicher Minuswert: "
     + (summeMinus / anzahlTempMinus));
   System.out.println
    ("Anzahl Temperaturwerte: " + anzahlTemp);
   System.out.println
    ("Anzahl Temperaturwerte im Minus: " + anzahlTempMinus);
  }
  else
   System.out.println("Kein Wert eingegeben");
 }
}
```

Ein Programmlauf sieht wie folgt aus:

```
Bitte Temperaturen eingeben
Ende bei Eingabe von 999:
12
-2
-15
0
20
25
999
Durchschnittlicher Minuswert: -8.5
Anzahl Temperaturwerte: 6
Anzahl Temperaturwerte im Minus: 2
```

Durch die `continue`-Anweisung werden die Berechnungen für die Frostwerte übersprungen und es wird am Anfang der Wiederholung fortgefahren.

Zeichnen Sie zu dem Programm Wetter ein Struktogramm und ein UML-Diagramm in Fluss-Notation.

4.5 Die Zählschleife und die Endlosschleife *

Ist es in einer Problemstellung nötig, Anweisungen zu wiederholen und ist die Anzahl der Wiederholungen im Voraus bekannt, dann wird dazu die Zählschleife verwendet – in Java lautet sie `for (int von; bis; schrittweite)`. Für Sonderfälle kann durch `for (;;)` eine Endlosschleife programmiert werden.

Sollen eine oder mehrere Anweisungen (Blöcke) für eine gegebene Zahl von Wiederholungen durchlaufen werden, so ist das Konzept der **Zählschleife** – auch Laufanweisung genannt – zu verwenden (Abb. 4.5-1) (siehe auch »Iteration über Felder: Die erweiterte for-Schleife«, S. 198).

Die Anzahl der Wiederholungen wird durch eine Zählvariable mitgezählt und die Bedingung so gewählt, dass nach der geforderten Wiederholungszahl der Abbruch erfolgt. Einige Probleme erfordern auch ein Abwärtszählen. Auch dies kann mit der Zählschleife programmiert werden.

Java-Syntax

Die Syntax sieht in Java wie folgt aus:

for-Statement ::=
`for` `(` *[ForInit]* `;` *[Expression]* `;` *[ForUpdate]* `)` *Statement*

Einsatz

Die Zählschleife eignet sich besonders gut für die **Erstellung von Tabellen** und für das **Durchlaufen von Feldern** (siehe »Felder«, S. 173).

Beispiel

Es wird eine Tabelle benötigt, die den Nettobetrag, die volle Mehrwertsteuer (MwSt), den Bruttobetrag, die ermäßigte MwSt

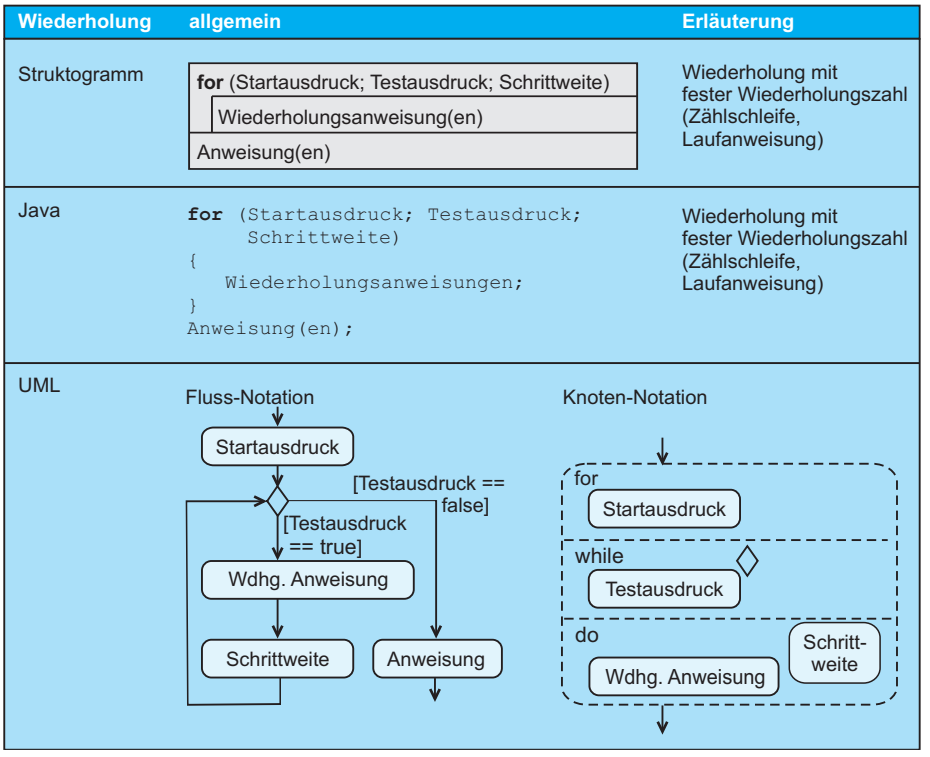


Abb. 4.5-1: Notationen für Zählschleifen.

und den Bruttobetrag für € 1.- bis € 100.- in Einerschritten aufführt.

MWSTTabelle

```
// Ausgabe einer MWST-Tabelle

public class MWSTTabelle
{
 public static void main (String args[])
 {
  final int MWST_VOLL = 19;
  final int MWST_ERMAESSIGT = 7;
  float mwst_v, mwst_e;

  System.out.println("MWST-Tabelle in €");
  System.out.println("Netto\t19%\tBrutto\t\t7%\tBrutto");

  for (int i = 1; i <= 30; i++)
  {
    mwst_v = i * MWST_VOLL / 100.0f;
    mwst_e = i * MWST_ERMAESSIGT / 100.0f;
    System.out.println(i +  "\t" + mwst_v
     + "\t" + (i + mwst_v) +  "\t\t"
    + mwst_e + "\t" + (i + mwst_e));
```

```
    }
  }
}
```

Die Ausgabe sieht wie folgt aus (Ausschnitt):

```
MWST-Tabelle in €
Netto   19%    Brutto       7% Brutto
1   0.19    1.19         0.07    1.07
2   0.38    2.38         0.14    2.14
3   0.57    3.57         0.21    3.21
4   0.76    4.76         0.28    4.28
5   0.95    5.95         0.35    5.35
6   1.14    7.14         0.42    6.42
7   1.33    8.33         0.49    7.49
8   1.52    9.52         0.56    8.56
9   1.71    10.71        0.63    9.63
10  1.90    11.90        0.70    10.7
```

1. Fügen Sie in das Programm nach jeweils 10 Ausgabezeilen einen Querstrich ein, um die Lesbarkeit zu verbessern.
2. Ändern Sie das Programm so ab, dass der Benutzer eingibt, von welchem Startwert bis zu welchem Zielwert er die Tabelle haben möchte. Stellen Sie dabei sicher, dass der Startwert kleiner als der Zielwert ist.
3. Zeichnen Sie ein Struktogramm und ein UML-Diagramm in Knoten-Notation für das Programm `MWSTTabelle`.

Java

In Java wird im Startausdruck der Zählschleife die **Zählervariable** deklariert und initialisiert. Die Zählervariable ist dann nur innerhalb der for-Schleife gültig und kann außerhalb *nicht* abgefragt werden. Der Testausdruck wird vor jedem Schleifendurchlauf geprüft. Sobald er false ergibt, wird die Schleifenausführung gestoppt. Bei Schrittweite handelt es sich um einen Ausdruck, der angibt, ob die Zählervariable erhöht oder erniedrigt werden soll und um welchen Betrag. Die Zählervariable wird als Nebeneffekt des Schrittweiten-Ausdrucks pro Durchlauf entsprechend modifiziert.

Beispiel

```
for (int grad = vonWinkel; grad <= bisWinkel;
         grad = grad  + 1)
{ ...
}
```

Soll der Wert der Zählervariablen auch außerhalb der `for`-Schleife abgefragt werden, z. B. nach einem Verlassen der Schleife durch `break`, dann muss die Zählervariable außerhalb der `for`-Schleife deklariert werden. Dies sollte jedoch der Sonderfall sein.

Programmierhinweis

Die Java-Syntax erlaubt noch mehr Möglichkeiten. Insbesondere können Ausdrücke weggelassen werden. Auf diese Möglichkeit sollte verzichtet werden. Die `for`-Anweisung in Java ist bei undis-

ziplinierter Verwendung eine gefährliche Konstruktion. Sie sollte daher nur defensiv – wie im obigen Beispiel – benutzt werden.

Konvention

Als Bezeichner für die Zählervariablen werden oft die Buchstaben i, j und k verwendet.

Vorsicht, wenn das Testkriterium eine Gleitpunktzahl ist und auf Gleichheit getestet wird. Durch Rundungsfehler kann es sonst zu einer Endlosschleife kommen (siehe »Rechengenauigkeit mit Gleitpunkt-Zahlen«, S. 77).

Beispiel

Die `for`-Schleife

```
for (double x = 0.0; x!= 25.0; x = x + 0.02 )
```

endet wahrscheinlich nie!

n + 1/2-Schleife

Wie bei der bedingten Wiederholung (siehe »Bedingte Wiederholung und n + 1/2-Schleife«, S. 122) kann eine `for`-Schleife mit den Anweisungen `break` und `continue` vorzeitig verlassen werden.

Endlosschleife

In einigen Sonderfällen ist es nötig, eine Schleife endlos zu durchlaufen.

Beispiele

- Eine Ampelsteuerung muss »rund um die Uhr« die Ampeln steuern.
- Eine Messdatenerfassung muss laufend die Messwerte auswerten.

In Java kann eine Endlosschleife auf drei Arten programmiert werden:

- `for (;; )`
- `while (true) { }` (siehe »Bedingte Wiederholung und n + 1/2-Schleife«, S. 122)
- `do { } while (true)`

4.6 Termination von Schleifen *

Um sicherzustellen, dass Programme mit bedingten Wiederholungen terminieren, müssen Anweisungen innerhalb des Wiederholungsteils eine Rückwirkung auf die Wiederholungsbedingung haben. Mit Hilfe einer Wertetabelle und einem manuellen Durchgehen von Wiederholungen lassen sich Terminationsfehler oft identifizieren.

Problem: bedingte Wiederholungen

Enthält ein Programm Wiederholungsanweisungen, dann muss sichergestellt sein, dass sie nach endlich vielen Wiederholungen beendet werden. Ausnahmen davon sind bewusst programmierte Endlosschleifen für Spezialfälle (siehe »Die Zählschleife und die Endlosschleife«, S. 131). Ein Programm endet in der Regel immer, wenn es keine bedingten Wiederholungen enthält. Das

Hauptaugenmerk ist daher auf die **bedingten Wiederholungen** zu richten.

Überlegen Sie, welche Bedingungen erfüllt sein müssen, damit eine bedingte Wiederholung überhaupt enden kann?

Beispiel 1a

Ein Teilprogramm zur Erstellung von Rechnungen soll die Rechnungsposten auf der Rechnung addieren.

Rechnungs posten

```
// Programm, das Rechnungsposten in einer Rechnung addiert

import inout.Console;

public class Rechnungsposten
{
 public static void main (String args[])
 {
  double teilsumme; //Eingabe
  double summe = 0.0; //Ausgabe
  int anzahl = 0; //Hilfsgröße

  System.out.println("Geben Sie bitte die Teilsumme ein:");
  teilsumme = Console.readInt();
  while (teilsumme > 0)
  {
      summe = summe + teilsumme;
      anzahl = anzahl + 1;
   }
  System.out.println("Anzahl Rechnungsposten: " + anzahl);
  System.out.println("Summe Rechnungposten: " + summe);
 }
}
```

Für eine Rechnung liegen folgende Teilsummen vor: 50, 200, 30, 40, 0.

Prüfen Sie mit diesen Teilsummen das Programm `Rechnungsposten`.

Trockentest mit Wertetabelle

Vor der Ausführung eines Programms ist es immer sinnvoll, zunächst einen »Trockentest« durchzuführen. Dazu wird eine Wertetabelle aufgestellt, in die die aktuellen Werte jeder Variablen eingetragen werden. Jeder Variablenname wird aufgeführt. Die Zuweisungen von Werten zu Variablen, die bei der Ausführung des Programms vorgenommen werden, werden in die Tabelle eingetragen. Wird einer Variablen ein neuer Wert zugewiesen, dann wird der alte Wert durchgestrichen und der neue notiert.

Beispiel 1b

Für das Beispiel 1a erhält man die Tab. 4.6-1. Wie die Tabelle zeigt, ist die Wiederholungsbedingung immer erfüllt. Die Wiederholung wird also unendlich oft ausgeführt. Das Programm

endet nicht. Der Wiederholungsteil enthält einen Fehler. Es wurde vergessen, im Wiederholungsteil den neuen Wert für Teilsumme einzulesen:

```
System.out.println("Geben Sie bitte die Teilsumme ein:");
teilsumme = Console.readInt();
```

Durch die Ausführung des Programms mit der Hand und dem Eintragen der aktuellen Variablenwerte in eine Wertetabelle erkennt man solche Fehler.

Variable	aktueller Inhalt
teilsumme	50
anzahl	0 | 1 | 2 | 3 | 4 | ...
summe	0 | 50 | 100 | 150 | 200 | ...
teilsumme > 0	ja | ja | ja | ja | ja | ...

Tab. 4.6-1: Wertetabelle.

Das Beispiel zeigt deutlich die Gefahren einer `while`- und einer `do`-Wiederholung. Es muss immer sichergestellt sein, dass die Anzahl der Wiederholungen nach endlich vielen Durchläufen endet, d. h. die Wiederholungen dürfen *nicht* für irgendeine Eingabe unendlich sein. Es kann jedoch auch vorkommen, dass in einem Programm eine bestimmte Eingabemöglichkeit nicht bedacht wurde, so dass die Forderung nach **Termination** oder **dynamischer Endlichkeit** des Programms verletzt wird. Ebenso ist es möglich, dass ein Fehler in der Problemlösung vorliegt.

Frage Wie kann man sicherstellen, dass ein Programm immer terminiert, d. h. für alle Eingaben endet? Wie kann man überprüfen, ob ein Programm in eine Endlosschleife kommen kann?

Antwort Ein Programm endet in der Regel immer, wenn es *keine* bedingten Wiederholungen enthält. Damit eine bedingte Wiederholung überhaupt enden kann, muss es innerhalb des Wiederholungsteils eine oder mehrere Anweisungen geben, die eine Rückwirkung auf die Wiederholungsbedingung haben.

Zur Sicherstellung der dynamischen Endlichkeit bei bedingten Wiederholungen muss deshalb folgende Minimalbedingung erfüllt sein:

Durch mindestens eine Anweisung im Wiederholungsteil muss eine Variable derart verändert werden, dass nach einer endlichen Anzahl von Durchläufen die Endebedingung erfüllt ist (siehe auch »Termination von Schleifen«, S. 313).

Methode Allgemein kann man folgende Methode anwenden: Man suche irgendeine ganzzahlige Größe `g`, die jedesmal, wenn die Wieder-

holung ausgeführt wird, einen niedrigeren (höheren) Wert hat als bei der vorherigen Wiederholung, aber nie kleiner (größer) werden kann als ein bestimmtes Minimum (Maximum).

Beispiel

Demo Termination

```
public class DemoTermination
{
 public static void main (String args[])
 {
  int g = 100; // ganzzahlig
  int minimum = 0, anzahl = 0;
  int summe = 0;

  while (g > minimum)
  {
      g = g - 1; //Verkleinern
      anzahl = anzahl + 1;
      summe = summe + anzahl;
  }
  System.out.println
   ("Summe von 1 bis " + anzahl + ": " + summe);
 }
}
```

Der Programmlauf ergibt folgendes Ergebnis:

```
Summe von 1 bis 100: 5050
```

Wie sieht ein Programm aus, bei dem die Variable g hochgezählt wird. Wie sehen entsprechende Programme bei der do-Wiederholung aus?

4.7 Der Aufruf *

Teilaufgaben werden in eigenständige Programme – in Java in eigenständige Methoden – ausgelagert. Solche Programme bzw. Methoden können von anderen Programmen aus aufgerufen werden. Ein Aufruf bewirkt, dass von der Aufrufstelle zum gerufenen Programm bzw. der gerufenen Methode verzweigt wird. Nach Ausführung des gerufenen Programms bzw. der gerufenen Methode wird *hinter* die Aufrufstelle des rufenden Programms zurückgekehrt und der Programmablauf wird im ursprünglichen Programm fortgesetzt. Da ein Aufruf den Kontrollfluss ändert, gehört er zu den Kontrollstrukturen.

Teilaufgaben = eigenständige Methoden

In vielen Programmen werden Teilaufgaben oft mehrmals benötigt. Solche Teilaufgaben werden daher als eigenständige Methoden formuliert und von der Methode aus, in der die Teilaufgabe benötigt wird, aufgerufen. **Aufruf** bedeutet, dass an der Stelle, an der die Teilaufgabe auszuführen ist, eine Verzweigung zu der Teilaufgabe, d. h. zu der entsprechenden Methode, erfolgt. Nach Abschluss der Teilaufgabe wird das Programm hinter der Aufruf- bzw. Verzweigungsstelle fortgesetzt. Der Methode, die aufgeru-

fen wird, können noch sogenannte aktuelle Parameter übergeben werden. Auf diese Konzepte wird detailliert in »Prozeduren, Funktionen und Methoden«, S. 207, eingegangen. Die verschiedenen Notationen zeigt die Abb. 4.7-1.

Aufruf	allgemein	Erläuterung
Struktogramm	Anweisung1 Methoden-Name (aktuelle Parameter) Anweisung2	Nach Ausführung der aufgerufenen Methode wird die rufende Methode hinter der Aufrufstelle fortgesetzt (Anweisung2).

Java	`Anweisung1;`
	`Methoden-Name (aktuelleParameter);`
	`Anweisung2;`
		Ein Aufruf erfolgt durch Angabe des Methoden-Namens, gefolgt von der Liste der aktuellen Parameter.

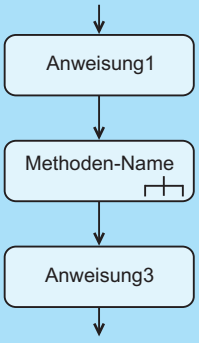
UML		Durch ein Rechensymbol rechts unten in einem abgerundeten Rechteck wird ein Aktivitätsaufruf angegeben *(sub activity indicator)*.

Abb. 4.7-1: Darstellungsformen des Aufrufs.

Da durch einen Aufruf die sequenzielle Ausführung der Anweisungen unterbrochen wird, zählt der Aufruf zu den Kontrollstrukturen.

Beispiel

```
public class DemoAufruf
{
 public static void druckeLinie() //Methode
 {
    System.out.println("-------------------------");
 }

 public static void main (String args[])
 {
  druckeLinie(); //1. Aufruf
  System.out.println("Top 5 der Bücher");
  druckeLinie(); //2. Aufruf
  System.out.println("Codeknacker");
  System.out.println("XHTML & CSS");
```

```
  System.out.println("Webdesign & Web-Ergonomie");
  System.out.println("Praktische Projektplanung");
  System.out.println("Tabellenkalkulation");
  druckeLinie(); // 3. Aufruf
 }
}
```

Der Programmlauf sieht wie folgt aus:

```
-------------------------
Top 5 der Bücher
-------------------------
Codeknacker
XHTML & CSS
Webdesign & Web-Ergonomie
Praktische Projektplanung
Tabellenkalkulation
-------------------------
```

In Java können in einer Klasse mehrere Methoden stehen, die sich gegenseitig aufrufen können. Beim Aufruf einer Methode steht hinter dem Methodennamen immer ein rundes Klammerpaar, in dem sich – falls vorhanden – die sogenannten aktuellen Parameter befinden.

Die Abb. 4.7-2 zeigt das Programm in der Struktogramm-Notation, die Abb. 4.7-3 in der UML-Notation. In der UML-Notation kann ein Programm in ein Rechteck mit abgerundeten Ecken eingeschlossen werden. Der Programmname wird links oben eingetragen. Dadurch ist es möglich den Programmnamen in einem Aktivitätsdiagramm beim Aufruf zu referenzieren.

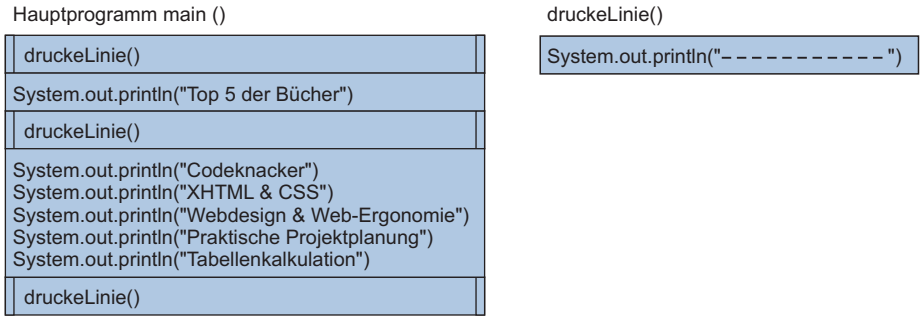


Abb. 4.7-2: Das Programm `DemoAufruf` in der Struktogramm-Notation.

Ergänzen Sie die Klasse `DemoAufruf` um eine weitere Methode, die statt einer Linie Gleichheitszeichen (=) ausgibt. Rufen Sie diese Methode hinter der Überschrift anstelle von `Linie()` auf.

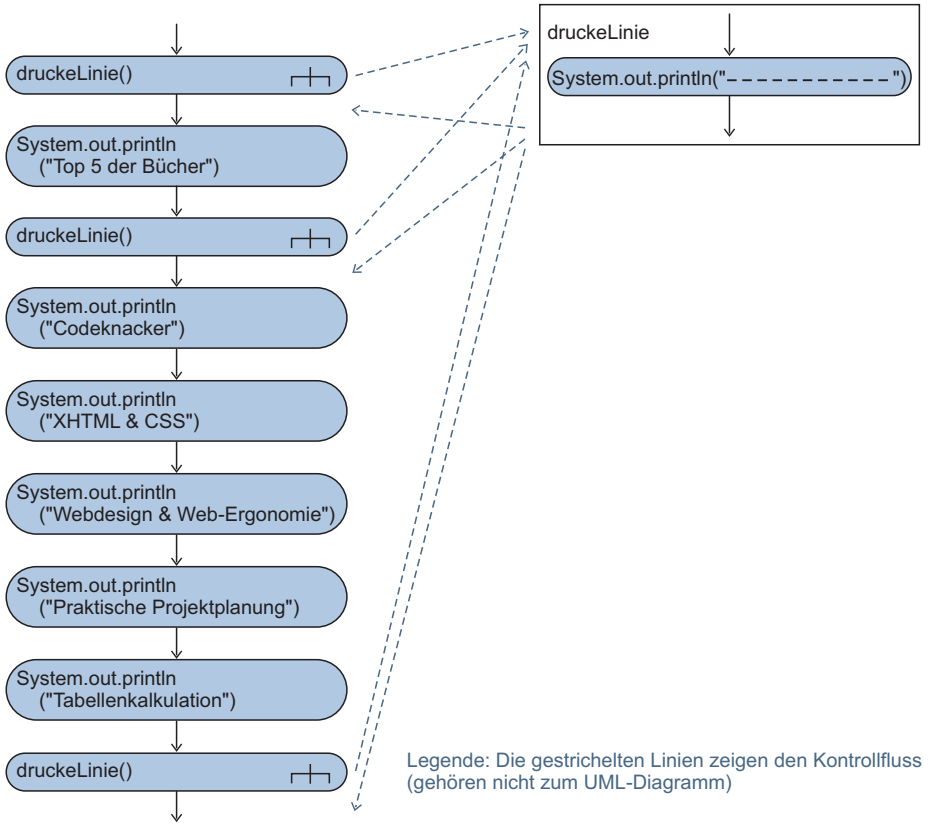


Abb. 4.7-3: Das Programm `DemoAufruf` in UML-Notation mit Veranschaulichung der Aufrufe.

4.8 Geschachtelte Kontrollstrukturen *

Kontrollstrukturen können beliebig tief ineinander geschachtelt werden, um komplexe Ablaufstrukturen zu programmieren. Auswahlketten liegen vor, wenn Auswahlanweisungen im else-Zweig wieder Auswahlanweisungen enthalten. In der UML-Knoten-Notation und in einigen Programmiersprachen gibt es dafür eine besondere Notation, in Java nicht. Bei ineinander geschachtelten Auswahlanweisungen ist durch Kommentare in Java (wegen seiner ungünstigen Syntaxstruktur) auf eine korrekte Schachtelung zu achten. Geschachtelte Kontrollstrukturen können mit Marken versehen werden, so dass mit `break` und `continue` verschachtelte Konstrukte verlassen werden können.

Schachtelung

Komplexe Abläufe können durch Schachtelung von Kontrollstrukturen beschrieben werden. Innerhalb von Wiederholungsanweisungen können wieder Wiederholungsanweisungen

oder/und Auswahlanweisungen stehen. Im Prinzip kann man die Kontrollstrukturen in beliebiger Kombination beliebig tief ineinander schachteln.

Beispiel

Rabattstaffel

```
// Ausgabe einer Rabattstaffel

public class Rabattstaffel
{
 public static void main (String args[])
 {
  System.out.println("Rabattstaffel");
  System.out.println("€\t5%\t10%\t15%\t20%\t25%");
 for (int betrag = 100; betrag <= 1000;
    betrag = betrag + 100)
  {
    System.out.print(betrag);
    for (int rabattsatz = 5; rabattsatz <= 25;
     rabattsatz = rabattsatz + 5)
      System.out.print
       ("\t" + betrag * rabattsatz / 100);
    System.out.println();
   }
 }
}
```

Das Ergebnis des Programmlaufs sieht wie folgt aus:

```
Rabattstaffel
€       5%      10%     15%     20%     25%
100     5       10      15      20      25
200     10      20      30      40      50
300     15      30      45      60      75
400     20      40      60      80      100
500     25      50      75      100     125
600     30      60      90      120     150
700     35      70      105     140     175
800     40      80      120     160     200
900     45      90      135     180     225
1000    50      100     150     200     250
```

Die Struktogramm-Darstellung zeigt die Abb. 4.8-1, die UML-Darstellung die Abb. 4.8-2.

Insbesondere bei geschachtelten Auswahlanweisungen ist auf den korrekten Abschluss jeder Anweisung zu achten.

Korrekter Abschluss

Das Struktogramm der Abb. 4.8-3 sieht in Java folgendermaßen aus:

Beispiele

```
if (n > 0)
  if (a > b)
    z = a;
  else
    z = b;
  //end if
//end if
```

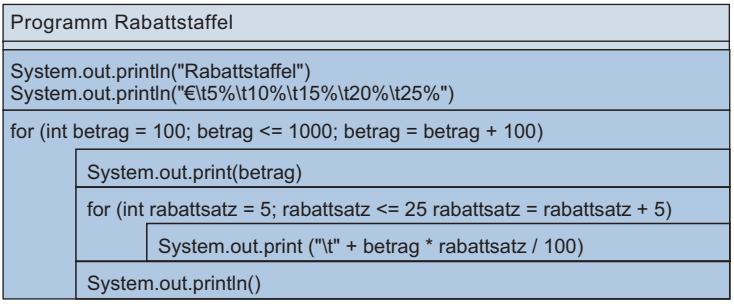


Abb. 4.8-1: Struktogramm des Programms `Rabattstaffel`.

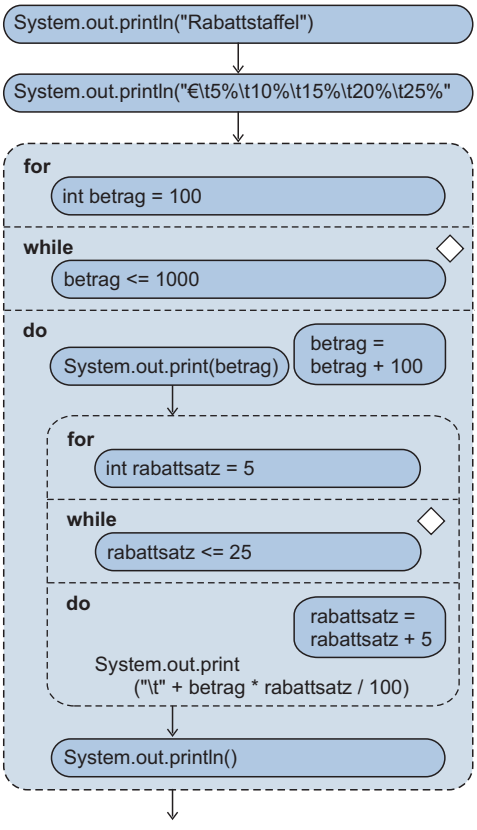


Abb. 4.8-2: UML-Aktivitätsdiagramm in Knotennotation des Programms `Rabattstaffel`.

Das in der Abb. 4.8-4 dargestellte, unwesentlich geänderte Programm sieht in Java folgendermaßen aus:

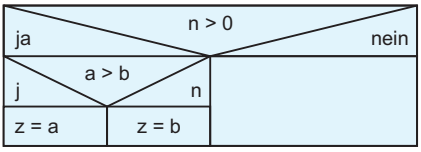


Abb. 4.8-3: Beispiel für die Schachtelung von `if`-Anweisungen.

```
if (n > 0)
  {
    if (a > b)
      z = a;
  } //end if
else
  z = b;
//end if
```

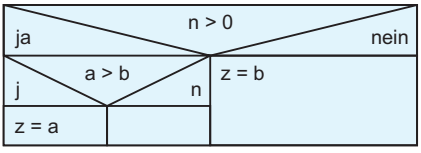



Abb. 4.8-4: Beispiel für die Schachtelung von `if`-Anweisungen.

Programmierhinweis

Ende-Kommentare einfügen:
Durch die ungünstige Syntaxstruktur in Java kann es leicht zu Fehlinterpretationen der Semantik beim Lesen eines Programms kommen. Daher sind zusätzliche Kommentare unbedingt nötig!

Auswahlketten

Manche Problemlösungen enthalten regelmäßig ineinandergeschachtelte Auswahlanweisungen, sogenannte **Auswahlketten**. Einige Programmiersprachen besitzen zur Formulierung dieser Auswahlketten besondere Sprachkonstrukte, z. B. Ada mit `elsif`. Da in Java ein `if`-Konstrukt nicht mit `end if` abgeschlossen wird, ist ein besonderes Sprachkonstrukt *nicht* erforderlich. In der UML-Knoten-Notation gibt es eine besondere Darstellung für Auswahlketten (Abb. 4.8-6).

Beispiel

Das Struktogramm der Abb. 4.8-5 sieht in Java folgendermaßen aus:

```
if (Bedingung1)
  Anweisung1
else
```

```
  if (B2)
    A2;
  else
    if (B3)
      A3;
    else A4;
//end if
```

Die Abb. 4.8-6 zeigt die Darstellung in der UML-Knoten-Notation.

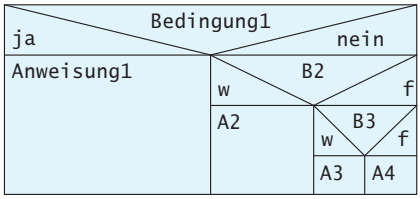


Abb. 4.8-5: Beispiel für die Darstellung einer Auswahlkette als Struktogramm.

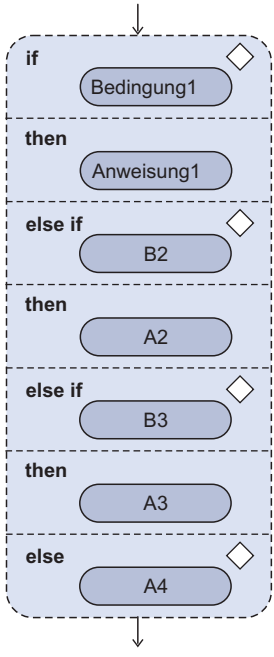


Abb. 4.8-6: Auswahlkette in UML-Notation als Aktivitätsdiagramm.

Beenden geschachtelter Anweisungen

In der Praxis müssen z. B. aufgrund erkannter Fehler ineinandergeschachtelte Wiederholungsanweisungen und Mehrfach-

Auswahlanweisungen vollständig verlassen werden. Ein Beenden der gerade aktiven Anweisung reicht dazu nicht aus.

Java

In Java ist es möglich, jede Anweisung und jeden Anweisungsblock mit einer Marke *(label)* zu versehen. Eine Marke ist ein Bezeichner gefolgt von einem Doppelpunkt, z. B. `Schleife1:`

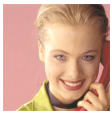
Durch Marken versehene Anweisungen werden von den Anweisungen `break` oder `continue` »angesprungen«, die irgendwo innerhalb der mit Marken versehenen Anweisung auftreten. Hinter `break` oder `continue` muss dann der entsprechende Markenbezeichner angegeben werden.

Beispiel

```
Schleife1: //Name der Schleife
while (Ausdruck)
{
  Anweisungen;
  while (Ausdruck)
  {
    Anweisungen;
    if (Fehler1) break Schleife1;
    Anweisungen;
    if (Fehler2) continue Schleife1;
    Anweisungen;
  }
  Anweisungen;
} //Ende Schleife1
```

Mit `break` wird hinter das Ende der `Schleife1`, mit `continue` auf die Marke `Schleife1` verzweigt.

4.9 OptiTravel: Zeitvergleich *



Frau Jung erhält den Auftrag, ein Programm zu schreiben, das es ermöglicht, die ermittelten Zeiten für Auto, Bahn und Flug zu vergleichen. Es sollen die Transportmittel sortiert nach minimaler Zeit, mittlerer Zeit und maximaler Zeit ausgegeben werden. Da Frau Jung in ihrer Ausbildung gelernt hat, immer vom konkreten Problem zu abstrahieren, damit ein Programm auch in anderen Kontexten eingesetzt werden kann, verallgemeinert sie die Aufgabenstellung. Es werden drei `int`-Werte eingegeben. Ausgegeben werden die Werte in aufsteigender Folge (min, mittel, max). Bevor Frau Jung mit der Programmierung beginnt, überlegt sie sich die Entscheidungsstruktur anhand eines **Entscheidungsbaums** (Abb. 4.9-1).

Das anhand des Entscheidungsbaums entwickelte Programm sieht wie folgt aus:

Vergleich

```
// Vergleich von 3 Werten

import inout.Console;
public class Vergleich
```

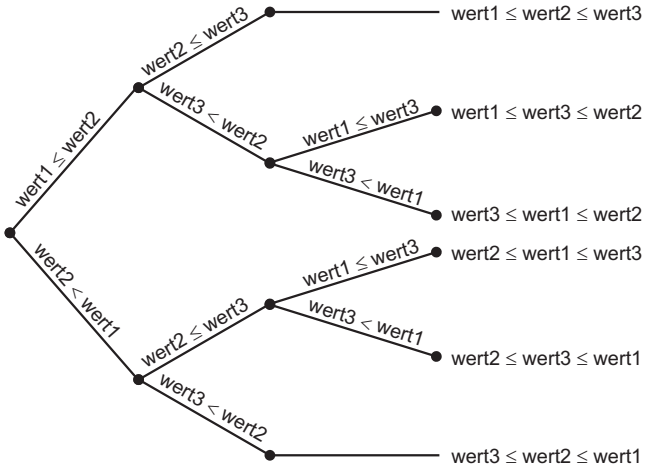


Abb. 4.9-1: Entscheidungsbaum, der die verschiedenen Alternativen anschaulich zeigt.


```
{
 public static void main (String args[])
 {
  int wert1, wert2, wert3; //3 Vergleichswerte
  System.out.print("Wert 1 eingeben: ");
  wert1 = Console.readInt();
  System.out.print("Wert 2 eingeben: ");
  wert2 = Console.readInt();
  System.out.print("Wert 3 eingeben: ");
  wert3 = Console.readInt();
  if (wert1 <= wert2)
     if (wert2 <= wert3) //wert1 <= wert2 <= wert3
      {
          System.out.println("Min: Wert 1 = " + wert1);
          System.out.println("Mittel: Wert 2 = " + wert2);
          System.out.println("Max: Wert 3 = " + wert3);
      }
      else //(wert1 <= wert2  und wert3 <= wert2)
        if (wert1 <= wert3)
         {
            System.out.println("Min: Wert 1 = " + wert1);
            System.out.println("Mittel: Wert 3 = " + wert3);
            System.out.println("Max: Wert 2 = " + wert2);
         }
         else //wert3 < wert1
         {
            System.out.println("Min: Wert 3 = " + wert3);
            System.out.println("Mittel: Wert 1 = " + wert1);
            System.out.println("Max: Wert 2 = " + wert2);
         }
   else //wert2 < wert1
   {
      if (wert2 <= wert3)
```

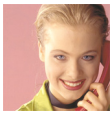
```
        if (wert1 <= wert3)
         {
            System.out.println("Min: Wert 2 = " + wert2);
            System.out.println("Mittel: Wert 1 = " + wert1);
            System.out.println("Max: Wert 3 = " + wert3);
         }
         else //wert3 < wert1
         {
            System.out.println("Min: Wert 2 = " + wert2);
            System.out.println("Mittel: Wert 3 = " + wert3);
            System.out.println("Max: Wert 1 = " + wert1);
         }

      else //wert3 < wert2
      {
          System.out.println("Min: Wert 3 = " + wert3);
          System.out.println("Mittel: Wert 2 = " + wert2);
          System.out.println("Max: Wert 1 = " + wert1);
      }
   }
 }
}
```

Ein Programmlauf sieht wie folgt aus:

```
Wert 1 eingeben: 55
Wert 2 eingeben: 6
Wert 3 eingeben: 44
Min: Wert 2 = 6
Mittel: Wert 3 = 44
Max: Wert 1 = 55
```

Zeichnen Sie ein Struktogramm für dieses Programm.

4.10 OptiTravel: Funktionsauswahl *



Zur Vorbereitung auf die Einbindung der verschiedenen Funktionen für das Programm `OptiTravel`, erhält Frau Jung in Ihrer Rolle als Junior-Programmiererin die Aufgabe, eine Funktionsanzeige und -auswahl für die Konsole zu programmieren. Frau Jung erinnert sich an die verschiedenen Möglichkeiten, die Kontrollstrukturen bieten, und setzt eine endlose `for`-Schleife mit anschließender Mehrfachauswahl ein.

Funktions-
auswahl

```
// Auswahl einer Funktion
import inout.Console;

public class Funktionsauswahl
{
 public static void main (String args[])
 {
  char auswahl;
  //StartAuswahl: //Name der Schleife
  for(;;)
  {
    System.out.println("Bitte Funktion auswählen:");
```

```
    System.out.println("Funktion 1");
    System.out.println("Funktion 2");
    System.out.println("Funktion 3");
    System.out.println("Abbruch: 9");
    System.out.println("Bitte Ziffer 1, 2, 3 oder 9 eingeben:");

    auswahl = Console.readChar();
    if (auswahl =='9') break;

    switch (auswahl)
    {
      case '1': System.out.println
       ("Funktion 1 wird ausgeführt"); break;
      case '2': System.out.println
       ("Funktion 2 wird ausgeführt"); break;
      case '3': System.out.println
       ("Funktion 3 wird ausgeführt"); break;

      default: System.out.println("Fehlerhafte Eingabe: " +
               "Bitte nur 1, 2, 3 oder 9 eingeben");
               continue; //StartAuswahl;
    }
  }
  System.out.println("Ende des Programms");
 }
}
```

Ein Programmlauf sieht folgendermaßen aus:

```
Bitte Funktion auswählen:
Funktion 1
Funktion 2
Funktion 3
Abbruch: 9
Bitte Ziffer 1, 2, 3 oder 9 eingeben:
2
Funktion 2 wird ausgeführt
Bitte Funktion auswählen:
Funktion 1
Funktion 2
Funktion 3
Abbruch: 9
Bitte Ziffer 1, 2, 3 oder 9 eingeben:
7
Fehlerhafte Eingabe: Bitte nur 1, 2, 3 oder 9 eingeben
Bitte Funktion auswählen:
Funktion 1
Funktion 2
Funktion 3
Abbruch: 9
Bitte Ziffer 1, 2, 3 oder 9 eingeben:
9
Ende des Programms
```

Zeichnen Sie für das Programm `Funktionsauswahl` ein Struktogramm und ein UML-Diagramm in Knoten-Notation.

4.11 Anordnung von Auswahlanweisungen *

Oft legen Problemstellungen ein Hintereinanderreihen von Auswahlanweisungen nahe. Um die Laufzeit zu optimieren, ist jedoch zu prüfen, ob durch eine Mehrfachauswahl oder geschachtelte Anweisungen die Bedingungsabfragen und die Berechnung von Ausdrücken reduziert werden können.

Ziel: geringste Laufzeit

Jede Bedingung, die bei der Ausführung eines Programms ausgewertet werden muss, kostet Zeit. In Abhängigkeit von der Aufgabenstellung muss daher genau analysiert werden, welche Anordnung der Auswahlanweisungen die geringste Laufzeit ermöglicht.

Beispiel 1a

Eine Schwachstromversicherung deckt alle Schäden an empfindlichen elektronischen Bürogeräten ab, die beispielsweise durch einen plötzlichen Stromausfall oder unsachgemäße Handhabung entstehen könnten. Der Jahresbeitrag für die Versicherungssumme errechnet sich im Falle einer Einzelversicherung für Büromaschinen aufgrund der Vertragsbedingungen (Auszug) der Tab. 4.11-1.

Versicherungssumme in Euro	Prämie in Promille
bis 5.000	16,8
von 5.001 bis 10.000	12,6
über 10.000	8,4

Tab. 4.11-1: Tarife für eine Schwachstromversicherung.

Eine unreflektierte Umsetzung dieser Aufgabenstellung führt zu der Lösung 1 (Abb. 4.11-1).

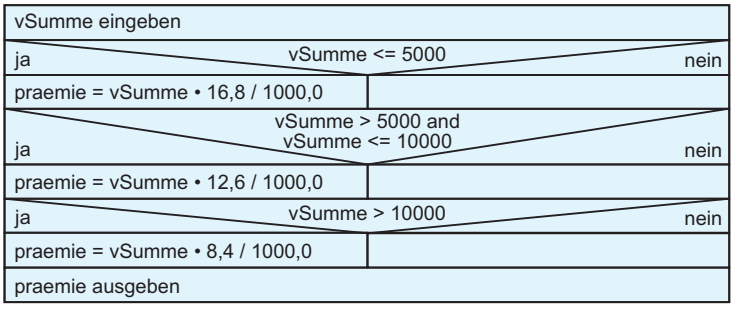


Abb. 4.11-1: Direkte Umsetzung einer Versicherungstabelle in Programmcode.

Bei dieser Lösung werden unabhängig von der konkreten Versicherungssumme nacheinander *immer* drei Bedingungen überprüft. Ist die konkrete `versicherungssumme <= 5000` Euro, dann sind die beiden folgenden Abfragen eigentlich überflüssig. Dennoch werden sie ausgeführt. Dies führt zu einer unnötigen Verlängerung der Laufzeit des Programms.

Von der Aufgabenstellung her handelt es sich um eine Mehrfachauswahl. Aus drei Fällen muss genau ein Fall ausgewählt werden.

Versicherung1

In Java lässt sich diese Aufgabe durch eine `switch`-Anweisung lösen.

```
// Laufzeitoptimierte Berechnung von
// Versicherungsprämien
// Einsatz des switch-Konstrukts
import inout.Console;
public class Versicherung1
{
 public static void main (String args[])
 {
  double vSumme;
  System.out.println
   ("Geben Sie die Versicherungssumme ein: ");
  vSumme = Console.readDoubleComma();
  double praemie = 0.0;
  // Lösung mit switch - case - Konstrukt
  int fall = (int)((vSumme - 0.01) / 5000.0);
  switch (fall)
  {
    case 0: praemie = vSumme * 16.8 / 1000.0; break;
    case 1: praemie = vSumme * 12.6 / 1000.0; break;
    default: praemie = vSumme * 8.4 / 1000.0; break;
  }
  System.out.println("Prämie: " + praemie);
  //floor ist eine Methode der Klasse Math
  //floor ermittelt den größten double-Wert, der nicht größer
  //als das Argument (hier praemie) und gleich
  //einer ganzen Zahl ist
  System.out.println("Prämie: " +
     Math.floor(praemie * 100.0) / 100.0);
 }
}
```

Mehrere Programmläufe zeigen folgende Ergebnisse:

```
Geben Sie die Versicherungssumme ein:
5000
Prämie: 84.0
Prämie: 84.0
Geben Sie die Versicherungssumme ein:
5001
Prämie: 63.0126
Prämie: 63.01
Geben Sie die Versicherungssumme ein:
```

```
10001
Prämie: 84.00840000000001
Prämie: 84.0
```

Die Anweisung `fall = (int)((vSumme - 0.01) / 5000.0);` dient dazu, die Versicherungsumme auf die drei Fälle `0`, `1` und `default` abzubilden.

In der Anweisung

```
Math.floor(praemie * 100.0) / 100.0
```

wird die berechnete Prämie zunächst mit 100 multipliziert. Damit liegen Cent vor. Dann werden mithilfe der Operation `floor` die Stellen nach dem Komma bzw. dem Punkt abgeschnitten, d. h. `floor` rundet ab. Anschließend wird das Ergebnis durch 100 dividiert. Dadurch erhält man wieder Euro mit Cents.

1 Prüfen Sie die Wirkung der beiden beschriebenen Anweisungen für verschiedene Werte.
2 Schreiben Sie ein Programm, das für die Werte −10.0 bis +10.0 in Schritten von 0.02 den Wert selbst und den mit `floor` abgeschnittenen Wert ausgibt.

Berechnungszeit

Bei der Lösung mit der Mehrfachauswahl muss die Zeit, die zum Berechnen des `switch`-Ausdrucks benötigt wird, mit berücksichtigt werden.

Schachtelung

Eine Verbesserung der 1. Lösung ist durch ein Schachteln von `if`-Anweisungen möglich. Dabei ist es wichtig zu wissen, welche Abfragen mit welcher Häufigkeit auftreten.

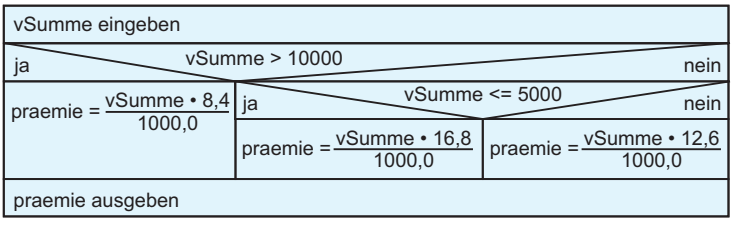


Abb. 4.11-2: Umsetzung einer Versicherungstabelle in geschachtelte `if`-Anweisungen.

Beispiel 1b

Ist beispielsweise in 70 Prozent aller Fälle die Versicherungssumme größer als 10 000,- Euro, dann ist die Lösung 2 (Abb. 4.11-2) die beste.

```
// Laufzeitoptimierte Berechnung von
// Versicherungsprämien
// Einsatz von geschachtelten if-Anweisungen

import inout.Console;
```

```
public class Versicherung2
{
 public static void main (String args[])
 {
  double vSumme;
  System.out.println
   ("Geben Sie die Versicherungssumme ein: ");
  vSumme = Console.readDoubleComma();
  double praemie = 0.0;
  // Lösung mit if - else - Konstrukt
  if(vSumme > 10000.0)
    praemie = vSumme * 8.4 / 1000.0;
   else
     if(vSumme <= 5000.0)
        praemie = vSumme * 16.8 / 1000.0;
     else
        praemie = vSumme * 12.6 / 1000.0;
  System.out.println("Prämie: " + praemie);
  //floor ist eine Methode der Klasse Math
  //floor ermittelt den größten double-Wert, der nicht größer
  //als das Argument (hier praemie) und gleich
  //einer ganzen Zahl ist
  System.out.println("Prämie: " +
      Math.floor(praemie * 100.0) / 100.0);
 }
}
```


Das Beispiel hat gezeigt, dass es in vielen Fällen sinnvoll ist, Auswahlanweisungen ineinander zu schachteln. In Abhängigkeit von den Annahmen und Voraussetzungen eines Problems spielt die richtige Wahl der Abfragereihenfolge eine große Rolle, um ein optimales Programm zu entwickeln.

Beispiel 1c

In 70 Prozent aller Fälle wird nur eine Bedingung ausgewertet, sonst zwei. Gegenüber der Lösung 1 ist die zusammengesetzte Bedingung entfallen.

4.12 Auswahl von Kontrollstrukturen *

Kontrollstrukturen sind so auszuwählen, dass sie die gegebene Problemstellung möglichst gut widerspiegeln. Wesentlich ist, dass klar zwischen einer **Auswahl** und einer **Wiederholung** unterschieden wird.

Auswahl

Eine Auswahl ist dadurch charakterisiert, dass entweder

- ein Fall in Abhängigkeit von einer Bedingung eintritt (einfache Auswahl) oder
- aus zwei Fällen bzw. Alternativen ein Fall ausgewählt werden muss (zweifache Auswahl) oder
- aus mehreren disjunkten Fällen ein Fall ausgewählt werden muss (Mehrfachauswahl).

Regeln

- Die Mehrfachauswahl ist immer dann zu verwenden, wenn es mehr als zwei disjunkte Alternativen gibt, aus denen genau eine zur Laufzeit auszuwählen ist.
- Die Mehrfachauswahl ist immer mit Fehlerausgang (`default`) zu verwenden, um übersehene Alternativen abfangen zu können.
- Wird in einer zweifachen Auswahl ein Zweig für die Fehlerbehandlung verwendet, dann sollte dies der `else`-Zweig sein.
- Bei geschachtelten Auswahlanweisungen sind wahrscheinliche Abfragehäufigkeiten zu berücksichtigen.

Wiederholung

Die Auswahlkriterien für die Wiederholungsanweisung zeigt die Abb. 4.12-1 (in Anlehnung an [MeMa86, S. 285]).

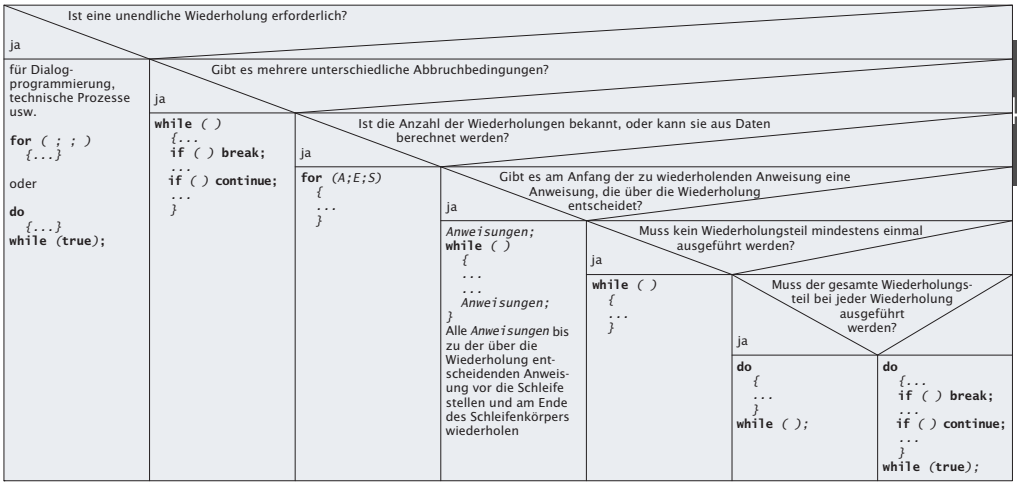


Abb. 4.12-1: Auswahlkriterien für die Wiederholungsanweisung.

Regeln

Folgende Regeln sollten bei Wiederholungsanweisungen beachtet werden:

- Endlos-Schleifen und n + ½ -Schleifen sind nur dann zu verwenden, wenn dies unbedingt erforderlich ist.
- Bei der `while`-Schleife muss die Bedingung am Anfang der Wiederholung bereits einen eindeutigen Wert besitzen.

4.13 Strukturierte Programmierung ***

Die strukturierte Programmierung i.e.S. erlaubt nur solche Kontrollstrukturen, die genau einen Ein- und einen Ausgang haben. Man nennt solche Kontrollstrukturen daher lineare Kontrollstrukturen. Es lassen sich vier verschiedene Arten unterscheiden: die Sequenz, die Auswahl (Fallunterscheidung, Verzweigung), die Wiederholung (Schleife) und der Aufruf. Alle Ar-

ten lassen sich beliebig miteinander kombinieren und ineinander schachteln. Sprünge (goto-Anweisungen), wie sie in manchen Programmiersprachen vorhanden sind, sind fehleranfällig und daher in der strukturierten Programmierung verboten. Gute grafische Darstellungen für lineare Kontrollstrukturen sind Struktogramme und Aktivitätsdiagramme der UML in Knoten-Notation. Die Java-Syntax ist *nicht* optimal und muss durch Kommentare ergänzt werden.

[BöJa66] haben nachgewiesen, dass alle Kontrollflüsse durch eine Auswahlkonstruktion und eine Wiederholungskonstruktion beschrieben werden können.

Die in Java verfügbaren Kontrollstrukturen haben ein gemeinsames Kennzeichen: Sie besitzen – bis auf die `break`- und `continue`-Anweisungen – jeweils genau einen Eingang und einen Ausgang.

Prinzip der Lokalität

Zwischen dem Eingang und dem Ausgang gilt das **Lokalitätsprinzip**, d. h. der Kontrollfluss verlässt den durch Eingang und Ausgang definierten Kontrollbereich *nicht*. Betrachtet man jede Kontrollstruktur makroskopisch, dann verläuft der Kontrollfluss linear durch ein Programm, d. h. streng sequenziell vom Anfang bis zum Ende. Daher bezeichnet man diese Kontrollstrukturen auch als »lineare Kontrollstrukturen«.

Termination

Zur Korrektheitsprüfung eines Programms gehört der Nachweis der Termination. Ein Programm terminiert, wenn es nach endlich vielen Schritten abgearbeitet ist. Sind die Abbruchbedingungen in Wiederholungen falsch gesetzt, dann kann ein Programm in eine »unendliche« Schleife geraten.

Bei ausschließlicher Anwendung der Java-Kontrollstrukturen müssen nur die *bedingten Wiederholungen* überprüft werden. Damit eine bedingte Wiederholung überhaupt enden kann, muss es innerhalb des Wiederholungsteils eine oder mehrere Anweisungen geben, die eine **Rückwirkung auf die Bedingung** haben (siehe »Termination von Schleifen«, S. 134).

Werden in einem Algorithmus oder in einem Programm nur lineare Kontrollstrukturen verwendet, dann spricht man auch von **Strukturiertem Programmieren im engeren Sinne**.

Dijkstra ([Dijk69], [Dijk72]) prägte den Begriff *Structured Programming* und subsumierte unter diesem Begriff verschiedene methodische Ansätze, die zur Verbesserung der Programmzuverlässigkeit beitragen sollten. Die Weiterentwicklung dieser Ansätze in verschiedenen Richtungen führte zu oft sehr weit auseinanderliegenden Definitionen des Begriffs »Strukturierte Programmierung«.

Sprung, goto

Das in älteren Programmiersprachen noch enthaltene Steuerkonstrukt »Sprung« bzw. »goto-Anweisung« verstößt gegen die oben aufgestellten Anforderungen:

- Eine Sprunganweisung verwischt völlig den semantischen Unterschied zwischen einer Auswahl und einer Wiederholung. In vielen Sprachen (Assembler, Basic) und grafischen Darstellungen, z. B. in **Programmablaufplänen**, wird eine Kombination von einfacher Auswahl und Sprung zur Darstellung einer bedingten Wiederholung benutzt. Eine einfache Auswahl hat aber semantisch nichts mit einer Wiederholung zu tun.
- Lokalität und Linearität sind *nicht* garantiert, da mit Sprüngen an beliebige Stellen des Programms und damit auch in andere Kontrollstrukturen hineingesprungen werden kann.
- Die Terminierung kann *nicht* oder nur mit großem Aufwand sichergestellt werden, da bei einem Sprung nicht klar ist, ob er dazu dient, Anweisungen zu wiederholen oder ob es sich um Sprünge zur Realisierung von Auswahlbedingungen handelt.
- Das strukturierte Denken in Sequenz, Auswahl und Wiederholung wird *nicht* unterstützt, da durch einen Sprung jederzeit die »Notbremse gezogen werden kann«, d. h. wenn man einen Algorithmus formuliert hat und am Ende merkt, dass das Problem einen Rücksprung an den Anfang erfordert, dann wird man nicht gezwungen, den Algorithmus mit einer Wiederholung neu zu strukturieren.
- Syntax und Semantik einer Sprunganweisung sind *nicht* selbsterklärend.
- Die Fehleranfälligkeit steigt bei der Verwendung der Sprunganweisung.

Eine eingeschränkte und damit disziplinierte Form des »Sprungs« stellen die `break`- und `continue`-Anweisungen dar. In Java gibt es *keine* Sprunganweisungen (`goto`).

Wertung

Für die Darstellungen von Kontrollstrukturen sind die grafischen und textuellen Beschreibungsmittel unterschiedlich geeignet:

Struktogramme

Struktogramme ermöglichen eine optimale grafische Darstellung von linearen Kontrollstrukturen, da es *nicht* möglich ist, Sprünge darzustellen. Der in Struktogrammen verfügbare Platz ermöglicht außerdem die Wahl aussagekräftiger Namen. Die Variablen können am Anfang in einem eigenen Rechteck beschrieben werden.

Das manuelle Zeichnen und Ändern ist aufwendig. Es gibt jedoch Struktogramm-Generatoren, die diese Arbeit automatisch erledigen. Besonders vorteilhaft bei Struktogrammen ist, dass die Auswahl in ablaufadäquater Form dargestellt wird, d. h. die Al-

ternativen werden horizontal angeordnet, während in textuellen Darstellungsformen eine vertikale Anordnung erfolgt.

Programmiersprachen

Die Syntax einiger **Programmiersprachen** ist so gestaltet, dass die Syntax die Semantik der linearen Kontrollstrukturen optimal unterstützt. Die Wortsymbole sind selbsterklärend, und jede Konstruktion wird durch ein spezifisches Wortsymbol explizit abgeschlossen (`loop ... end loop`; `if .. end if`). Die Schachtelungsstruktur von Kontrollstrukturen wird dadurch optisch hervorgehoben, dass eingeschachtelte Kontrollstrukturen textuell nach rechts eingerückt werden (manuell oder automatisch durch Formatierer).

Java

Die Programmiersprache Java ist in dieser Hinsicht *nicht* vorbildlich. Daher sind ein konsequentes Einrücken und zusätzliche Kommentare erforderlich, um gut lesbare Kontrollstrukturen zu erhalten.

Aktivitätsdiagramm, PAP

Als schwerwiegender Nachteil des **Aktivitätsdiagramms** der UML in Fluss-Notation – wenn es für Kontrollstrukturen verwendet wird – und des **Programmablaufplanes** (PAP) erweist sich, dass es für grundlegende Kontrollstrukturen *keine* eigenen Symbole gibt (Mehrfachauswahl, Wiederholungen). Beide bieten dem Programmierer zu große Freiheiten, sodass sie zur Beschreibung von linearen Kontrollstrukturen nur beschränkt geeignet sind. Schachtelungsstrukturen sind z. B. kaum erkennbar. Die Variablen-Deklaration kann in die Symbolik *nicht* integriert werden.

Die Knoten-Notation für die Aktivitätsdiagramme der UML ist dagegen ähnlich gut zur Darstellung von Kontrollstrukturen geeignet wie die Struktogramme.

Vorteile

Die ausschließliche Verwendung von **linearen Kontrollstrukturen** bringt folgende Vorteile:

+ Vereinheitlichung der Programmierstile, d. h. Standardisierung der Kontrollflüsse.
+ Übersichtliche, gut lesbare und verständliche Anweisungsteile von Programmen.
+ Leichte Überprüfbarkeit der Terminierung.
+ Für gleichartige Probleme entstehen gleichartige Kontrollfluss-Strukturen.
+ Die Auswirkungen jeder Kontrollstruktur sind übersehbar.

Methodik

Von der Methodik her ist folgende Reihenfolge einzuhalten:

1 Immer zuerst die Kontrollstrukturen entwerfen.
2 Erst dann die elementaren Anweisungen überlegen.

4.14 Behandlung von Ausnahmen *

Führt die Ausführung eines Programms zu einer Situation, in der eine reguläre Weiterarbeit *nicht* sinnvoll ist, z. B. bei einer Division durch Null, dann liegt eine Ausnahme-Situation *(exception)* vor. In vielen Programmiersprachen führt eine Ausnahmesituation zu einem Abbruch des laufenden Programms (das Programm »stürzt« ab). In Java können kritische Anweisungen in `try`-Blöcke eingeschlossen werden, die im Fehlerfall auf `catch`-Blöcke verzweigen, um eine Fehlerbehandlung durchzuführen. Sind Restarbeiten auszuführen, dann kann ein `finally`-Block angeschlossen werden.

Bei der Ausführung eines Programms kann es zu verschiedenen Fehlerarten kommen, die ein Weiterarbeiten unmöglich machen:

- Der Programmierer hat eine Fehlersituation übersehen, z. B. tritt bei einer mathematischen Berechnung eine Division durch Null auf.
- Der Benutzer des Programms gibt Werte ein, die nicht vorgesehen waren und nicht weiterverarbeitet werden können, z. B. wird ein Buchstabe statt einer Ziffer eingegeben.

Beide Fehlerarten können durch defensives Programmieren vermieden werden, d. h. jede mögliche Fehlersituation wird durch Abfragen versucht zu vermeiden.

Beispiel 1a

DemoAusnahme1

```
import inout.Console;
public class DemoAusnahme1
{
 public static void main (String args[])
 {
  int zahlEin, zahlAus;
  System.out.println("Geben Sie bitte eine ganze Zahl ein");
  zahlEin = Console.readInt();
  zahlAus = 100 / zahlEin;
  System.out.println
    ("ZahlEin: " + zahlEin + " 100/ZahlEin: " + zahlAus);
 }
}
```

Eine Eingabe des Wertes `0` führt zum »Absturz« des Programms:

```
C:\DemoAusnahme1>java DemoAusnahme1
Geben Sie bitte eine ganze Zahl ein
0
java.lang.ArithmeticException: / by zero
    at DemoAusnahme1.main(DemoAusnahme1.java:9)
```

Der »Absturz« eines Programms beim Endbenutzer muss auf jeden Fall vermieden werden. Defensive Programmierung führt zu folgendem Programm:

Beispiel 1b

DemoAusnahme2

```
import inout.Console;
public class DemoAusnahme2
{
 public static void main (String args[])
 {
  int zahlEin, zahlAus;
  System.out.println("Geben Sie bitte eine ganze Zahl ein");
  zahlEin = Console.readInt();
  if (zahlEin != 0)
   {
     zahlAus = 100 / zahlEin;
     System.out.println
      ("ZahlEin: " + zahlEin + " 100/ZahlEin: " + zahlAus);
   }
  else
   System.out.println("Keine 0 eingeben!");
 }
}
```


Probleme

In vielen Situationen ist eine solche einfache Lösung durch eine »Abfang-Abfrage« *nicht* möglich.

Beispiel

Soll verhindert werden, dass bei der Multiplikation von ganzen Zahlen der Zahlenbereich überschritten wird (siehe »Ganzzahlige Typen«, S. 64), wäre folgende Abfang-Abfrage nötig:

```
if (zahl1 * zahl2 <= groessteDarstellbareZahl)
```

Diese Abfrage führt aber genau zu dem Fehler, den man vermeiden möchte. Nur durch kompliziertere Abfragen wäre dieser Fehlerfall zu vermeiden.

Abfang-Abfragen haben außerdem den Nachteil, dass der Programmieraufwand steigt und das Programm unübersichtlicher wird, wenn man jeder möglichen Fehlerstelle eine Abfang-Abfrage voranstellt.

Lösung

Moderne Programmiersprachen wie Java stellen besondere Sprachkonstrukte zur Verfügung, um auf einfache und elegante Weise Fehler abzufangen und damit das defensive Programmieren zu erleichtern.

Java

Tritt in einem Java-Programm eine fehlerhafte Situation auf, dann wird eine sogenannte **Ausnahme** *(exception)* ausgelöst. Der Programmablauf verzweigt von der Stelle, an der das Ausnahmeereignis aufgetreten ist, zu einer vom Programmierer anzugebenden Stelle. Eine Ausnahme wird also an der Stelle des Auftretens ausgelöst *(thrown)* und an der Stelle abgefangen *(caught)*, an der der Programmablauf fortgeführt werden soll.

Vordefinierte Ausnahmen

In Java gibt es eine Reihe vordefinierter Ausnahmen, die in entsprechenden Situationen von der **JVM** ausgelöst werden.

Abfangen von Ausnahmen

Erwartet man, dass in einem Programmstück eine vordefinierte oder selbstdefinierte Ausnahme eintreten kann, dann schließt man dieses Programmstück in einen sogenannten try-Block ein.

Im Anschluss an den try-Block formuliert man eine oder mehrere Ausnahmebehandlungsroutinen *(exception handlers)*, die angeben, was beim Eintreten der entsprechenden Ausnahme geschehen soll.

Beispiel 1c

DemoAusnahme3

```
import inout.Console;
public class DemoAusnahme3
{
 public static void main (String args[])
 {
  int zahlEin, zahlAus;
  System.out.println("Geben Sie bitte eine ganze Zahl ein");
  zahlEin = Console.readInt();
  try //try-Block
  {
     zahlAus = 100 / zahlEin;
     System.out.println
      ("ZahlEin: " + zahlEin + " 100/ZahlEin: " + zahlAus);
  }
  catch (Exception e) //catch-Block
  {
   System.out.println("Keine 0 eingeben!");
  }
 }
}
```

Der Programmierer vermutet, dass in der Anweisung

```
zahlAus = 100 / zahlEin;
```

ein Fehler auftreten könnte und packt diese Anweisung daher in einen try-Block. *try* bedeutet übersetzt »etwas ausprobieren«. Für die **JVM** bedeutet der try-Block, dass bei allen Anweisungen im try-Block geprüft wird, ob ein Fehler auftritt. Tritt kein Fehler auf, dann werden die Anweisungen im try-Block nacheinander ausgeführt (Normalfall). Tritt ein Fehler auf, dann wird im catch-Block fortgefahren, d. h. der try-Block wird an der Stelle verlassen, an der der Fehler auftritt. *catch* bedeutet übersetzt »auffangen«, d. h. im catch-Block wird der Fehler aufgefangen und es wird eine vom Programmierer vorgegebene Fehlerbehandlung vorgenommen, hier die Ausgabe der Meldung: `"Keine 0 eingeben!"`

finally-Block

In manchen Fällen müssen sowohl im Standardfall als auch im Ausnahmefall noch Abschlussarbeiten durchgeführt werden, bevor das Programm beendet werden kann, z. B. das Schließen von Dateien. Daher ist es optional möglich, hinter den catch-Block noch einen finally-Block anzufügen, der auf jeden Fall durchlaufen wird.

Beispiel 1d

DemoAusnahme4

```
import inout.Console;
public class DemoAusnahme4
{
 public static void main (String args[])
 {
  int zahlEin, zahlAus;
  System.out.println("Geben Sie bitte eine ganze Zahl ein");
  zahlEin = Console.readInt();
  try //try-Block
  {
     zahlAus = 100 / zahlEin;
     System.out.println
      ("ZahlEin: " + zahlEin + " 100/ZahlEin: " + zahlAus);
  }
  catch (Exception e) //catch-Block
  {
   System.out.println("Keine 0 eingeben!");
  }
  finally
  {
   System.out.println("Das Programm ist zu Ende");
  }
 }
}
```

Zwei Programmläufe führen zu folgenden Ergebnissen:

```
Geben Sie bitte eine ganze Zahl ein
12
ZahlEin: 12 100/ZahlEin: 8
Das Programm ist zu Ende
Geben Sie bitte eine ganze Zahl ein
0
Keine 0 eingeben!
Das Programm ist zu Ende
```

Java-Syntax

Die Java-Syntax sieht folgendermaßen aus:

```
try
{
  Anweisung; Anweisung;  ...
}
catch (Exception e)
{
  Ausnahmebehandlung
}
finally
{
  Abschlussarbeiten
}
```

Kontrollfluss

Den Kontrollfluss im Normal- und im Fehlerfall zeigt die Abb. 4.14-1.

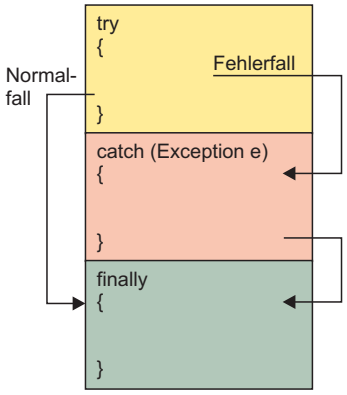


Abb. 4.14-1: Der Kontrollfluss im Normal- und im Fehlerfall bei einer `try-catch-finally`-Anweisung.

Hinter `catch` muss in Klammern ein sogenannter Ausnahmeparameter aufgeführt werden, der es ermöglicht, den Fehler genauer zu lokalisieren. Im einfachsten Fall reicht es `Exception e` anzugeben. Die vollständige Syntax zeigt die Abb. 4.14-2.

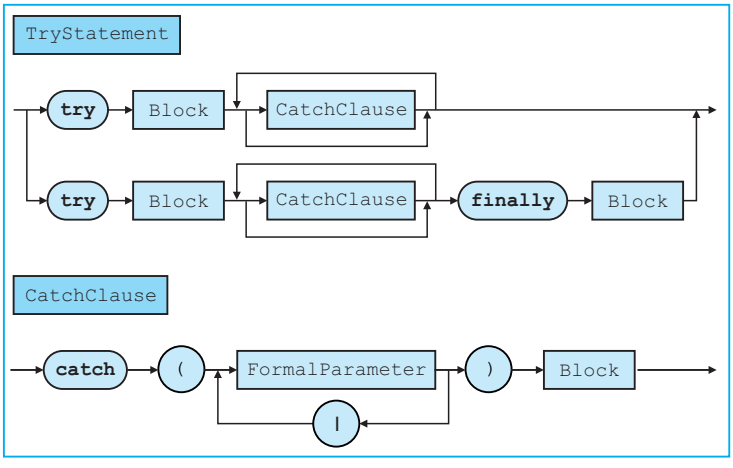


Abb. 4.14-2: Syntax von `try-catch` in Java.

Wie die Syntax zeigt, können auch mehrere `catch`-Blöcke hintereinander aufgeführt werden. Diese Möglichkeit wird benutzt, wenn auf verschiedene Fehlerarten unterschiedlich reagiert werden muss. Mehrere `catch`-Blöcke sollten dabei immer von den speziellen zu den allgemeinen Fehlerarten sortiert werden. Die `catch`-Blöcke werden von oben nach unten durchlaufen. Trifft ein `catch`-Block zu, dann werden die restlichen übersprungen.

Alles in try?

Es stellt sich die Frage, warum nicht alle Anweisungen vorsichtshalber in `try`-Blöcke eingeschlossen werden? Das ist eine Frage des Aufwands zur Laufzeit. Die JVM benötigt zusätzliche Zeit, um alle Fehlersituationen zu überprüfen. Daher sollten `try`-Blöcke auf die kritischen Anweisungen beschränkt werden.

4.15 Zusicherungen in Java **

Um Fehler beim Programmieren zu vermeiden, sollten an allen Programmstellen, wo Werte oder Kombinationen von Werten gelten müssen, sogenannte Zusicherungen *(assertions)* eingefügt werden, die dann automatisch während der Laufzeit ausgewertet werden. Wird die Zusicherung verletzt, dann wird eine Fehlermeldung ausgegeben und das Programm abgebrochen. In Java erfolgt eine Zusicherung in einer `assert`-Anweisung.

Defensiv programmieren

Je umfangreicher und komplexer Ihre Programme werden, desto mehr Fehler werden Ihre Programme enthalten. Daher ist es sehr wichtig, möglichst defensiv zu programmieren, d. h. potenzielle Fehlerquellen möglichst konstruktiv zu vermeiden. Eine mögliche Methode besteht darin, in den Programmcode sogenannte Zusicherungen einzufügen.

Zusicherungen

Zusicherungen *(assertions)* garantieren an bestimmten Stellen im Programm bestimmte Eigenschaften oder Zustände. Sie sind logische Aussagen über die Werte der Variablen an den Stellen im Programm, an denen die jeweiligen Zusicherungen stehen (siehe auch »Zusicherungen«, S. 301). Zusicherungen werden in Java durch eine `assert`-Anweisung spezifiziert, die folgende Syntax besitzt:

Syntax 1

`assert` *`Expression1`*
wobei *`Expression1`* ein boolscher Ausdruck ist. Zur Laufzeit wird dieser Ausdruck ausgewertet. Ist er falsch (`false`), dann wird die Ausnahme `AssertionError` ausgelöst. Es wird *keine* detaillierte Fehlermeldung ausgegeben.

Syntax 2

Optional kann noch ein weiterer Ausdruck angegeben werden:
`assert` *`Expression1`* `:` *`Expression2`*
wobei *`Expression2`* ein Ausdruck ist, der einen Wert besitzt. Dieser Wert wird zusätzlich ausgegeben, wenn die Zusicherung *nicht* zutrifft, und soll helfen, den Fehler schneller zu finden. Der Wert des Ausdrucks wird ausgegeben.

Beispiel

In einem Programm zur Prämienberechnung (siehe auch: »Die ein- und zweiseitige Auswahl«, S. 109) kann durch Zusicherungen sichergestellt werden, dass beim Programmieren keine Fehler in den Abfragen aufgetreten sind:

Praemie2

```
// Prämienberechnung in Abhängigkeit von den
// Dienstjahren und dem Alter mit Zusicherungen

import inout.Console;
public class Praemie2
{
 public static void main (String args[])
 {
  final int grundpraemie1 = 200, grundpraemie2 = 100,
  zulage1 = 20, zulage2 = 100; //Konstanten
  int dienstjahre, alter; //Eingabe
  int praemie = 0; //Ausgabe
  System.out.println("Dienstjahre?");
  dienstjahre = Console.readInt();
  System.out.println("Alter?");
  alter = Console.readInt();
  if (dienstjahre > 5)
    {
        assert(dienstjahre > 5):
        "Dienstjahre ist nicht > 5";
        praemie = grundpraemie2 + zulage1 * dienstjahre;
        if (alter > 45)
         {
            assert(dienstjahre > 5 && alter > 50):
            "Dienstjahre oder Alter falsch";
            praemie = praemie + zulage2;
         }
    }
  else //dienstjahre <= 5
    if (dienstjahre > 2)
    {
        assert(!(dienstjahre > 5)):
        "Dienstjahre ist nicht <= 5";
        praemie = grundpraemie1;
    }
  System.out.println("Dienstjahre:\t" + dienstjahre);
  System.out.println("Alter:\t\t" + alter);
  System.out.println("Prämie:\t\t" + praemie);
 }
}
```

Wenn Sie durch einen Tippfehler in der Zeile `if (Alter > 45)` geschrieben haben, statt `if (Alter > 50)`, dann erhalten Sie bei folgender Programmausführung den Fehlerhinweis, dass die Zusicherung `assert(Dienstjahre > 5 && Alter > 50) : "Dienstjahre oder Alter falsch";` verletzt wurde:

```
Dienstjahre?
8
Alter?
46
Exception in thread "main" java.lang.AssertionError:
```

```
Dienstjahre oder Alter falsch
        at Praemie2.main(Praemie2.java:25)
```

Hinweis

Die Auswertung von Zusicherungen muss eingeschaltet werden: Standardmäßig werden `assert`-Anweisungen in Java-Programmen *nicht* ausgeführt. Beim Programmstart muss daher der Schalter *(switch)* `-ea` (für *enableassertions*) gesetzt werden, z. B. `java -ea Programmname` (beim Start über das Konsolenfenster).

Warum müssen `assert`-Anweisungen an- und abschaltbar sein?

4.16 Vom Problem zur Lösung: Teil 2 **

Werden für die Problemlösung von Programmieraufgaben **Kontrollstrukturen** benötigt, dann ist der Problemlöseraum größer und es gibt in der Regel mehrere Lösungswege.

Es sollte folgende **Entscheidungsreihenfolge** eingehalten werden:

1 Zuerst die Kontrollstrukturen ermitteln.

a Zuerst Wiederholungen identifizieren.
b Dann Auswahlanweisungen programmieren.

2 Dann die einfachen Anweisungen erstellen (siehe »Vom Problem zur Lösung: Teil 1«, S. 95).

Im Vorfeld sollte für die Problemlösung Folgendes geklärt werden:

- Wie allgemein soll die Lösung sein?
- Wie leicht erweiterbar soll die Lösung sein?

Beispiel 1

Sie sollen folgendes Problem lösen:
Für ein Reisevergleichsportal sollen die Reiserücktrittsversicherungen von 3 Versicherungsunternehmen verglichen werden:

- Versicherung 1: Prämie = 50 € + 4 % vom Reisepreis
- Versicherung 2: Prämie = 5 % vom Reisepreis
- Versicherung 3: Prämie = 100 € + 3,5 % vom Reisepreis

Es sollen die Prämien für Reisepreise von 1000.- € bis 15.000 € in 200er-Schritten berechnet werden.

Für jeden Reisepreis soll angegeben werden, welche Versicherung am günstigsten ist. Zusätzlich ist der Abstand der günstigsten Prämie zur zweitgünstigsten auszugeben. Die Prämien sind auf ganze Zahlbeträge abzurunden.

Außerdem ist anzugeben, in wie vielen Fällen welche Versicherung am günstigsten ist. Die gewünschte Problemlösung soll *nicht* verallgemeinerbar und *nicht* erweiterbar sein.

Versuchen Sie *nicht* alle vorgegebenen Probleme auf einmal zu lösen, sondern konzentrieren Sie sich zunächst auf ein Problem. Entsprechend der vorgeschlagenen Entscheidungsreihenfolge sollten Sie sich zunächst auf die Wiederholungs-Kontrollstrukturen konzentrieren.

Mögliche Lösungsschritte sind:

Semiformale Lösung 1. Schritt

Es ist eine Tabelle auszugeben mit festgelegtem Anfangs- und Endwert sowie einer Schrittweite. Die passende Wiederholungstruktur dafür ist die Zählschleife.

Formale Java-Lösung

```
int startwert = 1000, endwert = 15000,
   schrittweite = 200;
for (int reisepreis = startwert; reisepreis <= endwert;
   reisepreis += schrittweite)
{
  //...
  System.out.println(reisepreis);
}
```

Semiformale Lösung 2. Schritt

Innerhalb der Schleife müssen die Prämien der verschiedenen Versicherungen berechnet und in der Tabelle ausgegeben werden (einfache Anweisungen).

Formale Java-Lösung

```
//Vor der Schleife
double praemie1, praemie2, praemie3;
for ..
{
  praemie1 = 50 + reisepreis * 0.04;
  praemie2 = reisepreis * 0.05;
  praemie3 = 100 + reisepreis * 0.035;
  //...
  System.out.println(reisepreis + "\t"
      + (int)praemie1 + "\t"
      + (int)praemie2 + "\t" + (int)praemie3);
}
```

Semiformale Lösung 3. Schritt

Die Ermittlung der jeweils günstigsten Versicherung bedeutet, aus den drei berechneten Prämien die billigste Prämie zu ermitteln. Dazu müssen die drei Prämien miteinander verglichen werden. Eine mögliche Lösungsidee besteht darin, zunächst die `praemie1` in einer Variablen `min` zu speichern und anzunehmen sie sei die billigste Prämie. Anschließend wird

geprüft, ob die `praemie2` noch billiger als `min` ist, wenn ja, dann erhält `min` diesen Wert. Anschließend wird `min` mit `praemie3` verglichen.

Formale Java-Lösung

```
for ...
{
  //...
  double min = praemie1;
  if (praemie2 < min) min = praemie2;
  if (praemie3 < min) min = praemie3;
  //....
}
```

4. Schritt

Semiformale Lösung

In der Tabelle muss noch ausgegeben werden, von welcher Versicherung die billigste Prämie stammt. Dazu ist es notwendig sich zu merken, zu welcher Versicherung die billigste Prämie gehört. Am besten wird sich dies bei der Ermittlung der billigsten Prämie gemerkt.

Formale Java-Lösung

```
for ...
{
  //...
 String versicherung = "Versich. 1";
 if (praemie2 < min)
 {
      min = praemie2;
      versicherung = "Versich. 2";
  }
  if (praemie3 < min)
  {
       min = praemie3;
       versicherung = "Versich. 3";
  }
  //...
  System.out.println(reisepreis + "\t"
    + (int)praemie1 + "\t" + (int)praemie2 + "\t"
    + (int)praemie3 + "\t" + Versicherung);
}
```

5. Schritt

Semiformale Lösung

Um festzustellen, wie häufig welche Versicherung die billigste Prämie anbietet, kann auf die Ermittlung des `min`-Wertes aufgesetzt werden. Die Zuordnung zur jeweiligen Versicherung durch die Variable `versicherung` kann innerhalb einer `switch`-Anweisung zur Zählung verwendet werden.

Formale Java-Lösung

```
for ...
{
  //...
```

```
  switch (versicherung)
  {
     case "Versich. 1": summe1++; break;
     case "Versich. 2": summe2++; break;
     case "Versich. 3": summe3++; break;
     default: System.out.println("Fehler");
    }
  //...
}
//außerhalb der for-Schleife
System.out.println("Am günstigsten:");
System.out.println("Versicherung 1: " + summe1 + " mal ");
System.out.println("Versicherung 2: " + summe2 + " mal ");
System.out.println("Versicherung 3: " + summe3 + " mal ");
```

6. Schritt

Semiformale Lösung

Um den Abstand der günstigsten Prämie zur zweitgünstigsten zu ermitteln, muss die Differenz zwischen `min` und den jeweils anderen Prämien ermittelt werden. Eine Möglichkeit besteht darin, diese Abfragen innerhalb der `switch`-Anweisung durchzuführen. Hinweis: Die Abfragen können mit dem ?:-Operator noch kompakter formuliert werden, z. B.

```
delta = praemie2 > praemie3? praemie3 - min : praemie2 - min;
```

Formale Java-Lösung

```
double delta = 0.0;
for ...
{
  //...
  switch (versicherung)
  {
     case "Versich. 1": summe1++;
          if (praemie2 > praemie3)
             delta = praemie3 - min;
          else
              delta = praemie2 - min;
          break;
      case "Versich. 2": summe2++;
            if (praemie3 > praemie1)
               delta = praemie1 - min;
            else
                delta = praemie3 - min;
             break;
      case "Versich. 3": summe3++;
              if (praemie1 > praemie2)
                 delta = praemie2 - min;
              else
                  delta = praemie1 - min;
               break;
       default: System.out.println("Fehler");
  }
  System.out.println(reisepreis + "\t"
     + (int)praemie1 + "\t" + (int)praemie2 + "\t"
     + (int)praemie3 + "\t" + versicherung + "\t"
```

```
     + (int)(delta + 0.5d));
}
```

Pro Schritt haben Sie bereits eine jeweils lauffähige Programmversion, so dass mögliche Fehler pro Schritt bereits erkannt werden können.

Das gesamte Programm sieht wie folgt aus:

```
/**
 * Vergleich von Reiserücktrittsversicherungen
 *
 * @author Helmut Balzert
 * @version V0.1
 */

public class Reiseruecktritt
{
   public static void main(String args[])
    {
        double praemie1, praemie2, praemie3;
        int summe1 = 0, summe2 = 0, summe3 = 0;
        //Anzahl am günstigsten
        int startwert = 1000, endwert = 15000,
          schrittweite = 200;
        System.out.println
        ("Preis\tVer1\tVers2\tVers3\tMinimum\t\tDelta Min.");
        for (int reisepreis = startwert; reisepreis <= endwert;
          reisepreis += schrittweite)
        {
            praemie1 = 50 + reisepreis * 0.04;
            praemie2 = reisepreis * 0.05;
            praemie3 = 100 + reisepreis * 0.035;
            double min = praemie1, delta = 0.0;
            String versicherung = "Versich. 1";
            if (praemie2 < min)
            {
               min = praemie2;
               versicherung = "Versich. 2";
            }
            if (praemie3 < min)
            {
               min = praemie3;
               versicherung = "Versich. 3";
            }
            switch (versicherung)
            {
                case "Versich. 1": summe1++;
                     if (praemie2 > praemie3)
                        delta = praemie3 - min;
                     else
                        delta = praemie2 - min;
                     break;
                case "Versich. 2": summe2++;
                     if (praemie3 > praemie1)
                        delta = praemie1 - min;
```

```
                    else
                       delta = praemie3 - min;
                    break;
                case "Versich. 3": summe3++;
                    if (praemie1 > praemie2)
                       delta = praemie2 - min;
                    else
                       delta = praemie1 - min;
                    break;
                default: System.out.println("Fehler");
            }
            System.out.println(reisepreis + "\t"
              + (int)praemie1 + "\t" + (int)praemie2 + "\t"
              + (int)praemie3 + "\t" + versicherung + "\t"
              + (int)(delta + 0.5d));
        }
        System.out.println("Am günstigsten:");
        System.out.println("Versicherung 1: "+summe1+" mal ");
        System.out.println("Versicherung 2: "+summe2+" mal ");
        System.out.println("Versicherung 3: "+summe3+" mal ");
    }
}
```

Ein Ausschnitt aus dem Ergebnis sieht wie folgt aus:

```
Preis    Ver1   Vers2   Vers3   Minimum     Delta Min.
1000    90      50      135     Versich. 2  40
1200    98      60      142     Versich. 2  38
1400    106     70      149     Versich. 2  36
1600    114     80      156     Versich. 2  34
1800    122     90      163     Versich. 2  32
2000    130     100     170     Versich. 2  30
...
8800    402     440     408     Versich. 1  6
9000    410     450     415     Versich. 1  5
9200    418     460     422     Versich. 1  4
9400    426     470     429     Versich. 1  3
9600    434     480     436     Versich. 1  2
9800    442     490     443     Versich. 1  1
10000   450     500     450     Versich. 1  0
10200   458     510     457     Versich. 3  1
10400   466     520     464     Versich. 3  2
10600   474     530     471     Versich. 3  3
10800   482     540     478     Versich. 3  4
...
14400   626     720     604     Versich. 3  22
14600   634     730     611     Versich. 3  23
14800   642     740     618     Versich. 3  24
15000   650     750     625     Versich. 3  25
Am günstigsten:
Versicherung 1: 26 mal
Versicherung 2: 20 mal
Versicherung 3: 25 mal
```

Die schrittweise Entwicklung des Beispiels kann mit einem Hausbau verglichen werden. Zunächst wird der Rohbau erstellt, hier die for-Schleife. Anschließend werden die einzelnen Wände und das Dach fertiggestellt sowie Sonderwünsche eingebaut.

Hinweis

Das Programm kann auch mit einem Feld realisiert werden. Felder werden jedoch erst im Kapitel »Felder«, S. 173, behandelt.

4.17 Box: Kreuzworträtsel 2 **

Lösen Sie das Kreuzworträtsel (Abb. 4.17-1). Die Musterlösung dazu finden Sie im Anhang.

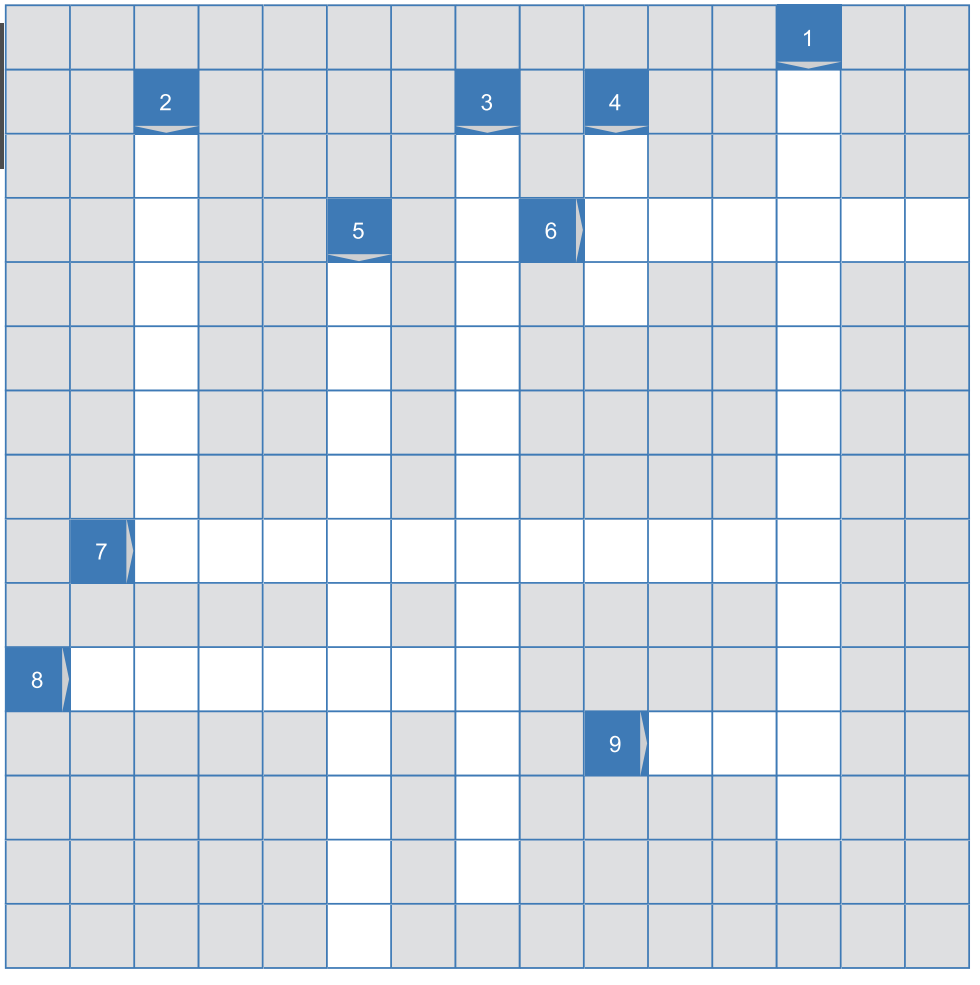


Abb. 4.17-1: Kreuzworträtsel zu Kontrollstrukturen.

Gesuchte Wörter:

1 Grafische Darstellung linearer Kontrollstrukturen.
2 Mehrere Anweisungen werden hintereinander, von links nach rechts und von oben nach unten, ausgeführt.
3 Mehrfache Ausführung von Anweisungen in Abhängigkeit von einer Bedingung oder für eine gegebene Anzahl.
4 Eine in DIN 66001 genormte, grafische Darstellungsform für Kontrollstrukturen von Algorithmen (Kurzform).
5 Programm, das nach endlicher Zeit beendet ist (Substantiv).
6 Wechsel der Kontrolle von der aufrufenden Stelle zu dem aufgerufenen Algorithmus.
7 Eigenschaft oder Zustand, der an einer bestimmten Stelle eines Programms immer gilt.
8 Ausführung von Anweisungen in Abhängigkeit von Bedingungen.
9 Interpreter, der den Java-Bytecode analysiert und ausführt (Kurzform).

5 Felder *

Problem

Variablen, die als Wertebereich einen einfachen Typ haben, z. B. `int`, können zu einem Zeitpunkt nur einen Wert von diesem Typ speichern. Erfordert eine Problemstellung jedoch die Speicherung von vielen, u. U. von Tausenden von Werten, dann müssen entsprechend viele Variablen deklariert werden, z. B. tausend Variablen vom Typ `int`. Das ist aufwendig und ab einer bestimmten Anzahl von Variablen praktisch *nicht* mehr durchführbar.

Felder

In fast allen Programmiersprachen gibt es daher das **Konzept des Feldes** *(array)*, das es ermöglicht, beliebig viele Variablen eines Typs zu einer Einheit zusammenzufassen und nur einmal zu deklarieren. Werden Variablen in einer bestimmten Art und Weise zu einer Einheit zusammengefasst, dann spricht man von einer **Datenstruktur**. Auf eine Datenstruktur wird in einer definierten Weise zugegriffen. Felder bilden eine bestimmte Art von Datenstruktur.

Eindimensionale Felder sind die einfachste Form von Feldern:

- »Eindimensionale Felder«, S. 174

In der Fallstudie OptiTravel werden eindimensionale Felder zur Berechnung und Darstellung von Balkendiagrammen benutzt:

- »OptiTravel: Balkendiagramm«, S. 179

Neben eindimensionalen Feldern können auch mehrdimensionale Felder programmiert werden:

- »Mehrdimensionale Felder«, S. 180

Manche Problemstellungen erfordern auch asymmetrische Felder:

- »Sonderformen von Feldern«, S. 187

In der Fallstudie OptiTravel werden Dreieckstabellen zur Darstellung von Entfernungstabellen verwendet:

- »OptiTravel: Tabellen«, S. 189

Für das Sortieren von Daten werden ebenfalls Felder benötigt:

- »Einfaches Sortieren«, S. 195

Sollen alle Elemente eines Feldes durchlaufen werden, dann kann dafür eine erweiterte `for`-Schleife in Java verwendet werden:

- »Iteration über Felder: Die erweiterte for-Schleife«, S. 198

In Java können eigene Aufzählungstypen definiert werden:

- »Aufzählungen mit enum«, S. 200

5.1 Eindimensionale Felder *

Inhaltlich zusammengehörende Daten, die alle vom gleichen Typ sind, werden als Feld *(array)* vereinbart. In Java werden eindimensionale Felder durch ein eckiges Klammerpaar vor oder nach dem Feldnamen deklariert. Im Anschluss daran kann die Feldgröße durch die Angabe der Elementanzahl (direkt) oder durch Angabe der Werte (indirekt) festgelegt werden. Der Zugriff auf ein Feld erfolgt über ganzzahlige Indizes. In Java wird jedes Feld ab 0 gezählt! Die Länge eines Feldes kann durch `length` abgefragt werden.

In der Praxis tritt oft das Problem auf, dass viele Variablen vom gleichen Typ vereinbart werden müssen, z. B. `int zaehler1, zaehler2, ....` Sicherlich ist es noch möglich, 10 oder 20 solcher Variablen zu deklarieren. Bei 1000 oder 10 000 ist dies aber *nicht* mehr möglich.

Auf der anderen Seite kommt der Zusammenhang zwischen den verschiedenen Variablen – außer bei einer geeigneten Namenswahl – durch die getrennte Deklaration *nicht* geeignet zum Ausdruck.

Feld

Aus diesen Gründen enthalten fast alle Programmiersprachen einen Konstruktionsmechanismus, der es ermöglicht, typidentische Variablen zu einer Einheit zusammenzufassen. Einen solchen Typkonstruktor bezeichnet man als **Feld** *(array)* – auch **Reihe** oder **Reihung** genannt. Felder stellen eine einfache Möglichkeit dar, typgleiche Variablen zu speichern.

Beispiel

Ein Feld besteht aus Elementen bzw. Komponenten, die alle den gleichen Typ haben. Auf die einzelnen Elemente wird über Indizes zugegriffen (Abb. 5.1-1).

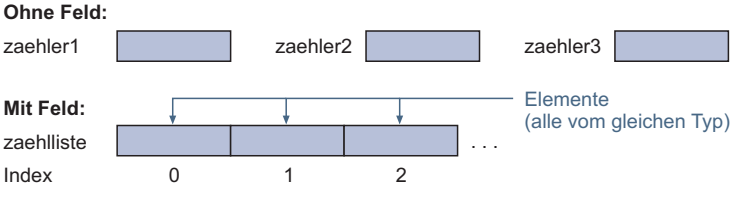


Abb. 5.1-1: Unterschied zwischen einzelnen Variablen und einem Feld von Variablen.

3 Schritte

Um ein Feld in Java zu erzeugen und zu benutzen, sind drei Schritte nötig:

1. Deklaration einer Feld-Variablen.
2. Festlegen der Feldgröße.
3. Schreibender und lesender Zugriff auf die Feldelemente.

Beispiel

In einem großen Verein finden Wahlen zum Präsidium statt. Das Präsidium umfasst 15 Mitglieder. Zur Wahl stellen sich 40 Vereinsmitglieder. Die Auszählung soll durch ein Programm unterstützt werden. Jedem Kandidaten ist eine Nummer zugeordnet. Ein Wahlhelfer tippt bei der Auszählung die jeweilige Nummer des Kandidaten ein, der auf dem Wahlzettel angekreuzt ist.

1. Deklaration einer Feldvariablen

Statt 40 Variablen vom Typ `int` zu deklarieren, wird ein Feld `zaehlliste` deklariert. Dass es sich um ein Feld handelt, wird in Java durch ein eckiges Klammernpaar hinter der Typangabe oder alternativ hinter dem Variablennamen angegeben:
`int[] zaehlliste;` oder `int zaehlliste[];`

2. Festlegen der Feldgröße

Die Anzahl der Elemente muss festgelegt werden, wobei die Zählung bei 0 beginnt. Die entsprechende Anweisung lautet (zusätzliches Element 0 für ungültige Stimmen):
`zaehlliste = new int[41];`

Nach dem Zuweisungszeichen steht das Schlüsselwort `new`, dann folgt eine Wiederholung des Feldtyps. In eckigen Klammern wird die Anzahl der Feldelemente angegeben. Alle Elemente des Feldes werden automatisch in Abhängigkeit vom Typ initialisiert. Numerische Felder werden mit Null (0) vorbesetzt.

3. Zugriff auf die Feldelemente

Der Zugriff auf die Feldelemente geschieht über einen ganzzahligen Index, der von `0` bis `n - 1` läuft, wobei `n` die Länge des Feldes bzw. die Anzahl der Feldelemente angibt. Auf das 4. Feldelement wird beispielsweise wie folgt zugegriffen:
`element4 = zaehlliste[3]; //Zählung beginnt bei 0!`

Auf einzelne Elemente wird immer über den Index zugegriffen, der in eckigen Klammern steht.

`Feldname.length`

Die Konstante `length`, die es für jedes Feld gibt, enthält die Obergrenze des Feldes.

Stimmenzaehlen

```
// Anzahl der Wahlstimmen pro Kandidat zählen
// Annahme: Kandidaten von 1 bis n durchnummeriert
// zaehlliste[0] dient zum Zählen ungültiger Stimmen
// Abbruch bei Eingabe einer Zahl >= 100

import inout.Console;

public class Stimmenzaehlen
{
 public static void main (String args[])
 {
  int[] zaehlliste; //Deklaration eines Feldes
  //Festlegung der Länge auf 41 Elemente 0 bis 40)
  zaehlliste = new int[41];
```

```
  int kandidatNr;
  int gueltigeStimmen = 0;

  do
  {
   System.out.println
    ("Bitte Nummer des Kandidaten eingeben");
   System.out.println("Ende: Nummer >= 100");
   kandidatNr = Console.readInt();
   if (kandidatNr >= 100) break;
   if (kandidatNr >= 1 && kandidatNr <= 40)
     //Zähler um  1 erhöhen
     zaehlliste[kandidatNr] = zaehlliste[kandidatNr] + 1;
   else //kandidatNr < 1 oder kandidatNr > 40
     zaehlliste[0] = zaehlliste[0] + 1; //ungültige Stimme
  } while (true);

  System.out.println("+++Stimmenzählliste+++");
  for (int i = 1; i <= zaehlliste.length -1; i++)
   {
       System.out.println("Kandidat mit der Nummer " + i +
       " erhielt " + zaehlliste[i] + " Stimme(n)");
       gueltigeStimmen = gueltigeStimmen + zaehlliste[i];
   }

  System.out.println("Ungültige Stimmen: " + zaehlliste[0]);
  System.out.println("Gültige Stimmen: " + gueltigeStimmen);
 }
}
```

Der Progammlauf sieht wie folgt aus (Ausschnitt):

```
Bitte Nummer des Kandidaten eingeben
Ende: Nummer >= 100
39
Bitte Nummer des Kandidaten eingeben
Ende: Nummer >= 100
34
Bitte Nummer des Kandidaten eingeben
Ende: Nummer >= 100
60
Bitte Nummer des Kandidaten eingeben
Ende: Nummer >= 100
39
Bitte Nummer des Kandidaten eingeben
Ende: Nummer >= 100
...
Kandidat mit der Nummer 34 erhielt 1 Stimme(n)
Kandidat mit der Nummer 35 erhielt 0 Stimme(n)
Kandidat mit der Nummer 36 erhielt 0 Stimme(n)
Kandidat mit der Nummer 37 erhielt 0 Stimme(n)
Kandidat mit der Nummer 38 erhielt 0 Stimme(n)
Kandidat mit der Nummer 39 erhielt 2 Stimme(n)
Kandidat mit der Nummer 40 erhielt 0 Stimme(n)
Ungültige Stimmen: 1
Gültige Stimmen: 3
```

Zeichnen Sie ein Struktogramm des Programms `Stimmenzaehlen`.

Obwohl Felder in Java automatisch initialisiert werden, sollten bei einem defensiven Programmierstil die Felder explizit vom Programmierer initialisiert werden. Es kann ja einen Java-Compiler geben, der die Initialisierung nicht automatisch vornimmt. Oder man konvertiert ein Java-Programm in ein C++-Programm und vergisst dann nachträglich eine Initialisierung hinzuzufügen.

Die Schritte 1 (Deklaration einer Feld-Variablen) und 2 (Festlegen der Feldgröße) können zu einem Schritt zusammengefasst werden.

Beispiel

Statt

```
int[] zaehlliste; //Deklaration eines Feldes
//Festlegung der Länge auf 41 Elemente 0 bis 40)
zaehlliste = new int[41];
```

kann geschrieben werden:

```
int[] zaehlliste = new int[41];
```

Feldgröße durch Initialisierung

Muss ein Feld mit individuellen Werten initialisiert werden und ist die Feldgröße überschaubar, dann kann anstelle der Feldgrößenangabe direkt die Initialisierung erfolgen. Die Initialisierung erfolgt durch eine in geschweifte Klammern eingeschlossene Initialisierungsliste, wobei die einzelnen Werte durch Kommata getrennt werden.

Beispiel

Für die Steuerung der Zimmertemperaturen in einem Wohnhaus werden die Solltemperaturen pro Zimmer vorgegeben.

```
int[] zimmertemp = {24, 18, 12, 22, 18, 16};
```

Es wird ein Feld mit 6 Elementen angelegt (von 0 bis 5) und mit den angegebenen Werten vorbelegt.

`char[]`

Es können auch Felder vom Typ `char` angelegt werden.

Beispiel

Es soll festgestellt werden, ob ein Wort, ein Satz oder eine Ziffernfolge vorwärts und rückwärts gelesen dasselbe ergibt (Palindrom). Zwischenräume in Sätzen sollen nicht berücksichtigt werden. Die Texte werden in Kleinbuchstaben eingegeben. Beispiele für Palindrome sind:

- Retter
- Reliefpfeiler
- Anna hetzte Hanna
- O Genie der Herr ehre dein Ego

- ○ Able was I ere I saw Elba
  (Napoleon, als er ins Exil nach Elba ging)
- ○ 123321
- ○ 975313579

Palindrom

```
// Feststellen , ob eine Zeichenkette vorwärts
// und rückwärts gelesen identisch ist  (Palindrom)

import inout.Console;

public class Palindrom
{
 public static void main (String args[])
 {
  char[] text; //Deklaration eines Feldes
  System.out.println
   ("Bitte Text eingeben (in Kleinbuchstaben):");

  text = Console.readCharArray();
  int laenge = text.length;
  int posAuf = 0, posAb = laenge-1;
  String merke = "ein Palindrom!";
  //Vergleich von Zeichen auf entgegengesetzten Positionen
  while (posAuf < posAb)
  {
   if (text[posAuf] == ' ') posAuf++; //Leerzeichen überlesen
   if (text[posAb] == ' ') posAb--; //Leerzeichen überlesen
   if (text[posAuf] != text[posAb])
    {
        merke = "kein Palindrom";
        break;
    }
    posAuf++;
    posAb--;
  }
  System.out.println("Der eingelesene Text: ");
  for (int i=0; i <= text.length-1; i++)
    System.out.print(text[i]);
  System.out.println("\nist " + merke);
 }
}
```

Mehrere Programmläufe führen zu folgenden Ergebnissen:

```
Bitte Text eingeben (in Kleinbuchstaben):
o genie der herr ehre dein ego
Der eingelesene Text:
o genie der herr ehre dein ego
ist ein Palindrom!
Bitte Text eingeben (in Kleinbuchstaben):
j a v a
Der eingelesene Text:  j a v a
ist kein Palindrom
```

1 Zeichnen Sie ein Struktogramm des Programms `Palindrom`.
2 Prüfen Sie was passiert, wenn zwei Leerzeichen aufeinander folgen.
3 Ändern Sie das Programm so, dass beliebig viele Leerzeichen zwischen Worten das Ergebnis nicht verfälschen.

5.2 OptiTravel: Balkendiagramm *

Damit der Reisende sich einen grafischen Überblick über die Zeitunterschiede der verschiedenen Transportmittel machen kann, soll Frau Jung als Junior-Programmiererin ein Programm schreiben, das folgende Anforderungen erfüllt:

/1/ Es sollen horizontale Balkendiagramme als Strichgrafik in einem Konsolenfenster ausgegeben werden.
/2/ Die maximale Breite des zur Verfügung stehenden Ausgabebereichs ist vorgegeben.
/3/ Pro Zeile sind der absolute Wert, der relative Wert in Prozent und anschließend die Strichgrafik auszugeben.

Verall-gemeinerung

Frau Jung entschließt sich dazu, das Programm von vornherein etwas allgemeiner zu konzipieren, um es später auch für andere Anwendungen einsetzen zu können. Sie legt die Werte und die zugehörigen Bezeichnungen in einem Feld ab. Die maximale Ausgabebreite legt sie in einer Konstanten ab, sodass spätere Änderungen nur hier vorgenommen werden müssen.

Balken diagramm

```
// Berechnung eines horizontalen Balkendiagramms
// für eine Konsolenausgabe

public class Balkendiagramm
{
 public static void main (String args[])
 {
  int[] werte = {435,320,125};
  String[] name = {"Auto", "Bahn", "Flug"};
  //Summe über alle Werte
  int summe = 0, anteil = 0, anteilMax = 0;
  final int breiteMax = 40;
  double faktor = 0.0;
  for  (int i = 0; i < werte.length; i++)
    summe = summe + werte[i];
  //max. Anteil berechnen
  //max. Balken darf nicht über Anzeigebereich
  //hinausgehen
  for  (int i = 0; i < werte.length; i++)
  {
    anteil = (100 * werte[i]) / summe; //wird abgeschnitten
    if (anteil > anteilMax)
        anteilMax = anteil;
  }
  //Normieren auf max. Breite des Anzeigebereichs
  if (anteilMax > breiteMax)
```

```
    faktor = (double)breiteMax / (double)anteilMax;
  //Ausgabe der Werte mit Strichgrafik
  System.out.println();
  for  (int i = 0; i < werte.length; i++)
   {
     anteil = (100 * werte[i]) / summe; //wird abgeschnitten
     System.out.print(name[i] + ": " + werte[i] + " = "
        + anteil + "%\t|");
     if (faktor > 0.0)
        anteil = (int)(anteil * faktor);
     for (int j = 1; j < anteil; j++)
       System.out.print("+");
     System.out.println();
   }
  System.out.println("Summe aller Werte: " + summe);
  System.out.println("Normierungsfaktor: " + faktor);
 }
}
```

Der Programmlauf führt zu folgendem Ergebnis:

```
Auto: 435 = 49%    |+++++++++++++++++++++++++++++++++++++++
Bahn: 320 = 36%    |++++++++++++++++++++++++++++
Flug: 125 = 14%    |++++++++++
Summe aller Werte: 880
Normierungsfaktor: 0.8163265306122449
```

Zeichnen Sie ein Struktogramm des Programms `Balkendiagramm`.

5.3 Mehrdimensionale Felder *

In Java werden mehrdimensionale Felder *(arrays)* durch geschachtelte (eindimensionale) Felder realisiert. Pro Dimension bzw. pro Schachtelungsebene wird ein eckiges Klammerpaar angegeben, z. B. `int[][][] tab3 = new int [100][20][5];`. Die Feldlänge pro Dimension wird mit der Konstanten `length` abgefragt, z. B. Länge der 2. Dimension durch `tab3[0].length`. Die typische Kontrollstruktur zum Durchlaufen von mehrdimensionalen Feldern sind ineinandergeschachtelte `for`-Schleifen.

In der Praxis reichen eindimensionale Felder *nicht* aus. Oft müssen Werte in zwei- oder mehrdimensionalen Tabellen gespeichert werden. Für viele mathematische Probleme werden zwei- oder mehrdimensionale Matrizen benötigt.

Syntax

In Java wird für jede gewünschte Dimension ein eckiges Klammerpaar in der Feldvereinbarung angegeben, wobei die Klammerpaare vor oder hinter dem Feldnamen stehen können. Die Angabe der Feldgröße kann direkt hinter der Vereinbarung oder als gesonderte Anweisung erfolgen, wobei pro Dimension die Größe bzw. die Elementanzahl anzugeben ist. Bei kleinen Feldern kann auch eine Festlegung der Elementanzahl durch Angabe der Initialisierungswerte erfolgen.

Beispiel

```
//zweidimensionale Tabelle
double[][] tab2; //Vereinbarung
tab2 = new double[3][4];
```

alternativ:

```
double[][] tab2 = new double[3][4];
```

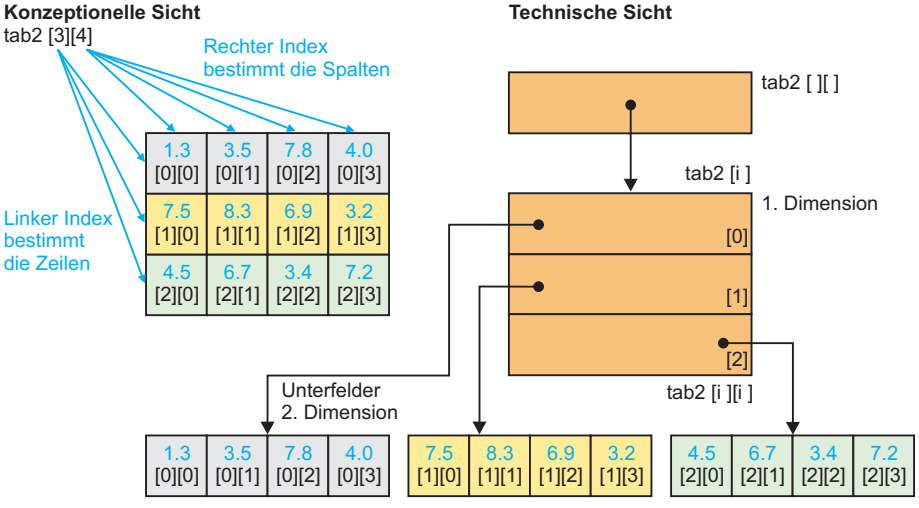
alternativ:

```
double[][] tab2  = {{1.3,3.5,7.8,4.0},
                    {7.5,8.3,6.9,3.2},{4.5,6.7,3.4,7.2}};
```

Geschachtelte Felder

Genau genommen gibt es in Java keine mehrdimensionalen Felder. Alle Felder sind in Java eindimensional. Die Komponenten eines Feldes können aber wieder Felder sein. Dadurch entstehen geschachtelte Felder. An Stelle mehrerer Dimensionen gibt es mehrere Schachtelungsebenen.

Geschachtelte Felder werden durch geschachtelte Aufzählungen initialisiert (siehe letztes Beispiel). Die Länge eines Feldes kann durch die Konstante `length` abgefragt werden. Mit `feld.length` erhält man die Länge der ersten Dimension, mit `feld[i].length` erhält man die Länge der 2. Dimension, wobei `i` ein beliebiger Wert innerhalb des Indexbereichs der 1. Dimension sein kann. Die konzeptionelle und die technische Sicht eines zweidimensionalen Feldes zeigt die Abb. 5.3-1.



Initialisierung:

```
double tab2[][] = {{1.3,3.5,7.8,4.0},{7.5,8.3,6.9,3.2},{4.5,6.7,3.4,7.2}};
```

Abb. 5.3-1: Konzeptionell sieht ein zweidimensionales Feld wie eine Tabelle aus, technisch besteht es in Java aus geschachtelten Feldern.

Beispiel

DemoFeld2

```
public class DemoFeld2
{
 public static void main (String args[])
 {
  double[][] tab2  =
   {{1.3,3.5,7.8,4.0},{7.5,8.3,6.9,3.2},{4.5,6.7,3.4,7.2}};
   System.out.println
    ("Länge 1. Dimension: " + tab2.length );
   System.out.println
    ("Länge 2. Dimension: " + tab2[1].length );
   for (int i = 0; i < tab2.length; i++)
   {
    for (int j = 0; j < tab2[0].length; j++)
        System.out.print(tab2[i][j] + "\t");
    System.out.println(); //Zeilenvorschub
   }
 }
}
```

Das Ergebnis sieht folgendermaßen aus:

```
Länge 1. Dimension: 3
Länge 2. Dimension: 4
1.3 3.5 7.8 4.0
7.5 8.3 6.9 3.2
4.5 6.7 3.4 7.2
```


Hinweis

Die Feldlänge weiterer Dimensionen kann variieren. Beispielsweise ist folgender Code möglich:

```
int[][] x = new int[2][];
x[0] = new int[3]; //Es gilt: x[0].length == 3
x[1] = new int[4]; //Es gilt: x[1].length == 4
```

Damit besitzt das erste Feld (i=0) in der zweiten Dimension die Länge 3 und das zweite Feld (i=1) die Länge 4.

Erhält man ein mehrdimensionales Feld (dem Typ sieht man nicht mehr an, wie er erzeugt wurde!) von irgendwoher, dann muss man beim Iterieren jedes mal neu prüfen wie lang nun die einzelnen gespeicherten Felder sind (siehe »Sonderformen von Feldern«, S. 187).

Mehrdimensionale Felder werden für viele Probleme benötigt.

Beispiel

Eine Bäckerei beliefert jeden Tag ihre Filialen. Jede Filiale gibt jeden Abend per Telefon die gewünschten Artikel für den nächsten Tag durch. Es wird jeweils die Menge und die Artikelnummer durchgegeben. Am Anfang wird die Filialnummer genannt. Nach Abschluss der Bestellung werden die Anzahl der Bestellpositionen und die Mengensumme dem Anrufer mitgeteilt. Alle Bestellungen aller Filialen sollen in einer Tabelle erfasst werden. Pro Artikel ist die Bestellmenge – über alle Fi-

lialen hinweg – aufzusummieren und in einer zusätzlichen Spalte am rechten Rand der Tabelle auszugeben. Pro Filiale sind die Summen aller Bestellmengen in einer zusätzlichen Zeile am Ende der Tabelle aufzuführen. Die Tabelle ist nach Abschluss aller eingegangenen Bestellungen auszugeben. Ein mögliches Programm, das diese Aufgabe löst, sieht folgendermaßen aus:

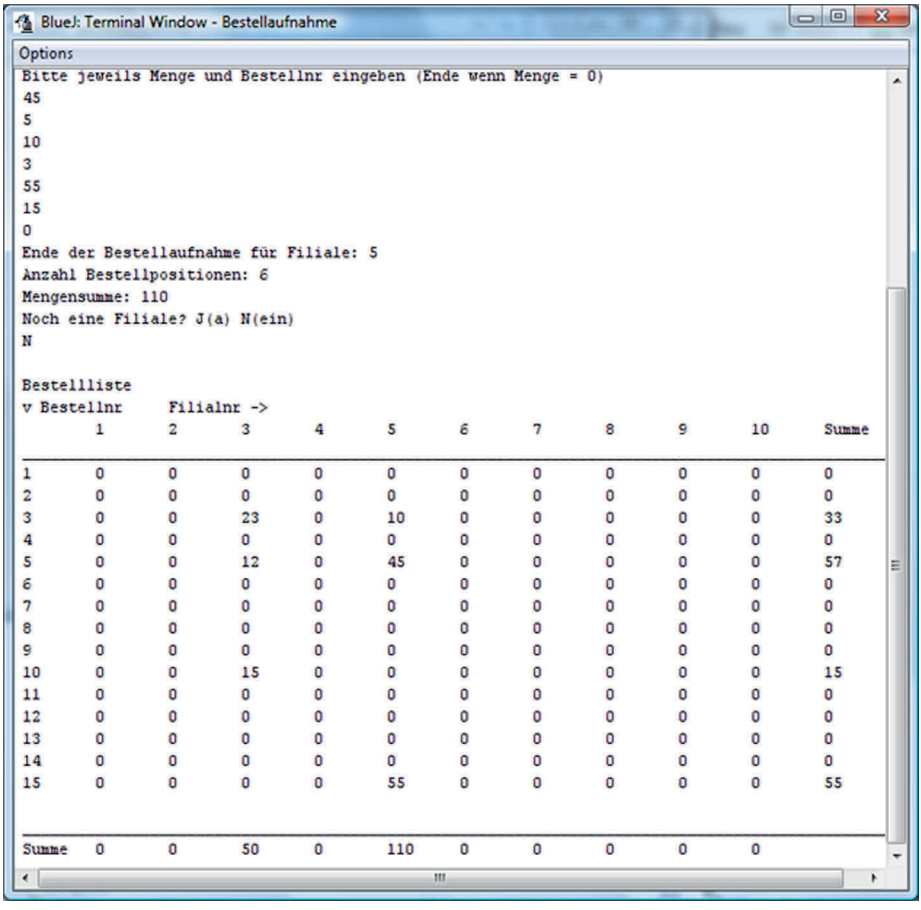
Bestell aufnahme

```
// Aufnahme von telefonischen Bestellungen
// 10 Filialen durchnummeriert von 1 bis 10
// 15 Artikel durchnummeriert von 1 bis 15

import inout.Console;

public class Bestellaufnahme
{
 public static void main (String args[])
 {
  final int  FILIAL_ANZAHL = 10;
    final int ARTIKEL_ANZAHL = 15;
    int filialnr; char weiter = 'N';
    int menge, bestellnr, mengensumme = 0,
        bestellpositionen = 0;
    //int Artikelsumme = 0;
    int bestelltab [][] =
        new int[FILIAL_ANZAHL + 1][ARTIKEL_ANZAHL + 1];
    //Vorbesetzung der Tabelle mit 0
    //i zählt die Filialen
    //j zählt die Artikelnr
    for (int i = 0; i < bestelltab.length; i++)
        for (int j = 0; j < bestelltab[0].length; j++)
        bestelltab [i][j] = 0;

  //Zeile 0 enthält Bestellmenge der jeweiligen Filiale
  //Spalte 0 enthält Bestellmenge des jeweiligen Artikels

  do
  {
   System.out.println();
   System.out.print
    ("Bitte Filialnr eingeben (zwischen 1 und 10): ");
   filialnr = Console.readInt();
   System.out.println
    ("Bitte jeweils Menge und Bestellnr eingeben " +
    "(Ende wenn Menge = 0)");
   menge = Console.readInt();
   mengensumme = 0;
   while (menge != 0)
   {
   bestellnr = Console.readInt();
   mengensumme = mengensumme + menge;
   bestellpositionen = bestellpositionen + 1;
   bestelltab[filialnr][bestellnr] = menge;
   bestelltab[0][bestellnr] = bestelltab[0][bestellnr] + menge;
```

```
    menge = Console.readInt();
    }
    System.out.println
     ("Ende der Bestellaufnahme für Filiale: " + filialnr);
    System.out.println
     ("Anzahl Bestellpositionen: " + bestellpositionen);
    System.out.println
     ("Mengensumme: " + mengensumme);
    bestelltab[filialnr][0] = mengensumme;
    //Weitere Filiale
    System.out.println("Noch eine Filiale? J(a) N(ein)");
    weiter = Console.readChar();
  } while (weiter =='J');

  //Ausgabe der Liste
  System.out.println();
  System.out.println("Bestellliste");
  System.out.println("v Bestellnr\tFilialnr ->");
  //Ausgabe der Filialnr
  System.out.print("\t");
  for (int i = 1; i < bestelltab.length; i++)
     System.out.print(i + "\t");
  System.out.print("Summe");
  //Ausgabe Linie
  System.out.println();
  for (int i = 1; i <= bestelltab.length + 1; i++)
     System.out.print("________");
  System.out.println();
  //Ausgabe der eigentlichen Tabelle
  for (int j = 1; j < ARTIKEL_ANZAHL + 1; j++)
  {
    //Ausgabe der Artikelnr
    System.out.print(j + "\t");
    //Ausgabe der Bestellmengen pro Artikelnr
    for (int i = 1; i <= FILIAL_ANZAHL; i++)
      System.out.print(bestelltab[i][j] + "\t");

    System.out.println(bestelltab[0][j]);//Artikelsumme
  }
  //Ausgabe Linie
  System.out.println();
  for (int i = 1; i <= bestelltab.length + 1; i++)
     System.out.print("________");
  System.out.println();
   //Mengensumme pro Filiale
   System.out.print("Summe\t");
   for (int i = 1; i < bestelltab.length; i++)
     System.out.print(bestelltab[i][0] + "\t");
  }
}
```

Die Ausgabe zeigt die Abb. 5.3-2 (Ausschnitt).

Zeichnen Sie ein Struktogramm des Programms `Bestellaufnahme`.



BlueJ: Terminal Window - Bestellaufnahme

Options

```
Bitte jeweils Menge und Bestellnr eingeben (Ende wenn Menge = 0)
45
5
10
3
55
15
0
Ende der Bestellaufnahme für Filiale: 5
Anzahl Bestellpositionen: 6
Mengensumme: 110
Noch eine Filiale? J(a) N(ein)
N

Bestellliste
v Bestellnr     Filialnr ->
        1       2       3       4       5       6       7       8       9       10      Summe
-------------------------------------------------------------------------------------------
1       0       0       0       0       0       0       0       0       0       0       0
2       0       0       0       0       0       0       0       0       0       0       0
3       0       0       23      0       10      0       0       0       0       0       33
4       0       0       0       0       0       0       0       0       0       0       0
5       0       0       12      0       45      0       0       0       0       0       57
6       0       0       0       0       0       0       0       0       0       0       0
7       0       0       0       0       0       0       0       0       0       0       0
8       0       0       0       0       0       0       0       0       0       0       0
9       0       0       0       0       0       0       0       0       0       0       0
10      0       0       15      0       0       0       0       0       0       0       15
11      0       0       0       0       0       0       0       0       0       0       0
12      0       0       0       0       0       0       0       0       0       0       0
13      0       0       0       0       0       0       0       0       0       0       0
14      0       0       0       0       0       0       0       0       0       0       0
15      0       0       0       0       55      0       0       0       0       0       55

-------------------------------------------------------------------------------------------
Summe   0       0       50      0       110     0       0       0       0       0
```

Abb. 5.3-2: So sieht das Ergebnis des Programmlaufs `Bestellannahme` aus (Ausschnitt).

Felder können beliebig viele Dimensionen besitzen. In der Regel werden mehrdimensionale Felder mit ineinandergeschachtelten `for`-Schleifen bearbeitet.

Beispiel

Ein Elektrogroßhändler bewahrt seine Artikel in einem Hochregallager auf (Abb. 5.3-3). Die Lagerverwaltung erfolgt durch ein Lagerverwaltungsprogramm. In einer Tabelle ist vermerkt, in welchem Fach welcher Artikel (Artikelnummer) und welche Menge davon gelagert sind. Bei einer eingehenden Bestellung wird durch Eingabe der Artikelnummer überprüft, ob noch die gewünschte Bestellmenge vorhanden ist. Ein Artikel kann in mehreren Fächern gelagert sein.

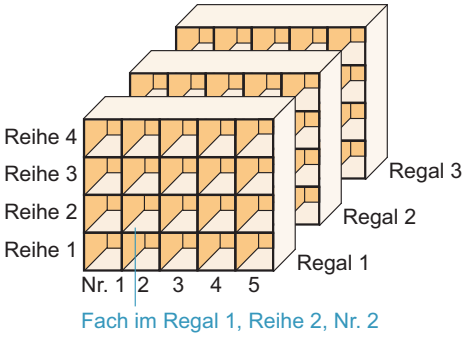


Abb. 5.3-3: So sieht das Regallager aus.

Lager Verwaltung


```
// Lagerverwaltung

import inout.Console;
public class LagerVerwaltung
{
 public static void main (String args[])
 {
  int lager [][][][] = new int[3][4][5][2];
  //1. Ebene: Lager besteht aus 3 Regalen
  //2. Ebene: Regal besteht aus 4 Reihen
  //3. Ebene: Reihe besteht aus 5 Fächern
  //4. Ebene: pro Fach wird Artikelnr und Menge gespeichert (2)
  int artikelnr;
  char weiter = 'N';
  //Initialisierung
  for (int i = 0; i < lager.length; i++) //alle Regale
   for (int j = 0; j < lager[0].length; j++) //alle Reihen
    for (int k = 0; k < lager[0][0].length; k++) //alle Fächer
     {
       lager[i][j][k][0]=(int)(Math.random()*100.0);//Artikelnr
       //Zufallszahlen zwischen 0 und 100
       lager[i][j][k][1]=(int)(Math.random()*50.0);//Menge
       //Zufallszahlen zwischen 0 und 50
     }

  do
  {
   int menge = 0;
   System.out.println();
   System.out.print
    ("Bitte Artikelnummer eingeben (zwischen 0 und 100): ");
   artikelnr = Console.readInt();
   for (int i = 0; i < lager.length; i++) //alle Regale
    for (int j = 0; j < lager[0].length; j++) //alle Reihen
        for (int k = 0; k < lager[0][0].length; k++) //alle Fä.
            if (lager[i][j][k][0] == artikelnr)
               menge =  menge + lager[i][j][k][1];
   System.out.println("Artikelnummer: " + artikelnr);
```

```
    System.out.println("Gelagerte Menge: " + menge);
    //Weitere Abfragen
    System.out.println("Noch eine Abfrage? J(a) N(ein)");
    weiter = Console.readChar();
   } while (weiter =='J' || weiter == 'j');
  }
}
```

Ein Ausschnitt aus dem Programmlauf sieht wie folgt aus:

```
Bitte Artikelnummer eingeben (zwischen 0 und 100): 17
Artikelnummer: 17
Gelagerte Menge: 76
Noch eine Abfrage? J(a) N(ein)
n
```

Erweitern Sie das Programm Lagerverwaltung so, dass die Lagerorte für die gegebene Artikelnummer mit ausgegeben werden.

Ist die Länge eines Feldes einmal festgelegt, dann kann sie nicht mehr geändert werden!

5.4 Sonderformen von Feldern **

In Java kann für jede Schachtelungsebene eines Feldes eine individuelle Länge angegeben werden, sodass beliebige Feldstrukturen deklariert werden können. Ein solches Feld kann direkt mit Werten initialisiert werden oder die Längenangabe kann mit `new` festgelegt werden. Die Längenangabe der obersten Ebene ist Pflicht, die anderen Angaben können dann entsprechend der Struktur vorgenommen werden. Wird eine untere Ebene festgelegt, dann müssen die Längen aller übergeordneten Ebenen ebenfalls angegeben werden, z. B. `char[][] tab = new char [5][]; tab[0] = new char[10]; tab[1] = new char[5];`.

Unterschiedliche Längen innerhalb einer Dimension

Bei mehrdimensionalen Feldern besitzen alle Felder einer bestimmten Dimension dieselbe Größe. Es gibt jedoch eine ganze Reihe von Anwendungsfällen, wo die Größe innerhalb einer Dimension unterschiedlich ist. Die Deklaration solcher Felder ist in Java möglich.

Beispiel

Die Firma Ersatzteile GmbH residiert in einem schönen alten Fabrikgebäude. Bei der Einrichtung eines Hochregallagers muss auf die Gebäudestruktur Rücksicht genommen werden. Daraus ergibt sich folgende Regalstruktur:

- 1. Regal: Höhe 5 Reihen, je Reihe 10 Fächer (= 50 Fächer)
- 2. Regal: Höhe 3 Reihen, Reihe 1: 10 Fächer, Reihe 2: 7 Fächer, Reihe 3: 5 Fächer (= 22 Fächer)
- 3. Regal: Höhe 6 Reihen, je Reihe 12 Fächer (= 72 Fächer)

Um eine solche Regalstruktur als Feld zu vereinbaren, erlaubt es Java, für jede einzelne Schachtelungsebene jeweils die Länge festzulegen. Das zugehörige Programm sieht wie folgt aus:

Demo FeldLaenge

```
// Beispiel für einen unregelmäßigen Feldaufbau

public class DemoFeldLaenge
{
 public static void main (String args[])
 {
  int[][][][] feld  = new int[3][][][]; //3 Regale
  //1. Regal: Höhe 5 Reihen, je Reihe 10 Fächer (= 50 Fächer)
  feld[0] = new int[5][10][2];
  //2. Regal: Höhe 3 Reihen
  feld[1] = new int[3][][]; //Festlegung der Höhe
  //2. Regal: Höhe 1: 10 Fächer, Höhe 2: 7 Fächer,
  //Höhe 3: 5 Fächer
  feld[1][0] = new int[10][2];//Festlegung der Fächer
  feld[1][1] = new int[7][2]; //Festlegung der Fächer
  feld[1][2] = new int[5][2]; //Festlegung der Fächer
  //3. Regal: Höhe 6 Reihen, je Reihe 12 Fächer
  feld[2] = new int [6][12][2];

  //Initialisierung
  for (int i = 0; i < feld.length; i++) //alle Regale
   for (int j = 0; j < feld[i].length; j++) //alle Reihen
     for (int k = 0; k < feld[i][j].length; k++) //alle Fächer
      {
       feld[i][j][k][0] = (int)(Math.random()*100.0);//Artikelnr
       //Zufallszahlen zwischen 0 und 100
       feld[i][j][k][1] = (int)(Math.random()*50.0);//Menge
       //Zufallszahlen zwischen 0 und 50
       System.out.println("Regal: " + (i+1) + " Reihe: "
         + (j+1) + " Fach: " + (k+1) + " Artikelnr: "
         + feld[i][j][k][0] + " Menge: " + feld[i][j][k][1]);
      }
 } //Ende main
}
```

Als Ergebnis wird folgendes ausgegeben (Ausschnitt):

```
Regal: 3 Reihe: 6 Fach: 9 Artikelnr: 10 Menge: 21
Regal: 3 Reihe: 6 Fach: 10 Artikelnr: 16 Menge: 6
Regal: 3 Reihe: 6 Fach: 11 Artikelnr: 18 Menge: 20
Regal: 3 Reihe: 6 Fach: 12 Artikelnr: 13 Menge: 30
```

Machen Sie eine Skizze des Hochregallagers und gehen Sie die Programmstruktur anhand der Skizze durch.

Angabe der Elementanzahl

Bei geschachtelten Feldern muss bei der Felddeklaration mindestens die Elementanzahl der obersten Schachtelungsebene angegeben werden, z. B. `int[][][][] feld = new int[3][][][];`

Bei den tieferen Ebenen ist die Angabe bei der Deklaration optional. Wird eine Längenangabe auf einer tieferen Ebene gemacht,

dann müssen alle darüberliegenden Ebenen ebenfalls festgelegt werden, z. B. `feld[0] = new int[5][10][2];`.

Lücken dürfen bei der Festlegung nicht entstehen, z. B. ist `feld[0] = new int[5][][2];` nicht erlaubt.

Unregelmäßige Felder können auch direkt mit Werten initialisiert werden.

Direkte Initialisierung

Beispiel

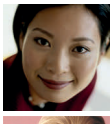
DemoDreieck

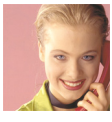
```
public class DemoDreieck
{
 public static void main (String args[])
 {
  int[][] dreieck = {{1}, {2,3},{4,5,6}, {7,8,9,10}};
  for (int i= 0; i < dreieck.length; i++)
  {
    for (int j = 0; j < dreieck[i].length; j++)
     System.out.print(dreieck[i][j] + " ");
    System.out.println();
  }
 }
}
```

Folgendes Ergebnis wird ausgegeben:

```
1
2 3
4 5 6
7 8 9 10
```


5.5 OptiTravel: Tabellen *





Die Anwendungs-Entwicklerin der Firma WebSoft, Frau Anton, setzt sich mit der Junior-Programmiererin, Frau Jung, zusammen und bespricht die Datenstrukturen für die Anwendung OptiTravel. Sie legen folgende Tabellen fest:

- **Zugtabelle:** Aufbau wie eine Entfernungstabelle als Dreieckstabelle, da bei einer Hinwärtsverbindung von einer Stadt zur anderen auch eine entsprechende Rückwärtsverbindung besteht. Anstelle der Entfernung wird jedoch die durchschnittliche Reisezeit mit einem IC oder ICE in Minuten eingetragen.
- **Flugtabelle:** Da bei Flügen *nicht* immer eine direkte Hin- und Rückflugverbindung besteht, muss eine vollständige Tabelle angelegt werden. In der Tabelle wird die in den Flugplänen angegebene Flugzeit in Minuten eingetragen. Zusätzlich wird getrennt angegeben, wie viel Zeit mit dem Auto in Minuten benötigt wird, um von der jeweiligen Innenstadt bis zum Flughafen zu gelangen. Besitzt ein Ort keinen Flughafen, dann wird der Wert Null angegeben. In einem gesonderten Eintrag wird dann vermerkt, welcher Nachbarort einen Flughafen besitzt. Die Anfahrtszeit mit dem Auto bezieht sich dann auf den Flughafen der Nachbarstadt.

- **Autotabelle:** Da beim Auto die Fahrzeit stark von den Straßenverhältnissen abhängt, sollen pro Verbindung folgende Informationen gespeichert werden: Entfernung in km, jeweils Streckenabschnitt Autobahn mit 130 km/h Geschwindigkeitsbeschränkung, Autobahn ohne Geschwindigkeitsbeschränkung, Anteil Bundesstraße, Anteil Landstraße, Anteil Stadtverkehr. Es genügt eine »halbe« Tabelle.

Beispiel

Zunächst entwickelt Frau Jung ein Beispiel für die Flugtabelle. Sie nimmt drei Flughäfen A, B und C mit folgenden Flugzeiten an:

```
A nach B: 60 min, B nach A: 70 min
A nach C: 80 min, C nach A: keine direkte Verbindung: 0
B nach C: 30 min, C nach B: 40 min
```

Das ergibt folgende doppelte Tabelle:

```
          nach
von    |  A      B      C
A      |   0     60     80
B      |  70      0     30
C      |   0     40      0
```

Da es von einem Ort zu sich selbst jeweils keine Verbindung gibt, wird dort ebenfalls 0 eingetragen, z. B. von A nach A, d. h. die Diagonale der Tabelle wird mit Nullen belegt.

Frau Jung entwickelt folgendes Programm, wobei sie besonderen Wert auf gute Lesbarkeit legt.

Der Zugriff auf die Tabellen ist gut lesbar, da als Indizes Konstantenbezeichnungen verwendet werden.

OptiTravel Tabellen

```
// OptiTravel-Tabellen: Speicherung der Daten

public class OptiTravelTabellen
{
 public static void main (String args[])
 {
  final int Berlin = 0, Hamburg = 1, Muenchen = 2,
   Koeln = 3, Frankfurt = 4, Dortmund = 5, Stuttgart = 6,
   Essen = 7, Duesseldorf = 8, Bremen = 9;
  final String[] staedte =
   {"Ber","Ham","Mue","Koe","Fra","Dor","Stu","Ess","Due","Brm"};
   //Zugtabelle:  Zeit in Minuten
   int[][] zug  =
    {{0}, //Von Berlin nach: Berlin
     {150, 0}, //Von Hamburg nach: Berlin, Hamburg
     {380,360,0}, // Von München nach: Berlin, Hamburg, München
     {250,240,270,0},//Von Köln nach: Berlin, Hamburg, München, Köln
     {240,210,240,70,0}, //Von Frankfurt nach:  Berlin, Hamburg,
                  //München, Köln, Frankfurt
     {250,240,360,70,210,0}, //Von Dortmund nach: Berlin, Hamburg,
                  //München, Köln, Frankfurt, Dortmund
```

```
 {330,310,140,135,80,210,0}, //Von Stuttgart: Berlin, Hamburg,
              //München, Köln, Frankfurt, Dortmund, Stuttgart
 {210,190,330,50,135,20,190,0}, //Von Essen: Berlin, Hamburg,
              //München, Köln, Frankfurt, Dortmund, Stuttgart,
              // Essen
 {250,180,300,30,100,55,165,30,0}, //Düsseldorf: Berlin,
              //Hamburg, München, Köln, Frankfurt, Dortmund,
              //Stuttgart, Essen, Düsseldorf
 {170,60,360,180,220,105,320,130,150,0}}; //Bremen: Berlin,
              //Hamburg, München, Köln, Frankfurt, Dortmund,
              //Stuttgart, Essen, Düsseldorf, Bremen

//Ausgabe der Zugtabelle
System.out.println("Direkte Zugverbindungen
                  (Fahrzeit in Minuten)");
for (int i = 0; i < zug.length; i++)
{
 //Tabellenbeschriftung links
 System.out.print(staedte[i] + " ");
 //Tabelleninhalt
 for (int j = 0; j < zug[i].length; j++)
    System.out.print(zug[i][j] + " ");
 System.out.println();
}
//Tabellenbeschriftung unten
System.out.print("    ");
for (int i = 0; i < zug.length; i++)
 System.out.print(staedte[i] + " ");
System.out.println("\n_____________________________________");


//Flugtabelle
//Flugzeit in Minuten, Fahrzeit mit Auto von der Innenstadt
//zum nächsten Flughafen(in Minuten)
//hat der Ort keinen Flughafen, dann wird der nächste
//Flughafenort angegeben und die Fahrzeit mit dem Auto
//bezieht sich auf diesen Flughafen, -1 bedeutet:
//eigener Flughafen, 0 bis 9: Flughafenort
int[][][] flug  =
//Von Berlin nach:
{{{0,0,-1},{0,0,-1},{70,50,-1},{65,50,-1},{65,50,-1},
    {70,50,-1},{70,50,-1},{65,50,8},{65,50,-1},{60,50,-1}},
 //Von Hamburg nach
 {{0,0,-1},{0,0,-1},{80,40,-1},{55,40,-1},{65,40,-1},
    {0,0,-1},{70,40,-1},{55,40,8},{55,40,-1},{0,0,-1}},
 //Von München nach
 {{70,60,-1},{80,60,-1},{0,0,-1},{70,60,-1},{65,60,-1},
    {90,60,-1},{55,60,-1},{70,60,8},{70,60,-1},{75,60,-1}},
 //Von Köln nach
 {{65,40,-1},{60,40,-1},{65,40,-1},{0,0,-1},{45,40,-1},
    {0,0,-1},{0,0,-1},{0,0,-1},{0,0,-1},{0,0,-1}},
 //Von Frankfurt nach
 {{65,15,-1},{60,15,-1},{60,15,-1},{40,15,-1},{0,0,-1},
    {0,0,-1},{50,15,-1},{45,15,8},{45,15,-1},{55,15,-1}},
 //Von Dortmund nach
 {{70,30,-1},{0,0,-1},{85,30,-1},{0,0,-1},{0,0,-1},
```

```
    {0,0,-1},{65,30,-1},{0,0,-1},{0,0,-1},{0,0,-1}},
 //Stuttgart nach
 {{70,40,-1},{70,40,-1},{55,40,-1},{0,0,-1},{50,40,-1},
    {215,40,-1},{0,0,-1},{65,40,8},{65,40,-1},{70,40,-1}},
 //Von Essen nach (Flughafen = Düsseldorf = 8)
 {{65,30,8},{60,30,8},{70,30,8},{0,0,8},{55,30,8},
    {0,0,-1},{65,30,8},{0,0,-1},{0,30,8},{160,30,8}},
 //Von Düsseldorf nach
 {{65,15,-1},{60,15,-1},{70,15,-1},{0,0,-1},{55,15,-1},
    {0,0,-1},{65,15,-1},{0,0,-1},{0,0,-1},{0,0,-1}},
 //Von Bremen nach
 {{60,15,-1},{0,0,-1},{75,15,-1},{0,0,-1},{60,15,-1},
    {260,15,-1},{65,15,-1},{0,0,8},{0,0,-1},{0,0,-1}},
 };
 //Nach: Berlin, Hamburg, München, Köln, Frankfurt,Dortmund,
 //Stuttgart, Essen, Düsseldorf, Bremen
 //Hinweis: Essen hat keinen eigenen Flughafen,
 //jedoch liegt Düsseldorf in der Nähe


//Ausgabe der Flugtabelle
System.out.println
 ("Flugverbindungen (Flugzeit, Zufahrt Auto in Minuten)");
for (int i = 0; i < flug.length; i++)
{
 //Tabellenbeschriftung links
 System.out.print(staedte[i] + " ");
 //Tabelleninhalt
 for (int j = 0; j < flug[i].length; j++)
    System.out.print("(" + flug[i][j][0] + ", "
    + flug[i][j][1] + ")" );
 System.out.println();
}
//Tabellenbeschriftung unten
System.out.print("    ");
for (int i = 0; i < flug.length; i++)
 System.out.print(staedte[i] + "     ");
 System.out.println
  ("\n_____");


//Autotabelle
final int KM = 0, Autobahn130 = 1, AutobahnFrei = 2,
Bundesstr = 3,
 Landstr = 4, Stadt = 5;
int[][][] auto  =
 //Von Berlin nach: Berlin
{{{0,0,0,0,0,0}},
 //Von Hamburg nach: Berlin, Hamburg
 {{280,140,100,0,0,40},{0,0,0,0,0,0}},
 // Von München nach: Berlin, Hamburg, München
 {{590,270,300,0,0,20},{770,420,300,0,0,50},
  {0,0,0,0,0,0}},
 //Von Köln nach: Berlin, Hamburg, München, Köln
 {{570,250,300,0,0,20},{420,170,200,0,0,50},
  {570,220,300,0,0,50},{0,0,0,0,0,0}},
```

```
  //Von Frankfurt nach:  Berlin, Hamburg,
  //München, Köln, Frankfurt
  {{550,250,250,0,0,50},{490,240,200,0,0,50},
   {400,180,200,0,0,20},{190,70,100,0,0,20},
   {0,0,0,0,0,0}},
  //Von Dortmund nach: Berlin, Hamburg, München,
  //Köln, Frankfurt, Dortmund
  {{490,200,270,0,0,20},{300,100,180,0,0,20},
   {500,270,200,0,0,30},{90,30,10,30,0,20},
   {200,70,100,0,0,30},{0,0,0,0,0,0}},
  //Von Stuttgart: Berlin, Hamburg, München,
  //Köln, Frankfurt, Dortmund, Stuttgart
  {{630,300,300,0,0,30},{500,200,280,0,0,20},
   {200,80,100,0,0,20},{300,70,200,0,0,30},
   {150,70,70,0,0,10},{350,230,100,0,0,20},{0,0,0,0,0,0}},
  //Von Essen: Berlin, Hamburg, München,
  //Köln, Frankfurt, Dortmund, Stuttgart, Essen
  {{450,200,200,30,0,20},{300,180,80,20,0,20},
   {530,200,300,0,0,30},{50,20,0,20,0,10},
   {150,140,0,0,0,10},{20,5,0,0,0,15},
   {350,330,0,0,0,20},{0,0,0,0,0,0}},
  //Düsseldorf: Berlin, Hamburg, München,
  //Köln, Frankfurt, Dortmund, Stuttgart, Essen, Düsseldorf
  {{550,220,230,0,20,30},{350,130,200,0,0,20},
   {500,250,200,0,0,50},{20,5,0,0,0,15},
   {180,70,100,0,0,10},{60,15,30,0,0,15},
   {300,180,100,0,0,20},{30,10,0,10,0,10},{0,0,0,0,0,0}},
  //Bremen: Berlin, Hamburg, München, Köln, Frankfurt,
  //Dortmund, Stuttgart, Essen, Düsseldorf, Bremen
  {{350,130,200,0,0,20},{80,40,30,0,0,10},{750,330,400,0,0,20},
   {250,130,100,0,0,20},{370,150,200,0,0,20},
   {180,70,100,0,0,20},{500,280,200,0,0,20},
   {200,80,100,0,0,20},{290,170,100,0,0,20},{0,0,0,0,0,0}}};

 //Ausgabe der Autotabelle
 System.out.println
  ("Entfernungen Auto in km " +
  "\nzusätzl. gespeichert: Autobahn 130, "
  +"Autobahn frei, Bundesstr, Landstr, Stadt");
 for (int i = 0; i < auto.length; i++)
 {
  //Tabellenbeschriftung links
  System.out.print(staedte[i] + " ");
  //Tabelleninhalt
  for (int j = 0; j < auto[i].length; j++)
     System.out.print(auto[i][j][0] + " ");
  System.out.println();
 }
 //Tabellenbeschriftung unten
 System.out.print("    ");
 for (int i = 0; i < auto.length; i++)
  System.out.print(staedte[i] + " ");
 System.out.println("\n_____________________________________");

System.out.println
 ("Die Auto-Entfernung von Dortmund nach Berlin beträgt: "
```

```
    + auto[Dortmund][Berlin][KM]);
   System.out.println
   ("Der Autobahnanteil von Dortmund nach Berlin beträgt: "
   + (auto[Dortmund][Berlin][Autobahn130]
   + auto[Dortmund][Berlin][AutobahnFrei]));
 }
}
```

Die Abb. 5.5-1 zeigt das Ergebnis des Programmlaufs.

```
Direkte Zugverbindungen (Fahrzeit in Minuten)
Ber 0
Ham 150 0
Mue 380 360 0
Koe 250 240 270 0
Fra 240 210 240 70 0
Dor 250 240 360 70 210 0
Stu 330 310 140 135 80 210 0
Ess 210 190 330 50 135 20 190 0
Due 250 180 300 30 100 55 165 30 0
Brm 170 60 360 180 220 105 320 130 150 0
    Ber Ham Mue Koe Fra Dor Stu Ess Due Brm
_____________________________________________
Flugverbindungen (Flugzeit, Zufahrt Auto in Minuten)
Ber (0, 0)(0, 0)(70, 50)(65, 50)(65, 50)(70, 50)(70, 50)(65, 50)(65, 50)(60, 50)
Ham (0, 0)(0, 0)(80, 40)(55, 40)(65, 40)(0, 0)(70, 40)(55, 40)(55, 40)(0, 0)
Mue (70, 60)(80, 60)(0, 0)(70, 60)(65, 60)(90, 60)(55, 60)(70, 60)(70, 60)(75, 60)
Koe (65, 40)(60, 40)(65, 40)(0, 0)(45, 40)(0, 0)(0, 0)(0, 0)(0, 0)(0, 0)
Fra (65, 15)(60, 15)(60, 15)(40, 15)(0, 0)(0, 0)(50, 15)(45, 15)(45, 15)(55, 15)
Dor (70, 30)(0, 0)(85, 30)(0, 0)(0, 0)(0, 0)(65, 30)(0, 0)(0, 0)(0, 0)
Stu (70, 40)(70, 40)(55, 40)(0, 0)(50, 40)(215, 40)(0, 0)(65, 40)(65, 40)(70, 40)
Ess (65, 30)(60, 30)(70, 30)(0, 0)(55, 30)(0, 0)(65, 30)(0, 0)(0, 30)(160, 30)
Due (65, 15)(60, 15)(70, 15)(0, 0)(55, 15)(0, 0)(65, 15)(0, 0)(0, 0)(0, 0)
Brm (60, 15)(0, 0)(75, 15)(0, 0)(60, 15)(260, 15)(65, 15)(0, 0)(0, 0)(0, 0)
    Ber     Ham     Mue     Koe     Fra     Dor     Stu     Ess     Due     Brm
_____________________________________________
Entfernungen Auto in km
zusätzl. gespeichert: Autobahn 130, Autobahn frei, Bundesstr, Landstr, Stadt
Ber 0
Ham 280 0
Mue 590 770 0
Koe 570 420 570 0
Fra 550 490 400 190 0
Dor 490 300 500 90 200 0
Stu 630 500 200 300 150 350 0
Ess 450 300 530 50 150 20 350 0
Due 550 350 500 20 180 60 300 30 0
Brm 350 80 750 250 370 180 500 200 290 0
    Ber Ham Mue Koe Fra Dor Stu Ess Due Brm
_____________________________________________
Die Auto-Entfernung von Dortmund nach Berlin beträgt: 490
Der Autobahnanteil von Dortmund nach Berlin beträgt: 470
```

Abb. 5.5-1: Ergebnis der Ausführung des Programms `OptiTravelTabellen`.

Gehen Sie das Programm Zeile für Zeile durch, um es vollständig zu verstehen.

5.6 Einfaches Sortieren *

Zum Sortieren von Informationen gibt es einfache – besser gesagt: langsame – und schnelle Sortierverfahren. Die bekannten einfachen Sortierverfahren sind »Sortieren durch Auswahl« *(selection sort)*, »Sortieren durch Austauschen« *(bubble sort, exchange sort)* und »Sortieren durch Einfügen« *(insertion sort)*. Sind n Informationen zu sortieren, dann benötigen sie eine Sortierzeit, die proportional n^2 ist. Schnelle Sortierverfahren benötigen dagegen nur eine Zeit, die proportional $n \log_2 n$ ist. Viele Sortierverfahren arbeiten auf Feldern.

Das Sortieren von Informationen ist eine der häufigsten Tätigkeiten, die von Programmen durchgeführt werden.

Es gibt daher viele Sortier-Algorithmen, die zum Teil auf sehr spezielle Situationen zugeschnitten sind.

2 Kategorien

Nach dem Zeitverhalten lassen sich einfache und schnelle Sortierverfahren unterscheiden. Einfache bzw. elementare Sortierverfahren sind einfach zu programmieren, benötigen für das Sortieren aber viel Zeit. Die schnellen Sortierverfahren sind aufwendig zu programmieren, aber schnell im Sortieren.

Quadratisch vs. logarithmisch

Der Zeitaufwand ist immer abhängig von der Menge der zu sortierenden Informationen. Sind n Informationen zu sortieren, dann benötigen die einfachen Sortierverfahren dazu n^2 an Zeit. Die schnellen Sortierverfahren haben einen Aufwand, der proportional zu $n \cdot \log_2 n$ verläuft.

Hinweis

Mit $\log_2$ wird der Zweierlogarithmus oder Logarithmus zur Basis 2 bezeichnet (oft wird statt $\log_2$ auch ld verwendet). Der Zweierlogarithmus von n ist der Exponent k, mit dem man eine Zahl 2 potenzieren muss, um den Wert n zu erhalten: $n / 2^k = 1$ oder $n = 2^k$ oder $k = \log_2 n$.

Beispiel

Sie wollen 10 Millionen Informationen sortieren ($n = 10^7$). Wird zum Sortieren einer Information eine Zeit von einer millionstel Sekunde ($c = 10^{-6}$ s) benötigt, dann werden mit einfachen Sortierverfahren

$n^2 \cdot c = (10^7)^2 \cdot 10^{-6} = 10^8 = 100\,000\,000$ Sekunden = 1157,4 Tage benötigt.

Schnelle Sortierverfahren benötigen dazu

$n \log_2 n \cdot c = 10^7 \cdot \log_2 10^7 \cdot 10^{-6} = 10^7 \cdot 23{,}25 \cdot 10^{-6} = 232{,}5$ Sekunden = 3,87 Minuten.

Der Unterschied ist deutlich!

3 einfache Verfahren

Es gibt drei bekannte einfache Sortierverfahren:

- **Sortieren durch Auswahl** *(selection sort)*
- **Sortieren durch Austauschen** *(bubble sort, exchange sort)*
- **Sortieren durch Einfügen** *(insertion sort)*

Für kleine Sortiermengen sind diese Verfahren effektiv. Die Programme sind kurz und leicht verständlich.

Beispiel 1a: Sortieren durch Auswahl

Die Kundendaten einer Firma sollen sortiert werden. Es soll das Verfahren **Sortieren durch Auswahl** verwendet werden, das wie folgt funktioniert:

Die Sortierstrategie wird anhand eines aus einzelnen Fächern bestehenden Karteikastens veranschaulicht (Abb. 5.6-1). In jedem Fach des Karteikastens befindet sich eine Karteikarte. Es werde angenommen, dass auf jeder Karteikarte der Nachname des Kunden steht. Die Karteikarten befinden sich in unsortierter Reihenfolge in dem Karteikasten. Um die Karteikarten zu sortieren, werden alle Karteikarten von vorne nach hinten durchgesehen. Gemerkt wird sich die Nummer des Fachs, in dem sich die Karteikarte mit dem lexikografisch kleinsten Kundennamen befindet. Beim lexikografischen Vergleich wir die alphabetische Ordnung auf die aus Zeichen gebildeten Worte übertragen. Gibt es mehrere gleiche Kundennamen, dann wird das Fach gemerkt, in dem der Name zum ersten Mal erscheint. Anschließend wird die 1. Karte mit der Karte ausgetauscht, auf der der lexikografisch kleinste Namen steht.

Der Vorgang wird wiederholt, jedoch wird jetzt bei der Karteikarte begonnen, die im 2. Fach steht usw.

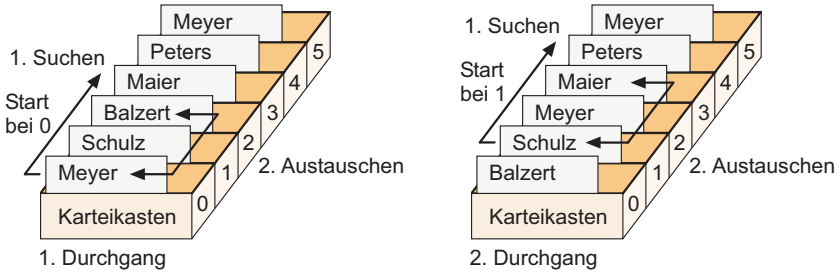


Abb. 5.6-1: Sortieren von Karteikarten.

Spielen Sie in Gedanken oder mit einem Karteikasten dieses Sortierverfahren durch und überlegen sich wie ein entsprechendes Programm aussehen kann.

Felder vom Typ String

In Java können auch Felder vom Typ String angelegt werden, z. B.

```
String kunden[] = {"Meyer","Schulz", "Balzert"};
```

Lexikografischer Vergleich

Anders als bei den einfachen Typen gibt es eine besondere Operation, um zwei Zeichenketten lexikografisch miteinander zu vergleichen. Die Syntax sieht folgendermaßen aus:

```
zeichenkette1.compareTo(zeichenkette2) # 0
```

wobei # für einen der Vergleichsoperatoren `<`, `<=`, `>`, `>=`, `==` oder `!=` steht.

Der Vergleich basiert auf dem **Unicode**-Wert jedes Zeichens in der Zeichenkette (siehe auch: »Der Zeichentyp char«, S. 85).

Beispiel 1b

SortAuswahl

Das folgende Programm sortiert die Zeichenketten in dem Feld `kunden`:

```
// Beispiel für das Sortieren durch Auswahl
// hier: Sortieren von Nachnamen

public class SortAuswahl
{
 public static void main (String args[])
 {
  String[] kunden = {"Meyer","Schulz", "Balzert",
      "Maier","Peters", "Meyer"};
  String min, merke;
  int pos, posMin;
  //Sortieren und Vertauschen
  for (int i = 0; i < kunden.length; i++)
  {
      System.out.println(i);
      //Kleinste Position ab i suchen
        posMin = i; min = kunden[i];
        for (pos = i + 1; pos < kunden.length; pos++)
         if (kunden[pos].compareTo(min)< 0) //Abfrage auf <
          {
              min = kunden[pos];
              posMin = pos; //Kleinste Position merken
              System.out.println(min);
          }
        //Vertauschen
        merke = kunden[i];
        kunden[i] = kunden[posMin];
        kunden[posMin] = merke;
  }
  System.out.println("Lexikografisch sortierte Namensliste");
  for (int i = 0; i < kunden.length; i++)
       System.out.println(kunden[i]);
 }
}
```

Das Ergebnis sieht folgendermaßen aus:

```
0
Balzert
1
Meyer
Maier
2
3
Peters
Meyer
4
5
Lexikografisch sortierte Namensliste
Balzert
Maier
Meyer
Meyer
Peters
Schulz
```

Ändern Sie das Programm `SortAuswahl` so ab, dass die gezogenen Lottozahlen der Größe nach sortiert werden. Was ist anders als bei dem Programm `SortAuswahl` (abgesehen vom Typ des Feldes)?

Hinweis

Die Java-Klasse `Arrays` stellt eine Vielzahl von Such- und Sortiermethoden über Felder zur Verfügung.

5.7 Iteration über Felder: Die erweiterte for-Schleife **

Sollen alle Elemente eines Feldes durchlaufen werden, dann kann dafür eine erweiterte `for`-Schleife in Java verwendet werden. Auf eine Zählvariable wird verzichtet, stattdessen wird der Typ des Feldelements sowie ein Bezeichner angegeben. Getrennt durch einen Doppelpunkt wird das zu durchlaufende Feld angegeben, z. B. `for (double feldelement : messreihe)`.

In vielen Anwendungsfällen müssen alle Elemente eines Feldes (`array`) durchlaufen werden.

for-Schleife

Eine Zählschleife bzw. `for`-Schleife wird verwendet, wenn die Anzahl der Wiederholungen bekannt ist. Beim Durchlaufen von Feldern ist die Anzahl der Wiederholungen bekannt, da die Länge eines Feldes über die Konstante `feld.length` abgefragt werden kann.

Beispiel 1a

DemoFor1

Es soll der Durchschnittswert einer Messreihe ermittelt werden:

```
// Beispiel für Iterieren über Felder
public class DemoFor
{
```

```
 public static void main (String args[])
 {
   double summe = 0;
   double durchschnitt;
   double messreihe[] = {1.0, 5.5, 3.72, 1.2, 7.89};
   int anzahl = messreihe.length;
   for (int i = 0; i < anzahl; i++)
      summe = summe + messreihe[i];
   System.out.println("Durchschnitt: " + summe / anzahl);
 }
}
```

Wie das Beispiel 1a zeigt, ist die Iterationsvariable i eigentlich überflüssig.

Für Iterationen über Felder kann daher auf die Zählvariable verzichtet werden.

Vereinfachtes for

Mit der erweiterten for-Schleife kann der Durchschnitt einer Messreihe wie folgt ermittelt werden:

Beispiel 1b

```
// Beispiel für Iterieren über Felder

public class DemoFor
{
 public static void main (String args[])
 {
   double summe = 0;
   double durchschnitt;
   double messreihe[] = {1.0, 5.5, 3.72, 1.2, 7.89};
   int anzahl = messreihe.length;
   for (double feldelement : messreihe)
      summe = summe + feldelement;
   System.out.println("Durchschnitt: " + summe / anzahl);
 }
}
```

DemoFor2

Das erweiterte for liefert alle Elemente des Feldes. Es ist aber *nicht* möglich, damit den Wert von Elementen im Feld schreibend, z. B. auf einen Anfangswert, zu setzen.

Sollen die Elemente eines Feldes auf einen Anfangswert gesetzt werden, dann sind folgende Code-Zeilen erforderlich:

Beispiel

```
tabelle = new int[MAX];
//Initialisierung einer Tabelle
for(int i=0; i < MAX; i++)
  tabelle[i] = -1; //Schreibender Zugriff
```

Die Syntax lautet:

Syntax

EnhancedForStatement ::=

for (*Type Identifier* : *Expression*) *Statement*

Expression muss ein Feld sein. Anstelle von for liest man am besten foreach bzw. (für) alle und anstelle des Doppelpunkts liest man am besten in, d. h. »Durchlaufe alle Elemente in dem Feld und tue folgendes« bzw. »Für alle Elemente in dem Feld tue folgendes«.

5.8 Aufzählungen mit enum ***

In Java können mit enum eigene Aufzählungstypen definiert werden, z. B. enum Ampel {Rot, Gelb, Gruen}. Die aufgeführten Werte werden als Konstanten angesehen, daher können enum-Konstanten in switch-Anweisungen verwendet werden. Mit einer erweiterten for-Schleife kann eine Aufzählung durchlaufen werden. Die Deklaration von eigenen Aufzählungen erfolgt innerhalb einer Klasse, aber *nicht* in Methoden, sondern gleichrangig neben Methoden.

Die einfachste Art, einen **Typ** zu beschreiben, d. h. seinen Wertebereich zu definieren, besteht in der Aufzählung der einzelnen Werte.

Beispiele

- Familienstand {ledig, verheiratet, geschieden, verwitwet}
- Geschlecht {weiblich, maennlich}
- Tage {Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag}

Problemnahe Beschreibung

Die Beispiele zeigen deutlich, dass es viele Anwendungsbereiche gibt, wo allein solche Aufzählungen eine anwendungsgerechte Formulierung ermöglichen.

Beispiel

```
// Beispiel für einen eigenen Aufzählungstyp
public class DemoAmpel
{
 enum Ampel {Rot, Gelb, Gruen}
 public static void main(String args[])
 {
  for (Ampel eineLampe : Ampel.values())
  //mit .values() wird auf einen Aufzählungswert zugegriffen
   System.out.println("Lampe: " + eineLampe);
 }
}
```

Der Programmlauf ergibt:

```
Lampe: Rot
Lampe: Gelb
Lampe: Gruen
```

Syntax

Im einfachsten Fall wird in Java eine Aufzählung mit dem Schlüsselwort enum begonnen. Danach folgt ein selbst vergebener Typnamen, z. B. Ampel. In geschweiften Klammern werden anschlie-

ßend die Werte der Aufzählung, durch Kommata getrennt, aufgeführt. Die aufgeführten Werte werden als Konstanten angesehen. Die Typdeklaration erfolgt in der Klasse, aber außerhalb der Methoden. Eine Typdeklaration steht gleichrangig mit den Methoden.

Konvention

Selbst gewählte Typnamen beginnen in Java immer mit einem Großbuchstaben, hier `Ampel`. Der Typname wird immer im Singular angegeben.

In switch einsetzbar

`enum`-Konstanten können in `switch`-Anweisungen verwendet werden, da sie intern über eine ganze Zahl identifiziert werden. Diese Zahl wird vom Compiler für die Aufzählung eingesetzt. Der Compiler übersetzt die `enum`-Konstanten in `final static int`.

Beispiel

Glauben

```
// Beispiel für Aufzählungen in Java
// Erstellen einer Konfessionsstatistik

public class Glauben
{
enum Konfession {Evangelisch, Katholisch, Sonstige, Keine}

 public static void main (String args[])
 {
   //Konfessionsstatistik
   int zaehler_ev = 0, zaehler_kath = 0,
       zaehler_sonst = 0, zaehler_keine = 0;

   Konfession[] person =
    {Konfession.Evangelisch, Konfession.Katholisch,
     Konfession.Evangelisch, Konfession.Sonstige,
     Konfession.Keine, Konfession.Keine};

   for (Konfession einePerson : person)
   {
    switch (einePerson)
    {
     case Evangelisch: zaehler_ev ++; break;
     case Katholisch: zaehler_kath ++; break;
     case Sonstige: zaehler_sonst ++; break;
     case Keine: zaehler_keine ++; break;
    }
   }

   System.out.println
       ("Anzahl evangelisch: " + zaehler_ev);
   System.out.println
       ("Anzahl katholisch:  " + zaehler_kath);
   System.out.println
       ("Anzahl sonstige:    " + zaehler_sonst);
   System.out.println
       ("Anzahl keine:       " + zaehler_keine);
```

```
 }
}
```

Das Ergebnis sieht folgendermaßen aus:

```
Anzahl evangelisch: 2
Anzahl katholisch:  1
Anzahl sonstige:    1
Anzahl keine:       2
```

Erweitertes for

Wie bei Feldern kann auch hier die erweiterte for-Schleife eingesetzt werden, um über alle Elemente der Aufzählung zu iterieren (»Iteration über Felder: Die erweiterte for-Schleife«, S. 198).

5.9 Vom Problem zur Lösung: Teil 3 **

Sind bei einer Problemlösung nur einfache Typen von Variablen notwendig, dann konzentriert sich die Lösungssuche in der Regel auf das Finden geeigneter Algorithmen in Form von Kontrollstrukturen (siehe »Vom Problem zur Lösung: Teil 2«, S. 164) und einfachen Anweisungen (siehe »Vom Problem zur Lösung: Teil 1«, S. 95).

Werden für die Problemlösung **Datenstrukturen**, zum Beispiel in Form von Feldern, benötigt, dann muss oft zuerst überlegt werden, wie die Datenstrukturen für die Problemlösung aussehen, bevor die Algorithmen dafür entwickelt werden. Die Perspektive verlagert sich also von den Algorithmen auf die Datenstrukturen. Letztendlich müssen natürlich beide optimal aufeinander abgestimmt werden. Es sollte folgende **Entscheidungsreihenfolge** eingehalten werden:

1. Zuerst die **Datenstrukturen** überlegen.
2. Dann ermitteln, mit welchen **Algorithmen** die Datenstrukturen am besten bearbeitet werden können.

Beispiel

Sie sollen folgendes Problem lösen: Von einem eingelesenen deutschen Text, der nur Zeichen aus dem ASCII-Code enthält, soll die Häufigkeit der einzelnen Zeichen festgestellt und ausgegeben werden. Zunächst ist festzustellen, dass die gewünschte Problemlösung *nicht* ausreichend spezifiziert ist. Es ist nicht klar, ob der deutsche Text über die Tastatur eingegeben oder aus einer Datei gelesen werden soll und über welche maximale Länge er verfügt. Für dieses Beispiel wird davon ausgegangen, dass der Text über die Konsole mit einer maximalen Länge einer Zeile eingelesen wird.

Um das Problem in Java lösen zu können, müssen Sie wissen, wie Zeichen in Java behandelt und gespeichert werden und wie der Zusammenhang mit ganzen Zahlen aussieht.

Mögliche Lösungsschritte sind:

Semiformale Lösung 1. Schritt

Beim Einlesen des Textes über die Tastatur gibt es mehrere Alternativen. Es kann jedes Zeichen einzeln für sich gelesen und in einer `char`-Variablen gespeichert werden, es kann die ganze Zeile gelesen und als Zeichenkette (`String`) oder als Zeichen-Feld (`char[]`) gespeichert werden. Da bei diesem Problem einzelne Zeichen betrachtet werden sollen, wird aus Verständnisgründen ein Zeichen-Feld gewählt. Auf das Lesen einzelner Zeichen wird verzichtet, da dies nicht laufzeiteffizient ist.

Formale Java-Lösung

```
System.out.println
  ("Textzeile bitte eingeben (nur ASCII-Zeichen):");
char[] text = Console.readCharArray();
int laenge = text. length;
```

Semiformale Lösung 2. Schritt

Als Datenstruktur zum Zählen der Häufigkeit eignet sich ein ganzzahliges Feld. Jedes gelesene Zeichen wird in eine ganze Zahl gewandelt und dient als Index für dieses Feld. Es gibt 128 ASCII-Zeichen, so dass ein Feld der Länge 128 benötigt wird. Das Feld muss am Anfang mit Wert 0 initialisiert werden (In Java *nicht* notwendig, da jedes Element eines Feldes mit dem jeweiligen Standardwert initialisiert wird, bei `int` mit `0`).

Formale Java-Lösung

```
//Datenstruktur zum Speichern der Häufigkeit
int[] anzahlZchnFeld = new int[128];
//Initialisierung mit Null-Werten, kann entfallen
for (int i = 0; i < anzahlZchnFeld.length; i++)
  anzahlZchnFeld[i] = 0;
```

Semiformale Lösung 3. Schritt

Das eingelesene Textfeld muss durchlaufen werden. Das jeweilige Zeichen wird in einen `int`-Wert gewandelt und dient als Index für das Häufigkeitsfeld. Der Inhalt wird um 1 erhöht. Zeichen mit einem ASCII-Wert $\geq$ 128 werden als fehlerhaft erkannt.

Formale Java-Lösung

```
for (int i = 0; i < laenge; i++)
{
  //ASCII-Zeichen >= 128 ignorieren, z.B. Umlaute
  if((int)text[i]<128)
  {
    System.out.println("Zchn [" + i + "] = "
      + text[i] + " ASCII-Wert = " + (int)text[i] );
    //Inhalt um 1 erhöhen
    anzahlZchnFeld[(int)text[i]]++;
 }
```

```
 else
 {
    System.out.println("Zchn [" + i + "] = "
      + text[i] + " = Ungültiges Zeichen");
  }
}
```

4. Schritt

Semiformale Lösung

Das Häufigkeitsfeld muss durchlaufen werden. Immer wenn der Inhalt für ein Zeichen größer 0 ist, muss der Wert ausgegeben werden.

Formale Java-Lösung

```
for (int i = 0; i < anzahlZchnFeld.length; i++)
  if (anzahlZchnFeld[i] > 0)
    System.out.println("Das Zeichen " + (char)i + " ist "
      + anzahlZchnFeld[i] + " mal im Text vorhanden");
```

Das gesamte Programm sieht wie folgt aus:

```
/**
 * Häufigkeit eingelesener Zeichen zählen
 *
 * @author Helmut Balzert
 * @version V1.0
 */
import inout.Console;

public class Zchnzaehlen
{
    public static void main (String args[])
    {
      System.out.println
        ("Textzeile bitte eingeben (nur ASCII-Zeichen):");
      char[] text = Console.readCharArray();
      int laenge = text. length;
      //Datenstruktur zum Speichern der Häufigkeit
      int[] anzahlZchnFeld = new int[128];
      //Initialisierung mit Null-Werten
      for (int i = 0; i < anzahlZchnFeld.length; i++)
        anzahlZchnFeld[i] = 0;

      for (int i = 0; i < laenge; i++)
      {
        //ASCII-Zeichen >= 128 ignorieren, z.B. Umlaute
        if((int)text[i]<128)
        {
          System.out.println("Zchn [" + i + "] = "
           + text[i] + " ASCII-Wert = "
           + (int)text[i] );
           //Inhalt um 1 erhöhen
           anzahlZchnFeld[(int)text[i]]++;
        }
        else
        {
          System.out.println("Zchn [" + i + "] = "
```

```
            + text[i] + " = Ungültiges Zeichen");
        }
      }
      for (int i = 0; i < anzahlZchnFeld.length; i++)
         if (anzahlZchnFeld[i] > 0)
          System.out.println("Das Zeichen " + (char)i + " ist "
            + anzahlZchnFeld[i] + " mal im Text vorhanden");
    }
}
```

Ein Beispiellauf ergibt folgende Ausgabe:

```
Textzeile bitte eingeben (nur ASCII-Zeichen):
Dies ist ein ASCII-Text!
Zchn [0] = D ASCII-Wert = 68
Zchn [1] = i ASCII-Wert = 105
Zchn [2] = e ASCII-Wert = 101
Zchn [3] = s ASCII-Wert = 115
Zchn [4] =   ASCII-Wert = 32
Zchn [5] = i ASCII-Wert = 105
Zchn [6] = s ASCII-Wert = 115
Zchn [7] = t ASCII-Wert = 116
Zchn [8] =   ASCII-Wert = 32
Zchn [9] = e ASCII-Wert = 101
Zchn [10] = i ASCII-Wert = 105
Zchn [11] = n ASCII-Wert = 110
Zchn [12] =   ASCII-Wert = 32
Zchn [13] = A ASCII-Wert = 65
Zchn [14] = S ASCII-Wert = 83
Zchn [15] = C ASCII-Wert = 67
Zchn [16] = I ASCII-Wert = 73
Zchn [17] = I ASCII-Wert = 73
Zchn [18] = - ASCII-Wert = 45
Zchn [19] = T ASCII-Wert = 84
Zchn [20] = e ASCII-Wert = 101
Zchn [21] = x ASCII-Wert = 120
Zchn [22] = t ASCII-Wert = 116
Zchn [23] = ! ASCII-Wert = 33
Das Zeichen   ist 3 mal im Text vorhanden
Das Zeichen ! ist 1 mal im Text vorhanden
Das Zeichen - ist 1 mal im Text vorhanden
Das Zeichen A ist 1 mal im Text vorhanden
Das Zeichen C ist 1 mal im Text vorhanden
Das Zeichen D ist 1 mal im Text vorhanden
Das Zeichen I ist 2 mal im Text vorhanden
Das Zeichen S ist 1 mal im Text vorhanden
Das Zeichen T ist 1 mal im Text vorhanden
Das Zeichen e ist 3 mal im Text vorhanden
Das Zeichen i ist 3 mal im Text vorhanden
Das Zeichen n ist 1 mal im Text vorhanden
Das Zeichen s ist 2 mal im Text vorhanden
Das Zeichen t ist 2 mal im Text vorhanden
Das Zeichen x ist 1 mal im Text vorhanden
```

Verallgemeinerung

Die gewählte Lösung kann wie folgt verallgemeinert werden:

Die Texte könnten alternativ über eine Datei eingelesen werden (Programm muss entsprechend modifiziert werden) und aus beliebig vielen Textzeilen bestehen. Bei umfangreichen Texten muss überlegt werden, ob der Datentyp des Feldes `anzahlZchnFeld` in `long` gewandelt wird, damit bei der Anzahl der aufgetretenen Zeichen kein Speicherüberlauf eintritt.

Erweiterbarkeit

Die gewählte Lösung kann wie folgt erweitert werden:

Es können alle Zeichen aus dem Latin-1-Zeichensatz oder alle Zeichen des Unicodes zugelassen werden. Das Feld `anzahlZchnFeld` muss dann entsprechend vergrößert werden.

Tipp

Es ist immer hilfreich, während der Programmentwicklung Testausgaben in das Programm einzufügen, um Fehler frühzeitig zu finden.

6 Prozeduren, Funktionen und Methoden *

Auslagerung von Teilaufgaben

Ein wichtiger Fortschritt wurde bei Programmiersprachen erreicht, als es gelang, unabhängige Teilaufgaben für sich zu formulieren und sie von verschiedenen Programmen aus aufzurufen.

Im einfachsten Fall werden mehrere Anweisungen zu einer Einheit – einer sogenannten Prozedur – zusammengefasst und mit einem Namen versehen. Eine solche Prozedur kann dann von einem anderen Programm durch Angabe des Namens aufgerufen, d. h. zur Ausführung veranlasst werden:

- »Parameterlose Prozeduren«, S. 208

Will man eine Prozedur für verschiedene Einsatzzwecke verwenden, dann ist es sinnvoll sie zu verallgemeinern. Eine mögliche Verallgemeinerung besteht darin, die Prozedur mit sogenannten Eingabeparametern zu versehen. In Abhängigkeit von den Parametern kann die Prozedur sich dann unterschiedlich verhalten:

- »Prozeduren mit Eingabeparametern«, S. 213

In manchen Anwendungsfällen, z. B. bei Sortieraufgaben, ist es erforderlich als Eingabeparameter Felder zu übergeben:

- »Felder als Eingabeparameter«, S. 219

In der Regel sollen von einer Prozedur auch Ergebnisse an den Aufrufer zurückgegeben werden. Wird nur ein Ergebnisparameter zurückgegeben, dann spricht man von einer Funktion:

- »Funktionen und Ausgabeparameter«, S. 223

Java stellt viele Funktionen zusammengefasst in Klassen zur Benutzung zur Verfügung:

- »Java-Funktionen nutzen«, S. 225

Sind mehrere Ergebnisse an eine aufrufende Methode zurückzugeben, dann kann dies in Java durch ein Feld als Ergebnisparameter erfolgen:

- »Felder als Ergebnisparameter«, S. 230

Für spezielle Zwecke ist es sogar möglich, variable Parameterlisten anzugeben:

- »Variable Parameterlisten«, S. 232

Damit gleichartige Methoden, die sich nur durch eine unterschiedliche Parameteranzahl unterscheiden, nicht unterschiedliche Namen bekommen müssen, können Methoden »überladen« werden:

- »Überladen von Methoden«, S. 233

Werden ausgehend von der `main`-Methode andere Methoden aufgerufen, dann verliert man leicht den Überblick, in welcher Reihenfolge die Methoden ausgeführt werden. Ein Hilfsmittel, um sich einen Überblick über den dynamischen Ablauf zu verschaffen, sind spezielle Diagramme:

- »UML-Sequenzdiagramme«, S. 236

Probleme, die sich auf gleichartige, einfachere Teilprobleme zurückführen lassen, können »rekursiv« programmiert werden:

- »Rekursion«, S. 239

Ein bekanntes Beispiel für ein rekursives Problem sind die »Türme von Hanoi«. An diesem Beispiel kann man gut sehen, wie der Aufwand eines Programms aussieht:

- »Rekursion: Türme von Hanoi«, S. 244

Neben der direkten Rekursion gibt es auch eine indirekte Rekursion:

- »Rekursion: direkt vs. indirekt«, S. 250

Stehen in einer Klasse mehrere Methoden, die auf gemeinsame Daten zugreifen müssen, dann werden diese Daten *nicht* einer Methode zugeordnet, sondern die Variablen und Konstanten werden der Klasse zugeordnet:

- »Datenabstraktion: Gemeinsame Daten«, S. 252

Mit den Möglichkeiten, die Methoden und die Datenabstraktion bieten, ist es möglich, die Fallstudie OptiTravel fertigzustellen:

- »OptiTravel: Gesamtlösung«, S. 258

Die Beherrschung dieser Grundkonzepte und ihre Realisierung in Java ermöglicht es Ihnen, bereits anspruchsvollere Programme zu schreiben.

6.1 Parameterlose Prozeduren *

In sich abgeschlossene, unveränderliche Teilprobleme können in den meisten Programmiersprachen in Form von parameterlosen Prozeduren bzw. Methoden beschrieben bzw. deklariert werden. Die so deklarierten Teilprobleme erhalten einen eigenen Namen (Prozedurnamen) und werden analog wie die Prozedur `main` in Java innerhalb einer Klasse deklariert. Diese Prozeduren können dann von anderen Programmen aus aufgerufen werden. Sie werden dann ausgeführt und nach Ende der Ausführung wird das rufende Programm hinter der Aufrufstelle fortgesetzt.

Problem

Wenn Sie Programme schreiben, dann werden Sie feststellen, dass Teile Ihres Programms im selben Programm oder in anderen eigenen oder fremden Programmen in gleicher oder ähnlicher Weise mehrfach auftreten.

Beispiel 1a

Sie wollen eine Artikelliste auf dem Bildschirm ausgeben. Um die Lesbarkeit zu verbessern, soll zwischen den Artikeln jeweils eine Linie dargestellt werden. Ein solches Programm kann folgendermaßen aussehen:

Artikel-liste

```
//Ausgeben einer Artikelliste in das Konsolenfenster
public class Artikelliste
{
 public static void main (String args[])
 {
  System.out.println
   ("Artikel-Nr         Artikelbezeichnung    Preis [€]");
  System.out.println
   ("_____________________________________________________");
  System.out.println
   ("978-3-937137-08-7  Java: Objektorientiert    24,90");
  System.out.println
   ("_____________________________________________________");
  System.out.println
   ("978-3-937137-16-2  SQL                       19,90");
  System.out.println
   ("_____________________________________________________");
  System.out.println
   ("978-3-937137-02-5  Webdesign & Web-Ergonomie 24,90");
  System.out.println
   ("_____________________________________________________");
 }
}
```

Die Programmzeile

```
System.out.println
  ("_____________________________________________________");
```

tritt viermal auf.

Ändert sich z. B. die Länge der Linie, weil beispielsweise die Lagermenge noch aufgeführt werden soll, dann müssen alle vier Zeilen geändert werden.

Lösung

Um das wiederholte Schreiben bzw. Kopieren solcher oft benötigter Programmteile zu vermeiden, hat man nach Möglichkeiten gesucht, diese nur einmal zu schreiben und dann beliebig oft in eigenen oder verschiedenen Programmen einzusetzen bzw. anzuwenden.

Parameterlose Prozeduren

Die einfachste Möglichkeit, eine in sich abgeschlossene, unveränderliche Teilaufgabe nur einmal hinzuschreiben, aber beliebig oft anzuwenden, besteht darin, eine sogenannte parameterlose **Prozedur** zu deklarieren – in Java spricht man von einer para-

meterlosen **Methode**. Analog wie bei Variablen, müssen Methoden, Prozeduren und Funktionen in einem Programm deklariert bzw. vereinbart werden.

Vergabe eines Prozedurnamens

Die Anweisung oder die Anweisungsfolge, die als eigenständige Teilaufgabe vereinbart werden soll, wird in Java in geschweifte Klammern eingeschlossen und erhält einen Prozedur- bzw. Methodennamen – analog wie die `main`-Methode. Hinter den Methodennamen wird ein leeres rundes Klammerpaar geschrieben. Vor dem Methodennamen muss `static void` stehen. Methodendeklarationen können in Java vor oder hinter der `main`-Methode stehen.

Beispiel 1b

Im Beispiel 1a kann die Anweisung

```
System.out.println
 ("_____");
```

als eine Methode bzw. Prozedur formuliert werden.

Artikel-liste2

```
//Ausgeben einer Artikelliste in das Konsolenfenster
//Linie_ausgeben als parameterlose Prozedur
public class Artikelliste2
{
 public static void main (String args[])
 {
  System.out.println
   ("Artikel-Nr         Artikelbezeichnung    Preis [€]");
  Linie_ausgeben(); //1. Aufruf
  System.out.println
   ("978-3-937137-08-7  Java: Objektorientiert    24,90");
  Linie_ausgeben(); //2. Aufruf
  System.out.println
   ("978-3-937137-16-2  SQL                       19,90");
  Linie_ausgeben(); //3. Aufruf
  System.out.println
   ("978-3-937137-02-5  Webdesign & Web-Ergonomie 24,90");
  Linie_ausgeben(); //4. Aufruf
 }
 //Name der Methode bzw. Prozedur
 public static void Linie_ausgeben()
 {
   //Rumpf der Methode bzw. Prozedur
   System.out.println("_____"
      + "_____");
 }
}
```

Die Anweisung, die die Linie ausgibt, wird in ein Programm, genauer gesagt in eine Methode bzw. Prozedur, mit dem Namen `Linie_ausgeben` ausgelagert.

Der Name `Linie_ausgeben` steht jetzt stellvertretend für die Anweisung `System.out.println("_____");`

Deklaration

Die Zuordnung eines Methoden- bzw. Prozedurnamens zu einer Anweisung bzw. einer Anweisungsfolge geschieht in der Methoden- bzw. Prozedurdeklaration. Die Deklaration besteht aus einem Methoden- bzw. Prozedurkopf und einem Methoden- bzw. Prozedurrumpf.

Soll die Methode auch von anderen Programmen außerhalb der Klasse aufrufbar sein, dann steht am Anfang der Methodendeklaration das Schlüsselwort `public`.

Soll eine Methode nur von anderen Methoden der eigenen Klasse nutzbar sein, dann steht am Anfang der Methodendeklaration das Schlüsselwort `private`.

Aufruf

Die Anwendung der so vereinbarten Anweisungsfolge geschieht durch einen sogenannten Methoden- bzw. Prozeduraufruf. Aufruf bedeutet, dass an der Stelle eines Programms, an der die deklarierte Anweisungsfolge ausgeführt werden soll, anstelle der Anweisungsfolge der Methoden- bzw. Prozedurname angegeben wird.

Beispiel 1c

Im Beispiel 1b wird die Methode bzw. Prozedur `Linie_ausgeben` an vier Stellen aufgerufen. Die Abarbeitung eines Programms mit Methoden- bzw. Prozeduraufrufen geschieht folgendermaßen:

Die Ausführung des Programms `main` beginnt mit der ersten Anweisung

```
System.out.println
 ("Artikel-Nr         Artikelbezeichnung    Preis [€]");
```

An der Stelle, an der ein Methoden- bzw. Prozedurname steht, verzweigt der Ablauf zum Methoden- bzw. Prozedurrumpf der entsprechenden Methode bzw. Prozedur.

Nach dem Durchlaufen des Methoden- bzw. Prozedurrumpfes wird die Ausführung hinter dem Methoden- bzw. Prozedurnamen, d. h. hinter der Aufrufstelle, im Anweisungsteil des aufrufenden Programms fortgesetzt.

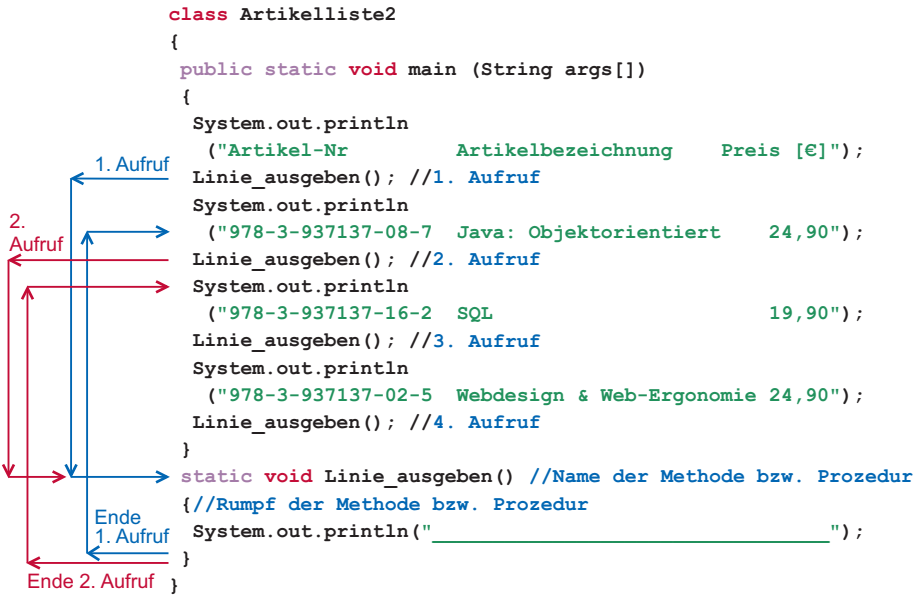
Den dynamischen Ablauf zeigt die Abb. 6.1-1.

Syntax

Die Syntax für die Deklaration einer parameterlosen Prozedur sieht in Java folgendermaßen aus:

MethodDeclaration ::=
[`public`*|* `private`*] [* `static` *]* `void` *MethodIdentifier*`()`
`{` *MethodBody* `}`

Wie die Syntax zeigt, können am Anfang der Deklaration noch optional die Schlüsselwörter `public` oder `private` stehen.




```
class Artikelliste2
{
 public static void main (String args[])
 {
  System.out.println
   ("Artikel-Nr          Artikelbezeichnung     Preis [€]");
  Linie_ausgeben(); //1. Aufruf
  System.out.println
   ("978-3-937137-08-7  Java: Objektorientiert    24,90");
  Linie_ausgeben(); //2. Aufruf
  System.out.println
   ("978-3-937137-16-2  SQL                       19,90");
  Linie_ausgeben(); //3. Aufruf
  System.out.println
   ("978-3-937137-02-5  Webdesign & Web-Ergonomie 24,90");
  Linie_ausgeben(); //4. Aufruf
 }
static void Linie_ausgeben() //Name der Methode bzw. Prozedur
{//Rumpf der Methode bzw. Prozedur
 System.out.println("_____________________________________");
}
}
```

Abb. 6.1-1: In dem Programm `Artikelliste2` wird die Methode bzw. die Prozedur `Linie_ausgeben` mehrfach aufgerufen. Der Programmablauf für die ersten zwei Aufrufe ist hier zu sehen.

Wenn Sie `public` davor setzen, dann können auch andere Programme diese Prozedur benutzen. Bei `private` können nur Prozeduren innerhalb der eigenen Klasse die Prozedur aufrufen.

Fehlen beide Schlüsselwörter, dann können alle Programme innerhalb eines Pakets die Prozedur benutzen.

Innerhalb einer Klasse

Alle Prozeduren bzw. Methoden befinden sich in Java immer innerhalb einer Klasse und sind untereinander gleichberechtigt – mit Ausnahme der `main`-Methode, die immer zuerst ausgeführt wird:

ClassDeclaration ::=
[`public`*]* `class` *ClassIdentifier ClassBody*

ClassBody ::=
{ *MethodDeclaration+* }

Programmschema

Das Programmschema zeigt anschaulich die Abb. 6.1-2.

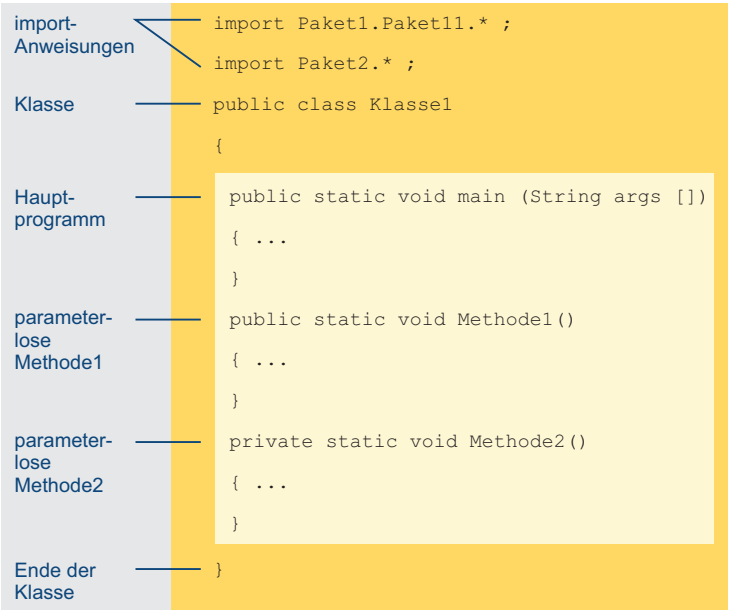


Abb. 6.1-2: Aufbau eines einfachen Java-Programmschemas mit mehreren parameterlosen Methoden.

6.2 Prozeduren mit Eingabeparametern *

Um mit Methoden bzw. Prozeduren nicht nur eine spezielle Aufgabe zu lösen, sondern möglichst eine Aufgabenklasse, ist es möglich, Informationen in Form von Eingabeparametern an das gerufene Programm zu übergeben. Auf einer formalen Parameterliste werden Platzhalter spezifiziert. Jeder formale Parameter wird durch seinen Typ und seinen Parameternamen deklariert. Beim Aufruf der Prozedur werden den formalen Parametern aktuelle Parameter gegenübergestellt, die die formalen Parameter initialisieren (falls es sich um einfache Typen handelt). Es erfolgt eine Wertübergabe *(call by value)*, d. h. der Wert des aktuellen Parameters wird in die Speicherzelle des formalen Parameters kopiert, sodass innerhalb der gerufenen Prozedur nur auf Kopien der aktuellen Parameter gearbeitet wird.

Eingabe-parameter

Eine Methode bzw. Prozedur ist umso besser wiederverwendbar, je allgemeiner sie ist. Allgemeiner bedeutet hier, ein möglichst flexibler Einsatz für verschiedene Anwendungsfälle. Parameterlose Prozeduren erlauben *keine* Flexibilisierung einer Aufgabe. Es gibt daher die Möglichkeit, Daten bzw. Informationen über sogenannte Eingabeparameter bzw. Argumente an eine Prozedur zu übergeben.

Beispiel 1a

Bei der Anzeige einer Artikelliste sollen nach jeder Artikelgruppe eine Leerzeile und nach jeder Artikelhauptgruppe zwei Leerzeilen ausgegeben werden. Die Listenüberschrift soll durch eine Linie bestehend aus Gleichheitszeichen und das Ende der Liste durch eine Linie bestehend aus Bindestrichen strukturiert werden. Anstatt nun Prozeduren zu schreiben, die genau die festgelegten Anforderungen erfüllen, werden zwei Prozeduren konzipiert, die sich auf wechselnde Anforderungen einstellen können.

Leerzeile_ausgeben

Die Prozedur `Leerzeile_ausgeben` erhält als Eingabeparameter eine Variable `Anzahl`, die angibt, wie viele Leerzeilen auszugeben sind.

Linie_ausgeben

Die Prozedur `Linie_ausgeben` erhält zwei Eingabeparameter. Der Parameter `laenge` gibt an, wieviele Zeichen auszugeben sind, der Parameter `Zeichen`, welches Zeichen auszugeben ist.

Es ergibt sich ein Programm mit der Methode `main` sowie zwei weiteren Methoden:

Artikelliste3

```
// Ausgeben einer Artikelliste in das Konsolenfenster
// Leerzeile_ausgeben mit Eingabeparameter anzahl
// Linie_ausgeben mit Eingabeparametern laenge und zeichen

public class Artikelliste3
{
 public static void main (String args[])
 {
  System.out.println
   ("Artikel-Nr         Artikelbezeichnung    Preis [€]");
  Linie_ausgeben(50,'='); //1. Aufruf Linie
  System.out.println
   ("978-3-937137-08-7  Java: Objektorientiert    24,90");
  Leerzeile_ausgeben(1); //1. Aufruf Leerzeile
  System.out.println
   ("978-3-937137-16-2  SQL                       19,90");
  Leerzeile_ausgeben(2); //2. Aufruf Leerzeile
  System.out.println
   ("978-3-937137-02-5  Webdesign & Web-Ergonomie 24,90");
  Linie_ausgeben(50,'-'); //2. Aufruf  Linie
 }
 public static void Leerzeile_ausgeben(int anzahl)
 {
   for (int zaehler = 1; zaehler <= anzahl; zaehler++)
     System.out.println(""); //1 Leerzeile
 }
 public static void Linie_ausgeben(int laenge, char zeichen)
 {
   for (int i = 1; i <= laenge; i++)
    System.out.print(zeichen);
   System.out.println();
 }
}
```

Ein Programmlauf führt zu folgender Ausgabe:

```
Artikel-Nr         Artikelbezeichnung    Preis [€]
=================================================
978-3-937137-08-7  Java: Objektorientiert    24,90

978-3-937137-16-2  SQL                       19,90


978-3-937137-02-5  Webdesign & Web-Ergonomie 24,90
-------------------------------------------------
```

Spezifikation der Eingabeparameter

Eingabeparameter werden in folgender Form spezifiziert:

- Bei der Methoden- bzw. Prozedurvereinbarung wird hinter dem Methodennamen eine Parameterliste – eingeschlossen in runde Klammern – aufgeführt. Im Beispiel:
  `Leerzeile_ausgeben(...)` und `Linie_ausgeben(..)`.
- Sollen Werte in eine Methode bzw. Prozedur übergeben werden, dann handelt es sich um Eingabeparameter. Für jeden Eingabeparameter wird sein Typ gefolgt von dem Parameterbezeichner angegeben. Im Beispiel:
  `Leerzeile_ausgeben(int Anzahl)`.
- Sollen mehrere Parameter übergeben werden, dann werden sie – jeweils durch Kommata getrennt – hintereinander aufgeführt. Im Beispiel:
  `Linie_ausgeben(int laenge, char zeichen)`.

Syntax

Die Syntax sieht in Java wie folgt aus:

MethodDeclaration ::=
[`public` | `private`*] [* `static` *]* `void` *MethodIdentifier*
`(` *[FormalParameterList]* `)` `{` *MethodBody* `}`

FormalParameterList ::= FormalParameter...
FormalParameter ::= Type Identifier

- Die Angabe eines **Parameters** in der Parameterliste bewirkt, dass (automatisch) eine entsprechende Variable vereinbart wird, d. h. es wird ein Speicherplatz reserviert. Diese Variable ist jedoch nur in der Methode bzw. Prozedur bekannt, in deren Parameterliste sie vereinbart ist. Im Beispiel: Die Parametervariable `anzahl` ist nur in der Methode `Leerzeile_ausgeben` bekannt. Die Parametervariablen `laenge` und `zeichen` sind nur in der Methode `Linie_ausgeben` bekannt. Die Variablen existieren jedoch nur, solange der Rumpf der Prozedur ausgeführt wird. Anschließend wird der Speicherplatz wieder für andere Zwecke benutzt. Man sagt daher, dass der **Gültigkeitsbereich der Parameter** auf die jeweilige Prozedur beschränkt ist.
- Die in einer Parameterliste einer Methoden- bzw. Prozedurvereinbarung aufgeführten Parameter werden als **formale**

Parameter bezeichnet. Ein formaler Parameter ist sozusagen ein **Platzhalter** oder **Stellvertreter** für den späteren aktuellen Wert.

- Es ist möglich, als Parameternamen in verschiedenen Methoden gleiche Namen zu verwenden. Es handelt sich dann jedoch um unterschiedliche Variablen – gültig nur in der jeweiligen Prozedur. Die Parameternamen innerhalb *einer* formalen Parameterliste müssen unterschiedlich sein!
- Innerhalb jeder Methoden- bzw. Prozedurvereinbarung können für die Methode bzw. Prozedur benötigte Variable zusätzlich vereinbart werden (im Beispiel `Zaehler` und `i`). Sie heißen **lokale Variable**, da sie nur innerhalb der Methode bzw. Prozedur gültig sind, in der sie vereinbart sind. Außerhalb der Methode bzw. Prozedur ist die lokale Variable nicht bekannt bzw. nicht sichtbar.
- Parameternamen dürfen `nicht` als lokale Variable innerhalb der Prozedur erneut deklariert werden, da die lokale Variable den Parameter sonst verbergen würde.

Aufruf

- Beim Aufruf der Methode bzw. der Prozedur wird in runden Klammern angegeben, welcher aktuelle Wert an den formalen Parameter übergeben werden soll, d. h. mit welchem Wert die Parametervariable initialisiert werden soll. Die Parameter, die beim Aufruf angegeben sind, werden daher als **aktuelle Parameter** oder **Argumente** bezeichnet. Im Beispiel: Beim ersten Aufruf von `Linie_ausgeben` sind die aktuellen Parameter `50` und `2`, d. h. der formale Parameter `laenge` wird mit dem Wert `50`, der formale Parameter `zeichen` mit dem Wert `'='` initialisiert.

Beispiel 1b

Die Übergabe des aktuellen Parameters `2` an den formalen Parameter `anzahl` ist in der Abb. 6.2-1 dargestellt.

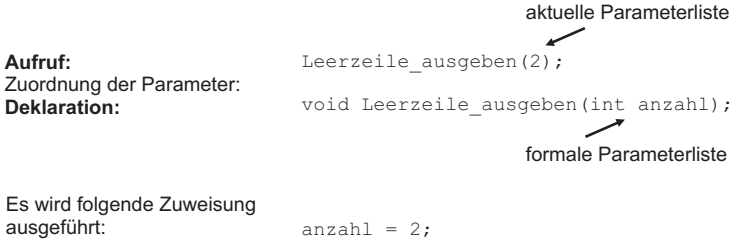


Abb. 6.2-1: Die Methode bzw. Prozedur `Leerzeile_ausgeben` wird mit dem aktuellen Parameter `2` aufgerufen. Dieser Wert wird der Parametervariablen `Anzahl` auf der formalen Parameterliste zugewiesen.

- Die Parameterübergabe erfolgt in der Form, dass die aktuelle Parameterliste der formalen Parameterliste gegenüberge-

stellt wird. Der Wert des jeweiligen aktuellen Parameters wird der Variablen der zugeordneten formalen Parameterliste zugeordnet. Dies entspricht einer normalen Zuweisung (im Beispiel: `anzahl = 1`; beim 1. Aufruf von `Leerzeile_ausgeben` bzw. `anzahl = 2` beim 2. Aufruf von `Leerzeile_ausgeben`).

- Enthält eine Parameterliste mehr als einen Parameter, dann erfolgt die Zuordnung zwischen formalen und aktuellen Parametern entsprechend ihrer Position. Das heißt, der erste aufgeführte formale Parameter wird dem ersten aufgeführten aktuellen Parameter zugeordnet, der zweite formale Parameter dem zweiten aktuellen usw. (**Positionszuordnung**). Im Beispiel: Beim Aufruf von `Linie_ausgeben(50,'=');` wird dem 1. formalen Parameter `laenge` der erste aktuelle Parameter `50` und dem 2. formalen Parameter `zeichen` der zweite aktuelle Parameter `'='` zugewiesen.

Syntax

Die Syntax für einen **Prozeduraufruf** sieht in Java folgendermaßen aus:

```
MethodInvocation ::= MethodName ( [ ArgumentList ] )
ArgumentList ::= Expression...
```

- Auf der aktuellen Parameterliste kann anstelle eines Literals – z. B. die Zahl `40` – auch eine Variable oder Konstante stehen, z. B. `zeilenlaenge`. Steht auf der aktuellen Parameterliste als ein Argument ein Ausdruck, z. B. `10 * 4` oder `spaltenbreite * 4`, dann wird dieser Ausdruck zunächst ausgewertet und dann an den formalen Parameter übergeben.
- Stimmen die Typen der aktuellen und der formalen Parameter nicht überein, dann sind Typumwandlungen erforderlich (siehe: »Typumwandlungen«, S. 91). Es liegt dieselbe Situation vor wie bei der Zuweisung eines berechneten Wertes aus einem Ausdruck an eine Variable. Typausweitungen werden automatisch vorgenommen. Typeinengungen sind explizit anzugeben.

Typumwandlungen

- In Java müssen die Anzahl der aktuellen und der formalen Parameter immer übereinstimmen. Eine Ausnahme sind variable Parameterlisten (siehe: »Variable Parameterlisten«, S. 232).

call by value

Steht auf der Parameterliste eine Variable, die einen einfachen Typ besitzt, dann wird als sogenannter **Parameterübergabemechanismus** eine **Wertübergabe** *(call by value)* vorgenommen. Aufgabe eines Eingabeparameters ist es, Informationen in eine Prozedur zu transportieren. Die aufgerufene Prozedur soll *keine* Möglichkeit haben, den aktuellen Wert im aufrufenden Programm zu ändern.

Bei *call by value* verhält sich der formale Parameter wie eine lokale Variable, die durch den Wert des aktuellen Parameters initialisiert wird (siehe Abb. 6.2-1). Bei diesem Parameterübergabemechanismus arbeitet die gerufene Prozedur nur mit der Kopie der übergebenen Informationen. Änderungen an dieser Kopie haben keine Rückwirkungen auf den aktuellen Parameter.

Beispiel

Demo Wertuebergabe

```
public class DemoWertuebergabe
{
 public static void main (String args[])
 {
  int anzahl = 20; char linienart = '=';
  System.out.println("Werte vor Aufruf");
  System.out.println("Anzahl: " + anzahl + " Linienart: "
    + linienart);
  Linie_ausgeben(anzahl, linienart);
  System.out.println("Werte nach Aufruf");
  System.out.println("Anzahl: " + anzahl + " Linienart: "
    + linienart);
 }
 public static void Linie_ausgeben(int laenge, char zeichen)
 {
   for (int i = 1; i <= laenge; i++)
    System.out.print(zeichen);
   System.out.println();
   laenge = 10; zeichen ='+';
 }
}
```

Die Ausführung des Programms ergibt folgende Ausgabe:

```
Werte vor Aufruf
Anzahl: 20 Linienart: =
====================
Werte nach Aufruf
Anzahl: 20 Linienart: =
```

Wie die Ausführung zeigt, haben die Zuweisungen `laenge = 10;` `zeichen ='+';` am Ende der Prozedur `Linie_ausgeben` keine Rückwirkungen auf die aktuellen Parameterwerte von `Anzahl` und `linienart` in der rufenden Prozedur `main`.

Eine Klasse mit all ihren Methoden – in der UML spricht man von Operationen – kann in der grafischen **UML**-Notation durch ein zweigeteiltes Rechteck dargestellt werden. Im oberen Rechteckteil steht der Klassenname. Er wird immer fett gedruckt und zentriert dargestellt. Er beginnt mit einem Großbuchstaben. Im unteren Rechteckteil stehen die Methoden, im einfachsten Fall nur mit ihren Namen, gefolgt von einem runden Klammerpaar. Eine Methode kann allerdings auch mit allen Parametern darge-

stellt werden. Die Syntax für eine Methode mit Eingabeparametern sieht aber etwas anders aus als in Java:

Operation(in *Parameter1*: *Typ*, in *Parameter2*: *Typ*):void

Alle Methoden, die mit static gekennzeichnet sind, werden unterstrichen dargestellt – solche Methoden heißen **Klassenmethoden**. Die Abb. 6.2-2 zeigt die UML-Darstellung des Beispiels 1a in der Kurz- und in der Langform.

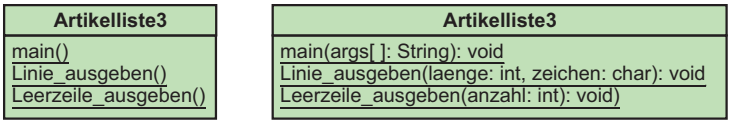


Abb. 6.2-2: UML-Darstellung der Klasse Artikelliste3 mit 3 Methoden (Kurz- und Langform).

6.3 Felder als Eingabeparameter *

Müssen bei einem Aufruf einer Prozedur Felder als Eingabeparameter übergeben werden, dann wird *keine* Kopie aller Feldelemente vorgenommen. Es wird vielmehr von der Feldvariablen der aufrufenden Prozedur nur eine Kopie der Speicheradresse auf die Feldelemente in die formale Feldvariable der gerufenen Prozedur vorgenommen. Alle Zugriffe innerhalb der gerufenen Prozedur – sowohl lesend als auch schreibend – geschieht über die kopierte Speicherreferenz direkt auf die Originaldaten des Feldes. Dieser Parameterübergabemechanismus heißt *call by reference*, genauer *passing a reference by value*.

Für manche Anwendungen müssen Felder an eine Prozedur als Eingabeparameter übergeben werden. Dies ist in Java möglich.

Beispiel

Die main-Prozedur in Java sieht immer wie folgt aus:

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

Sie enthält auf der Parameterliste das Feld args vom Typ String[]. Dieses Feld dient dazu, beim Start des Programms über die Konsole, aktuelle Parameter an das Programm zu übergeben. In Java heißen diese Parameter *Command-Line Arguments* (siehe auch: »Stapelverarbeitungsprogramme: .bat-Dateien «, S. 287). args ist ein Feldnamen. Sie können natürlich auch eine beliebigen anderen Namen wählen, z. B. argumente.

DemoFeld Parameter

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

```
  for (int i = 0; i < args.length; i++)
     System.out.println
       ("Uebergebenes Argument[" + i +"]: " + args[i]);
 }
}
```

Bei einer Konsoleneingabe werden hinter dem Dateinamen, getrennt durch Leerzeichen, die aktuellen Parameter eingegeben:

```
C:\Java>java DemoFeldParameter Balzert Helmut
Uebergebenes Argument[0]: Balzert
Uebergebenes Argument[1]: Helmut
```

In BlueJ können ebenfalls Parameter an die `main`-Prozedur übergeben werden. Sie werden wie folgt angegeben: `{"Balzert","Helmut"}`.

Felder als Ein- & Ausgabeparameter

Sortierprogramme arbeiten in der Regel auf Feldern und sortieren die Informationen innerhalb des Feldes. Wird ein Feld als Eingabeparameter vereinbart, dann steht nach dem Sortiervorgang im selben Feld auch das Ergebnis. In diesem Fall wirkt das Feld gleichzeitig als Ein- und Ausgabeparameter.

Beispiel 1a

SortAuswahl2

Das Sortierverfahren »Sortieren durch Auswahl« (siehe: »Einfaches Sortieren«, S. 195) lässt sich leicht als Prozedur formulieren:

```
public class SortAuswahl2
{
 public static void main (String args[])
 {
  String[] kunden = {"Meyer","Schulz", "Balzert",
      "Maier","Peters", "Meyer"};
  sortieren(kunden);//Aufruf
  System.out.println("Lexikographisch sortierte Namensliste");
  for (int i = 0; i < kunden.length; i++)
       System.out.println(kunden[i]);
 }
 public static void sortieren(String[] feld)
 {
  String min, merke;
  int pos, posMin;
  //Sortieren und Vertauschen
  for (int i = 0; i < feld.length; i++)
  {
    //Kleinste Position ab i suchen
    posMin = i; min = feld[i];
    for (pos = i + 1; pos < feld.length; pos++)
      if (feld[pos].compareTo(min)< 0) //Abfrage auf <
       {
          min = feld[pos];
          posMin = pos; //Kleinste Position merken
       }
     //Vertauschen
```

```
      merke = feld[i];
      feld[i] = feld[posMin];
      feld[posMin] = merke;
   }
  }
}
```

Der Programmlauf liefert die sortierten Nachnamen.

Felder können sehr groß sein. Beim Sortieren kann ein Feld beispielsweise eine Million Elemente umfassen. Stehen auf der Parameterliste Variablen, die einen einfachen Typ besitzen, dann werden die aktuellen Werte in die Parametervariablen kopiert (siehe: »Prozeduren mit Eingabeparametern«, S. 213). Dieser Parameterübergabemechanismus ist bei Feldern *nicht* sinnvoll, da doppelter Speicherplatz sowie die Zeit zum Kopieren der Inhalte benötigt würde.

passing a reference by value

Stehen Felder auf der Parameterliste, dann werden die Feldwerte *nicht* übergeben bzw. kopiert, sondern nur eine sogenannte Referenz auf das jeweilige Feld. Dieser Übergabemechanismus heißt in Java daher *passing a reference by value*. Der Zugriff auf die Feldelemente geschieht dann über die übergebene Referenz.

Beispiel 1b

Die Abb. 6.3-1 veranschaulicht den Mechanismus am Beispiel des Programms `SortAuswahl2` (Beispiel 1a).

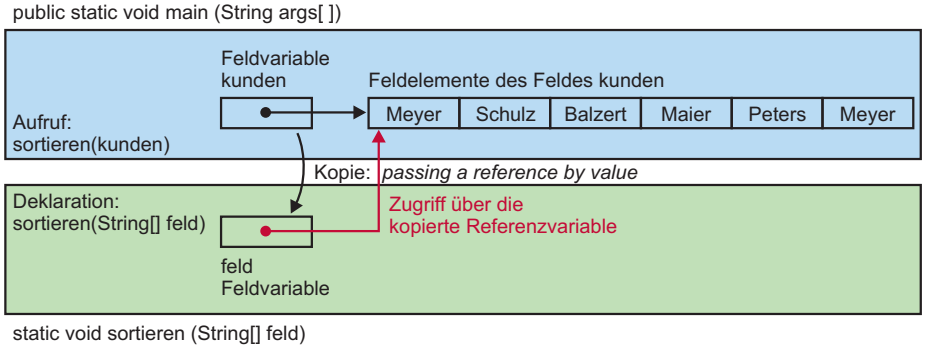


Abb. 6.3-1: Beispiel für die Parameterübergabe nach dem Prinzip *call by reference*.

Betrachtet man die Speicherung eines Feldes im Arbeitsspeicher genauer, dann sieht man, dass die Feldvariable, hier `kunden`, aus einer Speicherzelle besteht, in der eine Speicheradresse gespeichert ist. Diese Speicheradresse gibt an, wo im Arbeitsspeicher die eigentlichen Feldelemente abgelegt sind.

Man sagt, die Feldvariable `kunden` enthält eine Referenz bzw. einen Verweis auf die eigentlichen Feldelemente.

Innerhalb der gerufenen Prozedur, hier `sortieren(String[] feld)`, wird in die Feldvariable `feld` nun die Speicheradresse aus der Feldvariablen `kunden` der aufrufenden Prozedur kopiert.

Dadurch kann jetzt von der gerufenen Prozedur aus auch direkt auf die Feldelemente des Feldes `kunden` zugegriffen werden.

Für den formalen Parameter `feld` wird also genau ein Speicherplatz angelegt, der solange existiert, solange die Prozedur abgearbeitet wird.

Wird innerhalb der Prozedur `sortieren()` jetzt auf ein Feldelement, z. B. `feld[3]`, zugegriffen, dann wird über die Referenz auf die Speicheradresse vom Feld `kunden` verzweigt und dort dann das vierte Feldelement lokalisiert. Es erfolgt sowohl beim lesenden als auch beim schreibenden Zugriff ein Zugriff auf das Originalfeld.

Das sortierte Feld steht anschließend also im Feld `kunden`. Damit ist das Feld `kunden` gleichzeitig auch Ausgabeparameter.

Die Prozedur `sortieren()` arbeitet also *nicht* auf einer Kopie der Daten, sondern verändert direkt die Originaldaten.

Das ist effizient, aber auch gefährlich. Gefährlich dann, wenn die Prozedur `sortieren()` fehlerhaft ist.

Der wichtigste Unterschied des Mechanismus *passing a reference by value* gegenüber der reinen Werteübergabe *(call by value)* besteht also darin, dass alle Änderungen am Originalfeld und *nicht* an einer Kopie des Feldes vorgenommen werden.

Hinweis

Der Übergabemechanismus *passing a reference by value* in Java wird in der Literatur oft auch als *call by reference* bezeichnet. Dies ist nicht ganz korrekt, da in Java der übergebene Referenzwert nicht verändert werden kann. In der Programmiersprache C++ kann der Referenzwert manipuliert werden, deshalb ist dort die Bezeichnung `call by reference` üblich.

Gehen Sie das Beispielprogramm für mehrere Aufrufe durch und tragen Sie die Änderungen im Originalfeld ein.

Prozeduren so konzipieren, dass ...
- sie jeweils ein Teilproblem bzw. eine Teilaufgabe lösen (nicht mehrere).
- durch eine geeignete Parameterwahl eine möglichst allgemeine, d. h. in mehreren Kontexten einsetzbare Teilproblemlösung erreicht wird.
- auf der Parameterliste möglichst nur einfache Typen verwendet werden, Felder nur wenn unbedingt nötig.

6.4 Funktionen und Ausgabeparameter *

Soll eine Methode ein Ergebnis an die rufende Methode zurückliefern, dann geschieht dies in Java durch den Ergebnisparameter einer Funktion. Mehrere Ergebnisse (einfacher Typen) können in Java *nicht* zurückgegeben werden. Eine Funktion sollte im Unterschied zu einer Prozedur immer in einem Ausdruck aufgerufen werden, damit die Ergebnisse übergeben werden können (`call by result`). Innerhalb der Funktion werden die Ergebnisse durch `return` zurückgegeben.

In der Praxis kommt es nicht nur vor, dass durch Eingabeparameter Daten an eine Methode übergeben werden. Genauso häufig kommt es vor, dass Ergebnisse aus einer Methode an das aufrufende Programm übergeben werden müssen. Außerdem kann es vorkommen, dass ein Wert in eine Methode eingegeben wird und nach einer Modifikation wieder ausgegeben wird.

Einschränkung in Java

In Java können direkt über die Parameterliste *keine* Ergebniswerte einfacher Typen zurückgegeben werden. Stehen jedoch Felder auf der Parameterliste, dann können wegen des Referenzmechanismus indirekt auch Ergebnisse nach »außen« gegeben werden (siehe: »Felder als Eingabeparameter«, S. 219).

Genau ein Ergebniswert

In der Praxis treten oft Methoden bzw. Prozeduren auf, die genau ein Ergebnis ermitteln. D.h. als Ergebnis der Methodenausführung wird an das rufende Programm genau ein Wert übergeben. Diesen häufigen Sonderfall einer Methode bzw. Prozedur bezeichnet man als **Funktion**. Funktionsdeklaration und Funktionsaufruf unterscheiden sich von Prozeduren.

Beispiel 1a

Netto

```
import inout.Console;
public class Netto
{
 public static void main (String args[])
 {
     double bruttopreis, nettopreis;
     final double MWST_ER = 1.07;
     System.out.print("Bruttopreis eingeben: ");
     bruttopreis = Console.readDoubleComma();
```

```
    nettopreis = Netto_berechnen(bruttopreis, MWST_ER); //Auf.
    System.out.print("Nettopreis: " + nettopreis);
 }

 public static double Netto_berechnen
   (double brutto, double MWST_Satz)
 {
  double netto = brutto / MWST_Satz;
  return netto; //Ergebnis zurückgeben
 }
}
```

Ergebnisparameter

Kennzeichnend für eine Funktion ist, dass der Ergebnisparameter *nicht* auf der Parameterliste steht und *keinen* eigenen Parameterbezeichner erhält. Als Bezeichner wird vielmehr der Bezeichner der Funktion verwendet (im Beispiel: `Netto_berechnen`). Wie bei allen anderen Parametern muss jedoch auch für den Ergebnisparameter der Typ angegeben werden. Er steht vor dem Funktionsnamen (im Beispiel: `double`).

Syntax

MethodDeclaration ::=
[`public` | `private` *] [* `static` *]*
ResultType ResultIdentifier (*[FormalParameterList]*)
{ *MethodBody* }

Bei Methoden bzw. Prozeduren, die *kein* Ergebnis zurückgeben, steht anstelle des Ergebnistyps das Schlüsselwort `void`, das übersetzt »leer« bedeutet.

6

return

Innerhalb des Funktionsrumpfes muss mindestens eine `return`-Anweisung stehen, sonst ist die Funktion syntaktisch nicht korrekt (Ausnahme: `void`-Funktionen; sie enden automatisch am Ende des Funktionsrumpfes):

Syntax

ReturnStatement ::= `return` *Expression* ;

Der Ausdruck hinter `return` muss einen Wert ergeben, der mit dem Ergebnistyp der Funktion übereinstimmt oder verträglich ist.

Hinweis

> Methoden bzw. Prozeduren, die *kein* Ergebnis zurückgeben, d. h. mit `void` deklariert sind, dürfen `return`-Anweisungen enthalten, jedoch ohne Rückgabewert (ergibt sonst einen Compilerfehler), d. h. `return;`. Die `return`-Anweisung bewirkt, dass der Kontrollfluss an den Aufrufer zurückgeht.

Aufruf in Ausdrücken

Da eine Funktion immer einen Wert als Ergebnis zurückgibt, sollte sie sinnvollerweise innerhalb eines Ausdrucks aufgerufen werden. Sie kann auch als eigenständige Anweisung verwendet werden, das Ergebnis geht dadurch aber verloren.

Beispiel 1b

Aufruf der Funktion `Netto_berechnen`:

```
nettopreis = Netto_berechnen(bruttopreis, MWST_ER);
```

Die Parameterübergabe veranschaulicht die Abb. 6.4-1. Nach der Berechnung von `netto` innerhalb der gerufenen Funktion `Netto_berechnen()` bewirkt `return`, dass der Inhalt der Variablen `netto` in die Speicherzelle `nettopreis` kopiert wird.

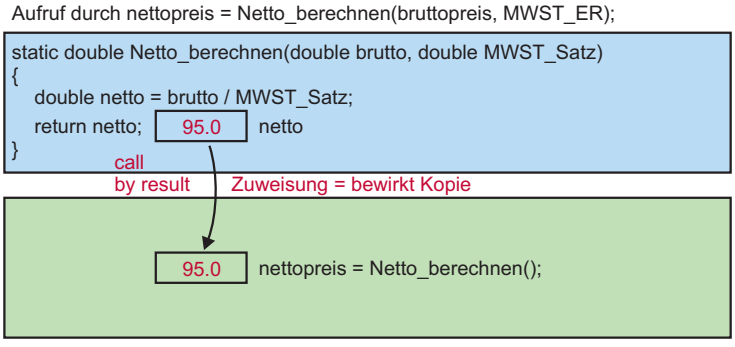


Abb. 6.4-1: Beispiel für die Parameterübergabe nach dem Prinzip *call by result*.

call by result

Als Parametermechanismus für den Ergebnisparameter bei Funktionen wird *call by result* verwendet, d. h. der Wert, der sich nach der Berechnung des Ausdrucks hinter `return` ergibt, wird in die Speicherzelle kopiert, die auf der linken Seite der Zuweisung steht, hier `nettopreis`. Befindet sich der Aufruf innerhalb eines Ausdrucks auf der rechten Seite einer Zuweisung, dann geht der Wert in die Berechnung des Ausdrucks ein.

6.5 Java-Funktionen nutzen *

Java stellt standardmäßig eine Vielzahl von Klassen zur Verfügung, die jederzeit benutzt werden können. Die Klassen befinden sich in Paketen. Um eine fremde Klasse nutzen zu können, muss sie mit der `import`-Anweisung bekannt gemacht werden. Eine oft benötigte Klasse ist die Klasse `Math`, die eine Vielzahl von mathematischen Funktionen zur Verfügung stellt.

Java stellt dem Programmierer eine ganze Reihe von fertigen Funktionen für verschiedene Zwecke zur Verfügung.

Klasse `Math`

Eine wichtige Klasse ist die Klasse `Math`, die einige Konstanten und eine Vielzahl mathematischer Funktionen enthält. Diese Klasse gehört in Java zu dem Paket `lang` (für *language*), das automatisch in jedem Programm zur Benutzung zur Verfügung steht.

Konstanten

Zwei wichtige, häufig benötigte Konstanten, die diese Klasse zur Verfügung stellt, sind:

- `static final double E`: Die Eulersche Zahl e = 2,718...
- `static final double PI`: Die Kreiszahl π = 3,14159...

Einige wichtige Funktionen sind:

Runden von Werten

- `public static double ceil (double x)`:
  Dient zum Aufrunden. Liefert die nächsthöhere Ganzzahl, wenn die Zahl nicht schon eine ganze Zahl ist, z. B. `ceil(2.1)` ergibt den Wert `3`, `ceil(-2.1)` ergibt den Wert `-2`.
- `public static double floor (double x)`:
  Dient zum Abrunden. Es wird die nächstniedrigere Ganzzahl zurückgegeben (ganze Zahlen werden nicht verändert), z. B. `floor(0.2)` ergibt `0.0`, `floor(-199.2)` ergibt `-200`, `floor(-199)` ergibt `-199`, `floor(-0.02)` ergibt `-1.0`.
- `public static long round (double x)`:
  Rundet auf die nächste Ganzzahl vom Typ `long` (kaufmännische Rundung). Ganze Zahlen werden *nicht* aufgerundet, z. B. `round(2.02)` ergibt `2`, `round(-3.3)` ergibt `-3`, `round(100)` ergibt `100`.
- `public static double rint (double x)`:
  Diese Funktion gibt den Wert zurück, der näher an dem übergebenen Wert ist und einem int-Wert entspricht. Für den Fall, dass zwei int-Werte in Frage kommen, wird der gerade int-Wert zurückgegeben. Die Werte 1.5 und 2.5 erhalten 2.0 als Rückgabe.

Tipp

Runden auf zwei Nachkommastellen

Die `rint()`-Funktion lässt sich verwenden, um Zahlen auf zwei Nachkommastellen zu runden. Ist `d` vom Typ `double`, dann ergibt der Ausdruck `Math.rint(d*100.0)/100.0` die gerundete Zahl. Wird statt `rint()` die Funktion `round()` verwendet, dann wird z. B. der Wert `5.125` nicht auf `5.12`, sondern auf `5.13` gerundet.

Zufallszahlen

- `public static double Math.random()`:
  Diese Funktion liefert Zufallszahlen zwischen `0.0` und `1.0`.

Tipp

Zufallszahlen zwischen uGrenze und oGrenze

Wird ein anderer Wertebereich benötigt, dann können die gelieferten Zahlen durch Multiplikation auf den gewünschten Wertebereich ausgedehnt und per Addition geeignet verschoben werden. Um ganzzahlige Zufallszahlen zwischen `uGrenze` und `oGrenze` (einschließlich) zu erhalten, wird folgendes berechnet:

```
uGrenze+(int)(Math.floor(Math.random()*(oGrenze-uGrenze+1)))
```

Außerdem kann der Modulo-Operator benutzt werden, um den Wertebereich zu beschneiden.

Beispiel

Wuerfel

```
//Beispiel für den Einsatz mathematischer Funktionen
//aus der Java-Klasse Math
//Berechnung von Zufallszahlen zwischen 1 und 6

public class Wuerfel
{
 public static void main (String args[])
 {
  double zufall;
  int wuerfel;
  for (int i = 1; i <= 20; i++)
  {
    zufall = Math.random();
    wuerfel = 1 + (int)(Math.floor(zufall * (6 - 1 + 1)));
    System.out.print(wuerfel + "\t");
    System.out.print(Math.rint(zufall * 100.0)/100.0 + "\t");
    System.out.println(zufall);
   }
  }
}
```

Der Programmlauf ergibt folgendes Ergebnis (Ausschnitt):

```
5   0.68    0.6751377163347234
6   0.99    0.9945680098216266
6   0.90    0.8961979500252908
6   0.89    0.8918723094957002
4   0.51    0.5126964182064753
6   0.92    0.9198718589574543
3   0.47    0.47329735836332576
```


Zugriff auf Konstanten & Funktionen

Auf die Konstanten und Funktionen kann durch das **Voransetzen des Klassennamens** `Math` zugegriffen werden, z. B. `double pi = Math.PI; long gerundet = Math.round(-22.56).`

Hinweis

Das Voransetzen des Klassennamens entfällt, wenn vor das Programm eine Import-Anweisung der folgenden Art gesetzt wird: `import static paket1.paket2.Klasse.*;`, z. B. `import static java.lang.Math.*;`

Winkelfunktionen

- `static double sin (double x)`:
  Liefert den Sinus von x zurück
- `static double cos (double x)`:
  Liefert den Cosinus von x zurück.
- `static double tan (double x)`:
  Liefert den Tangens von x zurück.

Die Winkel für `sin()`, `cos()`, `tan()` müssen im Bogenmaß (2 * π entspricht einem Vollkreis) und *nicht* im Gradmaß (360 Grad entspricht einem Vollkreis) übergeben werden.

- `static double asin (double x)`:
  Liefert den Arcus-Sinus von x, wobei $-\pi/2 \leq \text{asin} \leq \pi/2$.
- `static double acos (double x)`:
  Liefert den Arcus-Cosinus von x, wobei $0 \leq \text{acos} \leq \pi$.
- `static double atan (double x)`:
  Liefert den Arcus-Tangens von x, wobei $-\pi/2 \leq \text{atan} \leq \pi/2$.

Die Arcus-Funktionen sind die Umkehrfunktionen zu den trigonometrischen Funktionen. Der Parameter ist *kein* Winkel, sondern bei asin() beispielsweise der Sinuswert zwischen −1 und 1. Das Ergebnis ist ein Winkel im Bogenmaß, etwa zwischen $-\pi/2$ und $\pi/2$.

- `static double toRadians (double angdeg)`:
  Grad- in Bogenmaß umwandeln.
- `static double toDegrees (double angrad)`:
  Winkel von Bogen- in Gradmaß umwandeln

Weitere Funktionsgruppen

Die Klasse `Math` enthält weiterhin Exponentialfunktionen, Funktionen zur Ermittlung von Absolutwerten, Maximum und Minimum.

Geben Sie in eine Suchmaschine die Begriffe `Oracle`, `Java`, `Math` ein und Sie erhalten die Klassenbeschreibung von `Math`. Schauen Sie sich die Funktionen an und probieren Sie einige davon in eigenen Programmen aus.

Klasse `DecimalFormat`

Um das Ausgabeformat von Zahlen festzulegen – insbesondere bei Listen wichtig – gibt es die Klasse `DecimalFormat`. Diese Klasse gehört zu dem Java-Paket `text` und muss explizit importiert werden, um die Funktionen nutzen zu können:

```
import java.text.DecimalFormat;
```

Nützliche Funktionen

Für die Ausgabeformatierung sind folgende Funktionen hilfreich:

- Einstellung eines Formats:
  ```
  DecimalFormat meinFormat1 =
  new DecimalFormat( "Formatstring" );
  ```
  Der `Formatstring` kann folgende Zeichen enthalten:
  - `0`: Steht für eine Ziffer
  - `#`: Steht für eine Ziffer außer 0
  - `.`: Steht für einen Dezimalpunkt/Dezimalkomma
  - `,`: Steht für ein Gruppierungszeichen (3er-Gruppen)
  - `;`: Steht für einen Trenner zwischen positiv und negativ
  - `-`: Steht für ein negatives Vorzeichen
  - `%`: Steht für das Prozentformat (dividiert durch 100)
- Das definierte Format wird durch Aufruf der Funktion `format(auszugebendeZahl)` zugeordnet, wobei vor der Funktion, durch Punkt getrennt, der gewählte Variablenname stehen

muss, z. B.

```
System.out.println(meinFormat1.format(MWST19));
```

Beispiele

`DecimalFormat( "0.00" );` ergibt:
Genau zwei Nachkommastellen.

`DecimalFormat(" #,##0.0###; -#,##0.0###");` ergibt:
Vier Vorkomma- und vier Nachkommastellen, wobei führende Vorkomma-Nullen und Nachkomma-Nullen am Ende unterdrückt werden. Außerdem werden vorne zwei Stellen Platz gelassen, von denen eine ggf. für ein negatives Vorzeichen verwendet wird.

Hinweis

Vorkommastellen können mit `DecimalFormat()` höchstens mit Nullen gefüllt werden, *nicht* mit Leerzeichen.

Beispiel

MWSTTabelle Formatiert

```
// Ausgabe einer formatierten MWST-Tabelle

import java.text.DecimalFormat;

public class MWSTTabelleFormatiert
{
 public static void main (String args[])
 {
  final int MWST_VOLL = 19;
  final int MWST_ERMAESSIGT = 7;
  double mwstV, mwstE;

  DecimalFormat meinFormat1=new DecimalFormat("00.00 €");
  DecimalFormat meinFormat2=new DecimalFormat("#0.## €");

  System.out.println("MWST-Tabelle");
  System.out.println("Netto\t\t19%\t\tBrutto\t\t7%\t\tBrutto");
  for (int i = 1; i <= 30; i++)
  {
    mwstV = i * MWST_VOLL / 100.0;
    mwstE = i * MWST_ERMAESSIGT / 100.0;
    System.out.println(
      meinFormat2.format(i) +  "\t\t"
    + meinFormat1.format(mwstV) + "\t\t"
    + meinFormat1.format(i + mwstV) +  "\t\t"
    + meinFormat1.format(mwstE) + "\t\t"
    + meinFormat1.format((i + mwstE)));
    }
 }
}
```

Der Programmlauf erzeugt folgende Ausgabe:

```
MWST-Tabelle
Netto          19%      Brutto           7%       Brutto
1 €            00,19 €  01,19 €          00,07 €  01,07 €
2 €            00,38 €  02,38 €          00,14 €  02,14 €
3 €            00,57 €  03,57 €          00,21 €  03,21 €
```

```
4 €          00,76 €  04,76 €          00,28 €  04,28 €
5 €          00,95 €  05,95 €          00,35 €  05,35 €
6 €          01,14 €  07,14 €          00,42 €  06,42 €
7 €          01,33 €  08,33 €          00,49 €  07,49 €
8 €          01,52 €  09,52 €          00,56 €  08,56 €
9 €          01,71 €  10,71 €          00,63 €  09,63 €
10 €         01,90 €  11,90 €          00,70 €  10,70 €
11 €         02,09 €  13,09 €          00,77 €  11,77 €
12 €         02,28 €  14,28 €          00,84 €  12,84 €
13 €         02,47 €  15,47 €          00,91 €  13,91 €
14 €         02,66 €  16,66 €          00,98 €  14,98 €
15 €         02,85 €  17,85 €          01,05 €  16,05 €
16 €         03,04 €  19,04 €          01,12 €  17,12 €
17 €         03,23 €  20,23 €          01,19 €  18,19 €
18 €         03,42 €  21,42 €          01,26 €  19,26 €
19 €         03,61 €  22,61 €          01,33 €  20,33 €
20 €         03,80 €  23,80 €          01,40 €  21,40 €
21 €         03,99 €  24,99 €          01,47 €  22,47 €
22 €         04,18 €  26,18 €          01,54 €  23,54 €
23 €         04,37 €  27,37 €          01,61 €  24,61 €
24 €         04,56 €  28,56 €          01,68 €  25,68 €
25 €         04,75 €  29,75 €          01,75 €  26,75 €
26 €         04,94 €  30,94 €          01,82 €  27,82 €
27 €         05,13 €  32,13 €          01,89 €  28,89 €
28 €         05,32 €  33,32 €          01,96 €  29,96 €
29 €         05,51 €  34,51 €          02,03 €  31,03 €
30 €         05,70 €  35,70 €          02,10 €  32,10 €
```

Variieren Sie die Ausgabeformate und sehen Sie sich die Wirkung an.

Hinweis

Es gibt auch fertige Formate wie:

```
NumberFormat meinFormat= NumberFormat.getCurrencyInstance();
```

Sie müssen dann folgende Import-Anweisung ergänzen:

```
import java.text.NumberFormat;
```

6.6 Felder als Ergebnisparameter *

Müssen von einer Funktion in Java mehrere Ergebniswerte an die rufende Methode zurückgeliefert werden, dann kann dies durch ein Feld geschehen, das als Ergebnistyp angegeben wird, z. B. `float[] ermittleZimmertemperaturen()`.

In Java wird als Ergebnis einer Funktion in der Regel *genau* ein Wert übergeben, der anschließend einer Variablen zugewiesen wird (siehe »Funktionen und Ausgabeparameter«, S. 223).

In vielen Anwendungsfällen müssen jedoch mehrere Ergebniswerte übergeben werden. Eine Möglichkeit besteht in Java darin, als Ergebnistyp ein Feld zu deklarieren.

Beispiel

Stunden Minuten

Es liegen Zeitangaben in Minuten vor, die in Minuten und Stunden umgerechnet und zurückgegeben werden sollen. Folgende Funktion löst das Problem:

```
//Beispiel für ein Feld als Ergebnisparameter einer Funktion
//Umrechnung von Minuten in Stunden und Minuten

import inout.Console;
public class StundenMinuten
{
 public static int[] wandleMinutenInStunden(int minuten)
 {
  int stundenMinuten[] = {0,0};
  stundenMinuten[0] = minuten / 60;
  stundenMinuten[1] = minuten % 60;
  return stundenMinuten;
 }
 public static void main (String args[])
 {
   System.out.println("Minuten eingeben:");
   int minuten = Console.readInt();
   int stundenUndMinuten[] =
     wandleMinutenInStunden(minuten);
   System.out.println(minuten + " Minuten = "+
   stundenUndMinuten[0] + " h "
    + stundenUndMinuten[1] + " min");
 }
}
```

Ein Programmlauf sieht wie folgt aus:

```
Minuten eingeben:
183
183 Minuten = 3 h 3 min
```

Ändern Sie das Programm so, dass Sekunden eingegeben werden und Tage, Stunden, Minuten und Sekunden ausgegeben werden.

Funktionen...

- verwenden, wenn ein Ergebnis (einfacher Typ) oder mehrere Ergebnisse (Feld) an das rufende Programm übergeben werden müssen.
- müssen vor dem Funktionsnamen den Typ des Ergebnisses angeben.
- müssen innerhalb ihres Funktionsrumpfes mindestens eine `return`-Anweisung zur Ergebnisrückgabe enthalten.
- sollten immer innerhalb eines Ausdrucks aufgerufen werden, da sonst das Ergebnis verloren geht.
- verwenden für die Ergebnisübergabe den Parametermechanismus *call by result*.

6.7 Variable Parameterlisten ***

In Java kann pro Parameterliste am Ende *ein* variabler Parameter (gekennzeichnet durch drei Punkte) angegeben werden. Beim Aufruf einer solchen Methode können dann jeweils unterschiedlich viele Parameter angegeben werden.

Problem

Es gibt eine Reihe von Anwendungsfällen, bei denen für jeden Aufruf einer Methode eine unterschiedliche Anzahl von Parametern über die Parameterliste übergeben werden sollen. Gestattet eine Programmiersprache nur eine feste Anzahl von Parametern, dann ist es schwierig, einen solchen Anwendungsfall elegant zu programmieren.

Java

Eine variable Anzahl von Parametern wird in Java durch drei Punkte (sogenanntes Auslassungszeichen) hinter der jeweiligen Typangabe angegeben.

Beispiel

Familie

Es sollen von jeder Familie nach dem Familiennamen alle Vornamen der Familienmitglieder ausgegeben werden:

```
// Beispiel für variable Parameterlisten
public class Familie
{
 public static void
  ausgebenMitglieder(String familienname, String... vornamen)
 {
  System.out.println("Familienname: " + familienname);
  for (String aktuell: vornamen)
    System.out.println("Vorname: " + aktuell);
  System.out.println("--------------------------");
 }
 public static void main(String args[])
 {
  ausgebenMitglieder("Sonnenschein", "Frank","Anita","Max");
  ausgebenMitglieder("Klug", "Alexander", "Simone",
    "Udo", "Dirk", "Antje");
  ausgebenMitglieder("Ross", "Ralf", "Birgit");
 }
}
```

Der Programmlauf ergibt folgendes Ergebnis:

```
Familienname: Sonnenschein
Vorname: Frank
Vorname: Anita
Vorname: Max
-------------------------
Familienname: Klug
Vorname: Alexander
Vorname: Simone
Vorname: Udo
Vorname: Dirk
Vorname: Antje
-------------------------
```

```
Familienname: Ross
Vorname: Ralf
Vorname: Birgit
-------------------------
```

Hinweis

In diesem Beispiel wird eine vereinfachte Form der `for`-Schleife verwendet (siehe: »Iteration über Felder: Die erweiterte for-Schleife«, S. 198). Es wird auf die Zählvariable verzichtet und statt dessen `vornamen` angegeben, wobei `vornamen` hier für null, einen oder mehrere `vornamen` steht (Feld bestehend aus Strings). Die `for`-Schleife durchläuft alle vorhandenen Elemente des impliziten Feldes `vornamen`.

Pro Parameterliste kann nur ein variabler Parameter angegeben werden. Er muss am Ende der Parameterliste stehen!

Hinweis

Das Konzept variabler Parameterlisten stammt aus der Programmiersprache C und wird dort intensiv in den Funktionen `printf()` und `scanf()` eingesetzt.

6.8 Überladen von Methoden **

Innerhalb einer Klasse können Methoden mit gleichen Namen deklariert werden, wenn sie eine *unterschiedliche* Signatur besitzen. Die Signatur besteht aus dem Methodennamen, der Anzahl der Parameter und den Parametertypen. Der Ergebnistyp gehört *nicht* dazu. Der Vorteil dieses Überlagerns *(overloading)* von Methoden liegt darin, dass man keine unterschiedlichen Methodennamen wählen muss, nur weil die Parameter unterschiedlich sind, sonst aber die gleiche Aufgabe erledigt wird. Teilaufgaben, die durch bereits vorhandene Methoden gelöst werden, sollten benutzt und *nicht* neu programmiert werden (Delegationsprinzip).

In manchen Anwendungsfällen benötigt man *eine* Funktion, die eine Berechnung vornimmt. Die Berechnung kann von keinem, einem oder auch mehreren Parametern abhängen.

Beispiel 1a

Zur Erstellung einer Zeitstatistik müssen alle Zeitwerte in Minuten vorliegen (Sekunden werden abgeschnitten). Die Ursprungswerte liegen aber in unterschiedlicher Weise vor: In Stunden und Minuten, in Tagen und Stunden und Minuten sowie in Sekunden. Es kann auch vorkommen, dass Stunden und Minuten als `float`-Wert angegeben sind, z. B. 1,5 Stunden. Eine Möglichkeit der Umrechnung in Minuten besteht darin, vier verschiedene Funktionen mit unterschiedlichen Funktionsnamen zu schreiben:

```
int berechneMinuten1 (int minuten, int stunden);
int berechneMinuten2 (int minuten, int stunden, int tage);
int berechneMinuten3 (int sekunden);
int berechneMinuten4 (double stundenMinuten);
```

Nachteilig ist, dass entweder problemfremde Funktionsnamen entstehen wie hier. Oder es entstehen lange Funktionsnamen wie `int berechneMinutenAusMinutenUndStunden(int minuten, int stunden);`

Signatur

In Java ist es möglich, *einen* Funktionsnamen für mehrere Funktionen zu verwenden, wenn die Parameteranzahl und/oder der Parametertyp unterschiedlich sind – genauer gesagt wenn die Signatur unterschiedlich ist. Die **Signatur** einer Methode besteht aus dem Namen der Methode sowie der Anzahl und den Typen der formalen Parameter. Der Ergebnistyp einer Funktion zählt *nicht* zur Signatur.

Beispiel 1b

Minuten Berechnung

```
//Beispiel für das Überladen von Methoden
//Umrechnung in Minuten
public class MinutenBerechnung
{
 public static int berechneMinuten(int minuten, int stunden)
 {
  int minutenGesamt = stunden * 60 + minuten;
  return minutenGesamt;
 }
 public static int
   berechneMinuten(int minuten, int stunden, int tage)
 {
  int minutenGesamt = tage * 24 * 60 + stunden * 60 + minuten;
  return minutenGesamt;
 }
 public static int berechneMinuten(int Sekunden)
 {
  int minutenGesamt = Sekunden / 60;
  return minutenGesamt;
 }
 public static int berechneMinuten(float stundenMinuten)
 {
  int minutenGesamt =
  (int)(stundenMinuten * 100 / 100)* 60 +
  (int)(stundenMinuten * 100 % 100) * 60 / 100;
  return minutenGesamt;
 }
 public static void main (String args[])
 {
   int minuten = berechneMinuten(45, 3);
   System.out.println("Minuten insgesamt: " + minuten);
   minuten = berechneMinuten(45, 3, 2);
   System.out.println("Minuten insgesamt: " + minuten);
   minuten = berechneMinuten(3768);
   System.out.println("Minuten insgesamt: " + minuten);
   minuten = berechneMinuten(1.5f);
```

```
    System.out.println("Minuten insgesamt: " + minuten);
  }
}
```

Der Programmlauf liefert folgendes Ergebnis:

```
Minuten insgesamt: 225
Minuten insgesamt: 3105
Minuten insgesamt: 62
Minuten insgesamt: 90
```

Semantik

In einer Klasse dürfen *nicht* zwei Methoden mit derselben Signatur deklariert werden, sonst meldet der Compiler einen Fehler.

Überladen

Besitzen zwei Methoden derselben Klasse denselben Namen, aber ansonsten unterschiedliche Signaturen, dann bezeichnet man diesen Methodennamen als **überladen** *(overloaded).*

Delegationsprinzip

Ein Prinzip in der Programmierung lautet, bereits vorhandene Leistungen, hier angeboten in Form von Methoden, möglichst wieder zu verwenden und nicht erneut zu programmieren.

Betrachten Sie das Programm `MinutenBerechnung`. Welche Methode könnte welche andere Methode aufrufen, um eigene Arbeit zu sparen?

Beispiel 1c

In der Methode

```
static int berechneMinuten(int minuten, int stunden)
```

werden Minuten und Stunden in Minuten umgerechnet.

In der Methode

```
static int berechneMinuten(int minuten, int stunden, int tage)
```

wird diese Berechnung ebenfalls benötigt.

Statt Minuten und Stunden selbst in Minuten umzurechnen, kann diese Aufgabe die bereits vorhandene Methode

```
static int berechneMinuten(int minuten, int stunden)
```

übernehmen.

In der Methode

```
static int berechneMinuten(int minuten, int stunden, int tage)
```

ändert sich die Anweisung

```
int minutenGesamt = tage * 24 * 60 + Stunden * 60 + minuten;
```

in die Anweisung

```
int minutenGesamt = Tage * 24 * 60 + berechneMinuten(minuten,
stunden);
```

Im ersten Moment werden Sie vielleicht denken: Was bringt das? In diesem Beispiel zunächst nicht viel. Aber im Laufe einer Software-Nutzung gibt es immer wieder Änderungen.

Muss in der Methode

```
static int berechneMinuten(int minuten, int stunden)
```

eine Änderung oder Erweiterung vorgenommen werden, dann

muss diese Änderung bei dieser Lösung nicht auch noch in der Methode
`static int berechneMinuten(int minuten, int stunden, int tage)`
vorgenommen werden!

Hinweis

Auch die `main()`-Methode kann nach denselben Regeln überladen werden. Beim Start über die Konsole wird jedoch die Standardsignatur erwartet, sonst gibt es einen Fehler. `main()`-Methoden mit anderen Signaturen werden von der JVM ignoriert. Beim Start über Entwicklungsumgebungen kann aber u. U. eine andere `main()`-Methode gewählt werden. Außerdem können von innerhalb der jeweiligen Klasse andere `main()`-Methoden aufgerufen werden.

6.9 UML-Sequenzdiagramme **

Ein UML-Sequenzdiagramm ermöglicht die grafische, zeitbasierte Darstellung von Methodenaufrufen innerhalb und zwischen Klassen und/oder Akteuren, z. B. Benutzern. Die Zeitachse verläuft vertikal. Aufrufe werden durch horizontale Linien, Klassen und Akteure durch gestrichelte, vertikale Linien repräsentiert (Lebenslinien). Ist eine Methode oder ein Akteur aktiv, dann wird auf der Lebenslinie eine Aktionssequenz, dargestellt durch ein Rechteck, eingetragen.

Enthält eine Klasse mehrere Methoden, die sich gegenseitig aufrufen, dann verliert man leicht den Überblick über die Aufruffolge. Ein Hilfsmittel, um sich den zeitlichen Ablauf der Aufrufe für ein bestimmtes Szenario zu verdeutlichen, ist das Sequenzdiagramm der UML.

Aufgabe

Das **UML-Sequenzdiagramm** *(sequence diagram*, abgekürzt **sd**) dient dazu, den zeitlichen Ablauf von Methoden und die Kommunikation mit Akteuren (z. B. Menschen) darzustellen, um eine bestimmte Aufgabe zu erledigen (Abb. 6.9-1).

Kennzeichnend für diese Darstellungsform ist eine (gedachte) Zeitachse, die vertikal von oben nach unten führt. Klassen und Akteure, die miteinander kommunizieren, werden durch gestrichelte vertikale Geraden dargestellt (Lebenslinien). Jede **Lebenslinie** *(lifeline)* repräsentiert die Existenz einer Klasse oder eines Akteurs während einer bestimmten Zeit. Eine Lebenslinie beginnt nach der Existenz einer Klasse oder eines Akteurs und endet mit dem Löschen. Existiert eine Klasse oder ein Akteur während der gesamten Ausführungszeit, dann ist die Linie von oben nach unten durchgezogen. Am oberen Ende der Linie wird ein Klassensymbol und/oder ein Akteursymbol gezeichnet. Zu-

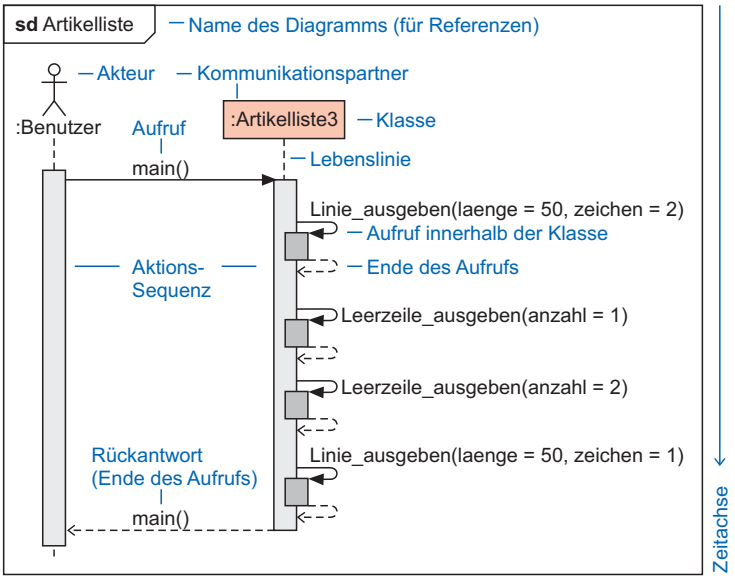


Abb. 6.9-1: Beispiel und Aufbau eines UML-Sequenzdiagramms.

sätzlich kann ein Schlüsselwort *(keyword)* in französischen Anführungszeichen hinzugefügt werden, z. B. <<actor>>>.

Akteur

Die erste vertikale Linie bildet in vielen Sequenzdiagrammen ein Akteur – in der Regel der Benutzer – oft dargestellt als »Strichmännchen«.

Aufrufe

In das Sequenzdiagramm werden die Aufrufe eingetragen, die zum Aktivieren der Methoden dienen. Jeder Aufruf wird als gerichtete Kante (mit gefüllter Pfeilspitze) vom Sender zum Empfänger gezeichnet. Der Pfeil wird mit dem Namen der aktivierten Methode beschriftet. Eine aktive Methode wird durch einen schmalen Aktivitäts-Balken auf der Lebenslinie angezeigt. Nach dem Beenden der Methode zeigt eine gestrichelte Linie mit offener Pfeilspitze, dass der Kontrollfluss zur aufrufenden Methode zurückgeht.

Referenzierung

Um ein Sequenzdiagramm in anderen Diagrammen referenzieren zu können, kann ein Sequenzdiagramm, wie auch bei anderen Diagrammarten der UML, durch einen Rahmen umgeben werden, in dessen linken oberen Ecke der Name des Diagramms eingetragen wird. Davor wird **sd** für ***s**equence **d**iagram* eingetragen.

Dauer

Die Aktivitäts-Balken zeigen die Dauer der jeweiligen Verarbeitung. Ist die Verarbeitung abgeschlossen, dann geht der Kontrollfluss wieder zur rufenden Methode zurück. Gehören Sender- und

Empfängermethode zur selben Klasse, dann werden die Aktivitäts-Balken übereinander »gestapelt«.

Beispiel

Zu folgendem Programm wird ein Sequenzdiagramm erstellt (siehe auch »Prozeduren mit Eingabeparametern«, S. 213):

```
// Ausgeben einer Artikelliste in das Konsolenfenster
// Leerzeile_ausgeben mit Eingabeparameter anzahl
// Linie_ausgeben mit Eingabeparametern laenge und zeichen

public class Artikelliste3
{
 public static void main (String args[])
 {
  System.out.println
   ("Artikel-Nr         Artikelbezeichnung    Preis [€]");
  Linie_ausgeben(50,'='); //1. Aufruf Linie
  System.out.println
   ("978-3-937137-08-7  Java: Objektorientiert    24,90");
  Leerzeile_ausgeben(1); //1. Aufruf Leerzeile
  System.out.println
   ("978-3-937137-16-2  SQL                       19,90");
  Leerzeile_ausgeben(2); //2. Aufruf Leerzeile
  System.out.println
   ("978-3-937137-02-5  Webdesign & Web-Ergonomie 24,90");
  Linie_ausgeben(50,'-'); //2. Aufruf  Linie
 }
 public static void Leerzeile_ausgeben(int anzahl)
 {
   for (int zaehler = 1; zaehler <= anzahl; zaehler++)
     System.out.println(""); //1 Leerzeile
 }
 public static void Linie_ausgeben(int laenge, char zeichen)
 {
   for (int i = 1; i <= laenge; i++)
    System.out.print(zeichen);
   System.out.println();
 }
}
```

Das zugehörige Sequenzdiagramm zeigt die Abb. 6.9-1. Das Diagramm verdeutlicht, dass der Benutzer das Programm startet und dadurch die Methode `main()` aufgerufen wird. In der Methode `main()` wird als erstes die Prozedur `Linie_ausgeben()` mit den Parameterwerten `laenge = 50` und `zeichen = '='` aktiviert. Da die Prozedur `Linie_ausgeben()` zur selben Klasse wie die Methode `main()` gehört, wird dieser Aufruf auf die Aktionssequenz von `main()` »gestapelt«. Nach der Abarbeitung von `Linie_ausgeben()` wird wieder zur Methode `main()`, d. h. deren Aktionssequenz zurückgekehrt. Anschließend wird in `main()` die Prozedur `Leerzeile_ausgeben()` aufgerufen. Die Darstellung ist analog wie vorher. Ist `main()` am Ende, wird das Programm beendet und die Kontrolle geht an den Benutzer zurück.

6.10 Rekursion *

Oft lassen sich Probleme auf einfachere Teilprobleme zurückführen, wobei die Teilprobleme fast identisch mit dem Ursprungsproblem sind. Solche Probleme lassen sich durch rekursive Algorithmen lösen. Programmiertechnisch beschreibt man rekursive Algorithmen dadurch, dass Methoden sich selbst aufrufen.

Analogie

```
...
Ein Hund kam in die Küche
und stahl dem Koch ein Ei.
Da nahm der Koch die Kelle
und schlug den Hund zu Brei.
Da kamen viele Hunde
und gruben ihm ein Grab
und setzten ihm einen Grabstein
worauf geschrieben stand:
                            Ein Hund kam in die Küche
                            und stahl dem Koch ein Ei.
                            Da nahm der Koch die Kelle
                            und schlug den Hund zu Brei.
                            Da kamen viele Hunde
                            und gruben ihm ein Grab
                            und setzten ihm einen Grabstein
                            worauf geschrieben stand: ...
```

Sie kennen vielleicht dieses Volkslied. Aber was hat das mit Programmieren zu tun? Als Programm würde es folgendermaßen aussehen:

DemoRekursion

```
public class DemoRekursion
{
 public static void LinieAusgeben() //Methode
 {
   System.out.println
     ("-----------------------------------");
 }
 public static void LiedAusgeben(int anzahl)
 {
  LinieAusgeben();
  System.out.println("... Ein Hund kam in die Küche");
  System.out.println("    und stahl dem Koch ein Ei.");
  System.out.println("    Da nahm der Koch die Kelle");
  System.out.println("    und schlug den Hund zu Brei.");
  System.out.println("    Da kamen viele Hunde");
  System.out.println("    und setzten ihm einen Grabstein");
  System.out.println("    worauf geschrieben stand: ");
  anzahl = anzahl - 1;
  if (anzahl > 0 )
     LiedAusgeben(anzahl); //rekursiver Aufruf
 }
 public static void main (String args[])
 {
  LiedAusgeben(3);
 }
}
```

Das Volkslied gibt eine sogenannte **rekursive Situation** wieder, d. h. die Liedstrophe »ruft« sich sozusagen selbst auf. In einem Programm kann man dies dadurch realisieren, dass sich eine Methode selbst aufruft, d.h im Methodenrumpf wird die eigene Methode aufgerufen (siehe Programm `DemoRekursion`).

Auf Termination achten

Bei einem rekursiven Aufruf muss immer darauf geachtet werden, dass ein Parameter übergeben wird, der in der Methode verändert wird (aufwärts oder abwärts zählen). Dieser Parameterwert wird in einer `if`-Abfrage abgefragt und muss nach endlich vielen Aufrufen zum Abbruch der Aufrufkette führen. Sonst liegt eine nicht terminierende rekursive Situation vor, die Ihr Computersystem lahmlegt bzw. ständig beschäftigt.

Das obige Beispiel stellt natürlich *nicht* die Standardsituation für eine rekursive Problemlösung dar. In der Regel handelt es sich bei rekursiven Problemen um Probleme, die sich auf einfachere Teilprobleme zurückführen lassen, wobei die Teilprobleme *fast identisch* mit dem Ursprungsproblem sind. Solche Probleme lassen sich elegant durch rekursive Programme lösen.

Rekursion

Programmiertechnisch beschreibt man eine **Rekursion** bzw. rekursive Algorithmen dadurch, dass Methoden sich selbst direkt oder indirekt aufrufen. Einige mathematische Fragestellungen sind von Natur aus bereits rekursiv definiert.

In Ihrem Freundeskreis wird über den Rennsport diskutiert. Es stellt sich die Frage, wie viel mögliche Startanordnungen es gibt, wenn für sechs Autos sechs Startpositionen zur Verfügung stehen, die seitwärts versetzt hintereinander angeordnet sind. Kennen Sie die Lösung?

Jedes der sechs Autos kann auf »dem ersten Platz starten«. Es gibt also sechs Möglichkeiten, die *Pole Position* zu besetzen. Wenn der erste Startplatz besetzt ist, bleiben noch fünf Autos für den zweiten Startplatz, ist auch dieser besetzt, nur noch vier Kandidaten für den dritten Platz, und so fort. Für den vorletzten Platz bleiben schließlich nur noch zwei Autos übrig, und der letzte Platz muss mit dem »übrig gebliebenen« Auto besetzt werden.

Es gibt also 6 * 5 * 4 * 3 * 2 oder 6! = 720 Möglichkeiten, sechs unterscheidbare Objekte anzuordnen. Das Ausrufezeichen steht für **Fakultät** und wird auch so gelesen, also »Sechs Fakultät«.

Allgemein gilt: Die Anzahl der Anordnungen – Permutationen genannt – von n verschiedenen Elementen ist n!

Beispiel 1a

Die Fakultät einer natürlichen Zahl ist folgendermaßen definiert:

n! = n * (n-1) *. .. * 3 * 2 * 1, wobei 0! = 1! = 1.

Rekursiv lässt sich dies folgendermaßen beschreiben:
n! = n * (n-1)!
(n-1)! = (n-1) * (n-2)! usw.

Man erhält die Fakultät also, indem man sie mit der Fakultät der vorhergehenden Zahl multipliziert. Es ergibt sich folgende **Rekursionsrelation**:

n! = n * (n-1)!, wobei 0! = 1 ist.

Beispiel: 5! = 120; 6! = 6 * 5! = 6 * 120 = 720

Die Funktion zur Berechnung der Fakultät lässt sich unter Ausnutzung dieser Rekursionsrelation wie folgt als rekursive Funktion schreiben:

Fakultaet

```
//Rekursive Berechnung der Fakultät
import inout.Console;

public class Fakultaet
{
 public static int nFak(int n)
 {  //Annahme: n > 0
    assert (n > 0):"n muss größer 0 sein";
    if (n < 2)
      return 1; //Ende, wenn n < 2 ist
    else
       return n * nFak(n-1); //Rekursiver Aufruf
 }

 public static void main (String args[])
 {
  System.out.print("Anzahl Fakultät? ");
  int anzahl = Console.readInt();
  int fak = nFak(anzahl);
   System.out.print("Ergebnis: " + fak);
 }
}
```

Ein Beispiellauf sieht so aus:

```
Fakultät? 6
Ergebnis: 720
```

Der Aufruf `nFak(4)` bewirkt folgenden dynamischen Ablauf:

```
// 1. Aufruf
if  (4 < 2) Bedingung nicht erfüllt
    else return 4 * nFak(3) // Rekursiver Aufruf
  //2. Aufruf
  if  (3 < 2) Bedingung nicht erfüllt
    else return 3 * nFak(2)
    //Erneuter rekursiver Aufruf
      //3. Aufruf
      if  (2 < 2) Bedingung nicht erfüllt
        else return 2 * nFak(1)
        //Erneuter rekursiver Aufruf
        //4. Aufruf
```

```
            if  (1 < 2) Bedingung erfüllt
            then return 1 //Ende der Aufruffolge
        //Ende des 4. Aufrufs,
        //Rückkehr an Aufrufstelle
        //Berechnung des Ausdrucks
        //hinter return jetzt möglich
        return 2 * 1 = 2
    //Ende des 3. Aufrufs, Rückkehr an Aufrufstelle
    return 3 * 2 = 6
  //Ende des 2. Aufrufs, Rückkehr an Aufrufstelle
   return 4 * 6 = 24
//Ende des 1. Aufrufs und damit Ende der Berechnung
```

Als Ergebnis liefert der Funktionsaufruf `nFak = 24`.

Der Programmablauf zeigt, dass durch die rekursiven Aufrufe eine **dynamische Schachtelungsstruktur** aufgebaut wird. Bei jedem Aufruf wird die gerade bearbeitete Anweisung *unterbrochen* und zunächst dieselbe Funktion erneut gestartet.

Zu einem Zeitpunkt gibt es im obigen Fall also *gleichzeitig vier verschiedene* Aktivierungen derselben Methode, d. h. es existieren vier Methodendurchläufe mit unterschiedlichen lokalen Variablenwerten.

Zur Veranschaulichung möge die Abb. 6.10-1 dienen (angelehnt an UML-Sequenzdiagramme, siehe auch: »UML-Sequenzdiagramme«, S. 236).

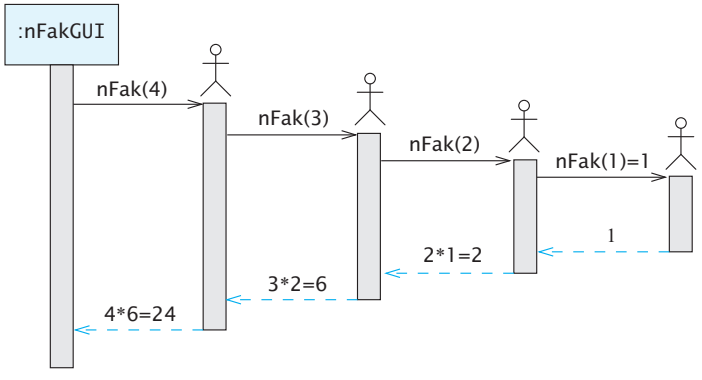


Abb. 6.10-1: Veranschaulichung des rekursiven Aufrufs des Programms `nFak`.

Ältester Bruder

Von vier Brüdern erhält der Älteste die Aufgabe `nFak(4)` zu berechnen. Diese Aufgabe ist ihm aber zu kompliziert. Da er die Rekursionsrelation `n! = n * (n-1)!` kennt, beschließt er, den zweitältesten Bruder damit zu beauftragen, zunächst einmal `(4-1)!` zu berechnen. Wenn er das Ergebnis von seinem Bruder erhält, ist er dann gern bereit, die Multiplikation `4 * (4-1)!` auszuführen.

Zweitältester Bruder

Der zweitälteste Bruder denkt, wenn sein ältester Bruder sich die Arbeit so vereinfacht, dann könne er das auch. Er delegiert daher die Aufgabe, `(3-1)!` zu berechnen, an den zweitjüngsten Bruder.

Zweitjüngster Bruder

Dieser, ebenfalls nicht dumm, gibt dem jüngsten Bruder die Aufgabe, `(2-1)!` zu ermitteln und das Ergebnis ihm dann mitzuteilen.

Jüngster Bruder

Der jüngste Bruder – etwas erbost darüber, dass man ihm nur eine so leichte Aufgabe zumutet – weiß sofort, dass `1! = 1` ist, sagt dem zweitjüngsten Bruder dieses Ergebnis und geht zornig davon.

Zweitjüngster Bruder

Der zweitjüngste Bruder multipliziert nun das ihm übermittelte Ergebnis mit der gemerkten Zahl `2`, teilt sein Ergebnis dem zweitältesten Bruder mit und entfernt sich ebenfalls.

Zweitältester Bruder

Der zweitälteste Bruder führt die Multiplikation `3 * 2` aus, denn er erinnert sich noch, dass er `3 * (3-1)!` berechnen sollte. Er sagt seinem ältesten Bruder seinen errechneten Wert `6` und verabschiedet sich dann, denn er wird nicht mehr benötigt.

Ältester Bruder

Der älteste Bruder berechnet nun noch `4 * 6` und hat damit das gestellte Problem gelöst. Freudestrahlend über seine intelligente Vereinfachung der Aufgabe reibt er sich die Hände und wartet auf eine neue Aufgabe.

Durch diese Darstellung werden einige Punkte noch deutlicher:

- Der älteste Bruder muss am längsten anwesend sein, da er zunächst eine Teilaufgabe delegieren und dann abwarten muss, bis er das Teilergebnis übermittelt bekommt. Über diese Zeitspanne hinweg muss er sich seine Aufgabe merken. Das heißt, die zuerst aufgerufene Methode existiert am längsten. Alle lokalen Variablen dieser Methode müssen so lange existieren, bis alle anderen Aufrufe beendet sind.
- Der jüngste Bruder wird nur kurzfristig benötigt; nach Erledigung seiner Aufgabe muss er sich nichts mehr merken. Das bedeutet, die zuletzt aufgerufene Methode existiert am kürzesten.
- Alle Brüder lösen ihre Teilaufgabe nach demselben Verfahren, d. h. nach demselben Algorithmus, jeder benutzt aber *andere* Daten. Das heißt, für jeden rekursiven Methodenaufruf wird ein neuer Speicherplatzbereich benötigt, der außerdem noch besonders verwaltet werden muss (dynamische Speicherver-

Beispiel 1b: iterative Lösung

Die Berechnung lässt sich auch ohne Rekursion lösen. Man spricht dann von einer iterativen Lösung:

Fakultaet Iterativ

```
static int nFak(int n)
{
    //Annahme: n > 0
    assert (n > 0):"n muss größer 0 sein";
```

```
    int fak = n;
    for (int i = n-1; i > 1; i--)
      fak = fak * i;
    return fak;
}
```

Vergleicht man die rekursive Lösung mit der iterativen Lösung, dann sieht man, dass die rekursive Fassung der Fakultätsberechnung *kürzer* als die nichtrekursive Fassung ist. Außerdem werden die Hilfsvariablen `Fak` und `i` *nicht* benötigt.

Aussagen

Anhand des Beispiels können folgende Aussagen zu rekursiven Methoden gemacht werden:

- Rekursion und Iteration sind verwandt. Jede rekursive Methode lässt sich im Prinzip durch eine iterative Methode ersetzen und umgekehrt.
- Der Programmcode einer rekursiven Methode ist im Allgemeinen kürzer als der Programmcode einer iterativen Methode.
- Die Laufzeit einer rekursiven Methode ist im Allgemeinen länger als die Laufzeit einer iterativen Methode.
- Damit eine rekursive Methode terminiert, d. h. nach endlich vielen Schritten beendet wird, muss im Anweisungsteil mindestens eine Bedingung existieren, die nach einer endlichen Zahl von Aufrufen erfüllt ist und den Abbruch der rekursiven Aufruffolge bewirkt.
- Der Laufzeitaufwand einer rekursiven Methode ergibt sich aus der Anzahl der rekursiven Aufrufe bei der Ausführung.

Bei diesem Beispiel handelt es sich um eine **direkte Rekursion**, da sich die Methode selbst aufruft.

6.11 Rekursion: Türme von Hanoi ***

Bei den »Türmen von Hanoi« müssen Scheiben von einem Turm zu einem anderen Turm transportiert werden, wobei einige Bedingungen einzuhalten sind. Es ergibt sich eine elegante rekursive Lösung. Die Analyse des Algorithmus ergibt, dass der Zeitaufwand proportional zu 2^n verläuft, d. h. bei großen n nimmt der Zeitaufwand sehr stark zu, sodass nur Probleme mit kleinem n gelöst werden können.

Legende

Nach einer alten Legende standen einmal drei goldene Säulen vor einem Tempel in Hanoi. Auf einer Säule befanden sich 100 Scheiben, jedesmal eine kleinere auf einer größeren Scheibe (in der Abb. 6.11-1 für 3 Scheiben gezeichnet).

Ein alter Mönch bekam die Aufgabe, den Scheibenturm von Säule 1 nach Säule 2 unter folgenden Bedingungen zu transportieren:

Abb. 6.11-1: Die Türme von Hanoi (hier mit drei Scheiben).

- Es darf jeweils nur die oberste Scheibe von einem Turm genommen werden.
- Es darf niemals eine größere Scheibe auf einer kleineren liegen.

Wenn der Mönch seine Arbeit erledigt habe, so berichtet die Legende weiter, dann werde das Ende der Welt kommen. Belastet mit dieser schweren Aufgabe setzte sich der alte Mönch in seinen Tempel und meditierte. Schon bald sah er ein, dass er zur Erledigung seiner Aufgabe auch Säule 3 benötigt.

Idee

Er dachte und meditierte weiter, bis ihm die göttliche Erleuchtung kam. Die Aufgabe konnte in 3 Schritten gelöst werden:

Schritt 1:

Transportiere den Turm, bestehend aus den oberen 99 Scheiben von Säule 1 nach Säule 3.

Schritt 2:

Transportiere die letzte, größte Scheibe von Säule 1 nach Säule 2.

Schritt 3:

Transportiere zum Schluss den Turm von 99 Scheiben von Säule 3 nach Säule 2.

Delegation

Da der Mönch schon sehr alt war, sah er ein, dass der 1. Schritt für ihn wohl doch zu viel Arbeit bedeutete. Er entschloss sich daher, diesen Schritt von seinem ältesten Schüler ausführen zu lassen. Wenn dieser mit seiner Arbeit fertig wäre, würde der alte Mönch selbst die große Scheibe von Säule 1 nach Säule 2 tragen, und dann würde er die Dienste seines ältesten Schülers nochmals in Anspruch nehmen.

Damit aber auch der älteste Lehrling des alten Mönchs keine zu schwere Arbeit zu verrichten habe, schlug der alte Mönch folgenden Algorithmus an die Tempeltür:

Algorithmus

Vorgehensweise, um einen Turm von n Scheiben von der einen nach der anderen Säule zu transportieren unter Verwendung einer dritten Säule:

- Besteht der Turm aus mehr als einer Scheibe, dann beauftrage deinen ältesten Lehrling, einen Turm von n-1 Scheiben von der ersten Säule zu der dritten Säule unter Verwendung der zweiten Säule zu transportieren;

- trage dann selbst eine Scheibe von der ersten zu der zweiten Säule;
- besteht der Turm aus mehr als einer Scheibe, dann bitte wiederum deinen ältesten Lehrling, einen Turm von n-1 Scheiben von der dritten Säule nach der zweiten Säule unter Verwendung der ersten Säule zu transportieren.

TuermeVonHanoi

In Java formuliert sieht der Algorithmus folgendermaßen aus:

```
//Rekursion am Beispiel der Türme von Hanoi
public class TuermeVonHanoi
{
 public static void transportiere
    (int n, int turm1, int turm2, int turm3)
 {
    if (n > 1)
       transportiere(n-1, turm1, turm3, turm2); //rekursiv. Aufruf
    System.out.println
      ("Schleppe Scheibe " + n +
       " von Turm " + turm1 + " nach Turm " + turm2);
    if (n > 1)
       transportiere(n-1, turm3, turm2, turm1); //rekursiv. Aufruf
 }
 public static void main (String args[])
 {
        transportiere(3,1,2,3);
 }
}
```


Der Programmlauf ergibt folgendes Ergebnis:

```
Schleppe Scheibe 1 von Turm 1 nach Turm 2
Schleppe Scheibe 2 von Turm 1 nach Turm 3
Schleppe Scheibe 1 von Turm 2 nach Turm 3
Schleppe Scheibe 3 von Turm 1 nach Turm 2
Schleppe Scheibe 1 von Turm 3 nach Turm 1
Schleppe Scheibe 2 von Turm 3 nach Turm 2
Schleppe Scheibe 1 von Turm 1 nach Turm 2
```

Gehen Sie das Java-Programm schrittweise durch. Machen Sie sich die Parameterübergabe und die innerhalb jedes Aufrufs gültigen Werte anhand von Wertetabellen klar. Zeichnen Sie die Türme und veranschaulichen Sie sich den Transport der Scheiben.

Interessant an dem Algorithmus ist, dass in ihm *keine* Zuweisung auftritt.

Die Darstellung der Abarbeitung des Algorithmus zeigt die dynamische Aufrufverschachtelung und die Aktionen, die in jedem Aufruf vorgenommen werden (Abb. 6.11-2). Für einen Turm mit 3 Scheiben ergibt sich folgende Ausführungsfolge:

```
transportiere(3, turm1, turm2, turm3); // 1. Aufruf
{
  if Auswahltest (3 > 1)
       transportiere(3-1, turm1, turm3, turm2);
     { // 2. Aufruf
```

```
        if Auswahltest (2 > 1)
            transportiere(2-1, turm1, turm2, turm3);
          { // 3. Aufruf
             if Auswahltest (1 > 1) Bedingung nicht erfüllt
             Ausdruck: Schleppe Scheibe1 von Turm1 nach Turm2;
             if Auswahltest (1 > 1)  Bedingung nicht erfüllt
          } // 3. Aufruf
          Ausdruck: Schleppe Scheibe2 von Turm1 nach Turm3;
          if Auswahltest (2 > 1)
             transportiere(2-1, turm2, turm3, turm1);
          { // 4. Aufruf
             if Auswahltest (1 > 1) Bedingung nicht erfüllt
             Ausdruck: Schleppe Scheibe1 von Turm2 nach Turm3;
             if Auswahltest (1 > 1) Bedingung nicht erfüllt
          } // 4. Aufruf
     } // 2. Aufruf
     Ausdruck: Schleppe Scheibe3 von Turm1 nach Turm2;
     if Auswahltest (3 > 1)
        transportiere(3-1, turm3, turm2, turm1);
     { // 5. Aufruf
         if Auswahltest (2 > 1)
             transportiere(2-1, turm3, turm1, turm2);
          { // 6. Aufruf
             if Auswahltest (1 > 1) Bedingung nicht erfüllt
             Ausdruck: Schleppe Scheibe1 von Turm3 nach Turm1;
             if Auswahltest (1 > 1) Bedingung nicht erfüllt
          } // 6. Aufruf
          Ausdruck: Schleppe Scheibe2 von Turm3 nach Turm2;
          if Auswahltest (2 > 1)
              transportiere(2-1, turm1, turm2, turm3);
          { // 7. Aufruf
             if Auswahltest (1 > 1) Bedingung nicht erfüllt
             Ausdruck: Schleppe Scheibe1 von Turm1 nach Turm2;
             if Auswahltest (1 > 1) Bedingung nicht erfüllt
          } // 7. Aufruf
     } // 5. Aufruf
} // 1. Aufruf
```

Die Darstellung der Abarbeitung des Algorithmus zeigt die dynamische Aufrufverschachtelung und die Aktionen, die in jedem Aufruf vorgenommen werden.

Obwohl die Problemlösung statisch sehr kurz und elegant ist, ergibt sich doch eine dynamische Komplexität bei der Ausführung.

Versuchen Sie folgende Fragen zu beantworten:

1. Wie viele Mönche werden in Abhängigkeit von der Anzahl der Scheiben n bei der Arbeit eingeschaltet, bevor die erste Scheibe geschleppt wird?
2. Wie sieht die Arbeit des i-ten Mönches aus?
3. Welcher Mönch verrichtet die meiste Arbeit, welcher die geringste?
4. Wodurch ist der Abbruch, d. h. die dynamische Endlichkeit, des Algorithmus sichergestellt?

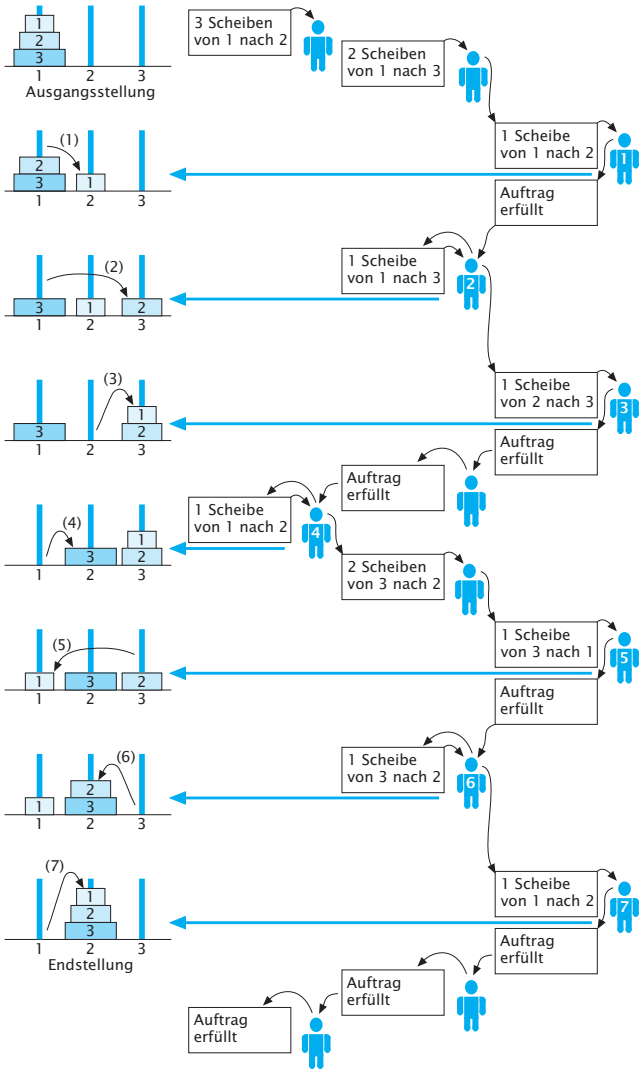


Abb. 6.11-2: Delegationsprinzip am Beispiel der »Türme von Hanoi«.

5 Wie groß ist der Speicher- und Zeitaufwand des Algorithmus?

6 Wie lange dauert es, bis die Arbeit an den Türmen von Hanoi beendet ist, wenn in jeder Sekunde eine Scheibe transportiert wird? Hat die Legende recht?

Anzahl benötigter Mönche: zu 1.
Für die Arbeit werden insgesamt n Mönche benötigt. Bevor eine Scheibe geschleppt wird, werden alle n Mönche eingeschaltet. Jeder Mönch reduziert das Problem um eine Scheibe und delegiert das reduzierte Problem.

Arbeit des i-ten Mönchs: zu 2.
Der i-te Mönch muss die (n−i+1)-te Scheibe 2^{i-1} mal schleppen.

Meiste Arbeit: zu 3.
Der jüngste Mönch verrichtet die meiste Arbeit (2^{n-1} mal die kleinste Scheibe schleppen), der älteste Mönch muss nur 1 Scheibe schleppen.

Termination: zu 4.
Termination ist gegeben, da bei jedem Aufruf die Scheibenzahl um 1 vermindert wird und bei der Scheibenzahl 1 *keine* Aufrufe mehr erfolgen.

Zeitaufwand: zu 5.
Da jeder Mönch 2^{i-1} mal eine Scheibe schleppt, transportieren alle Mönche zusammen
$\sum_{i=1}^{n} 2^{i-1} = 2^n - 1$ Scheiben.

Dieser Aufwand ist auch proportional dem Zeitaufwand des Algorithmus, da der Aufruf `SchleppeScheibe(...);` mit der Anzahl der Aufrufe übereinstimmt.

Der Zeitaufwand dieses Algorithmus ist also proportional 2^n, d. h. der Rechenzeitbedarf wird mit wachsendem n sehr schnell größer.

Der maximale Problemumfang, der mit einem solchen Algorithmus bewältigt werden kann, zeigt die Tab. 6.11-1, wobei für die Durchführung einer Rechenoperation eine Zeit von 10^{-6} Sekunden angenommen wird.

Zeitkomplexität	Maximaler Problemumfang bei 1 Sekunde	Maximaler Problemumfang bei 1 Minute	Maximaler Problemumfang bei 1 Stunde
2^n	19	25	31

Tab. 6.11-1: Zeit und zu bewältigender Problemumfang.

Die Tabelle sagt aus, dass für den Transport von 19 Scheiben eine Sekunde Rechenzeit benötigt wird, für den Transport von 25 Scheiben eine Minute und für den Transport von 31 Scheiben bereits eine Stunde. Dieser Algorithmus ist also für große n sehr rechenintensiv.

Der Speicheraufwand ist proportional n, da zu jedem Zeitpunkt maximal n Aufrufe aktiv sind.

zu 6. **Hat die Legende recht?**
Um 100 Scheiben von Säule 1 nach Säule 2 zu bringen, müssen 2^{100}-1 Scheiben transportiert werden.

Das ergibt: 2^{100}-1 ~ 1,2676 * 10^{30}. Wird zum Transport einer Scheibe eine Sekunde benötigt, dann ist der Turm in ~ 10^{30} Sekunden ~ 4 * 10^{22} Jahren von Säule 1 nach Säule 2 transportiert. Die Prophezeiung der Legende kann also durchaus in Erfüllung gehen.

6.12 Rekursion: direkt vs. indirekt **

Algorithmen können sich direkt oder indirekt rekursiv aufrufen. Eine rekursive Lösung liegt immer dann nahe, wenn sich das Originalproblem in einfachere Teilprobleme desselben Problems zerlegen läßt. Die meisten Programmiersprachen ermöglichen die rekursive Programmierung.

Direkt vs. indirekt

Ein Algorithmus kann direkt oder indirekt rekursiv aufgerufen werden. Beim direkten Aufruf ruft sich der Algorithmus selbst auf (siehe: »Rekursion«, S. 239, »Rekursion: Türme von Hanoi«, S. 244). Bei der **indirekten Rekursion** ruft ein Algorithmus A einen Algorithmus B auf, der seinerseits wieder A direkt oder indirekt aufruft.

Beispiel

DemoRekursion Indirekt

```
// Demo: Indirekter rekursiver Aufruf
// von zwei Methoden a und b
public class DemoRekursionIndirekt
{
 public static void a(String aText, int anzahl) //Operation a
 {
    if (anzahl > 0)
    {
      System.out.println(aText);
      anzahl--;
      //indirekter rekursiver Aufruf
      b("Test a und ", anzahl);
     }
 }
 public static void b(String bText, int anzahl)
 {
      System.out.println(bText);
      //indirekter rekursiver Aufruf
      a("   noch ein Test b", anzahl);
 }
 public static void main(String args[])
 {
    a("Start", 5); //1. Aufruf
 }
}
```

Folgendes Ergebnis wird ausgegeben:

```
Start
Test a und
   noch ein Test b
Test a und
   noch ein Test b
Test a und
   noch ein Test b
Test a und
   noch ein Test b
Test a und
```

Gehen Sie das Programm Schritt für Schritt durch. Nicht alle Programmiersprachen ermöglichen den rekursiven Aufruf von Algorithmen. Die Sprachen der ALGOL-Familie (ALGOL 60, Simula 67, ALGOL 68, PASCAL, MODULA-2, Ada) sowie PL/1, C und C++ erlauben rekursive Algorithmen, während FORTRAN und COBOL über dieses Konzept *nicht* verfügen. Es gibt jedoch auch Programmiersprachen, die überwiegend mit rekursiven Algorithmen arbeiten, z. B. LISP.

Einsatzbereiche

In der numerischen Mathematik treten nur wenige rekursive Probleme auf, mehr dafür in der Kombinatorik, der Übersetzungstechnik, der Linguistik und der Simulation. Auch Teile von Programmiersprachen sind rekursiv definiert, z. B. »Ausdruck« (indirekte Rekursion). Viele Sprachkonstrukte referenzieren sich direkt oder indirekt selbst.

Eigenschaften der Rekursion

Damit ein Problem ein Kandidat für eine rekursive Lösung ist, muss es folgende drei Eigenschaften besitzen:

1. Das Originalproblem muss sich in einfachere Exemplare desselben Problems zerlegen lassen.
2. Wenn all diese Teilprobleme gelöst sind, müssen diese Lösungen so zusammengesetzt werden können, dass sich eine Lösung des Ausgangsproblems ergibt.
3. Ein großes Problem ist derart in eine Folge weniger komplexer Teilprobleme zu zerlegen, dass diese Teilprobleme letztendlich so einfach werden, dass sie ohne weitere Unterteilung gelöst werden können.

Für ein Problem mit diesen Eigenschaften ergibt sich die rekursive Lösung auf eine ziemlich direkte Art:

- Der erste Schritt besteht darin, zu überprüfen, ob das Problem in die Kategorie einfacher Fall fällt. In diesem Fall wird das Problem direkt gelöst.
- Im anderen Fall wird das gesamte Problem in neue Teilprobleme aufgebrochen, die selbst durch die rekursive Anwendung des Algorithmus gelöst werden.

- All diese Lösungen werden dann schließlich zur Lösung des Ausgangsproblems zusammengesetzt.

Damit ein rekursiver Algorithmus terminiert, d. h. nach endlicher Zeit beendet wird, muss im Anweisungsteil mindestens eine Bedingung existieren (meist eine `if-then-else`-Auswahl), die nach einer endlichen Zahl von Aufrufen erfüllt ist und den Abbruch der Aufruffolgen bewirkt.

6.13 Datenabstraktion: Gemeinsame Daten *

Benötigen mehrere Methoden gemeinsame Daten, dann werden die Methoden und getrennt davon die gemeinsamen Daten in Form von Variablen und Konstanten zu einer sogenannten Datenabstraktion zusammengefasst. In Java geschieht dies durch eine Klasse. Die gemeinsamen Daten werden in der Klasse deklariert, getrennt davon die Methoden. Die Methoden der Klasse können auf die gemeinsamen Daten lesend und schreibend zugreifen.

Beliebig viele Methoden in einer Klasse

In Java können in einer Klasse (`class`) beliebig viele Methoden stehen, wobei eine Methode ein Hauptprogramm (`main`) sein muss. Natürlich sollen nur solche Methoden in einer Klasse als Einheit zusammengefasst werden, die fachlich etwas miteinander zu tun haben.

Gegenseitig aufrufen

Jede Methode kann jede andere Methode der Klasse aufrufen – mit oder ohne Parameterübergabe.

Sequenzielle Verarbeitung

Immer wenn eine Methode eine andere Methode aufruft, wartet sie hinter der Aufrufstelle, bis die aufgerufene Methode ihre Arbeit beendet hat, und setzt dann selbst ihren Programmablauf fort. Diese Art des Wartens auf das Ende aufgerufener Methoden bezeichnet man als sequenzielle Abarbeitung, das Programmieren solcher Programme als **sequenzielles Programmieren**. Bei nichtsequenziellen Programmen – man spricht dann von nebenläufigen Programmen – warten rufende Methoden nicht, bis die gerufene Methode ihre Arbeit erledigt hat, sondern arbeiten sofort nach dem Aufruf selbstständig weiter. Auf diese Thematik wird hier aber *nicht* weiter eingegangen.

Begrenzte Lebenszeit der Daten

Immer wenn eine Methode ihre Aufgabe erledigt hat, sind alle in dieser Methode gespeicherten Variablenwerte anschließend *nicht* mehr vorhanden.

Gemeinsame Daten

In vielen Anwendungsfällen sollen aber die Daten nach dem Ende einer Methode noch vorhanden sein, da mehrere Methoden nacheinander auf die Daten zugreifen. Die Daten gehören also *nicht* zu einer Methode, sondern gehören allen Methoden gemeinsam.

Zuordnung der Daten zur Klasse

In Java ist es möglich, Variablen und Konstanten der Klasse zuzuordnen und nicht einer einzelnen Methode. Jede Methode der Klasse kann auf die Variablen- und Konstantenwerte zugreifen. Die Werte bleiben solange erhalten, solange das Programm läuft.

Beispiel 1a

Es soll die Verkehrssteuerung für einen Tunnel (pro Fahrtrichtung eine Tunnelröhre mit einspuriger Verkehrsführung) vorgenommen werden. Folgende Anforderungen sollen berücksichtigt werden:

/1/ An der Einfahrt jeder Tunnelröhre steht eine Kamera, die die Kennzeichen der einfahrenden Fahrzeuge erfasst. In der Simulation kann zunächst von ganzen Zahlen ausgegangen werden.

/2/ Im Tunnel können nur eine bestimmte Anzahl Fahrzeuge gleichzeitig sein.

/3/ Am Tunnelausgang wird jedes ausfahrende Fahrzeug gezählt.

/4/ Für Notfälle muss es möglich sein, die im Tunnel befindlichen Fahrzeuge mit ihren Kennzeichen auszugeben, wobei die Reihenfolge der Einfahrtreihenfolge entsprechen muss.

/5/ Ist die Kapazität des Tunnels erschöpft, dann ist am Tunneleingang eine Ampel auf Rot zu schalten.

Abstraktion

Bei konkreten Problemstellungen ist immer zuerst zu überlegen, ob man von der konkreten Situation abstrahieren kann, um zu einer allgemeineren Problemstellung zu gelangen, deren Lösung vielleicht für verschiedene Anwendungen geeignet ist. Vielleicht gibt es für die allgemeinere Problemstellung auch bereits eine Lösung.

Warteschlange

Abstrakt betrachtet handelt es ich hier um eine sogenannte Warteschlangenverwaltung – ein bekanntes Problem in der Informatik. Eine **Warteschlange** *(queue)* speichert Elemente, wobei neu zu speichernde Elemente hinten an die Warteschlange angehängt werden (Methode Einfügen) und zu löschende Elemente vorne aus der Warteschlange (Methode Entfernen) entfernt werden (Abb. 6.13-1). Eine Warteschlange arbeitet also nach dem **FIFO-Prinzip** *(first in – first out)*.

Abb. 6.13-1: Prinzip einer Warteschlange.

Beispiel 1b

Zur Realisierung benötigt man in Java eine Klasse mit den beiden Methoden `einfuegen` und `entfernen` sowie einem Hauptprogramm für die Funktionsauswahl und Ausgabe der Warteschlange. Für die Laufzeit des Programms müssen die Daten, d. h. die Inhalte der Warteschlange, in einer Variablen gespeichert werden, die bei der Klasse angeordnet wird – und nicht bei einer einzelnen Methode. Zur Speicherung der Warteschlange kann ein sogenannter Ringpuffer verwendet werden, d. h. ein Feld, das man sich als Ring vorstellt (Abb. 6.13-2). In einer Variablen `anfang` wird sich gemerkt, wo die Warteschlange anfängt. In einer weiteren Variablen `ende` wird sich gemerkt, wo das nächste Element angehängt wird.

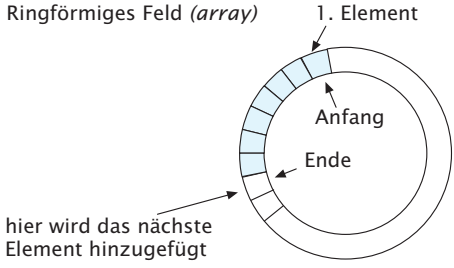


Abb. 6.13-2: Prinzip eines Ringspeichers.

Warteschlange

Es ergibt sich folgendes Programm:

```
// Beispiel für eine Datenabstraktion
import inout.Console;
public class Warteschlange
{
    private static final int LAENGE = 3;
    //Internes Gedächtnis
    private static int[] warteschlange = new int [LAENGE];
    private static int anfang = 0; // hier wird entfernt
    private static int ende = 0; // hier wird angefügt
    private static int anzahl = 0;

    public static void einfuegen(int element)
    {
        if (anzahl == LAENGE)
            System.out.println("Warteschlange ist voll");
        else
        {
            warteschlange[ende % LAENGE] = element;
            anzahl++;
            ende = (ende + 1) % LAENGE;
        }
    }
    public static int entfernen()
```

```
    {
        if (anzahl == 0)
        {
            System.out.println("Warteschlange ist leer");
            return 0;
        }
        else
        {
            int element = warteschlange[anfang % LAENGE];
            anzahl--;
            anfang = (anfang + 1) % LAENGE;
            return element;
        }
    }
    //Hauptprogramm
    public static void main (String args[])
    {
     int element;
     char auswahl;
     do
     {
      System.out.print
       ("Funktion Einfügen: 1 Entfernen: 2 ");
      System.out.println("Anzeigen: 3 Ende: 9");
      auswahl = Console.readChar();
      if (auswahl == '9') break;
      switch (auswahl)
      {
        case '1': System.out.print("Nummer des Elements: ");
                  element = Console.readInt();
                  einfuegen(element);
                  break;
        case '2': System.out.print("Entferntes Element: ");
                  element = entfernen();
                  System.out.println(element);
                  break;
        case '3': System.out.println
                    ("Inhalt der Warteschlange:");
                  System.out.print("Anfang ");
                  for (int i = anfang;
                               i < (anfang + anzahl); i++)
                  {
                     int Inhalt = warteschlange[i % LAENGE];
                     System.out.print(Inhalt + " ");
                  }
                  System.out.println(" Ende");
                  break;
         default: System.out.println
                        ("Auswahl wurde abgebrochen");
       }
      } while ( auswahl >= '1' && auswahl <='3');
   }
}
```

Ein Programmlauf sieht wie folgt aus:

```
Funktion Einfügen: 1 Entfernen: 2 Anzeigen: 3 Ende: 9
1
Nummer des Elements: 11
Funktion Einfügen: 1 Entfernen: 2 Anzeigen: 3 Ende: 9
1
Nummer des Elements: 12
Funktion Einfügen: 1 Entfernen: 2 Anzeigen: 3 Ende: 9
1
Nummer des Elements: 13
Funktion Einfügen: 1 Entfernen: 2 Anzeigen: 3 Ende: 9
1
Nummer des Elements: 14
Warteschlange ist voll Funktion
Einfügen: 1 Entfernen: 2 Anzeigen: 3 Ende: 9
3
Inhalt der Warteschlange:
Anfang 11 12 13  Ende
Funktion Einfügen: 1 Entfernen: 2 Anzeigen: 3 Ende: 9
2
Entferntes Element: 11
Funktion Einfügen: 1 Entfernen: 2 Anzeigen: 3 Ende: 9
3
Inhalt der Warteschlange:
Anfang 12 13  Ende
Funktion Einfügen: 1 Entfernen: 2 Anzeigen: 3 Ende: 9
```

1. Gehen Sie das Programm bitte Schritt für Schritt durch. Zeichnen Sie den Ringspeicher auf und tragen Sie für Beispiele die Werte ein. Machen Sie sich klar, wie die Modulo-Operationen funktionieren.
2. Passen Sie das Programm an die Problemstellung Verkehrssteuerung an (Speichern von Nummernschildern, Ampel auf Rot bzw. Grün usw.).

Java-Syntax

Die Java-Syntax für eine Klasse mit gemeinsamen Variablen und Konstanten zeigt die Abb. 6.13-3.

Private Daten

Auf gemeinsame Variablen und Konstanten einer Klasse sollte in der Regel nur von Methoden der eigenen Klasse aus zugegriffen werden. Daher sollte vor die Deklarationen das Schlüsselwort `private` gesetzt werden. Steht dort `public`, dann kann von außerhalb der Klasse direkt auf die Variablen und Konstanten zugegriffen werden, was im Allgemeinen aus Gründen der Wartbarkeit und Änderbarkeit unerwünscht ist.

Programmschema

Das Programmschema einer Klasse mit gemeinsamen Variablen und Konstanten zeigt die Abb. 6.13-4.

Datenabstraktion

In der Informatik bezeichnet man Daten, die von mehreren Methoden gemeinsam genutzt werden, als **Datenabstraktion**.

In der UML-Notation wird eine Klasse mit Datenabstraktion durch ein dreigeteiltes Rechteck dargestellt, wobei im obersten Rechteck der Klassenname, im mittleren Rechteck die gemeinsamen

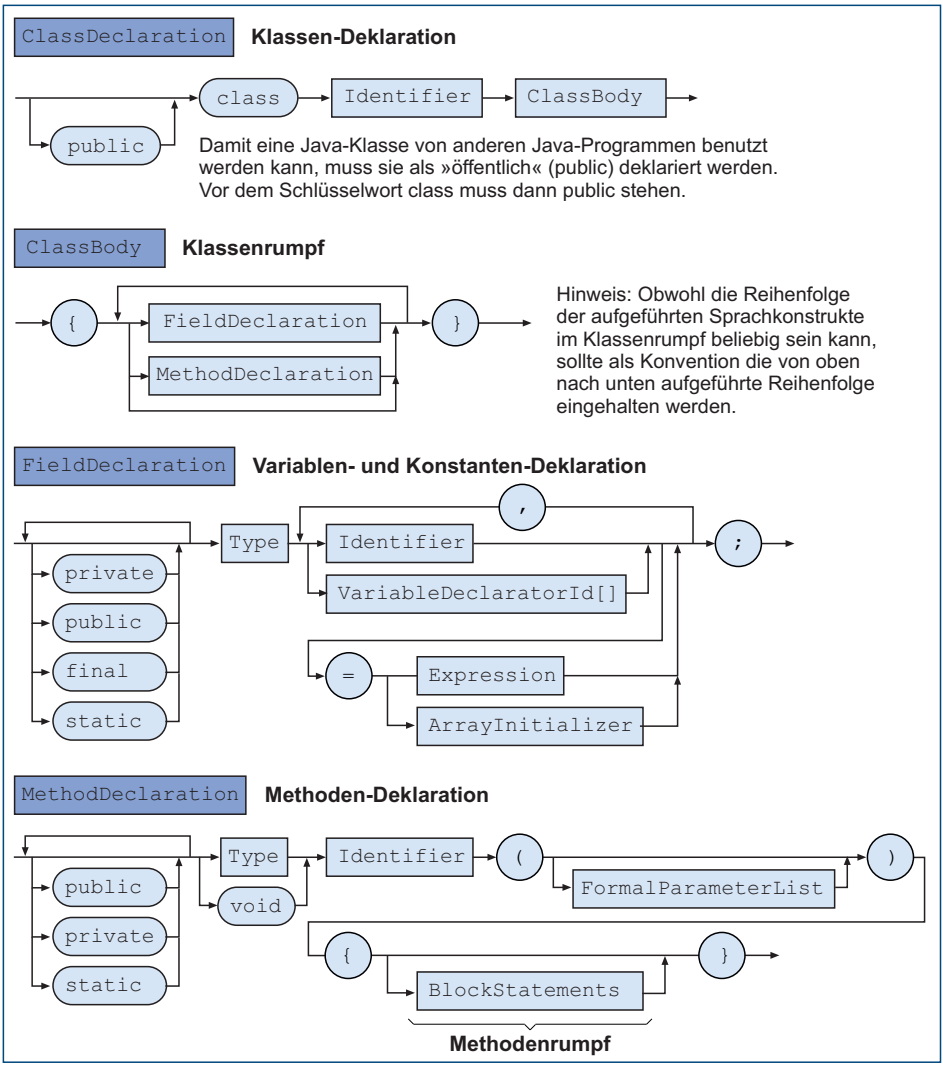


Abb. 6.13-3: Syntax einer Java-Klasse (Ausschnitt).

Variablen und Konstanten und im unteren Teil die Methoden aufgeführt sind. Die gemeinsamen Variablen und Konstanten werden unterstrichen dargestellt und heißen **Klassenvariablen** und Klassenkonstanten. In der UML werden gemeinsame Variablen und Konstanten zusammengefasst als Attribute bezeichnet. `private` wird durch ein - (Minus), `public` durch ein + (Plus) dargestellt. Die Notation für das Beispiel zeigt die Abb. 6.13-5.

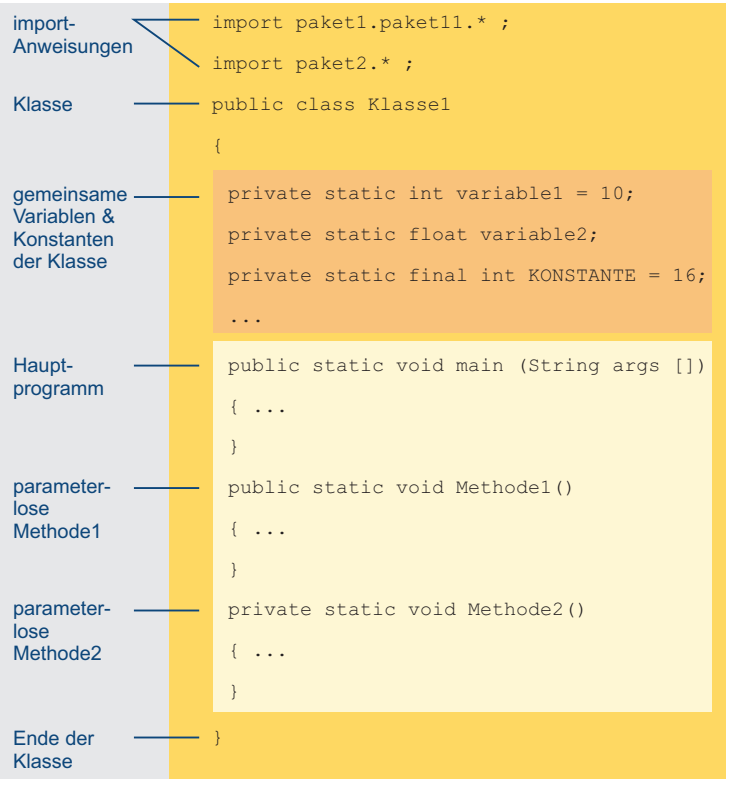


Abb. 6.13-4: Programmschema einer Klasse mit gemeinsamen Variablen und Konstanten sowie mehreren Methoden, die die gemeinsamen Variablen und Konstanten benutzen können.

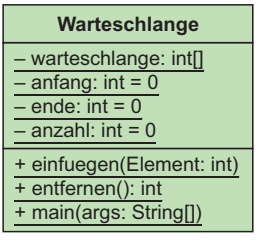


Abb. 6.13-5: UML-Darstellung einer Klasse, die eine Warteschlange realisiert.

6.14 OptiTravel: Gesamtlösung *

Die Anwendungs-Entwicklerin der Firma WebSoft, Frau Anton, erklärt der Junior-Programmiererin, Frau Jung, nochmals das Prinzip der Datenabstraktion. Sie bittet Frau Jung, die bisherigen Methoden, die für die Software `OptiTravel` entwickelt wurden, in

einer Klasse zusammenzufassen. Die bereits erstellten Tabellen sind als Konstanten und Variablen der Klasse zu deklarieren.

Frau Jung will gerade anfangen, als ihr der Projektleiter, Herr Pilot, mitteilt, dass die Firma ProManagement noch einige Änderungen der bisherigen Anforderungen (siehe »OptiTravel: Gespräch Auftraggeber – Auftragnehmer«, S. 51) sowie einige neue Wünsche per E-Mail mitgeteilt hat:

/1/ In einem Balkendiagramm sollen alle Zeiten, d. h. Auto zügig, Auto normal, Auto gemäßigt, Zug und Flug in aufsteigender Reihenfolge dargestellt werden. Die Darstellung soll etwa wie folgt aussehen:

```
Reisezeiten im Vergleich:

Flug : 2 h 45 min   |++++++++++++
AutoZ: 3 h 25 min   |+++++++++++++++
AutoN: 4 h 6 min    |++++++++++++++++++++
Zug  : 4 h 10 min   |++++++++++++++++++++
AutoG: 4 h 47 min   |++++++++++++++++++++++++
```

/2/ Der Benutzer soll für sein Auto seinen Benzin-/Dieselverbrauch pro 100 km sowie den aktuellen Preis pro Liter Benzin/Diesel angeben können. Fehlt die Angabe, dann sollen Voreinstellungswerte verwendet werden.

/3/ Der Benutzer soll sich alle Tabellen ansehen können.

Zuerst überlegt sich Frau Jung, wie die Bedienung für den Endbenutzer aussehen soll, auch wenn die Ein- und Ausgabemedien noch nicht feststehen. Da die Firma WebSoft über einen Mitarbeiter, Herrn Kaiser, verfügt, der für die **Software-Ergonomie** zuständig ist, bittet Frau Jung Herrn Kaiser um Rat. In einem Gespräch ermitteln sie folgende Lösung:
Der Benutzer soll aus folgenden drei Funktionen wählen können:

```
1: Start- und Zielort
2: Autodaten erfassen
3: Tabellen ausgeben
```

Dabei soll er die Funktionen in beliebiger Reihenfolge aufrufen und auch mehrfach ausführen können, bis er das Programm abbricht.

Wiederverwendung

Bevor Frau Jung mit dem Entwurf des Programms anfängt, sieht sie sich alle bisher für `OptiTravel` bereits erstellten Teilprogramme sowie die im Produktarchiv von WebSoft vorhandenen Programme an:

- `OptiTravelTabellen` (siehe: »OptiTravel: Tabellen«, S. 189:
  Die Tabellen können übernommen werden. Sie werden jedoch der Klasse zugeordnet, sodass alle Methoden darauf zugreifen können. Die Anweisungen für die Ausgabe der Tabellen fasst Frau Jung in einer neuen Methode `TabellenAusgeben()` zu-

sammen. Mit Hilfe von `DecimalFormat` (siehe »Java-Funktionen nutzen«, S. 225) verbessert sie noch das Ausgabeformat.

- `Funktionsauswahl` (siehe »OptiTravel: Funktionsauswahl«, S. 147:
  Dieses Programm kann fast unverändert übernommen werden. Es müssen nur die konkreten Funktionen beschrieben werden. In der `switch`-Anweisung erfolgen die Aufrufe der entsprechenden Methoden. Dieses Programm wird zum Hauptprogramm.
- `SortAuswahl` (siehe »Einfaches Sortieren«, S. 195):
  Da nur fünf Werte sortiert werden müssen, entscheidet sich Frau Jung für den Einsatz dieses Programms. Es muss jedoch auf den Vergleich zwischen ganzen Zahlen umgestellt werden. Zusätzlich wird in einer zweiten Dimension mitgeführt, auf was sich der jeweilige Wert bezieht (AutoZ, AutoN, AutoG, Zug, Flug). Die Werte in der zweiten Dimension werden ebenfalls mit umsortiert. Das ursprünglich für diesen Zweck vorgesehene Programm `Vergleich` (siehe »OptiTravel: Zeitvergleich«, S. 145) ist nicht geeignet, da es sich nicht sinnvoll auf fünf Vergleichswerte erweitern lässt.
- `Balkendiagramm` (siehe »OptiTravel: Balkendiagramm«, S. 179):
  Frau Jung stellt fest, dass sie die Grundstruktur dieses Programms übernehmen kann. Da die Firma ProManagement jedoch drei Fahrstile bei den Autofahrern unterscheiden will (Auto zügig, Auto normal, Auto gemäßigt) ergeben sich zusammen mit Flug und Zug fünf Vergleichswerte, die als Balkendiagramm dargestellt werden müssen. Das bisherige Programm ist nur auf drei Vergleichswerte ausgelegt. Sie legt folgenden Prozedurkopf fest:
  ```
  private static void Balkendiagramm_erstellen  (int[ ][ ] zeitenSortiert)
  ```
  Das Feld ist jetzt zweidimensional, da in der zweiten Dimension mitgeführt wird, um welche Zeiten es sich handelt (autoZ, autoN, autoG, zug, flug).

Entwurf

Als Konsequenz aus dieser Bedienung ergibt sich für Frau Jung, dass sie für die Autodaten, d. h. Benzinverbrauch und Benzinkosten, Voreinstellungen vorsehen muss.

Überlegen Sie warum?

Außerdem müssen diese Variablen der Klasse zugeordnet werden und *nicht* der Funktion `Autodaten_erfassen`, da die Funktion 1 nach dem Ende der Funktion 2 noch diese Daten benötigt. Die Entfernungs- und Zeittabellen müssen ebenfalls der Klasse zugeordnet werden, da verschiedene Methoden diese Daten benötigen.

Frau Jung stellt alle Konstanten und Variablen, die der Klasse zugeordnet werden, und alle benötigten Methoden mit ihren Parametern zusammen und zeichnet ein UML-Klassendiagramm (Abb. 6.14-1). Sie berücksichtigt dabei, dass in einem UML-Diagramm alle Variablen, die für andere Programme nicht sichtbar sein sollen, durch ein vorangestelltes Minus gekennzeichnet werden. Das Hauptprogramm wird durch ein vorangestelltes Pluszeichen als öffentlich gekennzeichnet. Alle anderen Methoden werden durch ein vorangestelltes Minus-Zeichen als private Methoden gekennzeichnet. In Java werden private Variablen und Methoden durch das Schlüsselwort `private`, öffentliche Methoden durch das Schlüsselwort `public` markiert.

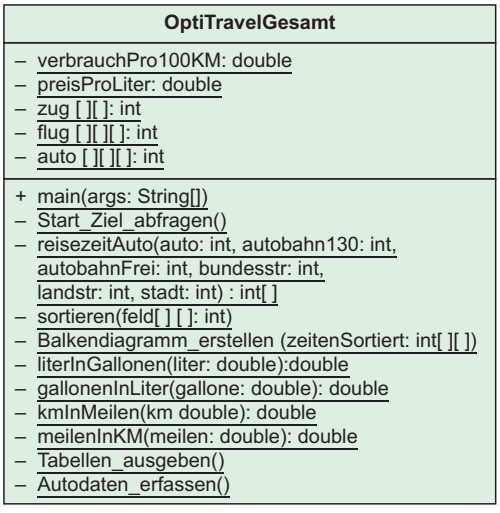


Abb. 6.14-1: Klassendiagramm für das Programm OptiTravelGeamt.

Frau Jung kopiert alle bereits vorhandenen Methoden sowie die Tabellen in die Klasse `OptiTravelGesamt`, nimmt die notwendigen Anpassungen vor und programmiert noch fehlende Teile hinzu. Das Programm wird bereits sehr umfangreich (über 500 Codezeilen). Ein Ausschnitt sieht wie folgt aus:

Programm

OptiTravel Gesamt

```
/***********************************************
OptiTravel-Gesamt: Berechnung der Reisezeit
***********************************************/
import inout.Console;
import java.text.DecimalFormat;

public class OptiTravelGesamt
{
  //Daten, die zur Klasse gehören
  static private double verbrauchPro100KM = 10.0,
   preisProLiter = 1.4;
```

```
 static private final int Berlin = 0, Hamburg = 1, Muenchen = 2,
  Koeln = 3, Frankfurt = 4, Dortmund = 5, Stuttgart = 6,
  Essen = 7, Duesseldorf = 8, Bremen = 9;

  static private final String[] staedte =
  {"Ber","Ham","Mue","Koe","Fra","Dor","Stu","Ess","Due","Brm"};

  //Alle Tabellen ------------------------------------------
  //....
  //Hauptprogramm*****************************************
  public static void main (String args[])
  {
  char auswahl;
  for(;;)
   {
    System.out.println("Bitte Funktion auswählen:");
    System.out.println("1: Start- und Zielort");
    System.out.println("2: Autodaten erfassen");
    System.out.println("3: Tabellen ausgeben");
    System.out.println("Abbruch: 9");
    System.out.println("Bitte Ziffer 1, 2, 3 oder 9 eingeben:");

    auswahl = Console.readChar();
    if (auswahl =='9') break;

    switch (auswahl)
    {
     case '1': Start_Ziel_abfragen(); break;//Aufruf XXXXXXXXXXX
     case '2': Autodaten_erfassen(); break;//Aufruf XXXXXXXXXXXX
     case '3': Tabellen_ausgeben(); break; //Aufruf XXXXXXXXXXXX
     default: System.out.println("Fehlerhafte Eingabe: " +
              "Bitte nur 1, 2, 3 oder 9 eingeben");
              continue;
    }
   }
   System.out.println("Ende des Programms");
}//Ende Hauptprogramm*****************************************

private static void Start_Ziel_abfragen() //*******************
{
   final String [] staedte =
    {"Berlin","Hamburg","München","Köln","Frankfurt","Dortmund",
     "Stuttgart","Essen","Düsseldorf","Bremen"};
   int startort, zielort;

   for(;;)
   {
     System.out.println("Berlin = 0, Hamburg = 1, Muenchen = 2,"+
      "\nKoeln = 3, Frankfurt = 4, Dortmund = 5, Stuttgart = 6,"+
      "\nEssen = 7, Duesseldorf = 8, Bremen = 9");

     System.out.println("Bitte Startort eingeben (als Ziffer): ");
     startort = Console.readInt();
     if(!(startort >= 0 && startort <= 9))
     {
       System.out.print
```

```
     ("Fehlerhafte Eingabe: Nur Ziffer 0 bis 9 eingeben!");
    continue;
  }
  System.out.println("Bitte Zielort eingeben (als Ziffer): ");
  zielort = Console.readInt();
  if(!(zielort >= 0 && zielort <= 9))
  {
    System.out.println
     ("Fehlerhafte Eingabe: Nur Ziffer 0 bis 9 eingeben!");
    continue;
  }

  if(zielort == startort)
  {
    System.out.println
    ("Fehlerhafte Eingabe: Startort ist gleich Zielort!");
    continue;
  }
  break;
}

System.out.println("\nFolgende Transportmittel gibt es von " +
 staedte[startort] + " nach " + staedte[zielort]);

//Wegen Dreieckstabellen
int startort2 = startort;
int zielort2 = zielort;

if (zielort < startort)
{
    //Vertauschen für Dreieckstabellen
    zielort2 = startort;
    startort2 = zielort;
}

//Zugverbindung
int zugInMinuten = zug[zielort2][startort2];
int stundenUndMinuten[] = minutenInStunden(zugInMinuten);//Aufruf
System.out.println("Zug: " + stundenUndMinuten[0] +
 " Stunden " + stundenUndMinuten[1] + " Minuten");
System.out.println();

//Flugverbindung
final int checkInZeitInMinuten = 45;
int flugInMinuten = flug[startort][zielort][0];

int autoanfahrtInMinuten = flug[startort][zielort][1];
int flugGesamtzeitInMinuten = flugInMinuten +
  autoanfahrtInMinuten + checkInZeitInMinuten;
stundenUndMinuten = minutenInStunden(flugInMinuten);//Aufruf

if (flugInMinuten == 0)
{
 System.out.println("Keine Flugverbindung vorhanden");
 flugGesamtzeitInMinuten = 0;
}
```

```
else
{
 System.out.println("Flug: " + stundenUndMinuten[0] +
 " Stunden " + stundenUndMinuten[1] + " Minuten");
 if (flug[startort][zielort][2]!= -1)
  System.out.println("Flughafen: "
  + staedte[flug[startort][zielort][2]]);


stundenUndMinuten = minutenInStunden(autoanfahrtInMinuten);
System.out.println
  ("Autoanfahrt zum Flughafen (von der Innenstadt aus): "
+ stundenUndMinuten[0] +
 " Stunden " + stundenUndMinuten[1] + " Minuten");

stundenUndMinuten = minutenInStunden(flugGesamtzeitInMinuten);
System.out.println("Gesamtzeit Flug: " + stundenUndMinuten[0] +
 " Stunden " + stundenUndMinuten[1] +
 " Minuten einschl. Check-in-Zeit von. " +
 checkInZeitInMinuten + " Minuten");
}
System.out.println();

//Autoverbindung
int autoInKM = auto[zielort2][startort2][0];
int autobahn130InKM = auto[zielort2][startort2][1];
int autobahnFreiInKM = auto[zielort2][startort2][2];
int bundesstrInKM = auto[zielort2][startort2][3];
int landstrInKM = auto[zielort2][startort2][4];
int stadtInKM = auto[zielort2][startort2][5];

System.out.println("Autoentfernung: " + autoInKM + " km (" +
  kmInMeilen(autoInKM)+ " Meilen)");

//Reisezeiten berechnen
//Aufruf ReisezeitAutoXXXXXXXXXXXXXXXX
int[]autoZeiten =
 reisezeitAuto(autoInKM, autobahn130InKM, autobahnFreiInKM,
 bundesstrInKM, landstrInKM, stadtInKM);
System.out.println();
System.out.println("Reisezeiten im Vergleich:");

//Sortieren vor Ausgabe
final int autoZ = 0, autoN = 1, autoG = 2, zug = 3, flug = 4;
int[][] zeiten = {{autoZeiten[0], autoZ}, {autoZeiten[1],autoN},
{autoZeiten[2],autoG}, {zugInMinuten,zug},
{flugGesamtzeitInMinuten, flug}};
//Aufruf Sortieren XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
sortieren(zeiten);
//Aufruf Balkendiagramm XXXXXXXXXXXXXXXXXXX
//Grafische Ausgabe als Balkendiagramm
Balkendiagramm_erstellen (zeiten);
//Kosten Autofahrt
System.out.println("Kosten für die Autobenutzung:");
System.out.println("Preis pro Liter (pro Gallone): " +
  preisProLiter + " ("
```

```
      + literInGallonen(preisProLiter) + ")") ;//Aufruf
    System.out.println("Verbrauch pro 100 km: "
      + verbrauchPro100KM + " Liter");
    System.out.println("Kosten für " + autoInKM +
      " km: "
     + ((int)((autoInKM / 100.0f * verbrauchPro100KM *
        preisProLiter)*100))/100 + " Euro");
    System.out.println();
 } //Ende Start_Ziel_erfassen ************************

// Alle anderen Methoden ****************************
//...
```

Frau Jung zeichnet noch einige Sequenzdiagramme, um sich über den zeitlichen Ablauf der Methoden Aufschluss zu verschaffen (Abb. 6.14-2).

Hinweis

Im UML-Sequenzdiagramm dürfen nur Methoden auftreten, die auch im UML-Klassendiagramm vorhanden sind. Die Umkehrung gilt nicht.

Test

Warum können in einem UML-Klassendiagramm mehr Methoden vorhanden sein, als in einem zugehörigen Sequenzdiagramm?

Anschließend plant sie, das Programm systematisch zu testen, indem sie jeweils einen Startort wählt und dann alle anderen Zielorte angibt.

Wie viele Möglichkeiten gibt es?

Ein typischer Programmlauf (Ausschnitt) sieht wie folgt aus:

```
Bitte Funktion auswählen:
1: Start- und Zielort
2: Autodaten erfassen
3: Tabellen ausgeben
Abbruch: 9
Bitte Ziffer 1, 2, 3 oder 9 eingeben:
1
Berlin = 0, Hamburg = 1, Muenchen = 2,
Koeln = 3, Frankfurt = 4, Dortmund = 5, Stuttgart = 6,
Essen = 7, Duesseldorf = 8, Bremen = 9
Bitte Startort eingeben (als Ziffer):
7
Bitte Zielort eingeben (als Ziffer):
2

Folgende Transportmittel gibt es von Essen nach München
Zug: 5 Stunden 30 Minuten

Flug: 1 Stunden 10 Minuten
Flughafen: Düsseldorf
Autoanfahrt zum Flughafen (von der Innenstadt aus):
  0 Stunden 30 Minuten
Gesamtzeit Flug: 2 Stunden 25 Minuten
  einschl. Check-in-Zeit von. 45 Minuten
```

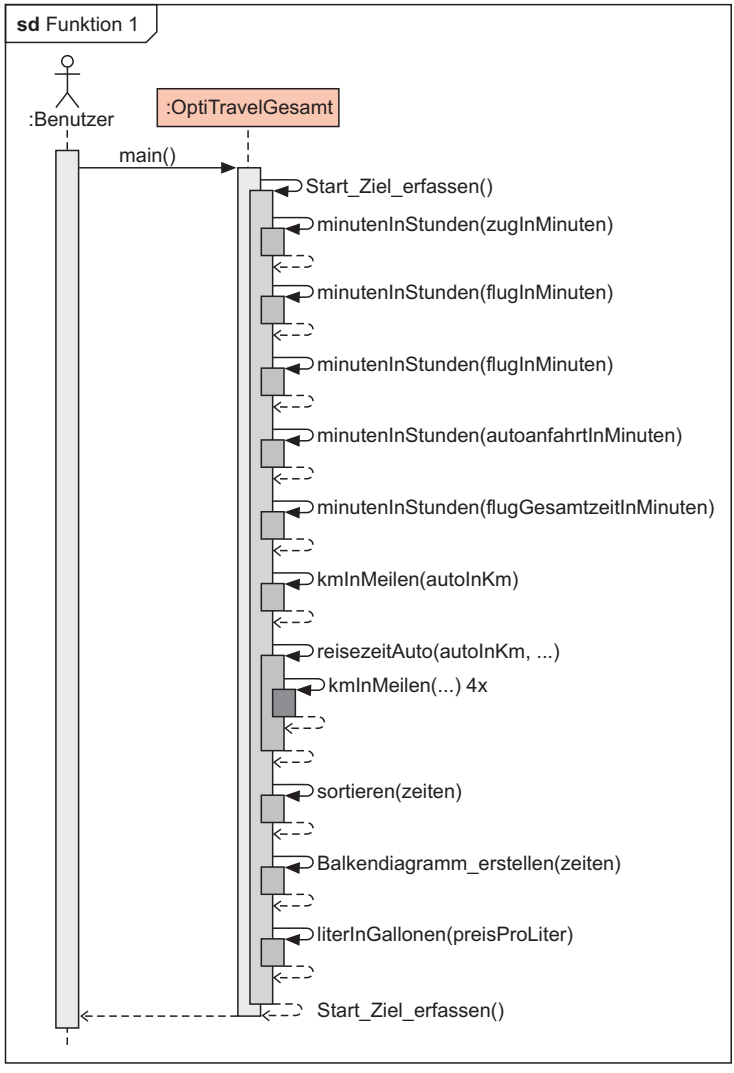


Abb. 6.14-2: Ablauf der Funktion 1 von `OptiTravelGesamt`, dargestellt als UML-Sequenzdiagramm.

```
Autoentfernung: 530 km (329.3267410993576 Meilen)
Autobahn mit 130 km/h Beschränkung: 200 km (124.27424192428589 Meilen)
Autobahn ohne Geschw.-Beschränkung: 300 km (186.41136288642883 Meilen)
Bundestraße: 0 km (0.0 Meilen)
Landstraße: 0 km (0.0 Meilen)
Stadtverkehr: 30 km (18.641136288642883 Meilen)

Reisezeiten im Vergleich:
```

```
Flug : 2 h 25 min   |++++++++++
AutoZ: 3 h 48 min   |+++++++++++++++
AutoN: 4 h 33 min   |++++++++++++++++++
AutoG: 5 h 19 min   |+++++++++++++++++++++
Zug  : 5 h 30 min   |++++++++++++++++++++++
Legende:
AutoZ: Autofahren zügig
AutoN = Autofahren normal (+ 20%)
AutoG = Autofahren gemässigt (+ 40% gegenüber AutoZ)

Kosten für die Autobenutzung:
Preis pro Liter (pro Gallone): 1.4 (0.36988000273704524)
Verbrauch pro 100 km: 10.0 Liter
Kosten für 530 km: 74 Euro
```

Lassen Sie das Programm auf Ihrem Computersystem laufen und rufen Sie die verschiedenen Funktionen auf. Sehen Sie sich dazu den entsprechenden Programmcode an.

6.15 Vom Problem zur Lösung: Teil 4 **

Stehen für eine Problemlösung Datenstrukturen, Prozeduren, Funktionen sowie die Datenabstraktion zur Verfügung, dann vergrößert sich dadurch der Problemlöseraum. Außerdem erhöhen sich die Anzahl der Lösungswege sowie die zu treffenden Entscheidungen.

In der Regel sollte folgende **Entscheidungsreihenfolge** eingehalten werden:

1. Überlegen, welche **Datenstruktur** oder welche Datenstrukturen für die Problemlösung geeignet sind. Mögliche Alternativen durchspielen.
2. **Algorithmen** konzipieren, die auf der gewählten Datenstruktur arbeiten.
3. Die Algorithmen in **Prozeduren** und **Funktionen** unterteilen, so dass sie jeweils nur ein Problem lösen bzw. eine Aufgabe erledigen. Bei der Konzeption von Prozeduren und Funktionen darauf achten, dass die Anzahl und Art der **Parameter** optimal gewählt wird, um eine allgemeine und flexible Lösung zu gewährleisten. Prüfen, ob eine **rekursive** Lösung besser als eine iterative ist oder nicht.
4. Prüfen, ob eine **Datenabstraktion** benötigt wird, d. h. die Datenstruktur muss noch zur Verfügung stehen, wenn einzelne Algorithmen in Form von Prozeduren und/oder Funktionen bereits beendet sind.

Beispiel

Ein Konferenzzentrum verfügt über eine Vielzahl von Konferenzräumen unterschiedlicher Größe. Jeder Raum kann auf verschiedene Art und Weise bestuhlt werden. Die möglichen Bestuhlungsarten zeigt die Abb. 6.15-1.

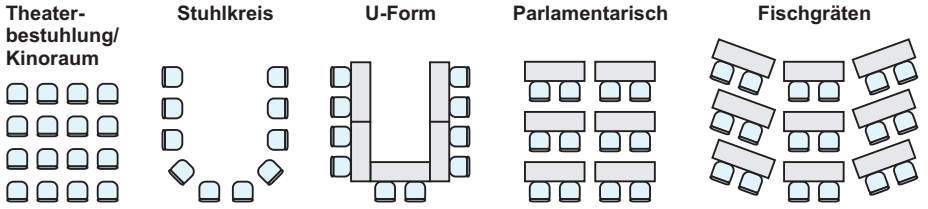


Abb. 6.15-1: Bestuhlungsarten.

Für jeden Raum soll ermittelt werden, für wie viele Personen er sich in Abhängigkeit von der Bestuhlungsart eignet.

Außerdem soll auf eine Anfrage hin in Abhängigkeit von der Personenanzahl und der gewünschten Bestuhlungsart ermittelt werden, welche Räume dafür zur Verfügung stehen.

Für jeweils eine Person wird folgender Platz benötigt (fiktive Größen):

- Theaterbestuhlung: 0,7 m Breite + 1,5 m Länge. Zusätzlich werden für Gänge links, rechts und hinten jeweils 1,5 m benötigt. Für Rednerpult und Platzbedarf für den Redner sind 2 m in der Länge abzuziehen. Sind mehr als 10 Personen in einer Reihe, dann ist ein zusätzlicher Gang der Breite von 1,5 m pro 10 Personen vorzusehen.
- Stuhlkreis: Basiert auf der Theaterbestuhlung. Pro Reihe sind 2 Personen zu platzieren + 1 zusätzliche vollständige Reihe mit Personen. Ein zusätzlicher Gang pro 10 Personen entfällt.
- U-Form: Basiert auf der Stuhlkreisberechnung. Wegen dem Platz für die Tische sind bei der vollständigen Reihe mit Personen 2 Personen abzuziehen. Bei den Reihen ist eine Reihe abzuziehen.
- Parlamentarisch: Basiert auf der Theaterbestuhlung, jedoch wird die Anzahl der Reihen halbiert.
- Fischgräten: Basiert auf der parlamentarischen Bestuhlung, jedoch ist die Platzlänge pro Person auf 2 m zu vergrößern.

Folgende Konferenzräume stehen zur Verfügung:

- Raum 1: Breite: 20 m, Länge 25 m
- Raum 2: Breite: 30 m, Länge 60 m
- Raum 3: Breite: 15 m, Länge 35 m
- Raum 4: Breite: 10 m, Länge 10 m
- Raum 5: Breite: 7 m, Länge 13 m

Mögliche Lösungsschritte sind:

1. Schritt

Semiformale Lösung

Für die Speicherung der Breiten und Längen der Konferenzräume wird ein zweidimensionales Feld als **Datenstruktur** gewählt.

Formale Java-Lösung

```
//Räume: Breite, Länge, Index 0 nicht benutzt,
//da Räume ab 1 gezählt werden
static private int[][] raeume =
  {{0,0},{20,25},{30,60},{15,35},{10,10},{7,13}};
```

2. Schritt

Semiformale Lösung

Um die Anzahl der Personen zu ermitteln, müssen alle Räume durchlaufen werden. Pro Raum müssen alle Bestuhlungsarten durchlaufen werden. Es sind also 2 ineinander geschachtelte Zählschleifen erforderlich. Pro Bestuhlungsart muss die Anzahl der Personen separat berechnet werden.

Formale Java-Lösung

```
//Alle Räume durchlaufen
for (int raum = 1; raum < raeume.length; raum ++)
{
  //Alle Bestuhlungsarten durchlaufen
  String bestuhlung = "Fehler";
  int anzahlPersonen = 0;

  for (int art = 1; art <= 5; art ++)
  {
    switch (art)
    {
      case 1:
         anzahlPersonen =
            berechneTheaterbestuhlung
               (raeume[raum][0], raeume[raum][1]);
            bestuhlung = "Theaterbestuhlung";
            break;
      //usw.

        default: anzahlPersonen = -1;
      }

      if (anzahlPersonen != -1)
       System.out.println("Anzahl Personen: "
         + anzahlPersonen + " in Raum " + raum
         + " mit " + bestuhlung);
    }
}
```

3. Schritt

Semiformale Lösung

Die Problemstellung gibt bereits einen Hinweis darauf, dass die Bestuhlungsberechnungen aufeinander aufbauen. Die Berechnung der U-Form kann auf die Berechnung des Stuhlkreises zurückgeführt werden.

Eine genauere Analyse ergibt jedoch, dass es sinnvoll ist, die Berechnung der Anzahl Reihen und die Berechnung der Personen pro Reihe als getrennte Funktionen zu programmieren.

Formale Java-Lösung

Es ergeben sich folgende Funktionsköpfe:

```
int personenProReihe
  (double raumBreite, double gangbreite) {...}
int anzahlReihen
  (double raumLaenge, double platzlaenge) {...}
int berechneTheaterbestuhlung
  (double raumBreite, double raumLaenge) {...}
int berechneStuhlkreis
  (double raumBreite, double raumLaenge) {...}
int berechneUForm
  (double raumBreite, double raumLaenge) {...}
int berechneParlamentarisch
  (double raumBreite, double raumLaenge) {...}
int berechneFischgraeten
  (double raumBreite, double raumLaenge) {...}
```

4. Schritt

Semiformale Lösung

Um geeignete Räume zu ermitteln, muss die Personenanzahl und die gewünschte Bestuhlungsart eingelesen werden. Anschließend sind alle Räume zu durchlaufen und pro Raum die Anzahl der Personen entsprechend der gewünschten Bestuhlungsart zu ermitteln.

Formale Java-Lösung

```
public static void raeumeErmitteln()
{
   int personenanzahl = Console.readInt();
    System.out.println("Theater = t, Stuhlkreis = s,
    U-Form = u, Parlamentarisch = p, Fischgräten = f:");
     char artChar = Console.readChar();

    //Alle Räume durchlaufen
    System.out.println("Folgende Räume sind geeignet:");
    for (int raum = 1; raum < raeume.length; raum ++)
    {
       String bestuhlung = "Fehler";
       int anzahlPersonen = 0;
       switch (artChar)
       {
         case 't':
           anzahlPersonen =
              berechneTheaterbestuhlung
                 (raeume[raum][0], raeume[raum][1]);
            bestuhlung = "Theaterbestuhlung"; break;
        // usw.
          default: anzahlPersonen = -1;
        }

       if (anzahlPersonen != -1 &&
```

```
            anzahlPersonen >= personenanzahl)
        {
          System.out.println("Anzahl Personen: "
            + anzahlPersonen + " in Raum " + raum
            + " mit " + bestuhlung);
        }
    }
 }
```

5. Schritt

Semiformale Lösung

Da es mehr als eine Funktion gibt, muss der Benutzer die Möglichkeit haben, die gewünschte Funktion auszuwählen. Da die Funktionen unabhängig voneinander sind, müssen die Informationen über die Räume über die Funktionen hinaus vorhanden sein. Daher ist eine Datenabstraktion notwendig.

Formale Java-Lösung

```
public static void main (String args[])
{
 char auswahl;
 for(;;)
  {
   System.out.println
     ("1: Personen pro Raum und Bestuhlungsart ermitteln");
   System.out.println
     ("2: Räume für Anzahl Personen und Bestuhlungsart");
   System.out.println("Abbruch: 9");
   System.out.println("Bitte Ziffer 1, 2 oder 9 eingeben:");

   auswahl = Console.readChar();
   if (auswahl =='9') break;

   switch (auswahl)
   {
     case '1': anzahlPersonenermitteln(); break;//Aufruf
     case '2': raeumeErmitteln(); break;//Aufruf
     default: System.out.println("Fehlerhafte Eingabe: " +
         "Bitte nur 1, 2 oder 9 eingeben");
         continue;
    }
 }
System.out.println("Ende des Programms");
}
```

Das gesamte Programm sieht wie folgt aus:

```
/**
 * Anzahl Personen pro Konferenzraum
 * in Abhängigkeit von der Bestuhlungsart
 * @author Helmut Balzert
 * @version V0.3
 */

import inout.Console;
```

```
public class Konferenzzentrum
{
  //Räume: Breite, Länge, Index 0 nicht benutzt,
  //da Räume ab 1 gezählt werden
  static private int[][] raeume =
    {{0,0},{20,25},{30,60},{15,35},{10,10},{7,13}};

  //Hauptprogramm*****************************************
  public static void main (String args[])
  {
   char auswahl;
   for(;;)
    {
     System.out.println("Bitte Funktion auswählen:");
     System.out.println
       ("1: Personen pro Raum und Bestuhlungsart ermitteln");
     System.out.println
       ("2: Räume für Anzahl Personen und Bestuhlungsart");
     System.out.println("Abbruch: 9");
     System.out.println("Bitte Ziffer 1, 2 oder 9 eingeben:");

     auswahl = Console.readChar();
     if (auswahl =='9') break;

     switch (auswahl)
     {
      case '1': anzahlPersonenermitteln(); break;//Aufruf
      case '2': raeumeErmitteln(); break;//Aufruf
      default: System.out.println("Fehlerhafte Eingabe: " +
               "Bitte nur 1, 2 oder 9 eingeben");
               continue;
     }
    }
    System.out.println("Ende des Programms");
 }//Ende Hauptprogramm*************************************

 public static void anzahlPersonenermitteln()
 {
     //Alle Räume durchlaufen
     for (int raum = 1; raum < raeume.length; raum ++)
     {
         //Alle Bestuhlungsarten durchlaufen
         String bestuhlung = "Fehler";
         int anzahlPersonen = 0;

         for (int art = 1; art <= 5; art ++)
         {
             switch (art)
             {
               case 1:
                 anzahlPersonen =
                   berechneTheaterbestuhlung
                     (raeume[raum][0], raeume[raum][1]);
                 bestuhlung = "Theaterbestuhlung";
                 break;
```

```
              case 2:
                anzahlPersonen =
                  berechneStuhlkreis
                    (raeume[raum][0], raeume[raum][1]);
                bestuhlung = "Stuhlkreis";
                break;
              case 3:
                anzahlPersonen =
                  berechneUForm
                    (raeume[raum][0], raeume[raum][1]);
                bestuhlung = "U-Form";
                break;
              case 4:
                anzahlPersonen =
                  berechneParlamentarisch
                    (raeume[raum][0], raeume[raum][1]);
                bestuhlung = "parlamentarischer Bestuhlung";
                break;
              case 5:
                anzahlPersonen =
                  berechneFischgraeten
                   (raeume[raum][0], raeume[raum][1]);
                bestuhlung = "Fischgräten-Bestuhlung";
                break;
              default: anzahlPersonen = -1;
            }

            if (anzahlPersonen != -1)
              System.out.println("Anzahl Personen: "
                + anzahlPersonen + " in Raum " + raum
                + " mit " + bestuhlung);
        }
    }
}

private static int personenProReihe
 (double raumBreite, double gangbreite)
{
    double platzbreite = 0.7;
    double freiraumBreite = raumBreite - 2 * gangbreite;
    int personenProReihe = 0;
    while(freiraumBreite > 0)
    {
        personenProReihe ++;
        freiraumBreite -= platzbreite;
        //Rundungsfehler vermeiden
        freiraumBreite = freiraumBreite * 10;
        freiraumBreite = Math.round(freiraumBreite * 10) / 10;
        freiraumBreite = freiraumBreite / 10;
        if(freiraumBreite > 0 && personenProReihe % 10 == 0)
        {
            freiraumBreite -= gangbreite;
        }
        if(freiraumBreite < 0)
        {
```

```
            personenProReihe --;
        }
    }
    return personenProReihe;
}

private static int anzahlReihen
  (double raumLaenge, double platzlaenge)
{
    double gangbreite = 1.5, rednerplatz = 2.0;
    double freiraumLaenge=raumLaenge-gangbreite-rednerplatz;
    int anzahlReihen = (int)(freiraumLaenge / platzlaenge);
    return anzahlReihen;
}

private static int berechneTheaterbestuhlung
  (double raumBreite, double raumLaenge)
{
   //Mindestgröße für einen Sitzplatz
   //Gangbreite links u. rechts je 1.5 m+Platzbreite 0.7=3.7
    if(raumBreite < 3.7) return 0;
    double gangbreite = 1.5;
    double platzlaenge = 1.5;
    return personenProReihe(raumBreite,gangbreite) *
       anzahlReihen(raumLaenge,platzlaenge);
}

private static int berechneStuhlkreis
 (double raumBreite, double raumLaenge)
{
   //Mindestgröße für einen Sitzplatz
    if(raumBreite < 3.7) return 0;
    double platzlaenge = 1.5;
    //Zusatzgang alle 10 Personen nicht notwendig,
    //daher 0.0 als Parameter
    int personenProReihe = personenProReihe(raumBreite,0.0);
    int anzahlReihen = anzahlReihen(raumLaenge,platzlaenge);
    return personenProReihe + 2 * anzahlReihen;
}

private static int berechneUForm
 (double raumBreite, double raumLaenge)
{
   //Mindestgröße für einen Sitzplatz
    if(raumBreite < 3.7) return 0;
      return berechneStuhlkreis(raumBreite,raumLaenge) - 4;
}

private static int berechneParlamentarisch
  (double raumBreite, double raumLaenge)
{
    //Mindestgröße für einen Sitzplatz
    if(raumBreite < 3.7) return 0;
    double gangbreite = 1.5;
    double platzlaenge = 1.5;
```

```
    return personenProReihe(raumBreite, gangbreite)*
    ((int)anzahlReihen(raumLaenge,platzlaenge)/2);
}

private static int berechneFischgraeten
 (double raumBreite, double raumLaenge)
{
   //Mindestgröße für einen Sitzplatz
   if(raumBreite < 3.7) return 0;
   double gangbreite = 1.5;
   double platzlaenge = 2.0;
   return personenProReihe(raumBreite, gangbreite)*
               ((int)anzahlReihen(raumLaenge,platzlaenge)/2);
}

public static void raeumeErmitteln()
{
    System.out.println
      ("Geben Sie bitte die Anzahl der Personen ein:");
    int personenanzahl = Console.readInt();
    System.out.println
     ("Geben Sie bitte die Bestuhlungsart durch "
      + "einen Buchstaben ein:");
    System.out.println
     ("Theater = t, Stuhlkreis = s, U-Form = u,"
      + "Parlamentarisch = p, Fischgräten = f:");
    char artChar = Console.readChar();
    switch (artChar)
    {
        case 't': case 's': case 'u': case'p': case'f': break;
        default: System.out.println("Eingabefehler");
    }
    //Alle Räume durchlaufen
    System.out.println("Folgende Räume sind geeignet:");
    for (int raum = 1; raum < raeume.length; raum ++)
    {
        String bestuhlung = "Fehler";
        int anzahlPersonen = 0;
        switch (artChar)
        {
          case 't':
            anzahlPersonen =
               berechneTheaterbestuhlung
                 (raeume[raum][0], raeume[raum][1]);
            bestuhlung = "Theaterbestuhlung"; break;
          case 's':
             anzahlPersonen =
               berechneStuhlkreis
                 (raeume[raum][0], raeume[raum][1]);
             bestuhlung = "Stuhlkreis"; break;
          case 'u':
             anzahlPersonen =
               berechneUForm
                 (raeume[raum][0], raeume[raum][1]);
             bestuhlung = "U-Form"; break;
```

```
            case 'p':
               anzahlPersonen =
                 berechneParlamentarisch
                   (raeume[raum][0], raeume[raum][1]);
               bestuhlung = "parlamentarischer Bestuhlung";
               break;
            case 'f':
               anzahlPersonen =
                 berechneFischgraeten
                 (raeume[raum][0], raeume[raum][1]);
               bestuhlung = "Fischgräten-Bestuhlung"; break;
          default: anzahlPersonen = -1;
         }

         if (anzahlPersonen != -1 &&
            anzahlPersonen >= personenanzahl)
         {
           System.out.println("Anzahl Personen: "
              + anzahlPersonen
              + " in Raum " + raum
              + " mit " + bestuhlung);
         }
      }
  }
}
```

Ein Beispiellauf ergibt folgende Ausgabe:

```
Bitte Funktion auswählen:
1: Personen pro Raum und Bestuhlungsart ermitteln
2: Räume für Anzahl Personen und Bestuhlungsart
Abbruch: 9
Bitte Ziffer 1, 2 oder 9 eingeben:
1
Anzahl Personen: 280 in Raum 1 mit Theaterbestuhlung
Anzahl Personen: 56 in Raum 1 mit Stuhlkreis
Anzahl Personen: 52 in Raum 1 mit U-Form
Anzahl Personen: 140 in Raum 1 mit parlamentarischer Bestuhlung
Anzahl Personen: 100 in Raum 1 mit Fischgräten-Bestuhlung
Anzahl Personen: 1184 in Raum 2 mit Theaterbestuhlung
Anzahl Personen: 116 in Raum 2 mit Stuhlkreis
Anzahl Personen: 112 in Raum 2 mit U-Form
Anzahl Personen: 576 in Raum 2 mit parlamentarischer Bestuhlung
Anzahl Personen: 448 in Raum 2 mit Fischgräten-Bestuhlung
Anzahl Personen: 315 in Raum 3 mit Theaterbestuhlung
Anzahl Personen: 63 in Raum 3 mit Stuhlkreis
Anzahl Personen: 59 in Raum 3 mit U-Form
Anzahl Personen: 150 in Raum 3 mit parlamentarischer Bestuhlung
Anzahl Personen: 105 in Raum 3 mit Fischgräten-Bestuhlung
Anzahl Personen: 40 in Raum 4 mit Theaterbestuhlung
Anzahl Personen: 22 in Raum 4 mit Stuhlkreis
Anzahl Personen: 18 in Raum 4 mit U-Form
Anzahl Personen: 20 in Raum 4 mit parlamentarischer Bestuhlung
Anzahl Personen: 10 in Raum 4 mit Fischgräten-Bestuhlung
Anzahl Personen: 30 in Raum 5 mit Theaterbestuhlung
Anzahl Personen: 22 in Raum 5 mit Stuhlkreis
```

```
Anzahl Personen: 18 in Raum 5 mit U-Form
Anzahl Personen: 15 in Raum 5 mit parlamentarischer Bestuhlung
Anzahl Personen: 10 in Raum 5 mit Fischgräten-Bestuhlung
Bitte Funktion auswählen:
1: Personen pro Raum und Bestuhlungsart ermitteln
2: Räume für Anzahl Personen und Bestuhlungsart
Abbruch: 9
Bitte Ziffer 1, 2 oder 9 eingeben:
2
Geben Sie bitte die Anzahl der Personen ein:
200
Geben Sie bitte die Bestuhlungsart durch einen Buchstaben ein:
Theater = t, Stuhlkreis = s, U-Form = u, Parlamentarisch = p,
Fischgräten = f:
t
Folgende Räume sind geeignet:
Anzahl Personen: 280 in Raum 1 mit Theaterbestuhlung
Anzahl Personen: 1184 in Raum 2 mit Theaterbestuhlung
Anzahl Personen: 315 in Raum 3 mit Theaterbestuhlung
Bitte Funktion auswählen:
1: Personen pro Raum und Bestuhlungsart ermitteln
2: Räume für Anzahl Personen und Bestuhlungsart
Abbruch: 9
```

Die gewählte Lösung kann wie folgt erweitert werden:

Erweiterbarkeit

Kommen zusätzliche Räume hinzu, dann muss nur das zweidimensionale Feld der Datenabstraktion entsprechend erweitert werden.

Kommen neue Bestuhlungsarten hinzu, dann müssen die jeweiligen `switch`-Anweisungen ergänzt sowie jeweils eine weitere Berechnungsfunktion hinzugefügt werden.

Neue Funktionen können ebenfalls leicht hinzugefügt werden. Es muss jeweils eine entsprechende Prozedur oder Funktion hinzugefügt sowie das Hauptprogramm um einen entsprechenden Aufruf ergänzt werden.

7 Das Wichtigste zum Testen *

Ziel jeder Programmentwicklung ist es, ein Programm zu erstellen, das die Anforderungen des Auftraggebers erfüllt und möglichst keine oder nur wenige Fehler enthält.

Testfälle aufstellen

Bevor Sie ein Programm entwickeln, sollten Sie aus den Programmanforderungen mögliche Eingabedaten und die zu erwartenden Ergebnisse in Form von **Testfällen** zusammenstellen. Ist das Programm oder sind Teile des Programms fertiggestellt, dann sollten Sie das Programm mit den vorher aufgestellten Testfällen durchführen:

- »Einfaches Testen«, S. 279

Regressionstest

Während einer Programmentwicklung entstehen durch Änderungen oder Erweiterungen immer neue Programmversionen, die jeweils neu mit Testfällen getestet werden müssen. Empfehlenswert ist es, die **Testdaten** in Dateien zu speichern und bei neuen Programmversionen die bisherigen Testdaten automatisch aus diesen Dateien zu lesen:

- »Regressionstest«, S. 281

Stapelverarbeitung

Um die Durchführung von Tests zu automatisieren, gibt es die Möglichkeit sogenannte Stapelverarbeitungsprogramme zu erstellen:

- »Stapelverarbeitungsprogramme: .bat-Dateien«, S. 287

Auswahl von Testdaten

Das Finden von Fehlern hängt wesentlich davon ab, welche Testdaten gewählt werden:

- »Zur Auswahl von Testdaten«, S. 292

7.1 Einfaches Testen *

Bevor man ein Programm entwickelt, sollte man eine klare Vorstellung davon haben, was das Programm tun soll. Auf dieser Grundlage wird ein Test erstellt, der prüft, ob das Programm die Anforderungen erfüllt. Während der Weiterentwicklung können sich Fehler in ein Programm einschleichen. Daher muss getestet werden, ob die neue Version die ursprünglichen Anforderungen weiterhin wie erwartet erfüllt.

Programme sollen bestimmten Anforderungen genügen, damit sie für vorgesehene Aufgaben eingesetzt werden können. Aus den Anforderungen ergeben sich mögliche Eingabewerte und die durch das Programm zu ermittelnden Ausgabewerte.

Beispiel 1a

Im Kapitel »OptiTravel: Zeitvergleich«, S. 145, wird das Programm `Vergleich` beschrieben. Drei einzugebene `int`-Werte sol-

len in aufsteigender Reihenfolge (min, mittel, max) sortiert ausgegeben werden. Um das Programm auszuführen, sind also immer drei Werte einzugeben, z. B. `wert1 = 3`, `wert2 = 1`, `wert3 = 2`.

Werden Programme mit Eingabewerten ausgeführt, um die Richtigkeit der Programmergebnisse zu überprüfen, dann nennt man diese Eingabewerte **Testdaten**. Im Beispiel 1a gibt es das Testdatum `wert1 = 3`, das Testdatum `wert2 = 1` und das Testdatum `wert3 = 2`. Oft benötigen Programme mehr als ein Testdatum zur Ausführung. Alle Testdaten, die benötigt werden, um einen Programmlauf von der Eingabe bis zum Ergebnis durchzuführen, und das erwartete Sollergebnis bezeichnet man als **Testfall**.

Beispiel 1b

Bevor ein Programm mit Testdaten ausgeführt wird, sollte sich der Tester vorher das zu erwartende Ergebnis überlegen. Für die Testdaten `wert1 = 3`, `wert2 = 1`, `wert3 = 2` muss folgendes Ergebnis ausgegeben werden: `Min: 1, Mittel: 2, Max: 3`.

Im Beispiel 1a und 1b bilden also die drei Testdaten sowie das Sollergebnis zusammen einen Testfall.

Tipp

Wird ein Programm verändert, dann muss das Programm erneut getestet werden. Die neue Programmversion sollte unter einem anderen Namen abgespeichert werden. Die alte Programmversion sollte *nicht* gelöscht werden. Die verschiedenen Programmversionen sollten durch aufsteigende Nummern am Ende des Namens gekennzeichnet werden, z. B. `Vergleich1`, `Vergleich2` usw.

7

Empfehlung

Eine inzwischen bewährte Vorgehensweise bei der Programmierung ist der **Test-First-Ansatz**. Zuerst werden ein oder mehrere Testfälle entworfen, bevor ein Teil des Programms entwickelt wird.

Frage

Führen Sie ein Programm mit einem Testfall aus und das Ergebnis ist richtig, dann haben Sie nur nachgewiesen, dass das Programm für genau diesen Testfall korrekt ist. Überlegen Sie sich weitere Testfälle für das Programm `Vergleich`, die es ermöglichen die Programmfunktionalität möglichst vollständig zu testen.

Antwort

Die Anzahl der möglichen Anordnungen – Permutationen genannt – von n verschiedenen Elementen ist n! (n-Fakultät, siehe auch »Rekursion«, S. 239). Bei 3 möglichen Anordnungen ergeben sich $3! = 1 \cdot 2 \cdot 3 = 6$ verschiedene Testfälle, z. B. 1–2-3, 1–3-2, 2–1-3, 2–3-1. 3–1-2 und 3–2-1.

Frage

Überlegen Sie, für welche Eingaben das Programm Vergleich besonders fehleranfällig sein könnte. Mit welchen Testfällen können Sie dies überprüfen?

Antwort

Bei einem Vergleichsprogramm könnten gleiche Eingabewerte, z. B. 3–3-3, oder die Werte um 0 herum, z. B. -1,0,+1, oder negative Werte, z. B. -3, -2, -1 oder negative Werte in Kombination mit positiven Werten, z. B. -3, 0, +2, besonders kritisch sein.

Prüfung von zwei Programmversionen

Am einfachsten kann man zwei Programmversionen überprüfen, indem man beide Programme mit den gleichen Eingaben aufruft und die Ausgabe vom Programm 1, z. B. `Vergleich1`, mit der Ausgabe vom Programm 2, z. B. `Vergleich2`, vergleicht. Der Vergleich zeigt dann, ob beide Programme bei gleichen Eingaben übereinstimmende Ausgaben produzieren.

Regressionstest

Der Vergleich der Ausgabe von zwei Programmversionen bei gleicher Eingabe wird **Regressionstest** genannt (siehe: »Regressionstest«, S. 281).

Sie benötigen für ein Zeiterfassungssystem ein Programm, das aus Stunden, Minuten und Sekunden die Gesamtzeit in Sekunden ermittelt. Überlegen Sie – bevor Sie das Programm entwerfen – einige Testfälle, d. h. schreiben Sie sich einige Eingabedaten auf und berechnen Sie mit dem Taschenrechner die Gesamtzeit. Prüfen Sie anschließend, ob Ihr Programm die richtigen Ergebnisse liefert.

7.2 Regressionstest *

Beim Regressionstest werden in Dateien gespeicherte Testfälle für den erneuten Test geänderter Programme genutzt. Die Testergebnisse werden ebenfalls in Textdateien gespeichert. Dadurch können sie automatisch auf Gleichheit überprüft werden.

Wiederholung von Tests nach Programmänderungen

Während einer Programmentwicklung werden Programme modifiziert und erweitert. Wurde die erste Programmversion mit Testfällen getestet, dann muss ein modifiziertes oder erweitertes Programm erneut getestet werden. Dieselbe Situation liegt auch vor, wenn aufgrund von Programmfehlern das Programm verbessert wird. Auch dann müssen die Tests wiederholt werden, um sicherzustellen, dass nach der Fehlerbeseitigung nicht neue Fehler durch die Korrektur entstanden sind.

Regressionstest

Wird ein Programm nach einer Änderung erneut mit den gleichen **Testfällen** ausgeführt, dann spricht man von einem **Regressionstest**.

Testfälle in Textdateien speichern

Um die Testfälle, die bei komplexen Programmen sehr umfangreich sein können, nicht alle »per Hand« neu eingeben zu müs-

sen, ist es sinnvoll die Testdaten in Textdateien zu speichern und die Eingabedaten – statt vom Konsolenfenster oder einer grafischen Benutzungsoberfläche aus – aus den Textdateien zu lesen.

Testergebnisse in Textdateien speichern

Die Testergebnisse des ursprünglichen Programms müssen mit den Testergebnissen des modifizierten Programms verglichen werden. Um diesen Vergleich automatisch vornehmen zu können, ist es sinnvoll, die Testergebnisse ebenfalls in eine Textdatei zu schreiben. Um zwei Dateien auf Gleichheit zu überprüfen, gibt es eine Reihe von Programmen, die dies tun.

FileIn FileOut

Damit Sie ein Gefühl dafür bekommen, wie ein solcher Regressionstest in Java ablaufen kann, wird das Paket `inout` um die Klassen `FileIn` und `FileOut` mit folgenden Methoden ergänzt:

- `public FileIn(String infilename)`: Öffnet die Text-Eingabedatei mit der Bezeichnung `infilename`. Diese Methode muss vor dem ersten lesenden Zugriff aufgerufen werden.
- `public String gibNaechsteZeile()`: Liest einen Text vom Typ `String` von einer Textdatei.
- `public void schliesseEingabedatei()`: Schließt die geöffnete Eingabedatei. Muss aufgerufen werden, wenn kein lesender Zugriff mehr erfolgt.
- `public FileOut(String outfilename)`: Öffnet die Ausgabedatei mit der Bezeichnung `filename`. Muss vor dem ersten schreibenden Zugriff aufgerufen werden.
- `public void schreibeNaechsteZeile(String zeile)`: Schreibt einen Text vom Typ `String` in eine Textdatei.
- `ppublic void schliesseAusgabedatei()`: Schließt die geöffnete Ausgabedatei. Aufruf muss nach dem letzten schreibenden Zu-griff erfolgen.
- Zeichenketten, die Zahlen repräsentieren, können durch sogenannte Hüllklassen in einfache Typen konvertiert werden:
  - `Integer.parseInt(string), Short.parseShort(string),`
  - `Byte.parseByte(string), Long.parseLonge(string)`
  - `Float.parseFloat(string), Double.parseDouble(string)`
- `int`-Werte können durch `Integer.toString(i)` in eine Zeichenkette gewandelt werden, analog `Float.toString(f)` usw.

Hinweis

Wenn Sie alle Programme bereits heruntergeladen haben, dann enthält der Ordner `inout` bereits diese zusätzlichen Klassen `FileIn` und `FileOut`. Öffnen Sie diese Dateien und sehen Sie sich die Methoden an.

Beispiel 1a

Sie wollen das Programm `VergleichTest` (siehe unten) mit folgenden Testfällen ausführen:

1–2-3, 1–3-2, 2–1-3, 2–3-1, 3–1-2, 3–2-1, 3–3-3, 3–3-2, 2–3-3.

Als Testergebnisse erwarten Sie:

1–2-3, 1–2-3, 1–2-3, 1–2-3, 1–2-3, 1–2-3, 3–3-3, 2–3-3, 2–3-3

Die Testfälle schreiben Sie zeilenweise – eine Zahl pro Zeile – in die Textdatei `Testfaelle.txt` und legen Sie in dasselbe Verzeichnis, in dem sich auch das Programm befindet. Das ursprüngliche Programm `Vergleich` (siehe »OptiTravel: Zeitvergleich«, S. 145) wandeln Sie in eine Methode um, die Sie vom Hauptprogramm aus ausrufen:

VergleichTest

```
// Vergleich von 3 Werten

import inout.*;

public class Vergleich
{

 public static void main (String args[])
 {
  FileIn infile = new FileIn("Testfaelle.txt");
  FileOut outfile = new FileOut("Testausgabe.txt");
  int wert1, wert2, wert3; //3 Vergleichswerte
  int[] werte = {0,0,0};

  for (int i=1; i<=9; i++)
  {
   werte[0] = Integer.parseInt(infile.gibNaechsteZeile());
   werte[1] = Integer.parseInt(infile.gibNaechsteZeile());
   werte[2] = Integer.parseInt(infile.gibNaechsteZeile());
   sortieren(werte);
   outfile.schreibeNaechsteZeile
                    ("Testfall " + i + " Min Mittel Max");
   outfile.schreibeNaechsteZeile(String.valueOf(werte[0]));
   outfile.schreibeNaechsteZeile(String.valueOf(werte[1]));
   outfile.schreibeNaechsteZeile(String.valueOf(werte[2]));

  }
   infile.schliesseEingabedatei();
   outfile.schliesseAusgabedatei();
 }

 public static void sortieren (int[] werte)
 {

  int wert1 = werte[0], wert2 = werte[1], wert3 = werte[2];

  if (wert1 <= wert2)
     if (wert2 <= wert3) //wert1 <= wert2 <= wert3
      {
        werte[0] = wert1; werte[1] = wert2; werte[2] = wert3;
      }
```

```
      else //(wert1 <= wert2  und wert3 <= wert2)
        if (wert1 <= wert3)
         {
           werte[0] = wert1; werte[1] = wert3; werte[2] = wert2;
         }
         else //wert3 < wert1
         {
           werte[0] = wert3; werte[1] = wert1; werte[2] = wert2;
         }
   else //wert2 < wert1
   {
      if (wert2 <= wert3)
        if (wert1 <= wert3)
         {
           werte[0] = wert2; werte[1] = wert1; werte[2] = wert3;
         }
         else //wert3 < wert1
         {
           werte[0] = wert2; werte[1] = wert3; werte[2] = wert1;
         }
      else //wert3 < wert2
      {
         werte[0] = wert3; werte[1] = wert2; werte[2] = wert1;
      }
   }
 }
}
```

Wenn Sie als Dateinamen für die Ausgabe `Testausgabe.txt` wählen, dann stehen die Ergebnisse anschließend wie folgt in dieser Textdatei (Ausschnitt):

```
Testfall 1 Min Mittel Max
1
2
3
Testfall 2 Min Mittel Max
1
2
3
```

Sie stellen fest, dass die Testergebnisse mit Ihren vorher ermittelten Ergebnissen übereinstimmen. Für alle diese Testfälle arbeitet das Programm also korrekt.

Überlegen Sie sich weitere Testfälle, ergänzen Sie die Datei `Testfaelle.txt` und führen Sie das Programm erneut aus.

Beispiel 1b

Sie stellen fest, dass es ein Programm `SortAuswahl` gibt (siehe »Einfaches Sortieren«, S. 195), das als Alternative zu dem Programm `Vergleich` eingesetzt werden kann. Sie ändern das Programm so, dass statt Zeichenketten ganze Zahlen sortiert werden können, und testen es mit denselben Testdaten aus der Datei `Testfaelle.txt`.

SortAuswahlTest

```
// Vergleich von 3 Werten
import inout.*;

public class SortAuswahl2
{
 public static void main (String args[])
 {
  FileIn infile = new FileIn("Testfaelle.txt");
  FileOut outfile = new FileOut("Testausgabe2.txt");

  int wert1, wert2, wert3; //3 Vergleichswerte
  int[] werte = {0,0,0};
  for (int i=1; i<=9; i++)
  {
   werte[0] = Integer.parseInt(infile.gibNaechsteZeile());
   werte[1] = Integer.parseInt(infile.gibNaechsteZeile());
   werte[2] = Integer.parseInt(infile.gibNaechsteZeile());
   sortieren(werte);
   outfile.schreibeNaechsteZeile
        ("Testfall " + i + " Min Mittel Max");
   outfile.schreibeNaechsteZeile(String.valueOf(werte[0]));
   outfile.schreibeNaechsteZeile(String.valueOf(werte[1]));
   outfile.schreibeNaechsteZeile(String.valueOf(werte[2]));
  }
   infile.schliesseEingabedatei();
   outfile.schliesseAusgabedatei();
 }

 public static void sortieren(int[] feld)
 {
  int min, merke;
  int pos, posMin;
  //Sortieren und Vertauschen
  for (int i = 0; i < feld.length; i++)
  {
    //Kleinste Position ab i suchen
    posMin = i; min = feld[i];
    for (pos = i + 1; pos < feld.length; pos++)
      if (feld[pos] < min) //Abfrage auf <
       {
          min = feld[pos];
          posMin = pos; //Kleinste Position merken
       }
     //Vertauschen
     merke = feld[i];
     feld[i] = feld[posMin];
     feld[posMin] = merke;
  }
 }
}
```

Die Testergebnisse werden diesmal in die Datei `Testausgabe2.txt` geschrieben.

Programm FC

Im Konsolenfenster können Sie mit dem Programm FC, das standardmäßig auf der Windows-Plattform zur Verfügung steht, zwei Dateien auf Gleichheit überprüfen:

- `fc filename1 filename2`:
  Befehl, der die Differenzen der beiden Dateien `filename1` und `filename2` ausgibt.

Hinweis

Auf Linux-Plattformen steht ein anderes Programm für den Vergleich zwischen zwei Dateien zur Verfügung. Der Befehl lautet dort `cmp filename1 filename2`. Einen Überblick über die Unterschiede zwischen Windows- und Linux-Plattformen finden Sie im Kapitel »Stapelverarbeitungsprogramme: .bat-Dateien«, S. 287.

Beispiel

Der Vergleich der beiden Dateien `Testausgabe.txt` und `Testausgabe2.txt` im Konsolenfenster führt zu folgendem Ergebnis:

```
C:\>fc c:\java\vergleichtest\testausgabe.txt
c:\java\sortauswahltest\testausgabe2.txt
Vergleichen der Dateien C:\JAVA\VERGLEICHTEST\Testausgabe.txt
und C:\JAVA\SORTAUSWAHL\Testausgabe2.txt
FC: Keine Unterschiede gefunden
```

Die Abb. 7.2-1 zeigt nochmals das Prinzip des Regressionstests.

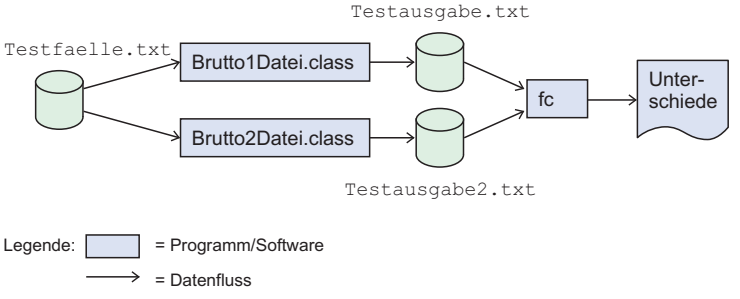


Abb. 7.2-1: Bei einem Regressionstest verwenden zwei Programmversionen dieselben Eingabedaten, erzeugen getrennte Ausgabedaten, die durch ein Programm – hier `fc` – auf Unterschiede hin überprüft werden.

Führen Sie diese Programme – wie oben beschrieben – auf Ihrem Computersystem aus. Ändern Sie das Programm `SortAuswahl2` so ab, dass ein fehlerhaftes Ergebnis entsteht, und führen Sie einen erneuten Regressionstest durch.

7.3 Stapelverarbeitungsprogramme: .bat-Dateien **

Befehlsfolgen, z. B. Start von Programmen, die oft in gleicher Reihenfolge ausgeführt werden müssen, können zu sogenannten Stapelverarbeitungsprogrammen *(batch processing)* zusammengefasst und in einer Datei abgespeichert werden (bei Windows sogenannte `.bat`-Dateien). Beim Aufruf der Datei, z. B. durch Eingabe des Dateinamens im Konsolenfenster, werden dann die in der Datei aufgeführten Befehle nacheinander ausgeführt. Dadurch erspart man sich das wiederholte Eintippen derselben Befehlsfolgen in das Konsolenfenster. Durch die Vergabe von Parametern können variable Befehlsfolgen »programmiert« werden.

Windows-Plattformen

Auf Windows-Plattformen ist es im Konsolenfenster möglich, Befehle hinzuschreiben und ausführen zu lassen, z. B. Start des Java-Programms `SortAuswahl2` durch `java SortAuswahl2`.

Java

Hinter dem Befehl `java` kann nicht nur der Name des Java-Programms angegeben werden, das ausgeführt werden soll, sondern es können weitere Angaben gemacht werden, die an das Java-Programm übergeben werden (siehe »Felder als Eingabeparameter«, S. 219).

Beispiel

An das Java-Programm `SortAuswahl3` sollen die Dateinamen `Testfaelle.txt` und `Testausgabe2.txt` übergeben werden. Aus der Datei `Testfaelle.txt` sollen die Testdaten gelesen und die Ergebnisse sollen in die Datei `Testausgabe2.txt` geschrieben werden. Der Befehl im Konsolenfenster sieht dann folgendermaßen aus:

```
java SortAuswahl3 Testfaelle.txt Testausgabe2.txt
```

Zwischen den einzelnen Angaben muss jeweils ein Leerzeichen stehen. Was geschieht nun mit diesen Angaben im Java-Programm? Jede Java-Anwendung besitzt eine `main`-Methode, die wie folgt aussieht:

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

In Klammern steht `String args[]`, das bedeutet, dass Angaben, die im Befehl `java` nach dem Programmnamen angegeben sind, als Eingabeparameter an die `main`-Methode übergeben werden. Den Wert des ersten Parameters, hier `"Testfaelle.txt"`, kann man einer Variablen vom Typ `String` zuweisen, z. B. durch `String Eingabedatei = args[0]`. Den Wert des zweiten Parameters kann man ebenfalls einer String-Variablen zuweisen, z. B. durch `String Ausgabedatei = args[1]`. Ändert man nun das Java-Programm so ab, dass diese Dateiangaben an die entsprechen-

den Ein- und Ausgabemethoden weitergegeben werden, dann kann das Programm mit wechselnden Dateien ausgeführt werden, d. h. die Dateinamen müssen nicht fest in das Programm »einprogrammiert« oder nach dem Start eingegeben werden:

SortAuswahl3

```
import inout.Console;
import inout.*;
public class SortAuswahl3
{

 public static void main (String args[])
 {
  int wert1, wert2, wert3; //3 Vergleichswerte
  int[] werte = {0,0,0};
  //Übernahme des Namens der Eingabedatei von args[0]
  String eingabedatei = args[0];
  FileIn infile = new FileIn(eingabedatei);
  //Übernahme des Namens der Ausgabedatei von args[1]
  String ausgabedatei = args[1];
  FileOut outfile = new FileOut(ausgabedatei);

  for (int i=1; i<=9; i++)
  {
   werte[0] = Integer.parseInt(infile.gibNaechsteZeile());
   werte[1] = Integer.parseInt(infile.gibNaechsteZeile());
   werte[2] = Integer.parseInt(infile.gibNaechsteZeile());

   sortieren(werte);
   outfile.schreibeNaechsteZeile
          ("Testfall " + i + " Min Mittel Max");
   outfile.schreibeNaechsteZeile(String.valueOf(werte[0]));
   outfile.schreibeNaechsteZeile(String.valueOf(werte[1]));
   outfile.schreibeNaechsteZeile(String.valueOf(werte[2]));
  }
  infile.schliesseEingabedatei();
  outfile.schliesseAusgabedatei();
  }
 public static void sortieren(int[] feld)
 {
  int min, merke;
  int pos, posMin;
  //Sortieren und Vertauschen
  for (int i = 0; i < feld.length; i++)
  {
    //Kleinste Position ab i suchen
    posMin = i; min = feld[i];
    for (pos = i + 1; pos < feld.length; pos++)
      if (feld[pos] < min) //Abfrage auf <
       {
          min = feld[pos];
          posMin = pos; //Kleinste Position merken
       }
     //Vertauschen
     merke = feld[i];
     feld[i] = feld[posMin];
```

```
      feld[posMin] = merke;
    }
  }
}
```

Und so sieht eine Ausführung im Konsolenfenster aus:

```
C:\Java\SortAuswahl3>
java SortAuswahl3 Testfaelle.txt Testausgabe2.txt
```

Wird das Programm mit BlueJ ausgeführt, dann öffnet sich ein Fenster mit `SortAuswahl3.main({})`. Zwischen die geschweiften Klammern muss Folgendes eingegeben werden: `"Testfaelle.txt", "Testausgabe2.txt"`, d. h. die Dateinamen müssen in Anführungszeichen (es sind Zeichenketten) gesetzt und durch Komma getrennt eingegeben werden.

Führen Sie das Programm `SortAuswahl3` auf Ihrem Computersystem aus.

In vielen Fällen sind mehrere Programme direkt hintereinander auszuführen, ohne dass der Benutzer eingreifen soll oder muss.

Beispiel

Beim Regressionstest werden zwei Programmversionen hintereinander ausgeführt. Beide Programme lesen aus derselben Testdatendatei und geben ihre Ergebnisse in unterschiedliche Dateien aus. Anschließend werden beide Ausgabedateien auf Gleichheit geprüft. Folgende Befehle sind dazu notwendig:

```
java SortAuswahl3 Testfaelle.txt Testausgabe3.txt
java SortAuswahl4 Testfaelle.txt Testausgabe4.txt
fc Testausgabe3.txt Testausgabe4.txt > Vergleich.txt
type Vergleich.txt
```

Der vorletzte Befehl bewirkt, dass die beiden Ausgabedateien durch das Programm `fc` verglichen werden. Das Vergleichsergebnis wird nicht im Konsolenfenster angezeigt, sondern in die Datei `Vergleich.txt` »umgelenkt«. Der Befehl dazu ist das Größer-Zeichen `>`. Die Ausgabe wird in die Datei umgelenkt *(redirect)*, die nach dem Größer-Zeichen angegeben ist. Der letzte Befehl `type` zeigt den Inhalt der Datei `Vergleich.txt` an.

Müssen Vorgänge oft mit kleinen Variationen wiederholt werden, dann ist es mühsam, die erforderlichen Befehle immer wieder neu hinzuschreiben.

Beispiel

Beim Regressionstest müssen neue Programmversionen mit vorhandenen Testdateien ausgeführt werden. Das erneute Eingeben von drei Befehlen ist mühsam und fehleranfällig.

.bat-Dateien

Auf **Windows-Plattformen** ist es möglich, Befehlsfolgen, die im Konsolenfenster ausgeführt werden sollen, in sogenannten `.bat`-Dateien – auch Batch-Dateien genannt – abzuspeichern. Bei die-

sen Dateien handelt es sich um normale Textdateien, die jedoch die Dateiendung .bat besitzen. Wird im Konsolenfenster dann der Name der .bat-Datei (mit oder ohne Dateiendung) angegeben, dann wird die Befehlsfolge, die in dieser Datei steht, abgearbeitet. Die Ausführung solcher abgespeicherten Befehlsfolgen bezeichnet man als **Stapelverarbeitung** *(batch processing)*.

Shell-Skripte

Auf **Linux-Plattformen** werden Shell-Skripte benutzt, um eine Abfolge von Befehlen auszuführen. Sie haben die gleiche Funktion wie Batch-Dateien unter Windows. Sie besitzen die Dateiendung .sh und werden ausgeführt, indem der folgende Befehl in eine bash eingegeben wird: sh Skriptname.sh.

Beispiel für Windows

Die oben bereits aufgeführte Befehlsfolge wird in einer Datei testsort.bat gespeichert:

```
java SortAuswahl3 Testfaelle.txt Testausgabe3.txt
java SortAuswahl4 Testfaelle.txt Testausgabe4.txt
fc Testausgabe3.txt Testausgabe4.txt > Vergleich.txt
type Vergleich.txt
```

Im Konsolenfenster muss dann nur noch testsort eingetippt werden und alle Befehle werden nacheinander ausgeführt:

```
C:\Java\SortAuswahl3>testsort

C:\Java\SortAuswahl3>
   java SortAuswahl3 Testfaelle.txt Testausgabe3.txt

C:\Java\SortAuswahl3>
   java SortAuswahl4 Testfaelle.txt Testausgabe4.txt

C:\Java\SortAuswahl3>
   fc Testausgabe3.txt Testausgabe4.txt  1>Vergleich.txt

C:\Java\SortAuswahl3>type Vergleich.txt
Vergleichen der Dateien Testausgabe3.txt und TESTAUSGABE4.TXT
FC: Keine Unterschiede gefunden
```

Das Shell-Skript unter Linux sieht ganz ähnlich aus. Lediglich zwei Befehle lauten dort anders.

Beispiel für Linux

```
java SortAuswahl3 Testfaelle.txt Testausgabe3.txt
java SortAuswahl4 Testfaelle.txt Testausgabe4.txt
cmp Testausgabe3.txt Testausgabe4.txt > Vergleich.txt
cat Vergleich.txt
```

Führen Sie die Programme mit unterschiedlichen Testfällen aus.

echo off / echo

Wird eine .bat-Datei ausgeführt, dann werden alle in ihr enthaltenen Befehle im Konsolenfenster angezeigt. Mit dem Befehl echo off können Sie die Befehlsausgabe im Konsolenfenster verhindern. Soll nach einem Befehl echo off dennoch etwas im Konsolenfenster ausgegeben werden, dann kann der Befehl echo –

gefolgt von dem anzuzeigenden Text – verwendet werden. Soll ein einzelner Befehl *nicht* angezeigt werden, dann steht vor ihm ein Klammeraffe (@). Soll beispielsweise der Befehl `echo off` selbst nicht angezeigt werden, dann muss `@echo off` geschrieben werden.

`rem`

Längere Stapelverarbeitungs-Programme sollten durch Kommentare beschrieben werden. Eine Kommentarzeile beginnt mit `rem`.

`pause`

Der Befehl `pause` bewirkt, dass die Abarbeitung der `.bat`-Datei angehalten wird und erst nach der Betätigung einer beliebigen Taste weiterläuft.

`more`

Soll der Inhalt einer Textdatei seitenweise angezeigt werden, dann kann dazu der Befehl `more` verwendet werden.

Parameter

Eine ganz wichtige Eigenschaft von Stapelverarbeitungs-Programmen ist die Möglichkeit, Daten, die sich von Ausführung zu Ausführung ändern, als sogenannte Parameter anzugeben.

Beispiel

Ist eine neue Programmversion `SortAuswahl5` entstanden, die mit dem Programm `SortAuswahl4` verglichen werden soll, dann muss in dem bisherigen Stapelverarbeitungsprogramm `testsort.bat` der Programmname `SortAuswahl3` gegen `SortAuswahl4` und der Programmname `SortAuswahl4` gegen `SortAuswahl5` ausgetauscht werden. Bei jeder neuen Programmversion sind solche Änderungen vorzunehmen.

`%1, %2 usw.`

Durch die formalen Parameter `%1`, `%2` usw. ist es möglich, sich ändernde Werte als Parameter in dem Stapelverarbeitungs-Programm anzugeben. Beim Start der Batch-Datei werden dann hinter dem Dateinamen die aktuellen Werte für die Parameter angegeben.

Beispiel für Windows

Anstelle von `SortAuswahl3` wird in dem Stapelverarbeitungs-Programm `%1` und anstelle von `SortAuswahl4` wird `%2` angegeben:

```
@echo off
rem Regressionstest
echo Regressionstest von %1 und %2
java %1 Testfaelle.txt Testausgabe3.txt
java %2 Testfaelle.txt Testausgabe4.txt
fc Testausgabe3.txt Testausgabe4.txt > Vergleich.txt
more Vergleich.txt
```

Im Konsolenfenster sieht dies folgendermaßen aus:

```
C:\Java\SortAuswahl3>testsort1 SortAuswahl3 SortAuswahl4
Regressionstest von SortAuswahl3 und SortAuswahl4
Vergleichen der Dateien Testausgabe3.txt und TESTAUSGABE4.TXT
FC: Keine Unterschiede gefunden
```

Tab. 7.3-1 zeigt die wichtigsten Befehle für Windows-Plattformen und Linux-Plattformen auf einen Blick.

Windows	Linux	Bedeutung
`rem`	`#`	Kommentar
`echo`	`echo`	Textausgabe
`type`	`cat`	Ausgabe von Dateien
`more`	`more`	Seitenorientierte Dateiausgabe
`%1`	`$1`	Parameter
`pause`	`sleep`	Unterbrechung
`cls`	`clear`	Inhalt des Konsolenfensters löschen
`fc`	`cmp`	Dateivergleich

Tab. 7.3-1: Windows- und Linux-Befehle auf einen Blick.

7.4 Zur Auswahl von Testdaten **

Um Programmierfehler zu finden, sollte ein Programm mit unterschiedlichen Eingabewerten getestet werden. Die Eingabedaten sollten repräsentativ für den erlaubten Wertebereich sein. Zusätzlich können Extremwerte gewählt werden, die am Rande erlaubter Wertebereiche liegen. Besonders sensibel reagieren Programme oft auf spezielle Eingabewerte, z. B. `0` oder `1`.

Unterschiedliche Testdaten

In Abhängigkeit von Eingabewerten können sich Programme unterschiedlich verhalten. Daher sollten Sie jedes Programm mit *unterschiedlichen* Datensätzen testen.

Die Auswahl der Eingabedaten sollten Sie systematisch vornehmen. Eine Möglichkeit besteht darin, **repräsentative Eingabedaten** zu verwenden.

Außerdem ist es möglich, **Extremwerte** als Eingabedaten auszuwählen. Oft treten Fehler auch bei **speziellen Werten** auf, z. B. bei dem Eingabewert `0`.

Bei der Eingabe von Zeichenketten sind Sonderzeichen oder Steuerzeichen besonders zu überprüfen. Besonders wirksam ist die Kombination dieser drei Möglichkeiten.

Beispiel 1a

Sie wollen in die USA verreisen. Sie stellen bei Ihren Web-Recherchen fest, dass in den USA alle Temperaturangaben in Fahrenheit sind.

Sie entschliessen sich, ein Umrechnungsprogramm `Fahrenheit in Celsius` und `Celsius in Fahrenheit` zu schreiben. Sie finden im Web eine entsprechende Umrechnungstabelle (Tab. 7.4-1).

° F	° C	° F	° C
0	-17,8	70	21,1
1	-17,2	80	26,7
5	-15,0	90	32,2
10	-12,2	100	37,8
20	-6,7	110	43,3
30	-1,1	120	48,9
32	0,0	130	54,4
40	4,4	140	60,0
50	10,0	150	65,6
60	15,6	212	100,0

Tab. 7.4-1: Umrechnungstabelle Fahrenheit – Celsius.

Beim Recherchieren im Web erfahren Sie noch, dass die Fahrenheit-Temperaturskala nach dem Physiker und Instrumentenbauer Daniel Gabriel Fahrenheit benannt wurde, der am 24. Mai 1686 in Danzig geboren wurde. Fahrenheit wählte als Nullpunkt seiner Temperaturskala die tiefste Temperatur des strengen Winters 1708/1709 in seiner Heimatstadt Danzig. Mit einer Mischung aus Eis, Salmiak und Wasser (Kältemischung) konnte er danach den Nullpunkt wieder herstellen (-17,8 °C). Fahrenheit wollte dadurch negative Temperaturen vermeiden, wie sie bei der Römer-Skala schon im normalen Alltagsgebrauch auftraten. Seine eigene Körpertemperatur legte er bei 100 Grad fest. Seine ursprüngliche Skala sah nur zwölf Unterteilungen vor, später teilte er diese jedoch noch einmal durch acht gleiche Gradabstände, wodurch er letztlich auf 96 Grade kam. Fahrenheit bemerkte, dass der Gefrierpunkt von reinem Wasser (Eispunkt) bei 32 °F und der Siedepunkt bei 212 °F liegen.

Als **repräsentative Testdaten** wählen Sie die Fahrenheit-Werte 50 und 80 Grad aus, als **Extremwerte** die Fahrenheit-Temperaturen 0 und 212 sowie als **speziellen Wert** 32.

Damit Sie Ihr Programm automatisch testen können, legen Sie die Testdaten jeweils in eine Textdatei. Außerdem erstellen Sie folgende Batch-Datei `fahrenheit.bat`:

```
@echo off
rem Test Fahrenheit
echo Test des Programms %1
java %1 testfaelle.txt testausgabe.txt
```

Sie finden im Web die Formel

```
Celsius = (Fahrenheit - 32) * 5 / 9
```

zur Umrechnung und erstellen damit folgendes Programm:

Fahrenheit1

```
// Berechnung der Celsius-Temperatur
// aus einer Fahrenheit-Temperatur.
// Die Fahrenheit-Testfälle wird aus einer Datei
// eingelesen und die Ergebnisse in eine Datei ausgegeben.

import inout.*;
public class Fahrenheit1
{
 public static void main (String args[])
 {
   double fahrenheit, celsius;
   //Übernahme des Namens der Eingabedatei von args[0]
   String eingabedatei = args[0];
   FileIn infile = new FileIn(eingabedatei);
   //Übernahme des Namens der Ausgabedatei von args[1]
   String ausgabedatei = args[1];
   FileOut outfile = new FileOut(ausgabedatei);
   for (int i=1; i<=5; i++)
   {
    celsius = berechneCelsius
      (Double.parseDouble(infile.gibNaechsteZeile()));
     outfile.schreibeNaechsteZeile(String.valueOf(celsius));
   }
   infile.schliesseEingabedatei();
   outfile.schliesseAusgabedatei();
 }

 private static double berechneCelsius(double temperatur)
 {
   double celsius = (temperatur - 32.0) * 5.0 / 9.0;
   //Testausgabe
   System.out.println
        ("Fahrenheit: " + temperatur + " Celsius: " + celsius);
   return celsius;
 }
}
```

Die Ausführung der Batch-Datei führt zu folgenden Ergebnissen, die mit den Werten in der Fahrenheit-Celsius-Tabelle übereinstimmen:

```
C:\Java\Fahrenheit>fahrenheit Fahrenheit1
Test des Programms Fahrenheit1
Fahrenheit: 50.0 Celsius: 10.0
Fahrenheit: 80.0 Celsius: 26.666666666666668
Fahrenheit: 0.0 Celsius: -17.77777777777778
Fahrenheit: 212.0 Celsius: 100.0
Fahrenheit: 32.0 Celsius: 0.0
```

Bei der Analyse Ihres Programms stellen Sie fest, dass es sinnvoll ist, die Berechnung zu optimieren. Anstelle von

```
celsius = (fahrenheit - 32.0) * 5.0 / 9.0;
```

berechnen Sie einmal `5/9 = 0.5555555555` und schreiben nun folgende Anweisung in Ihr Programm `Fahrenheit2`:

```
celsius = (fahrenheit - 32.0) * 0.5555555555;
```

Um einen Regressionstest durchzuführen, ändern Sie die Batch-Datei wie folgt:

```
@echo off
rem Regressionstest Fahrenheit
cls
if exist vergleich?.txt del vergleich?.txt
if exist testausgabe?.txt del testausgabe?.txt
rem ***************************************
echo Regressionstest von %1 und %2 echo Testdatensatz
java %1 testfaelle.txt testausgabe1.txt
java %2 testfaelle.txt testausgabe2.txt
fc testausgabe1.txt testausgabe2.txt > vergleich.txt
echo Datei vergleich
more vergleich.txt
pause rem ***************************************
echo Testende
```

.bat-Befehle

Mit dem Befehl `cls` wird das Konsolenfenster gelöscht (`cls` steht für *clear screen*).

Der Befehl `if exist vergleich?.txt del vergleich?.txt` besteht aus einer Abfrage und einer Aktion. Die Abfrage prüft, ob eine Datei, die zu dem Muster `vergleich?.txt` passt, existiert. Wenn ja, wird die nachstehende Aktion ausgeführt.

Der Befehl `del datei?.txt` löscht alle Dateien, die zum angegebenen Muster passen. Damit werden die Ergebnisse früherer Testläufe gelöscht.

Beispiel 1b

Sie führen nun einen Regressionstest durch:

```
Regressionstest von Fahrenheit1 und Fahrenheit2
Testdatensatz
Fahrenheit: 50.0 Celsius: 10.0
Fahrenheit: 80.0 Celsius: 26.666666666666668
Fahrenheit: 0.0 Celsius: -17.77777777777778
Fahrenheit: 212.0 Celsius: 100.0
Fahrenheit: 32.0 Celsius: 0.0
Fahrenheit: 50.0 Celsius: 9.999999999
Fahrenheit: 80.0 Celsius: 26.666666664
Fahrenheit: 0.0 Celsius: -17.777777776
Fahrenheit: 212.0 Celsius: 99.99999999
Fahrenheit: 32.0 Celsius: 0.0
Datei vergleich
Vergleichen der Dateien testausgabe1.txt und TESTAUSGABE2.TXT
***** testausgabe1.txt
10.0
26.666666666666668
-17.77777777777778
100.0
```

```
0.0
***** TESTAUSGABE2.TXT
9.999999999
26.666666664
-17.777777776
99.99999999
0.0
*****

Drücken Sie eine beliebige Taste . . .
Testende
```

Sie stellen zu Ihrem Erstaunen fest, dass die Genauigkeit von 10 Stellen hinter dem Komma nicht ausreicht. Daraufhin entschließen Sie sich, die erste Programmversion beizubehalten.

1. Sammeln Sie Ihre eigenen Erfahrungen mit der Auswahl von Testdaten und dem Regressionstest. Modifizieren Sie das obige Programm so, dass nach folgender Formel aus Celsius-Werten Fahrenheit-Werte berechnet werden:
   `Fahrenheit = [Grad Celsius] * 9 / 5 + 32`
2. Für ein Zeiterfassungssystem sollen die eingegebenen Stunden, Minuten und Sekunden in eine Gesamtzahl Sekunden umgerechnet werden. Überlegen Sie sich einige Testfälle mit geeigneten Testdaten, die die oben aufgeführten Kriterien erfüllen.

8 Die Grundideen der Verifikation ***

Die Verifikation beruht im Wesentlichen auf Arbeiten von [Floy67] und [Hoar69]

Beim Testen wird für *einige* möglichst gut ausgewählte Testfälle das Programm ausgeführt und beobachtet, ob das gewünschte Ergebnis ermittelt wird. Da in der Regel aus Aufwandsgründen nicht alle möglichen Kombinationen von Eingabedaten getestet werden können, liefert das Testen *keine* Gewissheit über die Korrektheit des Programms für die noch nicht getesteten Eingabedaten. Um diesen prinzipiellen Nachteil des Testens zu vermeiden, wurden Methoden entwickelt, um durch theoretische Analysen die **Korrektheit** eines Programms zu zeigen.

verifizieren= bewahrheiten, auf Stichhaltigkeit prüfen

Verifikation ist eine formal exakte Methode, um die Konsistenz zwischen der Programmspezifikation und der Programmimplementierung für *alle* in Frage kommenden Eingabedaten zu *beweisen*. Durch Verifizieren erreicht man daher eine wesentlich größere Sicherheit bezüglich der Fehlerfreiheit von Programmen als durch Testen. Schon bei der Programmentwicklung müssen alle Korrektheitsargumente gesammelt werden, um später die Korrektheit des fertigen Programms garantieren zu können.

Anhand eines Beispiels wird zunächst eine intuitive Einführung in die Verifikation gegeben:

- »Intuitive Einführung«, S. 297

Anschließend werden die einzelnen Bestandteile einer Verifikation detailliert betrachtet:

- »Zusicherungen«, S. 301
- »Spezifizieren mit Anfangs- und Endebedingung«, S. 303
- »Verifikationsregeln«, S. 306
- »Termination von Schleifen«, S. 313
- »Entwickeln von Schleifen«, S. 315
- »Vor- und Nachteile«, S. 320

Die Ausführungen in diesem Kapitel geben nur eine elementare Einführung in die Verifikation und sollen im Wesentlichen die Grundideen vermitteln. Sie orientieren sich an [Futs89, passim].

Literatur

Ausführlich wird die Verifikation z. B. in den Büchern [ApOl94], [Babe90], [Fran92] und [Futs89] behandelt.

8.1 Intuitive Einführung ***

Eine Möglichkeit, Programme zu überprüfen, ist die Verifikation. Die Verifikation ist systematisch, allgemein, abstrakt, deduktiv und induktiv orientiert. Sie liefert den abstrakten Beweis, dass die gesamte Problemlösung korrekt ist.

Die Idee der Verifikation wird zunächst an einem Beispiel demonstriert, bevor die einzelnen Konzepte genauer betrachtet werden.

Beispiel 1a

Es soll ein Programm geschrieben werden, das aus einer beliebigen reellen Zahl A und einer positiven ganzen Zahl B die Potenz A^B ermittelt. Ein Programm, das dieses Problem löst, ist dann korrekt, wenn die Beziehungen der Abb. 8.1-1 gelten.

Dieses Programm nutzt folgende mathematischen Beziehungen aus:

$Basis^{Exponent} = (Basis * Basis)^{Exponent / 2}$ für geraden Exponenten und

$Basis^{Exponent} = Basis * Basis^{Exponent - 1}$ für ungeraden Exponenten

Dasselbe Aktivitätsdiagramm ist in Abb. 8.1-3 nochmals angegeben, jedoch versehen mit Zusicherungen. Diese Kommentare beschreiben den Zustand des Programms an den einzelnen Stellen.

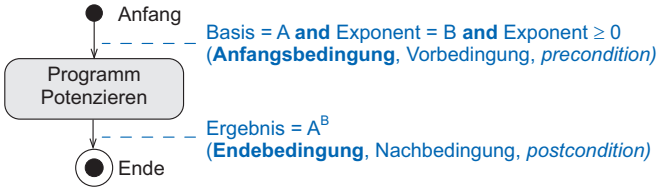


Abb. 8.1-1: Spezifikation des Programms `Potenzieren`.

Alle zulässigen Eingabewerte für die Basis werden durch A, alle zulässigen Eingabewerte für den Exponenten werden durch B dargestellt – im Gegensatz zu testenden Verfahren werden keine konkreten Werte wie `Basis = 5` und `Exponent = 4` angenommen. Die angegebenen Beziehungen werden als **Zusicherungen** *(assertions)* bezeichnet. Es wird nun ein Programm `Potenzieren` in Form eines Aktivitätsdiagramms angegeben, von dem gezeigt werden soll, dass es das Problem löst (Abb. 8.1-2).

Die Anfangs- und Endebedingung kann sofort hingeschrieben werden. Nun sind alle Anweisungen und alle Pfade des Programmablaufplans durchzugehen und zu zeigen, dass man aus der Anfangsbedingung durch das Programm die Endebedingung erhält. Die Schleife möge man sich vorläufig aufgeschnitten, d. h. ungeschlossen denken. Die an der Stelle **2** angegebene Zusicherung ist der Angelpunkt des ganzen Verfahrens.

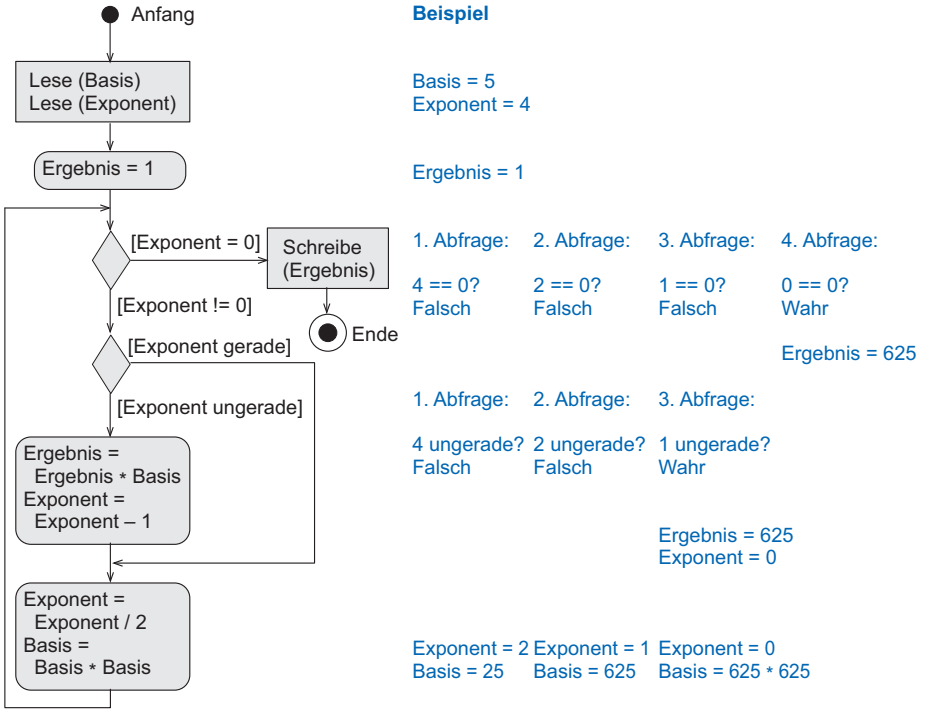


Abb. 8.1-2: Programm `Potenzieren`.

Sie muss zunächst intuitiv gefunden und in geeigneter Weise formuliert werden. Ausgehend von dieser Behauptung **2** können nun weitere Zusicherungen abgeleitet werden. Die Ausgangsbedingung **3** ergibt sich aus **2**, da `Exponent = 0` ist und $Basis^0 = 1$ gilt. **4** folgt unmittelbar aus **2**. Um **7** aus **5** abzuleiten, wird **5** in die äquivalente Form **6** umgeschrieben. Durch Ersetzen von Ergebnis · Basis durch Ergebnis und Exponent – 1 durch Exponent ergibt sich dann **7**. Interessant ist, dass **7** und **8** an der Zusammenführungsstelle übereinstimmen. **7** bzw. **8** kann in **9** umgeformt werden ($(x \cdot x)^{y/2} = x2 \cdot y/2 = xy$). Durch Ersetzen von Exponent / 2 durch Exponent und Basis · Basis durch Basis in **9** erhält man **10**. Schließt man nun die Rückwärtsschleife von **10** nach **2**, so sieht man, dass beide Zusicherungen übereinstimmen, d. h. die als Hypothese angenommene Zusicherung **2** bleibt bestehen. Würden die Zusicherungen **2** und **10** nicht identisch sein oder würde sich keine Beziehung zwischen beiden Zusicherungen herstellen lassen, dann wären die postulierten Zusicherungen falsch oder wenigstens untauglich.

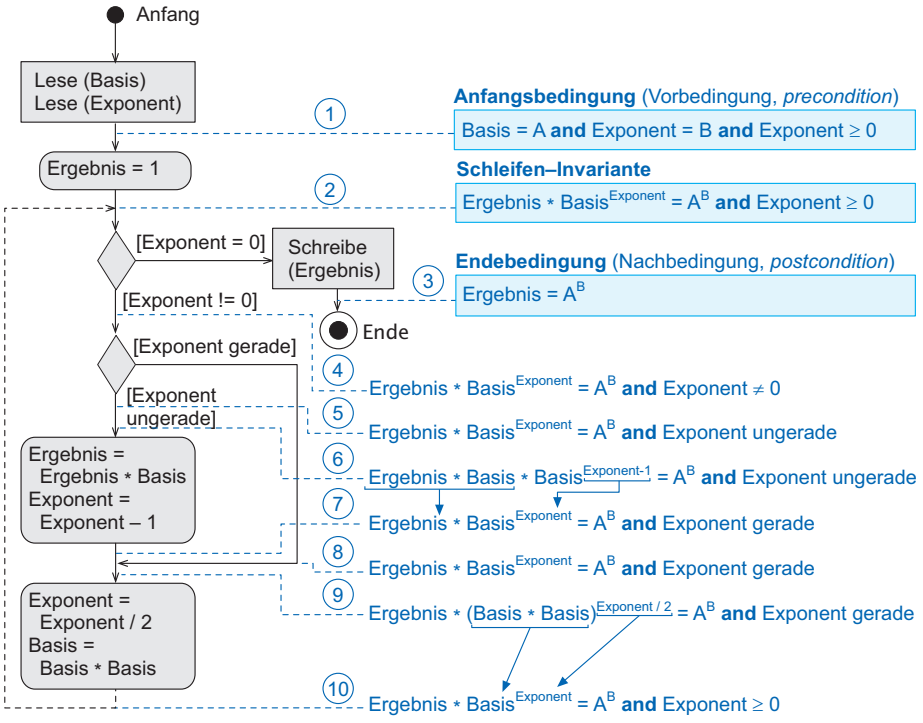


Abb. 8.1-3: Programm `Potenzieren` mit Zusicherungen.

Die Zusicherung **2** gilt also offenbar für den ersten Durchlauf der Schleife, da mit Ergebnis = 1, Basis = A und Exponent = B der Ausdruck Ergebnis · $\text{Basis}^{\text{Exponent}} = A^B$ wahr ist. Aufgrund des oben angegebenen Schlusses gilt die Zusicherung daher für den zweiten Durchlauf der Schleife sowie bei allen anderen Durchläufen. Eine solche Zusicherung an der Schnittstelle einer Schleife wird Schleifen-**Invariante** genannt, da diese Beziehung auch nach der wiederholten Ausführung der Wiederholungsanweisungen unverändert, d. h. invariant bleibt.

Damit ist bewiesen, daß dieses Programm das Ergebnis A^B liefert, wenn es das Ende erreicht. Dass das Programm nach endlich vielen Wiederholungen endet, d. h. terminiert, wurde nicht bewiesen. Dies muss gesondert gezeigt werden.

Totale Korrektheit

Der totale Korrektheitsbeweis eines Algorithmus besteht also aus zwei Teilen:

a Beweis, dass das korrekte Ergebnis bei Termination geliefert wird.

b Beweis der Termination.

8.2 Zusicherungen ***

Bei der Programm-Verifikation wird die Aufmerksamkeit auf allgemeine Eigenschaften von Zwischenzuständen von Berechnungen und die Relationen zwischen ihnen gelenkt.

Zusicherungen *(assertions)* garantieren an bestimmten Stellen im Programm bestimmte Eigenschaften oder Zustände. Sie sind logische Aussagen über die Werte der Programmvariablen an den Stellen im Programm, an denen die jeweiligen Zusicherungen stehen.

Beispiel 1

Im Programm `Potenzieren` (siehe »Intuitive Einführung«, S. 297) gilt an der Stelle **2** beispielsweise immer die Zusicherung:
Ergebnis · Basis^{Exponent} = A^{B} **and** Exponent >= 0

Beispiel 2

Nach dem Quadrieren einer reellen Zahl kann zugesichert werden, dass diese Zahl nicht negativ ist (Abb. 8.2-1).

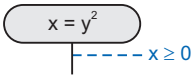


Abb. 8.2-1: Zusicherung beim Quadrieren.

Zur Formulierung

Es gibt mehrere Möglichkeiten, eine Zusicherung zu formulieren:

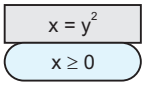
- Umgangssprachlich, z. B. x ist nicht negativ, oder
- formal, z. B. $x \geq 0$.

Die formale Notation von Zusicherungen besteht aus boole'schen Ausdrücken mit Konstanten und Variablen mit Vergleichsoperatoren ($<$, $\leq$, $=$, $\neq$, $\geq$, $>$) und logischen Operatoren (**and**, **or**, **not**, $\Leftrightarrow$, $\Rightarrow$).

Zur Notation

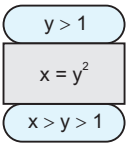
Drei Notationen lassen sich unterscheiden:

- Annotation durch gestrichelte Linien an einem Aktivitätsdiagramm (siehe Beispiel 2),
- Ergänzung von Struktogrammen durch Rechtecke mit abgerundeten Ecken [Futs89] und
- spezielle Kommentare oder Makros in Programmiersprachen, z. B. `assert (x >= 0); //Zusicherung ist ungültig, wenn x negativ ist.`



Beispiel 3

Ist vor der Zuweisung $x = y^2$ sichergestellt, dass y größer als 1 ist, dann kann man nach der Zuweisung zusichern, dass x größer als y ist. Nachher gilt auch $y > 1$.



Zusicherungen können das Verstehen der Wirkung von Programmen erleichtern.

Beispiel 4

In dem Programm `Vertausche` (Abb. 8.2-2, a) werden die Werte der Variablen a, b und c durch Vertauschungen so umgeordnet, dass am Schluss $a \leq b \leq c$ gilt.

In diesem Programm ist schwierig zu erkennen, ob in allen Zweigen des Programms das richtige Ergebnis erzielt wird. Mit Hilfe von eingefügten Zusicherungen ist die Wirkung besser zu verstehen (Abb. 8.2-2, b). Die einzelnen Zusicherungen gelten an den jeweiligen Stellen im Programm, unabhängig von den Anfangswerten der Variablen a, b und c.

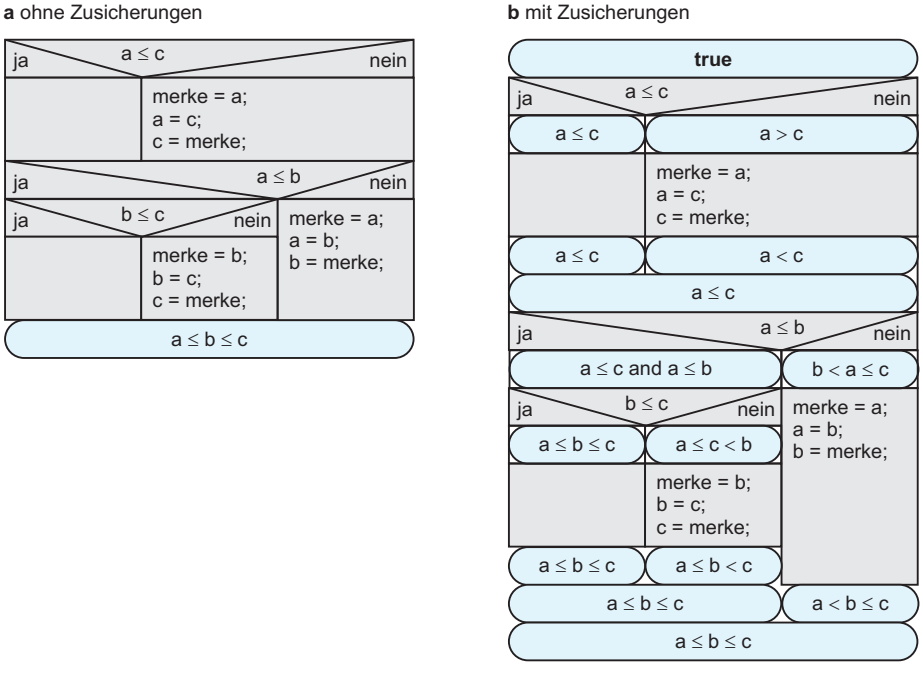


Abb. 8.2-2: Das Programm `Vertausche` ohne und mit Zusicherungen.

Während der Programmentwicklung sollte man Zusicherungen zunächst umgangssprachlich formulieren und dann erst formal hinschreiben.

Empfehlung

Generell sollte man sich angewöhnen, in jedem `else`-Zweig einer Auswahlanweisung die gültige Bedingung hinzuschreiben, da man sich oft nicht darüber im Klaren ist, wie die Negation der Bedingung aussieht.

Die Zusicherung `true` ist immer erfüllt. Sie schränkt den Wertebereich der Variablen in keinerlei Weise ein. Sie wird als erste Zusicherung (Anfangsbedingung) in einem Programm verwendet, wenn für die Werte der Variablen keinerlei Einschränkungen existieren.

Zusicherungen werden in Java durch eine `assert`-Anweisung spezifiziert (siehe »Zusicherungen in Java«, S. 162).

8.3 Spezifizieren mit Anfangs- und Endebedingung ***

Mit der formalen Methode der Verifikation kann man die Korrektheit eines Programms beweisen. Voraussetzung für die Verifikation ist, dass die Wirkung des Programms durch eine Spezifikation in Form einer Vorbedingung und einer Nachbedingung beschrieben ist.

Die Wirkung eines Programms kann durch die Zusicherungen Anfangsbedingung und Endebedingung spezifiziert werden.

precondition

Die **Anfangsbedingung** (Vorbedingung, *precondition*) gilt vor dem spezifizierten Programm und legt die zulässigen Werte der Variablen vor dem Ablauf des Programms fest.

postcondition

Die **Endebedingung** (Nachbedingung, *postcondition*) gilt nach dem spezifizierten Programm und legt die gewünschten Werte der Variablen und Beziehungen zwischen den Variablen nach dem Ablauf des Programms fest (Abb. 8.3-1).

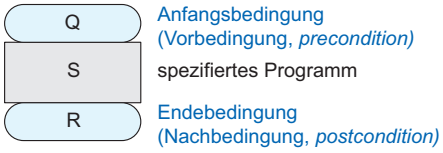


Abb. 8.3-1: Anfangs- und Endebedingung bei einem Programm.

Notation

In einem linearen Programmtext (siehe »Strukturierte Programmierung«, S. 153) setzt man die Anfangs- und Endebedingungen in geschweifte Klammern: {Q} S {R}. Betrachtet man eine Spezifikation, ohne sich auf ein konkretes Programm S zu beziehen, dann schreibt man: {Q}. {R}.

Spezifikation als Vorgabe für die Implementierung

Ist ein Programm durch eine Anfangs- und eine Endebedingung spezifiziert, dann ist es die Aufgabe des Programmierers, ein Programm S zu schreiben. Jedesmal, wenn vor dem Programm die Anfangsbedingung Q erfüllt ist, muss das Programm terminieren und nach der Termination die Endebedingung R erfüllen.

Beispiel 5

Es soll ein Programm Tausche geschrieben werden, das die Werte der zwei Variablen x und y vertauscht. Eine Spezifikation dieses Programms zeigt das Struktogramm in der Marginalspalte.

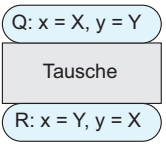


Abb. 8.3-2: Anfangs- und Endebedingung bei dem Programm Tausche.

Für alle Werte von x und y gilt: »Jedesmal, wenn vor dem Aufruf von Tausche x den Wert X und y den Wert Y hat, dann terminiert Tausche, und danach hat x den Wert Y und y den Wert X.«

X und Y stehen stellvertretend für beliebige Eingabewerte. Beim dynamischen Test wären X und Y konkrete Werte eines Testfalls. Da bei der Verifikation das Programm für alle Werte überprüft wird, werden sogenannte **externe Variable** verwendet, die alle Eingabewerte repräsentieren. Die Bezeichnung »externe Variable« wird verwendet, da die Variablen X und Y *keine* Programmvariablen sind, d. h. sie tauchen in der Implementierung nicht auf.

Externe Variable

Mithilfe der externen Variablen kann man einen Zusammenhang zwischen den Werten der Variablen vor dem Programm und den Werten nach dem Programm herstellen, da die Werte der externen Variablen durch das Programm nicht verändert werden können. Externe Variable werden im Folgenden immer in Großbuchstaben geschrieben.

8

Mehrere Spezifikationsmöglichkeiten

In der Regel gibt es mehrere Möglichkeiten, ein Programm zu spezifizieren.

Beispiel 6

Es soll eine Variable x quadriert werden. Die zwei Spezifikationen der Abb. 8.3-3 sind gleichwertig, da sie die gleiche Klasse von Programmen spezifizieren.

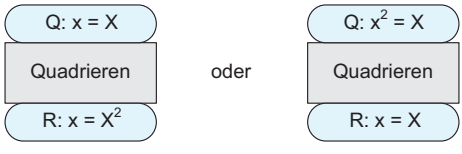


Abb. 8.3-3: Anfangs- und Endebedingung bei dem Programm Quadrieren.

Feste Variable

Variable, deren Werte im Programm nicht verändert werden sollen, werden als **feste Variablen** bezeichnet, d. h. es handelt sich um Konstanten. Sie werden analog wie externe Variable mit Großbuchstaben geschrieben.

Beispiel 7a

Das Programm `Maximum` soll den größeren Wert der Variablen `x` und `y` in der Variablen `m` ausgeben.

Bei der Spezifikation der Abb. 8.3-4,a, darf das Programm `Maximum` die Variablen `x` und `y` verändern. Sollen `x` und `y` unverändert bleiben, dann muss die Bedingung `x = X, y = Y` auch nach dem Programm gelten.

Da die Angabe der festen Variablen durch invariante Bedingungen der Form `x = X` und `y = Y` umständlich ist, werden in der Spezifikation der Abb. 8.3-4, b, feste Variablen verwendet. Auf diese Weise werden externe Variable eingespart.

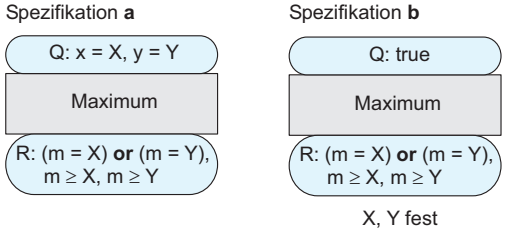


Abb. 8.3-4: Anfangs- und Endebedingung bei dem Programm `Maximum`.

Ist eine Spezifikation so formuliert, dass es kein Programm gibt, das die Spezifikation erfüllt, dann ist sie widersprüchlich.

Beispiel 8

Die Spezifikation der Abb. 8.3-5 ist widersprüchlich, da die Variable `x` nicht gleichzeitig unverändert sein kann und vorher und hinterher verschiedene Werte annehmen kann.

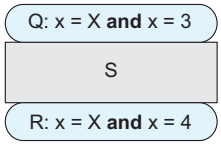


Abb. 8.3-5: Beispiel für eine widersprüchliche Spezifikation.

Unberechenbare Probleme

Schwieriger sind Widersprüche bei den sogenannten unberechenbaren Problemen zu erkennen. Diese Probleme sind so schwierig, dass es keine Programme gibt, die diese Probleme lösen. Folgende Probleme sind *nicht* berechenbar. Sie können daher auch *nicht* mit einem Programm allgemein gelöst werden:

- Feststellen, ob zwei Programme die gleiche Wirkung haben.
- Feststellen, ob ein beliebiges gegebenes Programm überhaupt terminiert.
- Feststellen, ob ein Programm eine Spezifikation erfüllt.
- Ein Programm zu einer Spezifikation generieren.
- Feststellen, ob zwei Spezifikationen die gleiche Klasse von Programmen festlegen.

Diese Aufgaben lassen sich nur in Einzelfällen für bestimmte Programme und Spezifikationen lösen, sind aber nicht für beliebige Programme und Spezifikationen algorithmisch lösbar.

Die obigen Beispiele zeigen, dass

- es unterschiedliche Spezifikationen gibt, die die gleiche Klasse von Programmen spezifizieren,
- es widersprüchliche Spezifikationen gibt, die kein Programm spezifizieren,
- Spezifikationen sorgfältig erstellt werden müssen, um genau die beabsichtigte Wirkung zu definieren und nicht mehr und nicht weniger.

8.4 Verifikationsregeln ****

Der Korrektheitsbeweis bei der Verifikation erfolgt dadurch, dass man zeigt, dass sich die Vorbedingung durch die Anweisungen des Programms in die Nachbedingung transformieren lässt. Dazu ist es erforderlich, dass die Semantik jeder Programmkonstruktion der verwendeten Programmiersprache formal beschrieben ist. Verifikationsregeln geben dann an, wie die Anfangsbedingung durch eine Programmkonstruktion, z. B. eine Zuweisung, eine Sequenz, eine Auswahl, eine Wiederholung, in eine Endebedingung gewandelt wird.

Programme setzen sich aus linearen Programmstrukturen zusammen (siehe »Strukturierte Programmierung«, S. 153).

Die Korrektheit eines Programms ergibt sich aufgrund der Korrektheit der Teilstrukturen. Dadurch kann ein komplexes Programm schrittweise durch korrektes Zusammensetzen aus einfacheren Strukturen verifiziert werden.

Es werden folgende **Verifikationsregeln** unterschieden:

Regeln

- Konsequenz-Regel,
- Zuweisungs-Axiom,
- Sequenz-Regel,
- `if`-Regel und
- `while`-Regel.

Diese Regeln können auch als axiomatisches Regelsystem zur Definition der Semantik der einzelnen Anweisungen interpretiert werden (axiomatische Semantik). Im Folgenden werden die Regeln einzeln behandelt.

Konsequenz-Regel

Die Konsequenz-Regel lautet:

- Ist {Q'} S {R'} gegeben, dann kann jederzeit die Vorbedingung Q' durch eine »schärfere« Vorbedingung Q und die Nachbedingung R' durch eine »schwächere« Nachbedingung R ersetzt werden, sodass weiterhin {Q} S {R} gilt.

Beispiel 1

Gegeben sei ein Programm S, das die Spezifikation a (Abb. 8.4-1) erfüllt. Es stellt sich die Frage, ob S auch die Spezifikation b (Abb. 8.4-1) erfüllt.

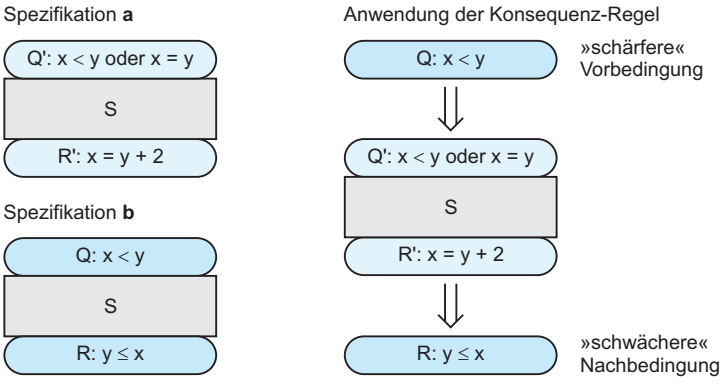


Abb. 8.4-1: Anwendung der Konsequenz-Regel.

Die Antwort lautet ja, denn es gelten folgende Implikationen:

$x < y \Rightarrow x < y$ oder $x = y$ und $x = y + 2 \Rightarrow y \leq x$.

Implikation ⇒

Die Implikation ⇒ spielt bei vielen Verifikationsregeln eine wichtige Rolle. Die in der Tab. 8.4-1 aufgeführten Formulierungen für Implikationen sind gleichwertig.

Arbeitet man sich vorwärts durch ein Programm, dann darf man Bedingungen schwächen. Durch Hinzufügen eines beliebigen Terms mit *oder*-Verknüpfung oder durch Weglassen eines vorhandenen, und-verknüpften Terms schwächt man eine Bedingung.

A ⇒ B	B wird von A impliziert
aus A folgt B	A ist hinreichend für B
B folgt aus A	B ist notwendig für A
wenn A gilt, dann gilt auch B	A ist schärfer als B
A impliziert B	B ist schwächer als A

Tab. 8.4-1: Gleichwertige Formulierungen für die Implikation.

Beispiel

Vorwärts durch das Programm der Abb. 8.4-1 (rechte Seite): Die schärfere Vorbedingung x < y wird abgeschwächt durch Hinzufügen des Terms x = y mit *oder*-Verknüpfung.

Arbeitet man sich rückwärts durch ein Programm, dann darf man Bedingungen verschärfen. Eine Bedingung kann man dadurch verschärfen, dass man einen beliebigen Term durch *und*-Verknüpfung hinzufügt oder dass man einen vorhandenen *oder*-verknüpften Term weglässt.

Beispiel

Rückwärts durch das Programm der Abb. 8.4-1 (rechte Seite): Die schwächere Vorbedingung x < y oder x = y wird verschärft durch Weglassen des Terms x = y mit *oder*-Verknüpfung.

Notation für Regeln

Verifikationsregeln werden oft in Form einer Schlussregel geschrieben:

```
Voraussetzungen
---------
Schlussfolgerung
```

Der Strich hat folgende Bedeutung: Aus der Gültigkeit der Bedingungen (Voraussetzungen) über dem Strich folgt die Gültigkeit der Bedingung (Schlussfolgerung) unter dem Strich.

Konsequenz-Regel als Schlussregel

Die Konsequenz-Regel kann auch in Form einer Schlussregel beschrieben werden:

```
Q => Q', {Q'} S {R'}, R' => R
-----------------
{Q} S {R}
```

Die Konsequenz-Regel liest sich dann folgendermaßen: Wenn die drei Bedingungen Q ⇒ Q', {Q' } S {R'} und R' ⇒ R erfüllt sind, dann gilt auch {Q} S {R}.

Zuweisungs-Axiom

Die Zuweisung x = A verändert den Wert der Variablen x. Gilt eine Zusicherung Q(A) vor der Zuweisung x = A, dann gilt danach R(x).

Beispiel 2a

Lautet die Vorbedingung Q(y + z = 10) und die Zuweisung x = y + z, dann ergibt sich die Nachbedingung R(x = 10).

Da die Zuweisung atomar in der Programmstruktur ist, wird ihre Semantik durch ein Axiom definiert. Das Zuweisungsaxiom lautet:

{RAx} x := A {R} wobei RAx bedeutet, dass alle x in R durch den Ausdruck A ersetzt sind.

Beispiel 2b

Der Ausdruck in der Zuweisung x := y + z ist y + z. Wird in der Nachbedingung x = 10 das x durch den Ausdruck y + z ersetzt, dann ergibt sich daraus die Vorbedingung y + z = 10.

Das Axiom gibt damit auch an, wie aus einer gegebenen Nachbedingung R eine passende Vorbedingung ermittelt werden kann:

Ableitung einer Vor- aus einer Nachbedingung

Es müssen alle Vorkommen der Variablen x in R durch den Ausdruck A ersetzt werden.

Beispiele

a {?} x := x + 25 {x = 2y}

Die Vorbedingung ergibt sich dadurch, dass in der Nachbedingung x = 2y alle x durch den Ausdruck x + 25 ersetzt werden: x + 25 = 2y. Es ergibt sich die Vorbedingung {2y = x +25}.

b {?} Ergebnis := Ergebnis · Basis {Ergebnis · $Basis^{Exponent} = A^B$ **and** Exponent gerade}

Das Einsetzen des Ausdrucks Ergebnis · Basis in die Nachbedingung ergibt folgende Vorbedingung:

{Ergebnis · Basis · $Basis^{Exponent} = A^B$ **and** Exponent gerade} oder vereinfacht:

{Ergebnis · $Basis^{Exponent + 1} = A^B$ **and** Exponent gerade}

Sequenz-Regel

Zwei Programmstücke S_1 und S_2 können zu einem Programmstück S_1; S_2 zusammengesetzt werden, wenn die Nachbedingung von S_1 mit der Vorbedingung von S_2 identisch ist (Abb. 8.4-2):

```
{Q} S1 {P}, {P} S2 {R}
--------------
{Q} S1 ; S2 {R}
```

Mithilfe der Konsequenz-Regel kann die Sequenz-Regel verallgemeinert werden (Abb. 8.4-3).

Die Nachbedingung von S_1 muss nicht mit der Vorbedingung von S_2 identisch sein. Es genügt, wenn die Nachbedingung von S_1 »schärfer« als die Vorbedingung von S_2 ist:

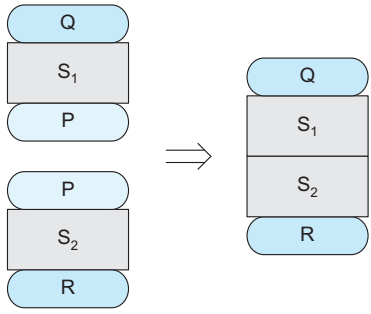


Abb. 8.4-2: Die Sequenz-Regel.

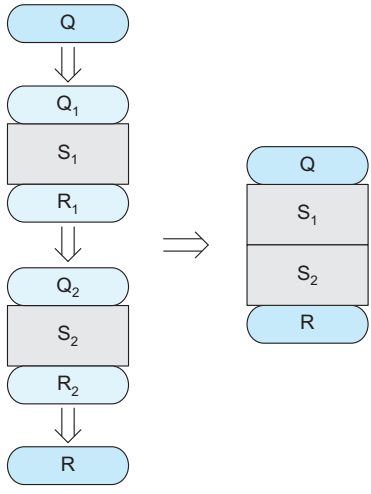


Abb. 8.4-3: Die Sequenz-Regel 1.

Sequenz-Regel 1

```
Q => Q1, {Q1} S1 {R1}, R1 => Q2, {Q2} S2 {R2}, R2 => R
------------------------------
{Q} S1;S2 {R}
```

Werden mehrere Programmstücke zusammengesetzt, dann wird die Sequenz-Regel mehrmals angewandt.

`if`-Regel

Die `if`-Regel (Abb. 8.4-4) gibt an, unter welchen Voraussetzungen zwei Programmstücke S_1 und S_2 und eine Bedingung B zu einer zweiseitigen Auswahl mit der Vorbedingung Q und der Nachbedingung R zusammengesetzt werden können:

```
{Q and B} S1 {R}, {Q and not B} S2 {R}
------------------------------
{Q} if B then S1 else S2 {R}
```

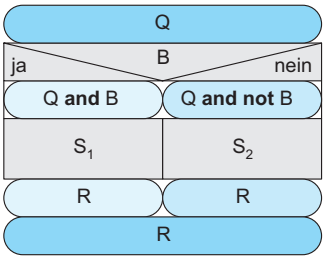


Abb. 8.4-4: Die `if`-Regel.

Gelten {Q **and** B} S_1 {R} und {Q **and not** B} S_2 {R}, dann können die Programme S_1 und S_2 zu einer `if`-Anweisung {Q} **if** B **then** S_1 **else** S_2 {R} zusammengesetzt werden.

Beispiel

Ein Programm S soll das Maximum der beiden festen Zahlen X und Y berechnen. Die Spezifikation lautet:

{Q : **true**} S {R : (m = X) or (m = Y), m≥X, m≥Y}

Das Maximum ist X oder Y. Das Maximum ist X, wenn $X \geq Y$ gilt, und Y, wenn **not** $X \geq Y$ gilt. Es gelten also die Vor- und Nachbedingungen der Abb. 8.4-5.

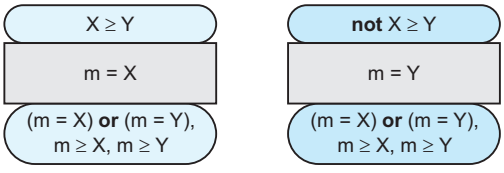


Abb. 8.4-5: Vor- und Nachbedingungen beim Programm `Maximum`.

Wählt man als Bedingung B: $X \geq Y$ und als Vorbedingung Q: **true**, dann sind die Voraussetzungen der `if`-Regel erfüllt und die beiden Anweisungen können zu einer `if`-Anweisung zusammengesetzt werden (Abb. 8.4-6).

`while`-Regel

Bei einer abweisenden Wiederholung (Abb. 8.4-7) wird der Rumpf der Wiederholungsanweisung solange wiederholt, bis die Wiederholungsbedingung B nicht mehr erfüllt ist:

while B **do** S

Invariante

Für die Verifikation jeder Wiederholungsanweisung oder Schleife spielt eine invariante Zusicherung P, die sogenannte Invariante, eine entscheidende Rolle. Die **Invariante** gilt nach jedem Schleifendurchlauf und beschreibt dadurch das im dynamischen

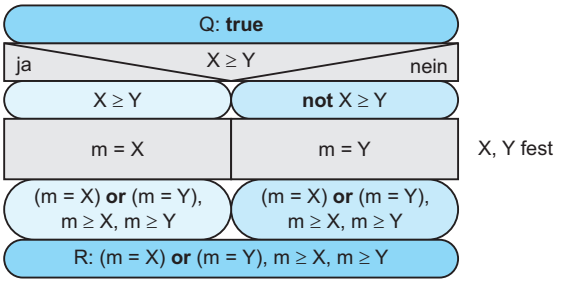


Abb. 8.4-6: Beispiel für die `if`-Regel.

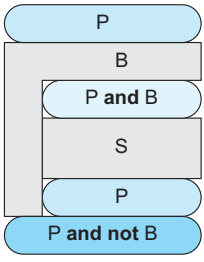


Abb. 8.4-7: Die `while`-Regel.

Ablauf gleichbleibende. Die Invariante P muss jedesmal erfüllt sein, wenn die Wiederholungsbedingung B ausgewertet wird. Damit die Invariante P bei jedem Auswerten von B erfüllt ist, muss sie vor der Schleife und nach dem Schleifenrumpf gelten. Es ergibt sich folgende **while**-Regel:

```
{P and B} S {P}
--------------
{P} while B do S {P and not B}
```

Diese Regel berücksichtigt nur die **partielle Korrektheit** der **while**-Schleife, da die Termination durch die Voraussetzungen dieser Regel nicht garantiert ist. Zur Feststellung der totalen Korrektheit einer Schleife muss also noch die Termination der Schleife zusätzlich bewiesen werden (siehe »Termination von Schleifen«, S. 313).

Beispiel

Das in der Abb. 8.4-8 dargestellte Programm berechnet die Fakultät eines eingegebenen Werts N. Die positive ganzzahlige Variable k wird bei jeder Wiederholung um 1 erniedrigt, sodass nach endlich vielen Schritten die Wiederholungsbedingung $k \neq 0$ nicht mehr erfüllt ist. Damit ist die Korrektheit von Fakultät bewiesen. Zu beachten ist, dass die Anweisungen fak := fak * k und k := k – 1 nicht vertauscht werden, da dann der Algorithmus nicht mehr korrekt ist.

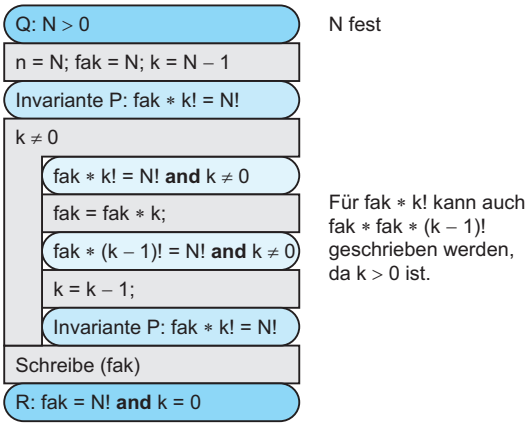


Abb. 8.4-8: Beispiel für die `while`-Regel.

8.5 Termination von Schleifen ****

Die totale Korrektheit erfordert zusätzlich noch den Nachweis der Termination einer Schleife. Zur Überprüfung der Termination führt man eine streng monoton fallende Terminationsfunktion ein.

Damit eine Schleife terminiert, darf die Wiederholungsbedingung B nach einer endlichen Anzahl von Schleifendurchläufen nicht mehr erfüllt sein.

Terminationsfunktion

Zur Prüfung der Termination führt man eine **Terminationsfunktion** t ein, die die Programmzustände auf ganze Zahlen abbildet. Der ganzzahlige Wert der Terminationsfunktion t muss bei jedem Schleifendurchlauf

1. um mindestens 1 kleiner werden
2. stets positiv bleiben.

Existiert eine solche Terminationsfunktion, dann muss die Schleife zwangsläufig nach endlicher Anzahl von Durchläufen terminieren. Da sich der Wert der Terminationsfunktion ändert, wird sie auch **Variante** im Gegensatz zur Invariante genannt.

Beispiel

Es soll der ganzzahlige Quotient q = (X / Y) und der Rest r = (X mod Y) zweier ganzer Zahlen X und Y unter ausschließlicher Verwendung von Addition und Subtraktion berechnet werden. In dem Programmskelett der Abb. 8.5-1 fehlt nur noch der Schleifenrumpf. Dieses Programm ist aufgrund der `while`-Regel für alle Schleifenrümpfe S mit der Vorbedingung P **and** $r \geq Y$ und der Nachbedingung P partiell korrekt.

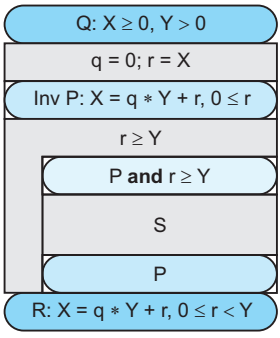
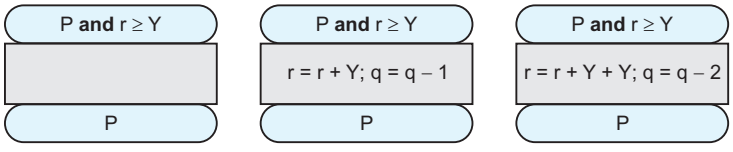


Abb. 8.5-1: Beispiel für die Termination von Schleifen.

Es gibt sehr viele verschiedene Programmstücke, die diese Bedingungen erfüllen und für den Schleifenrumpf S eingesetzt werden können (Abb. 8.5-2).



P: X = q * Y + r, 0 ≤ r

Abb. 8.5-2: Alternativen für den Schleifenrumpf.

Keiner dieser Schleifenrümpfe führt zur Termination des Gesamtprogramms, da keines der Programme einen Schritt näher zur Termination (Abbruchbedingung r < Y erfüllt) macht. Vor dem Schleifenrumpf gilt r ≥ Y und in endlicher Anzahl von Schleifendurchläufen soll r < Y erreicht werden, daher muss der Wert von r im Schleifenrumpf kleiner werden.

Die Aufgabe des Schleifenrumpfes ist also das »Verkleinern von r unter Invarianz von P« (Abb. 8.5-3).

Wird r um Y verkleinert, muss q um 1 erhöht werden, damit P: X = q*Y + r **and** 0 ≤ r invariant bleibt. Der Schleifenrumpf der Abb. 8.5-3 führt nach endlicher Anzahl von Schritten zu einem Wert r < Y und damit zur Termination des Programms.

Vorbedingung

Der Wert von r wird in jedem Schritt um Y (laut Vorbedingung gilt Y > 0) kleiner, bleibt aber stets positiv (laut Invariante: 0 ≤ r). Daher muss das Programm terminieren.

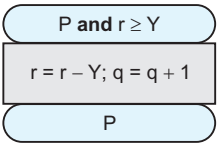


Abb. 8.5-3: Schleifenrumpf, der zur Termination führt.

Die beiden Bedingungen, die die Terminationsfunktion erfüllen muss, können formal *exakt* formuliert werden.

1 {P **and** B **and** t = T} S {t < T}
2 P **and** B ⇒ t ≥ 0

Die erste Bedingung verwendet eine externe Variable T, um auszudrücken, dass t im Schleifenrumpf S kleiner wird.

Die zweite Bedingung fordert, dass t vor jedem Ausführen des Schleifenrumpfes (P **and** B ist ja vor dem Schleifenrumpf erfüllt) nichtnegativ ist. Die zweite Bedingung zeigt auch, dass die Invariante P bzw. die Wiederholungsbedingung B so gewählt werden muss, dass aus P **and** B die Bedingung t ≥ 0 folgt.

Im Beispiel der Ganzzahldivision mit Terminationsfunktion t:r ist r ≥ 0 bereits Teil der Invarianten P: X = q · Y + r **and** 0 ≤ r.

In der Abb. 8.5-4 sind nochmals die Punkte zusammengestellt, die bei der Verifikation einer abweisenden Schleife erfüllt sein müssen.

8.6 Entwickeln von Schleifen ****

Es gibt Standardmethoden, um eine Invariante aus der Nachbedingung und/oder der Vorbedingung abzuleiten. Umgekehrt kann man aus einer Invariante einen Schleifenrumpf entwickeln.

Invariante und Terminationsfunktion sind die beiden Schlüsselkonzepte zur Verifikation von Schleifen.

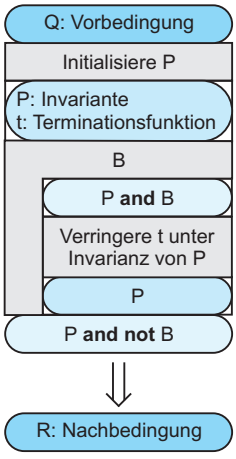
Invariante unbekannt

Ist die Invariante nicht bekannt, dann wird sie in den meisten Fällen aus der Nachbedingung der Spezifikation abgeleitet. Die Invariante muss eine Verallgemeinerung (Abschwächung) der Nachbedingung sein, damit sie nicht nur am Ende der Schleife, sondern auch bei allen Zwischenschritten und insbesondere auch am Anfang der Schleife in den Anfangszuständen nach einer geeigneten Initialisierung gilt (Abb. 8.6-1).

Zur Abschwächung der Nachbedingung R gibt es folgende Methoden:

- Weglassen einer Bedingung:
  R hat die Gestalt »A **and** B«. Die Invariante erhält man durch Weglassen einer der beiden Bedingungen A oder B. Wird zum

Bei gegebener Invariante P und Terminationsfunktion t muss eine **while**–Schleife die folgenden fünf Punkte erfüllen.



1 Die Invariante P gilt vor der Schleife.
Meist wird die Gültigkeit von P durch ein einfaches Programmstück zum Initialisieren von P erreicht:
{Q} Initialisiere P {P}
Gibt es keine Initialisierung, muss P direkt aus der Vorbedingung Q folgen (Konsequenz-Regel):
$Q \Rightarrow P$

2 Nach der Schleife gilt die Nachbedingung R.
P **and not** B $\Rightarrow$ R

3 P bleibt im Schleifenrumpf S invariant.
{P **and** B} S {P}

4 t wird bei jedem Ausführen des Schleifenrumpfes verringert.
{P **and** B **and** t = T} S {t < T}

5 t ist vor jedem Ausführen des Schleifenrumpfes nicht negativ.
P **and** B $\Rightarrow$ t $\geq$ 0

Die beiden ersten Punkte betreffen das Einbinden in die Spezifikation mit der Vorbedingung Q und der Nachbedingung R.
Der dritte Punkt garantiert die Invarianz von P im Schleifenrumpf. Die beiden letzten Punkte garantieren die Termination.
Punkt 3 und 4 werden üblicherweise getrennt verifiziert, können aber mit einer einzigen Bedingung formuliert werden:
3 und **4** {P **and** B **and** t} S {P **and** t < T}

Vorgehensweise, wenn P und t bekannt sind
Das Entwickeln einer Schleife besteht aus drei Teilaufgaben:

a Finde ein geeignetes Programmstück »Initialisiere P«, damit die Invariante P vor der Schleife gilt:
{Q} Initialisiere P {P}

b Finde eine geeignete Wiederholungsbedingung B, so dass nach der Schleife die gewünschte Nachbedingung R gilt:
P **and not** B $\Rightarrow$ R
Außerdem müssen die Invariante P und die Wiederholungsbedingung B so beschaffen sein, dass die Terminationsfunktion t vor dem Schleifenrumpf stets nichtnegativ ist, also
P **and** B $\Rightarrow$ t $\geq$ 0 gilt.

c Finde einen Schleifenrumpf S, der t verringert und P invariant lässt:
{P **and** B **and** t = T} S {P **and** t < T}.
Oft besteht der Schleifenrumpf wieder aus zwei Teilen. Der eine verringert die Terminationsfunktion, der andere stellt als Reaktion darauf die Gültigkeit der Invariante P wieder her.
S:»Verringere t«
»Stelle P wieder her«

Abb. 8.5-4: Verifikation und Entwicklung der abweisenden Wiederholung [Futs89, S. 75 ff.].

Beispiel B weggelassen, wird A zur Invariante und B zur Abbruchbedingung.

- Konstante durch Variable ersetzen:
  Die Invariante erhält man dadurch, dass eine in R vorkommende Konstante durch eine Variable mit einem bestimmten Wertebereich ersetzt wird.
- Kombinieren von Vor- und Nachbedingungen:
  Bei manchen Spezifikationen muss sowohl die Vorbedingung

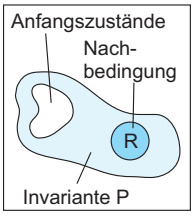


Abb. 8.6-1: Die Invariante ist eine Verallgemeinerung der Nachbedingung.

Q als auch die Nachbedingung R zu einer Invariante P verallgemeinert werden. Jede der beiden Zusicherungen Q und R wird zu einem Spezialfall der Invariante P.

Die beiden ersten Methoden sind die wichtigsten Standardmethoden.

Beispiel Weglassen einer Bedingung

Es soll die ganzzahlige Näherung der Quadratwurzel einer nichtnegativen ganzen Zahl A, die als fest angenommen wird, berechnet werden. Die Spezifikation lautet:

Q: $A \geq 0$
R: $x \geq 0$ **and** $x^2 \leq A < (x + 1)^2$

Die Nachbedingung R besteht aus den drei Bedingungen:

$x \geq 0$, $x^2 \leq A$ und $A < (x + 1)^2$

Eine davon, etwa die letzte, wird weggelassen. Dann erhält man als Invariante P:

$x \geq 0$ **and** $x^2 \leq A$ (Abb. 8.6-2).

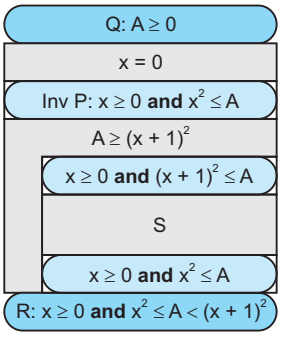


Abb. 8.6-2: Beispiel für das Weglassen einer Bedingung.

Die weggelassene Bedingung eignet sich hervorragend als Abbruchbedingung **not** B:

$A < (x + 1)^2$. Es gilt dann P **and not** B Þ R. Mit x := 0 findet sich eine einfache Initialisierung, sodass P vor der Schleife gilt. Im Programmskelett der Abb. 8.6-2 muss jetzt nur noch ein geeigneter Schleifenrumpf S gefunden werden.

Zur Entwicklung des Rumpfes S benötigt man eine Terminationsfunktion. Diese ergibt sich aus dem Vergleich zwischen der Initialisierung x = 0 und der Abbruchbedingung $A \geq (x + 1)^2$. Man sieht, dass für $x \geq 0$ die Variable x größer werden muss. Für die streng monoton fallende und nach unten beschränkte Terminationsfunktion wählt man t: A – x, da t bei wachsendem x fallend ist und bei Invarianz von P nicht negativ wird.

Der Schleifenrumpf muss x vergrößern. Eine geeignete Anweisung zum Vergrößern von x ist:

S: x = x + 1.

Mithilfe des Zuweisungsaxioms erhält man die Gültigkeit von $\{x \geq 0$ **and** $(x + 1)^2 \leq A\}$ x = x + 1 $\{P: x \geq 0$ **and** $x^2 \leq A\}$

Somit ist x = x + 1 bereits ein geeigneter Schleifenrumpf, der sowohl t verringert als auch P invariant lässt. In diesem Beispiel hätte man auch die Bedingung $x^2 \leq A$ von der Nachbedingung R weglassen können und damit eine andere Invariante und ein anderes Programm erhalten.

In der Abb. 8.6-3 ist zusammengestellt, wie man durch Weglassen einer Bedingung eine Schleife entwickelt. Die Methode »Weglassen einer Bedingung« eignet sich in jenen Fällen gut, in denen keine zusätzliche neue Variable in der Schleife verwendet werden muss.

Konstante durch Variable ersetzen

Ist hingegen die Verwendung einer neuen Variablen (etwa einer Laufvariablen) notwendig, empfiehlt es sich, die Methode »Konstante durch Variable ersetzen« zu verwenden (Abb. 8.6-4).

Kombinieren von Vor- und Nachbedingungen

Die Nachbedingung wird oft deswegen für die Konstruktion von Invarianten herangezogen, weil sie meist die wesentlichen Endergebnisse beschreibt und die Vorbedingung nur einige Randbedingungen festhält, die zu Beginn gelten sollen. Bei manchen Problemen ist für die Invariante die Vorbedingung genauso wichtig wie die Nachbedingung. Insbesondere dann, wenn ein Anfangszustand schrittweise in einen Endzustand überführt werden soll und dabei immer weniger Eigenschaften des Anfangszustandes und immer mehr Eigenschaften des Endzustandes angenommen werden sollen.

Methode
Gegeben sei eine Spezifikation {Q} . {R: A **and** B}. Die Nachbedingung R besteht aus mindestens zwei Bedingungen A und B.

1 Eine Invariante erhält man dadurch, dass man eine der Bedingungen weglässt. Wird B weggelassen, erhält man A als Invariante P.
2 Die weggelassene Bedingung **not** B wird zur Abbruchbedingung.
3 Die Invariante muss durch ein Programmstück initialisiert werden:
{Q} Initialisiere P {P: A}
4 Es bleibt ein Schleifenrumpf S zu entwickeln mit der Spezifikation
{A **and** B} S {A}.
Im Schleifenrumpf muss außerdem ein Fortschritt in Richtung Termination (Bedingung B ist erfüllt) gemacht werden. Die Terminationsfunktion ergibt sich oft aus dem Vergleich der Initialisierung mit der Abbruchbedingung **not** B.

Diese vier Schritte genügen, denn P **and not** B $\Rightarrow$ R braucht nicht bewiesen zu werden, da bei dieser Methode P **and not** B stets mit R identisch ist.

Besteht die Nachbedingung aus mehreren Bedingungen, dann gilt:
- Es bleiben die Bedingungen in der Invarianten erhalten, die sich leicht initialisieren lassen.
- Es werden die Bedingungen weggelassen, die sich gut als Abbruchbedingung **not** B eignen.

Beispiel
Die Nachbedingung R: X = q $*$ Y + r **and** $0 \leq r < Y$ bei der Ganzzahldivision wird dabei durch Weglassen der Bedingung r < Y zur Invarianten
P: X = q $*$ Y + r **and** $0 \leq r$
r < Y wird zur Abbruchbedingung und P kann dann leicht mit q = 0; r = X initialisiert werden.
Hätte man eine andere Bedingung weggelassen, wäre die Programmentwicklung schwieriger.
Wenn $0 \leq r$ weggelassen wird, könnte r zwar mit einer negativen Zahl initialisiert werden, aber es ist ungeklärt, mit welcher. Ebenso unklar ist die Frage der Initialisierung, wenn X = q $*$ Y + r weggelassen wird.
Daher kommt nur die vorgeschlagene erste Variante in Frage.

Abb. 8.6-3: Entwicklung einer Schleife durch Weglassen einer Bedingung [Futs89, S. 81 f.].

Methode
Eine Nachbedingung R kann dadurch abgeschwächt werden, dass eine in R vorkommende Konstante durch eine neue Variable ersetzt wird.

1 Für die Konstruktion der Invarianten P ersetze eine Konstante, etwa N, in der Nachbedingung R durch eine neue Variable, etwa n, und füge einen Wertebereich für n hinzu. Die Konstante N muss selbstverständlich im Wertebereich von n vorkommen.
2 Die Abbruchbedingung **not** B der Schleife ist n = N. P **and not** B $\Rightarrow$ R ist dann automatisch erfüllt.
3 Bestimme eine Initialisierung, so dass P vor der Schleife gilt.
{Q} Initialisiere P {P}
4 Finde einen Schleifenrumpf S mit
{P **and** B} S {P}
Die Terminationsfunktion ist häufig t: N – n, wenn n erhöht wird, und t: n, wenn n verringert wird.

Abb. 8.6-4: Entwicklung einer Schleife durch Variablenersetzung [Futs89, S. 87].

8.7 Vor- und Nachteile ***

Bei kurzen und einfachen Programmen kann die Korrektheit mithilfe der Verifikation gezeigt werden, bei umfangreicheren Programmen steigen die Schwierigkeiten stark an. Man sollte sich jedoch immer bemühen, die Invarianten als Kommentar in einem Programm anzugeben, da sie ein wichtiges Element der Programm-Dokumentation darstellen und bei der die Ermittlung von Invarianten bereits Fehler entdeckt werden können.

Frage Welche Vorteile hat die Verifikation?

Antwort
+ Es kann allgemeingültig bewiesen werden, dass ein Programm entsprechend seiner (formalen) Spezifikation, d. h. seiner Vor- und Nachbedingungen, implementiert ist.
+ Ein vollständiger Korrektheitsbeweis ist möglich.

Frage Welche Nachteile besitzt die Verifikation?

Antwort
- Für »nicht kleine« Programme immer noch nicht praktikabel.
- Die Aufbereitung der Programme für den Beweis erfordert eine hohe Qualifikation.
- Die verwendete Programmiersprache muss eine formale Semantik besitzen, um den Effekt jeder Sprachkonstruktion zu spezifizieren.
- Die Teile des Programms, in denen Sprachkonstrukte verwendet werden, die keine formale Semantik besitzen, wie Gleitpunktarithmetik, externes Ein-/Ausgabe-Verhalten, Interrupts, müssen weiterhin getestet werden.
- Maschineneigenschaften werden nicht berücksichtigt.
- Die Verifikation verlangt eine bestimmte Spezifikationstechnik (Anfangs- und Endebedingungen).

8.8 Box: Kreuzworträtsel 3 **

Lösen Sie das Kreuzworträtsel der Abb. 8.8-1. Die Musterlösung dazu finden Sie im Anhang.

Gesuchte Wörter:

1 Formaler Beweis, dass ein Programm das tut, was es nach der Spezifikation tun soll.
2 Speicherungsprinzip, bei dem das erste gespeicherte Element auch zuerst dem Speicher wieder entnommen wird (Kürzel).
3 Alle Daten, die benötigt werden, um ein Programm für einen Test auszuführen.
4 Was Sie von einer Operation unbedingt wissen müssen, um sie benutzen zu können.

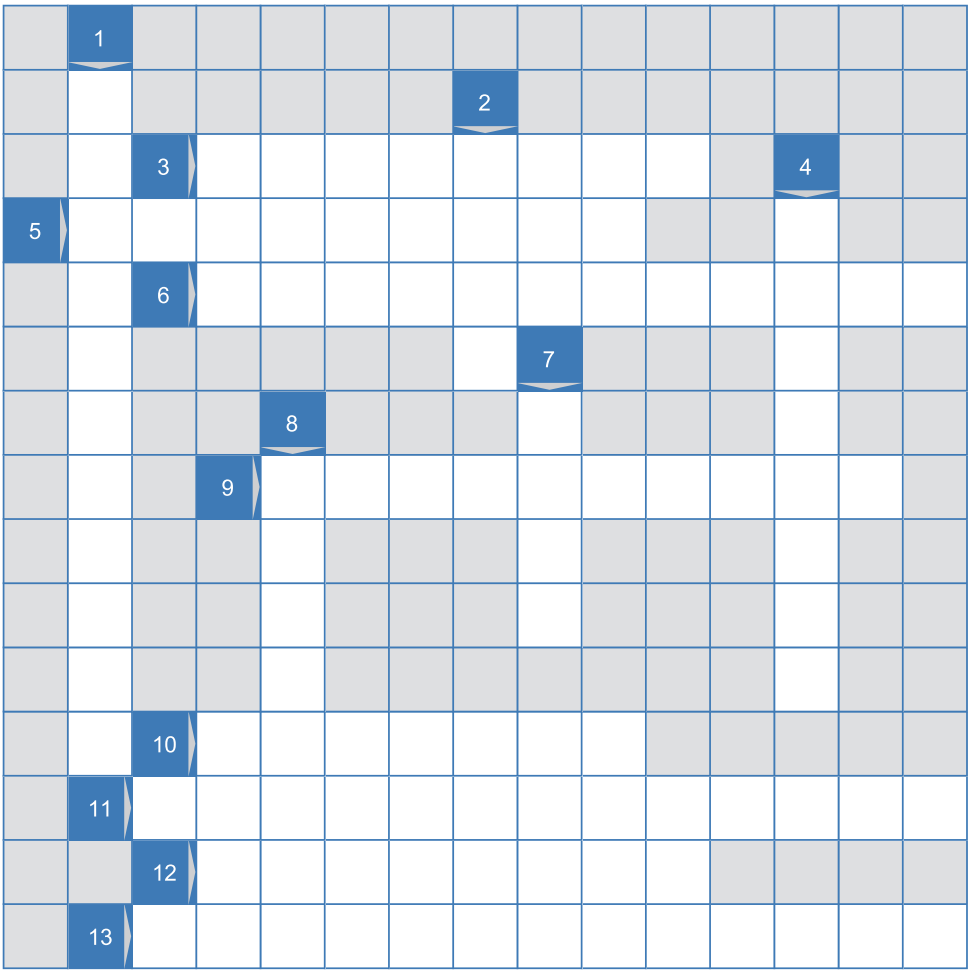


Abb. 8.8-1: Kreuzworträtsel zu Prozeduren, Testen und Verifikation.

5 Wenn sich ein Algorithmus direkt oder indirekt selbst aufruft.

6 Ermöglicht es, die partielle Korrektheit eines Programms zu beweisen.

7 Erlaubt es, Variablen vom gleichen Typ zu einer Einheit zusammenzufassen (deutscher Begriff).

8 Anweisungsfolge in einem Programm, die eine Dienstleistung erbringt, einen eigenen Namen besitzt, deklariert und aufgerufen wird.

9 Eine Programmiersprache ist ..., wenn sie es ermöglicht, Teilaufgaben in Prozeduren und Funktionen auszulagern.

10 In Programmiersprachen wie Java und Smalltalk als Bezeichnung für eine Prozedur und eine Funktion verwendet.
11 Spezifiziert eine Zusicherung nach dem Programmende.
12 Sonderfall einer Prozedur. Wird in Ausdrücken aufgerufen. Gibt ein Ergebnis zurück.
13 Datenstruktur mit den Operationen Einfügen und Entfernen

9 Die Programmiersprache C *

Damit Sie einen Eindruck von einer anderen Programmiersprache erhalten, wird hier eine kurze Einführung in C gegeben. C ist eine der am weitesten verbreiteten Programmiersprachen.

Zur Historie

Die Programmiersprache **C** wurde in den siebziger Jahren des letzten Jahrhunderts entwickelt – und sie zählt immer noch, neben Java, zu den beliebtesten Programmiersprachen. Dennis Ritchie, einer der Autoren des Klassikers »The C programming language« [KeRi88], hatte C für das UNIX-Betriebssystem entworfen und den ersten Compiler dafür geschrieben.

ANSI C

C fand so große Verbreitung, dass das American National Standards Institute (ANSI) die Sprache standardisierte. Das genannte Buch beschreibt den ANSI Standard für C, kurz ANSI C.

ISO

Die Fortentwicklung des Standards geschieht unter der Regie der International Standards Organisation (ISO).

Für die große Verbreitung und Beliebtheit von C gibt es folgende Gründe:

- C gibt es für nahezu jedes Betriebssystem.
- Mit C lässt sich die Hardware, einschließlich des Speichers, direkt ansprechen.
- C lässt sich leicht in Binärcode für einen Prozessor umsetzen.

Wegen der beiden zuletzt genannten Eigenschaften ist C sehr effizient. Im Vergleich zu anderen Programmiersprachen sind in C geschriebene Programme schnell, und der ausführbare Binär-Code hat eine geringe Größe. Deshalb wird C in der Systemprogrammierung und in eingebetteten Systemen eingesetzt. Auch die Java Virtual Machine ist in C geschrieben. C hat eine nur kleine Menge an Schlüsselwörtern. Für diverse Aufgaben, wie etwa das Schreiben einer Datei, stehen Bibliotheksfunktionen zur Verfügung.

Die genannten Vorteile können gleichzeitig Nachteile sein. So sind umfangreiche, problemorientierte Strukturen nicht verfügbar und müssen bei Bedarf nachgebildet werden. Ein Beispiel: Java stellt komfortable Elemente zur Programmierung von grafischen Benutzungsoberflächen als Teil der Sprache bereit. In C hingegen muss man sich dafür auf nicht standardisierte externe Bibliotheken verlassen. Ein weiteres Beispiel: Der genannte Vorteil, direkt auf den Speicher zugreifen zu können, ist bei falscher Handhabung ein Sicherheitsrisiko.

C diente den Programmiersprachen C++, Java und C# als Vorbild. Die Grunddatentypen und viele syntaktische Elemente wurden übernommen, zum Teil mit leichten Änderungen.

Diese Einführung in die Programmiersprache C ist wie folgt gegliedert:

- »»Hello World« in C«, S. 324
- »Einfache Datentypen«, S. 326
- »Einfache Ein- und Ausgabe«, S. 328
- »Kontrollstrukturen und Zusicherungen«, S. 331
- »Zeiger und Adressen«, S. 334
- »Felder«, S. 337
- »C-Zeichenketten«, S. 339
- »Strukturen«, S. 340
- »Dynamische Daten«, S. 342
- »Modularität«, S. 343

9.1 »Hello World« in C *

Jedes C-Programm muss genau eine `main`-Funktion besitzen, die beim Start zuerst ausgeführt wird. `main` gehört *nicht* zu einer Klasse – C kennt keine Klassen. Ein C-Compiler erzeugt keinen Zwischencode, der wie der Bytecode von Java von einem Programm interpretiert werden muss, sondern Maschinencode, der auf die CPU (Zentralprozessor) zugeschnitten und direkt von ihr ausführbar ist.

Der Tradition, als erstes Programm einer Programmiersprache ein Programm »Hello World« vorzustellen, wird auch hier gefolgt. Zum besseren Vergleich wird ein entsprechendes Java-Programm vorangestellt.

Beispiel

```
/*
  Hello World als Java-Programm
  Dies ist ein Kommentar.
*/
// Java-Zeilenkommentar
// Der Dateiname muss HelloWorld.java heißen!
public class HelloWorld
{
   public static void main(String args[])
   {
     System.out.println("Hello World");
   }
}
```

```
/*
  Hello World als C-Programm
  Kommentare sind wie in Java.
*/
// Auch Zeilenkommentare (seit C99).
// Ein Dateiname ist nicht vorgeschrieben,
// er muss nur auf .c enden.
#include<stdio.h>
```

```
int main(void)
{
   printf("Hello World\n");
   return 0;
}
```

Im Vergleich zu Java fällt auf, dass *keine* umgebende Klasse notwendig ist. C kennt keine Klassen.

`main()`

Die Funktion `main()` hat hier keinen Parameter (es kann welche geben). Das Schlüsselwort `void` kann entfallen. `main()` gibt einen Fehlercode an das aufrufende Programm zurück. Falls es keinen Fehler gab, ist der Fehlercode 0. Diese Information kann auf Betriebssystemebene ausgewertet werden, um zum Beispiel den Start anderer Programme davon abhängig zu machen. Das Zeichen `\n` in der Ausgabeanweisung sorgt für eine neue Zeile.

`include`

Die Zeile `#include<stdio.h>` bedeutet, dass an dieser Stelle die Datei `stdio.h` eingelesen wird. `stdio` steht für *standard input output*. In dieser Datei werden dem Compiler verschiedene Funktionen zur Ein- und Ausgabe bekanntgemacht, so auch `printf()`.

gcc-Compiler unter Linux

Um das Programm starten zu können, übertragen Sie es mit einem ASCII-Editor und speichern das Ergebnis in einer Datei mit dem Namen `hello.c` in einem Verzeichnis Ihrer Wahl. Danach öffnen Sie in diesem Verzeichnis ein Konsolenfenster. Mit dem Befehl `gcc` gefolgt von dem Dateinamen rufen Sie den C-Compiler auf, der dann die angegebene Quelldatei in eine ausführbare Binärdatei übersetzt. `gcc` steht für den GNU C Compiler, der auf jedem Linux-System vorhanden ist. Die ausführbare Binärdatei heißt meistens `a.out`, wenn der Dateiname *nicht* mit der Option `-o` vorgegeben wird. Das Programm wird durch Eingabe des Dateinamens im Konsolenfenster gestartet (Abb. 9.1-1).

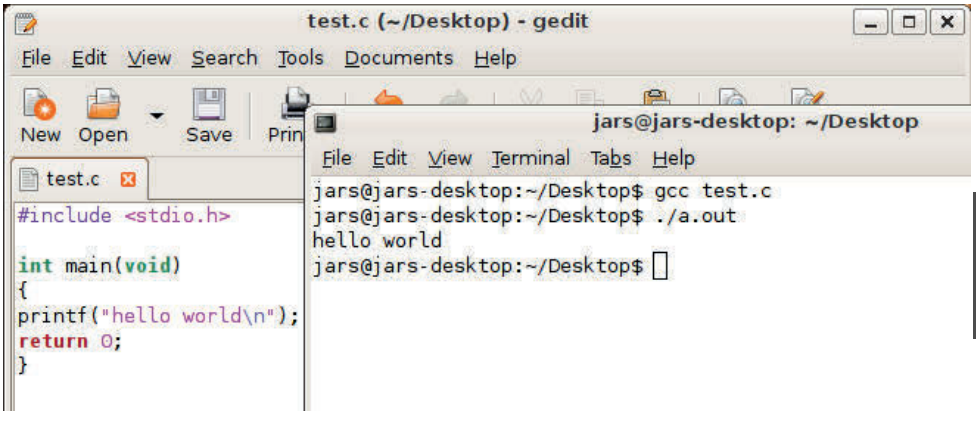


Abb. 9.1-1: »Hello World« in der Linux-Konsole.

gcc-Compiler unter Windows

Falls Sie Windows benutzen und eine Fehlermeldung bekommen, muss der Compiler erst installiert werden. Eine bekannte Implementierung des `gcc` für Windows ist der MinGW-Compiler, siehe Website MinGW (`http://www.mingw.org/`). Die Installation entsprechend der Anleitung auf der MinGW-Website ist etwas mühevoll. Einfacher ist es, die Datei `installcomp.exe` von der Website Software-Update für Windows (`http://www.cppbuch.de/downloads.html`) herunterzuladen und danach zu starten. Bei der Installation wird auch die Entwicklungsumgebung `Code::Blocks` eingerichtet. Nach der Installation müssen Sie sich als Benutzer in Windows »abmelden« und dann wieder »anmelden«, da Änderungen im Pfad bei Windows erst dann wirksam werden.

`Code::Blocks`

Im E-Learning-Kurs zu diesem Buch wird die Entwicklungsumgebung`Code::Blocks` näher beschrieben. Lassen Sie sich nicht von dem Namen »C++-Compiler« irritieren – ein C++-Compiler ist ebenso in der Lage C zu kompilieren, da es eine Untermenge von C++ darstellt. Die durch die Compilation entstehende Binärdatei heißt unter Windows `a.exe`, sofern kein Name angegeben wird.

Empfehlung

> Fügen Sie stets die Optionen `-std=c99` und `-Wall` hinzu! Die erste Option bewirkt, dass nach dem C-Standard von 1999 übersetzt wird (statt 1990). Die zweite Option veranlasst den Compiler, nicht nur Syntaxfehler, sondern auch Warnungen auszugeben. Viele Fehler können damit früher erkannt werden.

9.2 Einfache Datentypen *

In C gibt es einfache Typen – analog zu Java. Sie haben jedoch etwas andere Eigenschaften.

Es gibt die Ganzzahl-Typen `short`, `int`, `long int` und `long long int`. Anders als in Java hängt der benötigte Speicherplatz für jeden dieser Typen vom System ab. Mit »System« ist die Kombination von Computer, Betriebssystem und Compiler gemeint. Vorgeschrieben ist nur, dass ein Typ der obigen Aufzählung mindestens soviele Bytes beansprucht wie der vorhergehende.

sizeof

`short` muss mindestens zwei Bytes lang sein. Der tatsächlich benötigte Speicherplatz kann mit dem Operator `sizeof` ermittelt werden.

Probieren Sie es, indem Sie die Datei `hello.c` (siehe ››Hello World« in C«, S. 324) um die Zeile

```
printf("sizeof(long) = %i\n", sizeof(long));
```

ergänzen und ausführen!

Bei zusammengesetzten Typnamen kann das Wort `int` entfallen; zum Beispiel genügt es, `long` statt `long int` zu schreiben.

Das Prozentzeichen ist eine Formatanweisung und bedeutet, dass an der betreffenden Stelle die nachfolgende Zahl eingefügt wird.

unsigned

Das Schlüsselwort `unsigned` kann vor den Namen ganzzahliger Typen gestellt werden. In so einem Fall kann der Wert nicht negativ werden. Java kennt keine `unsigned`-Zahlen. Das für das Vorzeichen nicht mehr benötigte Bit kommt dem Zahlenbereich zugute. So liegt ein 32 Bit-`int` im Bereich von -2147483648 bis 2147483647, ein 32 Bit-`unsigned int` im Bereich von 0 bis 4294967295.

Vermeiden Sie die gemischte Verwendung von `unsigned int` und `int`! Um einen Vergleich zu berechnen, muss der Compiler den einen in den anderen Typ umwandeln.

Welches Verhalten erwarten Sie bei dem folgenden Programmstück? Probieren Sie es aus und vergleichen Sie das Ergebnis mit Ihrer Erwartung!

```
int i = -1;
unsigned int u = 0;
if(u < i)
   printf("%i ist kleiner als %i", u, i);
else
   printf("%i ist groesser oder gleich  als %i", u, i);
```

float, double

In C darf die Genauigkeit von `double`-Zahlen nicht schlechter sein als die von `float`-Zahlen, und die von `long double`-Zahlen darf nicht schlechter sein als die von `double`-Zahlen. Die Eigenschaften von Java-Gleitpunktzahlen sind in der Norm IEEE754 festgelegt. Für C gibt es keine vergleichbare Festlegung, sodass numerische C-Programme nur begrenzt portabel sind.

Variable werden, anders als in Java, *nicht* automatisch mit 0 initialisiert, wenn kein Wert angegeben ist!

char

In Java umfasst der Typ `char` Unicode-Zeichen, in C belegt er jedoch nur ein Byte und ist damit nur für ASCII-Zeichen und einige Sonderzeichen geeignet. Der Typ `wchar` *(wide character type)* dient dazu, jedes Zeichen der Landeseinstellung darzustellen zu können.

Ein `char`-Wert kann je nach System vorzeichenbehaftet sein (`signed char`) oder nicht (`unsigned char`). Dies muss bei der Umwandlung in eine `int`-Zahl oder zurück beachtet werden.

Wahrheitswerte

Der Wert einer ganzen Zahl wird in C auch als Wahrheitswert bzw. Boolescher Wert interpretiert. Dabei gilt 0 als `falsch` *(false)*. Jeder andere Wert gilt als `wahr` *(true)*. Das gilt auch für alle Typen, die sich leicht in eine `int`-Zahl umwandeln lassen.

Beispiel

Bei der Berechnung der Fakultät von 5 bricht die Schleife ab, wenn `n` den Wert 0 erreicht hat:

```
int n = 5;
long fak = 1;
while(n)
{
   fak = fak * n;
   --n; //Kurzform für n = n - 1;
}
```

Die Mischung des Konzepts »Wahrheitswert« mit ganzen Zahlen wurde als unbefriedigend empfunden. Der halbherzige Ausweg war die Einführung eines neuen Typs `_Bool`, der nur die Werte 0 oder 1 annehmen kann. Dabei wird jeder Wert ungleich 0 in eine 1 umgewandelt. Javas typsicheres `boolean` ist da sehr viel eleganter und zuverlässiger.

Konstante

Mit dem Schlüsselwort `const` werden unveränderliche Werte definiert. `const` hat bei einfachen Typen dieselbe Bedeutung wie das Java-Schlüsselwort `final`. Bei komplexeren Typen (Referenztypen) gilt das nicht.

Beispiel

```
const int KONSTANTE = 1000;
```

9.3 Einfache Ein- und Ausgabe *

Für die Eingabe mit der Tastatur (Standardeingabe) und die Ausgabe auf dem Bildschirm (Standardausgabe) stellt C Bibliotheksfunktionen bereit. Java wurde um die C-Funktion `printf()` ergänzt.

Ausgabe

Die Funktion `printf(X)` dient zur Ausgabe. Wenn der Parameter `X` eine Zeichenkette ist, die *kein* Prozentzeichen enthält, wird sie einfach ausgegeben. Ein Prozentzeichen ist der Beginn einer Formatanweisung. Die danach folgende Zahlen- und Buchstabenkombination bestimmt das Format. Wenn ein Prozentzeichen selbst ausgegeben werden soll, ist es zu verdoppeln. `printf()` gibt die Anzahl der ausgegebenen Zeichen zurück; dies wird hier nicht ausgewertet. Das folgende Beispiel zeigt die wichtigsten Möglichkeiten. `\n` sorgt jedesmal für eine neue Zeile.

Beispiel 1a

```
#include<stdio.h>

int main()
{
   printf("Beispiel\n");     // einfacher Text
   int n = 13;
   // Ausgabe des Strings Zahl,
   // gefolgt von der Zahl n in drei Formaten:
```

```
    printf("%s dezimal: %d oktal: 0%o hexadezimal: 0x%x\n",
           "Zahl", n, n, n);
    printf("%6d\n", n);       // Ausgabe mit Feldweite 6
    double z = -3141.5926;
    printf("%f\n", z);        // Standardformat
    printf("%10.2f\n", z);    // Feldweite 10, 2 Dezimal-Stellen
    // Prozentzeichen selbst ausgeben
    printf("Prozentzeichen %%\n");
    return 0;
}
```

Wie Sie sehen, kann `printf()` mit unterschiedlich vielen Parametern aufgerufen werden. Mit Einführung von Java 5 wurde auch die Funktion `printf()` in die API mit aufgenommen.

Zum Vergleich hier das entsprechende Java-Programm:

Beispiel 1b

```
public class Printf
{
   public static void main(String args[])
   {
      System.out.printf("Beispiel\n");     // einfacher Text
      int n = 13;
      // Ausgabe des Strings Zahl,
      // gefolgt von der Zahl n in drei Formaten:
      System.out.printf(
             "%s dezimal: %d oktal: 0%o hexadezimal: 0x%x\n",
             "Zahl", n, n, n);
      System.out.printf("%6d\n", n); // Ausgabe mit Feldweite 6
      double z = -3141.5926;
      System.out.printf("%f\n", z);        // Standardformat
      // Feldweite 10, 2 Dezimal-Stellen:
      System.out.printf("%10.2f\n", z);
      // Prozentzeichen selbst ausgeben:
      System.out.printf("Prozentzeichen %%\n");
   }
}
```

Eingabe

Die Bibliotheksfunktion `scanf()` dient zur Eingabe von der Konsole (Standardeingabe). Ähnlich wie bei `printf()` gibt es Formatzeichen zur Steuerung.

Beispiel 2a

Das folgende Programm liest eine `float`- und eine `double`-Zahl ein. Anschließend werden ganze Zahlen eingelesen, bis ein nicht passendes Zeichen eingegeben wird. `scanf()` gibt die Anzahl der eingelesenen Elemente zurück. In der Schleife gibt es nur die Möglichkeiten 1 (genau eine Zahl, wie vom Formatstring verlangt) oder 0 (es ist ein Fehler aufgetreten und es wurde nichts eingelesen). Um das Verhalten des Programms identisch mit dem Vergleichsprogramm in Java zu machen, wird bei beiden eingestellt, dass Dezimalpunkte statt -kommas einzugeben sind.

```
#include<stdio.h>
#include<locale.h>

int main()
{
   setlocale(LC_ALL, "C");  // Dezimalpunkt statt -komma
   // Standardeinstellung für C-Programme
   printf("Float-Zahl eingeben: ");
   float f;
   scanf("%f", &f); // & = Adressoperator
   printf("Eingelesen: f = %f\n", f);
   printf("Double-Zahl eingeben: ");
   double d;
   // Buchstabe l vor dem f kennzeichnet double
   scanf("%lf", &d);
   printf("Eingelesen: d = %f\n", d);
   printf("Bitte ganze Zahlen eingeben (X=Ende):");
   int n;
   while(scanf("%i", &n) == 1)
   {
      printf("Eingelesen: n = %i\n", n);
   }
   return 0;
}
```

Adressoperator

Neu ist der **Adressoperator** `&` in der Parameterliste von `scanf()`. Die eingelesene Zahl soll der Variablen `f` zugewiesen werden. Eine Übergabe von `f` per Wert würde scheitern, weil eine Modifikation von `f` innerhalb von `scanf()` beim Aufrufer ohne Wirkung bliebe. Eine Übergabe per Wert *(call by value)* bewirkt, dass in einer Funktion mit einer Kopie gearbeitet wird.

Beispiel

```
void quadriere(int x) { x = x*x; }

Aufruf:
int x = 3;
quadriere(x);
System.out.println(x); // 3! x ist nicht verändert.
```

Also wird in Beispiel 2a die **Adresse** von `f` übergeben, sodass `scanf()` weiß, wohin der eingelesene Wert zu schreiben ist. Eine Adresse ist ein Verweis oder Zeiger auf eine Stelle im Speicher. Dasselbe gilt für die Variablen `d` und `n` im Beispiel 2a.

Java

Java kennt keinen Adressoperator und es gibt auch keine Funktion, die vergleichbar mit `scanf()` ist. Es gibt aber die Klasse `Scanner`, die Ähnliches leistet, wie das folgende Java-Programm zeigt.

Beispiel 2b

```
import java.util.Locale;
import java.util.Scanner;

public class Scanf
```

```
{
   public static void main(String args[])
   {
      Scanner sc = new Scanner(System.in);
      sc.useLocale(Locale.US);  // Dezimalpunkt statt -komma
      System.out.print("Float-Zahl eingeben: ");
      float f = sc.nextFloat();
      System.out.println("Eingelesen: f = " +  f);
      System.out.print("Double-Zahl eingeben: ");
      double d = sc.nextDouble();
      System.out.println("Eingelesen: d = " + d);
      System.out.print("Bitte ganze Zahlen eingeben (X=Ende):");
      while(sc.hasNextInt())
      {
         int n = sc.nextInt();
         System.out.println("Eingelesen: n = " + n);
         sc.nextLine(); // Zeilenendekennung löschen
      }
   }
}
```

Führen Sie die C-Programme aus und modifizieren Sie die Parameter.

9.4 Kontrollstrukturen und Zusicherungen *

Bei den Kontrollstrukturen gibt es keine Unterschiede zu Java. Zusicherungen mit `assert` werden geringfügig anders gestaltet.

Kontrollstrukturen

Java hat die Kontrollstrukuren `if-else`, `for`, `while`, `do-while` und `switch-case` von C geerbt. Syntax und Verhalten sind gleich geblieben. Zu erweiterten Java-`for`-Schleifen gibt es in C *keine* Entsprechung.

Zusicherungen

Wie in Java werden Zusicherungen mit `assert` realisiert, die Handhabung ist jedoch anders.

`assert`

Beispiel

Das folgende Programm berechnet die Fakultät (n!) mit einer Funktion `fakultaet(n)`, wobei vor dem Aufruf mit `assert()` geprüft wird, ob der Parameter `n` nicht-negativ ist. 0 ist erlaubt, weil konventionsgemäß 0! = 1 gilt. Wenn die Prüfung fehlschlägt, wird das Programm mit einer Fehlermeldung abgebrochen.

```
#include<stdio.h>       // scanf(), printf()
#include<assert.h>      // assert()
```

```
#include<limits.h>     // maximaler unsigned long long Wert

unsigned long long fakultaet(int n)
{
   unsigned long long fak = 1;
   while(n > 1)
   {
     if(fak >= (ULLONG_MAX / n)) // Überlauferkennung, s.u.
     {
        return ULLONG_MAX;
     }
     fak *= n--;
   }
   return fak;
}

int main()
{
   printf("Fakultätsberechnung: Bitte Zahl >= 0 eingeben: ");
   int n;
   if(scanf("%i", &n) == 1)
   {
      assert(n >= 0);                // Zusicherung
      unsigned long long ergebnis = fakultaet(n); //Aufruf
      if(ergebnis != ULLONG_MAX)
      {
         // ll: long long, u: unsigned
         printf("%llu\n", ergebnis);
      }
      else
      {
         // Der Compiler fügt zwei aufeinanderfolgende
         // Zeichenketten zusammen:
         printf("Die eingegebene Zahl ist zu groß. "
            "Fakultät kann nicht berechnet werden.\n");
      }
   }
   else
   {
      printf("Falsche Eingabe!\n");
   }
   return 0;
}
```

limits.h

Weil die Ganzzahlarithmetik keinen Überlauf erkennt, wird dieser Fall besonders geprüft. ULLONG ist der größtmögliche unsigned long long Wert; seine Definition wie auch die Definition anderer Maximal- und Minimalwerte finden Sie in der Datei limits.h.

NDEBUG

Zusicherungen sind abschaltbar. Wenn in der Zeile *vor* dem Einschließen von assert.h das Makro NDEBUG*(= no debug)* eingefügt wird, sind nachfolgende assert-Anweisungen ohne Wirkung.

Zum Vergleich finden Sie hier das entsprechende Java-Programm. Damit die Zusicherung wirksam wird, muss es mit der Option `-ea` *(enable assertions)* ausgeführt werden, also zum Beispiel `java -ea Fakultaet`.

```
import java.util.Scanner;

public class Fakultaet
{
   public static long fakultaet(int n)
   {
      long fak = 1;
      while(n > 1)
      {
         if(fak >= (Long.MAX_VALUE / n)) // Überlauferkennung
         {
            return Long.MAX_VALUE;
         }
         fak *= n--;
      }
      return fak;
   }

   public static void main(String args[])
   {
      System.out.print(
           "Fakultätsberechnung: Bitte Zahlen >= 0 eingeben:");
      Scanner sc = new Scanner(System.in);
      if(sc.hasNextInt())
      {
         int n = sc.nextInt();
         assert(n >= 0);
         long ergebnis = fakultaet(n);
         if(ergebnis != Long.MAX_VALUE)
         {
            System.out.println(ergebnis);
         }
         else
         {
            System.out.println(
                "Die eingegebene Zahl ist zu groß." +
                "Fakultät kann nicht berechnet werden.");
         }
      }
      else
      {
         System.out.println("Falsche Eingabe!");
      }
   }
}
```

Führen Sie das C-Programm mit verschiedenen Werten aus.

9.5 Zeiger und Adressen *

Zeiger und Adressen haben kein direktes Gegenstück in Java. Die Referenzen in Java sind C-Zeigern ähnlich. In C lassen sich allerdings mit Zeigern echte Speicheradressen ermitteln, und man kann direkt in den Speicher hineinschreiben. Auch gibt es Zeiger auf einfache Typen.

Adresse

Zeiger *(pointer)* haben einen Namen und einen Wert und können mit Operatoren verändert werden. Der Unterschied zu anderen Variablen besteht darin, dass der Wert als **Speicheradresse** interpretiert wird. Zeiger werden in C sehr häufig verwendet, weil sie eine große Flexibilität gestatten. Mit Hilfe von Zeigern kann dynamisch, d. h. zur Laufzeit eines Programms, Speicher reserviert werden.

* in Deklarationen

In Deklarationen bedeutet ein `*` »Zeiger auf«, zum Beispiel `int *ip;`. `ip` ist ein Zeiger auf einen `int`-Wert oder anders ausgedrückt: In der Speicherzelle, deren Adresse in `ip` gespeichert ist, befindet sich ein `int`-Wert.

Dereferenzierung

In anderen Anweisungen bedeutet `*` eine **Dereferenzierung**, d. h., dass der Wert an der Stelle genommen wird, auf die der Zeiger verweist.

`*ip = 100;` setzt den Wert der Speicherzelle, auf die `ip` zeigt, auf 100. Sie können die Speicheradresse, auf die ein Zeiger verweist, direkt ausgeben, und auch den Inhalt des Speichers, unabhängig von der Art des Inhalts, wie untenstehendes Beispiel zeigt.

Zeiger erhalten bei der Deklaration zunächst eine *beliebige* Adresse, genau wie andere nicht-initialisierte Variable zunächst beliebige Werte annehmen. Daher muss vor Benutzung des Zeigers in einem Ausdruck erst eine sinnvolle Adresse zugewiesen werden, um nicht den Inhalt anderer Speicherzellen zu zerstören!

Beispiel

Zur Verdeutlichung wird eine Variable `i` mit der Anweisung `int i = 99;` definiert und initialisiert. Damit wird ein Speicherplatz mit dem symbolischen Namen `i` angelegt und dort die Zahl 99 eingetragen. Die unbekannte, vom Compiler für `i` festgelegte relative Speicherplatzadresse sei 10125 (Abb. 9.5-1).

Die Abb. 9.5-1 entspricht den folgenden Anweisungen:

```
int* ip;     // Deklaration eines nicht-initialisierten Zeigers.
int i = 99; // Variable i mit Wert 99 anlegen.
ip = &i;    // Dem Zeiger die Adresse von i zuweisen.
```

Bei der Deklaration ist es gleichgültig, wo das `*` zwischen Typ und Bezeichner steht.

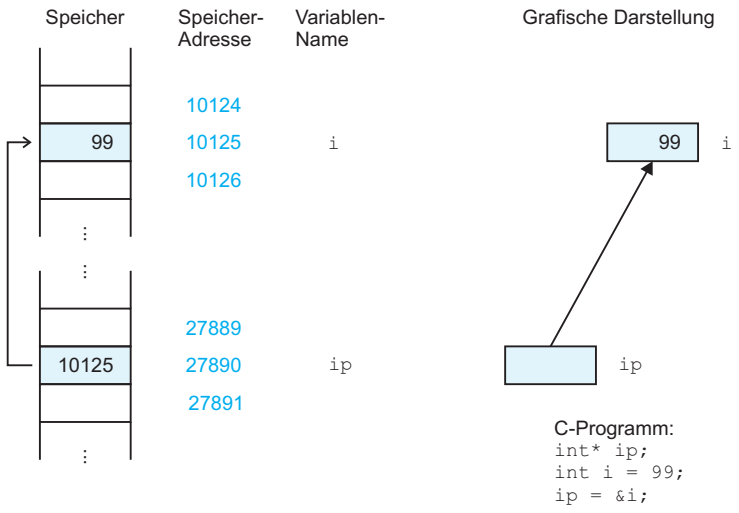


Abb. 9.5-1: Der Zeiger `ip` zeigt auf `i`.

Es gilt aber nur für den direkt folgenden Bezeichner. Die Deklarationen 1 bis 3 sind gleichwertig:

```
int*  ip, x;       // 1 (p steht für pointer)
int  *  ip, x;     // 2
// Empfehlung: Nur eine Deklaration pro Zeile, damit
// deutlich wird, dass x kein Zeiger ist:
int *ip;           // 3
int x;
```

In C gibt es einen speziellen Zeigerwert, nämlich `NULL`. `NULL` ist in der Header-Datei `stddef.h` definiert. Ein mit `NULL` initialisierter Zeiger zeigt nicht irgendwohin, sondern definitiv auf »nichts«. `NULL` ist als logischer Wert abfragbar. Um sich zu merken, dass ein Zeiger noch nicht oder nicht mehr auf ein definiertes Objekt zeigt, kann ein Zeiger auf `NULL` gesetzt werden: `int *iptr = NULL;`.

NULL

Das folgende Programm zeigt, wie mit einer Funktion zwei Zahlen vertauscht werden. Die Funktion `vertauschen()` hat zwei Zeiger (Adressen) als Parameter. Beim Aufruf werden die Adressen zweier Variablen übergeben.

Beispiel

```
#include<stdio.h>

void vertauschen(int *pa, int *pb) // zwei Zeiger auf int-Werte
{
   int tmp = *pa;  // Dereferenzierung: Wert an der Stelle pa
   *pa = *pb; // Wert an der Stelle pa = Wert an der Stelle pb
   *pb = tmp; // Wert an der Stelle pb = tmp
}
```

```
int main()
{
   int a = 17;
   int b = 100;
   printf("a = %i, b = %i\n", a, b);
   vertauschen(&a, &b);  // Adressen übergeben
   printf("a = %i, b = %i\n", a, b);
   return 0;
}
```

In Java ist es *nicht* möglich, nur mit einem Funktionsaufruf zwei Werte zu vertauschen. Der Grund ist, dass es nur die Parameterübergabe per Wert gibt. Allenfalls mit einer Hilfskonstruktion, wie etwa einem zurückgegebenen Feld mit dem Ergebnis der Vertauschung, lässt sich das Problem lösen:

```
public class Vertauschen
{
   public static int[] vertauschen(int x, int y)
   {
      return (new int[] {y, x});
   }

   public static void main(String args[])
   {
      int a = 17;
      int b = 100;
      System.out.println("a = " + a + "   b = " + b);
      int [] ergebnis =  vertauschen(a, b);
      a = ergebnis[0];
      b = ergebnis[1];
      System.out.println("a = " + a + "   b = " + b);
   }
}
```

Um zu zeigen, dass mit C direkt auf den Speicher zugegriffen werden kann, druckt das folgende Programm einen Teil des Speichers im Hexadezimalformat mitsamt den Speicheradressen aus. Falls ein Byte druckbar ist, wird es zusätzlich als ASCII-Zeichen ausgegeben, andernfalls erscheint nur ein Punkt. Zur Wiedererkennung beginnt der Ausdruck des Speichers an der Stelle der Variablen `text`.

Beispiel

```
#include<stdio.h>
#include<ctype.h>  // isprint()

int main()
{
   const char* text = "Ein String 12345";
   unsigned char* p = (unsigned char*)text;
   const unsigned SPALTEN = 8;
   char zeile[SPALTEN+1];  // Platz für ASCII-Zeichen
   zeile[SPALTEN] = '\0';  // terminierendes Null-Byte
   for(unsigned zeilen = 0; zeilen < 10; ++zeilen)
```

```
   {
     printf("%p   :", p);     // Formatierung: p für pointer
     for(unsigned i = 0; i < SPALTEN; ++i)
     {
        if(*p < 16)           // einstellig? Dann 0 voranstellen
           printf("0");
        printf("%x ", *p);    // Ausgabe im Hex-Format
        if(isprint(*p))       // druckbares Zeichen?
          zeile[i] = *p;
        else
          zeile[i] = '.';     // Ersatzzeichen
        ++p;                  // zum nächsten Byte gehen
     }
     // Speicherinhalt als ASCII drucken
     printf("   %s\n", zeile);
   }
   return 0;
}
```

Die Abfrage `isprint()` prüft, ob das Zeichen zwischen den Positionen 32 (Leerzeichen) und 126 der ASCII-Tabelle liegt. Zeichen mit einem Wert kleiner als 32 sind Steuerzeichen, wie zum Beispiel das Tabulatorzeichen. Ein `char`-Feld kann man ohne Schleife mit `printf()` ausgeben, wenn das letzte Zeichen das Null-Byte ist.

Der Speicherauszug sieht wie folgt aus:

```
00403064   :45 69 6e 20 53 74 72 69    Ein Stri
0040306C   :6e 67 20 31 32 33 34 35    ng 12345
00403074   :00 25 70 20 20 20 3a 00    .%p   :.
0040307C   :25 78 20 00 20 20 20 25    %x .   %
00403084   :73 0a 00 00 00 00 00 00    s.......
0040308C   :00 00 00 00 00 00 00 00    ........
00403094   :00 00 00 00 00 00 00 00    ........
0040309C   :00 00 00 00 00 00 00 00    ........
004030A4   :00 00 00 00 00 00 00 00    ........
004030AC   :00 00 00 00 00 00 00 00    ........

Process returned 0 (0x0)   execution time : 0.016 s
```

Führen Sie die C-Programme aus modifizieren Sie die Werte.

9.6 Felder *

Felder, auch Arrays genannt, gibt es in Java und in C. Feldnamen sind in C konstante Zeiger. Die Länge des Feldes ist nicht im Feld selbst gespeichert.

Es ist guter Programmierstil, die Größe eines Feldes als **Konstante** zu deklarieren und die Konstante im restlichen Programm zu verwenden. Dadurch kann ein Programm leicht an eine andere

Felder

Feldgrößen angepasst werden, indem nur der Wert der Konstanten geändert wird.

Beispiel

```
const int ANZAHL = 5;
// Beispiel eines eindimensionalen Feldes
int Tabelle[ANZAHL];
```

Der Compiler reserviert für alle Elemente ausreichend Speicherplatz. Die Anzahl der Tabellenelemente ist während des Programmlaufs nicht veränderbar. Arrays, deren Größe erst zur Laufzeit des Programms festgelegt wird, lassen sich auch konstruieren (siehe unten). Die Abb. 9.6-1 zeigt ein Array mit 5 ganzen Zahlen.

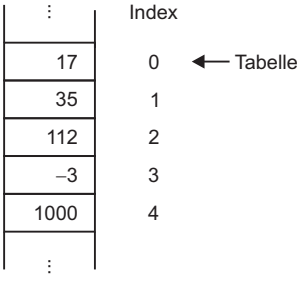


Abb. 9.6-1: int-Feld Tabelle.

Feldname = konstanter Zeiger

Der Name des Feldes zeigt auf die Startadresse, d. h. auf das erste Element, und ist wie ein Zeiger einsetzbar. Anders ausgedrückt: Es gilt `*Tabelle == Tabelle[0]`. Der Zugriff auf ein Element ist auch über die Zeigerschreibweise möglich. Die beiden Anweisungen

```
Tabelle[3]   = 1000;   // Zugriff mit []
*(Tabelle+3) = 1000;   // Zeigernotation
```

sind gleichwertig! Da dem Array bereits fest Speicherplatz zugewiesen ist, würde eine Änderung dieses Zeigers den Speicherplatz unzugänglich machen, weil die Information über die Adresse verloren geht. Der Feldname ist deshalb ein *konstanter* Zeiger.

Die Länge des Feldes ist nicht im Feld selbst gespeichert. `Tabelle.length` gibt es in Java, aber nicht in C. Aus diesem Grund muss eine Funktion, die mit dem Feld arbeitet, die Länge separat mitgeteilt bekommen, wie an der folgenden Funktion `maximum()` zu sehen.

```
#include<stdio.h>

double maximum(double *feld, unsigned anzahl)
{
```

```
   double max = *feld; // 1. Element (d.h. Nr. 0)
   for(unsigned i = 1; i < anzahl; ++i)
   {
      if(feld[i] > max)       // Indexoperator [ ]
         max = *(feld + i);   // Zeigernotation
   }
   return max;
}

int main()
{
   const unsigned ANZAHL = 5;
   double Feld[ANZAHL];      // Feld ist nicht initialisiert!
   // daher: Feld füllen
   for(unsigned i = 0; i < ANZAHL; ++i)
   {
      Feld[i] = (double)i*i; // beliebige Werte
   }
   printf("Maximum = %lf\n", maximum(Feld, ANZAHL));
   return 0;
}
```

Führen sie das C-Programm mit verschiedenen Varianten aus.

9.7 C-Zeichenketten **

Eine C-Zeichenkette oder C-String ist nur ein Spezialfall eines Arrays und hat deswegen nicht annähernd die Möglichkeiten eines Java-Strings. Eine C-Zeichenkette kann als Zeiger auf ein Stringliteral oder als Array formuliert werden.

Mit C-String meint man eine Folge von Zeichen des Typs `char`, die mit `\0` abgeschlossen wird. `\0` ist das ASCII-Zeichen mit dem Wert `0`, nicht das Ziffernzeichen `'0'`.

Abschluss mit Null-Byte

Der Typ für einen C-String ist `char*` und stellt einen Zeiger auf den Beginn der Zeichenfolge dar. Bei der Ausgabe einer Zeichenkette »weiß« der Compiler, dass `char*` *nicht* als Zeiger, sondern als null-terminierter String aufzufassen ist.

Beispiel

```
// Zeiger auf nicht veränderbare Zeichen
const char* str = "ABC";
printf("%s\n", str);   // Ausgabe: ABC
```

Hier wird `str` gleichzeitig definiert und initialisiert. Sie müssen an dieser Stelle `\0` nicht hinschreiben, weil es vom Compiler ergänzt wird. `str` zeigt also auf den Anfang einer Folge von 4 Bytes. Im Programm vorkommende Zeichenliterale liegen bei Ausführung zusammen mit dem Programm in einem nicht-veränderbaren Speicherbereich. Daher ist eine Änderung nicht erlaubt. Aber nur bei Verwendung von `const` gibt es gegebenenfalls eine Fehlermeldung des Compilers. Anders

ist es bei Feldern, die auf dem Laufzeit-Speicher *(stack)* angelegt werden:

```
char text[] = "hallo"; // 6 Bytes (einschließlich Null-Byte)
text[0] ='a'; // Änderung möglich
```

`strlen()` Für C-Zeichenketten gibt es eine Menge von Bibliotheksfunktionen. Hier sei nur die Funktion `strlen()` erwähnt, die die Anzahl der Zeichen (ohne Null-Byte) zurückgibt.

Beispiel Um zu sehen, wie `strlen()` intern arbeitet, zeigt dies Beispiel die Funktion `laenge()`, die dasselbe leistet.

```
// Gibt die Anzahl der Zeichen im String str zurück
unsigned laenge(const char* str)
{
   unsigned ergebnis = 0;
   while(*str++)
      ++ergebnis;
   return ergebnis;
}
```

Im Unterschied zur Übergabe eines Feldes, das keine Zeichenkette darstellt, muss die Anzahl der Elemente *nicht* übergeben werden. In der `while`-Schleife wird geprüft, ob das abschließende Null-Byte schon erreicht ist. `*str` ist das Zeichen an der Stelle `str`. Wenn `*str` nicht 0 ist, wird `ergebnis` hochgezählt. Das nachgestellte `++` sorgt dafür, dass nach der Prüfung der Zeiger um eine Position weitergeschaltet wird – ein typisches C-Idiom. Irgendwann verweist `str` auf das Null-Byte und die Schleife bricht ab.

Gehen Sie die C-Programme Zeile für Zeile durch und führen Sie sie auf Ihrem Computer aus.

9.8 Strukturen **

Eine Struktur kapselt Variablen, auch unterschiedlichen Typs, um sie unter nur einem Namen ansprechen zu können. In Java gibt es keine Strukturen.

`struct` Eine **Struktur** – auch Verbund genannt – definiert für inhaltlich zusammengehörende Daten einen neuen Typ. Anders als bei einem Feld, das Daten gleichen Typs gruppiert, können die Bestandteile einer Struktur verschiedene Typen besitzen. Mit dem Punkt-Operator (`.`) wird auf einzelne Elemente zugegriffen. Strukturen können geschachtelt sein. Das folgende Beispiel zeigt beides.

Beispiel

```
#include<stdio.h>

struct Punkt            // Struktur
{
   int x;
   int y;
};

struct Strecke          // geschachtelte Struktur
{
   struct Punkt anfang;
   struct Punkt ende;
};


int main()
{
   struct Punkt p1;
   p1.x = 0;    // Initialisierung über .-Operator
   p1.y = 0;    // Initialisierung über .-Operator
   struct Punkt p2 =  { 100, 200};  // direkte Initialisierung
   struct Strecke s = { p1, p2};    // direkte Initialisierung
   printf("Strecke von  (%i, %i) nach (%i, %i)\n",
          s.anfang.x, s.anfang.y, s.ende.x, s.ende.y);
   return 0;
}
```

Die Variable p1 im Programm ist vom Typ Punkt – sie ist ein Punkt-Objekt.

Das folgende Beispiel zeigt verschiedene Datentypen innerhalb einer Struktur und auch, wie ein Feld mit `struct`-Objekten verwendet wird. Die Anzahl der Elemente des Feldes kann berechnet werden, wenn sich `sizeof` in demselben Sichtbarkeitbereich *(scope)* befindet.

Beispiel

```
#include<stdio.h>

struct Person
{
   int alter;
   char name[50];
};


int main()
{
   struct Person p0 = {27, "Sebastian"};
   struct Person p1 = {20, "Annemarie"};
   struct Person p2 = {35, "Elvira"};
   // Compiler zählt die Anzahl
   struct Person mitarbeiter[] = {p0, p1, p2};
   // selbst berechnen:
   // Bytes für das Feld
```

```
    const unsigned ANZAHL = sizeof(mitarbeiter)
                      / sizeof(mitarbeiter[0]);
    // Bytes für ein Feld-Element
    unsigned i = 0;
    for( ; i < ANZAHL; ++i)
    {
       printf("Mitarbeiter %i: %s,  Alter: %i\n", i,
          mitarbeiter[i].name, mitarbeiter[i].alter);
    }
    return 0;
}
```

Führen Sie die Programme auf Ihrem Computersystem aus.

9.9 Dynamische Daten **

Der Platzbedarf dynamischer Daten ist beim Compilieren nicht unbedingt bekannt. Er kann auch erst zur Laufzeit ermittelt werden. Die C-Funktion `malloc()` reserviert den Speicherplatz. Sie ist vergleichbar mit Javas `new`. In C muss man den Speicherplatz nach Gebrauch mit `free()` freigeben, während dies in Java automatisch erledigt wird *(garbage collector)*.

Bisher wurden nur Datentypen behandelt, deren Speicherplatzbedarf bereits zur Compilierzeit berechnet und damit vom Compiler eingeplant werden konnte. Nicht immer ist es jedoch möglich, den Speicherplatz exakt vorherzuplanen, und es ist unökonomisch, jedesmal mit großen Feldern sicherheitshalber den maximalen Speicherplatz zu reservieren.

`malloc()`, `free()`

C bietet daher die Möglichkeit, mit der Funktion `malloc()` Speicherplatz auf der sogenannten Halde *(heap)* – einem speziellen Speicherbereich im Arbeitsspeicher – in der richtigen Menge und zum richtigen Zeitpunkt bereitzustellen und diesen Speicherplatz mit `free()` wieder freizugeben, wenn er nicht mehr benötigt wird. C hat keine automatische Speicherbereinigung wie Java. Die Freigabe sollte man daher nie vergessen! Ein Programm, das zyklisch Speicher benötigt und reserviert, wird ohne Freigabe nicht mehr benötigten Speichers plötzlich wegen Speichermangel stehen bleiben.

Beispiel

Um über einen Zeiger auf Elemente einer Struktur zugreifen zu könnne, gibt es in C einen besonderen Operator, den Zeigeroperator ->. Das Beispiel zeigt die Speicherbeschaffung und -freigabe für ein Punkt-Objekt, also eine Variable des `struct`-Typs Punkt.

```
#include<stdio.h>
#include<stdlib.h>      // malloc(), free()
```

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

int main()
{
   // malloc() benötigt die Anzahl der zu reservierenden Bytes
   struct Punkt *pz = malloc(sizeof(struct Punkt));
   pz->x = 0;    // Zugriff über Zeigeroperator ->
   // Alternative: Zugriff über Dereferenzierung und .-Operator
   (*pz).y = 1000;
   printf("Punkt (%i, %i)\n", pz->x, pz->y);
   free(pz);           // Freigabe nach Benutzung
   return 0;
}
```

Dynamische Felder

Große Felder sollten auf der Halde angelegt werden. Dort ist in der Regel mehr Platz als auf dem Stapel-Speicher *(stack)*. Bei dem Stapel-Speicher handelt es sich ebenfalls um einen speziellen Arbeitsspeicherbereich. Die Anzahl der Feldelemente muss zur Übersetzungszeit noch nicht bekannt sein.

Beispiel

```
unsigned n;
// hier n berechnen oder einlesen
...
// Feld mit n double-Werten anlegen
double *pfeld = malloc(n * sizeof(double));
// Zugriffsarten wie bei einem statisch angelegten Feld:
pfeld[1] = 3.1415926;
printf("Wert %lf\n", *(pfeld+1));
free(pfeld);  // Freigabe nach Benutzung
```

9.10 Modularität *

Die Modularität eines C-Programms wird erreicht, indem die verschiedenen Module Dateien zugeordnet werden. Unter einem Modul wird eine abgeschlossene funktionale Einheit verstanden, die als C-Funktion abgebildet wird. Ein wesentliches Prinzip bei der Aufteilung auf Dateien ist die Trennung von Schnittstelle und Implementierung.

In C wird in der Regel die Schnittstelle einer Funktion und die Implementierung einer Funktion textuell getrennt in unterschiedlichen Dateien gespeichert. Um die Anzahl der Dateien zu beschränken, werden oft mehrere zusammengehörige Funktionen einer Datei zugeordnet. So sind beispielsweise sowohl `printf()` wie auch `scanf()` der Datei `stdio.h` zugeordnet.

Trennung von Schnittstelle und Implementierung

Es gibt einen großen Unterschied in der Art, wie Programme in C im Vergleich zu Java strukturiert werden. Ein wesentliches Prinzip in C ist die Trennung von Schnittstelle und Implementierung. Mit Schnittstelle ist nur die Information gemeint, die benötigt wird, um eine Funktion benutzen zu können (Methodensignatur).

Beispiel

Zum Beispiel ist die Schnittstelle der Funktion `fakultaet()`:

```
unsigned long long fakultaet(int n);
```

Mehr braucht man zur reinen Benutzung nicht zu wissen. Die Kenntnis der Implementierung, dem eigentlichen Programmcode, ist zu einem Aufruf nicht notwendig, auch wenn sie letztlich zur Verfügung stehen muss.

Funktionsprototyp

Die Schnittstelle einer Funktion wird auch Funktionsprototyp genannt. Sie besteht aus Rückgabetyp, Name und Parameterliste.

Beispiel

Im »Hello World«-Programm wird mit `#include<stdio.h>` eine Datei eingelesen, die (unter anderem) den Funktionsprototyp von `printf()` enthält. Der Programmcode von `printf()` liegt als C-Quellprogramm gar nicht vor, sondern nur in übersetzter Form als Bibliotheksfunktion.

```
#include<stdio.h>  // enthält Funktionsprototyp von printf()

int main(void)
{
   printf("Hello World\n");
   return 0;
}
```

Natürlich wird der ausführbare Code von `printf()` am Ende der Übersetzung dazugebunden, sonst würde das Programm nicht laufen.

+ Der Vorteil ist, dass der Compiler für die syntaktische Analyse des Aufrufs nur die notwendige Informationen zu lesen bekommt.
+ Ein weiterer Vorteil liegt darin, dass `printf()` nicht jedesmal übersetzt werden muss, wenn das `main()`-Programm übersetzt wird. Bei einem großen Programm, das aus hunderten von Dateien besteht, wird bei der Übersetzung viel Zeit gespart, weil nur die jeweils geänderten Dateien neu übersetzt werden müssen.
− Der Nachteil besteht in der Verdopplung der Anzahl der Dateien: Man braucht eine für die Schnittstelle, wie `stdio.h`, und eine andere Datei für den eigentlichen Programmcode.

Dateien wie `stdio.h` heißen Header-Dateien oder kurz *Header*, weil ihre Namen am Kopf einer Datei zu finden sind.

Header

Compilationsablauf für eine Datei

Das Symbol `#` am Anfang der `#include`-Zeile bedeutet, dass `include` eine Anweisung an den sogenannten Präprozessor ist. Es gibt verschiedene Präprozessoranweisungen. Alle beginnen mit `#`. Tatsächlich startet der Compiler mehrere Programme, ohne dass man sich darum kümmern muss: den Präprozessor, den eigentlichen Compiler und den sogenannten Linker. Der Ablauf ist (vereinfacht) wie folgt:

Präprozessor

Das C-Quellprogramm wird zunächst von dem Präprozessor bearbeitet. Der Präprozessor liest alle mit `include` spezifizierten Dateien ein, bearbeitet alle anderen Präprozessoranweisungen, und entfernt alle Kommentare. Das Ergebnis wird dem eigentlichen Compiler übergeben.

Objektcode

Der Compiler erzeugt daraus den sogenannten Objektcode. Der besteht aus dem Maschinencode und Referenzen zu externen Funktionen wie `printf()`, deren entsprechende Adresse im Speicher noch undefiniert ist. Objektcode-Dateinamen enden meistens mit `.o` oder `.obj`. Zum Beispiel erzeugt der Aufruf `gcc -c main.c` die Objektdatei `main.o`.

Binder

Der Binder *(linker)* ist ein Programm, das die noch nicht aufgelösten Referenzen mit den tatsächlich vorhandenen Funktionen verknüpft, indem die Speicheradresse der betreffenden Funktion eingetragen wird. Der Aufruf `gcc -o main.exe main.o` ruft den Binder auf, der das direkt lauffähige Programm `main.exe` erzeugt (siehe unten). Wenn zur Abkürzung nur `gcc main.c` eingegeben wird, werden zwar alle genannten Schritte ausgeführt, aber die entstandenen Objektdateien werden gelöscht.

Compilationsablauf für viele Dateien

Wie wird eine modulare Dateistruktur für viele Funktionen angelegt? Um diese Frage möglichst einfach zu beantworten, wird hier das Schema für nur zwei verschiedene Funktionen (Module) `alcocheck()` und `bmi()`, die in `main()` verwendet werden, gezeigt.

Beispiel

Die Datei mit dem Hauptprogramm heiße `diagnose.c`.

```
// diagnose.c
#include<stdio.h>       // spitze Klammern: Systemdatei
#include"bmi.h"         // Anführungszeichen: eigene Datei
#include"alcocheck.h"   // Anführungszeichen: eigene Datei

int main(void)
{
```

```
   double gewicht; // in kg
   double groesse; // in cm
   printf("Bitte Gewicht in kg eingeben:");
   scanf("%lf", &gewicht);
   printf("Bitte Körpergröße in cm eingeben:");
   scanf("%lf", &groesse);
   double bmiwert = bmi(gewicht, groesse);
   printf("Der Body-Mass-Index beträgt %lf.\n", bmiwert);
   if(bmiwert  < 16 || bmiwert > 30)
      printf("Gewichtsproblem: Gehen Sie zum Arzt!\n");
   printf("Wieviel Alkohol trinken Sie täglich (in Gramm)?\n");
   printf("(0,7 l Bier oder 1 Viertel Rotwein: etwa 25 g)\n");
   double alk;
   scanf("%lf", &alk);
   switch(alcocheck(gewicht, alk))
   {
   case ZUVIEL :
      printf("Sie trinken deutlich zuviel Alkohol!\n");
      break;
   case RISIKO :
      printf("Sie sollten etwas weniger Alkohol trinken!\n");
      break;
   case OK :
      printf("Ihr Alkoholkonsum ist unbedenklich.\n");
      break;
   default :
      printf("Interner Programmfehler!\n");
   }
   return 0;
}
```

In den Header-Dateien sind nur die Schnittstellen enthalten. Die Datei `bmi.h` zeigt die typische Struktur einer Header-Datei:

```
// bmi.h
#ifndef BMI_H
#define BMI_H

// berechnet den Body-Mass-Index. Gewicht in kg, Groesse in cm
double bmi(double Gewicht, double Groesse);

#endif
```

Makro

Am Anfang stehen sogenannte Makros zur Steuerung. In diesem Fall soll verhindert werden, dass die Datei mehrfach gelesen wird. Im Klartext: Falls `BMI_H` nicht definiert ist (`ifndef` = if not defined), dann definiere `BMI_H` und lies die Datei bis zum schließenden `endif`. Wenn anschließend, zum Beispiel durch verschachtelte `include`-Anweisungen (`include`-Dateien, die selbst Dateien inkludieren), die Datei noch einmal gelesen werden sollte, ist `BMI_H` bereits definiert und alles bis `endif` wird übersprungen.

Die Implementierungsdatei liest typischerweise die Header-Datei, danach folgt der zugehörige Programmcode:

```
// bmi.c
#include"bmi.h"

double bmi(double gewicht, double gr)
{
   double groesseInMetern = gr/100.0;
   return gewicht/(groesseInMetern*groesseInMetern);
}
```

Die Header-Datei `alcocheck.h` zeigt eine weitere Variante von Makros: die Textersetzung. In den Dateien, die `alcocheck.h` per `include` einschließen (hier `diagnose.c` und `alcocheck.c`), wird der Text `ZUVIEL` durch eine 2 ersetzt usw.

```
// alcocheck.h
#ifndef ALCOCHECK_H
#define ALCOCHECK_H

#define ZUVIEL 2
#define RISIKO 1
#define OK 0

int alcocheck(double gew, double alc);

#endif
```

```
// alcocheck.c
#include"alcocheck.h"

int alcocheck(double gew, double alc)
{
   double grammAlkProKilo = alc/gew;
   if(grammAlkProKilo > 1.0)
      return ZUVIEL;
   if(grammAlkProKilo > 0.5)
      return RISIKO;
   return OK;
}
```

Die notwendigen Befehle zur Erzeugung einer lauffähigen Binärdatei sind:

```
gcc -std=c99 -Wall -c alcocheck.c
gcc -std=c99 -Wall -c bmi.c
gcc -std=c99 -Wall -c diagnose.c
gcc -o diag.exe diagnose.o bmi.o alcocheck.o
```

In der ersten Zeile wird `alcocheck.c` übersetzt. Die Option `-c` bewirkt, dass der Binder nicht aufgerufen wird. Es entsteht die Objektdatei `alcocheck.o`. In der zweiten und dritten Zeile geschieht entsprechendes bezüglich der anderen Dateien. In der letzten Zeile ruft `gcc` den Binder auf, der die Objektdateien zu einer ausführbaren Datei zusammenbindet. Der Name `diag.exe` dieser Datei wird mit der Option `-o` spezifiziert.

Der Binder trägt die Speicheradressen der eigenen Funktionen und auch die der benötigten Systemfunktionen wie etwa `printf()` in die ausführbare Datei ein.

Aufgabe des Binders

In `diagnose.o` gibt es einen Aufruf der Funktion `bmi()`. Der Maschinencode der Funktion `bmi()` liegt aber in der Datei `bmi.o`. Entsprechendes gilt für die Funktion `alcocheck()`. Erst wenn alle Teile zu einem ganzen Programm zusammengebunden werden, ist klar, wo genau der Programmcode dieser Funktionen im Speicherbereich liegt, der für dieses Programm insgesamt vorgesehen ist. Das zu ermitteln und an der Stelle des jeweiligen Aufrufs im Maschinencode von `main()` einzutragen, ist eine Aufgabe des Binders.

Wenn sich nur in `main()` etwas ändern sollte, müssen nur die beiden letzten Schritte wiederholt werden. Moderne Entwicklungsumgebungen steuern diese Vorgänge automatisch.

Um den Ablauf zu visualisieren, wird davon ausgegangen, dass `alcocheck.c` und `bmi.c` übersetzt worden sind und damit die Dateien `alcocheck.o` und `bmi.o` vorliegen. Jetzt seien nur noch Änderungen am Hauptprogramm `main()` in `diagnose.c` erforderlich. Die Abb. 9.10-1 verdeutlicht den erneuten Ablauf zur Erstellung des ausführbaren Programms.

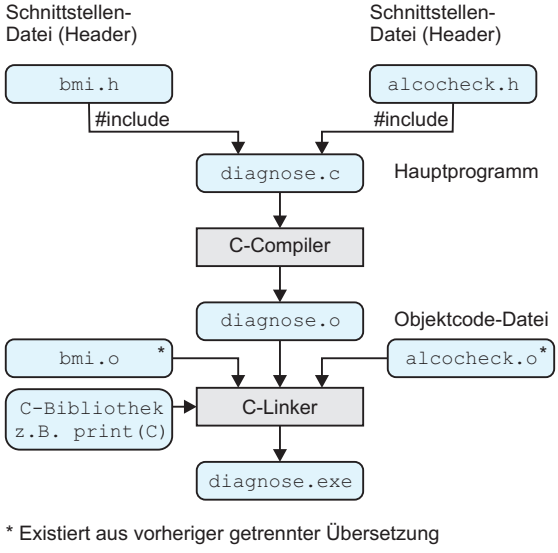


Abb. 9.10-1: Beispiel für das Übersetzen und Binden eines C-Programms.

10 Die Programmiersprache »Processing« *

Die Programmiersprache **Processing** ist eine vereinfachte Version der Programmiersprache Java. Bei Java handelt es sich eine sogenannte *General-purpose programming language* (GPL), d. h. eine Programmiersprache, die für einen breiten Anwendungszweck eingesetzt werden kann – von umfangreichen Unternehmensanwendungen bis hin zur Programmierung von mobilen Endgeräten. Im Gegensatz dazu handelt es sich bei Processing um eine *Domain-specific programming language* (DSL), d. h. eine Programmiersprache, die auf eine bestimmte Domäne bzw. auf einen bestimmten Anwendungsbereich zugeschnitten bzw. spezialisiert ist. Die Haupteinsatzgebiete von Processing sind Grafik, Animationen, Video, Sound, Typografie, 3D, Simulation. Sie richtet sich vorwiegend an Gestalter, Künstler und Programmieranfänger.

Zur Historie

Die Idee zu Processing hatten Ben Fry und Casey Reas. 2005 wurde Processing mit dem Prix Ars Electronica in der Kategorie Net Vision/Net Excellence ausgezeichnet.

Programmierumgebung

Die Programmierumgebung von Processing können Sie hier herunterladen: Website Processing Download (`https://processing.org/de/download/`).

Installieren Sie die Programmierumgebung auf Ihrem Computersystem.

Infos

Umfangreiche Informationen und Beispiele finden Sie hier:

- Website Processing (`http://www.processing.org/`)
- Website OpenProcessing (`http://www.openprocessing.org/`)

Die Sprachreferenz finden Sie hier:

- Sprachreferenz Processing (`http://www.processing.org/reference/`)

Der Einstieg in Processing erfolgt mit zweidimensionalen Grafikprogrammen:

- »Einstieg in die zweidimensionale Grafikprogrammierung«, S. 350

Auf einfache Art und Weise können Animationen programmiert werden:

- »Animationen«, S. 355

Wird die Maus oder die Tastatur betätigt, dann kann darauf reagiert werden:

- »Maus- und Tastaturereignisse«, S. 359

10.1 Einstieg in die zweidimensionale Grafikprogrammierung *

Processing-Programme haben dieselbe Syntax wie Java, werden in den Java-Byte-Code übersetzt und in der JVM ausgeführt. Die Ausgabe von Processing-Programmen erfolgt in der Regel in eine Grafikfläche. Standardfunktionen ermöglichen das Zeichnen von zweidimensionalen Grafiken. Durch Attributfunktionen können die Grafikdarstellungen beeinflusst werden. Beliebige Farbeinstellungen sind möglich.

Die Entwicklungsumgebung

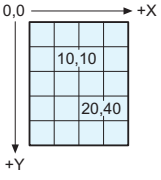
Bevor Sie mit dem ersten Processing-Programm beginnen können, müssen Sie die Entwicklungsumgebung PDE *(Processing Development Environment)* von der Website Processing-Download (`http://processing.org/download/`) herunterladen. Nach dem Entpacken der heruntergeladenen Datei kann die Entwicklungsumgebung mit `processing.exe` gestartet werden.

PDE

Die Abb. 10.1-1 zeigt Ihnen den Grundzustand der Entwicklungsumgebung, wenn Sie links oben den ersten Druckknopf mit dem Dreieck betätigt haben.

Grafikfläche

Im Gegensatz zur Konsolen-Ausgabe in Java finden in der Regel alle Ausgaben in Processing auf einer Grafikfläche statt (Anzeigefenster). Die Grafikfläche besteht aus **Pixeln**, wobei der Nullpunkt links oben liegt. Die X-Achse läuft von links nach rechts, die Y-Achse von oben nach unten (siehe Marginalie).



Mit den Anweisungen `print()` und `println()` können Informationen in der Konsole ausgegeben werden.

Das erste Processing-Programm

Processing-Programme besitzen dieselbe Syntax wie Java-Programme. Sie werden wie Java-Programme in den **Java-Bytecode** übersetzt und von der **JVM** ausgeführt.

Beispiel: `Erstes Programm`

Das erste Programm in Processing zeigt einige grundlegende Grafikoperationen:

```
//Größe der Zeichenfläche
size(300, 300);

//Gitterraster
for(int x=10; x<300; x=x+10)
 for(int y=10; y<300; y=y+10)
   point (x, y);

//Rechteck als Quadrat
```

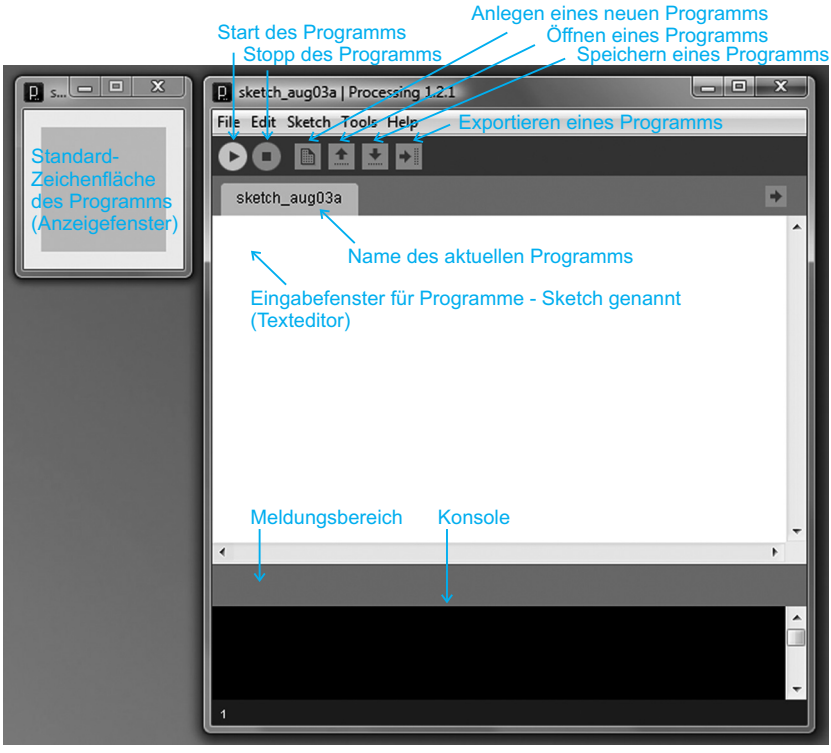


Abb. 10.1-1: Entwicklungsumgebung der Programmiersprache Processing.

```
noFill(); //Rechteck ist transparent
rect (10,10,280,280); //Rechteck

//Linien für ein W
smooth(); //Mit Anti-Aliasing (Glättung der Treppenstufen)
stroke(255,0,0);//Farbe der folgenden Grafik, hier: rot
strokeWeight(5);//Linienbreite, hier: 5 Pixel
line(30,30,70,270);//Linie von x1, y1 nach x2, y2
line(70,270,150,150);
line(150,150,230,270);
line(230,270,270,30);

//Halbkreise für eine 3
stroke(0,255,0); //Umstellung der Farbe auf grün
arc(150,95,110,110,-PI/1.5,PI/2); //Kreisbogen
arc(150,205,110,110,-PI/2,PI/1.5);

//Linien für ein L
stroke(0,0,255); //blau
line(80,20,80,280);
line(80,280,220,280);
```

Die Abb. 10.1-2 zeigt das Ergebnis. Um die Programmierung zu erleichtern wird zunächst ein Gitterraster aus Punkten erzeugt. Anschließend wird ein transparentes Rechteck (Rechteck ist nicht gefüllt) in der Größe von 280 x 280 Pixeln erzeugt (ergibt ein Quadrat). Die linke obere Ecke hat die Koordinaten x=10 und y=10. Standardmäßig wird das Rechteck durch eine 1 Pixel starke schwarze Linie dargestellt. Um bei schrägen Linien zu vermeiden, dass Treppenstufen zu sehen sind, wird mit der Funktion `smooth()` ein so genanntes Anti-Aliasing eingestellt. Die Funktion `stroke()` ermöglicht es, die Farben für die folgenden Grafiken einzustellen. Mit der Funktion `strokeWeight()` kann die Linienbreite in Pixeln festgelegt werden.

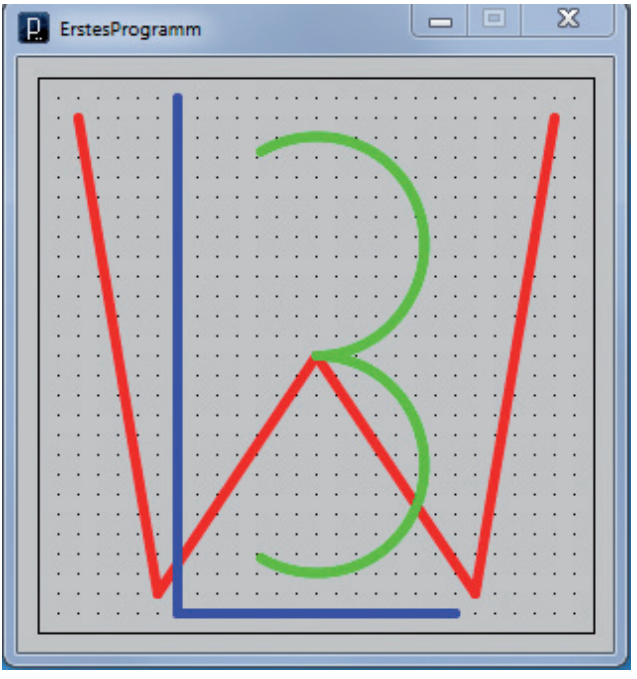


Abb. 10.1-2: Grundlegende Grafikoperationen in Processing.

Die wichtigsten Funktionen sehen wie folgt aus:

Shape
2D Primitives

- `point(x, y)`: Zeichnet einen 1 Pixel-Punkt.
- `line(x1, y1, x2, y2)`: Zeichnet eine Linie zwischen zwei Punkten.
- `rect(x, y, width, height)`: Zeichnet ein Rechteck; x, y = linke obere Ecke.
- `ellipse(x, y, width, height)`: Zeichnet eine Ellipse; x, y = Mittelpunkt; wenn `width=height`, dann wird ein Kreis gezeichnet.

- `arc(x, y, width, height, start, stop)`: Zeichnet einen Kreisbogen; Ausschnitt aus einer Ellipse.
- `quad(x1, y1, x2, y2, x3, y3, x4, y4)`: Zeichnet ein vierseitiges Polygon.
- `triangle(x1, y1, x2, y2, x3, y3)`: Zeichnet ein Dreieck.

Shape Attibutes

- `smooth()`: Linien werden geglättet (Anti-Aliasing).
- `noSmooth()`: Linien werden nicht geglättet (Aliasing).
- `strokeWeight(width)`: Linienbreite in Pixel (Voreinstellung: 1)
- `strokeCap(MODE)`: Form der Linienenden, `MODE=ROUND` abgerundet (Voreinstellung), `MODE=SQUARE` quadratisch, `MODE=PROJECT` quadratisch erweitert.
- `strokeJoin(MODE)`: Legt fest, wie Linienenden miteinander verbunden werden, `MODE=MITER` auf Gehrung verbunden (Voreinstellung), `MODE=BEVEL` abgeschrägte Kante, `MODE=ROUND` abgerundete Kante.

Color Setting

- `stroke(value1, value2, value3)`: Legt die Farbe für Linien und Umrisse fest. Bildschirme stellen Farben durch unterschiedliche Mischungen von **r**ot, **g**rün und **b**lau dar (RGB-Farbmodell). Die Intensität einer Farbe wird durch Werte von 0 bis 255 definiert. `stroke(255,0,0)` ergibt reines Rot. In Processing gibt auch noch andere Möglichkeiten für Farbangaben, z. B. in Hexadezimalform.
- `noStroke()`: Umrisslinien von Rechtecken usw. werden nicht gezeichnet.
- `fill(value1, value2, value3, alpha)`: Füllfarbe von Rechtecken usw. `alpha` ist optional und gibt die Licht*un*durchlässigkeit der Füllung an.
- `noFill()`: Keine Füllfarbe von Rechtecken usw.
- `background(value1, value2, value3)`: Hintergrundfarbe der Processing-Zeichenfläche.

Structure

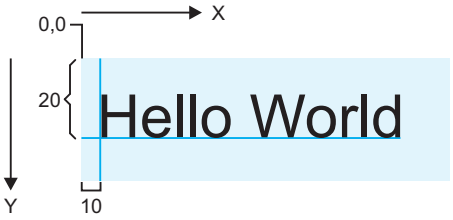
- `size(width, height)`: Legt die Größe der Processing-Zeichenfläche fest.

Environment

- `height`: Systemvariable, die die Höhe des Anzeigefensters enthält (wird durch die Funktion `size(width, height)` gesetzt).
- `width`: Systemvariable, die die Breite des Anzeigefensters enthält (wird durch die Funktion `size(width, height)` gesetzt).

Typography

- `text(data, x, y)`: Die Information im Parameter `data` wird ausgegeben. Die Positionierung im Grafik-Koordinatensystem zeigt die Abb. 10.1-3. Die Grundlinie *(baseline)* der auszugebenden Informationen befindet sich auf der mit y übergebenen y-Position. Die Information wird ab der x-Position von links nach rechts gezeichnet. Außerdem gibt es noch weitere Möglichkeiten.



text(„Zeichenkette“, x, y);
hier: text("Hello World",10,20);

Abb. 10.1-3: Positionierung von Text im Grafik-Koordinatensystem.

Eine Hommage an Josef Albers

Der deutsche Maler Josef Albers (geboren am 19. März 1888 in Bottrop; gestorben am 25. März 1976 in New Haven, Connecticut) gehört mit zu den Gründern der Opt-Art. Ein Bild aus seiner Serie »Hommage to the Square« wurde sogar auf einer deutschen Briefmarke verewigt (Abb. 10.1-4).

Abb. 10.1-4: Briefmarke mit einem Bild aus der Serie »Hommage to the Square« von Josef Albers (Quelle: Wikipedia, Herausgeber: Deutsche Bundespost, 1993).

Diese Grafik lässt sich als Processing-Programm »nachbauen«.

Tipp

Die Farben in einer Grafik oder in einem Foto lassen sich mit einem Grafikprogramm leicht bestimmen. In der Regel enthalten Grafikprogramme so genannte Pipetten. Positioniert man eine solche Pipette auf einer Farbe, dann werden die Farbwerte angezeigt.

Beispiel: HommageAn JosefAlbers

Das Programm zum Nachbau der Grafik sieht wie folgt aus:

```
//Größe der Zeichenfläche
size(300, 300);

//Keine Umrandung der Quadrate
noStroke();

fill(232,213,93); //Farbe 1. Quadrat
rect(0,0,300,300); //1. Quadrat

fill(227,164,67); //Farbe 2. Quadrat
rect(28,45,243,243); //2. Quadrat

fill(228,135,58); //Farbe 3. Quadrat
```

```
rect(56,90,182,182); //3. Quadrat

fill(215,105,42); //Farbe 4. Quadrat
rect(84,135,122,122); //4. Quadrat

//Textausgabe in Grafikbereich
text("Hommage an Josef Albers", 10,15);

//Konsolenausgabe
println("Hommage an Josef Albers");
```

Die Ausgabe zeigt die Abb. 10.1-5.

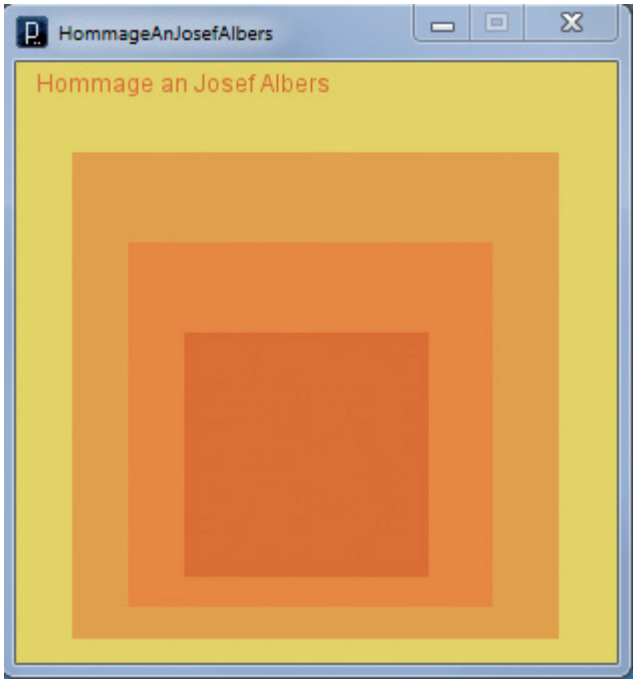


Abb. 10.1-5: Programmierung der Grafik »Hommage to the Square« von Josef Albers.

10.2 Animationen *

Die Funktion `setup()` ermöglicht es, initiale Einstellungen vorzunehmen. Die Funktion `draw()` führt den in ihr enthaltenen Code wie in einer Endlosschleife aus. Mithilfe der Funktion `frameRate()` wird festgelegt, wie oft der Bildschirminhalt neu angezeigt wird. Die Funktion `delay()` ermöglicht es, den Programmablauf um eine angegebene Anzahl von Millisekunden anzuhalten. Mit `noLoop()` und `loop()` kann der Programmablauf gestoppt und wieder gestartet werden.

Structure `setup()`

In Processing gibt es die Systemfunktion `setup()`, die genau *einmal* automatisch aufgerufen wird, wenn das Programm gestartet wird. In ihr werden alle Anweisungen angegeben, die dazu dienen, die initiale Umgebung für das Programm festzulegen, wie Bildschirmgröße, Hintergrundfarbe usw. Variablen, die in dieser Funktion deklariert werden, sind außerhalb der Funktion nicht sichtbar. Jedes Programm kann nur eine `setup()`-Funktion enthalten.

Structure `draw()`

Direkt nach der Ausführung der Funktion `setup()` wird die Systemfunktion `draw()` automatisch aufgerufen. Alle Anweisungen dieser Funktion werden wie in einer Endlosschleife kontinuierlich ausgeführt, bis das Programm gestoppt wird.

Environment `frameRate()`

Wie oft der Bildschirminhalt pro Sekunde neu angezeigt wird, wird durch die Funktion `frameRate(fps)` angegeben. Die Voreinstellung ist 60 mal pro Sekunde. Diese Funktion sollte in der Systemfunktion `setup()` aufgerufen werden.

Structure `delay()`

Die Funktion `delay(milliseconds)` sorgt dafür, dass das laufende Programm für die spezifizierte Zeit gestoppt wird. Die Funktion hat keine Wirkung innerhalb von `setup()`.

Structure `noLoop()` `loop()`

Die Funktion `noLoop()` stoppt die kontinuierliche Ausführung des Codes innerhalb von `draw()`. Ein anschließender Aufruf von `loop()` führte zu einer Fortsetzung des Ablaufs. Steht `noLoop()` in der Funktion `setup()`, dann sollte sie dort als letzte Anweisung stehen.

Math `random()`

Die Funktionen `random(high)` und `random(low, high)` liefern bei jedem Aufruf eine Zufallszahl zwischen `0` und `< high` bzw. zwischen `low` und `< high`. Die Parameter können vom Typ `int` oder `float` sein. Das Ergebnis ist vom Typ `float`. Die verschiedenen Funktionen werden in dem folgendem Programm demonstriert.

Beispiel Pulsierender Kreis

```
 1  void setup()
 2  {
 3    //Größe der Zeichenfläche
 4    size(300, 300);
 5
 6    //Gitterraster
 7    /*for(int x=10; x<300; x=x+10)
 8     for(int y=10; y<300; y=y+10)
 9       point (x, y);*/
10    frameRate(20);
11    smooth();
12    //noLoop();
13
14  }
15  int xy = 290;
16  boolean kleiner = true;
17  int anzahldurchlaeufe = 0;
18  void draw()
```

```
19  {
20    anzahldurchlaeufe ++;
21    //Kreis
22    noFill(); //Kreis ist transparent
23
24    ellipse(150, 150, xy, xy);
25    if (xy > 0 && kleiner) xy = xy - 10;
26    if (xy == 0)
27    {
28      kleiner = false;
29      stroke(int(random(256)),int(random(256)),
30         int(random(256)));
31      //fill(int(random(256)),int(random(256)),
32      //  int(random(256)));
33      delay(2000);
34    }
35    if (xy < 290 && !kleiner) xy = xy + 10;
36    if (xy == 290)
37    {
38      kleiner = true;
39      stroke(int(random(256)),int(random(256)),
40         int(random(256)));
41      //fill(int(random(256)),int(random(256)),
42      //  int(random(256)));
43      delay(2000);
44    }
45
46    if (anzahldurchlaeufe == 100)
47        {
48          noLoop();
49          delay(5000);
50          loop();
51        }
52  }
```

Einen Ausschnitt aus dem laufenden Programm zeigt die Abb. 10.2-1.

- Kennzeichnen Sie die Anweisung 22 als Kommentar und entfernen Sie aus den Anweisungen 30 und 38 die Kommentarzeichen. Was ändert sich am Programmablauf?
- Variieren Sie die Parameter der verschiedenen Funktionen und sehen Sie sich die Effekte an.

Die Abb. 10.2-2 zeigt ein animiertes Smiley-Programm.

Beispiel

Das folgende Programm erzeugt einen springenden Ball (in Anlehnung an [Wart09, S. 155]):

Beispiel Animierter Ball

```
int x = 0, y = 0, radius = 0;
float daempfung = 0.0;
float gravitation = 0.0;
float reibung = 0.0;
float geschwX = 0.0, geschwY = 0.0;
```

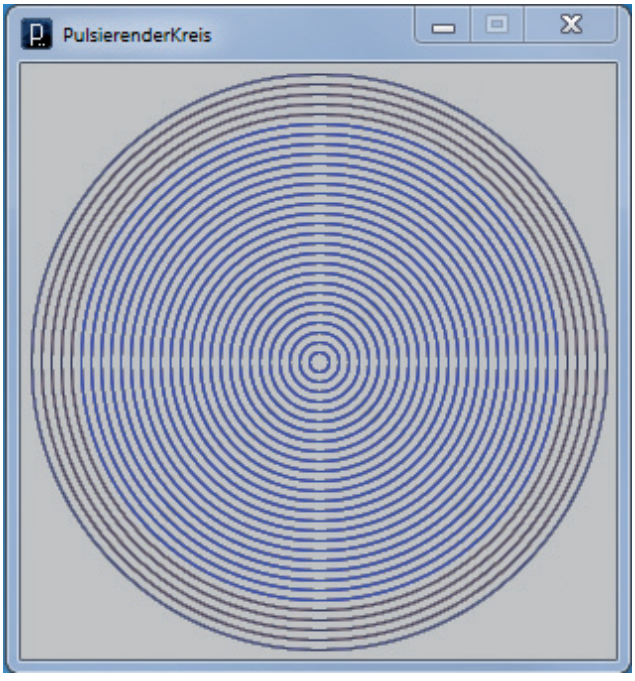


Abb. 10.2-1: Ausschnitt aus dem Programmlauf »Pulsierender Kreis«.

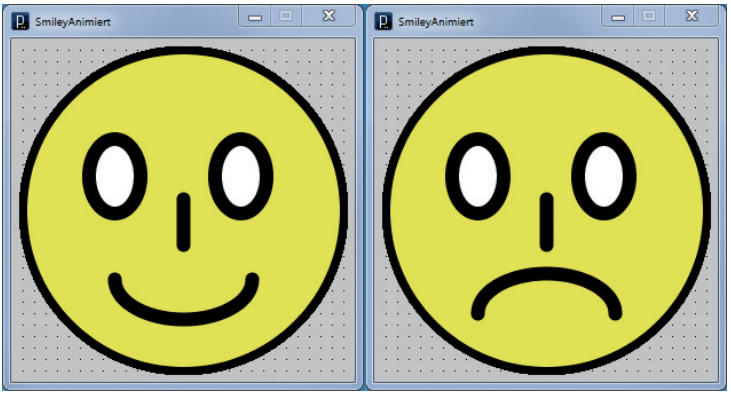


Abb. 10.2-2: Ausschnitt aus einem animierten Smiley-Programm.

```
void setup()
{
  size(400,400);
  x = width / 2;
  radius = 50;
  fill(0);
  geschwX = 4;
```

```
  gravitation = 0.5;
  daempfung = 0.8;
  reibung = 0.9;
  smooth();
}

void draw()
{
  //Bildschirm aufräumen
  fill(0,0,0,255);
  rect(0,0,width,height);
  fill(0,0,255,255);
  ellipse(x,y,radius,radius);
  x = x + int(geschwX);
  geschwY = geschwY + gravitation;
  y = y + int(geschwY);
  //Bildschirmränder prüfen
  if (x > width - (radius/2))
  {
    x = width - (radius / 2);
    geschwX = -geschwX;
  }
  else if (y > height - (radius / 2))
  {
    y = height - (radius /2);
    geschwY = -geschwY;
    geschwY = geschwY * daempfung;
    geschwX = geschwX * reibung;
  }
  else if (y < 0)
  {
    y = 0;
    geschwY = -geschwY;
  }
}
```

Führen Sie das Programm auf Ihrem Computersystem aus und sehen Sie sich die Animation an.

Hinweis

Verwenden Sie in einem Programm die Funktionen `setup()` und `draw()`, dann spricht man von einem kontinuierlichen Modus *(continuous mode),* sonst von einem Basismodus *(basic mode).*

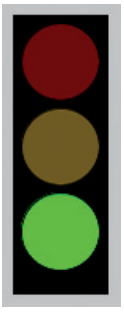
10.3 Maus- und Tastaturereignisse *

Maus- und Tastaturereignisse können durch Funktionen und Systemvariablen abgefragt werden. Folgende Funktionen stehen zur Verfügung: `mouseClicked()`, `mouseDragged()`, `mouseMoved()`, `mousePressed()`, `mouseReleased()`, `keyPressed()`, `keyReleased()` und `keyTyped()`. Folgende Systemvariablen können abgefragt werden: `mouseButton`, `mousePressed`, `mouseX`, `mouseY`, `pmouseX`, `pmouseY`, `key`, `keyCode` und `keyPressed`.

In Processing können verschiedene Maus- und Tastaturereignisse im Programm abgefragt werden.

Beispiel Ampel

Im folgenden Programm wird die Funktion `mouseClicked()` immer dann aufgerufen, wenn die Maus auf der Zeichenfläche gedrückt und wieder losgelassen wird:



```
void setup()
{
  //Größe der Zeichenfläche
  size(300, 300);

  //Gitterraster
  for(int x=10; x<300; x=x+10)
   for(int y=10; y<300; y=y+10)
     point (x, y);
  frameRate(20);
  smooth();
}
int rot1, rot2, rot3;
int gelb1, gelb2,gelb3;
int gruen1, gruen2, gruen3;
int zustand = 0;

void draw()
{
  fill(0);
  rect(120,70,60,180);
  fill(rot1,rot2, rot3);
  ellipse(150,100,50,50);
  fill(gelb1,gelb2,gelb3); //grün
  ellipse(150,155,50,50);
  fill(gruen1,gruen2,gruen3); //grün
  ellipse(150,210,50,50);
  switch (zustand)
  {
   case 0: //Rotphase
    rot1 = 255; rot2 = 0; rot3 = 0;
    gelb1 = 118; gelb2 = 92; gelb3 = 6;
    gruen1 = 1; gruen2 = 93; gruen3 = 0;
    break;

   case 1: //Gelbphase
    rot1 = 116; rot2 = 1; rot3 = 5;
    gelb1 = 255; gelb2 = 255; gelb3 = 0;
    gruen1 = 1; gruen2 = 93; gruen3 = 0;
    break;

   case 2: //Grünphase
     rot1 = 116; rot2 = 1; rot3 = 5;
     gelb1 = 118; gelb2 = 92; gelb3 = 6;
     gruen1 = 0; gruen2 = 255; gruen3 = 0;
     break;
  }
}
```

```
void mouseClicked()
{
  zustand = (zustand + 1) % 3;
}
```

Die Marginalie zeigt die Grünphase der Ampel.

Ändern Sie das Programm `Ampel` bitte so, dass vor der Grünphase eine Gelbgrünphase kommt.

Mouse

Folgende Ereignisse können durch Funktionen abgefragt werden:

- `mouseClicked()`: Wird einmal aufgerufen nachdem ein Mausknopf gedrückt und wieder losgelassen wurde.
- `mouseDragged()`: Wird jedes Mal aufgerufen, wenn die Maus bewegt und der Mausknopf gleichzeitig gedrückt ist.
- `mouseMoved()`: Wird jedes Mal aufgerufen, wenn die Maus bewegt und der Mausknopf *nicht* gedrückt ist.
- `mouseReleased()`: Wird jedes Mal aufgerufen, wenn der Mausknopf losgelassen wird.

Folgende Ereignisse können durch Systemvariablen abgefragt werden:

- `mouseX`: Enthält die aktuelle horizontale Koordinate der Maus.
- `mouseY`: Enthält die aktuelle vertikale Koordinate der Maus.
- `pmouseX`: Enthält die horizontale Position der Maus im letzten Rahmen *(frame)*.
- `pmouseY`: Enthält die vertikalen Position der Maus im letzten Rahmen *(frame)*.

Verwenden Sie `pmouseX` und `pmouseY` innerhalb von `draw()`, wenn sie Werte relativ zum vorherigen Rahmen *(frame)* haben wollen. Verwenden Sie `pmouseX` und `pmouseY` innerhalb der Maus-Funktionen, wenn Sie eine kontinuierliche Antwort erwarten.

- `mouseButton`: In Abhängigkeit davon, welcher Mausknopf gedrückt ist, hat diese Variable entweder den Wert `LEFT`, `RIGHT` oder `CENTER`.
- `mousePressed`: Der Wert dieser Variablen ist `true`, wenn der Mausknopf gedrückt ist, sonst `false`.

Beispiel Zeichenbrett

Auf ein Zeichenbrett soll immer dann mit Linien gezeichnet werden, wenn die Maus gedrückt ist. Durch einen Klick auf das Plus- oder Minusquadrat kann die Strichstärke umgestellt werden:

```
1  void setup()
2  {
3    //Größe der Zeichenfläche
4    size(300, 300);
5    background(0,0,0); //schwarz
6
```

```
 7    //Gitterraster
 8    for(int x=10; x<300; x=x+10)
 9     for(int y=10; y<300; y=y+10)
10     {
11       stroke(200,200,80);//dunkelgelb
12       point (x, y);
13     }
14    frameRate(20);
15    smooth();
16    noFill();
17    rect(200,250,40,40);
18    strokeWeight(3);
19    line(215,270,225,270);//Pluszeichen
20    line(220,265,220,275);
21    strokeWeight(1);
22    rect(250,250,40,40);
23    strokeWeight(3);
24    line(265,270,275,270); //Minuszeichen
25  }
26
27  int linienbreite = 1;
28  void draw()
29  {
30
31    //lights();
32    stroke(255); //weiß
33    strokeWeight(linienbreite);
34
35    if(mousePressed)
36    {
37      line(mouseX, mouseY, pmouseX, pmouseY);
38      //ellipse(mouseX, mouseY, 1, 1);
39    }
40  }
41  void mouseClicked()
42  {
43    if (mouseX > 200 && mouseX < 240 && mouseY > 250
44        && mouseY < 290)
45    {
46       linienbreite++;
47    }
48    if (mouseX > 250 && mouseX < 290 && mouseY > 250
49        && mouseY < 290)
50    {
51       linienbreite--;
52    }
53  }
```

Die Abb. 10.3-1 zeigt ein Beispiel.

Kommentieren Sie die Zeile 37 aus und entfernen Sie in der Zeile 38 den Kommentar. Was stellen Sie fest?

Erweitern Sie das Programm so, dass auch Farben umgestellt werden können.

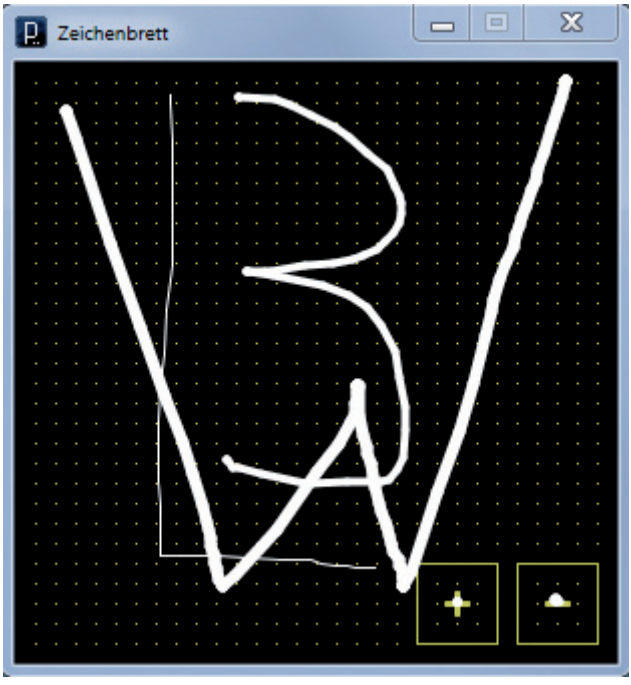


Abb. 10.3-1: Ausschnitt aus dem Programm »Zeichenbrett«.

Neben den Mausereignissen können auch Tastaturereignisse durch Funktionen abgefragt werden:

Keyboard

- `keyPressed()`: Die Funktion wird jedes Mal aufgerufen, wenn eine Taste gedrückt wird. Die Taste, die gedrückt wurde, wird in der Systemvariablen `key` gespeichert. `keyPressed()` wird immer auch aufgerufen, wenn `keyTyped()` aufgerufen wird (siehe unten), aber nicht umgekehrt.
- `keyReleased()`: Die Funktion wird jedes Mal aufgerufen, wenn eine Taste losgelassen wird. Die Taste, die losgelassen wurde, wird in der Systemvariablen `key` gespeichert.
- `keyTyped()`: Die Funktion wird jedes Mal aufgerufen, wenn eine Taste gedrückt wird. Aktionstasten wie `Ctrl`, `Shift` und `Alt` werden ignoriert, d. h. `keyTyped()` wird beim Drücken von Sondertasten *nicht* aufgerufen. Wird eine Taste lange gedrückt, dann werden mehrere Aufrufe ausgeführt.

Hinweis

Ein Aufruf der Funktionen `keyPressed()` und `keyTyped()` findet nicht nur beim Drücken einer Taste statt, sondern wann immer das Betriebssystem das entsprechende Ereignis auslöst. Wann das geschieht, ist abhängig von den Einstellungen des Systems. Das geschieht auch durchaus mehrmals, wenn ei-

ne Taste gedrückt gehalten wird, aber auch bei rein virtuellen Tastendrücken, wie zum Beispiel über eine Bildschirmtastatur. `keyReleased()` hingegen wird nur einmal am Ende des »Tastendrucks« ausgelöst, also beim Loslassen.

Systemvariablen enthalten folgende Informationen:

- `key`: Enthält den Wert der zuletzt gedrückten oder losgelassenen Taste. Für Nicht-ASCII-Zeichen ist die Systemvariable `keyCode` abzufragen.
- `keyCode`: Dient dazu, spezielle Tasten wie `UP`, `DOWN`, `LEFT`, `RIGHT` und `ALT`, `CONTROL`, `SHIFT` abzufragen. Vorher muss gefragt werden, ob die Systemvariable `key` einen Wert enthält, z. B. `if (key == CODED) { if (keyCode == UP) { fillVal = 255;...`
- `keyPressed`: Diese boolesche Systemvariable ist `true`, wenn irgendeine Taste gedrückt ist, sonst `false`.

Ändern Sie das Programm `Zeichenbrett` so, dass die Strichstärken und Farben über die Tastatur geändert werden.

10.4 Ausblick ***

Processing erlaubt es, Fotos und Bilder zu bearbeiten und pixelweise zu verändern. Videos können geladen, manipuliert, erzeugt und gespeichert werden. Dreidimensionale Grafiken und Animationen können erstellt werden. Processing-Programme können als lauffähige Programme für die Betriebssysteme Windows, MacOS und Linux sowie als Java-Applets für die Integration in Webseiten exportiert werden.

Mit Processing ist einfach möglich, Fotos und Videos zu manipulieren.

Bilder

Fotos und Bilder können mit einfachen Bildverarbeitungsmethoden bearbeitet, gefiltert, überblendet und auf Pixelebene verändert werden.

Beispiel Foto manipulation

Das folgende Programm manipuliert ein eingelesenes Foto mithilfe verschiedener Filterfunktionen:

```
PImage foto;
size(400, 600);
foto = loadImage("MeinFoto.jpg");
image(foto,0,0,200,300);
filter(BLUR,10);
image(foto,0,300,200,300);
filter(GRAY);
image(foto,200,0,200,300);
filter(THRESHOLD);
image(foto,200,300,200,300);
```

Die Abb. 10.4-1 zeigt die Ausgabe des Programms.

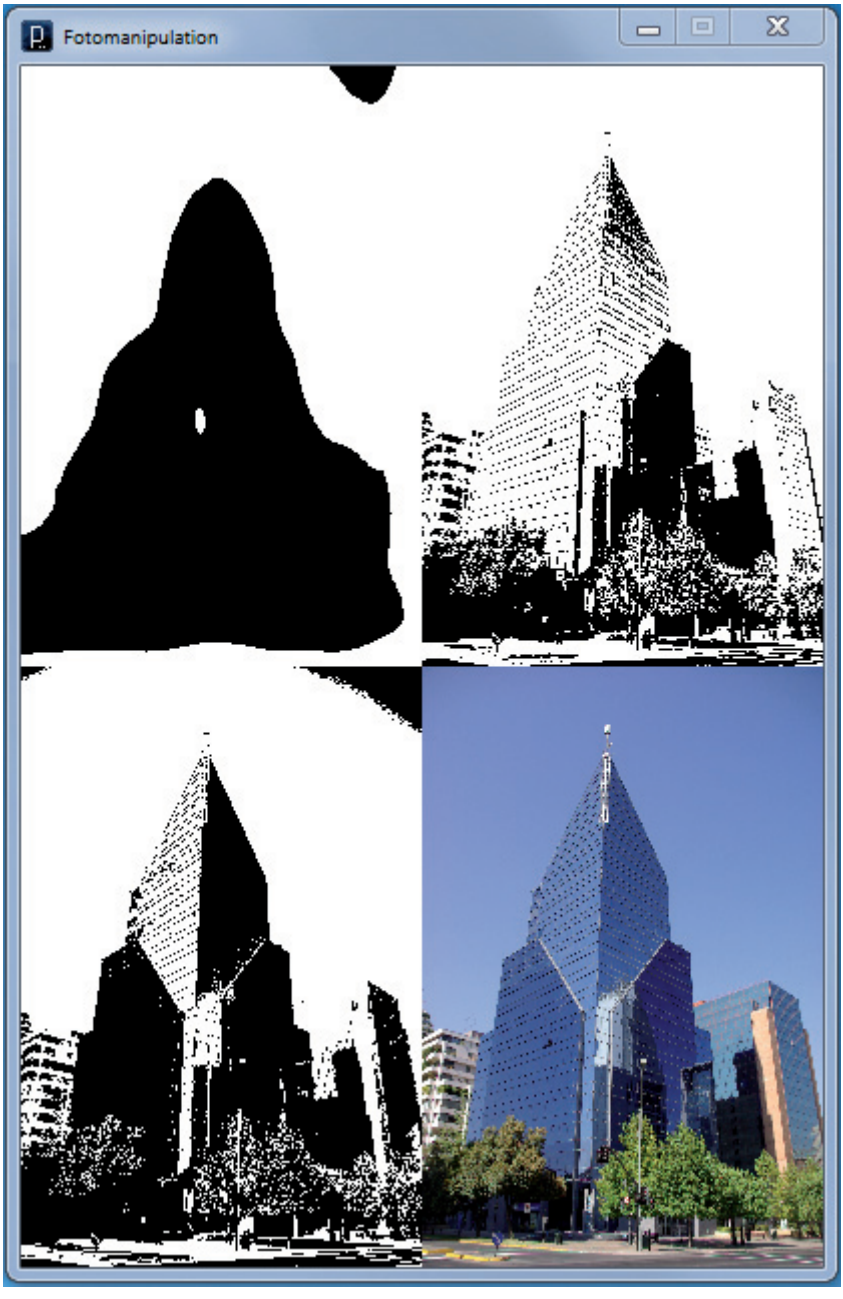


Abb. 10.4-1: Modifikation eines Fotos durch den Einsatz von Filterfunktionen.

Sehen Sie sich die verwendeten Funktionen in der Processing-Referenz in der Kategorie *Image* an und wenden Sie die Funktionen auf eigene Bilder an.

Videos

Videos im QuickTime-Format können gelesen, bearbeitet, erzeugt und gespeichert werden.

3D

Analog zu 2D-Grafiken und -Animationen können auch 3D-Grafiken und -Animationen erstellt werden.

Beispiel Wuerfel3D

Einen animierten Würfel erzeugt folgendes Programm:

```
void setup()
{
  size(300,300,P3D);
  frameRate(20);
}

void draw()
{
  background(255,204,0); //gelb
  lights();
  translate(width/2, height/2);
  rotateY(frameCount*PI/60);
  rotateX(frameCount*PI/40);
  fill(0,0,255); //blau
  strokeWeight(5);
  box(120,120,120);
}
```

Die Abb. 10.4-2 zeigt einen Ausschnitt aus der Animation.

Machen sie sich mit den Funktionen vertraut und variieren Sie die Parameter.

Export

Erstellte Processing-Programme können als lauffähige Windows-, Mac OS X- und Linux-Programme exportiert werden (Menü `File/Export Application`). Außerdem ist es möglich, sie als **Java-Applets** zu exportieren, so dass sie in Webseiten eingebunden werden können (Menü `File/Export Applet`).

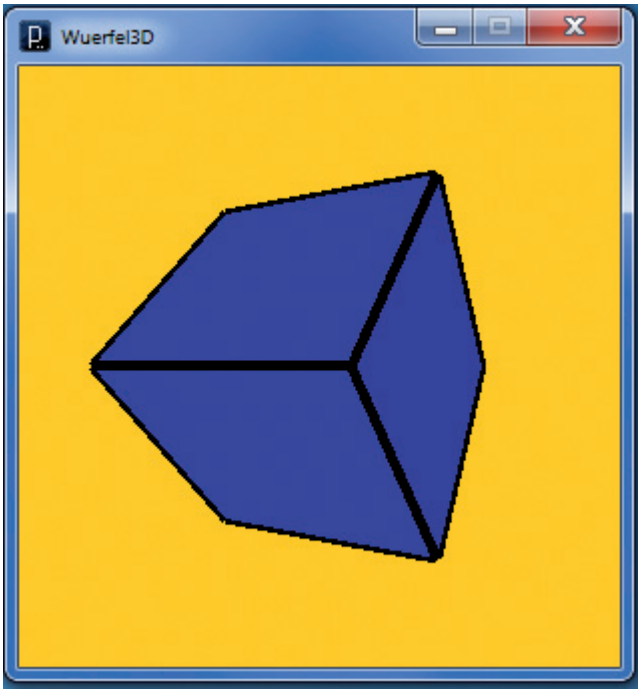


Abb. 10.4-2: Animierter Würfel.

Anhang A Kreuzworträtsel 1: Lösung **

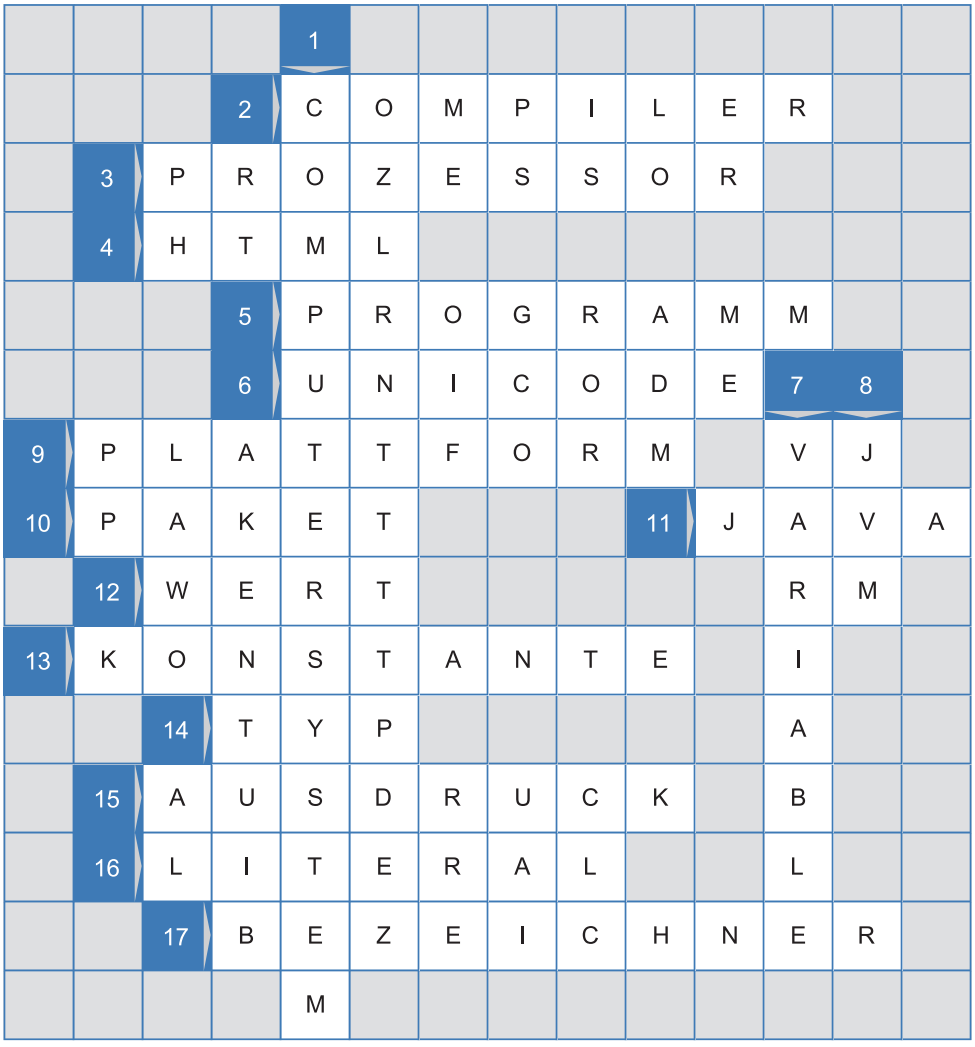


Abb. 1.0-1: Lösung des Kreuzworträtsels zu Basiskonzepten der Programmierung.

Gesuchte Wörter:

1 Hardware und Software als Einheit.
2 Sorgt dafür, dass problemorientierte Programme (Quellprogramme) in Objektprogramme transformiert werden.
3 Der Teil eines Computers, in dem die Programme Schritt für Schritt verarbeitet werden.

4 Kurzform für: hypertext markup language.
5 Handlungsvorschrift für einen Computer.
6 Genormter 16-Bit-Zeichensatz (65.469 Positionen), der die Schriftzeichen aller Verkehrssprachen der Welt aufnehmen soll.
7 Gegenteil von Konstante.
8 Interpreter, der den Java-Bytecode analysiert und ausführt (Kurzform).
9 Kombination aus Prozessortyp und verwendetem Betriebssystem.
10 Wird in der Softwaretechnik nicht durch die Post ausgeliefert.
11 Programmiersprache, die ursprünglich für die Programmierung elektronischer Konsumgeräte entwickelt wurde, ihren Siegeszug aber erst im Web antrat.
12 Inhalt einer Variablen.
13 Variable, die nach der Initialisierung nicht mehr verändert werden kann.
14 Dieses Mädchen ist nicht mein ... Bei Programmiersprachen beschwert sich der Compiler.
15 Verarbeitungsvorschrift zur Ermittlung eines Wertes.
16 Darstellung des Werts einer Variablen oder Konstanten.
17 Namen für Variable, Konstante, Typen, Funktionen, Prozeduren, Klassen, Objekte usw. in Programmiersprachen, um sie eindeutig identifizieren zu können.

Anhang B Kreuzworträtsel 2: Lösung **

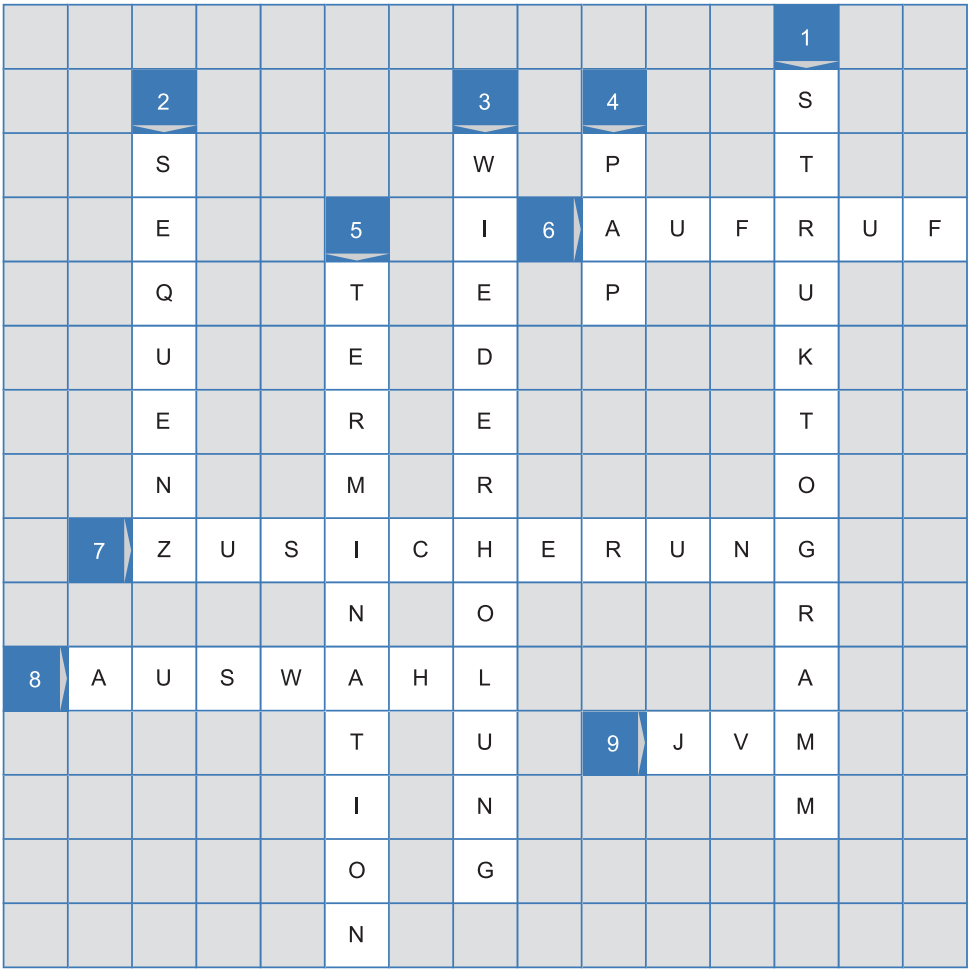


Abb. 2.0-1: Lösung des Kreuzworträtsels zu Kontrollstrukturen.

Gesuchte Wörter:

1 Grafische Darstellung linearer Kontrollstrukturen.
2 Mehrere Anweisungen werden hintereinander, von links nach rechts und von oben nach unten, ausgeführt.
3 Mehrfache Ausführung von Anweisungen in Abhängigkeit von einer Bedingung oder für eine gegebene Anzahl.
4 Eine in DIN 66001 genormte, grafische Darstellungsform für Kontrollstrukturen von Algorithmen (Kurzform).
5 Programm, das nach endlicher Zeit beendet ist (Substantiv).

6 Wechsel der Kontrolle von der aufrufenden Stelle zu dem aufgerufenen Algorithmus.
7 Eigenschaft oder Zustand, der an einer bestimmten Stelle eines Programms immer gilt.
8 Ausführung von Anweisungen in Abhängigkeit von Bedingungen.
9 Interpreter, der den Java-Bytecode analysiert und ausführt (Kurzform).

Anhang C Kreuzworträtsel 3: Lösung **

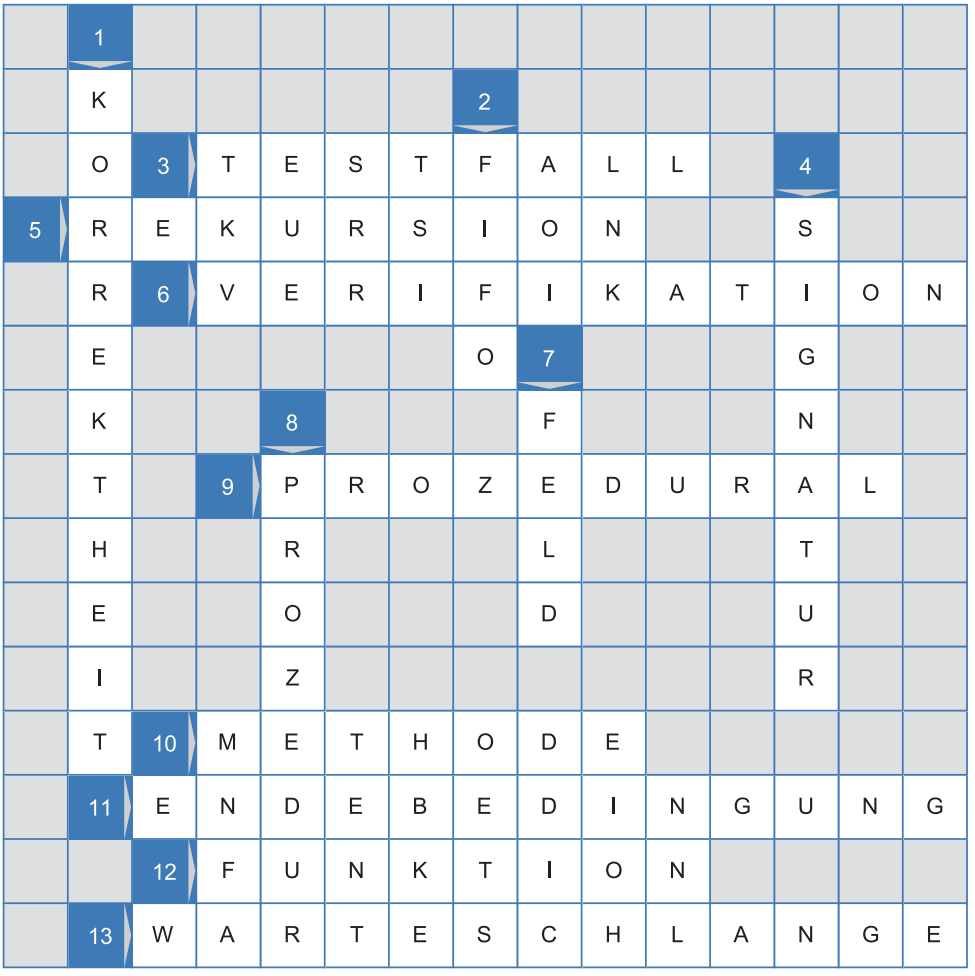


Abb. 3.0-1: Lösung des Kreuzworträtsels zu Prozeduren, Testen und Verifikation.

Gesuchte Wörter:

1 Formaler Beweis, dass ein Programm das tut, was es nach der Spezifikation tun soll.

2 Speicherungsprinzip, bei dem das erste gespeicherte Element auch zuerst dem Speicher wieder entnommen wird (Kürzel).

3 Alle Daten, die benötigt werden, um ein Programm für einen Test auszuführen.

4 Was Sie von einer Operation unbedingt wissen müssen, um sie benutzen zu können.
5 Wenn sich ein Algorithmus direkt oder indirekt selbst aufruft.
6 Ermöglicht es, die partielle Korrektheit eines Programms zu beweisen.
7 Erlaubt es, Variablen vom gleichen Typ zu einer Einheit zusammenzufassen (deutscher Begriff).
8 Anweisungsfolge in einem Programm, die eine Dienstleistung erbringt, einen eigenen Namen besitzt, deklariert und aufgerufen wird.
9 Eine Programmiersprache ist ..., wenn sie es ermöglicht, Teilaufgaben in Prozeduren und Funktionen auszulagern.
10 In Programmiersprachen wie Java und Smalltalk als Bezeichnung für eine Prozedur und eine Funktion verwendet.
11 Spezifiziert eine Zusicherung nach dem Programmende.
12 Sonderfall einer Prozedur. Wird in Ausdrücken aufgerufen. Gibt ein Ergebnis zurück.
13 Datenstruktur mit den Operationen Einfügen und Entfernen.

Glossar

.NET *(.NET)*
Das *.NET-Framework* ist eine Umgebung für die Entwicklung und den Einsatz von *Web Services* und anderen Web-Anwendungen. Es wurde von Microsoft entwickelt, soll aber auch für andere Plattformen zur Verfügung stehen.

Algorithmus *(algorithm)*
Intuitiv lässt sich der Begriff Algorithmus folgendermaßen definieren: Ein Algorithmus ist eine eindeutige, endliche Beschreibung eines allgemeinen, endlichen Verfahrens zur schrittweisen Ermittlung gesuchter Größen aus gegebenen Größen. Die Beschreibung erfolgt in einem Formalismus mit Hilfe von anderen Algorithmen und, letztlich, elementaren Algorithmen. Ein Algorithmus muss ausführbar sein, d. h. ein Prozessor, der den Formalismus kennt und die elementaren Algorithmen beherrscht, muss ihn abarbeiten können. Bei der Ausführung eines Algorithmus werden Objekte manipuliert, insbesondere erfolgt eine Ein- und Ausgabe von Objekten.

Anfangsbedingung *(precondition)*
Eine Anfangsbedingung ist Teil der Spezifikation eines Programms oder Programmteils, der eine Zusicherung vor Programmbeginn beschreibt. (Syn.: Vorbedingung)

Anweisung *(statement)*
Eine Anweisung beschreibt eine auszuführende Handlung in einem Programm einer problemorientierten Programmiersprache oder in einem Algorithmus. Mehrere nacheinander auszuführende Anweisungen werden in den meisten Programmiersprachen durch Semikola voneinander getrennt. Eine solche Folge von Anweisungen wird auch Sequenz genannt. Anweisungen lassen sich gliedern in einfache bzw. elementare Anweisungen (Beispiel: Zuweisung wie `brutto = netto * mwst`) und Steueranweisungen beziehungsweise Kontrollstrukturen (Beispiel: einfache Auswahl wie `if (a > 10) b = 20;`). (Syn.: Programm-Anweisung)

Arbeitsspeicher *(RAM; random access memory)*
Ein Arbeitsspeicher ist ein Medium zur kurzfristigen Aufbewahrung nicht zu umfangreicher Informationen in einem Computer; Bestandteil der Zentraleinheit. (Abk.: RAM)

Ausdruck *(expression)*
Ein Ausdruck ist eine Verarbeitungsvorschrift zur Ermittlung eines Wertes. Besteht aus Operatoren und Operanden. Steht auf der rechten Seite einer Zuweisung.

Auslöschung *(cancellation)*
Bei der Subtraktion von ungefähr gleich großen Kommazahlen werden führende Ziffern gelöscht. Die Differenz verliert dadurch an Genauigkeit.

Ausnahme *(exception)*
Fehlerhafte Situationen während der Ausführung eines Programms (Laufzeitfehler) führen zur Auslösung von Ausnahmen, auf die der Programmierer durch spezielle Sprachkonstrukte reagieren kann, um einen Absturz der Anwendung zu verhindern. In der Programmiersprache Java dient dazu das `try-catch`-Konstrukt.

Auswahl *(selection)*
Ausführung von Anweisungen in Abhängigkeit von Bedingungen. Man unterscheidet die einseitige, die zweiseitige und die Mehrfachauswahl. (Syn.: Verzweigung, Fallunterscheidung)

Betriebssystem *(operating system)*
Ein Betriebssystem ist ein spezielles Programm eines Computersystems, das alle Komponenten eines Computersystems verwaltet und steuert sowie die Ausführung von Aufträgen veranlasst. (Abk.: BS; Syn.: OS)

Bezeichner *(identifier)*
Ein Bezeichner ist ein Name für eine Variable, eine Konstante oder einen Typ in Programmiersprachen, um ihn eindeutig identifizieren zu können. Ein Bezeichner muss eine vorgegebene Syntax einhalten. (Syn.: Name)

BMI *(BMI; Body Mass Index)*
Der BMI ist eine anerkannte Methode, um Über- oder Untergewicht festzustellen. Er wird berechnet nach der Formel: Gewicht in kg / (Größe in m * Größe in m). Er gilt für Frauen und Männer. Das Idealgewicht liegt bei einem BMI zwischen 20 und 24 vor. Ein BMI zwischen 25 und 30 zeigt ein leichtes Übergewicht, ein BMI über 30 zeigt Fettsucht an. Ein BMI unter 18 gilt als Untergewicht.

C
Problemorientierte, prozedurale Programmiersprache, die von Dennis Ritchie in den Bell Laboratories von 1969 bis 1974 entwickelt wurde. Sie ist eine Weiterentwicklung der Sprache B (daher die Bezeichnung C). C erlaubt eine sehr hardwarenahe Programmierung.

C# *(C#)*
Objektorientierte Programmiersprache, die von Microsoft entwickelt und im Jahr 2002 erstmalig präsentiert wurde. Sie besitzt große Ähnlichkeit mit Java. Gesprochen: C Sharp. Sie ist zugeschnitten auf die .NET-Plattform. (Syn.: C Sharp)

C++ *(C++)*
C++ ist eine hybride, problemorientierte Programmiersprache, die sowohl prozedurale als auch objektorientierte Konzepte unterstützt. Sie ist eine Obermenge der Programmiersprache C (1974). Sie entstand zwischen 1980 und 1983 und wurde 1998 als ANSI-Standard festgelegt. Gesprochen: Cplusplus.

Compiler *(compiler)*
Ein Compiler ist ein kompliziertes Programm, das Quell-Programme, geschrieben in einer problemorientierten Programmiersprache, in Objekt-Programme (Sprachvorrat des automatischen Prozessors) umwandelt; Sonderfall eines Übersetzers.

Computersystem *(computer system)*
1 Computer (Hardware) und Programme (Software).
2 Einheit von Anwendungssoftware, Systemsoftware und Hardware.

Datenabstraktion *(data abstraction)*
Daten, die von mehreren Methoden (Prozeduren, Funktionen) gemeinsam genutzt werden und deren Lebensdauer über das Aufrufende einzelner Methoden hinaus bis zum Programmende reicht.

Datenstruktur *(data structure)*
In einer Datenstruktur sind Daten in einer bestimmten Art und Weise strukturiert zu einer Einheit zusammengefasst. Für den Zugriff auf die Daten und die Verwaltung der Daten stehen charakteristische Operationen und Methoden zur Verfügung.

EBNF *(Extended Backus-Naur-Form)*
Die EBNF ist ein Formalismus zur Beschreibung der Syntax von Programmiersprachen (siehe auch Syntaxdiagramm).

Endebedingung *(postcondition)*
Eine Endebedingung ist Teil der Spezifikation eines Programms oder Programmteils, die eine Zusicherung nach Programmende beschreibt. (Syn.: Nachbedingung)

Feld *(array)*
Ein Feld erlaubt es, Variablen vom gleichen Typ zu einer Einheit zusammenzufassen. Auf jedes Element eines Feldes wird über einen Index zugegriffen, der im Allgemeinen erst zur Laufzeit berechnet wird. (Syn.: Reihe, Reihung)

Funktion *(function)*
Eine Funktion ist eine Anweisungsfolge in einem Programm, die eine Dienstleistung erbringt, einen eigenen Namen besitzt und deklariert wird. An der Stelle in einem Programm, an der diese Dienstleistung benötigt wird, steht ein Funktionsaufruf (Angabe des Funktionsbezeichners und der aktuellen Parameter), der eine Verzweigung zur Funktionsdeklaration bewirkt. Eingabeinformationen werden über Parameter übergeben. Jede Funktion besitzt einen Ausgabeparameter bzw. einen Ergebnisparameter. Der Name des formalen Ausgabeparameters ist identisch mit dem Funktionsnamen. Eine Funktion wird in Ausdrücken aufgerufen (siehe auch Prozedur).

Genauigkeit *(accuracy)*
Maß für die Exaktheit von Kommazahlen.

Halblogarithmische Darstellung *(floating point)*
Bei einer halblogarithmischen Darstellung wird eine reelle Zahl durch den Mantissenteil, die Basis und den Exponenten ($\pm m * b^{\pm e}$) beschrieben. (Syn.: Gleitpunktdarstellung, Gleitkommadarstellung)

HTML *(HTML; hypertext markup language)*
Dokumentenauszeichnungssprache, die es mit Hilfe von HTML-Befehlen erlaubt, inhaltliche Kategorien von HTML-Dokumenten, z. B. Überschriften und Absätze, zu kennzeichnen. So ausgezeichnete Dokumente werden von Web-Browsern interpretiert und dargestellt. Die Dateiendung einer HTML-Datei lautet `.html` bzw. `.htm`.

Interpreter *(interpreter)*
Ein Interpreter analysiert Anweisung für Anweisung eines Programms und führt jede analysierte Anweisung sofort aus, bevor er die nächste analysiert. Im Gegensatz zu einem Compiler wird kein Maschinencode erzeugt und abgespeichert. Interpreter werden für Makrosprachen und Skriptsprachen (z. B. Perl, JavaScript, VBScript) eingesetzt.

Invariante *(invariant)*
Eine Invariante ist eine Zusicherung, die innerhalb von Schleifen unabhängig von der Anzahl der Durchläufe immer gültig ist.

Java *(Java)*
Eine der am meisten eingesetzten objektorientierten Programmiersprachen, 1990 von der Firma Sun Microsystems entwickelt. Man unterscheidet Java-Applets, die in einem Web-Browser ausgeführt werden, Java-Servlets, die auf einem Web-Server ausgeführt werden und Java-Anwendungen, die als eigenständige Programme auf einem Computersystem laufen. Java verfügt über ein besonderes Sicherheitskonzept.

Java-Applet *(Java applet)*
Programm, geschrieben in der Programmiersprache Java, das in einem Webbrowser abläuft.

Java-Bytecode *(Java bytecode)*
Plattformunabhängiges Zwischencodeformat, in das Java-Programme übersetzt werden. Wird zur Laufzeit von der *Java Virtual Machine* ausgeführt.

Java-Paket *(Java package)*
In Java können Klassen zu Paketen zusammengefasst und in einem Ordner abgelegt werden. Vor jede Klasse muss die Anweisung `package Paketname;` geschrieben werden. Klassen in einem Paket können in anderen Programmen verwendet werden. Dies geschieht durch die Angabe einer `import`-Anweisung der Art `import Paketname.Klassenname;` bzw. `import Paketname.*;`, wenn alle Klassen eines Pakets verwendet werden sollen. Ein Paket kann selbst Pakete enthalten. (Syn.: Subsystem, subject, category)

Java-Servlet *(Java servlet)*
Ein Java-Servlet ist ein Java-Programm, das auf einem Web-Server läuft, Anfragen von Web-Clients entgegennimmt und HTML-Code als Ausgabe erzeugen kann. Außerdem können Java-Servlets auf Anwendungs-Server zugreifen. JSPs (JavaServer Pages) werden von einem JSP-Server in Java-Servlets übersetzt. (Syn.: Servlet, servlet)

JavaScript *(JavaScript)*
JavaScript ist die am meisten verbreitete Skriptsprache zur Verknüpfung von Programmcode mit statischen HTML-Seiten. Sie ermöglicht es, Webseiten dynamisch zu verändern. Obwohl es der Name vermuten lässt, handelt es sich *nicht* um eine Teilmenge von Java. JavaScript gilt als *unsicher*, da Webseiten »bösartige« JavaScript-Programme enthalten können, die dann auf dem Web-Client ausgeführt werden.

JDK *(JDK; Java Development Kit)*
Entwicklungsumgebung der Firma Oracle für Java. Enthält einen Java-Compiler, einen Java-Interpreter, eine Java-Laufzeitumgebung, Javadoc usw.

JScript *(JScript)*
JScript ist eine firmenspezifische JavaScript-Version von Microsoft.

JSP *(JSP; JavaServer Pages)*
Eine JSP ist ein HTML-Dokument, das eingebettete Java-Anweisungen enthält, um dynamische Web-Inhalte zu erzeugen. Die Java-Anweisungen werden durch besondere Markierungen *(tags)* innerhalb von HTML gekennzeichnet. Eine JSP wird von einem JSP-Server in ein Java-Servlet übersetzt und anschließend ausgeführt.

JVM *(JVM; Java Virtuelle Maschine)*
JVM ist die Bezeichnung für den Java-Interpreter, der den Java-Bytecode zur Laufzeit analysiert und interpretiert (Java-Laufzeitumgebung). (Syn.: Java virtual machine, VM, Virtuelle Maschine)

Konkatenation *(concatenation)*
Eine Konkatenation ist in der Programmierung eine Operation (oft durch + gekennzeichnet), mit der zwei Zeichenketten (Strings) zusammengefügt werden, z. B. `String Text = "Hello" + " World";`

Konstante *(constant)*
Eine Konstante ist ein Sonderfall einer Variablen, bei der nach der Zuweisung eines Wertes (Initialisierung) dieser nur noch gelesen, aber nicht mehr verändert werden darf.

Kontinuum *(continuum)*
Reelle Zahlen bilden ein Kontinuum. Jedem Punkt der Zahlenachse entspricht eine reelle Zahl. In jedem noch so kleinen Intervall befinden sich unendlich viele reelle Zahlen.

Kontrollstrukturen *(control structures)*
Kontrollstrukturen geben an, in welcher Reihenfolge (Sequenz) und ob (Auswahl) bzw. wie oft (Wiederholung) Anweisungen ausgeführt werden sollen

(lineare Kontrollstrukturen genannt) bzw. ob andere Programme aufgerufen werden (Aufruf).

Korrektheit *(correctness)*
Die partielle Korrektheit eines Programms, d. h. die Konsistenz zwischen Spezifikation und Implementierung, kann durch Verifikation gezeigt werden. Ist außerdem bewiesen, dass das Programm stets terminiert, dann ist die totale Korrektheit gezeigt.

Literal *(literal)*
Ein Literal ist eine nicht veränderliche Repräsentation des Werts eines Variablen bzw. einer Konstanten. Die Zahl »Einhundertundzehn« kann in Java beispielsweise durch folgende Literale dargestellt werden: `110`, `0156`, `0x6E`, `110L`, die Zeichenkette »Hallo Welt« z. B. durch `"Hallo Welt"`.

Methode *(method)*
In Programmiersprachen wie Java und Smalltalk ist eine Methode eine Bezeichnung für eine Prozedur und eine Funktion. Eine Methode löst eine eigenständige Teilaufgabe innerhalb eines Programms und kommuniziert über Ein- und Ausgabeparameter mit dem rufenden Programm (Parameterübergabemechanismus). Eine Methode stellt dem aufrufenden Programm eine Dienstleistung in Form einer funktionalen Abstraktion zur Verfügung. (Syn.: Prozedur, Funktion, Operation, procedure, function, operation)

Nassi-Shneiderman-Diagramm *(Nassi-Shneiderman diagram)*
Grafische Darstellung linearer Kontrollstrukturen, auch Struktogramm-Notation genannt.

Nicht-terminales Symbol *(nonterminal symbol)*
Ein nicht-terminales Symbol ist ein Symbol, das als Platzhalter dient und durch Syntaxregeln letztendlich vollständig auf Sequenzen terminaler Symbole zurückgeführt wird.

normalisiert *(normalized)*
Zahlen in Gleitpunkt-Darstellung sind normalisiert, wenn der Mantissenteil die Bedingung `1 / b <= m < 1` erfüllt. Dadurch wird eine eindeutige Zahldarstellung erreicht.

Objektorientierte Programmiersprache *(object-oriented programming language)*
Problemorientierte Programmiersprache, die die Konzepte der Objektorientierung wie Klassen, Objekte und Vererbung unterstützt. Beispiele für objektorientierte Programmiersprachen sind Smalltalk-80 (die erste objektorientierte Sprache), Eiffel, Java und C#. (Abk.: OOPL)

Parameter *(parameter)*
Variable bzw. Literale, die am Informationsaustausch zwischen Prozeduren, Funktionen oder Methoden beteiligt sind. Sie werden in der Parameterliste aufgeführt. Formale Parameter stehen in der Parameterliste der Deklaration, aktuelle Parameter befinden sich in der Parameterliste beim Aufruf (Parameterübergabemechanismus).

Pixel *(pixel)*
Kleinste Informationseinheit, die dargestellt werden kann. Pixel sind von quadratischer Form. Je mehr Pixel ein Bild enthält, desto besser ist seine Qualität. Der Begriff »Pixel« wurde aus den Wörtern »Picture Element« gebildet.

Plattform *(platform)*
Hardware-Architektur eines bestimmten Modells oder einer bestimmten Familie von Computersystemen, z. B. Windows-Plattform. Eine Plattform ist in der Regel eine Kombination aus Betriebssystem und Prozessortyp (Prozes-

sor). Software wird in der Regel für eine Plattform entwickelt. (Syn.: Computer-Plattform, computer platform)

Problemorientierte Programmiersprache *(problem oriented programming language)*
Programmiersprache, deren Sprachvorrat und Sprachkonstruktion problemnahe Formulierungen von Algorithmen ermöglicht. Programme müssen von einem Compiler in die Maschinensprache des jeweiligen Prozessors umgesetzt werden, damit eine automatische Abarbeitung möglich ist. Alternativ können Programme auch durch Interpreter analysiert und ausgeführt werden. (Syn.: Höhere Programmiersprache, Benutzernahe Programmiersprache)

Processing
Auf die Einsatzbereiche Grafik, Animationen, Simulationen, Video, Sound, Typografie und 3D spezialisierte objektorientierte Programmiersprache, die auf Java basiert.

Programm *(program)*
1 Streng formalisierte, eindeutige und detaillierte Handlungsanleitung bzw. Vorschrift, die maschinell ausgeführt werden kann.
2 Eine Folge von Anweisungen in einer Programmiersprache oder in einer Maschinensprache. Die Anweisungen werden von einem Prozessor abgearbeitet.

Programmablaufplan-Notation *(control flow notation)*
Die Programmablaufplan-Notation ist eine in DIN 66001 genormte, grafische Darstellungsform für Kontrollstrukturen von Algorithmen. (Abk.: PAP; Syn.: Flussdiagramm-Notation)

Prozedur *(procedure)*
Anweisungsfolge in einem Programm, die eine Dienstleistung erbringt, einen eigenen Namen besitzt und deklariert wird. An der Stelle in einem Programm, an der diese Dienstleistung benötigt wird, steht ein Prozeduraufruf (Angabe des Prozedurbezeichners und der aktuellen Parameter), der eine Verzweigung zur Prozedurdeklaration bewirkt. Der Informationsaustausch geschieht über Parameter. Ein Prozeduraufruf ist eine Anweisung im Gegensatz zu einem Funktionsaufruf (Funktion). (Syn.: Methode, method)

Prozeduraufruf *(procedure call)*
Ein Prozeduraufruf ist eine Anweisung, die die Ausführung einer deklarierten Prozedur bewirkt. Nach dem Prozedurnamen können in runden Klammern die aktuellen Parameter übergeben werden.

Prozessor *(processor; central processing unit)*
1 Allgemein: Eine Maschine, die die Anweisungen eines Programms abarbeiten kann.
2 Speziell: Der Teil der Zentraleinheit, der Programmanweisungen aus dem Arbeitsspeicher liest, ihre Ausführungen vornimmt, zu verarbeitende Informationen aus dem Arbeitsspeicher liest sowie Zwischenergebnisse und Ergebnisse im Arbeitsspeicher ablegt. (Abk.: CPU)

Regressionstest *(regression test)*
Wiederholung der bereits durchgeführten Tests nach Änderungen des Programms. Er dient zur Überprüfung der korrekten Funktion eines Programms nach Modifikationen, z. B. Fehlerkorrekturen.

Rekursion
Eine Rekursion liegt vor, wenn sich Algorithmen (Prozeduren, Funktionen, Methoden) direkt oder indirekt selbst aufrufen. Rekursive Probleme sind dadurch gekennzeichnet, dass sie sich auf weitgehend identische einfachere Teilprobleme reduzieren lassen.

Repräsentant *(representative)*
Ein Repräsentant ist eine numerisch-reelle Zahl, die reelle Zahlen aus einem Intervall des Kontinuums darstellt.

Sequenz *(sequence)*
Bei einer Sequenz werden mehrere Anweisungen hintereinander, von links nach rechts und von oben nach unten, ausgeführt (Aneinanderreihung). Die Anweisungen werden in der Regel durch ein Semikolon voneinander getrennt.

Signatur *(signature)*
Die Signatur einer Operation besteht aus ihrem Namen sowie den Namen und Typen ihrer formalen Parameter. Die Signatur enthält alle Daten, die notwendig sind, um die Operation nutzen, d. h. aufrufen zu können.

Skriptsprache *(scripting language)*
Programmiersprache, die von einem Interpreter übersetzt und ausgeführt wird. Skriptsprachen verzichten oft auf Sprachkonstrukte, die bei klassischen Programmiersprachen üblich sind, z. B. die explizite Deklaration aller Variablen.

Software-Ergonomie *(usability, software ergonomics)*
Menschengerechte Gestaltung eines Software-Arbeitsplatzes, d. h. der Anwendungssoftware und der Arbeitsoberfläche.

Struktogramm-Notation *(Nassi-Shneiderman diagram)*
Grafische Darstellung linearer Kontrollstrukturen, auch Nassi-Shneiderman-Diagramm genannt. (Syn.: Nassi-Shneiderman-Diagramm)

Strukturiertes Programmieren i.e.S. *(Structured Programming)*
Strukturiertes Programmieren im engeren Sinne liegt vor, wenn zur Beschreibung eines Algorithmus ausschließlich lineare Kontrollstrukturen verwendet werden (keine `goto`- bzw. Sprungkonstrukte).

Syntaxdiagramm *(syntax diagram)*
Ein Syntaxdiagramm ist eine grafische Darstellung der Backus-Naur-Form (siehe auch EBNF); üblich zur Beschreibung der Syntax von Programmiersprachen.

Terminales Symbol *(terminal symbol)*
Ein terminales Symbol ist ein nicht weiter zerlegbares Symbol einer Sprache; siehe auch nicht-terminales Symbol.

Termination *(termination)*
Algorithmen und Programme müssen dynamisch endlich sein, d. h. müssen in endlicher Zeit ausgeführt werden können. (Syn.: dynamische Endlichkeit)

Terminationsfunktion *(termination function)*
Eine Terminationsfunktion dient zur Prüfung der Termination einer Schleife. Muss bei jedem Schleifendurchlauf um mindestens eins kleiner werden und stets positiv bleiben.

Test-First-Ansatz *(test first approach)*
Der Test-First-Ansatz geht davon aus, dass der Entwurf von Tests und deren Ausführung auch das Design des Programms vorantreibt. Bevor ein Stück eines Programms geschrieben wird, wird zuerst ein Test entworfen. Anschließend wird nur soviel Programmcode geschrieben, wie der Test verlangt. Dementsprechend findet die Programmentwicklung in einer raschen Folge von Test- und Implementierungsschritten statt.

Testdatum *(test data)*
Stichprobe der möglichen Eingabewerte eines Programms, die für die Testdurchführung verwendet wird. In der Regel ist ein Testdatum Teil eines Testfalls.

Testfall *(test case)*
Ein Testfall ist ein Satz von Testdaten, der die vollständige Ausführung eines Pfads des zu testenden Programms verursacht sowie das erwartete Sollergebnis.

Texteditor *(text editor)*
Ein Texteditor ist ein Programm, das es ermöglicht Texte zu erfassen, zu speichern und zu bearbeiten. Ein Texteditor speichert – im Gegensatz zu einem Textverarbeitungssystem wie z. B. Microsoft Word – keine Steuer- oder Formatierungsangaben. Die Dateiendung ist in der Regel `.txt`.

Typ *(type)*
Gibt an, aus welchem Wertebereich die Werte sein dürfen, die einer Variablen bzw. einer Konstanten zugewiesen werden können. (Syn.: Datentyp)

Typumwandlung *(type conversion)*
Explizite oder implizite Anpassung der Typen von Variablen und Konstanten, wenn in einem Ausdruck oder einer Zuweisung Variablen bzw. Konstanten mit unterschiedlichen Typen vorhanden sind.

Überladen *(overloading)*
Operationen (Methoden, Prozeduren, Funktionen) mit gleichen Namen können mehrmals in einer Klasse vorkommen, wenn die Anzahl und/oder die Typen der Parameter unterschiedlich sind. Der Compiler ordnet anhand der Signatur die richtige Operation beim Aufruf zu.

Überlauf *(overflow)*
Überschreitung eines darstellbaren Wertebereichs.

UML *(UML; Unified Modeling Language)*
Die UML ist eine Notation zur grafischen Darstellung von objektorientierten Konzepten. Zur grafischen Darstellung gehören unter anderem Klassendiagramme, Sequenzdiagramme und Aktivitätsdiagramme.

Unicode *(Unicode)*
Genormter 16-Bit-Zeichensatz (65.469 Positionen), der die Schriftzeichen aller Verkehrssprachen der Welt aufnehmen soll (siehe auch ASCII und Latin). (Abk.: UCS)

Variable *(variable)*
Bezeichnet in problemorientierten Programmiersprachen einen logischen Speicherplatz mit dessen Wert. Jede Variable besitzt einen Bezeichner, unter dem man sie ansprechen und ihren Wert verändern kann. In vielen typisierten Programmiersprachen muss einer Variablen ein Typ – oft Datentyp genannt – zugeordnet werden. Der Typ legt fest, welche Elemente als Werte der Variablen auftreten und welche Operationen auf die Variable angewendet werden dürfen. Variablen müssen in vielen Programmiersprachen im Deklarationsteil einer Programmeinheit (Klasse, Prozedur, Funktion, Block usw.) vereinbart werden. Sie existieren, solange die Programmeinheit nicht verlassen wird (Lebensdauer), und sie sind sichtbar in allen implizit oder explizit festgelegten Programmeinheiten, in denen der Bezeichner nicht umdefiniert wurde (Sichtbarkeitsbereich, Gültigkeitsbereich). Ein Sonderfall einer Variablen ist eine Konstante, der nur einmal ein Wert zugewiesen werden kann, der anschließend nicht mehr verändert werden kann.

VBScript *(VBScript)*
Skriptsprache der Firma Microsoft, die eine Teilmenge der Programmiersprache Visual Basic bildet.

Verbalisierung *(verbalization)*
Gedanken und Vorstellungen in Worten ausdrücken und damit ins Bewusstsein bringen. In der Softwaretechnik bedeutet dies, aussagekräftige Namen

und geeignete Kommentare zu wählen und selbstdokumentierende Konzepte, Methoden und Sprachen einzusetzen.

Verifikation *(verfication)*
Die Verifikation ist eine formale Methode, die mit mathematischen Mitteln die Konsistenz zwischen der Spezifikation (siehe Vorbedingung, siehe Nachbedingung) eines Programms und seiner Implementierung für alle möglichen und erlaubten Eingaben beweist. Im Rahmen der Verifikation wird die partielle Korrektheit eines Programms bewiesen.

Verifikationsregeln *(verification rules)*
Verifikationsregeln beschreiben die Auswirkungen einzelner Programmkonstruktionen (Zuweisung, Sequenz, Auswahl, Wiederholung) auf Zusicherungen bzw. geben an, wie Programmkonstruktionen kombiniert werden können.

Warteschlange *(queue)*
Eine Warteschlange ist eine Datenstruktur mit den Operationen Einfügen und Entfernen; sie realisiert das FIFO-Prinzip (*first-in – first-out*).

Wert *(value)*
In Programmiersprachen wird jeder Variablen ein Wert zugeordnet, d. h. im zugeordneten Speicherbereich wird ein Wert bzw. Inhalt durch eine Zuweisung eingetragen. Der Typ der Variablen legt den zulässigen Wertebereich fest. (Syn.: Inhalt)

Wiederholung *(iterative construct)*
Eine Wiederholung bescheibt die wiederholte Ausführung von Anweisungen in Abhängigkeit von einer Bedingung oder für eine gegebene Wiederholungszahl. Man unterscheidet Wiederholungen mit Abfrage der Wiederholungsbedingung vor jedem Wiederholungsdurchlauf, nach jedem Wiederholungsdurchlauf und Wiederholungen mit fester Wiederholungsanzahl. (Syn.: Schleife)

Zentraleinheit *(central unit)*
Die Zentraleinheit ist der Teil eines Computers, in der die eigentliche Informationsverarbeitung stattfindet. Sie besteht aus Prozessor und Arbeitsspeicher. (Abk.: CPU)

Zusicherung *(assertion)*
Eine Zusicherung ist eine meist in Form eines booleschen Ausdrucks beschriebene Eigenschaft oder beschriebener Zustand, der an einer bestimmten Stelle eines Programms immer gilt. Im Rahmen der Verifikation kann das Eingangs- und Ausgangsverhalten eines Programms durch Anfangszusicherungen (siehe Vorbedingungen) und Endezusicherungen (siehe Nachbedingungen) spezifiziert werden.

Zuweisung *(assignment)*
Eine Zuweisung ist eine einfache Anweisung, bei der einer Variablen ein errechneter oder fester Wert zugewiesen wird, d. h. dieser Wert wird in die Speicherzelle(n) der Variablen eingetragen und »überschreibt« bzw. löscht einen eventuell bereits vorhandenen Wert (schreibender Zugriff).

Literatur

[ApOl94]
Apt, K.R.; Olderog, E.-R.; *Programmverifikation? Sequentielle, parallele und verteilte Programme*, Berlin, Springer-Verlag, 1994.
Theoretisch orientierte Einführung in die Verifikation, die auch parallele und nichtdeterministische Programme berücksichtigt.

[Babe90]
Baber, R.L; *Fehlerfreie Programmierung für den Software-Zauberlehrling*, München, Oldenbourg Verlag, 1990.
Gute Einführung in die Verifikation mit vielen methodischen Hinweisen.

[BaKi08]
Bapst, Frederic; Kilchoer, Francois; *Signalling Integer Overflows in Java*, in: Dr. Dobb's Journal, September 2008, 2008, S. 54–58.

[BöJa66]
Böhm, C.; Jacopini, G.; *Flow Diagrams, turing machines and languages with only two formations rules*, New York, ACM Press, 1966.
Beweis, dass alle Programmiersprachen, die über zumindest ein Auswahl- und ein Wiederholungskonstrukt verfügen in Bezug auf ihre Ausdruckskraft gleich mächtig sind (Turing-Vollständigkeit).

[Dijk69]
Dijkstra, Edsger Wybe; *Structured Programming*, New York, Petrocelli.

[Dijk72]
Dijkstra, Edsger Wybe; *Notes on Structured Programming, in: Structured Programming, S. 1–82*, London, Academic Press, 1972.

[DIN EN28631]
Programmkonstrukte und Regeln für ihre Anwendung, Berlin, Beuth-Verlag, 1994.

[DIN66001]
Sinnbilder und ihre Anwendung, Berlin, Beuth-Verlag, 1983.

[DIN66261]
Sinnbilder für Struktogramme nach Nassi-Shneiderman, Berlin, Beuth-Verlag, 1985.

[Floy67]
Floyd, R.W.; *Assigning meanings to Programs*, in: Proceedings of the American Mathematical Society Symposium in Applied Mathematics, Vol. 19, 1967, 1967, S. 19–32.

[Fran92]
Francez, N.; *Program Verification*, Wokingham, Addison-Wesley, 1992.
Sehr theoretisch orientierte Einführung in die Verifikation. Neben nichtdeterministischen Programmen werden auch nichtsequenzielle und verteilte Programme behandelt.

[Futs89]
Futschek, G.; *Programmentwicklung und Verifikation*, Wien, Springer-Verlag, 1989.
Empfehlenswertes Lehrbuch, das systematisch in die Verifikationsmethodik einführt.

[FuZu06]
Funke, Joachim; Zumbach, Jörg; *Problemlösen*, in: Handbuch Lernstrategien, Göttingen, Hogrefe, 2006, S. 206–220.

[Hoar69]
Hoare, C.A.R.; *An Axiomatic Basis for Computer Programming*, in: Communications of the ACM, Vol. 12, No. 10, October 1969, 1969, S. 576–583.

[KeRi88]
Kernighan, Brian W.; Ritchie, Dennis; *C Programming Language* , Prentice Hall, 1988.

[MeMa86]
Messer, P.; Marshall, I.; *Modula-2: Constructive Program Development*, Oxford, Blackwell Scientific Publications, 1986.

[NaSh73]
Nassi, Ike; Shneiderman, Ben; *Flowchart Techniques for Structured Programming*, in: SIGPLAN, August, 1973, S. 12–26.

[Wart09]
Wartala, Ramon; *Auf Logos Spuren*, in: iX, 3, 2009, S. 154–158.

Sachindex

A

Aktivitätsdiagramm 102, 139, 156
Algorithmus **12**
Anfangsbedingung **303**
Anforderungen des Kunden 51
Animationen 355
Anweisung **14, 15**, **34**
Arbeitsspeicher **8**, **28**
Argumente 213
array 174
ASCII 86
assertions 162
assignment 32
Assoziativgesetz 81
Aufruf 137, 216
Aufwand
  Speicher 249
  Zeit 249
Aufzählung 200
Ausdruck **33**
Auslöschung **79**
Ausnahme **158**
Ausnahmebehandlungsroutinen 159
Auswahl **109**, **152**
  ein- und zweiseitig 109
Auswahlketten 143
Autotabelle 190

B

Backus-Naur-Form 57
Basiskonzepte 2
.bat-Befehle 295
.bat-Dateien 289
Batch-Dateien 289
batch processing 290
Betriebssystem **10**
Bezeichner **28**
  in Java 30
  Konventionen 29
Block 108
BlueJ 50
BMI **38**, **40**
BNF 57
Body Mass Index 38
boolean 61
Boolesche Gesetze 63
break-Anweisung 130, 134, 145
bubble sort 196
Bytecode 14, 22

C

C **323**
C++ **14**
C# **14**
call by result 225
call by value 217
cancellation 79
casting 94
caught 158
.class-Datei 21
CLASSPATH 44, 49
CLR 14
Command-Line Arguments 219
Common Language Runtime 14
Compiler **11**, **14**
Computersystem **8**
continue-Anweisung 130, 134, 145
control structures 101

D

Datenabstraktion **256**
Datenstruktur **173**, **202**
Defensive Programmierung 157, 162, 177
Delegationsprinzip 235, 245
Didaktisches Schichtenmodell 5
Distributivgesetz 81
do-Wiederholung 129
DSL 349
Dynamik 98
dynamische Endlichkeit 136

E

EBNF **57**
ECMA-6 86
Eingabeparameter 213
  als Felder 219
Eingabe von Gleitpunktzahlen 39
Endebedingung **303**
Endlosschleife 134
Entscheidungsbaum 145
Ergebnisparameter 224
Erweiterbarkeit 164, 206, 277
exception 158
exception handlers 159
exchange sort 196
Extended Backus-Naur-Form 57
Extremwerte 292

F

Fakultät 240

Fallstudie 3
Fallunterscheidung 109
Feld **174**
  als Eingabeparameter 219
  geschachtelt 181
  Iteration über ein 198
  mehrdimensional 181
  Sonderformen 187
finally-Block 159
first in – first out 253
floor 151
Flugtabelle 189
Flussdiagramme 102
for-Schleife 133
Formatierung von Programmen 25
Funktion **223**
  rekursive 240

G

Gültigkeit 31
Genauigkeit **79**
goto 105, 155
GPL 349

H

Halblogarithmische Darstellung **71**, **73**
Hello World 19
HTML **27**

I

identifier 28
if-Regel 310
Implikation 307
import-Anweisung 43
insertion sort 196
Integrated Development Environments 50
Interpreter 14, **14**
Invariante **300**, **311**

J

Java **14**, **17**, **19**
Java-Applet **18**, **366**
Java-Bytecode **350**
Java-Entwicklungsumgebungen 50
Java-IDEs 50
Java-Klasse
  DecimalFormat 228
  Math 225
Java-Paket **43**
Java-Programmschema
  Datenabstraktion 256
  Klasse und Hauptprogramm 27
  mit Methoden 212
Java-Servlet **18**
javac 21
JavaScript **14**
Java virtual machine 14, 50
JDK **43**, **50**
  Arbeiten mit dem 20
JIT-Compiler 16
JScript **14**
JSP **18**
Just-in-Time-Compiler 16
JVM **14**, **19**, **50**, **158, 159**, **350**

K

Kamelhöcker-Notation 30
Klasse 24
Klassenmethoden 219
Klassenvariable 257
Kommentar 26
Konkatenation **37**
Konsequenz-Regel 307
Konsolenfenster 20
Konstante **31**, **37**
Kontinuum **75**
Kontrollstrukturen **101**, 152
  Überblick 103
  Auswahl 152
  lineare 154
Korrektheit **297**
  totale 300
Kreuzworträtsel 99, 170, 321

L

Lösungsmuster 96
label 145
Laufanweisung 131
Laufzeit 149
learning by analogy 3
learning by doing 2
learning by example 3
learning by exploration 3
Lebensdauer 31
Lebenslinie 236
lifeline 236
lineare Kontrollstrukturen 154
Lineare Notation 35
Literal **29**, **33**, **37**
lokale Variable 108
Lokalitätsprinzip 154

M

Marke 145
Maschinensprache 8
Mausereignisse 359
Mehrfachauswahl 117
Methode 25, **210**

parameterlos 209
Microsoft Intermediate Language 14
Modulo-Operation 66
MSIL 14

N

Namenssystematik für Variablen 30
Nassi-Shneiderman-Diagramm **102**
.NET **14**
Nicht-terminales Symbol **56**
normalisiert **74**
n plus 1/2-Schleife 129

O

Objektorientierte Programmiersprache **17**
Operation 25
?-Operator 64
OptiTravel 52
overflow 67
overloaded 235

P

package 43
Paket
inout 44
PAP 102, 156
Parameter **215**
aktueller 216
formaler 216
Gültigkeitsbereich 215
Typumwandlungen 217
variable Anzahl 232
Parameterübergabe 216
Parameterübergabemechanismus 217
Parameterliste 215
passing a reference by value 221
Pixel **350**
Plattform **10**, **19**
Positionszuordnung 217
Potenzschreibweise 71
print() 26
println() 26
Prinzip der Lokalität 154
private
Methode 210
Variablen 256
Problem 95
Probleme
unberechenbare 305
Problemlösen 96
Problemlöseraum 95, 267
Problemorientierte Programmiersprache **8**, **17**
Processing **349**
2D-Grafik 350
Animationen 355
Ausblick 364
Entwicklungsumgebung 350
Mausereignisse 359
Tastaturereignisse 359
Programm **8**
Ampel 360
Animierter Ball 357
Ansprungmarken 145
Artikelliste 209, 238
Artikelliste2 210
Artikelliste3 213
Auswahlkette 143
Balkendiagramm 179
Bestellaufnahme 182
BMI 38
BMI2 40
BMI3 48
DemoAmpel 200
DemoAufruf 138
DemoAusnahme1 157
DemoAusnahme2 157
DemoAusnahme3 159
DemoAusnahme4 159
DemoAuswahl 110
DemoBlock 108
DemoBoolean 63
DemoBrief 90
DemoChar 88
DemoDreieck 189
DemoFeld2 181
DemoFeldLaenge 187
DemoFeldParameter 219
DemoFor1 198
DemoFor2 199
DemoOverflow 68
DemoRekursion 239
DemoRekursionIndirekt 250
DemoSequenz 106
DemoTermination 137
DemoTypausweitung 93
DemoWertuebergabe 218
Diagnose 345
Dreisatz 96
Endlosschleife 134
Fahrenheit 292
fahrenheit.bat 293
Fakultaet 240, 331
FakultaetIterativ 243
Familie 232
FC 285
FileIn, FileOut 282

Fotomanipulation 364
Frachtberechnung 117
Funktionsauswahl 147
Glauben 201
HelloWorld 20, 25, 324
Hommage an Josef Albers 354
IfSchachtelung 141
Kasse 128
KfzKennzeichen 120
Klasse 103
Konferenzzentrum 267
LagerVerwaltung 185
Liter 53
Maximum 305, 311
MinutenBerechnung 234
MWST 42
MWSTTabelle 131
MWSTTabelleFormatiert 229
Netto 223
OptiTravelGesamt 261
OptiTravelTabellen 190
Palindrom 178
Parkscheinautomat 123
Person 341
Potenzieren 298
Praemie 113
Praemie2 162
Pruefziffer 98
Pulsierender Kreis 356
Punkt 340, 342
Rabattstaffel 141
Rechengenauigkeit 80
Rechnungsposten 135
Reiseruecktritt 164
SecuredArithmetics 68
SortAuswahl 197, 220
SortAuswahl3 287
SortAuswahlTest 284
Stimmenzaehlen 174
StundenMinuten 230
Tausche 303
testsort 290
TuermeVonHanoi 246
Typwandlung 94
Vergleich 145, 279
VergleichTest 282
Versandkosten 111
Versicherung 149
Versicherung1 150
Versicherung2 151
Vertausche 302
Vertauschen 97, 335
W3L als Grafik 350
Warteschlange 253
Wetter 130
Wuerfel 227
Wuerfel3D 366
Zchnzaehlen 202
Zeichenbrett 361
Zinstage 64
Programmablaufplan-Notation **102**, **155**
Programme
DemoByteCode 23
Programmierkonzepte v
Programmiersprache
maschinennah 8
problemorientiert 8
Prozedur **209**
parameterlos 209
Prozeduraufruf **217**
Prozessor **8**
Pseudo-EBNF 57
public
class 24
Methode 25, 210
Variablen 256

Q

Quellprogramm 11
Querverweise x
queue 253

R

Regressionstest **281**, 289
Reihe 174
Reihung 174
Rekursion 240, **240**
direkt 244, 250
Eigenschaften 251
indirekt 250
Repräsentant **76**
repräsentative Eingabedaten 292
return 224

S

Schachspiel 68
Schachtelung von
Kontrollstrukturen 140
Schlüsselwort 26
Schleife 123
Schlussregel 308
selection sort 196
sequence diagram 236
Sequenz **106**
Sequenz-Regel 309
sequenzielles Programmieren 252
Shell-Skript 290
Sichtbarkeitsbereich 31
Signatur **234**
Skriptsprache **14**

Software-Ergonomie **259**
Sortieren
  durch Austauschen 196
  durch Auswahl 196
  durch Einfügen 196
Sortierverfahren 195
Sprung 155
Stapelverarbeitung 290
statement 34
Sternesystem ix
Structured Programming 154
Struktogramm-Notation **102**, **155**
Strukturiertes Programmieren i.e.S. **154**
switch-Anweisung 119
Symbol
  nicht-terminales 56
  terminales 57
Syntax
  Auswahl 109
  Block 103
  enum 200
  erweiterte for-Schleife 199
  für einfache Typen 57
  Funktion 224
  Gleitpunkt-Literale 71
  Klasse 256
  Konstantendeklaration 58
  Mehrfachauswahl 117
  Methodendeklaration 103, 215
  return 224
  Sequenz 106
  Statement 103
  try-catch 161
  Variablendeklaration 58
  Wiederholung 123
  Zählschleife 131
  Zuweisungen 60
Syntaxdiagramm **56**
Systemanalytiker 52

T

Türme von Hanoi 244
Tastaturereignisse 359
Terminales Symbol **57**
Termination **136**, 154, 240, 300
Terminationsbedingung 136
Terminationsfunktion **313**
Test-First-Ansatz **280**
Testdatum **280**
Testfall **280, 281**
Texteditor **19**
thrown 158
Tipps ix
Trockentest 135
try-Block 159
Typ **30**, **37**, **55**, **200**
  boolean 61
  byte 65
  char 87
  double 70
  float 70
  int 65
  long 65
  short 65
Typausweitung 92
Typeinengung 94
Typumwandlung **92**

U

Überladen **235**
Überlauf **67**
Übersetzer 10
UML **102**, **218**
  Aktion 106
  Aktivitätsaufruf 138
  Aktivitätsdiagramm 114, 120, 127, 131, 141
  Entscheidungsknoten 109
  Fluss-Notation 109, 120, 131
  guard 109
  Klasse 218, 256, 261
  Knoten-Notation 109, 120, 127, 131, 141
  Operationen 219
  Rechensymbol 138
  Sequenzdiagramm 236, 265
Unicode **30**, **85**, **197**

V

value 28
Variable **28**, **31**, **37**
  externe 304
  feste 305
Variablenkonzept 29
VBScript **14**
Verallgemeinerung 164, 205
Verbalisierung **30**
Verifikation **297**
  von Schleifen 316
Verifikationsregeln **306**
Verzweigung 109
Vom Problem zur Lösung 95, 164, 202

W

Warteschlange **253**
WebSoft-Team 51
Weglassen einer Bedingung 317
Wert **28**, **32**
while-Regel 311

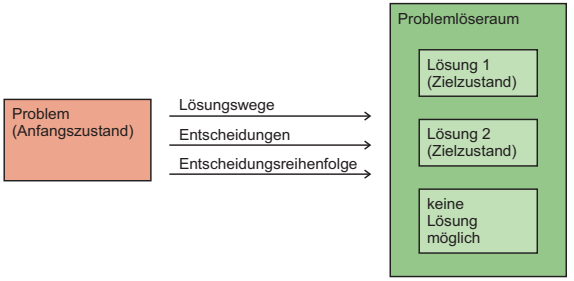
while-Wiederholung 129
Wiederholung **123**, **152**
Wortsymbol 26

Z

Zählschleife 131
Zählvariable 131
Zentraleinheit **9**
Zufallszahlen 226
Zugriff auf Konstanten & Funktionen 227
Zugtabelle 189
Zusicherung **162**, **298**, **301**
Zuweisung **32**, **37**
Zuweisungs-Axiom 308
Zwischencode 14

In 10 Schritten vom Problem zur Lösung

Um zu einem gegebenen Problem eine geeignete Problemlösung in Form eines Programms zu finden, sollten folgende Hinweise beachtet werden:



Es wird von folgenden **Voraussetzungen** ausgegangen:

- Das gegebene Problem ist algorithmisch, d.h. in Form eines Programms, lösbar.
- Es stehen zur Problemlösung *nur* strukturierte und prozedurale Programmierkonzepte einschließlich der Datenabstraktion zur Verfügung.
- Das Problem und die gewünschte Lösung sind verstanden und ausreichend spezifiziert.
- Die Problemlösung besteht aus wenigen Seiten Programmquellcode.

In der Regel sollte folgende **Entscheidungsreihenfolge** eingehalten werden:

1. Prüfen, ob ein ähnliches oder **vergleichbares Problem** einschließlich der Problemlösung bereits bekannt ist. Wenn ja, dann die Lösung übernehmen und unter Umständen anpassen.
2. Prüfen, ob ein **allgemeineres Problem** einschließlich der Problemlösung bereits bekannt ist. Wenn ja, dann überlegen, ob das zu lösende Problem als Sonderfall der allgemeinen Problemlösung behandelt werden kann.
3. Bei komplexen Daten überlegen, welche **Datenstruktur** oder welche Datenstrukturen für die Problemlösung geeignet sind (z.B. Felder, siehe Vom Problem zur Lösung: Teil 3, S. 202). Mögliche Alternativen durchspielen.
4. **Algorithmen** konzipieren, die das Problem lösen. Liegt eine Datenstruktur oder liegen mehrere Datenstrukturen vor, dann prüfen, ob die konzipierten Algorithmen die Datenstruktur(en) optimal »bearbeiten«.
5. Die Algorithmen in **Prozeduren und Funktionen** unterteilen, so dass sie jeweils nur ein Problem lösen bzw. eine Aufgabe erledigen und im Umfang überschaubar sind (siehe Vom Problem zur Lösung: Teil 4, S. 267).
6. Bei der Konzeption von Prozeduren und Funktionen darauf achten, dass die Anzahl und Art der **Parameter** optimal gewählt wird, um eine allgemeine und flexible Lösung zu gewährleisten.
7. Prüfen, ob eine **rekursive** Lösung besser als eine iterative ist oder nicht.
8. Für jede Prozedur und Funktion zuerst die geeigneten **Kontrollstrukturen** (Wiederholung, Auswahl) (siehe Vom Problem zur Lösung: Teil 2, S. 164) ermitteln und anschließend die **einfachen Anweisungen** (siehe Vom Problem zur Lösung: Teil 1, S. 95) programmieren.
9. Prüfen, ob eine **Datenabstraktion** benötigt wird, d.h. die Datenstruktur muss noch zur Verfügung stehen, wenn einzelne Algorithmen in Form von Prozeduren und/oder Funktionen beendet sind.
10. Folgende Prüfungen sind durchzuführen:
    a. Gibt es alternative Lösungen?
    b. Kann die Lösung verallgemeinert werden?
    c. Ist die Lösung erweiterbar?
    d. Ist die Lösung defensiv programmiert?
    e. Wurden geeignete Sprachelemente verwendet?

Es empfiehlt sich folgende **iterative Vorgehensweise** (siehe Vom Problem zur Lösung: Teil 4, S. 267):

1. Zuerst Erstellung einer umgangssprachlichen oder semiformalen Lösungsbeschreibung.
2. Dann Umsetzung in die formale Syntax der gewählten Programmiersprache.

Iterativ bedeutet, dass schrittweise eine Lösungsbeschreibung erstellt, programmiert und getestet wird, anschließend der nächste Schritt der Lösung erstellt, programmiert und getestet wird usw.