

# Einführung in die Programmierung

Rudolf Berrendorf

Juni 2025

© 2025 Rudolf Berrendorf

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung auf elektronischen Medien.

Für Hinweise auf Fehler und Verbesserungsvorschläge bin ich dankbar: [rudolf.berrendorf@h-brs.de](mailto:rudolf.berrendorf@h-brs.de)

# Vorwort

Das vorliegende Buch richtet sich an Studienanfänger der Informatik und anderer verwandter Fachgebiete, die sich grundlegende *Konzepte des Programmierens* erarbeiten wollen. In einem Informatikstudiengang geschieht solch eine Einführung in die Programmierung üblicherweise im ersten Semester und bildet die Grundlage für folgende weiterführende Veranstaltungen der Kerninformatik.

Warum ist dieses Buch nicht einfach eine Anleitung für die Programmiersprache X oder Y, wo vielleicht sogar anhand möglichst vieler einfacher Beispielprogramme das Programmieren in dieser Programmiersprache vorgestellt wird? Und wieso wird es in einigen Teilen dieses Buchs so formal, was oft nicht den Neigungen von Studienanfänger entgegenkommt? Und wieso muss man sich mit so vielen Details wie zum Beispiel der Darstellung von Zahlen in einem Rechner befassen? Diese beim ersten Durchblättern des Buchs für einen Programmieranfänger vielleicht aufkommenden Fragen möchte ich bereits in diesem Vorwort beantworten und gleichzeitig dieses Vorgehen auch motivieren.

**Programmieren gleich Beherrschen einer Programmiersprache?** Die Erfahrung zeigt, dass Studienanfänger manchmal andere Vorstellungen und damit verbunden auch Erwartungen hinsichtlich des Inhalts einer solchen einführenden Veranstaltung zur Programmierung haben. Oft liegt es daran, dass sie Programmierung mit der Beherrschung einer Programmiersprache gleichsetzen. Niemand käme aber auf die Idee, von der Tatsache, dass jemand ein Kochbuch fließend lesen kann daraus ableiten zu wollen, dass derjenige damit auch ein Mehrsternekoch ist. Um bei der Analogie zu bleiben: Ein Koch muss neben der Beherrschung der Fachsprache (dies ist ganz klar eine Grundvoraussetzung!) für einen optimalen Erfolg auch ganz genau wissen, wie zum Beispiel Zutaten selbst in kleinsten Mengen sich auf das Gesamtgericht auswirken, welche Prozesse beim Kochen stattfinden, aber auch genauso, welche Ansprüche der Gast (als Auftraggeber und Geldgeber) stellt und wie man ein Menu bei einer vorgegebenen Qualitätsanforderung kostenoptimal und termingerecht gestaltet. Und manche Köche kochen nicht nur einfach Rezepte nach, manche entwickeln auch neue Rezepte, und wiederum andere entwickeln sogar neue Kochprozesse oder untersuchen neue Zutaten ...

Und ähnlich ist es auch in der Informatik. Eine Programmiersprache ist ein wichtiges Werkzeug, ohne dessen Beherrschung eine sinnvolle Programmierung nicht möglich ist. Diesem Anspruch wird in diesem Buch auch genüge getan, indem der grundlegende Teil der Programmiersprache Java umfassend und auch tiefgehend behandelt wird. Solch eine Beherrschung einer Programmiersprache an sich ist aber nur die notwendige und nicht die hinreichende Bedingung, um gegebene Probleme des Berufsalltags zu erfassen, in die Sprache der Informatik zu übersetzen und mit den Hilfsmittel der Informatik diese Probleme systematisch dann umzusetzen. Das Ziel einer Einführung in die Programmierung muss es sein, die *Systematik des Programmierens* zu vermitteln, die *Gesamtzusammenhänge* zu verdeutlichen und die gezielte Auswahl und Anwendung von Methoden und Techniken aus dem Gesamtreertoire der Möglichkeiten bei konkreten Problemstellungen zu ermöglichen. Hat man erfolgreich solche Kenntnisse und Fähigkeiten erlangt, so ist man auch in der Lage, neuartige Problemstellungen und veränderte Randbedingungen einzuschätzen und zu bearbeiten. Und ebenfalls soll es ein – wenn auch weiterführendes – Ziel sein, dass ein Informatiker die Fähigkeit hat, selbst Anstöße zur Weiterentwicklung von Methoden und Techniken der Informatik zu geben. Insofern ist die

konkret vermittelte Programmiersprache bei einer Einführung in die Grundlagen der Programmierung (hier ist es Java) eigentlich eher nebensächlich. Durchschaut man hinreichend die Gesamtzusammenhänge, so ist auch das Erlernen einer neuen Programmiersprache oder das Aneignen neuer Methoden der Informatik mit überschaubarem Zeitaufwand zu bewerkstelligen. Denn darüber sollte sich jeder Informatikneuling im Klaren sein: Die Informatik ist eine der innovativsten und dynamischsten Branchen. Während sich viele Grundkonzepte immer wieder finden, so ist die Halbwertzeit für viele konkrete Ansätze sehr gering, und damit ergibt sich auch und gerade in diesem Bereich ganz eminent der Zwang zum lebenslangen Lernen.

**Muss es denn (manchmal) so formal sein?** Die Vermittlung von Grundlagen der Programmierung als Teil einer Einführung in die Gesamtinformatik muss auch die Motivation zum Verständnis, der Anwendung und der Entwicklung formaler Methoden erbringen, auch wenn Formalismus oft zu Beginn von Studierenden als Mühsal und hohe Hürde empfunden wird. Die Informatik ist eine Naturwissenschaft mit nach wie vor starkem Bezug zur Mathematik. Viele grundlegende und weiterführende Konzepte der Informatik sind aufgrund ihrer Komplexität ohne ein mathematisches Grundgerüst undenkbar oder einfach auch nicht beherrschbar. Oder wollen Sie, lieber Leserin und lieber Leser, sich auf Systeme verlassen, die Flugzeuge steuern, ihr Antiblockiersystem automatisch betätigen oder ihre Kontoführung durchführen, und die in manchen Fällen schon funktioniert haben, aber niemand zum Beispiel aufgrund der Komplexität dieser Systeme so richtig weiß, ob sie auch das nächste Mal richtig funktionieren? Ich nicht! Die Einsicht in die Notwendigkeit formaler Methoden entsteht (leider) erst bei der Konfrontation mit großen oder komplexen Problemstellungen oder dem Nachweis beziehungsweise der Zusicherung von Eigenschaften, die mit "Fingerspitzengefühl", "langer Erfahrung auf dem Gebiet", "scharfem Nachdenken" oder "aus dem Bauch heraus" definitiv nicht mehr lösbar sind!

**Wie lerne ich effektiv Informatik?** Studienanfänger der Informatik haben zu Beginn ihres Studiums zum Teil keine oder falsche Vorstellungen, was sie in einem Informatikstudium erwartet. So ist die Informatik zum Beispiel nicht das unsystematische "Hacken" von Programmen. Die wirkliche Programmierung (die Kodierung des Programms in einer Programmiersprache) wird bei vielen fertig ausgebildeten Informatikern nur einen Bruchteil ihrer Arbeitszeit im späteren Berufsleben ausmachen, einige Informatiker werden gar nicht mehr mit der eigentlichen Programmierung in Berührung kommen. Informatiker werden vielmehr dazu ausgebildet, konzeptionell zu denken und gegebene Probleme systematisch unter Anwendung wissenschaftlich anerkannter Methoden in Lösungen (fertige Systeme, Arbeitsabläufe und so weiter) zu überführen. Dies schließt aber natürlich auch ein, dass Details in diesem Umsetzungsprozess gelöst werden müssen. Und dies schließt ebenso ein, dass Informatiker sich der Werkzeuge (Modellierungswerkzeuge, Programmiersprachen, Programm-/Klassenbibliotheken und so weiter) bewusst sein müssen, die ihnen zur Verfügung stehen und die sie bei diesem Problemlösungsprozess einsetzen können beziehungsweise müssen. Im Laufe des Berufslebens werden sich sicherlich die konkreten Werkzeuge dem technischen, wissenschaftlichen und gesellschaftlichen Wandel anpassen. Die hier verwendete Programmiersprache Java wurde beispielsweise erst 1995 definiert und ist eine relativ junge Programmiersprache. Viele grundlegende Konzepte (Rekursion, Kellerspeicher, Variablenkonzept, Kontrollstrukturen um nur einige zu nennen) sind aber zeitinvariant und finden sich – um bei diesen Beispielen zu bleiben – in allen praxisrelevanten Programmiersprachen der letzten 50 Jahre und deren Implementierungen wieder. Umso wichtiger ist es, dass der Leser solche grundlegenden Konzepte erkennt und versteht!

Die hohen Abrecherquoten in Informatikstudiengängen (wie in den meisten naturwissenschaftlichen- und ingenieurwissenschaftlichen Studiengängen) sind ein lange bekanntes Problem, und die Hochschulen versuchen durch zahlreiche Maßnahmen, diese Problematik anzugehen, ohne an der Qualität der Ausbildung Abstriche machen zu müssen. Zu den Maßnahmen zählen eine frühzeitige Aufklärung über Inhalte und Ziele der Informatik ebenso, wie eine frühzeitige Motivation für die Informatik; nach Aufnahme des Studiums

gibt es ebenfalls weitere Maßnahmen. Die bei weitem erfolgversprechendste Maßnahme kann aber von den Studierenden selbst ergriffen werden: die Beschäftigung mit der Materie, das Verständnis der Prinzipien, die Anwendung des Erlernten auf möglichst viele Probleme (Übung macht den Meister!) und eine kontinuierliche zeitnahe Mitarbeit beziehungsweise Nacharbeit des Stoffs.

**Wie finde ich mich in diesem Buch zurecht?** Das Buch ist in thematische Kapitel aufgeteilt. Am Ende jedes Kapitels werden die wichtigsten Aussagen des jeweiligen Kapitels kurz zusammengefasst, auf Fehlerquellen hingewiesen sowie auch weitere Hinweise gegeben. Zu jedem thematischen Block sollten nach dem Verständnis der Inhalte Übungsaufgaben und Reflektionsfragen gelöst werden, entweder selbstständig oder in einer kleinen und leistungsmäßig ausgeglichenen Arbeitsgruppe. Erst durch die selbstständige Beschäftigung mit einer komplexen Materie kann man diese verstehen, nicht durch das alleinige Lesen eines Buches oder von Musterlösungen! Übungsaufgaben wenden den Stoff direkt an, während Reflektionsfragen zum Denken anregen sollen.

Dieses Buch ist im Rahmen einer Lehrveranstaltung entstanden, die seit mehreren Jahren an der Hochschule Bonn-Rhein-Sieg in Sankt Augustin in zwei Informatikstudiengängen gehalten wird. Diese Veranstaltung ist der erste einführende Teil einer viersemestrigen Reihe zur Programm- und Systementwicklung. Diese Veranstaltungsreihe ist so konzipiert, dass sukzessive Probleme und deren Lösungen behandelt werden, die mit zunehmender Systemkomplexität und -größe entstehen beziehungsweise erforderlich werden (also ein Bottom-Up Ansatz). Dies geht von der hier behandelten Einführung in die Programmierung über eine nachfolgende Veranstaltung zu weiterführenden Java-Konzepten (generische Typen) und Datenstrukturen hin zu zwei Veranstaltungen, die sich im Wesentlichen auf Systementwicklungsaspekte des Software Engineering konzentrieren.

Der Stoff in diesem Buch ist weitgehend aufeinander aufbauend. Einzelne Kapitel können aber auch, je nach Zeitbedarf und Ausrichtung der Lehrveranstaltung, ausgelassen werden. Kapitel 1 hat einführenden Charakter, einige Kapitel, deren Überschriften mit \* markiert sind, sind optional als Erweiterung zu sehen.

**Dank** Sigrid Weil hat mich über viele Jahre in vielfältiger Weise in dieser Lehrveranstaltung unterstützt, wozu ich ihr an dieser Stelle herzlich danken möchte. Andreas Priesnitz hat zahlreiche Verbesserungsvorschläge zu Inhalt und Form gemacht, wofür ich ihm ebenfalls dankbar bin. Unterstützung bei der Durchführung der zugehörigen Lehrveranstaltung gab es durch Javed Razzaq, Moritz Balg, Andre Kless, Stefan Baum und Robert Hartmann; auch Ihnen an dieser Stelle meinen herzlichen Dank. Javed Razzaq hat die Veranstaltung zudem in vielfältiger Weise kontinuierlich unterstützt. Danken möchte ich auch allen, die mich auf Fehler hingewiesen haben.

Rudolf Berrendorf



# Contents

<b>1 Einleitung</b>	<b>1</b>
1.1 Programmierung als Teil eines umfassenden Problemlösungsprozesses . . . . .	1
1.2 Entwicklungsumgebungen zur Software-Entwicklung . . . . .	4
1.3 Kurze Übersicht über die Architektur von Rechnern . . . . .	8
1.4 Zusammenfassung und Hinweise . . . . .	9
<b>2 Einführung in den Algorithmenbegriff</b>	<b>11</b>
2.1 Algorithmus . . . . .	11
2.2 Algorithmische Grundbausteine . . . . .	14
2.2.1 Einzelanweisung . . . . .	15
2.2.2 Sequenz . . . . .	15
2.2.3 Selektion . . . . .	16
2.2.4 Mehrfachselektion . . . . .	16
2.2.5 Iteration . . . . .	17
2.3 Modularität bei komplexeren Lösungen . . . . .	18
2.4 Darstellungsmöglichkeiten von Algorithmen . . . . .	19
2.4.1 Umgangssprachliche Formulierung . . . . .	20
2.4.2 Programmablaufplan . . . . .	21
2.4.3 Aktivitätsdiagramm der Unified Modeling Language . . . . .	22
2.4.4 Struktogramm . . . . .	23
2.4.5 Programm . . . . .	24
2.5 Präzisierung des Algorithmusbegriffs *	24
2.5.1 Universelle Registermaschine . . . . .	27
2.5.2 Funktionales Berechnungsmodell . . . . .	31
2.6 Zusammenfassung und Hinweise . . . . .	34
<b>3 Programmiersprachen</b>	<b>37</b>
3.1 Wichtige Bestandteile von Programmiersprachen . . . . .	42
3.1.1 Formatierung von Programmen . . . . .	43
3.1.2 Kommentare . . . . .	43
3.1.3 Bezeichner . . . . .	45
3.1.4 Variable . . . . .	46
3.1.5 Ausdruck . . . . .	48
3.1.6 Anweisung . . . . .	49
3.2 Spezifikation von Programmiersprachen . . . . .	50
3.2.1 Grammatik . . . . .	51
3.2.2 BNF und EBNF . . . . .	60

3.2.3	Syntaxdiagramme . . . . .	65
3.2.4	Semantik . . . . .	67
3.3	Übersetzung und Ausführung von Programmen . . . . .	69
3.3.1	Interpretierer . . . . .	70
3.3.2	Übersetzer . . . . .	71
3.3.3	Linker . . . . .	76
3.3.4	Virtuelle Maschinen . . . . .	77
3.4	Zusammenfassung und Hinweise . . . . .	77
<b>4</b>	<b>Einfache Datentypen</b>	<b>81</b>
4.1	Informationsdarstellung in Rechnern . . . . .	81
4.2	Wahrheitswerte . . . . .	86
4.3	Ganze Zahlen . . . . .	91
4.3.1	Positive ganze Zahlen . . . . .	92
4.3.2	Positive und negative ganze Zahlen . . . . .	98
4.3.3	Ganze Zahlen in Java . . . . .	101
4.4	Fließkommawerte . . . . .	109
4.4.1	Darstellungsformate für Fließkommazahlen . . . . .	110
4.4.2	Mögliche Probleme bei arithmetischen Operationen . . . . .	115
4.4.3	Fließkommazahlen in Java . . . . .	116
4.5	Zeichen . . . . .	121
4.5.1	ASCII . . . . .	122
4.5.2	8-Bit Erweiterungen . . . . .	123
4.5.3	Unicode und Universal Character Set . . . . .	123
4.5.4	Komprimierung gemäß UTF-8 . . . . .	124
4.5.5	Darstellung von Zeichen . . . . .	126
4.5.6	Zeichen in Java . . . . .	127
4.6	Zeichenketten . . . . .	129
4.7	Einfache Ein- und Ausgabe von Daten . . . . .	131
4.7.1	Daten beim Programmstart übergeben . . . . .	132
4.7.2	Daten während des Programms lesen . . . . .	135
4.7.3	Daten aus einer Datei lesen . . . . .	136
4.7.4	Formatierte Ausgabe . . . . .	136
4.8	Zusammenfassung und Hinweise . . . . .	136
<b>5</b>	<b>Kontrollstrukturen</b>	<b>141</b>
5.1	Ablaufkontrolle über Anweisungen . . . . .	141
5.2	Ausdrucksanweisung . . . . .	141
5.2.1	Zuweisung . . . . .	142
5.2.2	Methodenaufruf . . . . .	143
5.3	Sequenz und Block . . . . .	143
5.4	Selektion . . . . .	144
5.4.1	Einfachselektion . . . . .	145
5.4.2	Mehrachselektion . . . . .	151
5.5	Iteration . . . . .	155
5.5.1	Zählschleife . . . . .	156
5.5.2	Kopfgesteuerte Schleife . . . . .	166
5.5.3	Fußgesteuerte Schleife . . . . .	170

5.5.4	Überführen von Schleifenformen . . . . .	172
5.6	Abweichung von der normalen Schleifenausführung . . . . .	175
5.7	Leere Anweisung . . . . .	180
5.8	Zusammenfassung und Hinweise . . . . .	180
<b>6</b>	<b>Variablen</b>	<b>183</b>
6.1	Deklaration . . . . .	186
6.2	Gültigkeitsbereich . . . . .	189
6.3	Sichtbarkeit . . . . .	192
6.4	Lebensdauer . . . . .	193
6.5	Zusammenfassung und Hinweise . . . . .	194
<b>7</b>	<b>Ausdrücke</b>	<b>197</b>
7.1	Wert eines Ausdrucks . . . . .	197
7.2	Weitere Operatoren . . . . .	200
7.3	Überladen von Operatoren . . . . .	203
7.4	Priorität und Assoziativität von Operatoren . . . . .	203
7.5	Konstante Werte . . . . .	206
7.6	Typumwandlung . . . . .	207
7.7	Zusammenfassung und Hinweise . . . . .	211
<b>8</b>	<b>Programmstrukturierung durch Methoden</b>	<b>215</b>
8.1	Definition einer Methode . . . . .	216
8.2	Aufruf einer Methode . . . . .	221
8.3	Überladen von Methoden . . . . .	228
8.4	Rekursiv definierte Methoden . . . . .	233
8.5	Strategien der Parameterübergabe . . . . .	236
8.6	Umwandlung von Rekursion in Iteration . . . . .	247
8.7	Zusammenfassung und Hinweise . . . . .	254
<b>9</b>	<b>Komplexere Datentypen</b>	<b>257</b>
9.1	Felder . . . . .	258
9.1.1	Eindimensionale Felder . . . . .	260
9.1.2	Mehrdimensionale Felder . . . . .	266
9.2	Referenzen . . . . .	269
9.2.1	Deklaration von Refenzvariablen . . . . .	269
9.2.2	Operationen mit Referenzen . . . . .	271
9.2.3	Referenzen bei Methodenaufrufen . . . . .	276
9.2.4	Strings genauer angeschaut . . . . .	278
9.3	Zusammenfassung und Hinweise . . . . .	279
<b>10</b>	<b>Abstrakte Datentypen</b>	<b>283</b>
10.1	Abstrakter Datentyp . . . . .	283
10.2	Folge . . . . .	285
10.3	Binärbaum *	292
10.3.1	Arithmetische Ausdrücke . . . . .	296
10.3.2	Suchbäume . . . . .	301
10.4	Kellerspeicher . . . . .	303

10.5 Schlangen *	306
10.6 Zusammenfassung und Hinweise	308
<b>11 Grundlagen Objektorientierter Programmierung</b>	<b>309</b>
11.1 Modellierung der Realität	310
11.2 Klassenbildung	311
11.3 Instanzvariablen und Instanzmethoden	314
11.4 Instanziierung, Instanzen, Konstruktoren und Bezugsobjekt	317
11.4.1 Instanziierung und Instanzen	317
11.4.2 Konstruktoren	317
11.4.3 Operator new	318
11.4.4 Bezugsobjekt	321
11.4.5 Lebensdauer von Objekten	323
11.5 Klassenvariablen und -methoden	324
11.6 Datenkapselung	329
11.7 Komplette Beispiele	330
11.7.1 Komplettes Zoo-Beispiel	330
11.7.2 Komplettes Beispiel zu einfachen geometrischen Objekten	332
11.8 Zusammenfassung und Hinweise	334
<b>12 Vererbung</b>	<b>339</b>
12.1 Vererbung	339
12.1.1 Das Prinzip der Vererbung	339
12.1.2 Vererbung in Java	340
12.1.3 Klassenhierarchien	344
12.2 Typanpassungen bei Referenzen	347
12.3 Pakete und Module*	350
12.4 Zugriffsrechte	353
12.4.1 Zugriffsrechte bei Klassen und Schnittstellen	353
12.4.2 Zugriffsrechte bei Attributen und Methoden	354
12.5 Das Prinzip des Polymorphismus	357
12.6 Überladen und Überschreiben von Methoden	358
12.7 Substitutionsprinzip	360
12.8 Kovarianz, Kontravarianz und Invarianz *	361
12.9 Statische und dynamische Bindung	363
12.10 Operator instanceof	366
12.11 Implizite Aktionen	368
12.12 Zusammenfassung und Hinweise	371
<b>13 Weiterführende Konzepte der Objektorientierten Programmierung</b>	<b>375</b>
13.1 Finalisieren	375
13.1.1 Finalisieren von Variablen	375
13.1.2 Finalisieren von Methoden	376
13.1.3 Finalisieren von Klassen	377
13.2 Abstrakte Klassen und Methoden	377
13.2.1 Abstrakte Klassen	377
13.2.2 Abstrakte Methoden	379
13.3 Schnittstellen	381

13.4 Besondere Arten von Klassen . . . . .	387
13.4.1 Wrapper-Klassen *	387
13.4.2 Geschachtelte Klassen . . . . .	388
13.5 UML Klassendiagramme *	391
13.6 Zusammenfassung und Hinweise . . . . .	392
<b>14 Fehlerbehandlung</b>	<b>397</b>
14.1 Klassen von Ausnahmefällen definieren . . . . .	400
14.2 Ausnahmen auslösen . . . . .	402
14.3 Aufgetretene Ausnahmen behandeln . . . . .	403
14.4 Propagieren von Ausnahmen . . . . .	406
14.5 Assertions . . . . .	409
14.6 Zusammenfassung und Hinweise . . . . .	411
<b>15 Komplexitätsbegriff und einfache Such- und Sortierverfahren *</b>	<b>413</b>
15.1 Komplexitätsbegriff . . . . .	413
15.2 Einfache Suchverfahren . . . . .	424
15.2.1 Sequentielle Suche . . . . .	424
15.2.2 Binäre Suche . . . . .	426
15.3 Einfache Sortierverfahren . . . . .	430
15.3.1 Sortieren durch Auswählen . . . . .	430
15.3.2 Sortieren durch Einfügen . . . . .	433
15.3.3 Bubble Sort . . . . .	435
15.3.4 Sortieren mit großen Datenelementen . . . . .	435
15.4 Zusammenfassung und Hinweise . . . . .	438
<b>16 Allgemeine Strategien in Algorithmen</b>	<b>441</b>
16.1 Einführendes Beispiel . . . . .	442
16.2 Besuchsstrategien . . . . .	444
16.3 Erschöpfende Suche . . . . .	446
16.4 Gieriger Ansatz (Greedy) . . . . .	449
16.5 Problemlösung durch Rekursion . . . . .	449
16.5.1 Türme von Hanoi . . . . .	449
16.5.2 Rekursives Sortieren . . . . .	451
16.6 Backtracking . . . . .	455
16.7 Zusammenfassung und Hinweise . . . . .	460
<b>17 Einführung in das Testen und Verifizieren</b>	<b>461</b>
17.1 Testen von Programmen . . . . .	462
17.2 Verifikation von Programmen . . . . .	464
17.2.1 Zusicherungen in Struktogrammen . . . . .	465
17.2.2 Hoare-Kalkül . . . . .	471
17.3 Zusammenfassung und Hinweise . . . . .	483
<b>18 Programmierparadigmen</b>	<b>487</b>
18.1 Imperativer Ansatz . . . . .	487
18.2 Funktionaler Ansatz . . . . .	488
18.3 Logikorientierter Ansatz . . . . .	489

18.4 Zusammenfassung und Hinweise . . . . .	491
<b>A Notation</b>	<b>493</b>
<b>B Syntaxdiagramme</b>	<b>495</b>
<b>C Priorität und Assoziativität von Operatoren</b>	<b>503</b>
<b>D Regeln zum Gültigkeitsbereich und der Lebensdauer von Java-Variablen</b>	<b>505</b>
<b>Literatur</b>	<b>507</b>
<b>Programme</b>	<b>509</b>
<b>Index</b>	<b>515</b>

# Chapter 1

## Einleitung

Dieses Buch dient dazu, den Leser in die Welt der Programmierung einzuführen. Programmieren ist kein Selbstzweck, sondern dient der Formulierung einer geeigneten Problemlösung oder besser sogar der Formulierung einer Lösung für eine ganze Klasse ähnlicher Probleme.

### 1.1 Programmierung als Teil eines umfassenden Problemlösungsprozesses

Programmentwicklung findet heute vielfach und vermehrt unter komplexen Rahmenbedingungen statt. Zu solchen Rahmenbedingungen zählen der Umfang und die Komplexität der Aufgabenstellung an sich, die Zusammenarbeit mit mehreren auch externen Partnern, die Verwendung oft mehrerer komplexer Frameworks (umfangreiche existierende Programmpakete), die Entwicklung unter Zeit- und Kostendruck, die Zuverlässigkeit des zu entwickelnden Systems und vieles weitere. Mögliche Reaktionen auf solche Rahmenbedingungen widersprechenden sich teilweise (beispielsweise Zeitdruck versus Zuverlässigkeit).

Dieses Kapitel dient dazu, zu Beginn dieses Buches dem Leser ein großes Übersichtsbild zum Prozess der Programm- oder Software-Entwicklung zu geben. Diese Ansätze sind auf die Entwicklung großer Systeme ausgerichtet und werden nachfolgend nur teilweise und übersichtsartig vorgestellt und im Buch dann auch nicht weiter behandelt. Die Disziplin des *Software Engineering* behandelt diese Thematik. Dieses Buch behandelt dagegen im Detail die Aspekte in dem Gesamtprozess, die überwiegend im Kodierungsteil eines Phasenmodells (siehe unten) zu finden sind und als Lösung nicht über eine oder wenige Klassen einer objektorientierten Sprache hinausgeht.

Probleme der realen Welt, die von Informatikern gelöst werden sollen, liegen in den allermeisten Fällen leider nicht in einer Form vor, dass man sie direkt in ein Programm, in einen Algorithmus oder ein EDV-System umsetzen kann. Schon alleine eine adäquate, präzise und vollständige Beschreibung eines vorliegenden Problems angeben zu müssen, ist oft eine erste große Hürde. Um einen Problemlösungsprozess – das heißt das Vorgehen, um von einem gegebenen Problem zu einer Lösung zu kommen – gezielt angehen zu können, ist ein systematisches Vorgehen notwendig. Diese Systematik ist umso dringlicher, je umfangreicher oder komplexer das Projekt ist.

Ausgangspunkt für einen Problemlösungsprozess ist meistens ein Problem der realen Welt, wie etwa die Optimierung und/oder Automatisierung eines Ablaufs, das Erkennen von Einbruchsversuchen in ein existierendes EDV-System, das Auffinden von Zusammenhängen in großen Datenbeständen, die Prognose eines Aktienkurses und vieles mehr.

Betrachtet man einen solchen Problemlösungsprozess *allgemeiner*, so soll ein Problem eines **Auftraggebers** (Person, Unternehmen, Hochschule, ...) von einem **Auftragnehmer** (Informatiker, Informatikfachabteilung,

Gruppe mit Personen aus mehreren Fachabteilungen, ...) in ein System umgesetzt werden, das dieses Problem löst. Ein solcher Problemlösungsprozess wird aufgrund der Komplexität eines Vorhabens oft in einzelne *Phasen* unterteilt (**Phasenmodell**), in denen jeweils spezifische Aktivitäten innerhalb einer Phase stattfinden. Im Laufe der Zeit wurden eine Vielzahl von solchen Entwicklungsmodellen von Wissenschaftlern und Praktikern vorgestellt, die alle jeweils ihre Vor- und Nachteile haben. Teilweise haben große Unternehmen selbst eigene Modelle entwickelt, die von den Mitarbeitern bei Produktentwicklungen zwingend genutzt werden müssen. Ein sehr einfaches und auch sehr knapp beschriebenes Beispiel für ein Phasenmodell wäre etwa:

Phase	Beschreibung
Spezifikationsphase	Fachliche Anforderungen ermitteln (Fachliches Problem exakt spezifizieren), Testfälle beschreiben
Entwurfsphase	Aus den Anforderungen eine (Grob-)Lösung in Form einer Software-Architektur entwerfen
Kodierungsphase	Eine detaillierte Software-Lösung unter Beachtung der Software-Architektur implementieren
Testphase	Die Software-Lösung wird systematisch daraufhin getestet, ob sie eine Lösung für das spezifizierte Problem ist (allen Anforderungen entspricht).

Nach diesem sehr einfachen Entwicklungsmodell muss man also, bevor man mit der eigentlichen Algorithmusformulierung anfängt, erst sehr genau das Problem formulieren, was es denn zu lösen gilt. Dies geschieht üblicherweise zusammen mit dem Auftraggeber, der dieses Problem gelöst haben möchte, denn nur er weiß (hoffentlich), was er will. Diese Phase heißt dementsprechend dann auch **Problemspezifikation** oder Problemdefinition. Nach der Spezifikation folgt der (Grob-)Entwurf des Algorithmus in einer allgemeinen Form, ohne zum Beispiel konkreten Bezug zu einer Programmiersprache oder einer bestimmten Rechnerklasse zu nehmen. Diesen Schritt nennt man auch **Entwurfsphase**. Anschließend folgt die Umsetzung des allgemein formulierten Algorithmus in ein konkretes Computerprogramm, das in einer ganz konkreten Programmiersprache implementiert wird. Diesen Schritt nennt man **Programmierung** oder **Kodierung**. In einer nachfolgenden **Testphase** wird das fertige Programm systematisch dahingehend untersucht, ob es allen spezifizierten Anforderungen genügt. Dies schließt natürlich nicht aus, dass auch schon während der Kodierungsphase Teile des Programms getestet werden. Das eben vorgestellte Phasenmodell ist ein sehr einfaches Modell. Es sind in der Praxis weitere Phasenmodelle entwickelt worden, die auch eine andere Aufteilung vorsehen.

In den seltensten Fällen ist in der Praxis das Problem von vorneherein exakt bekannt. Meist ist es so, dass die Person, die das Problem hat und die Person, die das Problem lösen soll (die den Algorithmus entwerfen soll, die das Programm schreiben soll) nicht identisch sind. Zudem haben diese beiden Personen (-gruppen) oft einen anderen Sprachgebrauch und Hintergrundwissen. Zudem machen sie oft unterschiedliche "selbstverständliche" Annahmen. Es ist deshalb sehr wichtig, dass Auftraggeber (Person mit Problem) und Auftragnehmer (Person, die das Problem lösen soll) *gemeinsam* die Spezifikation erarbeiten und diese möglichst exakt formulieren. Fehler, die in einer frühen Phase eines Entwurfsprozesses gemacht werden, sind später (falls überhaupt möglich) nur mit großem Aufwand zu korrigieren.

Neben einem Phasenmodell beschreibt ein zusätzliches **Prozessmodell** (auch **Vorgehensmodell** genannt), wie diese Phasen zeitlich ablaufen sollen, wie sie verzahnt sind, welche Verantwortlichkeiten existieren und weiteres. Ein wiederum sehr einfaches Beispiel eines Prozessmodells wäre die Vorgabe, dass vier Phasen

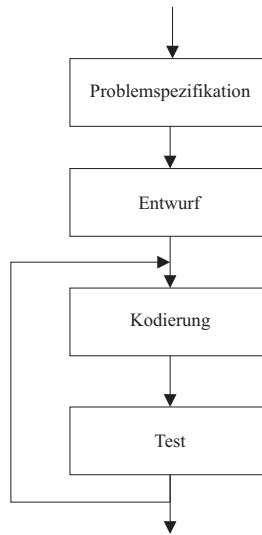


Figure 1.1: Beispiel zur Abhängigkeit der Phasen in einem einfachen Prozessmodell.

existieren (zum Beispiel die oben beschriebene Spezifikationsphase, Entwurfsphase, Kodierungsphase und Testphase) und eine Phase erst beginnen kann, wenn die vorherige Phase vollständig abgeschlossen ist. Abbildung 1.1 zeigt ein Beispiel für solch eine Abfolge der einzelnen Phasen, wobei hier zusätzlich noch nach der Testphase wieder die Kodierungsphase folgen könnte (falls man Programmierfehler während der Testphase entdeckt hätte). Zu erwähnen ist, dass bei einem strikten Vorgehen nach diesem einfachen Prozessbeispiel ein Fehler in der Problemspezifikation (Phase 1) in einer späteren Phase nicht mehr korrigierbar wäre.

### **Beispiel 1.1:**

Das prinzipielle Vorgehen soll an einem Beispiel mit drei Akteuren (Albert, Annette, Frank) und einem Handy erläutert werden. Es werden hier nur grob die einzelnen Schritte erläutert.

**Ausgangssituation:** Albert hätte gerne die Telefonnummer seines Freundes Bernd. Neben Albert steht Annette mit ihrem Handy. Im elektronischen Telefonbuch auf dem Handy ist Bernds Nummer abgespeichert. Annette hat aber leider beide Arme in Gips, so dass sie selber das Handy nicht bedienen kann. Neben ihr steht aber Frank, der zwar wie Albert keine Ahnung von Handys hat, dafür aber alles macht, was Annette ihm sagt. Er versteht leider nur transsyvanisch.

**Problemspezifikation:** Zuerst setzen sich Albert (Auftraggeber) und Annette (Auftragnehmer) zusammen und sprechen darüber, was denn nun genau Albert überhaupt will, das heißt was sein Problem ist. Albert kennt sich nicht mit Handys aus, ihm ist es auch eigentlich egal, wo er Bernds Telefonnummer herbekommt, er ist nur an der Lösung seines Problems interessiert. Er formuliert also gegenüber Annette sein Problem in der folgenden Form: "Ich brauche Bernds Telefonnummer, möglichst schnell". Annette versteht in diesem Fall genau, was Albert will.

**Entwurfsphase:** Annette (Auftragnehmer) erkennt sehr schnell, dass das Problem Bernds Nummer im Handytelefonbuch zu finden allgemeiner Natur ist, nämlich allgemein einen beliebigen vorgegebenen Namen in einem Telefonbuch dieses Handytyps zu finden. Wenn sie für Alberts konkretes Problem eine allgemeine Lösung finden würde, könnte Annette diese Lösung mit einmaligem Entwicklungsaufwand in anderen aber ähnlichen Problemstellungen wiederverwenden (also nochmals verkaufen).

Annette (Auftragnehmer) überlegt, wie sie Frank (Ausführer) erklären kann, wie man für einen gegebenen Namen an die entsprechende Telefonnummer im Handy kommen kann. In einer allgemeinen Formulierung würde dies lauten: Das Handy muss eingeschaltet sein. Das Telefonbuch findet man im Hauptmenü unter Punkt 2. Dort muss man solange durch die Einträge blättern, bis der gesuchte Name im Display erscheint. Die Telefonnummer steht dann unter diesem Namen in der zweiten Zeile des Displays. Es wird in diesem einfachen Beispiel angenommen, dass alle angefragten Namen auch im Telefonbuch eingetragen sind.

**Kodierung:** Annette (Auftragnehmer) sagt Frank (Ausführer) unzweideutig auf transsylvanisch (Franks Sprache), was er Schritt für Schritt machen soll (hier auf deutsch):

- 1) Falls das Handy noch nicht eingeschaltet ist, Handy durch kurzes Tippen des Knopfes unten links einschalten.
- 2) Taste *Menü* drücken.
- 3) Zweimal Taste *Pfeil unten* drücken (es wird damit der Menupunkt Telefonbuch aufgerufen).
- 4) Anschließend Taste *OK* drücken.
- 5) Solange der gesuchte Name nicht in der obersten Zeile des Display erscheint, Taste *Pfeil unten* drücken.
- 6) Unter diesem Namen steht dann in der 2. Zeile des Displays die gesuchte Telefonnummer.

**Testen:** Bevor Annette nun die Lösung für Alberts konkretes Problem einsetzt, testet sie anhand einiger (systematisch gewählter) Testdaten, die sie während der Spezifikation von Albert bekommen hat, ob diese Lösung auch wirklich die Anforderungen erfüllt. ♦

Dieses Beispiel auf Projekte im Informatikumfeld übertragen würde bedeuten, dass der Auftraggeber (zum Beispiel eine Firma, eine Fachabteilung etc.) mit dem Auftragnehmer (einem Informatiker / einer Informatikerin, einer Gruppe von Informatikern etc.) gemeinsam die Problemspezifikation entwirft. Hierbei kann es durchaus zu Verständigungsproblemen kommen (die beseitigt werden müssen), weil beide Seiten eventuell eine unterschiedliche Fachsprache reden oder unterschiedliche Grundannahmen machen. In unserem Entwicklungsmodell beginnt erst nach einer exakten Spezifikation des Problems die Entwurfsphase, in der ein Algorithmus in allgemeiner Form entworfen wird. Dieser wird dann in der Kodierungs- oder Programmierphase in ein Programm in der gewählten Sprache umgesetzt und systematisch getestet. Das fertige Programm wird dann eingesetzt und dabei ausgeführt (Prozessor).

Wie oben bereits erwähnt soll die Thematik der Phasen- und Prozessmodelle und damit zusammenhängende weitere Aspekte an dieser Stelle nicht weiter vertieft werden. Ziel dieser knappen Schilderung war es, die weiteren Themen des Buches in einen Gesamtzusammenhang zu bringen. Der systematische Entwurf großer Software-Systeme wird in einem eigenen Teilgebiet der Informatik mit Namen *Software Engineering* behandelt (siehe Literaturhinweise am Ende des Kapitels). Dort werden zum Beispiel auch Aspekte der Arbeitsteilung (Arbeiten im Team), der Spezifikation sehr großer Probleme und so weiter in den Problemlösungsprozess mit einbezogen. In diesem Buch wird sich auf auf die Beschreibung und Behandlung relativ kleiner, überschaubarer Problemlösungen konzentriert, die in einer Programmiersprache formuliert auf ca. eine Seite passen.

## 1.2 Entwicklungsumgebungen zur Software-Entwicklung

Zur Entwicklung von Programmen in gängigen Programmiersprachen stehen mächtige Entwicklungsumgebungen (die selber wiederum sehr komplexe Programme sind) von verschiedenen Herstellern, teilweise auch als kostenfreie Software zur Verfügung. Diese Entwicklungsumgebungen unterstützen einen Programmierer oder auch Teams von Entwicklern in vielfältiger Weise im Entwicklungsprozess. Eine weit verbreitete Programmierumgebung unter anderem zur Entwicklung von

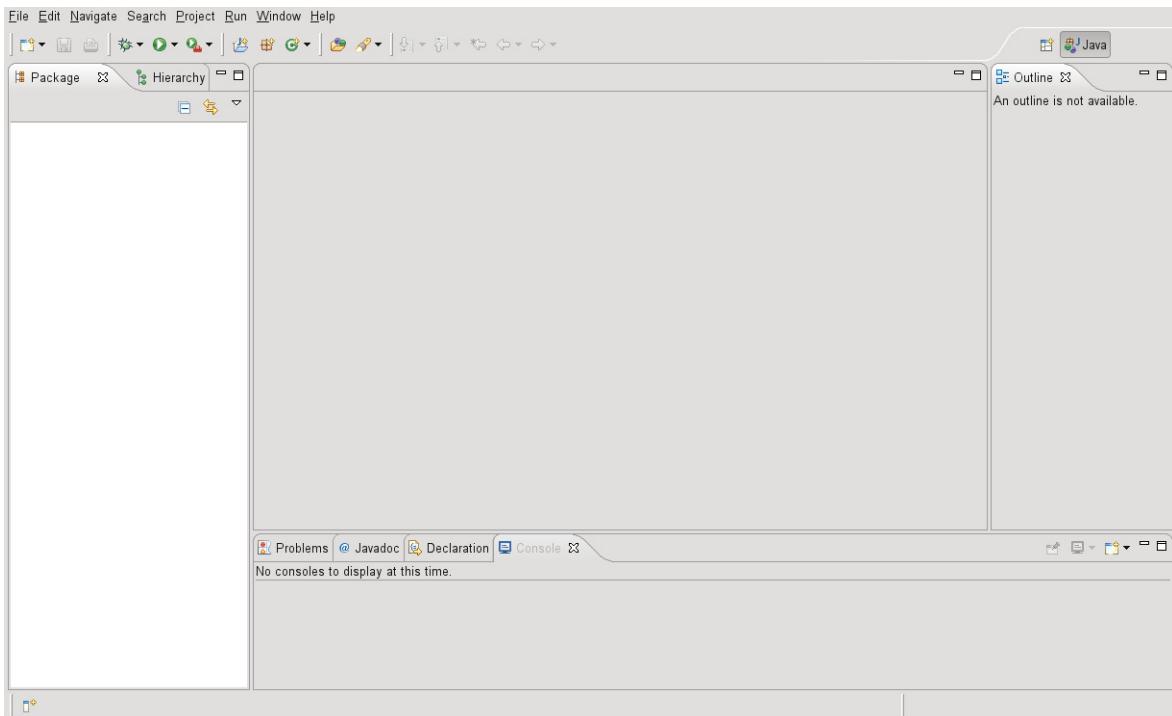


Figure 1.2: eclipse-Ansicht.

Java-Programmen ist **eclipse**[ecl], auf das im Folgenden Bezug genommen wird. Nachfolgend wird eine knappe, sehr grundlegende Einführung in diese mächtige Entwicklungsumgebung gegeben. Auf die Installation und den Start von eclipse unter Windows, AppleOS, Linux oder anderen Betriebssystem wird hier nicht eingegangen, über die oben als Literatur angegebene Webseite zu eclipse findet man entsprechende Anleitungen.

Nach dem Start von eclipse sieht man (nach Wegklicken des Begrüßungsbildschirm über das kleine rote Kreuz) eine Oberfläche, die wie in Abb. 1.2 aussieht. Das Programm eclipse strukturiert alle Programmieraufgaben in Projekte, die wiederum selber Programmdateien enthalten können. Es erleichtert den Umgang mit eclipse gerade für einen Anfänger sehr, wenn man auch kleinere Programmieraufgaben als eigenständige eclipse-Projekte auffasst und entsprechend als solche anlegt.

Für jede Programmieraufgabe wird nach diesem Vorschlag ein Projekt angelegt, wozu folgende Aktionen durchzuführen sind:

1. Anlegen eines neuen Java-Projektes. Jede für sich abgeschlossene Aufgabe wird als Projekt angesehen. Dazu klickt man auf der oberen Menüleiste auf den Punkt **File**, in dem aufklappenden Untermenü auf den Punkt **New** und in dem darunter aufklappenden Untermenü auf den Punkt **Java Project**. Danach erscheint auf dem Bildschirm ein neues Fenster zur Beschreibung des neuen Projekts. Siehe Abb. 1.3.

Tragen Sie in der obersten Zeile den Namen des Projekts ein, lassen alles andere unverändert und klicken auf **Finish**. Danach erscheint links in der eclipse-Umgebung ein neuer Eintrag für das geschaffene Projekt. Der Projektname kann aus Klein- und Großbuchstaben sowie Ziffern bestehen, wobei das erste Zeichen keine Ziffer sein darf.

2. Als nächstes ist zu dem Projekt eine neue Java Programmdatei anzulegen, der Inhalt dieser Datei ist eine sogenannte Klasse. Ein Projekt besteht aus mindestens einer solchen Programmdatei. Gehen Sie dabei so vor, dass Sie wieder in der oberen Menüleiste auf den Eintrag **File** klicken. Im ebenfalls wie

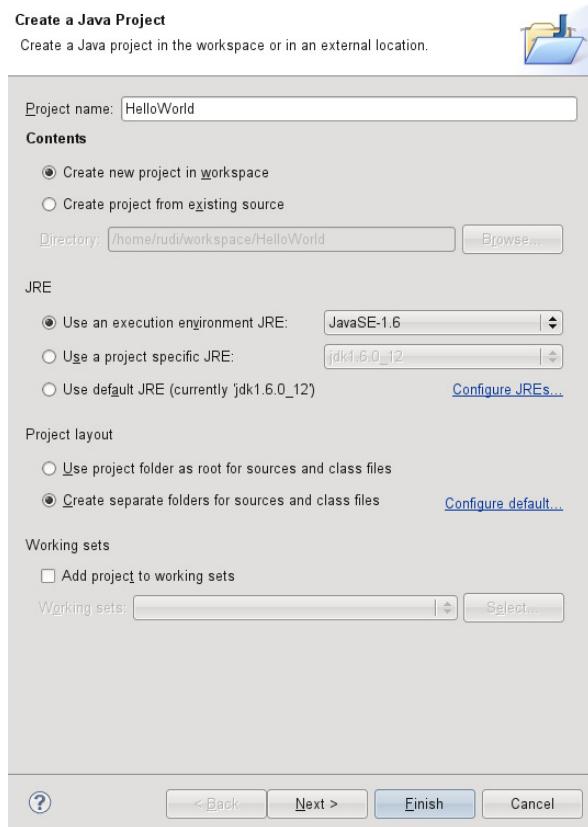


Figure 1.3: Anlegen eines Projekts.

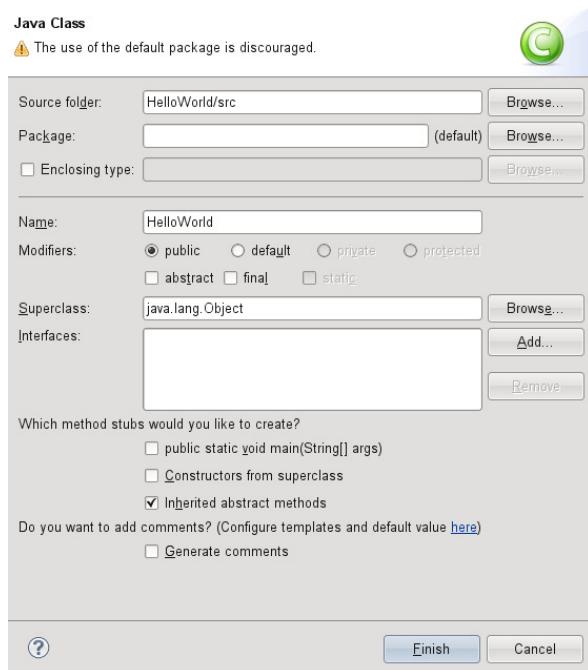


Figure 1.4: Anlegen einer Java-Klasse.

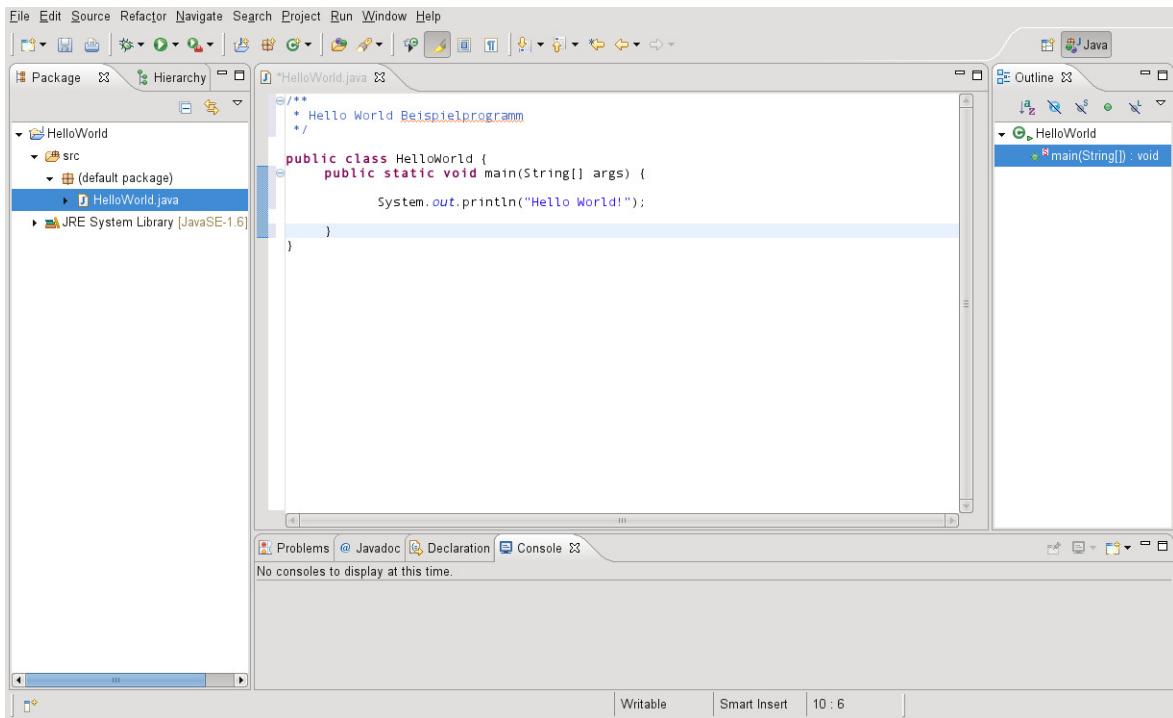


Figure 1.5: Hello World in eclipse.

Listing 1.1: Hello World in Java.

```

1-1 /**
1-2  * Hello World Beispielprogramm
1-3 */
1-4 public class HelloWorld {
1-5
1-6     public static void main(String [] args) {
1-7         System.out.println("Hello World!");
1-8     }
1-9 }
```

eben erscheinenden Untermenü wählen Sie wieder **New** und nun im erscheinenden Untermenü den Punkt **Class** für eine neue Klasse. Siehe Abb. 1.4.

Tragen Sie in der Feld, das mit **Name** benannt ist, den Namen des neuen Programms ein. Für kleinere Programme ist es zu Beginn eine gute Idee, den Programmname wie den Projektnamen zu wählen. Als Übereinkunft gilt, dass der Name mit einem Großbuchstaben beginnt, zusammenhängend (also ohne Leerzeichen) geschrieben wird und die ersten Buchstaben von Teilwörtern ebenfalls groß geschrieben werden. Vermeiden Sie wo immer es geht Umlaute!

3. Nach diesen Vorarbeiten können Sie nun mit dem Programmieren loslegen. Dazu geben Sie ihr Programm in das mittlere große Fenster ein, in das eclipse schon einige wenige Zeilen für Sie eingefügt hat. Weltweit nutzt man als Aufgabenstellung für das erste kleine Einstiegsprogramm, dass auf dem Bildschirm einfach nur *Hello World!* ausgegeben werden soll. Das Programm ist in Listing 1.1 gezeigt. Siehe auch Abb. 1.5.

Geben Sie das Programm wie in der Abbildung bereits getan in das mittlere Fenster ein und speichern

die Datei, indem Sie auf das entsprechende kleine Symbol (Icon) in der Menüleiste klicken. Dabei wird eclipse automatisch das Java-Programm übersetzen (warum dies nötig ist, wird später noch erläutert) und auf etwaige Fehler kontrolliert und markiert. Dies kann zum Beispiel dann der Fall sein, wenn Sie nicht exakt das gleiche Programm eingegeben haben, das oben angegeben war. Wie später zu sehen ist, geht es bei Programmiersprachen sehr genau zu! Falls noch Fehler angezeigt werden (die Zeilen sind dann links mit einem roten Kreuz markiert und im unteren Fenster erscheint eine Fehlermeldung) kontrollieren Sie erst ihr Programm auf korrekte Schreibweise inklusive Groß-/Kleinschreibung, korrigieren den Fehler und speichern das Programm erneut. Solange noch Fehler im Programm angezeigt werden können Sie nicht mit dem nächsten Schritt fortfahren.

4. Starten des Programms. Drücken Sie dazu auf das Symbol mit dem großen grünen Pfeil / Dreieck in der Menüleiste. Im unteren Ausgabefenster sollte jetzt eine Ausgabe erscheinen, wie es im Bild oben angezeigt wird.

### 1.3 Kurze Übersicht über die Architektur von Rechnern

Auch wenn es an den meisten Stellen des Buches kaum relevant ist, für welchen konkreten Rechner man programmieren möchte oder wie ein Rechner/Prozessor aufgebaut ist, so ist ein allgemeines Verständnis über den Aufbau und die Arbeitsweise von Rechnern sehr hilfreich, um die im Weiteren gemachten Herangehensweisen zu verstehen. So hat beispielsweise die (Entwicklung der) Prozessorarchitektur erheblichen Einfluss darauf, welche primitiven Datentypen in Programmiersprachen angeboten werden und welche Restriktionen damit auch verbunden sind (siehe Kapitel 4). An dieser Stelle soll deshalb ein Überblick über den Aufbau und die Architektur von Rechnern gegeben werden, soweit sie für das Verständnis der folgenden Inhalte notwendig und insbesondere auch zum Verständnis einiger Aspekte (zum Beispiel der Darstellung von Daten) hilfreich sind. Bezuglich der Details zur Rechnerarchitektur sei auf Speziallehrbücher zu diesem Thema (siehe Literaturhinweise).

Programmierbare Rechner gibt es in sehr vielen Ausprägungen, von Waschmaschinensteuerungen bis hin zu Supercomputern, wo ein einzelner Rechner eine ganze Maschinenhalle ausfüllt. Die Anforderungen an einen Rechner sind oft sehr unterschiedlich, aber die Grundbestandteile und strukturellen Beziehungen der Komponenten eines Rechners untereinander sind teilweise sehr ähnlich. An dieser Stelle soll die Architektur eines PC (Personal Computer) beispielhaft erläutert werden, weil dies eine sehr verbreitete und allgemein anwendbare Rechnerplattform ist.

Personal Computer besitzen eine Architektur bestehend aus einer oder mehreren Zentraleinheiten (CPU, Central Processing Unit, Prozessor), Hauptspeicher, permanenten Speicher für Dateien und Möglichkeiten der Ein-/Ausgabe (Tastatur, Grafik, Netzwerk, ...). Abbildung 1.6 zeigt die strukturellen Beziehungen der einzelnen Komponenten, das in der benachbarten Abbildung gezeigte Bild einer Platine (Mainboard) ist Teil einer konkreten Realisierung dieser abstrakten Architektur.

Die **Zentraleinheit** nimmt, wie der Name schon andeutet, eine zentrale Rolle ein. Sie ist in der Lage, ein vorliegendes Programm bestehend aus einer Folge sehr einfacher Befehle abzuarbeiten. Die **Befehle** sind spezifisch für diese Prozessorarchitektur und arbeiten auf einer begrenzten Anzahl von **Registern** der CPU, die jeweils genau einen einfachen Wert aus einem begrenzten Wertebereich aufnehmen können (zum Beispiel eine Zahl, einen Buchstaben, einen Wahrheitswert). Welcher Befehl als nächstes ausgeführt werden soll, steuert die **Kontrolleinheit** in der CPU. Die **Funktionseinheit** der CPU ist in der Lage, einfache Operationen (Addition, Multiplikation, Wurzel ziehen, ...) auf Daten auszuführen, die in Registern vorliegen.

Der **Hauptspeicher** ist aufgeteilt in einzelne Zellen, die ebenfalls jeweils genau einen einfachen Wert aufnehmen können (ein Byte). Diese Speicherzellen sind fortlaufend durchnummieriert und besitzen somit eine eindeutige **Hauptspeicheradresse** zur Identifizierung. Der Inhalt des Hauptspeichers ebenso wie der Inhalt

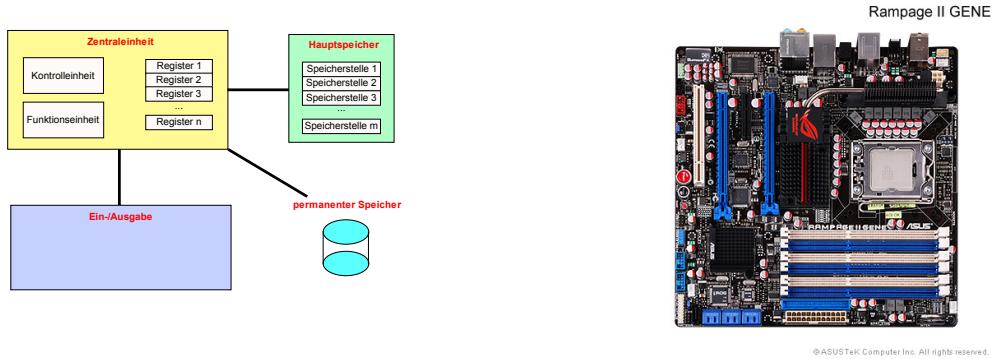


Figure 1.6: Abstrakte Rechnerarchitektur (links) und konkrete Ausprägung in Form einer Platine (rechts).

der Zentraleinheit geht allerdings verloren, wenn der Rechner ausgeschaltet wird. Möchte man Daten über das Ausschalten des Rechners hinweg erhalten, so muss man sie auf Permanentsspeichermedien (Festplatte, SSD, USB-Stick, ...) ablegen.

Heutige Rechner sind Digitalrechner, die ausschließlich mit Bits arbeiten, die jeweils den Wert Null oder Eins annehmen können. Alle Daten müssen dementsprechend als Folge von Nullen und Einsen dargestellt werden. Dieser Aspekt wird später ausführlich in Kapitel 4 behandelt, da damit auch gewisse Konsequenzen verbunden sind.

Ein Beispiel für eine Abfolge von CPU-Befehlen ist etwa:

- Hole aus dem Hauptspeicher den Wert an der Adresse 4711 und speichere ihn in das Register mit der Nummer 5
- Hole aus dem Hauptspeicher den Wert an der Adresse 4753 und speichere ihn in das Register mit der Nummer 6
- Addiere zwei Werte, die im Register 5 und 6 stehen, und speichere das Ergebnis in Register 7
- Speichere den Wert aus Register 7 in den Hauptspeicher an der Adresse 4723

Diese Befehlssequenz nimmt zwei Werte, die im Hauptspeicher an bestimmten Stellen abgelegt sind, addiert diese, legt das Ergebnis der Addition in einem weiteren Register ab und speichert das Ergebnis wiederum im Hauptspeicher an einer bestimmten Stelle.

## 1.4 Zusammenfassung und Hinweise

### Literaturhinweise

Einführende Lehrbücher zu Software Engineering sind u.a. [Bal05] [Bal08]. Die primäre Einstiegsseite zu eclipse ist [ecl], worüber das Programm selbst, Erweiterungen u.a. erhältlich sind. Verbreitete Lehrbücher zur Rechnerarchitektur sind [HP05] [PH09].

### Verstehen

Der Sinn von systematischen Entwicklungsmodellen im Rahmen komplexer Problemlösungsprozesse sollte erkannt sein. Der grundsätzliche Aufbau von Rechnern sollte verstanden sein.

## Kurz und knapp merken

- Phasen- und Prozessmodelle dienen dazu, bei komplexen Problemen systematisch vom Ursprungsproblem zu einer Lösung dieses Problems zu kommen. Im Laufe der Zeit sind sehr viele verschiedene Modelle entwickelt worden.
- Entwicklungsumgebungen sind Programme auf Rechner, die einem Programmierer in vielfältiger Weise bei der Programmierung helfen. Die Programmierumgebung eclipse wird häufig dazu genutzt.
- Der prinzipielle Aufbau von Rechnern ist sehr ähnlich. Bestimmte Bestandteile wie etwa Prozessor, Register und Hauptspeicher sind in allen wichtigen Rechnerarchitekturen zu finden und sind im Wesentlichen auch ähnlich/gleich organisiert.

## Häufige Fehler

Programmierumgebungen sind sehr komplexe Programme, die für bestimmte Zwecke konfigurierbar sind (zum Beispiel über Menüs). Anfänger machen oft den Fehler, dass sie (ohne wirkliche Kenntnisse der Wirkung) an Konfigurationen Änderungen vornehmen wollen, die sie ohne fremde Hilfe nicht mehr rückgängig machen können und die normale Nutzung verhindern/erschweren.

## Übungsfragen

- Wie ist ein Prozessor aufgebaut?
- Was ist die kleinste adressierbare Einheit in einem Hauptspeicher?

## Reflektion des Stoffs

- Beschreiben Sie *mit eigenen Worten* die Aufgaben der Phasen in dem beschriebenen Entwicklungsmodell.
- Welche Aufgaben unterstützt die Entwicklungsumgebung eclipse bei der Programmierung, die Sie bis jetzt kennen?
- Beschreiben Sie *mit eigenen Worten* den Aufbau eines Rechners.

## Chapter 2

# Einführung in den Algorithmenbegriff

In diesem Kapitel wird der Algorithmusbegriff eingeführt und aus verschiedenen Blickwinkeln betrachtet. Zur Einführung des Algorithmusbegriffs soll zuerst genau definiert werden, was nun ein Algorithmus ist. Dazu werden verschiedene Präzisierungen vorgestellt und die Vor- und Nachteile der einzelnen Ansätze erläutert. Anschließend werden erste, einfache Schritte gemacht, wie man systematisch von einer Problemspezifikation zu einem Algorithmus gelangt. All diese in diesem Kapitel behandelten Themen werden auf natürliche Weise zu Programmiersprachen als Vehikel der Mitteilung (Beschreibung von Algorithmen) führen.

Der Algorithmus-Begriff ist ein zentraler Begriff der Informatik. Der Name leitet sich aus einem Namensteil eines bedeutenden arabischen Gelehrten ab, der im neunten Jahrhundert lebte: Abu Ja'far Mohammed ibn Musa *al-Khowarizmi*<sup>1</sup>. Um einen ersten Einstieg in die Thematik zu haben kann man sich einen Algorithmus als eine Vorschrift vorstellen, die exakt angibt, wie man ein Problem sukzessive durch Anwendung wohldefinierter Einzelschritte lösen kann. Es gibt viele Möglichkeiten einen Algorithmus anzugeben, zum Beispiel, um ihn umgangssprachlich einer anderen Person mitzuteilen, oder aber auch in Form eines Programms einem Rechner zur Bearbeitung vorzugeben. Einige dieser Möglichkeiten werden im Weiteren vorgestellt, dazu zählt wie gesehen unter anderem die (exakte) umgangssprachliche Formulierung oder aber die Formulierung in einer Programmiersprache. Ein Algorithmus an sich, das heißt die Handlungsvorschrift, sollte jedoch (bis auf wenige Ausnahmen) unabhängig von einer konkreten Programmiersprache oder etwa Rechner-Hardware formulierbar sein, möglichst also allgemein formulierbar sein. Ähnliches kann man auch in unserer Umwelt beobachten, wo zum Beispiel eine Bedienungsanleitung eines Gerätes in verschiedenen Sprachen geschrieben sein kann, die beschriebene Handlungsweise an sich aber unabhängig von der konkreten Sprache ist.

## 2.1 Algorithmus

### Definition 2.1 (Algorithmus):

Ein **Algorithmus** ist ein Verfahren/Arbeitsanweisung zur systematischen und schrittweisen Lösung einer Klasse gleichartiger Probleme. ◆

Eine Algorithmus beschreibt also eine Abfolge von Einzelschritten, die wiederum in ihrer Bedeutung unmissverständlich sind. Wichtig ist auch, dass Algorithmen im Normalfall nicht nur zur Lösung genau eines spezifischen Problems entwickelt werden, sondern dass man durch eine Verallgemeinerung der Problemstellung und des Lösungsansatzes ein gegebenes konkretes, aber auch gleichzeitig viele gleichartige Probleme damit lösen kann.

---

<sup>1</sup>In der Literatur erscheint der Name oft auch in anderen Schreibweisen.

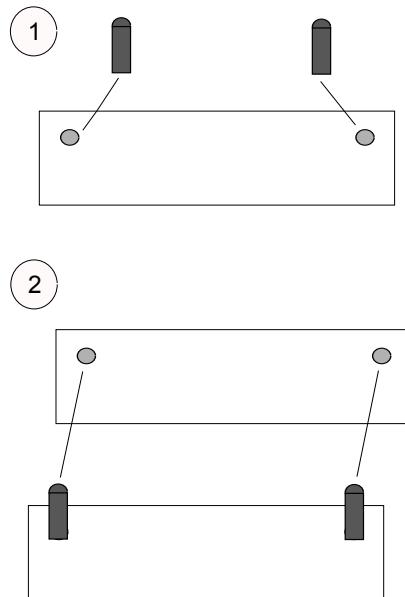


Figure 2.1: Beispiel einer grafischen Handlungsanweisung aus einer Montageanleitung eines Möbelstücks.

Algorithmen lassen sich dabei auf verschiedene Weise angeben: als umgangssprachliche Beschreibung, als Programm, unter Nutzung eines mathematischen Formalismus, als Schemazeichnung und weiteres. Wir kommen darauf in Kapitel 2.4 zurück.

Sie kennen sicherlich Beispiele für Handlungsanweisungen aus dem Alltagsleben. Ein Beispiel für solch eine Handlungsanweisung ist die Aufbauanleitung eines Möbelregals, dass etwa in einer Abfolge von Pictogrammen die einzelnen zu tätigen Schritte in grafischer Form darstellt (Abbildung 2.1). Weitere Beispiele für textuelle Handlungsanweisungen sind nachfolgend aufgeführt.

### **Beispiel 2.1:**

Auszug aus dem Deckblatt einer Informatikklausur:

1. Prüfen Sie die Vollständigkeit Ihres Exemplars. Jede Klausur umfasst diese Hinweise sowie 9 Aufgaben- und Lösungsblätter auf den Seiten 1 bis 9.
2. Tragen Sie auf jedem Aufgaben-/Lösungsblatt oben an der vorgesehenen Stelle Ihre Matrikelnummer ein. Blätter ohne diese Angaben werden nicht bewertet.
3. Unterschreiben Sie dieses Deckblatt.
4. ...

Die Beschreibung legt fest, welche Einzelaktionen (Prüfen auf Vollständigkeit, Eintragen Matrikelnummer, Unterschrift) in welcher Reihenfolge (nämlich nacheinander) ausgeführt werden sollen. In diesem Beispiel sind es allerdings voneinander unabhängige Einzelaktionen, die man auch in einer anderen Reihenfolge hätte ausführen können, ohne dass sich das Gesamtergebnis geändert hätte. ♦

### **Beispiel 2.2:**

Anleitung zu einem Handmixer:

1. ...
2. Bevor Sie den Handmixer einsetzen, stellen Sie sicher, dass die Rührschüssel auf einer rutschfesten und ebenen Unterlage steht.
3. Geben Sie nun die gewünschten Zutaten in eine Rührschüssel.
4. Halten Sie den ausgeschalteten Handmixer mit den Rührbesen oder Knethaken in die Zutaten.
5. Schalten Sie den Handmixer auf Stufe 1. Beginnen Sie immer mit einer langsamen Geschwindigkeit und erhöhen Sie bei Bedarf erst, wenn die Zutaten vermischt sind. So vermeiden Sie, dass lose Zutaten herausgeschleudert werden.

In dieser Handlungsanweisung ist im Gegensatz zum ersten Beispiel die Reihfolge der Aktionen sehr wichtig, da sie aufeinander aufbauen. Das Rühren im letzten Schritt macht nun mal keinen Sinn, wenn vorher keine Zutaten in die Schüssel gegeben wurden. Weiterhin wird im letzten Schritt die Aktion so beschrieben, dass sie von einer bestimmten Situation oder Bedingung gesteuert wird (erhöhen erst, wenn Zutaten vermischt sind). ♦

### **Beispiel 2.3:**

Aus einer Anleitung zur Einnahme eines Arzneimittels:

1. Nehmen Sie morgens und abends jeweils vor dem Essen 10 Tropfen.
2. Wiederholen Sie dies 5 Tage lang.

In diesem Beispiel soll eine Einzelaktion (Tropfen morgens/abends einnehmen) eine bestimmte und feste Anzahl mal durchgeführt werden (5 mal). ♦

### **Beispiel 2.4:**

In einem weiteren Beispiel soll zu zwei natürlichen Zahlen  $x, y \in \mathbb{N}$  mit  $x, y$  nicht gleichzeitig 0 der größte gemeinsame Teiler (ggT) gefunden werden. Seit über 2000 Jahren ist dazu ein Verfahren bekannt.

- Formulierung 1:

```

2-1 ggT(x,y):
2-2 falls x = 0 ist , dann ist y das Ergebnis
2-3 ansonsten
2-4     wiederhole , solange y ungleich 0 gilt
2-5         falls x > y ersetze x durch x - y
2-6         ansonsten ersetze y durch y - x
2-7     x ist das Ergebnis

```

- Formulierung 2:

$$ggT(x,y) = \begin{cases} y & \text{falls } x = 0 \\ x & \text{falls } y = 0 \\ ggT(x-y,y) & \text{falls } x > y \\ ggT(x,y-x) & \text{falls } x < y \end{cases}$$

Eine Beispiel zur Berechnung des ggT für  $x = 16, y = 10$  nach der letzten Formulierung ist:  $ggT(16, 10) = ggT(6, 10) = ggT(6, 4) = ggT(2, 4) = ggT(2, 2) = ggT(2, 0) = 2$ . ♦

Zu den beiden Beschreibungen des letzten Verfahrens sind einige Dinge anzumerken. Es werden in den Formulierung **Variablen** benutzt, wie sie auch in aus der Schulmathematik bekannt sind (zum Beispiel beim Umgang mit Funktionen). Sie sind Platzhalter für konkrete Werte. Weiterhin treten gewisse wiederkehrende Grundbausteine in beiden Formulierungen auf, die durch die Wörter **falls**, **wiederhole** oder **ersetze** eingeleitet werden. Durch einige dieser Wörter wird die Reihenfolge der Ausführung von Einzelanweisungen beeinflusst.

### **Beispiel 2.5:**

Eine weitere Problemstellung ist es, das Alter der ältesten Person in einem Raum zu bestimmen. Ein mögliches Verfahren dazu ist etwa:

- Wähle eine beliebige Person im Raum aus und frage nach dem Alter. Dieses Alter ist das höchste Alter aller bisher befragten Personen.
- Markiere diese Person als befragt (wie auch immer: Kreuzchen auf die linke Hand, in die hintere Ecke des Raums stellen, ...).
- Wiederhole, solange es noch ungefragte Personen im Raum gibt:
  - Wähle eine beliebige ungefragte Person aus
  - Frage nach dem Alter
  - Falls dieses Alter höher als das bisher bestimmte höchste Alter ist, so ist dieses Alter das neue höchste Alter

◆

Hierbei werden mögliche Probleme ignoriert, dass etwa jemand sein Alter nicht nennen will, Personen den Raum verlassen/hinzukommen, ...

### **Beispiel 2.6:**

Aus einer Meldung des Heise-Newstickers:

Google will mit Software Mitarbeiter-Abwanderung stoppen

Der Suchmaschinen-Spezialist Google will einer drohenden Abwanderung wichtiger Mitarbeiter zuvorkommen – mit Hilfe einer speziellen Software. Das Unternehmen sammle derzeit Daten von Mitarbeitern wie Ausbildungsstand und Gehaltsentwicklung, berichtet das Wall Street Journal heute. Mit einem speziellen Algorithmus solle aus der Datenmenge ermittelt werden, welche der 20.000 Mitarbeiter sich zum Beispiel unterfordert fühlen könnten und deshalb möglicherweise offen für neue Herausforderungen sind. Der Algorithmus helfe Google über Wechselwillige Bescheid zu wissen, bevor die Betroffenen selbst wüssten, dass sie wechseln wollen, zitiert die Zeitung Googles Personalchef Laszlo Bock. ...

◆

## 2.2 Algorithmische Grundbausteine

In vielen Fachdisziplinen wie Mathematik, Maschinenbau, Medizin, Jura, ... gibt es häufig wiederkehrende Muster, die als Bausteine zur Lösung komplizierter Zusammenhänge genutzt werden können. Für sehr häufig genutzte Bausteine gibt es dann sogar eigene Symbole. Beispiele aus der Mathematik sind etwa  $\Sigma$  für Summen,  $\Pi$  für Produkte oder  $\int$  für Integrale.

Solche Grundbausteine gibt es auch zur Beschreibung von Algorithmen. Diese gleich aufgezählten Bausteine sind in allen gängigen Programmiersprachen, oft sogar mit der gleichen Bezeichnung verfügbar. Im Folgenden

findet eine Beschränkung uns auf Grundbausteine zur Steuerung der Abfolge von Anweisungen statt. Der zentrale Begriff dabei ist die **Anweisung**. Nachfolgend wird eine Anweisung durch eine Aufzählung der Möglichkeiten definiert. Es sei angemerkt, dass dies eine vereinfachte Notation ist und nicht exakt die, die in der Programmiersprache Java verwandt wird.

In der folgenden Beschreibung tritt mehrfach der Begriff **Ausdruck** auf, der später noch im Detail behandelt wird. Bis dahin kann ein Ausdruck als eine Berechnungsformel angesehen werden, die bei Auswertung (Ausrechnen der Formel) genau einen Wert liefert, den Wert des Ausdrucks.

### 2.2.1 Einzelanweisung

An dieser Stelle soll die genau Definition einer Einzelanweisung offen gelassen werden. Eine **Einzelanweisung** ist ein Einzelschritt (siehe Definition Algorithmus), der in seiner Bedeutung unmissverständlich ist. Das kann etwa eine umgangssprachliche Beschreibung sein, eine Berechnungsformal oder gar eine Zeichnung.

Eine Einzelanweisung ist eine Anweisung.

Die Bedeutung einer Einzelanweisung ist die Bedeutung, die sich aus der Ausführung der Einzelanweisung ergibt (röhre mit dem Knethaken, ...)

#### Beispiel 2.7:

- Ersetze x durch x-y (Andere Notationen dafür:  $x=y$ ;  $x:=y$ ;  $x \leftarrow y$ )
- Frage nach dem Alter
- x ist das Ergebnis
- Man nehme 200 g Mehl
- Morgens eine halbe Tablette nehmen



### 2.2.2 Sequenz

Hat man mehrere Anweisungen  $A_1, \dots, A_n$ , so lässt sich daraus eine Sequenzanweisung oder kurz **Sequenz** angeben:

$A_1$

$A_2$

...

$A_n$

oder auch in der Notation  $A_1; A_2; \dots; A_n$ .

Eine Sequenz ist wieder eine Anweisung.

Die Bedeutung einer Sequenz ist, dass jede der Anweisungen der Sequenz genau einmal und nacheinander ausgeführt wird. Also zuerst  $A_1$ , dann  $A_2, \dots$  und zuletzt  $A_n$ . Ist die letzte Anweisung  $A_n$  beendet, so ist auch die Sequenz beendet. Man spricht auch von einer **sequentiellen Ausführung**.

An dieser Stelle sei angemerkt, dass jedes  $A_i$  eine Anweisung ist, also auch selber wieder eine Sequenz, aber auch jeder der noch nachfolgend definierten weiteren Möglichkeiten für eine Anweisung.

#### Beispiel 2.8:

Im Klausurbogenbeispiel kam vor:

1. Prüfen Sie die Vollständigkeit Ihres Exemplars. Jede Klausur umfasst diese Hinweise sowie 9 Aufgaben- und Lösungsblätter auf den Seiten 1 bis 9.
2. Tragen Sie auf jedem Aufgaben-/Lösungsblatt oben an der vorgesehenen Stelle Ihre Matrikelnummer ein. Blätter ohne diese Angaben werden nicht bewertet.
3. Unterschreiben Sie dieses Deckblatt. ♦

### 2.2.3 Selektion

Hat man eine Anweisungen  $A$  und eine Bedingung  $B$  (ein logischer Ausdruck, der bei Auswertung **wahr** oder **falsch** ergibt), so kann man damit eine **Selektion** formulieren:

**if  $B$  then  $A$**

Eine Selektion ist wieder eine Anweisung.

Die Bedeutung dieser Selektion ist, dass zuerst die Bedingung  $B$  ausgewertet wird. Ergibt sich daraus der Wert **wahr**, so wird anschließend die Anweisung  $A$  ausgeführt, wonach auch die gesamte Selektionsanweisung komplett ausgeführt ist. Ergibt sich aus der Bedingung jedoch der Wert **falsch**, so ist die Ausführung der Selektion sofort nach diesem Test beendet.

Eine Variante dazu eine Selektion mit zwei Anweisungen  $A_1$  und  $A_2$ :

**if  $B$  then  $A_1$  else  $A_2$**

Die Bedeutung dieser Variante ist, dass wiederum zuerst die Bedingung ausgewertet wird. Ist das Ergebnis **wahr**, so wird Anweisung  $A_1$  anschließend ausgeführt, ansonsten die Anweisung  $A_2$ . Also genau einer der beiden Anweisungen wird ausgeführt, in Abhängigkeit der Bedingung.

#### Beispiel 2.9:

Aus den obigen Beispiel in der neuen Notation (if-then-else):

- **if** dieses Alter höher als das bisher bestimmte höchste Alter ist **then** ist dieses Alter das neue höchste Alter
- **if**  $x > y$  **then** ersetze  $x$  durch  $x - y$  **else** ersetze  $y$  durch  $y - x$  ♦

### 2.2.4 Mehrfachselektion

Manchmal soll nicht nur in einer ja/nein-Entscheidung zwischen zwei Fällen unterschieden werden, sondern eine mehrfache Fallunterscheidung durchgeführt werden, eine **Mehrfachselektion**:

```
switch Ausdruck :
  case  $x_1$ :  $A_1$ ;
  case  $x_2$ :  $A_2$ ;
  ...
  case  $x_n$ :  $A_n$ ;
end switch
```

Eine Mehrfachselektion ist wieder eine Anweisung.

Die Bedeutung einer solchen Mehrfachselektion ist wie folgt. Zuerst wird der Ausdruck ausgewertet. Danach wird dieser Wert der Reihe nach mit den Werten  $x_1, \dots, x_n$  verglichen. Bei der ersten Übereinstimmung, zum Beispiel beim Wert  $x_i$ , so wird die Anweisung  $A_i$  ausgeführt, womit die Ausführung der Mehrfachselektion beendet ist. Stimmt der Wert des Ausdrucks mit keinem  $X_i$  überein, so wird die Mehrfachselektion beendet (ohne dass eine der Anweisungen  $A_j$  ausgeführt worden ist).

### Beispiel 2.10:

Es wird in diesem Beispiel angenommen, dass der Ausdruck **Handy-Hersteller** den Hersteller des Handys liefert, auf dem der Algorithmus ausgeführt wird.

```
switch Handy-Hersteller :
  case Nokia: nehme Batterietyp 1;
  case HTC: nehme Batterietyp 2;
  case Samsung: nehme Batterietyp 3;
end switch
```



### 2.2.5 Iteration

Oft hat man eine Vielzahl von Daten, mit denen man jeweils eine bestimmte Aktion ausführen möchte. Im Beispiel zu Bestimmung des Alters der ältesten Person im Raum muss man wiederholt eine der ungefragten Personen fragen, was ihr Alter ist. Dazu formuliert man allgemein für eine Abbruchbedingung  $B$  und zu wiederholende Anweisung  $A$ :

**while**  $B$  **do**  $A$

Im Beispiel also:

```
while ungefragte Personen im Raum sind do
  Frage eine ungefragte Person nach dem Alter
```

Ein algorithmischer Grundbaustein der oben beschriebenen Art nennt man **Iteration**.

Eine Iteration ist eine Anweisung.

Die Bedeutung ist wie folgt. Zuerst wird die Bedingung ausgewertet. Ist dieser Wert **falsch**, so ist damit die Iteration beendet. Ansonsten – der Wert des Ausdrucks war **wahr** – wird die Anweisung einmal ausgeführt. Danach wird wieder die Bedingung ausgewertet, im Fall von **falsch** bricht die SDelektion ab, im Fall von **wahr** wird die Anweisung  $A$  ausgeführt usw.

Im Normalfall ist natürlich wichtig, dass das Abbruchkriterium irgendwann einmal erfüllt sein muss (also die Auswertung der Bedingung **falsch** ergibt), ansonsten (was meist ungewollt ist und dann einen Fehler im Algorithmusentwurf darstellt) spricht man von einer **Endlosschleife**, die niemals abbricht.

### Beispiel 2.11:

```
while y ungleich 0 do
  falls x > y then ersetze x durch x - y
  else ersetze y durch y - x
```



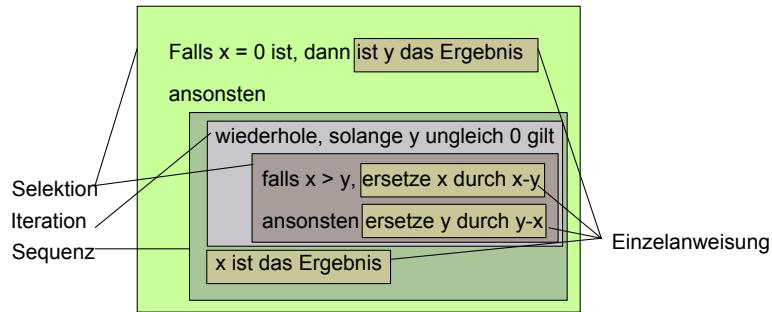


Figure 2.2: Anweisungsstruktur des ggT-Beispiels.

Schaut man sich das gesamte Beispiel der ggT-Berechnung unter dem Aspekt dieser algorithmischen Grundbausteine an, so kann man die geschachtelte Struktur der Anweisungen wiedererkennen (siehe Abb. 2.2). Die zweite Anweisung der äußeren Selektion ("Falls  $x = 0$  ist dann ... ansonsten ...") ist etwa eine Sequenz, bestend aus einer Iteration als erste Anweisung gefolgt von einer Einzelanweisung als zweiter Anweisung der Sequenz.

## 2.3 Modularität bei komplexeren Lösungen

Werden Software-Systeme komplexer oder treten gleichartige Teilaufgaben auf, so bietet es sich eine Strukturierung der Gesamtaufgabe in Teilaufgaben an. Aus der Mathematik ist diese Idee in Form von Funktionen bekannt, die über die Nutzung von Parametern flexibel verwendet werden können. Beispiel:  $f(x) = x^2$ .

Diese als **Module** bezeichneten Teillösungen sind in sich abgeschlossene Einheiten, die nicht notwendigerweise im Zusammenhang mit Lösungen anderer Detailprobleme stehen müssen. Im Handy-Beispiel ist der Algorithmus zum Auffinden von Bernd im Telefonbuch unabhängig vom Algorithmus, der das Handy ausschaltet. Insofern stellt sich hier auch die Frage, ob diese in sich abgeschlossenes Modul nicht auch an anderer Stelle genutzt werden können (ein anderes Detailproblem des gleichen Gesamtproblems oder auch ein Detailproblem eines anderen Gesamtproblems) oder bereits existierende Module für ein Detailproblem eingesetzt werden können.

Module kann man sich als Lego-Bausteine vorstellen. Für ein bestimmtes Detailproblem braucht man einen rechteckigen Baustein mit 8 Noppen, für ein anderes Detailproblem einen flachen Baustein mit 4 Noppen. Um für ein Detailproblem eine fertige Komponente einsetzen zu können, muss diese den Anforderungen des Detailproblems genügen (auf das Lego-Modell übertragen: die Anzahl der Noppen muss zum Beispiel übereinstimmen). Diese Anforderungen stellen die **Schnittstellenspezifikation** dar.

### Beispiel 2.12:

Problemspezifikation: Berechne den Flächeninhalt der beiden Rechtecke  $a$  und  $b$ , wobei die Längen jeweils 10 m (a) und 20 m (b) sind sowie die Breiten 10 m (a) und 5 m (b).

Lösung:

- 1) Berechne Flächeninhalt von Rechteck  $a$  über  $L \cdot ange \cdot Breite$ .
- 2) Berechne Flächeninhalt von Rechteck  $b$  mit  $L \cdot ange \cdot Breite$ .

Wie man einfach sieht, ist der Algorithmus zur Berechnung der Fläche für beide Rechtecke gleich. Es bietet sich also an, einen allgemeinen Algorithmus zu entwerfen, der den Flächeninhalt beliebiger Rechtecke anhand der Angaben zu Länge und Breite bestimmt. Die Schnittstellenspezifikation solch einer

Listing 2.1: Verfahren zur Berechnung des größten gemeinsamen Teilers.

```

2-1 falls x = 0 ist , dann ist y das Ergebnis
2-2 ansonsten
2-3   wiederhole , solange y ungleich 0 gilt
2-4     falls x > y ersetze x durch x - y
2-5     ansonsten ersetze y durch y - x
2-6 x ist das Ergebnis

```

Flächenberechnungskomponente wäre also, dass zwei Zahlen angegeben werden (Länge und Breite) und das Resultat der Flächeninhalt ist. Die Schnittstellenbeschreibung ist eine exakte Angabe der Argumente und des/der Resultate. ♦

Zur Spezifikation solcher wiederverwendbaren Module wollen wir folgende Notation verwenden:

**Modul** *Modulname*(*Parametername*<sub>1</sub>, ..., *Parametername*<sub>n</sub>)

Algorithmus für Modul (**Rumpf** )

**Modulende**

Unter dem Modulnamen ist dieses Modul anderswo referenzierbar, das heißt (mehrfach) verwendbar. Genauso wie man einer mathematischen Funktion konkrete Argumentwert über gibt, für das ein Resultatwert berechnet werden soll (zum Beispiel oben:  $f(2)$ ), kann man auch zu Modulen Parameter angeben. Und genauso wie bei mathematischen Funktionen kann ich durch Verwendung eines Parameternamens im Rumpf des Moduls auf diesen Parameter Bezug nehmen.

### Beispiel 2.13:

**Modul** *Fl"achenberechnung*(*L"ange*,*Breite*)

Flächenberechnung = Länge · Breite

**Modulende**

Algorithmus:

- 1) Flächeninhalt von Rechteck  $a = \text{Flächenberechnung}(10,10)$ .
- 2) Flächeninhalt von Rechteck  $b = \text{Flächenberechnung}(20,5)$ . ♦

Die Parameternamen in der Modulspezifikation (im Beispiel *L"ange*,*Breite*) nennt man **formale Parameter**, die Parameterangaben beim Aufruf des Moduls (im Beispiel 10, 10 beziehungsweise 20, 5) nennt man **aktuelle Parameter**.

## 2.4 Darstellungsmöglichkeiten von Algorithmen

An dieser Stelle sollen nun prinzipielle Darstellungs-/Beschreibungsmöglichkeiten diskutiert werden, die für vom Umfang überschaubare Algorithmen verbreitet sind. Für große Software-Systeme gibt es eigene Darstellungformen, auf die hier nur sehr beschränkt eingegangen wird (Kapitel 2.4.3).

Als durchgängiges Beispiel wird die Berechnung des größten gemeinsamen Teiler in den verschiedenen Darstellungsformen angegeben. Zur Erinnerung ist in Listing 2.1 nochmals das Verfahren beschrieben.

### 2.4.1 Umgangssprachliche Formulierung

Aus dem täglichen Leben kennt man bereits Vorschriften, wie bestimmte Tätigkeiten auszuführen sind. Beispiele:

- 1) Nehmen Sie drei mal täglich 20 Tropfen der Medizin XYZ vor dem Essen.
- 2) Kochrezept: Man nehme 500 g Mehl, siebe das Mehl in eine Schüssel und bilde eine Vertiefung in der Mitte des Mehlhügels ...
- 3) Stricken: zwei links, eins fallen lassen
- 4) Musik: Notenblatt
- 5) Spielregel: ... wenn eine sechs gewürfelt wird, darf noch einmal gewürfelt werden ...
- 6) Pythagoreischer Lehrsatz:  $a^2 + b^2 = c^2$ .
- 7) Für eine gegebene Variable  $a$  wiederhole Schritt 1 gefolgt von Schritt 2 solange, bis  $a$  den Wert 0 hat.  
Schritt 1: Addiere 1 zu  $a$ . Schritt 2: Subtrahiere 1 von  $a$ .

Schaut man sich diese Vorschriften an, so erkennt man einige Gemeinsamkeiten (siehe dazu auch Kapitel 2.2). Die Gesamtvorschrift besteht aus einer Folge von Einzelanweisungen, die miteinander kombiniert werden. Für die Einzelanweisungen gilt, dass klar definiert sein muss, was die *Bedeutung der einzelnen darin vorkommenden Begriffe* ist. Es muss also klar sein, was mit "Würfel" exakt gemeint ist und was exakt die Bedeutung von "würfeln" ist. Ist dies nicht klar, so müsste man dies zusätzlich präzisieren. Wer schon einmal die Aufbauanweisung eines Möbelstücks in den Händen hatte, weiß, dass dies nicht selbstverständlich ist und zu Problemen führen kann. Die Ausführungsreihenfolge der Anweisungen kann direkt der Reihenfolge im Text entsprechen, sie kann aber auch von Bedingungen abhängig gemacht werden (*wenn eine Sechs gewürfelt wird, darf noch einmal gewürfelt werden*; eine Wenn-Dann-Regel).

Die Ausführung der Handlungsvorschriften geschieht auf Objekten, wie zum Beispiel dem Würfel oder den Medizintropfen (**Eingabe**). Durch die Ausführung der Handlungsanweisungen werden Objekte verändert oder neu erzeugt, wie im Fall des Kuchenrezeptes (**Ausgabe**).

In den Beispielen ist die *Reihenfolge der auszuführenden Schritte* ebenfalls exakt beschrieben. Können mehrere Schritte gleichzeitig ausgeführt werden, so spricht man von **paralleler** oder **nebenläufiger Ausführung**. Beispiel aus einem Backrezept: *Während der Kuchenboden von einem Koch angefertigt wird, beginnt der zweite Koch mit der Zubereitung des Belags*.

Die Handlungsvorschrift ist von *endlicher Länge*. Ein Beispiel für eine unendliche Vorschrift wäre etwa folgendes Kinderlied:

Ein Mops kam in die Küche und stahl dem Koch ein Ei.  
 Da nahm der Koch den Löffel und schlug den Mops entzwei.  
 Es kamen viele Möpse und gruben ihm ein Grab  
 und setzten einen Grabstein, auf dem geschrieben stand:  
  
 Ein Mops kam in die Küche und stahl dem Koch ein Ei.  
 Da nahm der Koch den Löffel und schlug den Mops entzwei.  
 Es kamen viele Möpse und gruben ihm ein Grab  
 ...

Ein Beispiel für eine Vorschrift, das zwar endlich beschrieben werden kann, aber bei seiner Ausführung für alle Eingabewerte ungleich Null *niemals abbricht*, ist das letzte der obigen Beispiele.

Im Beispiel mit dem Pythagoreischen Lehrsatz ist zu sehen, dass die Handlungsvorschrift auch so formuliert sein kann, dass sie auf eine ganze Klasse von Problemen angewandt werden kann. Während die Ägypter diesen Satz offenbar nur für  $a = 3, b = 4, c = 5$  kannten und ihn (wohl schon um 3000 v.Chr.) zur jährlichen Neuvermessung der Felder nach der periodischen Überschwemmung des unteren Niltals einsetzen, kannten dazu im Gegensatz die Babylonier und später die Inder den Satz in der allgemeinen Form, die  $a = 3, b = 4, c = 5$  als Spezialfall enthält.

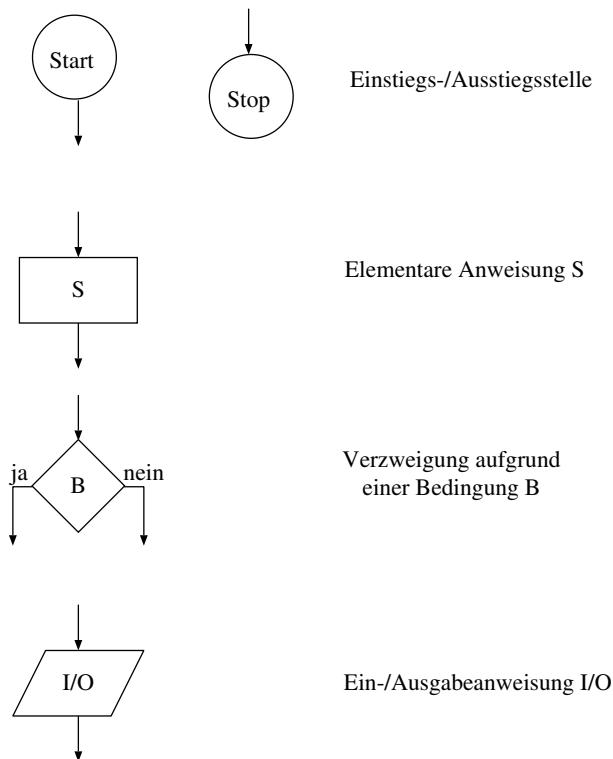


Figure 2.3: Basiselemente von Programmablaufplänen.

Der gravierende Nachteil von umgangssprachlichen Beschreibungen (für nicht-triviale Algorithmen) ergibt sich durch die Möglichkeit der Mehrdeutigkeit von Formulierungen in menschlicher Sprachen, also etwa durch die unterschiedliche Interpretation eines Satzes durch mehrere Menschen.

## 2.4.2 Programmablaufplan

Eine (ältere) Möglichkeit der graphischen Darstellung von Algorithmen ist die Nutzung von **Programmablaufplänen (PAP)**, die im Wesentlichen den Kontrollfluss, also die Abfolge von Schritten, in einem Algorithmus widerspiegeln. Ein weiterer Name dafür ist **Flussdiagramm**. Der **Kontrollfluss** spiegelt die Reihenfolge der auszuführenden Elementarschritte wider. Abbildung 2.3 zeigt die graphischen Basiselemente von Programmablaufplänen auf. Jedes solche Basiselement bis auf das Startelement besteht aus genau einem Eingangspfeil, einer durchzuführenden Aktion und keinem, einem oder zwei Ausgangspfeilen. Die durchzuführende Aktion wird auch Knoten genannt, die Pfeile auch Kanten. Aus diesen Basiselementen lassen sich komplexere Programmablaufpläne angeben, indem eine ausgehende Kante eines Elementes mit einer eingehenden Kante eines anderen Elementes verbunden wird. Es muss genau einen Startknoten geben und mindestens einen Stopknoten. Programmablaufpläne sind in den Normen DIN 66001 und ISO 5807 beschrieben.

Die Bedeutung eines Programmablaufplans ist wie folgt: Ausgehend von dem eindeutigen Startknoten folgt man den Kanten. Beim Durchlauf eines Knotens führt man die darin spezifizierte Aktion aus. In einem Knoten mit einer Elementaranweisung führt man diese Elementaranweisung aus (zum Beispiel addiere x zu y). Trifft man auf einen Verzweigungsknoten, überprüft man die darin angegebene Bedingung und folgt entweder der mit "ja" beziehungsweise der mit "nein" markierten Ausgangskante. Ein Ein-/Ausgabeknoten dient der visuellen Markierung von Eingaben an den Algorithmus sowie Ausgaben des Algorithmus. Dieses Vorgehen entlang von Kanten setzt man solange fort, bis man einen Stopknoten erreicht.

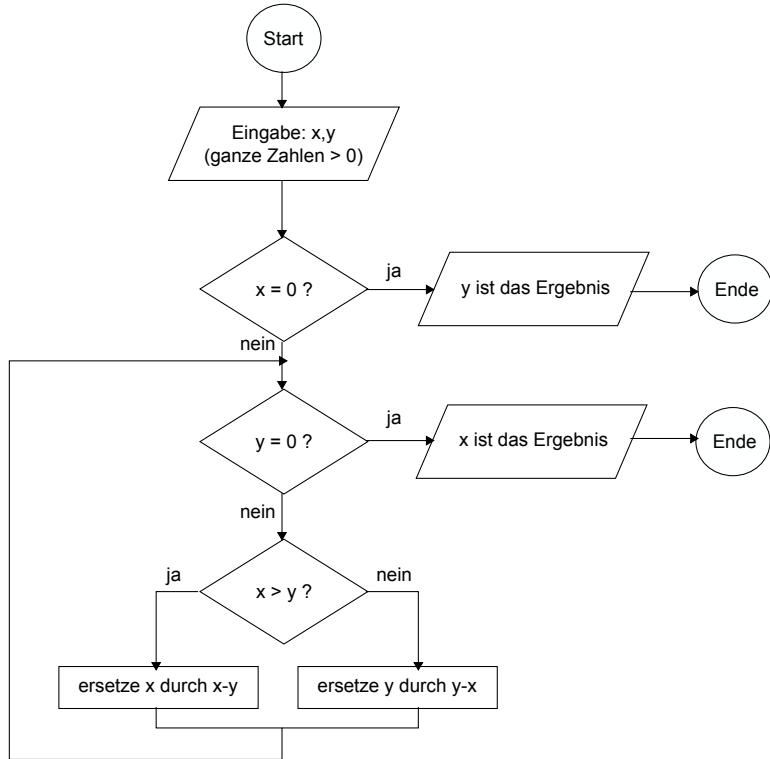


Figure 2.4: ggT-Algorithmus in Form eines Programmablaufplans.

Abbildung 2.4 zeigt einen Programmablaufplan, der den obigen ggT-Algorithmus darstellt. Beginnend beim Startknoten erwartet man zuerst als Eingabe zwei Zahlen  $x, y \in \mathbb{N}$ . Dann wird überprüft, ob der Wert von  $x$  gleich 0 ist. In diesem Fall (Abzweigung rechts wird genommen) ist der Wert von  $y$  und der Algorithmus bricht mit der Ende-Anweisung ab. Ansonsten wird der Wert von  $y$  überprüft usw.

Anhand des Beispiels wird ein gravierender Nachteil von Programmablaufplänen klar: Algorithmen werden in dieser Darstellung oft unübersichtlich, weil insbesondere der häufig auftretende Grundbaustein der Iteration nicht mehr explizit in einem Flussdiagramm erscheint, sondern implizit durch Selektion und Pfeile simuliert wird (folgt man den Pfeilen, entsteht eine Schleife, man kommt zur gleichen Stelle im Diagramm zurück). Programmablaufpläne sind eher dort sinnvoll einsetzbar, wo es im Wesentlichen auf die einfache Abfolge weniger Aktionen ankommt.

### 2.4.3 Aktivitätsdiagramm der Unified Modeling Language

Die **Unified Modeling Language** (UML) bietet sehr umfangreiche Möglichkeiten der Darstellung von (insbesondere sehr großen) Software-Systemen im Allgemeinen. Dazu gehören viele verschiedene Darstellungsmöglichkeiten von Sachverhalten, die die UML bietet und mit denen Menschen Sachverhalte oder Zusammenhänge veranschaulicht werden sollen. In diesem Buch wird nur einführend auf einige Aspekte der UML eingegangen (Aktivitätsdiagramm hier und Klassendiagramm in Kapitel 13.5). Für eine umfassende Beschreibung sei auf die Primärliteratur [Gro] sowie relevante Lehrbücher verwiesen [RQZ07].

Ähnlich den Flussdiagrammen bestehen **Aktivitätsdiagramme** aus visuellen Elementen, die im Wesentlichen die Abfolge von Schritten darstellen und verdeutlichen sollen. Abbildung 2.5 zählt diese Basiselemente auf. Ein Aktivitätsdiagramm wird als ein großes Rechteck mit abgerundeten Ecken dargestellt (also selbst als Aktivität). Oben links kann man dem Algorithmus ein Namen geben. Innerhalb dieser Umrandung werden Aktionselemente wiederum durch Pfeile miteinander verbunden, die eine mögliche Abfolge darstellen. Eine

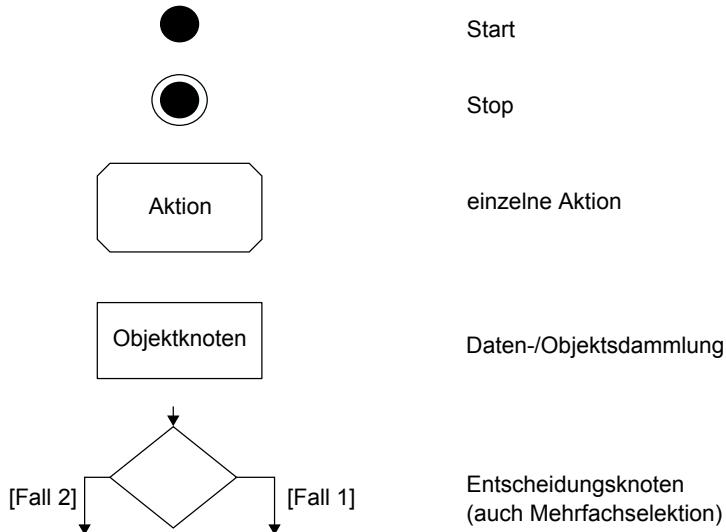


Figure 2.5: Wesentliche Basiselemente von Aktivitätsdiagrammen.

Ausführung eines Verfahrens ist ebenso analog wie beim Programmablaufplan das Durchlaufen eines solchen Diagramms beginnend beim eindeutigen Startknoten bis hin zum Erreichen eines Stop-Knotens, oder alternativ das Verlassen des Diagramms bzw. dessen Begrenzung (siehe folgendes Beispiel). Eingabedaten, die dem Verfahren zugeführt werden, werden als Datensammlung (also als Rechteck mit eckigen Kanten) dargestellt, das halb außerhalb und halb innerhalb der Diagrammmumrandung liegt und aus dem ein Pfeil abgeht. Ebenso werden Ausgabedaten analog dargestellt, nur dass ein Pfeil in das Rechteck mündet.

Abbildung 2.6 zeigt das entsprechende Aktivitätsdiagramm zum ggT-Verfahren. Beginnend beim Startknoten durchläuft man das Diagramm entlang der Kanten, bis man irgendwann der Kante folgt, die zur Datensammlung rechts führt, von wo eine weitere Ausführung mehr möglich ist.

#### 2.4.4 Struktogramm

Die Defizite von Flussdiagrammen – fehlende Unterstützung gängiger algorithmischer Grundbausteine – versucht das Struktogramm zu beheben. Struktogramme, die auch als Nassi-Shneiderman-Diagramme bezeichnet werden (benannt nach Ike Nassi und Ben Schneiderman), bieten Grundelemente, die den üblichen Grundbausteinen und damit auch Kontrollkonstrukten von modernen Programmiersprachen entsprechen. Zu Struktogrammen gibt es ebenfalls eine eigene DIN 66261.

Ein Gesamtstruktogramm wird durch genau ein Rechteck dargestellt. Dieses Rechteck kann wiederum weitere Rechtecke enthalten, die Struktur der Schachtelung ergibt sich aus dem Aufbau des Programms. Als Strukturelemente des Programms sind gerade die bekannten Grundbausteine Sequenz, Selektion und Iteration erlaubt. Abbildung 2.7 zeigt die möglichen Elemente und deren grafische Darstellung.

##### Beispiel 2.14:

Der Euklidische Algorithmus zur Berechnung des ggT ist als Struktogramm in Abbildung 2.8 gegeben. ♦

Spätestens mit ein klein bisschen Übung / Erfahrung mit Struktogrammen sieht man sehr schnell die Struktur eines Algorithmus. Ein weiterer Vorteil ist, dass genau (nicht mehr und nicht weniger) Strukturlemente verfügbar sind, wie auch Grundbausteine vorgestellt wurden.

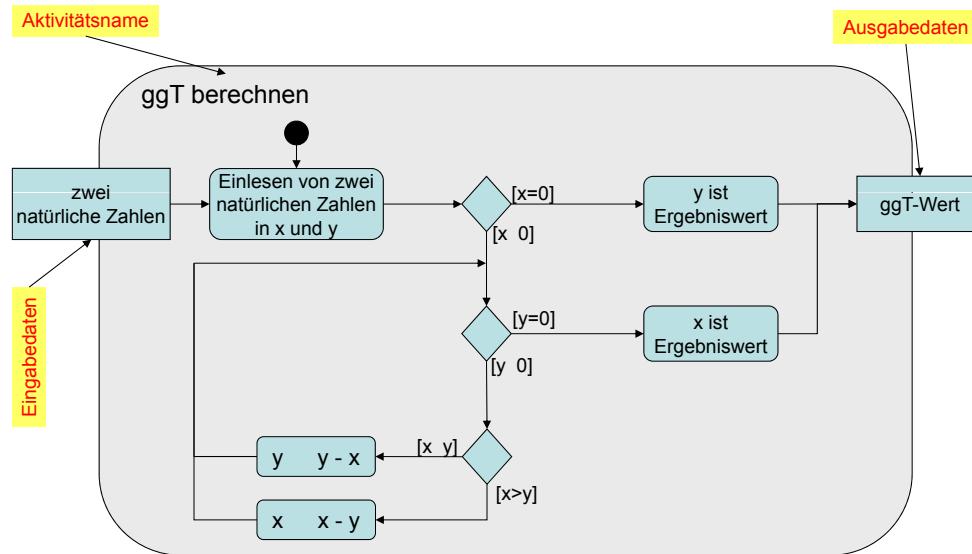


Figure 2.6: ggT-Algorithmus in Form eines Aktivitätsdiagramms.

### 2.4.5 Programm

Der ggT-Algorithmus in Form eines kompletten Java-Programms ist in Listing 2.2 zu sehen. Auf eine Diskussion der einzelnen Bestandteile des Programms wird an dieser Stelle verzichtet, da darauf an der relevanten und passenderen Stellen noch ausführlich eingegangen wird.

## 2.5 Präzisierung des Algorithmusbegriffs \*

Die nachfolgenden Kapitel haben vertiefenden Charakter und können gegebenenfalls übersprungen werden.

Prinzipiell stellt sich die Frage: Gibt es für jede Problemstellung (mindestens) einen Algorithmus, der dieses Problem löst? Diese Frage muss man verneinen. Gödel, Turing und andere Wissenschaftler haben sich mit solchen Fragestellungen in den 30er Jahren des letzten Jahrhunderts befasst und bewiesen, dass es prinzipiell unentscheidbare Probleme gibt.

1. Ein Beispiel für ein prinzipiell **unentscheidbares Problem** ist das sogenannte **Halteproblem**: Finde einen Algorithmus, der feststellt, ob ein beliebiger Algorithmus in endlicher Zeit ein Ergebnis liefert, d.h. anhält. Solche Fragestellungen werden in der Theoretischen Informatik zum Beispiel behandelt.
2. Das **Traveling Salesman Problem** ist ein Beispiel für ein Problem, das zwar prinzipiell lösbar ist, aber die Lösungsalgorithmen prinzipiell so **komplex** sind, dass für die meisten Eingabedaten die Berechnung selbst auf den schnellsten Rechnern der Welt (auch in absehbaren Zukunft) zu lange dauern würde. Die Aufgabenstellung bei dem Traveling Salesman Problem ist es, den kürzesten Rundweg zwischen einer Anzahl von Städten zu finden, die über Straßen verbunden sind. Dabei muss jede Stadt (bis

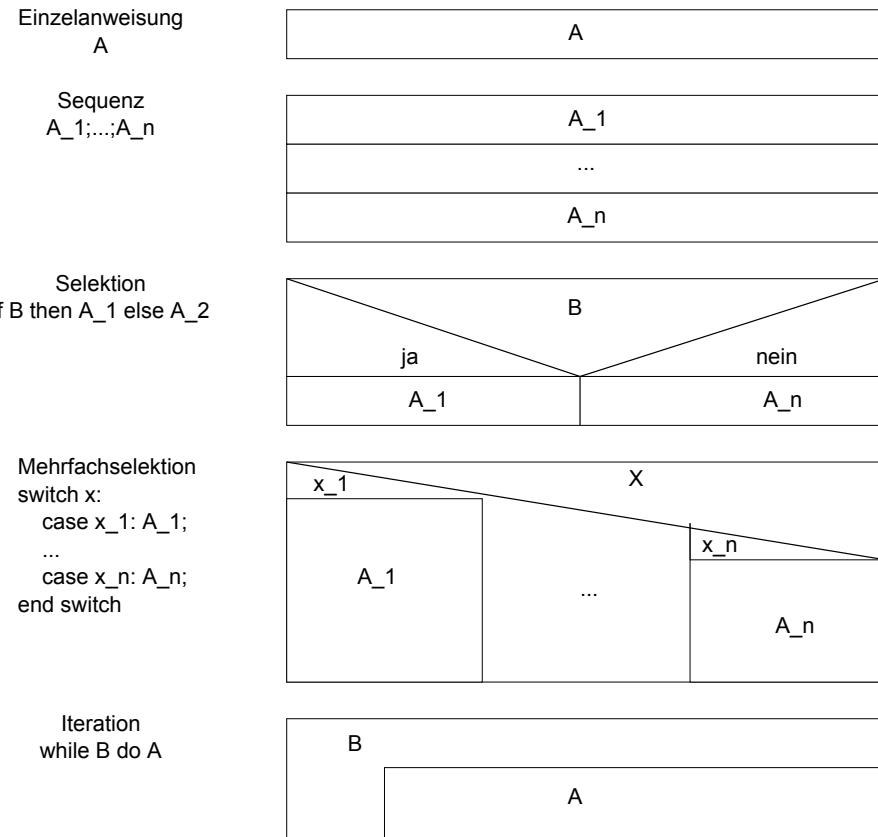


Figure 2.7: Basic elements of Structograms.

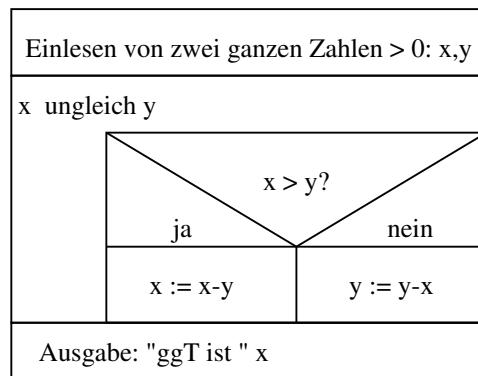


Figure 2.8: ggT-Algorithmus in Form eines Struktogramms.

Listing 2.2: ggT-Programm in Java.

```

2-1  /**
2-2   * groesster gemeinsamer Teiler ggT
2-3   */
2-4  public class ggT {
2-5      public static void main (String [] args) {
2-6          // Beispielwerte
2-7          int x = 43158;
2-8          int y = 26364;
2-9
2-10         // gebe die Werte von x und y aus
2-11         System.out.print ("Der ggT von " + x + " und " + y + " ist ");
2-12
2-13         if (x == 0) {
2-14             // gebe das Ergebnis / den ggT aus
2-15             System.out.println (y);
2-16         } else {
2-17             while (y != 0) {
2-18                 if (x > y) {
2-19                     x = x - y;
2-20                 } else {
2-21                     y = y - x;
2-22                 }
2-23             }
2-24             // das Verfahren brach ab und in x (und y) steht das Resultat
2-25             System.out.println (x);
2-26         }
2-27     }
2-28 }
```

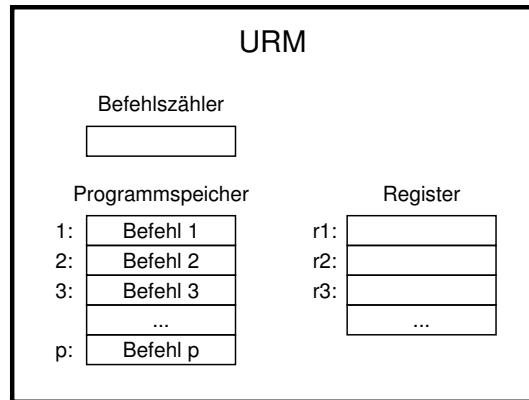


Figure 2.9: Aufbau der URM.

auf die Ausgangsstadt) genau einmal besucht werden. Ein Algorithmus ist zum Beispiel, dass man alle Kombination von Städten betrachtet (1. Fall: zuerst Stadt 1, dann Stadt 2, usw.; 2. Fall: zuerst Stadt 1, dann Stadt 3 usw.) und die entsprechenden Weglängen aufaddiert. Um den optimalen Weg zu bestimmen, muss man die Weglänge für  $n$  Fakultät ( $n!$ ) Wege ausrechnen. Betrachtet man den Verlauf der Fakultätsfunktion ist dies selbst für eine kleine Anzahl von Städten zu komplex. Beispiel: Für 3 Städte wären es 6 mögliche Wege, für 5 Städte müsste man 120 mögliche Wege betrachten, für 10 Städte wären es schon 3.628.800 mögliche Wege und für 20 Städte ca. 2.500.000.000.000.000 mögliche Wege.

### 2.5.1 Universelle Registermaschine

Um Fragestellungen wie etwa das Halteproblem beantworten (sprich: beweisen!) zu können, reicht die intuitive Definition des Algorithmus nicht aus. Vielmehr benötigt man ein formales Modell, mit dem man mathematisch exakt Eigenschaften nachweisen oder widerlegen kann. Die in diesem Abschnitt eingeführte Universelle Registermaschine URM ist ein solches formales Modell. Die URM ist ein mathematisches Modell eines sehr einfachen Rechners. Diese Maschine ist nicht dazu gedacht, reale Programme für das Alltagsleben darauf zu entwerfen. Wenn Sie die Übungsaufgaben zu diesem Kapitel lösen werden, werden Sie selbst sehen, wie aufwändig dies sein kann. Aufgrund ihrer einfachen Struktur ist man jedoch in der Lage, formal Eigenschaften nachzuweisen. Das liegt daran, dass die URM so einfach aufgebaut ist, dass man mit (relativ) einfachen Mitteln sagen kann, was diese Maschine für ein gegebenes Programm exakt macht.

Die **Churchsche These**<sup>2</sup> besagt jedoch, dass alle "vernünftigen" Definitionen eines Algorithmus (ob in URM-Sprache, Java, C, Pseudocode usw.) äquivalent sind. Man kann im Prinzip ein Java-Programm, das eine Funktion  $f : \mathbb{N}^n \rightarrow \mathbb{N}$  beschreibt, in ein URM-Programm übersetzen und es auf der URM ausführen (und umgekehrt).

Die **Universelle Registermaschine (URM)** dient zur Berechnung von Funktionen auf natürlichen Zahlen. Die Registermaschine enthält eine unendliche Zahl von **Registern**  $r_1, r_2, \dots$ , die jeweils genau einen Wert  $n \in \mathbb{N}$  aufnehmen können. Man kann sich ein Register als eine Art Schublade vorstellen, in das genau ein Wert passt. Weiterhin hat die URM einen **Programmspeicher** und einen **Befehlszähler**. Der Programmspeicher enthält eine endliche nicht-leere Folge von Befehlen  $\beta_1, \dots, \beta_p$ . Die Programmspeicherplätze können jeweils genau einen Befehl aufnehmen und sind von  $1, \dots, p$  nummeriert, d.h. der  $i$ -te Befehl steht an Position  $i$  im Programmspeicher. Der Befehlszähler enthält eine positive Zahl zur Markierung des nächsten auszuführenden Befehls.

<sup>2</sup>Eine These ist eine unbewiesene Aussage. In der Fachwelt geht man jedoch davon aus, dass die Churchsche These gilt.

Der Befehlsvorrat der URM umfasst folgende drei arithmetische Befehle (1–3) und zwei Sprungbefehle zum Steuern des Programmablaufes (4–5):

1.  $r_i \leftarrow 0 (i \in \mathbb{N}_+)$

Schreibe den Wert 0 in das Register  $i$ . Die Inhalte aller anderen Register verändern sich nicht. Der Befehlszähler wird zusätzlich um eins erhöht.

2.  $r_i \leftarrow r_i + 1 (i \in \mathbb{N}_+)$

Erhöhe den Inhalt des Registers  $r_i$  um 1: Nimm den alten Inhalt des Registers  $r_i$ , addiere auf diese Zahl eins auf und speichere die so gewonnene Zahl im Register  $r_i$  wieder ab. Die Inhalte aller anderen Register verändern sich nicht. Der Befehlszähler wird zusätzlich um eins erhöht.

3.  $r_i \leftarrow r_i - 1 (i \in \mathbb{N}_+)$

Erniedrige den Wert eines Registers  $r_i$  um eins, wenn dieser positiv ist: Nimm den alten Inhalt des Registers  $r_i$ . Falls diese Zahl echt größer null ist, subtrahiere eins von dieser Zahl und speichere den so gewonnenen neuen Wert im Register  $r_i$  wieder ab. Ansonsten (alter Inhalt war null) belasse den alten Wert in Register  $r_i$ . Die Inhalte aller anderen Register verändern sich nicht. Der Befehlszähler wird zusätzlich um eins erhöht.

4. **goto**  $l (l \in \mathbb{N}_+)$

Setze den Programmzähler auf  $l$ .

5. **if**  $r_i = 0$  **goto**  $l (i \in \mathbb{N}_+, l \in \mathbb{N}_+)$

Teste, ob der Inhalt des Registers  $r_i$  gleich null ist. Falls dies der Fall ist, setze den Programmzähler auf  $l$ , ansonsten erhöhe den Programmzähler ums eins.

Soll eine URM eine  $n$ -stellige Funktion  $f : \mathbb{N}^n \rightarrow \mathbb{N}$  berechnen, so vereinbaren wir, dass die Argumente  $x_1, \dots, x_n$  mit dem Befehl **in**  $(r_{i_1}, \dots, r_{i_n}), n \in \mathbb{N}, r_{i_j} \in \mathbb{N}_+$  in die Register  $r_{i_1}, \dots, r_{i_n}$  gespeichert werden können. Ebenso kann man mit dem Befehl **out**  $(r_i), i \in \mathbb{N}$  den Wert des Registers  $r_i$  ausgeben lassen.

Die intuitive Arbeitsweise der URM ist nun wie folgt:

1. **Eingabe:** Die Eingabedaten werden auf die in der **in**-Anweisung spezifizierten Register geschrieben, der Befehlszähler auf 1 gesetzt und alle übrigen Register auf den Wert 0 gesetzt.

2. **Ausführung des Befehlsteils:** Sei  $q$  der Wert des Befehlsregisters. Falls der  $q$ -te Befehl, d.h. der Befehl im  $q$ -ten Programmspeicher, ein arithmetischer Befehl ist, so wende diesen Befehl entsprechend an und erhöhe anschließend den Programmzähler um eins. Falls der  $q$ -te Befehl ein Sprungbefehl der Form **goto**  $l$  ist, so wird der Befehlszähler auf den Wert  $l$  gesetzt. Falls der  $q$ -te Befehl ein Sprungbefehl der Form **if**  $r_i = 0$  **goto**  $l$  ist, so wird der Befehlszähler auf den Wert  $l$  gesetzt, falls das Register  $r_i$  den Wert 0 enthält, ansonsten wird der Befehlszähler auf den Wert  $q + 1$  gesetzt.

Dieses Vorgehen wird wiederholt. Es sind dabei zwei Fälle möglich:

- (a) Nach endlich vielen Programmschritten weist der Programmzähler auf eine leere Programmspeicherstelle. In diesem Falle bricht die Berechnung ab.

- (b) Nach jeder Anwendung eines Befehles ist ein weiterer Befehl ausführbar. In diesem Falle bricht die Berechnung nicht ab.

3. Für den Fall, dass die Berechnung nicht abbricht, ist die Ausgabe nicht definiert. Für den Fall, dass die Berechnung abbricht, steht die Ausgabe in dem durch die **out**-Anweisung angegeben Register.

Es soll nun das formale Modell definiert werden, das die Arbeitsweise einer URM beschreibt. Dazu sei vereinbart, dass der Programmzähler mit  $r_0$  bezeichnet wird.

Ein **Zustand**  $z : \mathbb{N} \rightarrow \mathbb{N}$  der URM beschreibt mit:

- $z(0)$  den Inhalt des Befehlszählers
- $z(i)$  den Inhalt des i-ten Registers ( $i \in \mathbb{N}_+$ )

Der **Zustandsraum**  $\mathcal{Z}$  der URM sei definiert durch

$$\mathcal{Z} := \{z \mid z : \mathbb{N} \rightarrow \mathbb{N}\}$$

als die Menge aller Zustände der URM. Jeder URM-Befehl  $\beta$  bestimmt dann eine **Zustandstransformation**

$$[\beta] : \mathcal{Z} \rightarrow \mathcal{Z}$$

mit (Für die Klammerung  $[x]$  könnte man umgangssprachlich sagen: Die Bedeutung von  $x$  ist ...):

$$\begin{aligned} [r_i \leftarrow 0](z) &:= z' \text{ mit } z'(0) := z(0) + 1, z'(i) = z(j) \text{ für } j \neq i \\ [r_i \leftarrow r_i + 1](z) &:= z' \text{ mit } z'(0) := z(0) + 1, z'(i) = z(i) + 1, z'(j) = z(j) \text{ für } j \neq i \\ [r_i \leftarrow r_i - 1](z) &:= z' \text{ mit } z'(0) := z(0) + 1, z'(i) = z(i) - 1, z'(j) = z(j) \text{ für } j \neq i \\ [\text{goto } l](z) &:= z' \text{ mit } z'(0) := l, z'(j) = z(j) \text{ sonst} \\ [\text{if } r_i = 0 \text{ goto } l](z) &:= z' \text{ mit } z'(0) := l \text{ falls } z(i) = 0, z'(0) = z(0) + 1 \text{ falls } z(i) \neq 0, z'(j) = z(j) \text{ sonst} \end{aligned}$$

Der Befehlsteil  $b := \beta_1, \dots, \beta_p$  ( $p \in \mathbb{N}_+$ ) bestimmt eine Zustandstransformation

$$\delta(b) : \mathcal{Z} \rightarrow \mathcal{Z},$$

die Einzelschrittfunktion von  $b$  mit ( $\perp$  bedeutet **undefiniert**):

$$\delta(\beta_1, \dots, \beta_p)(z) := \begin{cases} [\beta_i](z) & \text{falls } z(0) = i \\ \perp & \text{falls } z(0) > p \end{cases}$$

### Definition 2.2 (URM-Programm):

Ein **URM-Programm**  $P$  besteht aus einer Eingabevereinbarung **in**  $(r_{i_1}, \dots, r_{i_n})$  ( $n \in \mathbb{N}$ ), einem Befehlsteil  $b = (\beta_1, \dots, \beta_p)$  ( $p \in \mathbb{N}_+$ ) und einer Ausgabevereinbarung **out**  $(r_i)$  ( $i \in \mathbb{N}_+$ ):  $P = \mathbf{in} (r_{i_1}, \dots, r_{i_n}); \beta_1, \dots, \beta_p; \mathbf{out} (r_i)$ . Dabei berechnet  $P$  eine partiell arithmetische Funktion  $[P] : \mathbb{N}^n \rightarrow \mathbb{N}$ . ♦

Für die einzelnen Komponenten eines URM-Programms gilt:

1. Für die Eingabevereinbarung gilt:

$$\begin{aligned} [\mathbf{in} (r_{i_1}, \dots, r_{i_n})] : \mathbb{N}^n &\rightarrow \mathcal{Z} \\ (x_1, \dots, x_n) &\mapsto z_0 \\ \text{mit: } z_0(0) &= 1, z_0(i_k) = x_k (k = 1, \dots, n), z_0(j) = 0 \text{ sonst} \end{aligned}$$

2. Für die iterierte Zustandstransformation des Befehlsteils  $b = \beta_1, \dots, \beta_p$ :

(a) Falls die Berechnung abbricht:

$$\delta(b)(z_0) = r_1$$

...

$$\delta(b)(z_{s-1}) = z_s \text{ mit } z_s(0) > p$$

dann gilt:

$$[b](z_0) := z_s$$

(b) Falls die Berechnung nicht abbricht, d.h. für jeden Folgezustand  $z_i$  gilt:

$$\delta(b)(z_i) = z_{i+1} \text{ mit } 1 \leq z_{i+1}(0) \leq p$$

dann gilt

$$[b](z_0) = \perp$$

3. Für die Ausgabeabbildung gilt:

$$[\mathbf{out}(r_i)] : \mathcal{L} \rightarrow \mathbb{N}$$

$$z \mapsto z(i)$$

Es gilt dann: P berechnet  $[P] : \mathbb{N}^n - \rightarrow \mathbb{N}$  mit:

$$[P] := [\mathbf{out}(r_i)] \circ [b] \circ [\mathbf{in}(r_{i_1}, \dots, r_{i_n})]$$

- bedeutet die Komposition, d.h. Hintereinanderausführung von Funktionen.  $f \circ g$  angewandt auf ein Argument  $x$  bedeutet demnach  $f(g(x))$ .

### Definition 2.3 (URM-berechenbar):

$f : \mathbb{N}^n - \rightarrow \mathbb{N}$  heißt **URM-berechenbar** genau dann, wenn ein URM-Programm P existiert mit  $[P] = f$ . ◆

### Beispiel 2.15:

Es sollen zwei Zahlen  $x_1$  und  $x_2$  addiert werden. Das Ergebnis soll in Register  $r_3$  stehen und ausgegeben werden. Da die URM keinen Befehl für eine allgemeine Addition besitzt, muss die allgemeine Addition auf die wiederholte Addition mit 1 zurückgeführt werden. Dazu wird  $r_3$  sooft inkrementiert, wie man  $r_1$  dekrementieren kann. Anschließend inkrementiert man  $r_3$  sooft, wie man  $r_2$  dekrementieren kann. Abschließend ist in  $r_3$  der Wert von  $x + y$  (beziehungsweise  $r_1 + r_2$ ).

$P = \mathbf{in}(r_1, r_2); b; \mathbf{out}(r_3)$  mit  $b =$

- 1:  $r_3 \leftarrow 0$
- 2: **if**  $r_1 = 0$  **goto** 6
- 3:  $r_1 \leftarrow r_1 - 1$
- 4:  $r_3 \leftarrow r_3 + 1$
- 5: **goto** 2
- 6: **if**  $r_2 = 0$  **goto** 10
- 7:  $r_2 \leftarrow r_2 - 1$
- 8:  $r_3 \leftarrow r_3 + 1$
- 9: **goto** 6

berechnet die Addition zweier Zahlen  $x_1$  und  $x_2$ .

$$\begin{aligned} [P](2, 1) &= [\mathbf{out}(r_3)] \circ [b] \circ [\mathbf{in}(r_1, \dots, r_2)](2, 1) \\ &= [\mathbf{out}(r_3)] \circ [b](z_0) \text{ mit: } z_0(0) = 1, z_0(1) = 2, z_0(2) = 1, z_0(j) = 0 \text{ sonst} \end{aligned}$$

$[b](z_0)$  ist definiert als die iterierte Zustandstransformation von  $\beta_1, \dots, \beta_9$  angewandt auf den Startzustand  $z_0$ . Als Kurznotation eines Zustandes wird die Tupelschreibweise benutzt: Statt  $z(0) = i, z(1) = j, z(2) = k, \dots$

schreibt man  $(i, j, k, \dots)$ . Damit ergeben sich folgende Schritte bei der Anwendung der Einzelschrittfunktion:

$$\begin{aligned}
 \delta(b)(1, 2, 1, 0, 0, \dots) &= [r_3 \leftarrow 0](1, 2, 1, 0, 0, \dots) = (2, 2, 1, 0, 0, \dots) \\
 \delta(b)(2, 2, 1, 0, 0, \dots) &= [\text{if } r_1 = 0 \text{ goto } 6](2, 2, 1, 0, 0, \dots) = (3, 2, 1, 0, 0, \dots) \\
 \delta(b)(3, 2, 1, 0, 0, \dots) &= [r_1 \leftarrow r_1 - 1](3, 2, 1, 0, 0, \dots) = (4, 1, 1, 0, 0, \dots) \\
 \delta(b)(4, 1, 1, 0, 0, \dots) &= [r_3 \leftarrow r_3 + 1](4, 1, 1, 0, 0, \dots) = (5, 1, 1, 1, 0, \dots) \\
 \delta(b)(5, 1, 1, 1, 0, \dots) &= [\text{goto } 2](5, 1, 1, 1, 0, \dots) = (2, 1, 1, 1, 0, \dots) \\
 \delta(b)(2, 1, 1, 1, 0, \dots) &= [\text{if } r_1 = 0 \text{ goto } 6](2, 1, 1, 1, 0, \dots) = (3, 1, 1, 1, 0, \dots) \\
 \delta(b)(3, 1, 1, 1, 0, \dots) &= [r_1 \leftarrow r_1 - 1](3, 1, 1, 1, 0, \dots) = (4, 0, 1, 1, 0, \dots) \\
 \delta(b)(4, 0, 1, 1, 0, \dots) &= [r_3 \leftarrow r_3 + 1](4, 0, 1, 1, 0, \dots) = (5, 0, 1, 2, 0, \dots) \\
 \delta(b)(5, 0, 1, 2, 0, \dots) &= [\text{goto } 2](5, 0, 1, 2, 0, \dots) = (2, 0, 1, 2, 0, \dots) \\
 \delta(b)(2, 0, 1, 2, 0, \dots) &= [\text{if } r_1 = 0 \text{ goto } 6](2, 0, 1, 2, 0, \dots) = (6, 0, 1, 2, 0, \dots) \\
 \delta(b)(6, 0, 1, 2, 0, \dots) &= [\text{if } r_2 = 0 \text{ goto } 10](6, 0, 1, 2, 0, \dots) = (7, 0, 1, 2, 0, \dots) \\
 \delta(b)(7, 0, 1, 2, 0, \dots) &= [r_2 \leftarrow r_2 - 1](7, 0, 1, 2, 0, \dots) = (8, 0, 0, 2, 0, \dots) \\
 \delta(b)(8, 0, 0, 2, 0, \dots) &= [r_3 \leftarrow r_3 + 1](8, 0, 0, 2, 0, \dots) = (9, 0, 0, 3, 0, \dots) \\
 \delta(b)(9, 0, 0, 3, 0, \dots) &= [\text{goto } 6](9, 0, 0, 3, 0, \dots) = (6, 0, 0, 3, 0, \dots) \\
 \delta(b)(6, 0, 0, 3, 0, \dots) &= [\text{if } r_2 = 0 \text{ goto } 10](6, 0, 0, 3, 0, \dots) = (10, 0, 0, 3, 0, \dots)
 \end{aligned}$$

Die iterierte Zustandstransformation bricht an dieser Stelle ab, so dass gilt:  $[b](z_0) = (10, 0, 0, 3, 0, \dots)$ . Setzt man dies in die ursprüngliche Berechnung ein, so gilt:

$$[\text{out } (r_3)] \circ [b](z_0) = [\text{out } (r_3)](10, 0, 0, 3, 0, \dots) = 3$$



## 2.5.2 Funktionales Berechnungsmodell

Im letzten Abschnitt wurde anhand eines abstrakten Maschinenmodells die Bedeutung (Wirkung) eines Algorithmus über die Anwendung der Einzelschrittfunktion definiert. Jeder einzelne Schritt der Berechnung bewirkte eine Zustandstransformation. Man hat noch also für die Details der "internen" Wirkung interessiert. Zudem war die Angabe eines Programms sehr aufwendig, man musste sehr detailliert die einzelnen Berechnungsschritte angeben und dabei im Kopf haben, welches Register für welchen Wert stand.

Alternativ dazu kann man auch ein funktionales Berechnungsmodell zugrunde legen, wo zunächst nicht gesagt wird, *wie* eine Berechnung stattfindet, sondern die Betonung darin liegt, *was* berechnet wird. Man abstrahiert hierbei also von der konkreten Berechnungsschritten in einer Maschine.

Hierbei geht man von einer Menge von Basisfunktionen aus:

1. Nachfolgerfunktion  $\text{succ}(x) := x + 1$
2. Modifizierte Vorgängerfunktion  $\text{pred}(x) := \begin{cases} x - 1 & \text{falls } x > 0 \\ x & \text{falls } x = 0 \end{cases}$
3. Test  $\text{test}(x) := \begin{cases} 1 & \text{falls } x = 0 \\ 0 & \text{sonst} \end{cases}$

Aus diesen Basisfunktionen und weiteren bereits definierten Funktionen lassen sich anhand von Schemata (Mustern) neue Funktionen erzeugen.

Das **Einsetzungsschema** erlaubt das Einsetzen einer Funktion als Argument einer anderen Funktion.

**Beispiel 2.16:**

$$\text{plus3}(x) := \text{succ}(\text{succ}(\text{succ}(x)))$$



Hierbei wird eine Funktion  $\text{plus3} : \mathbb{N} \rightarrow \mathbb{N}$  definiert, die angewandt auf ein Argument  $x$  das Ergebnis  $x + 3$  durch geschachtelte Anwendung der  $\text{succ}$ -Funktion liefert.

Die **Primitive Rekursion** ist ein Rekursionsschema, mit dem man ebenfalls eine neue Funktion definieren kann. Das Rekursionsschema ist folgendermaßen definiert: Sei  $g$  eine  $n$ -stellige Funktion (wir betrachten hier nur Funktionen über  $\mathbb{N}$ , d.h.  $g : \mathbb{N}^n \rightarrow \mathbb{N}$ ),  $h$  eine  $(n+2)$ -stellige Funktion. Dann entsteht eine  $(n+1)$ -stellige Funktion  $f$  durch primitive Rekursion aus  $g$  und  $h$ , wenn für alle  $x_1, \dots, x_n$  gilt:

$$\begin{aligned} f(x_1, \dots, x_n, 0) &= g(x_1, \dots, x_n) \\ f(x_1, \dots, x_n, y+1) &= h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)) \end{aligned}$$

Eine neue Funktion  $f$ , dies es zu definieren gilt, wird also durch Angabe von zwei Funktionen  $g$  und  $h$  sowie durch das Rekursionsschema bestimmt. Um dieses rekursive Schema auszunutzen wird man also versuchen, die Berechnung eines Funktionswertes, zum Beispiel  $f(y)$ , auf eine weitere Berechnung mit kleinerem Argument, zum Beispiel  $f(y-1)$  zurückzuführen, bis man auf einen einfachen Fall kommt, der sich mit Hilfe der  $g$ -Funktion direkt angeben lässt (vgl. Vollständige Induktion mit Induktionsanfang, Induktionsbehauptung und Induktionsschluss). Zur Definition der neu zu definierenden Funktion  $f$  wird man also zwei Funktionen angeben, die im Zusammenhang mit dem Rekursionsschema die gewünschte Funktion  $f$  definieren. Man programmiert also  $f$  durch Angabe geeigneter Funktionen  $g$  und  $h$ .

**Beispiel 2.17:**

Wie lässt sich die Additionsfunktion  $\text{add} : \mathbb{N}^2 \rightarrow \mathbb{N}$  über die Primitive Rekursion definieren? Bekannt sind bereits die beiden Grundfunktionen  $\text{succ}, \text{pred} : \mathbb{N} \rightarrow \mathbb{N}$ , die hier nützlich sein können. Im Hinblick auf das Rekursionsschema kann man die Addition ausdrücken durch:

$$\begin{aligned} \text{add}(x, 0) &= x \\ \text{add}(x, y) &= \text{add}(x, y-1) + 1 = \text{succ}(\text{add}(x, \text{pred}(y))) \end{aligned}$$

Um  $\text{add}$  mit Hilfe der Primitiven Rekursion zu definieren, müssen wir jetzt zwei Funktion  $g$  und  $h$  angeben, die dem obigen allgemeinen Rekursionsschema im konkreten Fall genügen:

$$\begin{aligned} f(x, 0) &= g(x) \\ f(x, y+1) &= h(x, y, f(x, y)) \end{aligned}$$

Man wählt  $g$  und  $h$  nun gerade so, dass sie in dieses Schema passen, auf bereits bekannten Funktionen aufbauen

und die gewünschte Funktionalität besitzen:

$$\begin{aligned} add(x, 0) &= g(x) \text{ mit } g(x) := x \\ add(x, y + 1) &= h(x, y, add(x, y)) \text{ mit } h(x, y, z) := succ(z) \end{aligned}$$

Das heißt also, dass die beiden gesuchten Funktionen  $g$  und  $h$ , die die neue Funktion  $f$  in der gewünschten Weise definieren, lauten:  $g : \mathbb{N}^1 \rightarrow \mathbb{N}$ ,  $h : \mathbb{N}^3 \rightarrow \mathbb{N}$  mit:

$$\begin{aligned} g(x) &:= x \\ h(x, y, z) &:= succ(z) \end{aligned}$$

Zur Berechnung von  $add(2, 1)$  kann man die Bestimmungsgleichungen nutzen, die die Funktion definieren:

$$\begin{aligned} add(2, 1) &= succ(add(2, 0)) \text{ mit Regel 2} \\ &= succ(2) \text{ mit Regel 1} \\ &= 3 \end{aligned}$$

### Beispiel 2.18:

Die Multiplikation lässt sich auf die Addition, die eben definiert wurde, leicht zurückführen.

$$\begin{aligned} mult(x, 0) &= 0 \\ mult(x, y) &= mult(x, y - 1) + x \\ &= add(x, mult(x, y - 1)) \end{aligned}$$

Zusammen mit dem Rekursionsschema ergibt sich:

$$\begin{aligned} mult(x, 0) &= g(x) \text{ mit } g(x) := 0 \\ mult(x, y + 1) &= h(x, y, mult(x, y)) \text{ mit } h(x, y, z) := add(x, z) \end{aligned}$$

Als geeignete Funktionen  $g$  und  $h$  lassen sich also angeben:  $g : \mathbb{N}^1 \rightarrow \mathbb{N}$ ,  $h : \mathbb{N}^3 \rightarrow \mathbb{N}$  mit:

$$\begin{aligned} g(x) &:= 0 \\ h(x, y, z) &:= add(x, z) \end{aligned}$$

Zur Berechnung von  $mult(3, 2)$  kann man die Gleichungen benutzen, die die Funktion definieren:

$$\begin{aligned} mult(3, 2) &= add(3, mult(3, 1)) \\ &= add(3, add(3, mult(3, 0))) \\ &= add(3, add(3, 0)) \\ &= add(3, 3) \\ &= 6 \end{aligned}$$

**Beispiel 2.19:**

Die Fakultätsfunktion  $fac : \mathbb{N} \rightarrow \mathbb{N}$  lässt sich auf die eben definierte Multiplikation zurückführen:

$$\begin{aligned} fac(0) &= 1 \\ fac(y+1) &= fac(y) * (y+1) \end{aligned}$$

Somit ergibt sich:

$$\begin{aligned} fac(0) &= g() \text{ mit } g() := 1 \\ fac(y+1) &= h(y, fac(y)) \text{ mit } h(y, z) := mult(add(y, 1), z) \end{aligned}$$

mit  $g : \mathbb{N}^0 \rightarrow \mathbb{N}$ ,  $h : \mathbb{N}^2 \rightarrow \mathbb{N}$ . Zur Berechnung von  $fac(3)$  kann man die Gleichungen benutzen, die die Funktion definieren:

$$\begin{aligned} fac(3) &= mult(add(2, 1), fac(2)) \\ &= mult(3, mult(add(1, 1), fac(1))) \\ &= mult(3, mult(2, mult(add(0, 1), fac(0)))) \\ &= mult(3, mult(2, mult(1, 1))) \\ &= mult(3, mult(2, 1)) \\ &= mult(3, 2) \\ &= 6 \end{aligned}$$

In der Theoretischen Informatik wurde bewiesen, dass beide (und weitere) Modelle, URM und funktionales Berechnungsmodell, gleich mächtig sind, d.h. alle Funktionen (im mathematischen Sinne), für die man in dem einen Modell einen Algorithmus angeben kann, sind auch im anderen Modell berechenbar.

## 2.6 Zusammenfassung und Hinweise

### Literaturhinweise

Die vorgestellten Darstellungsmöglichkeiten von Algorithmen in Diagrammform sind als DIN 66001 beziehungsweise ISO 5807 (Programmablaufplan) und DIN 66261 (Struktogramm) normiert. Die UML wird von der Object Management Group spezifiziert [Gro]. Ein Übersichtsbuch zu UML ist etwa [RQZ07].

### Verstehen

Die Daseinsberechtigung von algorithmischen Grundbausteinen sollte erkannt sein. Die Vor- und Nachteile einiger Darstellungsmöglichkeiten für Algorithmen sollten verstanden sein.

### Kurz und knapp merken

- Ein Algorithmus ist eine Arbeitsanweisung zur Lösung eines (meist kleineren) Problems.
- Aus der Praxis heraus haben sich algorithmische Grundbausteine entwickelt, die oft wiederkehrende Bausteine beim Programmieren darstellen.

- *Anweisung* ist der Fachbegriff für solch einen Baustein.
- Die hier betrachteten Grundbausteine kann man in allen gängigen Programmiersprachen finden/verwenden.
- Algorithmen lassen sich auf unterschiedliche Weise angeben, so etwa:
  - Ein Programmablaufplan oder Flussdiagramm stellt im Wesentlichen die Reihenfolge von durchzuführenden Aktivitäten grafisch dar.
  - Die UML (Unified Modelling Language) kennt Aktivitätsdiagramme, in denen im Wesentlichen die Reihenfolge von durchzuführenden Aktivitäten grafisch dargestellt wird.
  - Ein Struktogramm eignet sich sehr gut, um für kleine Algorithmen die geschachtelte Struktur von Anweisungen darzustellen (die algorithmischen Grundbausteine).
  - Programmablaufplan, Struktogramm und UML Diagramm sind im Wesentlichen für Menschen gedacht.
  - Ein Programm ist nötig, um einen Algorithmus auf einem Rechner ausführen zu können.

## Übungsfragen

- Was sind die algorithmischen Grundbausteine?
- Welche Darstellungsmöglichkeiten für Algorithmen haben Sie kennengelernt?
- Was sind die Vor- und Nachteile eines Struktogramms?
- Was sind die Vor- und Nachteile eines Aktivitätsdiagramms?

## Reflektion des Stoffs

- Diskutieren Sie die Vorteile von Mustern/Bausteinen im Maschinenbau und danach die Vorteile beim Programmieren. Was würde es bedeuten, wenn es solche Grundbausteine nicht geben würde?
- Diskutieren Sie, wieso UML Aktivitätsdiagramme und Struktogramme Vorteile für einen menschlichen Betrachter haben (und falls Sie es schon wissen oder einschätzen können, weshalb diese Darstellungsformen eigentlich nicht so gut als Basis einer Programmausführung auf einem Computer geeignet sind).



## Chapter 3

# Programmiersprachen

Dieses Kapitel soll erklären aber auch motivieren, weshalb Programmiersprachen sehr systematisch und in den Konstrukten aufeinander aufbauend definiert sind. Bis zum Ende des Kapitels wird klar, dass solche Sprachen einen sehr beschränkten Sprachumfang haben und nicht vergleichbar sind mit lebenden Sprachen wie deutsch oder englisch. Wenn man in einer Programmiersprachen programmiert, so wird man sich allerdings auch exakt an die Regeln dieser Sprache halten müssen.

Um einem Rechner einen Algorithmus vorzugeben verwendet man eine Programmiersprache. Programmiersprachen dienen der Verständigung zwischen einem (menschlichem) Programmierer und einem Rechner beziehungsweise einem programmierbaren komplexen System. Programmiersprachen werden nicht nur in der klassischen Programmierung eingesetzt, was im Wesentlichen in diesem Buch getan werden soll, sondern dienen ganz allgemein in weiten Bereichen der Technik und darüber hinaus der flexiblen Interaktion und Steuerung mit beziehungsweise von Rechnern, Telefonanlagen, Steuerungsgeräten, interaktiven Problemlösungsumgebungen und so weiter. Weiterhin dienen spezielle Sprachen der Spezifikation von Software-Systemen (Spezifikationssprachen), der Verifikation von Programmen (Logikkalkül), werden in Datenbanken (Abfragesprachen) und Betriebssystemen (Kommandosprachen) genutzt, dienen dem Entwurf und der Simulation von integrierten Schaltkreisen und vieles mehr.

Eigentlich existieren doch schon sehr viele Sprachen auf der Welt. Wieso nimmt man nicht einfach eine weit verbreitete Sprache und nutzt diese, um damit Programme zu formulieren? Der Grund, dass man dies nicht tut liegt daran, dass Sprachen, die Menschen untereinander nutzen, sich nur sehr schlecht für den in diesem Zusammenhang angedachten Zweck eignen. Gründe sind unter anderem:

- Es lassen sich in menschlichen Sprachen mehrdeutige Sätze formulieren. Beispiel: *Ein Junggeselle ist ein Mann, dem zum Glück noch die Frau fehlt.*
- Menschliche Sprachen sind redundant. Verschiedene Formulierungen drücken den gleichen Sachverhalt aus. Man müsste also erkennen, dass zwei verschiedene Aussagen die gleiche Bedeutung haben.
- Menschliche Sprachen lassen sich nur sehr schwer automatisch (mit einem Programm) verarbeiten. Das liegt unter anderem an den komplexen grammatischen Regeln, je nach Sprache teilweise mit sehr vielen Ausnahmen und Sonderfällen. Dadurch ist es sehr schwer, die unzweifelbare Bedeutung eines Satzes durch ein Programm erkennen zu lassen.

Aus diesen und auch anderen hier nicht aufgeführten Gründen sind Programmiersprachen entstanden, die verglichen mit menschlichen Sprachen eine sehr einfache Struktur haben, sich sehr kompakt angeben lassen (zumindest die Syntax; s.u.) und wo die Bedeutung eines Programms im Wesentlichen auf die Bedeutung der einzelnen Teile zurückgeführt werden kann. Diese Sprachen sind in gewisser Weise Kompromisse zwischen

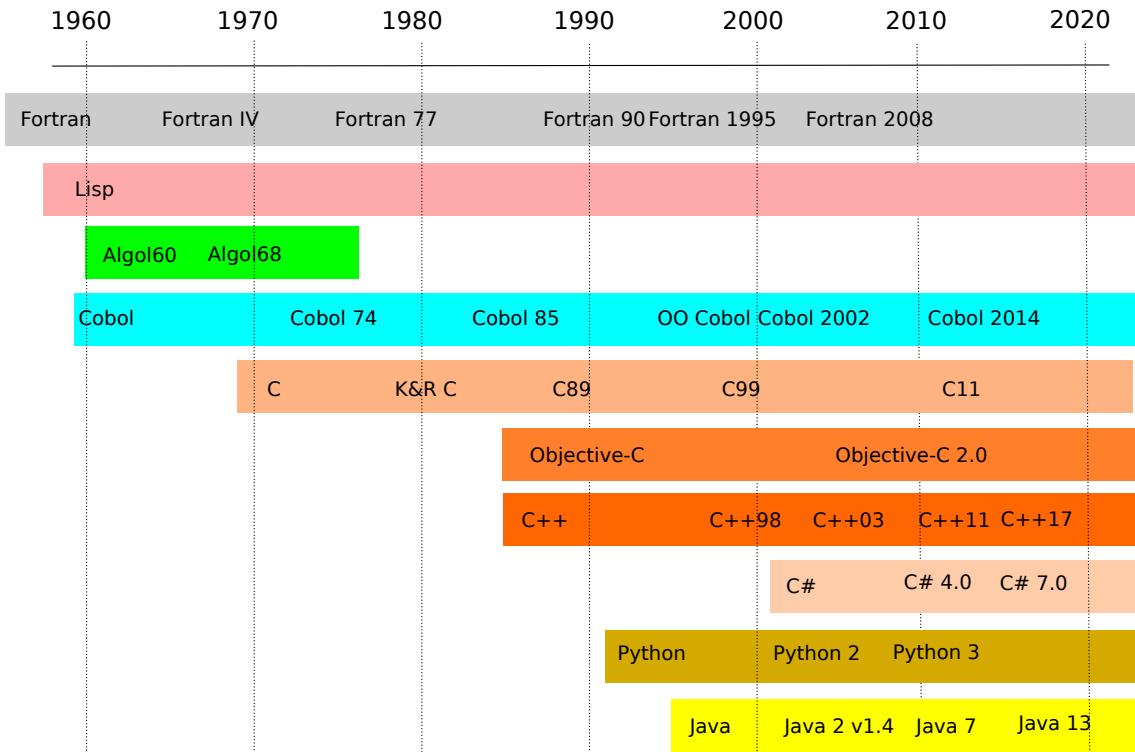


Figure 3.1: Entwicklungsgeschichte ausgewählter Programmiersprachen

der zum Teil mehrdeutigen menschlichen Sprache (deutsch, englisch,...) und der für einen Menschen mehr oder weniger unlesbaren und unschreibbaren Sprache der Computer (Binärcodierung von Maschinenbefehlen). Zu den Aufgaben von Programmiersprachen gehört es allgemein:

- Die Steuerung der Abfolge. Das Stichwort dazu ist **Anweisung**.
- Die Angabe und Manipulation von Werten. Dazu gehört die Bereitstellung einer Menge von Basiswerten (Zeichen, ganze Zahlen, Wahrheitswerte,...) und die Möglichkeit der Definition weiterer, aufgabenrelevanter Wertemengen mit darauf definierten Operationen (zum Beispiel Bankkonto, MP3-Musikstück). Stichwörter dazu sind **Ausdruck**, **primitiver Datentyp** und **Klasse**.

Auf den ersten Blick mag es für einen Informatikanfänger nun wichtig sein, welche Programmiersprache im Folgenden genommen wird. Wie nachfolgend aber begründet wird ist die konkrete Wahl einer Programmiersprache aber eher zweitrangig! Denn die zugrundeliegende *Konzepte* von jeweils bestimmten Klassen von Programmiersprachen sind sehr ähnlich oder sogar gleich. Aus diesem Grund wird in diesem Buch sehr großer Wert darauf gelegt, dass der Leser die zugrundeliegenden Konzepte versteht und einschätzen sowie sinnvoll anwenden kann. Versteht man die grundlegenden Konzepte ist die Einarbeitung in eine neue Programmiersprache einfach!

Programmiersprachen unterliegen wie vieles andere auch einem zeitlichen Wandel, weil die tägliche Nutzung in der Praxis und die Forschung auf diesem Gebiet neue Erkenntnisse und Herangehensweisen liefert. Abbildung 3.1 zeigt die historische Entwicklung einer kleinen Auswahl von wichtigen Programmiersprachen auf. In Tabelle 3.1 sind beispielhaft und grob aufgeführt, welche Programmiersprachen in der Ausbildung und Studium (weltweit) der Informatik in einem Abstand von 10 Jahren bevorzugt genutzt wurden. Wie man dort leicht sieht hat es permanent Änderungen gegeben und wird es auch in Zukunft weiter geben. Im Berufsleben gibt es auch solche Änderungen in der Nutzung von Programmiersprachen; nur sind hier die Gründe für einen

Jahr	bevorzugte Programmiersprache
1950	Binärkodierung
1960	Fortran, Cobol, Algol
1970	Algol68, PL/I
1980	Pascal
1990	C / C++
2000	Java
2010	Java

Table 3.1: Programmiersprachen in der Informatikausbildung

Wechsel (oder auch für einen langen Verbleib bei einer Programmiersprache) oft verbunden mit einem Produkt. Während der Lebenszeit eines umfangreichen Softwareprodukts, die eventuell Jahrzente betragen kann, sind es meist ökonomische Gründe bei einer bestimmten Programmiersprache zu bleiben. Die Neuformulierung einer Problemlösung von mehreren Millionen Programmzeilen ist ein anderes Kaliber als die Neuprogrammierung einer Übungsaufgabe von Java nach C++ im ersten Semester eines Informatikstudiums!

Hoffentlich wurde mit diesen Ausführungen hinreichend motiviert, dass die spätere Vorstellung und Anwendung von bestimmten Sprachkonstrukten in der Programmiersprache Java sicherlich wichtig und notwendig ist, aber weitaus wichtiger ist es zu verstehen, was die zugrundeliegenden Konzepte sind, was sie bewirken und wie man diese sinnvoll zur Problemlösung einsetzt. Denn diese Konstrukte sind universell über Programmiersprachen hinweg.

Es gibt hunderte wenn nicht tausende von Programmiersprachen, die mehr oder weniger weit verbreitet sind und eingesetzt werden. Man kann Programmiersprachen nach bestimmten Kriterien klassifizieren, ein wichtiges Kriterium ist dabei die grundlegende Ausrichtung, worauf eine Programmiersprache zielt. Je nach Ausrichtung kann man sehr detailgenau einen bestimmten Rechner mit einer bestimmten ganz konkreten Hardware programmieren (zum Beispiel ein Embedded System, das die Steuerung kleiner und größerer Geräte übernimmt), braucht aber zur Lösung einer bestimmten Aufgabe ein sehr umfangreiches weil detailreiches Programm. Auf der anderen Seite gibt es Programmiersprachen, in denen man für die gleiche Problemstellung mit wenigen Zeilen Programmcode eine Lösung angeben kann, aber irgendwelche Details der Umsetzung lassen sich damit nicht mehr steuern.

- **Maschinenorientierte Sprachen** bieten als Grundoperationen im Wesentlichen die Operationen an, die ein Prozessor oder eine Prozessorfamilie bereit stellt. Sprachen für bestimmte Prozessoren oder Prozessorklassen nennt man auch **Assemblersprachen**. Ein Vorteil davon ist, dass spezielle Möglichkeiten / Funktionen eines Prozessors damit genutzt werden können. Ein weiterer Vorteil ist eine mögliche Leistungssteigerung (schnellere Ausführung), weil bestimmte Operationen in Spezialfällen explizit und direkt genutzt werden können. Diesen Vorteilen stehen allerdings gravierende Nachteile gegenüber. Da Prozessoren nur sehr einfache Operationen kennen sind zur Angabe einer Problemlösung in solch einer Programmiersprache sehr viele Einzeloperationen nötig. Weiterhin ist dieses Programm, das für eine Prozessorfamilie entwickelt wurde, nicht einfach übertragbar auf eine andere Prozessorfamilie (fehlende **Portabilität**). Aufgrund der Nachteile versucht man nach Möglichkeit solche Sprachen zu vermeiden. Einsatzgebiete sind spezielle maschinennahe Teile in Betriebssystemen oder auch in der Entwicklung von Treibern, mit denen sich spezielle Eigenschaften von Hardware (zum Beispiel eines Grafikprozessors) effizient nutzen lassen.
- **Problemorientierte Sprachen** sind für eine eingegrenzte Problemklasse maßgeschneidert. Für genau die Problemstellungen dieser Klasse werden adäquate Lösungsbausteine angeboten, so dass sich in

diesen Programmiersprachen oft Lösungen sehr kompakt angeben lassen. Ein damit verbundener Nachteil ist es aber dann auch, dass sich diese Sprache auch nur für Probleme dieser Klasse eignet; liegt ein Problem einer anderen Problemklasse vor, muss man eine andere Programmiersprache einsetzen (und eventuell erst lernen). Beispiele für solche problemorientierten Programmiersprachen mit der jeweiligen Problemklasse sind Matlab/Octave (Mathematik), LaTeX (Textverarbeitung) oder PDF (Dokumentbeschreibung).

- **Universelle Sprachen** bieten Bausteine an, die in vielen unterschiedlichen Problemstellungen genutzt werden können. Sie sind einerseits unabhängig von einer bestimmten Prozessorklasse und damit maschinenunabhängig, andererseits bieten sie aber auch nicht die maßgeschneiderten Lösungsbauteile für eine bestimmte Problemklasse an, wie dies die problemorientierten Programmiersprachen leisten. Die wesentlichen Vorteile dieser universellen Sprachen liegen darin, dass die Kenntnis einer Programmiersprache ausreichend ist um darin Lösungen für sehr unterschiedliche Probleme anzugeben und dass Programme in diesen Sprachen sehr portabel sind, das heißt auf vielen verschiedenen Rechnern ablauffähig sind. Beispiele für universelle Programmiersprachen sind Java, C/C++, Pascal, Cobol und Fortran.

In Tabelle 3.2 wird in jeweils einer Programmiersprache der drei Klassen beispielhaft eine Problemlösung für die (einfache) Berechnung des Skalarprodukt angegeben. Für zwei gleichgroße Vektoren  $x, y \in \mathbb{R}^n$  ist das Skalarprodukt  $\text{skalarProdukt} : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$  definiert als  $\text{skalarProdukt}(x, y) := \sum_{i=1}^n x_i \cdot y_i$ . Wie man leicht in Tabelle 3.2 sieht, gibt es zur Berechnung der gleichen mathematischen Funktion drei sehr unterschiedliche Lösungen.

Es gibt eine Reihe von gewichtigen Argumenten, weshalb man *nicht* in einer maschinenorientierten Sprache (der Sprache des Prozessors) programmiert. Maschinensprachen haben nur sehr einfache Operationen, die über Befehle angegeben werden, die ein Prozessor in Binärform (als feste Folge von Nullen und Einsen) erwartet; diese Befehle können aber von der Zentraleinheit sehr effizient ausgeführt werden. Zum Beispiel kann eine Anweisung **a = b+c** (berechne die Summe der beiden Werte, die in den Variablen **b** und **c** stehen, und speichere das Resultat in der Variablen **a**) nicht mit einem Befehl eines Prozessors ausgeführt werden, sondern muss aufgespalten werden in eine Reihe von einfacheren verfügbaren Maschinbefehlen des Prozessors. Zum Beispiel könnte dies für eine hypothetische Zentraleinheit folgendermaßen aussehen:

3-1	1. Befehl: 00000001 00000001 00000110 00000000
3-2	2. Befehl: 00000001 00000010 00000111 00000000
3-3	3. Befehl: 00000010 00000001 00000010 00000011
3-4	4. Befehl: 00000011 00000011 00001000 00000000

Wie man leicht einsieht, ist diese Schreibweise für einen Menschen sehr schwer lesbar. Die Bedeutung einer Binärfolge **00000001 00000001 00000110 00000000** erschließt sich einem Menschen nicht unbedingt auf den ersten (und zweiten) Blick, während ein Prozessor dies korrekt und effizient als ein Befehl erkennt und ausführt. Die vier Befehle des Beispiels sind in der Sprache der hypothetischen Zentraleinheit aufgeteilt in "Päckchen" von 8 Binärziffern (0 oder 1). Das erste Päckchen könnte beispielsweise angeben, welcher Befehl ausgeführt werden soll, also in der ersten Zeile ist dies der Befehl **00000001**, der auch in der zweiten Zeile genutzt wird. Die restlichen Päckchen geben (in Binärdarstellung) die Operanden zu diesem Befehl an. Die Bedeutung wird etwas klarer, wenn man sich diese Befehle in einer nach wie maschinenorientierten Sprache anschaut, die aber für bestimmte Folgen von Nullen und Einsen in den Päckchen lesbare Abkürzungen anbietet (hier mit zusätzlichen Kommentaren jeweils nach **;** in einer Zeile versehen):

3-1	movl -4(%rbp), %edx ;; lade erstes Register mit Wert b
3-2	movl -8(%rbp), %eax ;; lade zweites Register mit Wert c
3-3	addl %edx, %eax ;; addiere erstes und zweites Register
3-4	movl %eax, -12(%rbp);; speichere Ergebnis in a

<b>Assembler (maschinen- orientiert)</b>	<pre> 3-1      .text 3-2      .p2align 4,,15 3-3 .globl skalarProdukt 3-4      .type   skalarProdukt, @function 3-5 skalarProdukt: 3-6 .LFB2: 3-7      testl    %edi, %edi 3-8      xorpd    %xmm1, %xmm1 3-9      jle     .L3 3-10     xorpd    %xmm1, %xmm1 3-11     xorl     %eax, %eax 3-12     .p2align 4,,10 3-13     .p2align 3 3-14 .L4: 3-15     movsd    (%rsi,%rax,8), %xmm0 3-16     mulsd    (%rdx,%rax,8), %xmm0 3-17     addq     \$1, %rax 3-18     cmpl     %eax, %edi 3-19     addsd    %xmm0, %xmm1 3-20     jg      .L4 3-21 .L3: 3-22     movapd    %xmm1, %xmm0 3-23     ret </pre>
<b>Java (universell)</b>	<pre> 3-1 double skalarProdukt(double[] x, double[] y) { 3-2     double summe = 0.0; 3-3     for (int index = 0; index &lt; a.length; index++) { 3-4         summe += x[index] * y[index]; 3-5     } 3-6     return summe; 3-7 } </pre>
<b>Matlab (problem- orientiert)</b>	<pre> 3-1 dot(x,y) </pre>

Table 3.2: Beispielhafte Implementierungen des Skalarprodukts

Listing 3.1: Umrechnung von Dollar nach Euro.

```

3-1  /**
3-2   * Dieses Programm wandelt eine Dollar-Angabe in Euro um
3-3   */
3-4
3-5 public class DollarNachEuro {
3-6
3-7     public static void main(String [] args) {
3-8
3-9         double dollarBetrag = 37.48;      // Dollar-Betrag
3-10        double umrechnungskurs = 0.72;  // Umrechnungskurs
3-11
3-12        // berechne den Euro-Betrag
3-13        double euroBetrag = dollarBetrag * umrechnungskurs;
3-14
3-15        // gebe das Ergebnis auf dem Bildschirm aus
3-16        System.out.println("Der Euro-Betrag ist " + euroBetrag);
3-17    }
3-18 }
```

Diese eng mit einer konkreten Maschinensprache verbundenen Notation ist selber wiederum eine Programmiersprache, man nennt diese maschinenorientierten Sprachen **Assemblersprachen**. Auch bei diesen Assemblersprachen gibt es Unterschiede in der Notation, also verschiedene Sprachen. Was damit aber an dieser Stelle schon bewusst sein muss ist, dass jemand ein Programm in einer Assemblersprache in die Maschinensprache (Folge von Nullen und Einsen) übersetzen muss, also in die Sprache, die letztendlich ein Prozessor nur beherrscht! Jemand könnte ein Mensch sein, üblicherweise übernimmt diese Rolle aber ein Programm, der **Assembler**. Dieser Aspekt wird später noch eingehend betrachtet.

Sowohl für die im Prinzip unlesbare Maschinensprache, die ein Prozessor nur versteht, als auch für die eher (aber sehr eingeschränkt) lesbare Assemblersprache ist klar, dass eine Programmierung auf solch einer tiefen Abstraktionsebene sehr aufwändig und fehleranfällig ist.

Ein weiteres Argument gegen Maschinen- oder Assemblersprachen ist der Umstand, dass Programme in der Sprache einer Zentraleinheit im höchsten Grade abhängig von dieser Zentraleinheit sind. Möchte man das Programm auf einem anderen Rechner ablaufen lassen, so ist dies nicht direkt möglich, wenn in diesem zweiten Rechner nicht die gleiche Zentraleinheit (oder eine Zentraleinheit der gleichen Familie) ist.

Wie man oben an dem Beispiel gesehen hat entsprechen die Befehlssätze von Maschinensprachen nicht direkt den algorithmischen Grundbausteinen, wie sie üblicherweise genutzt werden. Ein Baustein wie zum Beispiel die Iteration hat keine direkte Entsprechung in Maschinensprachen, sondern muss übersetzt werden in die Konstrukte, die die Maschinensprache bietet unter Beibehaltung der Bedeutung des Konstrukts. Beispielsweise wird eine Iteration simuliert durch einen Test und anschließendem bedingtem Sprungbefehl.

## 3.1 Wichtige Bestandteile von Programmiersprachen

In diesem einführenden Unterkapitel sollen vorab schon einige wichtige und in allen Programmiersprachen genutzten Begriffe im Zusammenhang mit Programmiersprachen aus pragmatische Sicht eingeführt werden, um so auch den Einstieg in die späteren folgenden präziseren Formulierungen zu erleichtern.

Um die Diskussion zu erleichtern sollen diese Begriffe anhand eines kleinen Java-Programms eingeführt werden, das in Listing 3.1 schon komplett angegeben wird und auf das in diesem Unterkapitel Bezug genommen wird.

### 3.1.1 Formatierung von Programmen

Programme lassen sich in einzelne Bestandteile zerlegen, ähnlich wie dies bei der deutschen Sprache auch möglich ist.

#### Beispiel 3.1:

Der Satz

*Dies ist ein schönes Beispiel, das auch einen Nebensatz enthält.*

besteht aus einem Hauptsatz und einem Nebensatz, beide folgen weiteren grammatischen Regeln zum Aufbau von Haupt- und Nebensätzen. Weiterhin lässt sich der gesamte Satz aufteilen in einzelne Wörter und diese wiederum in einzelne Buchstaben. ♦

Auf den nächsten Seiten werden zum Einstieg in die Gesamtthematik einige dieser Bestandteile von Programmiersprachen übersichtsartig eingeführt und in nachfolgenden Kapiteln im Detail behaldelt. Der obige Satz im Beispiel hätte auch – ohne die Bedeutung des Satzes zu verändern – notiert werden können als:

*Dies ist ein schönes Beispiel,  
das auch einen Nebensatz enthält.*

Bei dieser Formatierung würde sogar die geschachtelte Struktur dieses Satz durch die Art der Formatierung auf den ersten Blick ersichtlich. Dieser Ansatz, die Struktur durch Formatierung hervorheben, wird auch in vielen (aber nicht allen) Programmiersprachen wie Java oder C/C++ unterstützt, indem die Programmiersprache fast beliebig Leerzeichen zwischen den einzelnen Elementen der Sprache und die Aufteilung von zusammengesetzten Konstrukten auf mehrere Zeilen zulässt. Im Beispiel oben bestand der komplette Satz (übertragen: das komplette Programm) aus dem Hauptsatz und einem davon abhängigen Nebensatz. Im Programmbeispiel zur Umrechnung von Dollar nach Euro ist die Struktur des Programms auch schon durch Einrücken der entsprechenden Zeilen sichtbar gemacht worden (Details dazu kommen später).

Nun hat ein jeder Programmierer seine persönlichen Vorlieben, wie er ein Programm am liebsten formatieren möchte. Da die Geschmäcker bekanntmaßen aber eben unterschiedlich sind und damit die Lesbarkeit aufgrund der Unterschiede im Erscheinungsbild leiden würde, gibt es gewisse Regeln (im Sinne von freiwilliger Übereinkunft) zur Formatierung. Auch wenn man in der Notation von Java-Programmen viele Freiheiten hat (zum Beispiel, ob man das gesamte Programm in eine einzige sehr, sehr lange Zeile schreibt oder es wie oben angegeben auf mehrere Zeilen aufteilt, um so die Struktur hervorzuheben), so haben sich doch im Laufe der Zeit gewisse Regeln entwickelt, die der Lesbarkeit von Programmen förderlich sind. Diese Regeln sind für Java in den *Java Code Conventions* [Javb] festgelegt, auf die an dieser Stelle bereits hingewiesen werden soll. Einzelne Details werden an den entsprechenden Stellen später gegeben.

Diese Möglichkeit der Formatierung durch Einfügen von Leerzeichen und Aufteilung auf mehrere Zeilen in Programmiersprachen nutzen Programmierwerkzeuge wie *eclipse*, indem sie beim Eintippen eines Programms durch den Programmierer automatisch die strukturierten Elemente des Programms entsprechend einrücken und somit für den Leser eines Programms diese Struktur klar ersichtlich machen. Man kann in *eclipse* auch nachträglich ein Programm formatieren ("Schönschreiben"), indem man den zu formatierenden Bereich markiert und anschließend über den Menüpunkt **source** den Unterpunkt **Format** wählt.

### 3.1.2 Kommentare

In manchen Lehr- und Schulbüchern zum Beispiel zur Mathematik oder Physik werden zu den "harten Fakten" in der Fachsprache (zum Beispiel mathematische / physikalische Formeln), die das eigentlich zu Vermittelnde präzise und kompakt in der Fachsprache angeben, erläuternde Kommentare und Beispiele

hinzugefügt. Ohne diese zusätzlichen Kommentare und Beispiele wären die Aussagen in den Formeln natürlich genauso richtig und würden die relevanten Fakten genauso präzise dargestellt, allerdings in der Fachsprache, die von vielen menschlichen Lesern sehr hohe Konzentration und Abstraktionsgabe verlangt. Viele werden sicherlich vor der Situation gestanden haben, dass eine knappe Formel auf den ersten Blick den Zusammenhang nicht klar werden lässt, erläuternde Beispiele und zusätzliche Erläuterungen in umgangssprachlicher Form aber sehr schnell die Zusammenhänge und damit auch die Bedeutung der Formel klar werden lassen.

Ähnlich ist es in Programmiersprachen, wo oft in sehr kompakter Form von wenigen Zeilen Programmcode komplizierte Aktionen umgesetzt werden können. Aber auch hier bieten Programmiersprachen die Möglichkeit, zusätzliche Erläuterungen zum besseren und schnelleren Verständnis zu geben in sogenannten Kommentaren. Die Form, wie diese Kommentare angegeben werden können, ist im Detail bei Programmiersprachen sehr verschieden.

In Java gibt es zwei verschiedene Formen von Kommentaren, die an fast beliebigen Stellen im Programmcode eingefügt werden dürfen und dabei nicht die Bedeutung des eigentlichen Programm verändern.

1. Bei der ersten Form gibt man an:

```
3-1  double dollarBetrag = 37.48;      // Dollar-Betrag
3-2  double umrechnungskurs = 0.72;   // Umrechnungskurs
```

Die beiden Schrägstriche beginnen einen Kommentar, der bis zum Ende dieser Zeile reicht. Die Bedeutung ist, dass alles, was vor den beiden Schrägstrichen in der Zeile steht als normaler Programmtext aufgefasst wird und alles, was in genau dieser Zeile nach den beiden Schrägstrichen kommt inklusive den Schrägstrichen als Kommentar aufgefasst wird. Diese Form von Kommentar verwendet man üblicherweise innerhalb eines Programms zur knappen umgangssprachlichen Beschreibung kleiner Programmstücke ("Einzeiler").

2. Bei der zweiten Möglichkeit kann der Programmtext auch über mehrere Zeilen gehen und damit auch umfangreicher sein. Diese Kommentare haben die Form:

```
3-1  /*
3-2  * Dieses Programm wandelt eine Dollar-Angabe in Euro um
3-3  */
```

Der Kommentar wird eingeleitet durch die Kombination `/*` und wird beendet durch das nächste Vorkommen von `*/` (jeweils ohne Leerzeichen dazwischen). Die Endemarkierung kann im Gegensatz zur ersten Kommentarform auch in einer späteren Zeile erscheinen. Alles, was zwischen diesen beiden Begrenzern steht, wird als Kommentar aufgefasst und nicht als Teil des eigentlichen Programmcodes. Diese Form von Kommentar dient der umfangreicheren Dokumentation, zum Beispiel wenn man ausführlich umgangssprachlich beschreiben möchte, was und wie das nachfolgende Programm oder der Programmteil arbeiten soll.

Als Spezialfall davon ist ein Kommentar anzusehen, der die Form hat:

```
3-1  /**
3-2  * Dieses Programm wandelt eine Dollar-Angabe in Euro um
3-3  */
```

also nach dem öffnenden Kommentarsymbol `/*` ein weiterer Stern sowie innerhalb des Kommentars eine besondere Formatierung. Dies ist ein spezieller Dokumentationskommentar und wird später im Zusammenhang mit Methoden (Kapitel 8) weiter aufgegriffen und dort im Detail vorgestellt.

Im einführenden Beispiel kommen beide Kommentarversionen vor. Der einführende Kommentar zu Beginn über mehrere Zeilen enthält einige Angaben zum Programm, während innerhalb des eigentlichen Programms

Listing 3.2: Umrechnung von Dollar nach Euro.

```

3-1  /**
3-2   * Dieses Programm wandelt eine Dollar-Angabe in Euro um
3-3   */
3-4
3-5 public class DollarNachEuro {
3-6
3-7     public static void main(String [] args) {
3-8
3-9         double dollarBetrag = 37.48;      // Dollar-Betrag
3-10        double umrechnungskurs = 0.72;  // Umrechnungskurs
3-11
3-12        // berechne den Euro-Betrag
3-13        double euroBetrag = dollarBetrag * umrechnungskurs;
3-14
3-15        // gebe das Ergebnis auf dem Bildschirm aus
3-16        System.out.println("Der Euro-Betrag ist " + euroBetrag);
3-17    }
3-18 }
```

(nach der Zeile, die `public static void main` enthält) nur Einzeilerkommentare verwandt werden, die kurze Programmstücke beschreiben.

### 3.1.3 Bezeichner

Aus der Mathematik sind Formulierungen wie  $f(x) = x^2$  bekannt. Diese funktionale Beschreibung der Quadratfunktion könnte vollkommen äquivalent beschrieben werden durch  $f(y) = y^2$ , also indem statt des Namens  $x$  für den Funktionsparameter und in dem beschreibenden Funktionsterm ein anderer Name wie  $y$  gewählt wird. Genauso hätte man äquivalent auch  $g(hugo) = hugo^2$  angeben können, wenn denn bei der Nutzung dieser Funktion an anderer Stelle dann auch der Name  $g$  statt  $f$  verwendet würde. In diesen Beispielen waren  $f$ ,  $g$ ,  $x$ ,  $y$  und  $hugo$  jeweils Namen für etwas ganz bestimmtes. Welcher konkrete Name aber gewählt wurde ist eher nebensächlich, wie die Beispiele gezeigt haben. Wichtig ist aber, dass man etwas über einen Namen referenzieren kann, also darauf Bezug nehmen kann.

In Programmiersprachen ist dies genauso der Fall. Auch hier werden an vielen Stellen Namen benötigt für etwas, was man an anderer Stelle referenzieren möchte. Der konkrete Name ist auch hier eigentlich egal, man muss nur den gleichen Namen für gleiche Sachen verwenden. Der entsprechende Fachbegriff ist **Bezeichner**.

In Programmiersprachen gibt es Vorgaben, wie solche Bezeichnernamen aufgebaut sein müssen. In Java ist die Regel, dass ein Bezeichner mit einem Klein- oder Großbuchstaben anfangen muss, gefolgt von beliebig vielen Buchstaben oder Ziffern.

Bezeichner werden für verschiedene Sachen benötigt, wie das Beispiel aus der Mathematik ja auch schon gezeigt hat. In  $f(x) = x^2$  ist  $f$  ein Bezeichner für eine Funktion und  $x$  ein Bezeichner für einen formalen Parameter dieser Funktion.

Im Programmbeispiel in Listing 3.2 (siehe auch früher) tauchen mehrere Bezeichner auf, die für unterschiedliche Sachen stehen (alle weiteren Details dazu später):

`DollarNachEuro` ist ein Bezeichner für eine Klasse, `main` und `System.out.println` Bezeichner für Methoden (ähnlich mathematischer Funktionen) und `dollarBetrag`, `umrechnungskurs` und `euroBetrag` für Variablen (ähnlich den Parametern in mathematischen Funktionen). Was Klassen, Methoden und Variablen konkret sind, wird erst im Laufe dieses Buches genau erklärt und ist an dieser Stelle auch nicht wichtig. Wichtig ist nur, dass zum Beispiel an *allen* Stellen, wo `euroBetrag` steht, auch ein anderer Name wie zum Beispiel `huzzlipuzzli`

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

Table 3.3: Java Schlüsselwörter

hätte verwandt werden können, ohne dass sich die Bedeutung des Programms verändert hätte.

Manche Bezeichner in Programmen haben eine vordefinierte Bedeutung, sie darf man auch nicht zur Bezeichnung eigener Sachen verwenden. Solche reservierten Bezeichner nennt man **Schlüsselwörter**. Im Beispielprogramm sind dies etwa **public**, **class**, **static**, **void** und **float** und sind typografisch durch Fettdruck hervorgehoben. In Java gibt es ungefähr 50 reservierte Schlüsselwörter, die in Tabelle 3.3 schon einmal im Zusammenhang aufgeführt werden. Zusätzlich haben die Bezeichner **true**, **false** und **null** eine vordefinierte Bedeutung (als Literale).

### 3.1.4 Variable

Sie werden vermutlich schon einmal mit einem Tabellenkalkulationsprogrammen gearbeitet haben (zum Beispiel Microsoft Excel, Openoffice Calc,...). In diesen Programmen gibt es eine rechteckige Anordnung von Zellen, Tabelle oder Tabellenblatt genannt. Die Spalten der Tabelle sind mit Buchstaben beschriftet (**A**, **B**, **C**, ...) und die Zeilen der Tabelle mit Zahlen (1, 2, 3, ...) (siehe Abbildung 3.2). Durch die Kombination von Spaltenname und Zeilennamen lässt sich damit jede Zelle eindeutig identifizieren, eine Zelle hat also beispielsweise den Namen **C4**. In einer Zelle kann genau ein Wert stehen. Dieser Wert kann einerseits direkt angegeben sein (Beispiel: **4711**) oder aber über eine Formel spezifiziert sein, wobei die Formel in der Zelle gespeichert ist (Beispiel: **=4700+11**). Auf dem Bildschirm erscheint als Anzeige dieser Zelle immer der jeweils aktuelle berechnete Wert, im Falle einer formelmäßigen Spezifikation also nicht die Formel selbst. Weiterhin kann in einer solchen Formel auch auf andere Zellen Bezug genommen werden (**=A1+11**), wobei die Bedeutung einer solchen Formel dann ist, dass bei Zellangaben in der Formel der aktuelle Werte jeder referenzierten Zelle in der Berechnung der Formel genutzt werden soll. Eine Nebenbemerkung: ein praktische wie theoretische interessantes Problem tritt auf, wenn Sie beispielsweise in Zelle **A1** als Formel **=A1+1** schreiben.

Zu beachten ist bei Tabellenkalkulationen auch weiterhin, dass verschiedene Wertarten unterschieden werden. Mögliche Wertarten sind etwa Zahlen, Text, Wahrheitswert, Datum, Währungsangabe und vieles andere mehr. Eine Zelle kann also einerseits zur Aufnahme eines Wertes einer bestimmten Wertart genutzt werden (der eventuell über eine Formel zu berechnen ist), andererseits kann auf diesen Wert aber auch in Formeln anderer Zellen Bezug genommen werden.

Analog zu dem Konzept einer Zelle in einer Tabellenkalkulation gibt es auch ähnliche Konzepte bei Programmiersprachen. Das Analogon einer Zelle heißt hier allerdings **Variable**. Jede Variable hat einen Namen, über den diese Variable "ansprechbar" ist. Jede Variable hat zu einem festen Zeitpunkt genau einen Wert, der sich aber im Laufe der Zeit auch ändern kann (ähnlich aber nicht gleich den Zellen in einer Tabelle). Der Wert einer Variablen lässt sich angeben über Formeln ähnlich den Zellformeln, in Programmiersprachen sind es sogenannte Ausdrücke (gleich mehr). Einer Variablen einen neuen Wert zuweisen geht auch ähnlich wie

	A	B	C	D	E
1		Anzahl	Einzelwert	Summe	
2	Produkt 1	1	5,00	5,00	
3	Produkt 2	3	3,98	11,94	
4					
5					
6					
7					

Figure 3.2: Formelmäßige Angabe zum Wert einer Zelle in einer Tabellenkalkulation.

in der Tabelle und ebenso kann man in einer Formel auch den aktuellen Wert anderer Variablen einbeziehen. Im Gegensatz zu Zellen einer tabellenkalkulation ändert sich der Wert einer Variablen nicht automatisch, wenn sich ein Wert einer Variablen ändert, die im formelmäßigen Zusammenhang steht. Ebenso im Unterschied zu Zellen kann eine Variable auch nur Werte bestimmter Wertarten aufnehmen; diese Wertarten werden **Typen** genannt. Auch im Gegensatz zu Tabellen muss man in den meisten Programmiersprachen auch jede Variable, die man nutzen will, explizit angeben inklusive der Angabe, welchen Typ sie hat, also aus welcher Wertart sie aufnehmen kann. Man spricht dann von einer **Variablen Deklaration**. Der Typ einer Variablen kann nach der Deklaration nicht mehr geändert werden.

In Java geschieht solch eine Deklaration wie im Beispielausschnitt gezeigt:

```

3-1   float dollarBetrag = 37.48;      // Dollar-Betrag
3-2   float umrechnungskurs = 1.37;    // Umrechnungskurs Dollar nach Euro
3-3
3-4   // berechne den Euro-Betrag
3-5   float euroBetrag = dollarBetrag * umrechnungskurs;

```

Es werden in dem Beispiel drei neue Variablen eingeführt mit Namen `dollarBetrag`, `umrechnungskurs` und `euroBetrag` und einer vorangestellten Typangabe mit Namen `float`, die diese Variablen dahingehend auszeichnet, dass sie reelle Zahlen aufnehmen können (später mehr dazu). Gleichzeitig wird in den entsprechenden Zeilen auch angegeben, welchen (Anfangs-)Wert die Variable haben sollen in der Form `Variablename = Formel`. In den ersten beiden Fällen ist dies ein Zahlenwert (`37.48` und `1.37`), im letzten Fall wird eine Formel angegeben, in der der aktuelle Wert der Variablen `dollarBetrag` mit dem aktuellen Wert der Variablen `umrechnungskurs` multipliziert werden soll; der Wert der Variablen `euroBetrag` wäre anschließend das Produkt dieser beiden Werte.

Ein wichtiger Unterschied zu Zellen in Tabellenkalkulationsprogrammen ist aber die Möglichkeit, dass bei Variablen der formelmäßige Zusammenhang wie beispielsweise `variablename = variablename + 1` im Gegensatz zur Tabellenkalkulation sehr wohl Sinn macht. Auch dazu später mehr.

Da es insbesondere bei Programmieranfängern oft Verständnisschwierigkeiten mit Variablen und Zuweisungen gibt sei an dieser Stelle der Zusammenhang an einem einfachen Beispiel im Detail erläutert. Gegeben sei folgendes Programmstück:

```

3-1   int a = 2;                  // Zeile 1
3-2   a = a + 1;                // Zeile 2
3-3   System.out.println(a);    // Zeile 3

```

Die erste Zeile `int a = 2;` bewirkt, dass eine neue Variable mit Namen `a` eingeführt wird, die ganzzahlige Werte speichern kann. Gleichzeitig wird in dieser Programmzeile mit `= 2` angegeben, dass der Wert dieser

Variablen nach der Deklaration auf 2 gesetzt werden soll, also diese Variable `a` den Wert 2 enthalten soll. Die nachfolgende Zeile `a = a + 1;` wird nun streng nach obigen Regeln ausgewertet. Zuerst wird der Wert des Ausdrucks auf der rechten Seite dieser Zuweisung berechnet; das Ergebnis ist  $2 + 1 = 3$ , weil der aktuelle Wert der Variablen `a` zum Zeitpunkt der Berechnung dieses Ausdrucks 2 ist. Das Ergebnis dieses Ausdrucks, also der Wert 3, wird anschließend der Variablen `a` als neuer Wert zugewiesen, also in der Variablen gespeichert. Deshalb erfolgt in der nachfolgenden Zeile 3 die Ausgabe der Zahl 3 auf dem Bildschirm.

Es ist also das mathematische Zeichen  $=$  wie in der Gleichung  $x + 3 = y$  verwendet vom Zuweisungsoperator `=` in Java zu unterscheiden, was bei Anfängern manchmal zu Irritation führt. In der Mathematik ist die Bedeutung des Gleichheitszeichens in einer Aussage wie etwa  $a = a + 1$ , dass auf beiden Seiten des Gleichheitszeichens äquivalente Teilaussagen stehen müssen. In Java ist die Bedeutung des Gleichheitszeichens – wie oben beschrieben – anders, nämlich dass der Ausdruck auf der rechten Seite komplett ausgewertet wird und anschließend dieser resultierende Wert der Variablen auf der linken Seite des Gleichheitszeichens zugewiesen wird (in dieser Variablen gespeichert wird). Manche Programmiersprachen verwenden, um diesen Unterschied auch explizit zu machen, deshalb auch andere Zeichen für diese Operation wie etwa `a := a + 1`.

### 3.1.5 Ausdruck

Aus der Schulmathematik sind einige einfache Regeln und Begriffe zur Bildung von Rechentermen bekannt:

- Summe = Summand + Summand
- Differenz = Minuend – Subtrahend
- Produkt = Faktor · Faktor
- Quotient = Dividend / Divisor

Das Gemeinsame an diese Regeln ist, dass in kompakter Form ganz allgemein angeben werden kann, wie zum Beispiel eine Summe gebildet werden kann, ohne die konkreten Zahlen angeben zu müssen, die die Summanden ausmachen. Ein Summand kann jede beliebige Zahl sein, aber zum Beispiel wäre auch ein geklammerter Term wie  $(3 * 4)$  möglich. Was ein Summand genau sein kann, müsste natürlich an anderer Stelle zusätzlich festgelegt werden.

Nachfolgend soll unter dem Begriff **arithmetischer Ausdruck** in einfacher Form Regeln zu den Grundrechenarten mit Zahlen zusammengefasst werden. Ganz allgemein ist ein Ausdruck aufgebaut aus Operatoren und Operanden. Ein arithmetischer Ausdruck in einer vorerst sehr einfachen Form besteht aus Operanden in Form von Zahlen, der Klammerung zur Gruppierung von Teilausdrücken und den binären (zweistelligen) Operatoren  $+, -, *, /$ , die zwischen die Operanden geschrieben werden. Solche einfachen arithmetischen Ausdrücke können auf folgende Weise anhand einer induktiven<sup>1</sup> Definition angegeben werden.

#### Definition 3.1 (arithmetischer Ausdruck):

Gegeben sein eine Zahlenmenge  $Z$  und Operatoren  $\{+, -, *, /\}$ . Die arithmetischen Ausdrücke über  $Z$  sind induktiv definiert durch:

1. Für jedes  $z \in Z$  ist  $z$  ein arithmetischer Ausdruck.
2. Seien  $E, E_1, E_2$  arithmetische Ausdrücke. Dann sind ebenfalls arithmetische Ausdrücke:
  - (a)  $(E)$
  - (b)  $E_1 + E_2$

---

<sup>1</sup>Induktiv bedeutet, dass man von kleinen Einzelbausteinen ausgehend das Große definiert.

- (c)  $E_1 - E_2$
- (d)  $E_1 * E_2$
- (e)  $E_1 / E_2$



Es sei erwähnt, dass Ausdrücke anders als hier in dieser einfachen Form angegeben natürlich komplexer sein können, wie dies etwa in der Schulmathematik, in Java und anderen Programmiersprachen der Fall ist. Statt Zahlen kann man etwa auch Variablen verwenden ( $a + 1$ ) oder auch eine Funktion auswerten ( $f(x) + 1$ ). In den obigen Regeln werden weiterhin Strich-Operatoren gleich behandelt wie Punkt-Operatoren, das heißt, die alte Regel "Punkt-vor-Strich"-Rechnung wird ignoriert. Später wird auf diese Sachverhalte im Zusammenhang mit Grammatiken in Kapitel 3.2.1 eingegangen und Lösungen und Erweiterungen dazu angeben und weiterhin in Kapitel 7 Ausdrücke nochmals ausführlich besprochen.

Durch die induktive Definition eines arithmetischen Ausdrucks können beliebig lange Ausdrücke angegeben werden, da zu jeder Zeit ein Ausdruck anhand der obigen Regeln in zwei weitere Ausdrücke expandiert werden kann, diese beiden Ausdrücke in weitere Ausdrücke und so weiter. Aber gleichgültig, wie lang ein Ausdruck ist: jeden Ausdruck kann man auswerten (ausrechnen) und als Ergebnis kommt immer genau ein Wert als Ergebnis heraus, der auch stellvertretend für den gesamten Ausdruck genommen werden kann.<sup>2</sup>

### Beispiel 3.2:

Statt  $(3 + 4) * 5$  kann man auch  $7 * 5$  schreiben, ohne dass sich an der Bedeutung des Ausdrucks etwas ändern würde (aber sehr wohl am Aussehen).  
◆

Der Begriff eines Ausdrucks spielt in Programmiersprachen eine ähnliche (wichtige) Rolle wie in der Mathematik. Ein Ausdruck steht für die formelmäßige Beschreibung eines Wertes. Diesen Wert erhält man, wenn man den Ausdruck ausrechnet / auswertet. Im Programmbeispiel 3.1 ist dies zum Beispiel zu sehen in der Zeile:

```
3-1 // berechne den Euro-Betrag
3-2 float euroBetrag = dollarBetrag * umrechnungskurs;
```

wo sich ein Betrag in Euro aus dem Dollarbetrag multipliziert mit dem Umrechnungskurs ergibt. Für konkrete Werte für `dollarBetrag` und `umrechnungskurs` (Beispiel: 100 Dollar und 1,30 für den Umrechnungskurs) kann man durch Auswerten des Ausdrucks dann den Wert dieses Ausdrucks  $100 \cdot 1,30 = 130$  ermitteln.

### 3.1.6 Anweisung

Wie eben diskutiert, dienen Ausdrücke dazu, die Berechnung genau eines Wertes exakt anzugeben ("Formel"). In einem Programm soll aber nicht nur ein Wert berechnet werden, sondern der Normalfall ist, dass es umfangreiche und sehr komplexe Abläufe zu beschreiben gibt, die insgesamt den Algorithmus darstellen. Es muss also neben Ausdrücken weiterhin eine Möglichkeit geben, wie man die Ablauffolge in einem Algorithmus / einem Programm steuern kann. Dazu dienen Anweisungen.

Es gibt in Programmiersprachen gewissen Grundanweisungen, die elementare Aufgaben erledigen und weiterhin die Möglichkeit darauf aufbauend oft genutzte algorithmische Grundbausteine anzugeben, um komplexere oder zu wiederholende Aktionen angeben zu können. Eine Grundanweisungen ist beispielsweise die Speicherung eines Wertes gegeben durch einen Ausdruck in einer Variablen. Eine komplexere Aktion wäre

---

<sup>2</sup>Auf einige Probleme wie etwa eine mögliche Division durch 0 wird hier nicht eingegangen.

etwa die wiederholte Ausführung bestimmter Aktionen, bis ein Abbruchkriterium erfüllt ist. In Kapitel 3.2.2 wird auf die Struktur von Anweisungen weiter eingegangen.

## 3.2 Spezifikation von Programmiersprachen

Nach dieser eher pragmatischen Einführung einiger wichtiger Begriffe soll nun im Detail der Aufbau und die Beschreibungsmöglichkeiten von Programmiersprachen behandelt werden. Schaut man sich eine menschliche Sprache an so ist bei solch einer Sprache eine Struktur zu erkennen. Es wird im Folgenden die deutsche Sprache als Beispiel angenommen, analoge Aussagen gelten aber auch für andere lebende Sprachen.

Die deutsche Sprache kennt ein Alphabet bestehend aus Buchstaben  $a, b, c, \dots, z, A, B, C, \dots, Z$ , Ziffern  $0, 1, 2, \dots, 9$  und einigen Sonderzeichen wie zum Beispiel  $(,), \{, \}, -, +, \dots$ . Aufbauend auf diesem Grundalphabet lassen sich Wörter bilden, die nur aus diesen Zeichen aufgebaut sind. Und aufbauend auf den Wörtern lassen sich Sätze bilden. Bekanntermaßen ist nicht jede mögliche Kombination von Buchstaben eine korrektes Wort und nicht je mögliche Kombination von (korrekten) Wörtern ein korrekt geformter Satz. Vielmehr gibt es weiterhin die deutsche Grammatik in Form von Regeln, die genau beschreiben, wie korrekt geformte Sätze aussehen müssen. Aufbauend auf den Sätzen können dann ganze Textwerke angegeben werden, wie etwa Aufsätze, Artikel oder auch dieses Buch, das durch eine Einteilung in Kapitel und Unterkapitel weiter strukturiert ist. Es gibt in der deutschen Sprache weiterhin noch eine Vielzahl von grammatischen Regeln wie etwa zur Deklination, Konjugation, Tempi (Präsens, Futur, ...), Wörtergruppen (Substantiv, Verben, Adjektive, ...), Satzaufbau.

Beispiele zur deutschen Sprache:

Alphabet	$\{a, b, c, \dots, z, A, B, C, \dots, Z, 0, 1, 2, \dots, 9, (,), \{, \}, -, +, \dots\}$
Grundwortschatz	$\{\text{Mensa, Mensaessen, Mensch, Menschheit, ...}\}$
Grammatik	R59: Groß schreibt man das erste Wort eines Ganzsatzes. R60: Substantivistisch gebrauchte Einzelbuchstaben schreibt man im Allgemeinen groß.

Genauso wie die deutsche (oder eine andere) Sprache durch Alphabet, Vokabular und grammatische Regeln definiert ist, so haben auch Programmiersprachen einen wohldefinierten Aufbau mit Alphabet, Grundsymbolen und grammatischen Regeln. Im Gegensatz zu lebenden Sprachen haben Programmiersprachen zwar ein ähnliches (oder gleiches) Alphabet, aber der Grundwortschatz und die Grammatik ist wesentlich einfacher und kompakter verglichen mit lebenden Sprachen.

### Beispiel 3.3:

Gegeben sei das Alphabet  $\{a, b, c, \dots, z, A, B, C, \dots, Z, !\}$  und ein Grundwortschatz  $\{\text{Lola, rennt}\}$  und folgende grammatischen Regeln, wobei Metasymbole, das heißt Symbole der Beschreibungssprache für die Regeln selber, zur Unterscheidung von Wörtern des Grundwortschatzes in spitze Klammern gestellt sind:

1.  $<\text{Satz}>$  ist  $<\text{Nomen}> <\text{Verb}> !$
2.  $<\text{Nomen}>$  ist  $\{\text{Lola}\}$
3.  $<\text{Verb}>$  ist  $\{\text{rennt}\}$

Regel 1 besagt anders ausgedrückt, dass ein Satz aus einem Nomen gefolgt von einem Verb gefolgt von einem Ausrufezeichen besteht. Es gilt mit diese Beispielspezifikation:

- **Lola rennt!** ist ein in Bezug auf diese Regeln syntaktisch korrekter Satz, weil er den Aufbau nach Regel 1 hat und **Lola** ein Nomen ist und **rennt** ein Verb ist.

- **!rennt Lola!** ist nach diesen grammatischen Regeln syntaktisch inkorrekt, weil die Regeln diese Konstruktion nicht ermöglichen würden.

Unter der **Syntax** einer Sprache versteht man die formal durch die Grammatik korrekt herleitbaren Sätze, ohne allerdings auf die Bedeutung der Sätze einzugehen. Unter der **Semantik** eines (syntaktisch korrekten) Satzes versteht man dessen Bedeutung. Die Unterscheidung zwischen Syntax und Semantik ist sehr wichtig.

#### **Beispiel 3.4:**

Die oben angeführte Minigrammatik soll etwas verändert werden, um den Unterschied zwischen Syntax und Semantik zu verdeutlichen. Neben dem Alphabet  $\{a, b, c, \dots, z, A, B, C, \dots, Z, !\}$  gibt es einen etwas erweiterten Grundwortschatz **{Lola, Buch, rennt}** und grammatischen Regeln.

1.  $\langle \text{Satz} \rangle$  ist  $\langle \text{Nomen} \rangle \langle \text{Verb} \rangle !$
2.  $\langle \text{Nomen} \rangle$  ist **{Lola, Buch}**
3.  $\langle \text{Verb} \rangle$  ist **{rennt}**

Nach den Regeln können nun folgende korrekt geformte Sätze gebildet werden:

- **Lola rennt!**
- **Buch rennt!**

Beide Sätze sind entsprechend den angegebenen Grammatikregeln syntaktisch korrekt. Während aber der erste Satz für die meisten Menschen eine klare sematische Aussage enthält, ist dies für den zweiten Satz eher zweifelhaft.

Ob beliebiger Text ein *syntaktisch korrektes Programm* ist, ist mit über Grammatiken von Programmiersprachen relativ einfach zu entscheiden. Die *Bedeutung* eines beliebigen Programms automatisch durch ein weiteres festes Programm zu verstehen ist prinzipiell unmöglich!

Im Folgenden werden verschiedene Möglichkeiten angegeben, wie man die Grammatik einer (Programmier-)Sprache angeben kann und umgekehrt eine gegebene Grammatik lesen kann, um so ein syntaktisch korrektes Programm zu entwerfen. Diese Grammatikformulierungen sind für einen Anfänger sicherlich nicht einfach zu lesen, da sie in kompakten fachlich adäquaten Formulierungen abgefasst werden. Aber dies sind die Formulierungen, die ein Informatiker lesen und verstehen muss, wenn er sich eingehend mit einer Programmiersprache auseinandersetzen will.

### **3.2.1 Grammatik**

Jede Programmiersprache hat ihre eigene Grammatik, unter der von nun an die Zusammenfassung von Grundalphabet und grammatischen Regeln verstanden wird. Durch eine solche Grammatik wird die Syntax dieser Programmiersprache festgelegt. In einem ersten Ansatz soll eine mehr formale Beschreibungsmöglichkeit für Programmiersprachen angegeben werden. Später werden weitere Darstellungsformen vorgestellt.

#### **Definition 3.2 (Alphabet, Wort, $\Sigma^*$ ):**

- Ein **Alphabet** ist eine endliche nichtleere Menge  $\Sigma = \{a_1, \dots, a_n\}$  von Symbolen (Buchstaben).

- Ein **Wort** ist eine endliche Sequenz von Symbolen aus einem gewählten Alphabet  $\Sigma$ .
- Die **Länge** eines Wortes  $w$  (Anzahl der Symbole) bezeichnet man mit  $|w|$ .
- Die Menge aller Wörter über  $\Sigma$  (Bezeichnung dafür  $\Sigma^*$ ) ist induktiv definiert durch:
  1.  $\epsilon \in \Sigma^*$  (leeres Wort)
  2.  $a \in \Sigma \Rightarrow a \in \Sigma^*$  (Symbole des Alphabets)
  3.  $x, y \in \Sigma^* \Rightarrow xy \in \Sigma^*$  (Konkatenation von Wörtern)

◆

Diese sehr formale Definition nochmals umgangssprachlich erläutert.

- Ein Alphabet ist eine beliebige Menge. Ein Beispiel dafür wäre das lateinische Alphabet mit Groß- und Kleinbuchstaben  $\Sigma_1 = \{a, b, c, \dots, z\}$  oder das lateinische Alphabet mit zusätzlichen Ziffern  $\Sigma_2 = \{a, b, c, \dots, z, 0, \dots, 9\}$  oder das Alphabet, das nur zwei Ziffern enthält:  $\Sigma_3 = \{0, 1\}$ .
- Hat man ein Alphabet ausgewählt, so ist ein Wort über diesem Alphabet eine beliebig lange Folge (Sequenz) von Symbolen, die also hintereinander geschrieben werden.
- Zu einem gewählten Alphabet  $\Sigma$  ist die Menge aller Wörter über diesem Alphabet, also  $\Sigma^*$  die Menge aller konstruierbaren Wörter. Dazu gibt es die drei oben angegeben Erzeugendenregeln. In der Menge einmal enthalten ist das leere Wort (mit  $\epsilon$  bezeichnet). Dies erscheint etwas exotisch und hat seinen Sinn im Wesentlichen darin, wenn man mit Wörtern arbeitet. Denken Sie an die Zahl 0, die bei der Addition das neutrale Element darstellt. Ähnlich verhält es sich hier mit dem leeren Wort. Neben dem leeren Wort sind alle Wörter in der Menge  $\Sigma^*$  enthalten, die aus genau einem der möglichen Symbole bestehen. Und in der dritten Erzeugendenregel steckt jetzt die ganze Mächtigkeit dieser Formulierung. Denn diese Regel besagt, dass für zwei beliebige Wörter  $x$  und  $y$ , die man mit Hilfe der drei Regeln vorher konstruiert hat (also auch it Hilfe dieser dritten Regel), wofür also gilt  $x, y \in \Sigma^*$ , dass man diese beiden Wörter hintereinander schreiben kann und daraus entsteht dann ein neues Wort  $xy$ . Wenn zum Beispiel  $x$  für das Wort  $ab$  steht und  $y$  für das Wort  $cd$ , so steht  $xy$  für das Wort  $abcd$ .

### Beispiel 3.5:

Nachfolgende sind einige kleinere Beispiele zu möglichen  $\Sigma$  und  $\Sigma^*$  angegeben.

- 1)  $\Sigma = \{a, b\}$   
 $\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$
- 2)  $\Sigma = \{0, 1\}$   
 $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$
- 3)  $\Sigma = \{a, \dots, z, A, \dots, Z\}$   
 $\{Hallo, SigmaStern, lurumuku\} \subset \Sigma^*$
- 4)  $\Sigma = \{0, \dots, 9\}$   
 $\{1, 129, 3981, 2001\} \subset \Sigma^*$
- 5)  $\Sigma = \{0, \dots, 9\}$   
 $|3981| = 4$   
 $|\epsilon| = 0$

Wie man leicht an den Beispiel sieht, ist für ein nichtleeres Alphabet  $\Sigma$  die Menge aller Wörter über diesem Alphabet eine unendliche Menge.

◆

Worte werden angeben durch das Hintereinanderschreiben der Symbole oder der Teilworte, die das Wort ausmachen. Beispiel: für die beiden Symbole/Buchstaben  $a$  und  $b$  kann man das zusammengesetzte Wort aus beiden Buchstaben als  $ab$  notieren. Für zwei Wörter  $u, v \in \Sigma^*$  gibt  $uv$  das Wort an, das durch das Hintereinanderschreiben der Symbole/Buchstaben aus  $u$  gefolgt von den Buchstaben aus  $v$  gewonnen wird. Es haben sich einige Notationen eingebürgert.

- Mit  $\Sigma^+ := \Sigma^* \setminus \{\epsilon\}$  bezeichnet man die nichtleeren Worte über  $\Sigma$ , also alle Worte außer dem leeren Wort  $\epsilon$ .
- Für  $v \in \Sigma^+$  bezeichnet  $v^n = vv\dots v$  ist die n-te Potenz von  $v$ , also das n-fache Hintereinanderschreiben des Wortes  $v$ , was ein weiteres (langes) Wort ergibt.
- Sei  $w = xyz \in \Sigma^*$  ein Wort bestehend aus den Bestandteilen  $x, y$  und  $z$  mit  $x, y, z \in \Sigma^*$ . Dann heißt  $x$  **Präfix** von  $w$ ,  $y$  **Infix** von  $w$  und  $z$  **Postfix** von  $w$ . Oder anders ausgedrückt: wenn man ein Wort beliebig in drei Teile teilt (inklusive dre Mögllichkeit, dass Teile auch leer sprich  $\epsilon$  sein können), so ist ein Präfix ein erster Teil des Wortes, ein Infix ein mittlerer Teil und ein Postfix ein hinterer Teil des Wortes.

### Beispiel 3.6:

Sei  $\Sigma = \{a, b, c\}$ . Für  $v = ab$  ist  $v^3 = ababab$ .

Für  $w = abc$  sind  $\epsilon, a, ab$  und  $abc$  mögliche Präfixe. Mögliche Infixe sind  $\epsilon, a, b, c, ab, bc, abc$ . Mögliche Postfixe sind  $\epsilon, c, bc$  und  $abc$ . Ein Beispiel der Zusammensetzung des Wortes  $abc$  in Präfix, Infix und Postfix ist  $\epsilon$  als Präfix,  $ab$  als Infix und  $c$  als Postfix. Eine weitere Zusammensetzung wäre  $ab$  als Präfix,  $\epsilon$  als Infix und  $c$  als Postfix. ♦

### Definition 3.3 (Sprache):

Sei  $\Sigma$  ein Alphabet.  $L \subseteq \Sigma^*$  heißt (formale) Sprache über  $\Sigma$ . ♦

Eine formale Sprache ist also ganz simpel eine Menge von Wörtern über einem Alphabet. Und jede Menge, die aus Wörtern über  $\Sigma$  besteht, ist damit auf der anderen Seite auch eine formale Sprache über diesem Alphabet.

### Beispiel 3.7:

Sei  $\Sigma = \{a, b\}$ . Formale Sprachen über  $\Sigma$  sind dann beispielsweise:

- $L_1 = \{a, aa, aaa, aaaa\}$
- $L_2 = \{ab, ba, aabb, bbaa\}$
- $L_3 = \{\epsilon\}$

$L_4 = \{a, b, c\}$  ist keine formale Sprache über  $\Sigma$ .  $L_5 = \{\}$  ist eine formale Sprache über  $\Sigma$ , es gilt jedoch  $L_5 \neq L_3$ . ♦

Nachdem jetzt definiert ist, was prinzipiell eine Sprache ist, hätte man gerne eine Möglichkeit, solch eine Sprache auch etwas kompakter angeben zu können. Denn die bisher genutzte Möglichkeit in den Beispielen war ja, die Menge aller erlaubten Wörter der Sprache explizit in einer Mengennotation aufzuzählen. Man müsste also für die deutsche Sprache dazu alle möglichen Texte explizit angeben, was garnicht möglich wäre (unter anderem, weil diese Menge unendlich ist). Deshalb muss man also auf eine andere, bessere Möglichkeit für diese Aufgabe zurückgreifen.

Der Linguist Noam Chomsky hat in den 1950er Jahren in einer wichtigen Arbeit eine allgemeine Beschreibungsmöglichkeit für Grammatiken entwickelt (nachfolgend angegeben) und weiterhin eine Hierarchie von Grammatikklassen basierend auf deren Struktur identifiziert (später folgend). Mit den Stufen dieser Hierarchie sind wichtige Eigenschaften der Grammatiken verbunden. Diese Chomsky-Ansatz ist in der Notation sehr mathematisch orientiert, später werden weitere Möglichkeit der äquivalenten Formulierung folgen, die für einen Anfänger von der Notation etwas zugänglicher sind.

#### **Definition 3.4 (Grammatik):**

Eine **Grammatik** (oder **Chomsky-Grammatik**) hat die Form  $G = < \Sigma, N, P, S >$  mit  $\Sigma$ ,  $N$  endlich und disjunkt,  $P \subset ((N \cup \Sigma)^* \setminus \Sigma^*) \times (N \cup \Sigma)^*$ ,  $S \in N$ .  $\Sigma$  ist die Menge der **Terminalsymbole**,  $N$  ist die Menge der **Nichtterminalsymbole** oder **Variablen**,  $P$  die Menge der **Produktionen** und  $S$  das **Startsymbol** der Grammatik. Statt einer Produktion  $(x, y)$  schreibt man auch  $x \rightarrow y$ . ♦

Eine Grammatik gibt die Regeln an, nach denen alle Wörter einer durch die Grammatik definierten Sprache hergeleitet werden können. Ein wichtiger Punkt wurde im letzten Satz angegeben: durch die Grammatik beziehungsweise durch die Regeln der Grammatik wird eine Sprache definiert. Also nicht mehr durch das explizite Aufzählen der in der Sprache enthaltenen Wörter, sondern über (wenige) Regeln, wie man alle diese Wörter erzeugen kann. Nachfolgend die formale Definition wiederum mit anderen Worten ausgedrückt:

- Die Menge  $\Sigma$  bildet das Grundalphabet. Alle Wörter der Sprache bestehen nur aus Symbolen dieses Alphabets. In einfachen Beispielgrammatiken, wo man irgendwelche Symbole haben möchte, wählt man vereinbarungsgemäß üblicherweise Kleinbuchstaben  $a, b, c, \dots$  für Terminalsymbole.
- Nichtterminalsymbole aus der Menge  $N$  kann man als eine Art "Hilfssymbole" ansehen, die auf dem Weg zur Erzeugung von Wörtern einer Sprache mit Hilfe der Grammatikregeln der "Speicherung" von Zwischenzuständen dienen. Auch hier gibt es eine Übereinkunft zur einfacheren Unterscheidung: für Nichtterminalsymbole verwendet man üblicherweise Großbuchstaben  $A, B, C, \dots$ .
- Die Menge  $P$  der Produktionen gibt die grammatischen Regeln an, mit deren Hilfe man aus dem Startsymbol  $S$  der Grammatik alle Wörter der Sprache herleiten kann. Dies ist das eigentliche "Fleisch" einer Grammatik, denn durch diese Regeln wird letztendlich die Sprache beschrieben.
- Das Startsymbol  $S$ , ein beliebiges Symbol aus der Nichtterminalsymbolmenge, gibt eindeutig an, mit welchem Nichtterminalsymbol eine Ableitung (siehe unten) beginnen muss.

Zur weiteren Erläuterung und leichteren Lesbarkeit obiger Definition: die Menge  $(N \cup \Sigma)^*$  ist die Menge aller Wörter, die beliebig viele Symbole in beliebiger Reihenfolge aus  $N$  und/oder  $\Sigma$  enthalten (siehe Definition 3.2 mit  $\Sigma^*$ ; hier ist die Grundmenge  $N \cup \Sigma$  statt  $\Sigma$  alleine). Jedes Wort aus der Menge  $(N \cup \Sigma)^* - \Sigma^*$  enthält mindestens ein Nichtterminalsymbol.

#### **Beispiel 3.8:**

Die beiden nachfolgenden Beispiele geben einerseits eine Grammatik für Binärzahlen (Ziffernfolgen, die nur aus den Ziffern 0 und 1 bestehen) an und andererseits eine Grammatik für natürlichen Zahlen in Dezimaldarstellung (Ziffern 0, ..., 9). Die Regeln der zweiten Grammatik stammen original aus der Sprachdefinition von Java [Javc]. Später wird noch näher darauf eingegangen, wieso die Regeln in den Grammatiken genau so sind, wie sie sind, und was sie im Einzelnen bewirken.

1. Sei  $G_{bin} = < \Sigma, N, P, S >$  mit:

- $\Sigma = \{0, 1\}$
- $N = \{\text{BinZahl}, \text{BinRest}\}$
- $S = \text{BinZahl}$
- $P$  enthält die Regeln:
  - (a)  $\text{BinZahl} \rightarrow 0$
  - (b)  $\text{BinZahl} \rightarrow 1$
  - (c)  $\text{BinZahl} \rightarrow 1\text{BinRest}$
  - (d)  $\text{BinRest} \rightarrow 0\text{BinRest}$
  - (e)  $\text{BinRest} \rightarrow 1\text{BinRest}$
  - (f)  $\text{BinRest} \rightarrow 0$
  - (g)  $\text{BinRest} \rightarrow 1$

2.  $G_{dez} = < \Sigma, N, P, S >$  mit:

- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $N = \{\text{DecimalNumeral}, \text{Digits}, \text{Digit}, \text{NonZeroDigit}\}$
- $S = \text{DecimalNumeral}$
- $P$  enthält die Regeln:
  - (a)  $\text{DecimalNumeral} \rightarrow 0$
  - (b)  $\text{DecimalNumeral} \rightarrow \text{NonZeroDigit}$
  - (c)  $\text{DecimalNumeral} \rightarrow \text{NonZeroDigit Digits}$
  - (d)  $\text{Digits} \rightarrow \text{Digit}$
  - (e)  $\text{Digits} \rightarrow \text{Digits Digit}$
  - (f)  $\text{Digit} \rightarrow 0$
  - (g)  $\text{Digit} \rightarrow \text{NonZeroDigit}$
  - (h)  $\text{NonZeroDigit} \rightarrow 1$
  - (i) ... (für 2-8 analog)
  - (j)  $\text{NonZeroDigit} \rightarrow 9$



Hat man eine Grammatik definiert (und damit eine Sprache) möchte man oft gerne wissen, ob eine vorliegendes Wort ein korrekt geformtes Wort in dieser Sprache ist oder nicht. Dazu hilft der nachfolgende Begriff der Ableitung.

### Definition 3.5 (Ableitung, erzeugte Sprache):

Sei  $G = < \Sigma, N, P, S >$  eine Grammatik,  $u, v \in (N \cup \Sigma)^*$  Wörter. Dann sind definiert:

- $u \vdash v : \Leftrightarrow \exists x, y, y', z \in (N \cup \Sigma)^*, u = xyz, v = xy'z, y \rightarrow y' \in P.$   
Man sagt auch:  $u$  ist in einem Schritt (oder direkt) ableitbar nach  $v$ .
- $u \vdash^n v : \Leftrightarrow \exists u_0, \dots, u_n \in (N \cup \Sigma)^*, u = u_0 \vdash u_1 \vdash \dots \vdash u_n = v$   
Die Folge  $u_0 \vdash u_1 \vdash \dots \vdash u_n$  heißt Ableitung (der Länge n).
- $u \vdash^* v : \Leftrightarrow \exists n \geq 0, u \vdash^n v$

- $u \in (N \cup \Sigma)^*$  ist ableitbar in  $G = < \Sigma, N, P, S >$ : $\Leftrightarrow S \vdash^* u$ .

$L(G) := \{w \in \Sigma^* \mid S \vdash^* w\}$  ist die von  **$G$  erzeugte Sprache**, das heißt die Wörter, die aus dem Startsymbol ableitbar sind und nur aus Terminalsymbolen bestehen. ♦

Mit Kleinbuchstaben vom Ende des Alphabets ( $u, v, w, x, y, z$ ) bezeichnet man vereinbarungsgemäß Wörter oder Teilwörter, die aus Terminal- und/oder Nichtterminalsymbolen bestehen können.

Auch hier wieder diese formale Definition mit anderen Worten:

- Die Definition  $u \vdash v$  besagt mit anderen Worten, dass sich ein beliebiges Wort  $v$  aus einem beliebigem Wort  $u$  ableiten / herleiten lässt, wenn das Wort  $u$  einen Infix  $y$  hat, zu dem es eine Grammatikregel  $y \rightarrow y'$  gibt und das Wort  $v$  den gleichen Präfix  $x$  und den gleichen Postfix  $z$  wie  $u$  hat und  $v$  den Aufbau  $v = xy'z$  hat.
- Die Definitionen  $u \vdash^n v$  und  $u \vdash^* v$  besagen, dass man durch mehrfaches Anwenden von auch unterschiedlichen Regeln eine Kette von Einzelableitungsschritten erhält, die zusammengenommen ein Eingangswort in ein Ausgangswort umwandelt. In einem Eingangs- wie Ausgangswort sind beliebige Kombinationen aus Terminal- (Buchstaben) und Nichtterminalsymbolen (Hilfssymbolen) erlaubt. Zur Erinnerung: die Wörter der Sprache selber dürfen aber nur Terminalssymbole enthalten. Insofern ist ein Wort, das Nichtterminalsymbole enthält, als ein Zwischenzustand auf dem Weg zu einem Wort der Sprache zu sehen, das ja eben nur aus Nichtterminalsymbolen bestehen darf.
- Wenn  $S \vdash^* u$  gilt, so ist  $u$  vom Startsymbol der Grammatik ableitbar. Ist in so einem Fall  $u \in \Sigma^*$ , so bedeutet dies, dass das Wort  $u$  ausschließlich aus Terminalssymbolen besteht und in der Sprache enthalten ist.

Für eine gegebene Grammatik kann man also alle Wörter der Sprache aus dem Startsymbol  $S$  mit einer endlichen Folge von Ableitungsschritten herleiten. Jeder einzelne Ableitungsschritt ist die Anwendung genau einer grammatischen Regel aus  $P$ . Gibt es keine Möglichkeit, ein Wort zusammengesetzt aus Symbolen des Alphabets vom Startsymbol abzuleiten, so ist dieses Wort auch nicht in der von der Grammatik erzeugten Sprache enthalten.

### Beispiel 3.9:

Zu der Grammatik  $G_{bin}$  lassen sich mit Hilfe der Regel  $BinRest \rightarrow 1BinRest$  folgende Ableitungsschritte durchführen:

- $BinRest \vdash 1BinRest$ . Das Ausgangswort  $BinRest$  kann mit Hilfe dieser Regel in das Resultatwort  $1BinRest$  umgewandelt werden.
- $0BinRest1 \vdash 01BinRest1$ .  $u$  und  $v$  können also auch Teilwörter eines größeren Wortes sein, also in einem Kontext stehen. In dem Beispielwort  $0BinRest1$  ist  $0$  der Präfix und  $1$  der Postfix, der im Resultatwort übernommen wird. Der mit Hilfe der Grammatikregel  $BinRest \vdash 1BinRest$  umgewandelte Infix ist  $BinRest$ . ♦

### Beispiel 3.10:

1. Eine Ableitung von  $G_{bin}$  ist zum Beispiel (erkennen Sie selber, welche Regeln in jedem Einzelschritt angewandt wurden):

$$\begin{aligned}
 \text{BinZahl} &\vdash 1\text{BinRest} \\
 &\vdash 10\text{BinRest} \\
 &\vdash 100\text{BinRest} \\
 &\vdash 1001
 \end{aligned}$$

das heißt  $S \vdash^4 1001$ , oder anders ausgedrückt  $S \vdash^* 1001$  und somit ist das nur aus Terminalsymbolen bestehende Wort 1001 ein korrektes Wort der Sprache, die durch  $G_{bin}$  erzeugt wird, also  $1001 \in L(G_{bin})$ .

Man kann auch in geschlossener Form angeben, welche Sprache durch die Grammatik  $G_{bin}$  erzeugt wird (hier ohne Beweis; siehe Literaturangaben dazu):

$$\begin{aligned}
 L(G_{bin}) &= \{0\} \cup \text{Menge der 0-1-Wörter, die mit 1 beginnen} \\
 &= \text{Menge der Binärdarstellungen natürlicher Zahlen ohne führende Null}
 \end{aligned}$$

2. Zwei mögliche Ableitungen in  $G_{dez}$  sind:

$$\begin{aligned}
 \text{DecimalNumeral} &\vdash \text{NonZeroDigit} \\
 &\vdash 4 \\
 \text{DecimalNumeral} &\vdash \text{NonZeroDigit Digits} \\
 &\vdash 4 \text{ Digits} \\
 &\vdash 4 \text{ Digits Digit} \\
 &\vdash 4 \text{ Digit Digit} \\
 &\vdash 40 \text{ Digit} \\
 &\vdash 40 \text{ NonZeroDigit} \\
 &\vdash 402
 \end{aligned}$$

Auch hier kann man die von der Grammatik  $g_{dez}$  erzeugt Sprache in geschlossener Form angeben:

$L(G_{dez}) = \text{Menge der Dezimaldarstellungen von natürlichen Zahlen ohne führende Null und die 0 selber.}$

◆

Eine wichtige Fragestellung zu einer Grammatik  $G$  ist, ob ein bestimmtes vorgegebenes Wort  $w$  in der von  $G$  erzeugten Sprachen enthalten ist oder nicht (**Worterkennungsproblem**) und eventuell sogar noch zusätzlich, konkret welche Regeln der Grammatik angewandt werden müssen, um  $S \vdash w$  zu erhalten. Dazu gibt es zwei Ansätze:

1. Ausgehend vom Startsymbol  $S$  der Grammatik versucht man mit Hilfe der Grammatikregeln das Wort  $w$  abzuleiten. Man sucht also ausgehend vom Startsymbol  $S$  den konstruktiven Ableitungsweg  $S \vdash w_1 \vdash w_2 \vdash \dots \vdash w$ .
2. Ausgehend vom Wort  $w$  versucht man rückwärts eine Ableitungskette  $w_n \vdash w, w_{n-1} \vdash w_n, \dots, w_1 \vdash w_2, S \vdash w_1$  aufzubauen. Gelangt man dabei zu  $S$ , so ist das Wort ableitbar in der Grammatik.

Beide Strategien werden in der Praxis eingesetzt, worauf hier aber nicht weiter eingegangen wird. Eine praktische Anwendung findet dies in Compilern, auf die später (Kapitel 3.3.2) eingegangen wird.

Die Festlegung einer Grammatik in Definition 3.4 ist sehr allgemein gehalten. Der oben bereits angeführte Noam Chomsky hat eine hierarchische Klassifizierung aller nur denkbaren Grammatiken entwickelt. Die entsprechenden Klassen von Grammatiken haben wichtige Eigenschaften, die für alle Grammatiken einer Klasse gelten.

#### Definition 3.6 (Chomsky-Klassifikation, kontextsensitiv, kontextfrei, regulär):

Eine Grammatik  $G = < \Sigma, N, P, S >$  ist

1. vom **Typ 0**, falls  $P \subseteq ((\Sigma \cup N)^* - \Sigma^*) \times (\Sigma \cup N)^*$

2. vom **Typ 1** oder **kontextsensitiv**, falls  $P \subseteq ((\Sigma \cup N)^+ - \Sigma^*) \times (\Sigma \cup N)^*$  und zusätzlich gilt, dass für jede Regel  $\alpha \rightarrow \beta \in P$  gilt:  $|\alpha| \leq |\beta|$ .
3. vom **Typ 2** oder **kontextfrei**, falls  $P \subseteq N \times (\Sigma \cup N)^*$
4. vom **Typ 3** oder **regulär**, falls  $P \subseteq N \times (N\Sigma \cup \Sigma \cup \{\})$  (linkslineare Grammatik) beziehungsweise falls  $P \subseteq N \times (\Sigma N \cup \Sigma \cup \{\})$  (rechtslineare Grammatik)

Eine Sprache  $L$  heißt vom Typ  $i$  ( $1 \leq i \leq 3$ ), falls es eine Grammatik vom Typ  $i$  gibt mit  $L = L(G)$ . ♦

Anders ausgedrückt bedeuten diese Definitionen folgendes:

1. Die Regeln einer Typ-0 Grammatik sehen so aus, dass auf der linken Seite einer Produktion beliebige nichtleere Wörter über den Terminal- und Nichtterminalsymbolen erlaubt sind, wovon aber mindestens ein Symbol ein Nichtterminalsymbol sein muss. Auf der rechten Seite können beliebige Wörter über den Terminal- und Nichtterminalsymbolen stehen.
2. Die Regeln einer Typ-1 Grammatik schränken die Regeln einer Typ-0 Grammatik dahingehend ein, dass die linke Seite jeder Produktion von der Anzahl der Symbole (Terminal- und Nichtterminalsymbole) kleiner oder gleich sein muss als die Anzahl der Symbole auf der rechten Seite der Regel.
3. Die Regeln einer Typ-2 Grammatik schränken die Regeln einer Typ-1 Grammatik dahingehend ein, dass auf der linken Seite einer Produktion nur noch genau ein Nichtterminalsymbol erlaubt ist.
4. Die Regeln einer Typ-3 Grammatik schränken die Regeln einer Typ-2 Grammatik dahingehend ein, dass auf der rechten Seite einer Produktion maximal ein Nichtterminalsymbol erlaubt ist und zwar je nach Grammatiktyp nur an einer ganz bestimmten Stelle der Produktion.

Bezeichne  $L_i$  die Familie von Sprachen vom Typ  $i$ , so ist gezeigt worden, dass  $L_i$  eine echte Teilmenge von  $L_j$  ist für  $i > j$ .

### Beispiel 3.11:

Die von den nachfolgenden Grammatiken  $G_1, G_2, G_3$  erzeugten Sprachen  $L(G_1), L(G_2), L(G_3)$  sehen auf den ersten Blick sehr ähnlich aus. Sie werden aber von Grammatiken erzeugt, die zu unterschiedlichen Chomsky-Klassen gehören.

1. Die Grammatik  $G_3 = < \{a, b\}, \{S\}, \{S \rightarrow aS, S \rightarrow bS, S \rightarrow a, S \rightarrow b\}, S >$  ist vom Typ 3.  $L(G_3) = \{a, b\}^+ = \{w \mid w \in \{a, b\}^*, w \neq \epsilon\}$  ist die von  $G_3$  erzeugte Sprache.
2. Die Grammatik  $G_2 = < \{a, b\}, \{S\}, \{S \rightarrow aSb, S \rightarrow ab\}, S >$  ist vom Typ 2. Die von  $G_2$  erzeugte Sprache ist  $L(G_2) = \{a^n b^n \mid n \geq 1\}$ .
3. Die Grammatik  $G_1 = < \{a, b, c\}, \{S, B, C\}, \{S \rightarrow aSBC, S \rightarrow aBC, CB \rightarrow BC, aB \rightarrow ab, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc\}, S >$  ist vom Typ 1. Die von  $G_1$  erzeugte Sprache ist  $L(G_1) = \{a^n b^n c^n \mid n \geq 1\}$ . ♦

Zur Definition von Programmiersprachen werden ausschließlich kontextfreie und reguläre Grammatiken verwendet. Der Grund liegt im Problem der effizienten Analyse von Programmen (siehe Kapitel 3.3.2), wobei das Worterkennungsproblem ein wichtiger Teil dieser Analyse ist. Das Worterkennungsproblem für Typ 3 Grammatiken ist relativ einfach und effizient automatisch zu lösen. Dazu gibt es das Konzept der Endlichen Automaten. Auch für kontextfreie Grammatiken (Typ 2) gibt es einen entsprechenden Mechanismus, den

Kellerautomaten. Jedoch ist deren Aufgabe schwieriger zu lösen, so dass der Aufwand zur Erkennung von Sprachen der Typ-2-Klasse erheblich höher ist. Die gesamte Problematik wird später nochmals aufgegriffen. In Kapitel 3.1.5 wurde der Begriff eines arithmetischen Ausdrucks eingeführt. Zur Erinnerung: Für eine Zahlenmenge  $Z$  und Operatoren  $\{+, -, *, /\}$  sind die arithmetischen Ausdrücke über  $Z$  induktiv definiert durch:

1. Für jedes  $z \in Z$  ist  $z$  ein arithmetischer Ausdruck.
2. Seien  $E, E_1, E_2$  arithmetische Ausdrücke. Dann sind ebenfalls arithmetische Ausdrücke:
  - (a)  $(E)$
  - (b)  $E_1 + E_2$
  - (c)  $E_1 - E_2$
  - (d)  $E_1 * E_2$
  - (e)  $E_1 / E_2$

An dieser Stelle sei nochmals auf den Unterschied zwischen Syntax und Semantik hingewiesen. Nach obigen Regeln (Syntax) ist zum Beispiel  $3/0$  ein durchaus gültiger arithmetischer Ausdruck. Wenn der Wert / die Bedeutung dieses Ausdrucks bestimmt wollte (Semantik) hat man aber ein Problem: die Division durch Null ist ungültig.

Eine entsprechende Grammatik für arithmetische Ausdrücke lässt sich aus dieser induktiven Definition direkt ableiten, die ja gerade über die Struktur eines arithmetischen Ausdrucks geht:  $G_{expr} = < \Sigma, N, P, S >$  mit

- $N = \{\text{Ausdruck}\}$
- $\Sigma = \{(,), +, -, *, /\} \cup \text{Zahlendarstellung}$
- $S = \text{Ausdruck}$
- $P$  wie folgt:
  1.  $\text{Ausdruck} \rightarrow \text{Zahl}$
  2.  $\text{Ausdruck} \rightarrow (\text{Ausdruck})$
  3.  $\text{Ausdruck} \rightarrow \text{Ausdruck} + \text{Ausdruck}$
  4.  $\text{Ausdruck} \rightarrow \text{Ausdruck} - \text{Ausdruck}$
  5.  $\text{Ausdruck} \rightarrow \text{Ausdruck} * \text{Ausdruck}$
  6.  $\text{Ausdruck} \rightarrow \text{Ausdruck} / \text{Ausdruck}$

An dieser Stelle wird vorerst angenommen, dass alle Zahlen als jeweils ein eigenes Symbol in der Menge  $\Sigma$  enthalten sind (Beispiel: 4711 ist ein Symbol des Alphabets). Man gibt normalerweise auch hierfür eigene Produktionen an, wie zum Beispiel in  $G_{dez}$  bereits geschildert. Die Grammatik wird später auch noch entsprechend erweitert.

$G_{expr}$  ist eine Typ-2-Grammatik, da auf der linken Seite aller Regeln nur Nichtterminalsymbole stehen. Sie ist in dieser Form keine reguläre Grammatik (Typ-3), weil die Regeln mit den Operatoren die entsprechende Bedingung verletzen.

### Beispiel 3.12:

Der arithmetische Ausdruck  $((1 + 2) - 3)/4$  lässt sich wie folgt aus dem Startsymbol von  $G_{expr}$  ableiten:

```

Ausdruck   ⊢ Ausdruck/Ausdruck
          ⊢ Ausdruck/4
          ⊢ (Ausdruck)/4
          ⊢ (Ausdruck - Ausdruck)/4
          ⊢ (Ausdruck - 3)/4
          ⊢ ((Ausdruck) - 3)/4
          ⊢ ((Ausdruck + Ausdruck) - 3)/4
          ⊢ ((Ausdruck + 2) - 3)/4
          ⊢ ((1 + 2) - 3)/4

```

Also gilt:  $\text{Ausdruck} \vdash^* ((1 + 2) - 3)/4$  und somit ist  $((1 + 2) - 3)/4$  ein gültiges Wort in  $L(G_{expr})$ . ♦

### 3.2.2 BNF und EBNF

Zur Spezifikation der Syntax von Programmiersprachen verwendet man **kontextfreie Grammatiken (Context Free Grammars, CFG)**, die vom Grammatiktyp 2 sind, das heißt nur Regeln haben, die auf der linken Seite nur ein Nichtterminalsymbol enthalten. Um die Produktionen von Grammatiken von Programmiersprachen, die sehr umfangreich werden können, (maschinen-) lesbarer zu gestalten, hat sich die **Backus-Naur-Form** (abgekürzt BNF) eingebürgert, die nach ihren Erfindern benannt ist. Sie wurde 1960 im Rahmen der Definition von Algol60 entwickelt. Dabei werden die Produktionen einer Grammatik in einer besonderen Form notiert:

1. Nichtterminalsymbole setzt man in spitze Klammern. Beispiel: für das Nichtterminalsymbol  $A$  schreibt man überall in BNF hin:  $\langle A \rangle$ .
2. Statt des Zeichens  $\rightarrow$  verwendet man zur Spezifikation von Regeln die Zeichenfolge  $::=$
3. Haben mehrere Regeln das gleiche Nichtterminalsymbol auf der linken Seite ihrer Produktion, so kann man diese Regeln zusammenfassen, indem man die rechten Seiten jeweils durch einen senkrechten Strich trennt.

#### Beispiel 3.13:

Die Produktionen der bereits bekannten Grammatiken  $G_{bin}$ ,  $G_{expr}$  und  $G_{dez}$  werden nachfolgende in BNF angegeben.

1. Die Produktionen der Grammatik  $G_{bin}$  sehen in BNF wie folgt aus:

```

⟨Binzahl⟩ ::= 0 | 1 | 1⟨BinRest⟩
⟨BinRest⟩ ::= 0 | 1 | 0⟨BinRest⟩ | 1⟨BinRest⟩

```

2. Die Produktionen der Grammatik  $G_{expr}$  sehen in Backus-Naur-Form demnach wie folgt aus:

```

⟨Ausdruck⟩ ::= Zahl
             | ( ⟨Ausdruck⟩ )
             | ⟨Ausdruck⟩ + ⟨Ausdruck⟩
             | ⟨Ausdruck⟩ - ⟨Ausdruck⟩
             | ⟨Ausdruck⟩ * ⟨Ausdruck⟩
             | ⟨Ausdruck⟩ / ⟨Ausdruck⟩

```

Wie sehr einfach an den spitzen Klammern zu erkennen ist ist  $\text{Ausdruck}$  ein Nichtterminalsymbol, wohingegen  $+$  ein Terminalsymbol ist. Das Symbol  $|$  ist ein Metasymbol der BNF selber.

3. Die Regeln der Grammatik  $G_{dez}$  lassen sich in BNF-Notation angeben durch:

```

<DecimalNumeral> ::= 0
                  | <NonZeroDigit>
                  | <NonZeroDigit> <Digits>
<Digits>        ::= <Digit>
                  | <Digits> <Digit>
<Digit>         ::= 0
                  | <NonZeroDigit>
<NonZeroDigit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

Die für den praktischen Umgang mit Grammatiken gedachte Backus-Naur-Form hat einige wichtige Vorteile gegenüber der ursprünglichen Grammatiknotation, die aus der Theoretischen Informatik kommt und eher für prinzipielle Überlegungen geeignet ist:

- Sie ist insbesondere bei vielen Produktionen übersichtlicher darstellbar.
- Sie lässt sich ausschließlich mit Zeichen angeben, die auf einer Computertastatur vorhanden sind.
- Sie ist sehr leicht automatisch erkennbar und zu verarbeiten, auch dadurch, dass alle Nichtterminalsymbole speziell durch  $<>$  markiert sind.

Insbesondere die letzten beiden Punkte waren der Grund ihrer Entwicklung. Man ist mit der BNF nun in der Lage, maschinenlesbar eine Grammatik anzugeben und diese automatisch (in einem Programm) zu verarbeiten. Wenn man in der Praxis in BNF formulierte Grammatikregeln für einige Programmiersprachen anschaut wird man feststellen, dass oft wiederkehrende Muster dort vorkommen. So findet man häufig Situationen, wo eine bestimmtes Konstrukt mehrmals / beliebig oft vorkommen kann.

### **Beispiel 3.14:**

Ein Beispiel dafür kommt in der Grammatik für Binärzahlen  $G_{bin}$  vor, die bereits vorgestellt wurde. Die entsprechenden Regeln waren in BNF:

```

<Binzahl> ::= 0 | 1 | 1<BinRest>
<BinRest>  ::= 0 | 1 | 0<BinRest> | 1<BinRest>

```

Die Regel zu  $BinRest$  besagt eigentlich nichts anderes, als dass  $BinRest$  dafür steht, dass eine nichtleere Folge beliebig vieler Nullen und Einsen vorkommen kann, wo immer  $BinRest$  in anderen Regeln genutzt wird. ♦

Ein weiteres häufig vorkommendes Muster ist die Auswahlmöglichkeit: man möchte ein bestimmtes Konstrukt (genau einmal) vorhanden haben oder nicht.

### **Beispiel 3.15:**

Ein Beispiel dafür ist das optionale Vorzeichen bei ganzen Zahlen. Eine Regel, um aufbauend auf den Regeln zu  $G_{dez}$  auch Zahlen mit oder ohne Vorzeichen beschreiben zu können, wäre etwa folgende Regel:

```

<GanzeZahlen> ::= <DecimalNumeral>
                  | + <DecimalNumeral>
                  | - <DecimalNumeral>

```

Hiermit lassen sich Zahlen angeben, die kein Vorzeichen haben oder ein positives Vorzeichen oder ein negatives Vorzeichen. ♦

Aus diesen aus der Praxis stammenden Überlegungen heraus hat man die Backus-Naur-Form zur sogenannten

**Extended Backus Naur Form** oder kurz **EBNF** erweitert. Die EBNF wurde in einer ursprünglichen Form von Niklaus Wirth 1977 vorgeschlagen und später an einigen Stellen geringfügig modifiziert und wird auch in verschiedenen Varianten genutzt. EBNF ist von der ISO (International Organisation for Standardization) als ISO-Standard **ISO/IEC 14977 : 1996(E)** [ISO96] spezifiziert. In der EBNF werden gegenüber der BNF folgende Veränderungen / Erweiterungen vorgenommen:

1. Die Zeichenfolge `::=` zur Trennung der rechten und linken Seite einer Regeln wird durch das einfache Gleichheitszeichen `=` ersetzt.
2. Rechte Seiten von Regeln oder Teile davon kann man durch `{}` und `[]` klammern. Die Bedeutung dieser Klammerung ist, dass durch `{}` geklammerte Teile 0, 1 oder mehrfach vorkommen dürfen, durch `[]` geklammerte Teile optional sind, das heißt 0 oder 1 mal vorkommen dürfen.
3. Runde Klammern `()` kann man zur Gruppierung nutzen.
4. Terminalsymbole werden in Hochkommas eingeschlossen. Nutzt man EBNF zur Dokumentation zum Beispiel für einen Menschen, so werden üblicherweise Terminalsymbole stattdessen typografisch hervorgehoben. Dies kann zum Beispiel durch Einschluss in Hochkommans ein oder aber durch spezielle Schreibweisen wie etwa Fettdruck oder farbliche Hervorhebung.

Die ISO-Norm kennt noch weitere Unterschiede und Erweiterungen (zum Beispiel n-fache Wiederholung, Abschlusszeichen nach jeder Regel), auf die hier aber nicht weiter eingegangen wird.

### Beispiel 3.16:

1. Die Regeln zur Binärzahl-Grammatik  $G_{bin}$  in EBNF lauten:

$$\begin{aligned} <\text{Binzahl}> &= \mathbf{0} \mid \mathbf{1} \mid \mathbf{1} <\text{BinRest}> \\ <\text{BinRest}> &= (\mathbf{0} \mid \mathbf{1}) \{ \mathbf{0} \mid \mathbf{1} \} \end{aligned}$$

oder noch kompakter:

$$<\text{Binzahl}> = \mathbf{0} \mid \mathbf{1} \mid (\mathbf{1} \{ \mathbf{0} \mid \mathbf{1} \})$$

Zu beachten ist, dass die Zeichen `()` und `{}` Metasymbole der EBNF selber sind. `()` dienen der Gruppierung, `{}` sind die Zeichen der EBNF zur beliebigen Wiederholung des eingeschlossenen Argumentes.

2. Nachfolgend wird die bereits bekannte Grammatik  $G_{dez}$  nochmals unter dem Aspekt EBNF betrachtet. Die Regeln der Grammatik  $G_{dez}$  lassen sich in einer direkten Umsetzung in EBNF-Notation angeben durch:

$$\begin{aligned} <\text{DecimalNumeral}> &= \mathbf{0} \\ &\quad \mid <\text{NonZeroDigit}> [ <\text{Digits}> ] \\ <\text{Digits}> &= [ <\text{Digits}> ] <\text{Digit}> \\ <\text{Digit}> &= \mathbf{0} \\ &\quad \mid <\text{NonZeroDigit}> \\ <\text{NonZeroDigit}> &= \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9} \end{aligned}$$

Hier sei erwähnt, dass bewusst die aus der Originalspezifikation von Java entnommenen Regeln zu EBNF erweitert wurden, die für eine automatische Erkennung durch einen Automaten optimiert sind. Ganze Zahlen lassen auch mit weniger und einfacheren Regeln in EBNF angeben (siehe zum Beispiel nachfolgende Regeln). ◆

In Kapitel 3.2.1 wurde eine Grammatik  $G_{expr}$  für arithmetische Ausdrücke vorgestellt. Diese Grammatik hatte allerdings noch einige Defizite, die hier korrigiert werden sollen. Hier soll diese Grammatik dahingehend erweitert werden, dass Zahlen aus einer Folge von Ziffern gebildet werden können und ein optionales Vorzeichen enthalten können. Weiterhin kennen die bis jetzt angegebenen Regeln keine "Punkt-vor-Strich"-Rechnung, das heißt, die Operatoren für Addition und Differenzbildung werden gleich behandelt wie die Operatoren für Multiplikation und Division. Über die Nutzung verschiedener Nichtterminalsymbole und entsprechend angepasster Regeln kann diese Priorität allerdings erreicht werden. Die erweiterte Grammatik  $G_{expr2}$  in EBNF für Ausdrücke sieht nun wie folgt aus.  $G_{expr2} = <\Sigma, N, P, S>$  mit:

- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, *, /, (, )\}$

- $N = \{\text{Ausdruck}, \text{Zahl}, \text{Ziffer}, \text{Ziffer0}\}$

- $S = \text{Ausdruck}$

- $P$  wie folgt:

$$\begin{aligned}
 <\text{Ausdruck}> &= <\text{Term}> [ (+ <\text{Term}> | - <\text{Term}>) ] \\
 <\text{Term}> &= <\text{Faktor}> [ (* <\text{Faktor}> | / <\text{Faktor}>) ] \\
 <\text{Faktor}> &= <\text{Zahl}> \\
 &\quad | ( <\text{Ausdruck}> ) \\
 <\text{Zahl}> &= [ + | - ] <\text{Ganzzahl}> \\
 <\text{Ganzzahl}> &= <\text{Ziffer0}> \{ <\text{Ziffer}> \} \\
 <\text{Ziffer}> &= \mathbf{0} | <\text{Ziffer0}> \\
 <\text{Ziffer0}> &= \mathbf{1} | \mathbf{2} | \mathbf{3} | \mathbf{4} | \mathbf{5} | \mathbf{6} | \mathbf{7} | \mathbf{8} | \mathbf{9}
 \end{aligned}$$

Die Produktionen zur Ableitung einer Zahl lesen sich wie folgt: Eine Zahl besteht aus einem optionalen Vorzeichen (+ oder -), das vorhanden sein kann aber nicht muss, gefolgt von einem ganzzahligen Anteil. Dieser beginnt mit einer Ziffer ungleich 0 und darauf folgt eine beliebige Anzahl von Ziffern inklusive der 0. Zu beachten ist, dass die Regeln zu Ausdruck, Term und Faktor die Prioritätsregeln (wie die Punkt-vor-Strich-Regel) widerspiegeln.

Ähnlich wie für arithmetische Ausdrücke lassen sich auch logische Ausdrücke angeben, deren Wert bei Auswertung statt einer Zahl ein Wahrheitswert ist. Ein logischer Ausdruck ist ähnlich aufgebaut wie ein arithmetischer Ausdruck. Es gibt Basisfälle (Zahlen bei arithmetischen Ausdrücken, Wahrheitswerte **true** und **false** bei logischen Ausdrücken) und komplexere Fälle mit Operatoren (bei logischen Ausdrücken zum Beispiel  $\wedge$  für die Undverknüpfung und  $\vee$  für die Oderverknüpfung), wobei die Operanden wieder Ausdrücke / Logikausdrücke sein können. Weiterhin ist ein logischer Ausdruck der Vergleich zweier arithmetischer Ausdrücke mit Hilfe von Vergleichsoperatoren wie  $<, \leq, =, \neq, \geq, >$ . Entsprechende Grammatikregeln werden dem Leser zur Übung überlassen.

In Kapitel 3.1 wurde weiterhin die Bedeutung von Anweisungen vorgestellt, die die Ablauffolge in einem Programm steuern. Auch solche Anweisungen lassen sich in Form von EBNF-Regeln angeben.

$G_{anweisung} = <\Sigma, N, P, S>$  mit:

- $\Sigma = \{A, B, C, \dots, Z, a, b, c, \dots, z, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, =, ;, (, )\}$

- $N = \{\text{Anweisung}, \text{Zuweisung}, \text{Schleife}, \text{Selektion}, \text{Block}, \text{Bezeichner}, \text{Ausdruck}\}$

- $S = \text{Anweisung}$

- $P$  wie folgt:

```

<Anweisung>   =   <Zuweisung>
                  | <Schleife>
                  | <Selektion>
                  | <Block>
<Zuweisung>   =   <Bezeichner> = <Ausdruck> ;
<Schleife>     =   while ( <Ausdruck> ) <Anweisung>
<Selektion>    =   if ( <Ausdruck> ) <Anweisung> [ else <Anweisung> ]
<Block>         =   { { <Anweisung> } }

```

Bemerkung: In der letzten Regel zum Nichtterminalsymbol *<Block>* enthält die rechte Seite der Regel zuerst die geschweifte Klammer als Terminalsymbol (fettgedruckt) gefolgt von einer geschweiften Klammer als Metasymbol der EBNF, besagend, dass *<Anweisung>* beliebig oft wiederholt werden kann.

Regeln für Bezeichner und Ausdruck sind hier nicht explizit angeben. In den Regeln erkennt man wiederum den rekursiven Aufbau von Anweisungen. Es gibt einen oder mehrere Basisfälle (hier nur die Zuweisung). Darauf aufbauend können dann komplexere Anweisungen rekursiv aufgebaut werden.

### Beispiel 3.17:

Nachfolgendes kleine Beispielprogramm sei gegeben. Dies ist so kein korrektes Java-Programm, es soll lediglich als Beispiel zur Grammatik dienen. Als Ausdruck sei neben arithmetischen auch logische Ausdrücke erlaubt.

```

3-1 {
3-2   x = 0;
3-3   y = 2;
3-4   while (x < y)
3-5   {
3-6     x = x + 1;
3-7   }
3-8 }
```

Eine Ableitung ausgehend vom Startsymbol der Grammatik ist wie folgt:

```

<Anweisung>   ⊢   <Block>
                ⊢   { <Anweisung> <Anweisung> <Anweisung> }
                ⊢   { <Zuweisung> <Zuweisung> <Schleife> }
                ⊢   { x=0; y=2; while ( <Ausdruck> ) <Anweisung> }   Da ein Ableitung vom
                ⊢   { x=0; y=2; while (x < y) <Block> }
                ⊢   { x=0; y=2; while (x < y) { <Anweisung> } }
                ⊢   { x=0; y=2; while (x < y) { x=x+1; } }
```

Startsymbol der Grammatik zu diesem Programmtext möglich ist, ist dieses Programm ein syntaktisch korrekt geformtes Programm der Sprache, die durch die Grammatik definiert wird. ◆

Ein großer Vorteil der BNF oder EBNF-Notation ist die Maschinenlesbarkeit einer Grammatikspezifikation. Für jede Typ-3-Grammatik lässt aus den Grammatikregeln mit einem Standardverfahren ein Programm erzeugen, dass für beliebige Eingabetexte entscheidet, ob die vorliegende Eingabe ein gültiges Wort der Grammatik ist oder nicht. Das Verfahren, das zur Erkennung in dem Programm genutzt wird, ist zudem sehr effizient. Auch für beliebige Typ-2-Grammatiken lässt sich aus der Angabe aller Grammatikregeln ein Programm erzeugen, das entscheidet, ob für eine vorliegende Eingabe ein gültiges Wort der Grammatik vorliegt oder nicht. Das Verfahren, das hierbei genutzt wird, ist allerdings komplexer und weniger effizient. Neben dem Erkennen liefern diese Verfahren auch gleichzeitig, dass man anhand der genutzten Regeln die Struktur des Eingabewortes erkennen kann.

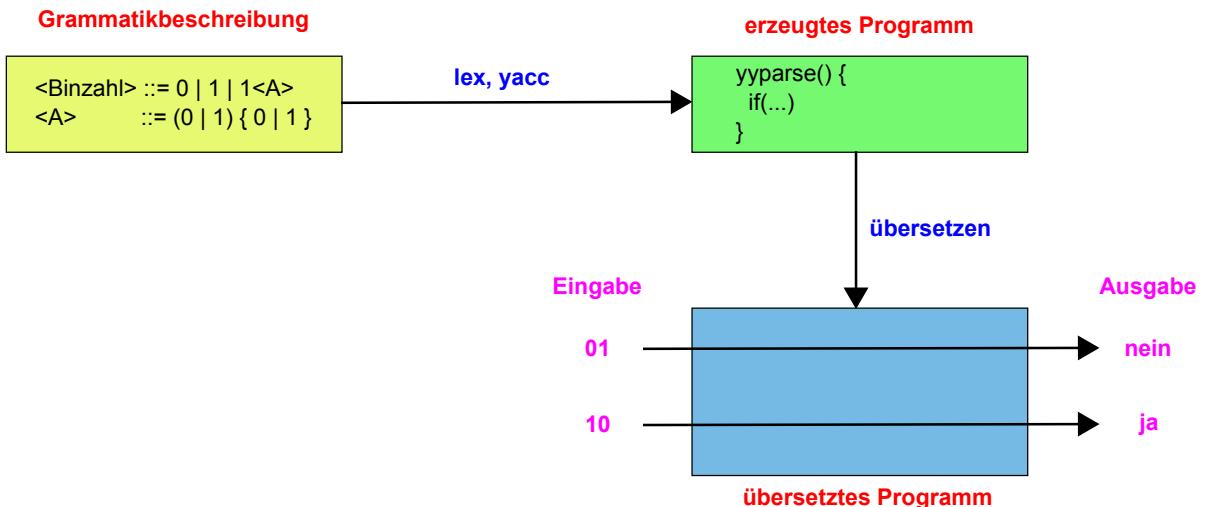


Figure 3.3: Automatische Erzeugung eines Erkennungsprogramms

Beispielhaft seien hier die Programme *lex* und *yacc* genannt, die auf Unix-Systemen eine Grammatikbeschreibung in EBNF-Notation als Eingabe nehmen und daraus ein C-Programm erzeugen. Das (übersetzte) C-Programm überprüft für eine beliebige Eingabe, ob die Eingabe in der durch die Grammatik gegebenen Sprache enthalten ist oder nicht (Abbildung 3.3). Dazu wird in dem erzeugten C-Programm versucht, eine Ableitung vom Startsymbol der Grammatik zum vorgegebenen Eingabewort (beziehungsweise umgekehrt) herzuleiten. Der Vorteil der Nutzung solcher Programme ist, dass der Programmierer nur die Grammatikdefinition entwickeln muss, das Programm zur Erkennung von Wörtern der Sprache, die durch die Grammatik erzeugt wird, wird automatisch erzeugt.

### 3.2.3 Syntaxdiagramme

Auch wenn Syntaxspezifikationen in EBNF deutlich lesbarer sind als die Regeln in Chomsky-Grammatiken, so können sie doch für komplexere Grammatiken, insbesondere durch die Verwendung vieler Nichtterminalsymbole, weniger gut lesbar sein und sind damit für ein schnelles Nachschlagen in einem Sprachreferenzdokument ungeeignet. Syntaxdiagramme sind nun eine weitere Form der Spezifikation von Sprachgrammatiken, die sich von der Zielsetzung her aber von BNF/EBNF unterscheiden. Waren bei BNF/EBNF Aspekte wie Maschinenlesbarkeit treibende Kräfte, so ist dies bei Syntaxdiagrammen eher die Lesbarkeit durch einen Menschen über eine grafische Darstellung von Zusammenhängen. In ihrer grafischen Form bilden sie ein übersichtliches Hilfsmittel zur Angabe von syntaktischen Regeln. In der Praxis sind sie oft in (Lehr-)Büchern zu Programmiersprachen zu finden. Syntaxdiagramme bestehen aus:

1. Terminalsymbolen, die in einem Kreis oder Oval geschrieben werden
2. Nichtterminalsymbolen, die von einem Rechteck umgeben sind
3. gerichtete Kanten (Pfeile), die eine mögliche Abarbeitungsreihenfolge beschreiben, entsprechend einer Ableitungsregel

Zu einer gegebenen Grammatik in EBNF lässt sich leicht ein Syntaxdiagramm konstruieren, indem man Erzeugungs-/Übersetzungsregeln von EBNF nach Syntaxdiagramm rekursiv über den Aufbau der rechten Seiten von EBNF-Produktionen definiert. Eine EBNF-Regel hat auf der linken Seite exakt ein Nichtterminalsymbol. Auf der rechten Seite können als Strukturierungselemente verwendet werden: Terminalsymbole, Nichtterminalsymbole, Hintereinanderschreiben, Alternative |, optionale Auswahl []

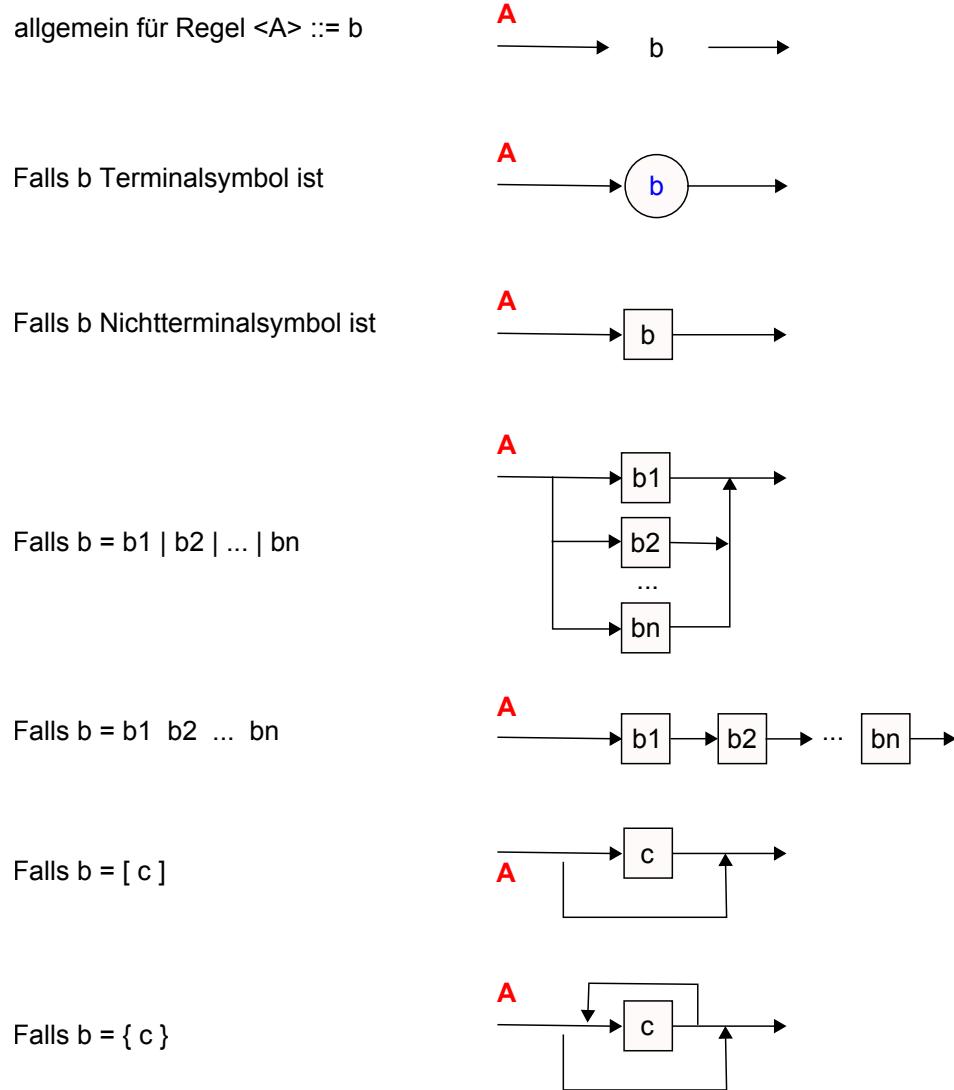


Figure 3.4: Übersetzungsvorschrift von EBNF-Regeln auf Syntaxdiagramme über den Aufbau der rechten Seite  $b$  einer Regel  $\langle A \rangle ::= b$

und Wiederholung durch  $\{ \}$ . Abbildung 3.4 zeigt die entsprechenden Erzeugungsregeln. Zu jedem Nichtterminalsymbol, also den linken Seiten von EBNF-Regeln, gibt es ein Teildiagramm, das die rechte(n) Seite(n) zu diesem Symbol angibt. Terminalsymbole auf der rechten Seite einer Regel werden in Kreise oder Ovale eingeschlossen, Nichtterminalsymbole in Rechtecke. Weitere Strukturierungselemente von Regeln, wie etwa Alternativen, finden ein entsprechendes Äquivalent in der grafischen Darstellung. Einer Ableitung entspricht dann ein gerichteter Weg durch das Syntaxdiagramm, wobei zu jedem Terminalsymbol auf dem Weg im Diagramm genau das gleiche Symbol im Eingabetext vorhanden sein muss, und jedes Nichtterminalsymbol in einem Rechteck durch das entsprechende Syntaxdiagramm zu diesem Rechteck expandiert wird.

In Abbildung 3.5 ist das Beispiel der arithmetischen Ausdrücke aufgezeigt und Abbildung 3.6 zeigt ein Syntaxdiagramm für leicht modifizierte Grammatik der Ganzzahlen in Java.

Abschließend sei anzumerken, dass alle vorgestellten Formen einer Grammatikspezifikation – Chomsky-Grammatik, BNF, EBNF, Syntaxdiagramm – letztendlich alle ineinander überführbar sind und damit nur verschiedene Darstellungsformen des gleichen Sachverhalts sind. Je nach Nutzung kann die eine oder andere Form geeigneter sein.

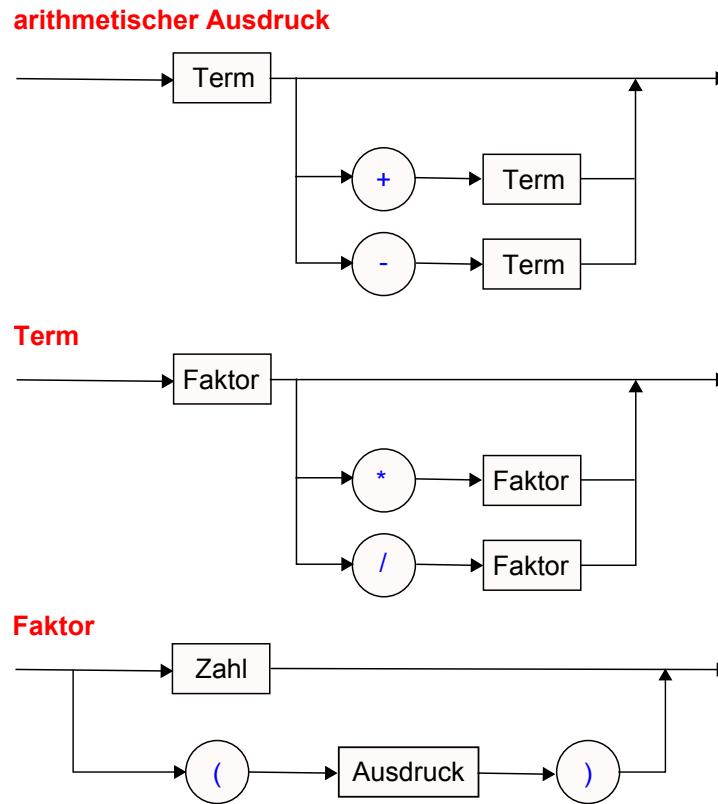


Figure 3.5: Syntaxdiagramm für die vorgestellten arithmetische Ausdrücke

### 3.2.4 Semantik

Während die Syntax die "äußere" Form einer Programmiersprache beschreibt, gibt die **Semantik** den Sprachkonstrukten eine Bedeutung. Bei einem arithmetischen Ausdruck  $4+3$  ist auf syntaktischer Ebene zuerst einmal nichts darüber gesagt, was die Bedeutung dieser Ableitung des Startsymbols *Ausdruck* zu dem Wort  $4+3$  der Sprache, bestehend aus Terminalsymbolen, ist. Einem Menschen ist die Bedeutung des "Wortes"  $4+3$  sofort einsichtig, weil ein Mensch in diesem Beispiel Syntax und Semantik sofort kombinieren würde, ohne dass ihm meistens der Unterschied bewusst ist. Die Frage nach der Semantik / Bedeutung erübrigts sich, wenn die Syntak schon nicht korrekt ist.

#### Beispiel 3.18:

Das Beispiel bezieht sich auf die deutsche Sprache.

1. *Geben Sie die erste gerade natürliche Zahl größer als Null an.*  
Dies ist syntaktisch und semantisch ein korrekter Satz.
2. *Geben Sie die erste gerade natürliche Zahl grüßer als Null an.*  
Dies ist syntaktisch inkorrekt. Damit erübrigts sich die Frage nach der Semantik.
3. *Geben Sie die erste gerade natürliche Zahl kleiner als Null an.*  
Dies ist syntaktisch korrekt aber semantisch inkorrekt / macht keinen Sinn.
4. *Geben Sie die n-te gerade natürliche Zahl größer als Null und kleiner als 100 an.*  
Dies ist syntaktisch korrekt und semantisch korrekt für  $1 \leq n \leq 49$ .



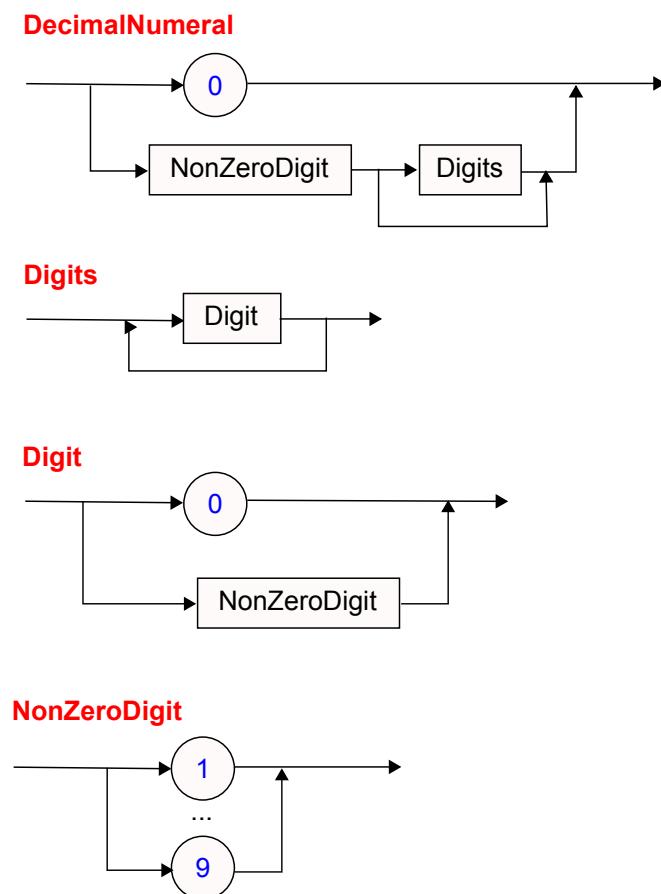


Figure 3.6: Syntaxdiagramm für Ganzzahlen in Java

Die Bedeutung eines Programms kann man auf verschiedene Weise formal definieren. Dazu geht man über den Aufbau / die Struktur einer Programmiersprache und definiert zuerst, was die Bedeutung jedes elementaren Programmkonstruktes ist. Die Bedeutung nichtelementarer also zusammengesetzter Konstrukte wird wiederum über die Bedeutung der zusammengesetzten Struktur definiert. Zweckmäßigerweise geschieht dies über den syntaktischen Aufbau einer Sprache, das heißt über die Regeln der Grammatik. Wird nun während einer Ableitung eine Produktion angewandt, so ist damit eine entsprechende semantische Aktion verbunden.

### **Beispiel 3.19:**

Ein Beispiel, wie mit der Anwendung einer Regel eine bestimmte semantische Aktion verbunden ist, sind die Regeln für Überschriften in HTML. HTML ist eine Seitenbeschreibungssprache für WWW-Seiten. In HTML-Dokumenten (WWW-Seiten) kann man genau wie in einem Buch Kapitelüberschriften auf verschiedenen Hierarchiestufen angeben (Kapitel, Unterkapitel und so weiter) Überschriften haben (in vereinfachter Form hingeschrieben) die Form:

```
<Überschrift>    =  <Überschrift1> [ <Überschrift2> [ <Überschrift3> ] ]
<Überschrift1>  =  <h1> <Inhalt> </h1>
<Überschrift2>  =  <h2> <Inhalt> </h2>
<Überschrift3>  =  <h3> <Inhalt> </h3>
<Inhalt>        =  ...
```

Mit Anwendung der Regel  $\langle\text{Überschrift1}\rangle$  würde zum Beispiel Fettschrift in einer bestimmten Größe ausgewählt, in der die Überschriftbezeichnung ausgegeben würde, mit der Anwendung der Regel  $\langle\text{Überschrift2}\rangle$  eine kleinere Schriftgröße und so weiter. ♦

## **3.3 Übersetzung und Ausführung von Programmen**

Programmiersprachen sind ein Bindeglied zwischen dem menschlichen Programmierer und einem Rechner. Auf den letzten Seiten wurde ihre Bedeutung und ihre Struktur erläutert und motiviert. Weder ist eine Programmiersprache gedacht als lebendige Sprache für einen Menschen noch als eine Sprache, die ein Rechner direkt versteht, dazu ist sie normalerweise (abgesehen von Maschinensprachen) zu komplex und zu mächtig. Auch wenn es sehr viele Programmiersprachen mit sehr unterschiedlicher Syntax und auch Programmierausrichtung gibt, so sind die zugrundeliegenden Konzepte doch oft sehr ähnlich. Der Hauptgrund ihrer Struktur und ihrer Ausdrucksmöglichkeiten ist, dass sie die präzise unzweifelsfreie Angabe eines Algorithmus mit üblichen algorithmischen Grunfbausteinen erlauben und darüber hinaus automatisch verarbeitbar sind. Während man den Umgang mit einer Programmiersprache auf Seiten des Menschen weitgehend ihm selber überlässt (er/sie muss diese Sprache und den Umgang damit erlernen), sieht es auf Seiten eines Rechners anders auch. Hier ist eine automatische Verarbeitung des in der Programmiersprache formulierten Algorithmus verlangt.

Wie gelingt es nun, ein in einer Programmiersprache verfasstes Programm auf einer Zentraleinheit ablaufen zu lassen? Es gibt prinzipiell zwei Ansätze, um dies zu gewährleisten: Interpretierer und Übersetzer. In beiden Ansätzen gibt es eine gemeinsame Teilaufgabe: die Analyse eines vorliegenden Programms. Es muss das Programm als solches erkannt werden, also welche Sprachkonstrukte verwandt wurden. Dazu dient die formale Spezifikation einer Grammatik zu der Sprache und das Erkennen einer Folge von Ableitungen, um vom Startsymbol der Grammatik zu dem vorliegenden Programm zu gelangen.

### 3.3.1 Interpretierer

Ein **Interpretierer** analysiert ein Programm und interpretiert jedes gefundene Einzelkonstrukt. Zur Analyse wird die Eingabe nach und nach in Grundsymbole (*Token*) der Sprache aufgeteilt, aus denen das Programm aufgebaut ist. Interpretieren heißt, dass zu jedem erkannten Programmkonstrukt ein entsprechendes Modul im Interpreter aufgerufen wird. Ein Interpretierer für eine Programmiersprache durchläuft also eine Endlosschleife der folgenden Art:

```

3-1 Wiederhole immer wieder
3-2   Lese das naechste Symbol der Eingabe
3-3   Fuehre die entsprechende Aktion aus

```

Die Ausführung der Aktion für ein Eingabesymbol kann zum Beispiel die Verarbeitung weiterer Eingabesymbole erfordern (Beispiel: Argumente eines Funktionsaufrufes) oder aber auch ein bereits interpretierte Eingabe wiederum ausführen lassen (Beispiel: Schleife).

#### Beispiel 3.20:

Gegeben sei folgende Grammatik:

```

<Programm>      = { <Anweisung> }
<Anweisung>      = <Leseanweisung> | <Schreibanweisung> | <Zuweisung>
                  | <Schleife>
<Leseanweisung>  = read <Bezeichner> ;
<Schreibanweisung> = write <Bezeichner> ;
<Zuweisung>       = <Bezeichner> = <Ausdruck> ;
<Schleife>        = while <RelAusdruck> do { <Anweisung> } endwhile
<Ausdruck>        = <Zahl>
                  | <Bezeichner>
                  | <Ausdruck> + <Ausdruck>
                  | <Ausdruck> * <Ausdruck>
<RelAusdruck>     = <Ausdruck> < <Ausdruck>
<Bezeichner>       = ...
<Zahl>             = ...

```

Zu dieser durch die angegebene Grammatik spezifizierten hypothetischen Minisprache liegt das folgende Programm vor:

```

3-1 read x;
3-2 i = 1;
3-3 while i < 3 do
3-4   x = x * x;
3-5   i = i + 1;
3-6 endwhile;
3-7 write x;

```

Ein Interpretierer zerlegt das Programm nach und nach in seine Grundbestandteile (Token):

<b>read</b>	<b>x</b>	<b>;</b>	<b>i</b>	<b>=</b>	<b>1</b>	<b>;</b>	<b>while</b>	<b>i</b>	<b>&lt;</b>	<b>3</b>	<b>do</b>	<b>x</b>	<b>=</b>	<b>x</b>	<b>*</b>	<b>x</b>	<b>;</b>	<b>i</b>	<b>=</b>	<b>i</b>	<b>+</b>	<b>1</b>	<b>;</b>	<b>endwhile</b>	<b>write</b>	<b>x</b>	<b>;</b>
-------------	----------	----------	----------	----------	----------	----------	--------------	----------	-------------	----------	-----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	-----------------	--------------	----------	----------

Zur Beschleunigung der Interpretierung wird diese Zerlegung in Token oftmals nur einmal zu Beginn der Interpretierung gemacht und im Weiteren nur mit den Token gearbeitet. Token haben eine sehr einfache Struktur und diese kann durch eine Typ-3-Grammatik angegeben werden. Die Erkennung der Token kann wie bereits vorher erwähnt demzufolge effizient mit einem Endlichen Automaten erfolgen.

Nachdem also die Folge von Token vorliegt beginnt der Interpretierer mit der Ausführung. Nach Erkennen des ersten Tokens `read` weiß der Interpretierer aufgrund der entsprechenden Grammatikregel, dass ein weiteres Token folgen muss, in diesem Fall `x`. `x` muss ein Bezeichner (für eine Variable) sein und hinter diesem Bezeichner muss ein Semikolon als Abschluss dieser Anweisung folgen. Ist eine dieser beiden Bedingungen nicht erfüllt, würde der Interpretierer einen Syntaxfehler melden. Anschließend wird aufgrund der Bedeutung von `read` die entsprechende Aktion im Interpreter ausgelöst, das Einlesen eines Wertes in die Variable `x`, für die der Interpretierer im Hauptspeicher Speicherplatz reservieren muss, weil dies das erste Vorkommen von `x` ist. Für nachfolgende Vorkommen von `x` muss sich der Interpretierer merken, dass der soeben angelegte Speicherplatz den aktuellen Wert der Variablen enthält. Nach Anlegen des Speicherplatzes für `x` wird der eingelesene Wert in diesem Speicherplatz abgespeichert. Die Folge von Token `i = 1 ;` bewirkt die Reservierung eines Speicherplatzes für die Variable `i` und die Abspeicherung des Wertes 1 in diesem Speicherplatz.

Nach Erkennen des Tokens `while` weiß der Interpretierer, dass eine Schleife vorliegt (die Bedeutung der Schlüsselwörter sind im Interpreter hinterlegt). Dies bewirkt, dass sich der Interpretierer diese Stelle im Programm merken muss, weil der Schleifentest und der Schleifenrumpf eventuell mehrmals ausgeführt werden müssen. Da Schleifen auch geschachtelt vorkommen können, das heißt im Schleifenrumpf kann selber wieder eine Schleife vorkommen und in deren Schleifenrumpf wieder und so weiter, muss der Interpretierer eine geeignete Möglichkeit dafür finden. An dieser Stelle kann man einen Stack einsetzen, auf dem man die aktuelle Position am Schleifenanfang hinterlegt. Für den Test `i < 3` wird der Inhalt der Speicherstelle der Variablen `i` genommen und mit dem Wert 3 verglichen. Da dieser positiv verläuft, wird der Schleifenrumpf interpretiert. Am Ende des Schleifenrumpfes, das heißt wenn der Interpretierer das Token `endwhile` erkennt, holt sich der Interpretierer die Position des Schleifenbeginns vom Stack, legt sie sofort wieder auf dem Stack ab (für den Fall, dass er wieder zum `endwhile` kommt) und führt den Schleifentest wieder aus.

Nach zweimaliger Interpretierung des Schleifentestes und Schleifenrumpfes fällt der Test dann negativ aus. Dies bewirkt, dass der Interpretierer hinter dem Token `endwhile` weitermacht und aufgrund des Tokens `write` den Wert der Variablen `x` auf dem Bildschirm ausgibt. ♦

Da ein Interpretierer jeweils das nächste Eingabesymbol zeitaufwändig analysieren muss – und zum Beispiel bei einem Schleifenkonstrukt auch mehrfach – machen Interpretierer dort Sinn, wo das gleiche Programm nur einmal oder wenige Male ausgeführt wird oder wo die Aktionen, die ausgelöst werden, sehr mächtig / aufwändig sind. Beispiele für interpretierte Sprachen sind Kommandozeileninterpretierer, Steuerungssprachen, Entwicklung von Programmprototypen und so weiter, aber auch universelle Programmiersprachen wie zum Beispiel Python können interpretiert werden.

Der große Nachteil von interpretierten Programmen ist oft ihre geringe Effizienz in der Ausführung, da der Aufwand des Interpretierens nicht unerheblich ist.

### 3.3.2 Übersetzer

Ein **Übersetzer** (englisch: **Compiler**) analysiert ein Programm und übersetzt das Programm genau einmal in die Maschinensprache der Zentraleinheit, ohne es wie ein Interpretierer auszuführen. Anschließend kann das übersetzte Programm beliebig oft gestartet und direkt von der Zentraleinheit ausgeführt werden. Die Übersetzung geschieht im Compiler durch Analyse des Programms hinsichtlich der Grammatik der Sprache. Der Compiler versucht, eine Ableitung des Startsymbols der Grammatik zu dem vorliegenden Programm zu finden. Für die verschiedenen Sprachkonstrukte wie zum Beispiel Zuweisung, Schleife und Sequenz, muss der Übersetzer entsprechenden Code generieren, der auf Maschinensprachebene die gleiche Semantik bewirkt wie das Hochsprachenkonstrukt.

Das vorliegende in der Programmiersprache formulierte Programm nennt man **Quellprogramm** (englisch: *source program*), das übersetzte Programm Objektprogramm (englisch: *object program*). Da Programme durchaus sehr umfangreich sein können (mehrere Millionen Zeilen Programmcode), kann das Gesamtprogramm aufgeteilt werden auf mehrere Programmdateien, die einzeln durch einen Compiler übersetzt werden. Moderne Programmiersprachen bieten die Möglichkeit der Aufteilung des Gesamtprogramms in einzelne Teile an. Nach dem eigentlichen Übersetzungsvorgang wird aus den einzelnen Objektdaten ein **ausführbares Programm** erzeugt, was durch einen **Binder** (englisch: *Linker*) geschieht.

### Beispiel 3.21:

Nutzt man keine komfortable Programmentwicklungsumgebung für Java, die die nachfolgenden Schritte automatisch ausführt, so sind folgende Schritte zur Übersetzung, Bindung und Ausführung eines Java-Programms durchzuführen. Angenommen, das Java-Quellcodeprogramm ist aufgeteilt in die beiden Dateien **Datei1.java** und **Datei2.java**, wobei **Datei1.java** den Startpunkt des Programms enthalten soll (eine Methode **main**).

```
3-1 javac Datei1.c  
3-2 javac Datei2.c  
3-3 java Datei1
```

**javac** ist der Name des Java-Compilers. Der Java-Compiler würde aufgrund der obigen Befehlssequenz die Quellprogrammdatei **Datei1.java** in eine Objektprogrammdatei **Datei1.class** übersetzen, die Quellprogrammdatei **Datei2.java** in eine Objektprogrammdatei **Datei2.class** übersetzen und anschließend im dritten Schritt die Datei **Datei1.class** mit Hilfe des Java-Laufzeitsystems **java** ausführen und dabei implizit binden. ♦

Die Aufgabe eines Compilers ist also die Übersetzung eines (beliebigen) Programm der Quellcodesprache in Maschinencode der Zielrechners, der bei Ausführung exakt die Bedeutung (Semantik) des Quellprogrammes widergibt. Dabei sollen Fehler im Programmcode, die der Compiler erkennen kann, gemeldet werden. Zur Erledigung dieser sehr komplexen Aufgabe sind Compiler aufgeteilt in Phasen (Abbildung 3.7), die jeweils eine bestimmte Teilaufgabe erledigen. Im Folgenden werden diese Phasen übersichtsartig beschrieben.

### Lexikale Analyse

In der lexikalischen Analyse wird die Folge von Eingabezeichen ähnlich wie in einem Interpretierer zerlegt in eine Folge von Token, den Grundsymbolen der Sprache. Die Struktur von Token ist wie bereits erwähnt im Allgemeinen so einfach, dass sie durch eine reguläre Grammatik (Chomsky-Typ 3) beschrieben werden kann und demzufolge die Token mit einem Endlichen Automaten sehr effizient erkannt werden können. Es gibt verschiedene Kategorien von Token:

- **Konstanten** oder **Literele** wie zum Beispiel **4711** als numerische Konstante
- **Bezeichner** für Variablen, Klassen, Methoden und so weiter
- **Schlüsselwörter** wie zum Beispiel **float** oder **while**. Schlüsselwörter sind reservierte Bezeichner mit einer vordefinierten Bedeutung.
- **Sonderzeichen** wie zum Beispiel **=**, **-** oder **\***. Sonderzeichen können auch zusammengesetzt aus mehr als einem Zeichen bestehen, wie zum Beispiel **!=**, was als ein Sonderzeichen (Token) aufgefasst wird.

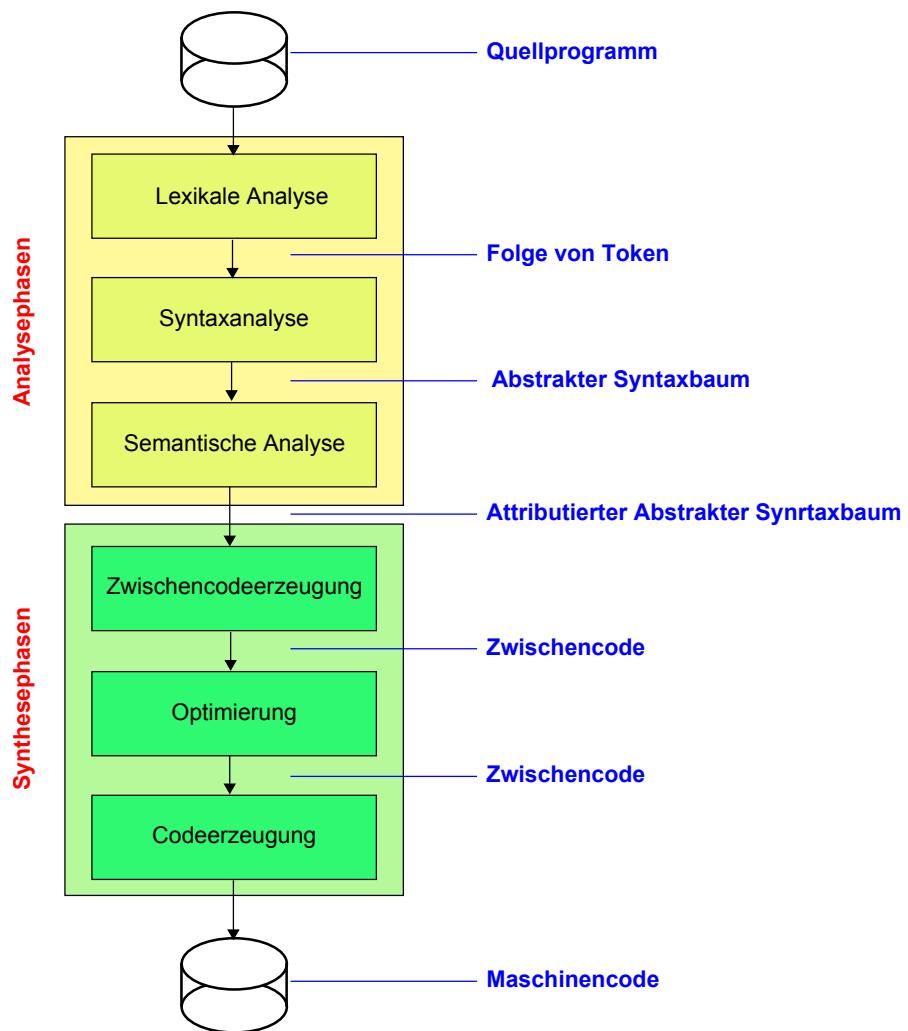


Figure 3.7: Phasen eines Compilers

- **Trennzeichen**, wozu zum Beispiel Leerzeichen, Zeilenendezeichen oder auch Kommentare zählen.  
Durch ein Trennzeichen wird ein vorhergehendes Token beendet. Beispiel dazu: **Bezeichner Name**. Hier beendet das Leerzeichen zwischen den beiden Zeichenfolgen das erste Token **Bezeichner** vom nachfolgenden Token **Name**.

Zeichen, die nicht zum Alphabet der Sprache gehören, oder fehlerhafte Token werden als solche erkannt und eine entsprechende Fehlermeldung wird ausgegeben.

### Beispiel 3.22:

Gegeben sei folgende reguläre Grammatik für eine hypothetische Sprache:

<code>&lt;Bezeichner&gt;</code>	<code>=</code>	<code>&lt;Buchstabe&gt; { &lt;Buchstabe&gt; }</code>
<code>&lt;Wert&gt;</code>	<code>=</code>	<code>&lt;Bezeichner&gt;   &lt;Zahl&gt;</code>
<code>&lt;Zahl&gt;</code>	<code>=</code>	<code>&lt;Ziffer0&gt; { &lt;Ziffer&gt; }</code>
<code>&lt;Ziffer&gt;</code>	<code>=</code>	<code>0   &lt;Ziffer0&gt;</code>
<code>&lt;Ziffer0&gt;</code>	<code>=</code>	<code>1   2   3   4   5   6   7   8   9</code>
<code>&lt;Buchstabe&gt;</code>	<code>=</code>	<code>a   ...   z</code>
<code>&lt;Schlüsselwort&gt;</code>	<code>=</code>	<code>int   while   do   endwhile</code>

Für folgende Eingabe:

```

3-1 int i ;
3-2 int j ;
3-3 i = 3;
3-4 while i > 0 do
3-5   j = 2;
3-6   while j > 1 do
3-7     j = j - 1;
3-8   endwhile;
3-9   i = i - 1;
3-10 endwhile;

```

würde die lexikalische Phase des Compilers die folgende Tokenfolge als Ergebnis dieser Phase an die nachfolgende syntaktische Analyse weitergeben:

int [i] ; int [j] ; i = [3] ; while [i] > [1] do [j] = [2] ; while [j] > [1] do [j] = [j] - [1] ; endwhile [i] = [i] - [1] ; endwhile [;

Für `endwhile` würde zum Beispiel die Information weitergegeben, dass das Token ein Schlüsselwort ist und welches Schlüsselwort es ist. Für `i` würde weitergegeben, dass dies ein Bezeichnertoken ist und als Zusatzinformation, dass der "Wert" des Bezeichnertokens `i` ist. ♦

### Syntaktische Analyse

In der syntaktischen Analyse wird die Tokenfolge mit Hilfe der Grammatikregeln der Sprache analysiert. Programmiersprachen haben nicht mehr eine so einfache Struktur wie Token. Zum Beispiel besteht eine Schleife aus einem Testausdruck und einem Schleifenrumpf, welcher wiederum eine beliebige Anweisung wie etwa eine Schleife bestehen kann und so weiter (Eine "Rekursion" ist also möglich). Aus diesem Grund ist eine kontextfreie Grammatik (Chomsky Typ 2) notwendig, die durch einen Kellerautomaten erkannt werden kann. Der Kellerautomat versucht, eine Ableitung des Startsymbols der Grammatik zu dem Tokenstrom (Folge von Terminalsymbolen der Grammatik) zu finden. Gelingt dies, so liegt ein syntaktisch korrektes Programm vor. Andernfalls meldet der Kellerautomat, an welcher Stelle im Tokenstrom ein Syntaxfehler vorliegt.

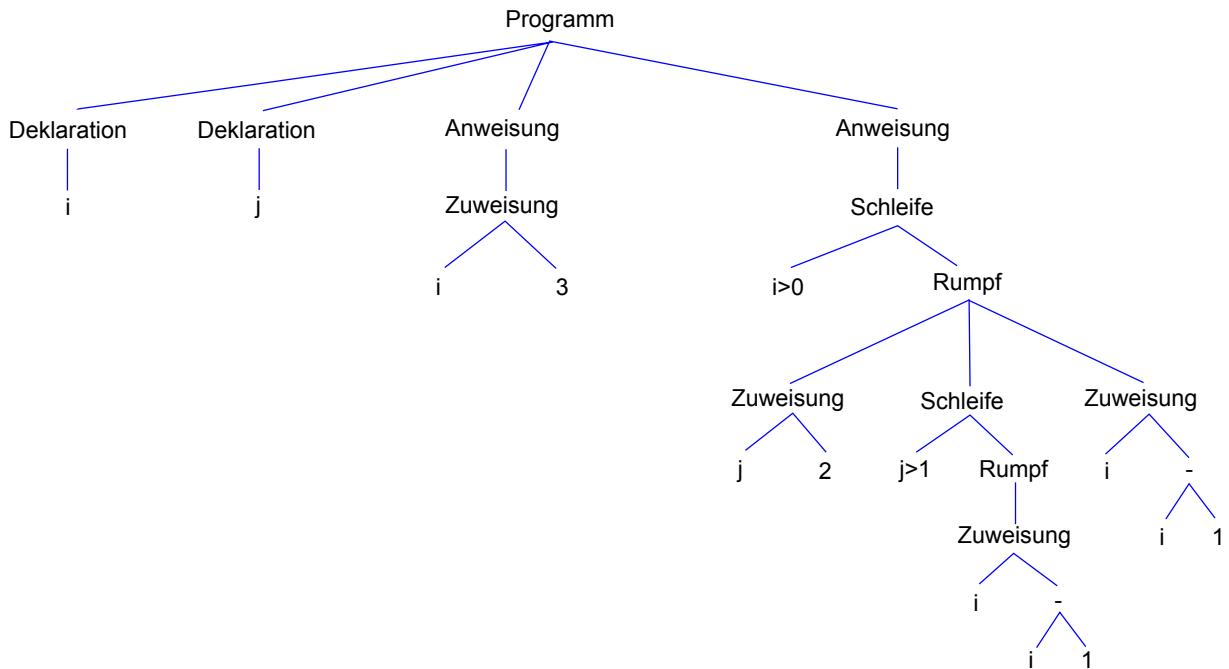


Figure 3.8: Abstrakter Syntaxbaum zum Beispielprogramm

**Beispiel 3.23:**

Zur Definition der Syntax der (sehr eingeschränkten) Miniprogrammiersprache ist folgende kontextfreie Grammatik gegeben:

$$\begin{aligned}
 <\text{Programm}> &= \{ <\text{Deklaration}> \} \{ <\text{Anweisung}> \} \\
 <\text{Deklaration}> &= \text{int } <\text{Bezeichner}> ; \\
 <\text{Anweisung}> &= <\text{Zuweisung}> \mid <\text{Schleife}> \\
 <\text{Zuweisung}> &= <\text{Bezeichner}> = <\text{Wert}> ; \\
 <\text{Schleife}> &= \text{while } <\text{Wert}> > <\text{Wert}> \text{ do } \{ <\text{Anweisung}> \} \text{ endwhile} \\
 <\text{Wert}> &= <\text{Bezeichner}> \\
 &\quad \mid <\text{Zahl}> \\
 &\quad \mid <\text{Wert}> - <\text{Wert}>
 \end{aligned}$$

Zur Ableitung der Tokenfolge aus dem Startsymbol *<Programm>* würden in der lexikalischen Analyse etwa folgende Ableitungsschritte gemacht:

$$\begin{aligned}
 <\text{Programm}> &\vdash^* <\text{Deklaration}>; <\text{Deklaration}>; <\text{Anweisung}>; <\text{Anweisung}>; \\
 &\vdash^* \text{int} <\text{Bezeichner}>; \text{int} <\text{Bezeichner}>; <\text{Zuweisung}>; <\text{Schleife}>; \\
 &\vdash^* ...
 \end{aligned}$$

Da das Programm aus dem Startsymbol ableitbar ist, ist es syntaktisch korrekt. Gleichzeitig mit den Ableitungsschritten wird in der lexikalischen Analyse ein abstrakter Syntaxbaum (Abbildung 3.8) aufgebaut, der nur die wesentliche Struktur des Programmes widerspiegelt. Die Programmstruktur kann ja gerade durch die Anwendung bestimmter Ableitungsregeln ermittelt werden. Wendet man die Regel für eine Schleife an einer bestimmten Stelle im Ableitungsprozess an, so liegt an dieser Stelle dann natürlich auch eine Schleife im Programmtext vor und diese Schleife erscheint demnach entsprechend an der entsprechenden Stelle im abstrakten Syntaxbaum. Im abstrakten Syntaxbaum werden alle Details der konkreten Syntax ignoriert. Das

Ergebnis dieser Phase ist der abstrakte Syntaxbaum, der an die nachfolgende Compilerphase weitergereicht wird.



### Semantische Analyse

In der semantischen Analyse, die oft mit der syntaktischen Analyse verzahnt ist, wird soweit möglich die semantische Korrektheit überprüft. Dazu gehört zum Beispiel die Überprüfung, ob ein Variablenbezeichner in einer Zuweisung auch vorher in einer Deklaration vorkam. In der semantischen Analyse wird der abstrakte Syntaxbaum um Attribute an den Knoten des Baumes erweitert, die Zusatzinformationen zu diesem Knoten mitführen, wie etwa den Typ eines Ausdrucks.

### Zwischencoderzeugung

Bevor der eigentliche Maschinencode erzeugt wird, wird in den meisten Compilern vorher ein Zwischencode erzeugt, der von der Struktur her ähnlich einer Assemblersprache ist, also recht nah an einer Machinensprache. Ein Vorteil des Zwischencodes gegenüber dem richtigen Maschinencode ist, dass die nachfolgenden Optimierungen unabhängig von einer konkreten Zentraleinheit ausgeführt werden können.

### Optimierung

In der **Optimierungsphase** finden auf dem Zwischencode Optimierungen statt, um den Code schneller zu machen oder so zu optimieren, dass weniger Ressourcen (zum Beispiel Hauptspeicher) notwendig sind.

### Codeerzeugung

Als letzte Phase wird in der **Codeerzeugungsphase** der Maschinencode erzeugt und in einer Objektdatei abgelegt. Erst in dieser letzten Phase ist man auf eine konkrete Zentraleinheit festgelegt. Die Umsetzung des Zwischencodes in Maschinencode ist meist einfach, jedoch können auch hier noch komplexe Optimierungen für die konkrete Zielzentraleinheit stattfinden.

### 3.3.3 Linker

Der Compiler erzeugt aus der Quelldatei in den oben beschriebenen Schritten eine Objektdatei, die für Java-Dateien die Endung `.class` hat (Objektcode für die Java Virtual Machine JVM), für andere Programmiersprachen auf Unix-Systemen die Namensendung `.o` und auf Windows-Systemen die Namensendung `.obj` besitzt. Diese Objektdatei ist noch nicht ausführbar. Ein Gesamtprogramm kann auf mehrere Quelldateien verteilt sein, zum Beispiel jedes Modul, das eine Teilaufgabe löst, kann in einer Quelldatei vorliegen. All diese Quelldateien können getrennt voneinander einzeln übersetzt werden mit dem Resultat, dass im fehlerfreien Fall  $n$  Objektdateien anschließend vorliegen.

In den Quelldateien können Referenzen zu anderen Modulen existieren, etwa Funktionen / Methoden anderer Klassen in Java. Aber nicht nur die eigenen entwickelten Methoden sind hier nutzbar. Oft gebrauchte Methoden wie etwa die Sinusfunktion oder die Ausgabe eines Zeichens auf dem Bildschirm werden vom System bereitgestellt.

### Beispiel 3.24:

Gegeben sei folgendes Programm in der hypothetischen Programmiersprache.

3-1	Programm MeinTest
3-2	read x;

```

3-3   y = sin(x);
3-4   z = Flaechenberechnung(x,y);
3-5   write z;
3-6 EndeProgramm.

```

Bevor ein Programm ausführbar ist, müssen alle Objektdateien zu einer Datei zusammengefügt werden und die Aufrufe zwischen den Modulen angepasst werden. Diesen Vorgang nennt man **Binden** (in deutschenglisch **Linken**) und das entsprechende Programm, das diesen Vorgang ausführt, nennt man **Linker**.

Oft gebrauchte Module werden vom System in Software-**Bibliotheken** bereitgestellt. Aus eigenen Modulen, die wiederverwendet werden können, kann man ebenso Software-Bibliotheken erzeugen. In Unix haben solche Bibliotheken für Nichtjavadateien die Namensendung **.a** (für *archive*) oder **.so**, in Windows haben sie die Namensendung **.dll** (für *dynamic link library*).

### 3.3.4 Virtuelle Maschinen

Generiert ein Compiler in der letzten Codegenerierungsphase Maschinencode für eine bestimmte Prozessorfamilie und einem bestimmten Betriebssystem, so ist dieser Code nur auf dieser Zielplattform ausführbar. Möchte man ein Programm für  $n$  solche Plattformen anbieten, so muss man dementsprechend  $n$  Binärprogramm erzeugen, was einen nicht zu unterschätzenden (Verwaltungs-)Aufwand bedeuten kann.

Ein moderner Ansatz ist nun, dass man selbst diesen letzten Schritt noch unabhängig von einer Zielplattform hält. Dies geschieht so, dass dazu ein virtueller, nicht real existierender Prozessor mit einem bestimmten Befehlssatz definiert wird und Code für diesen Prozessor im Compiler erzeugt wird. Da es dazu aber keinen realen Prozessor gibt, kann dieser Code natürlich auch nicht direkt auf irgendeinem Rechner ausgeführt werden. Dazu benötigt man dann zur Laufzeit des Programms (und *nicht* schon zur Übersetzungszeit) eine zusätzliche Laufzeitunterstützung, die den erzeugten Maschinencode des virtuellen Prozessors noch in Maschinencode des realen Prozessors auf diesem Rechner übersetzt. Diese Aufgabe erledigt ein weiterer Übersetzer, der jedoch sehr einfach ist und diese Aufgabe schnell durchführbar ist, weil dies mehr oder weniger eine 1:1 Übersetzung von Maschinenbefehlen ist.

Beispiele für diesen Ansatz sind die **Java Virtual Machine**(JVM) sowie von Microsoft die .NET-Umgebung mit der Prozessorsprache CIL (*Common Intermediate Language*) und der Laufzeitunterstützung CLR (*Common Language Runtime*).

## 3.4 Zusammenfassung und Hinweise

### Literaturhinweise

[HMU02] ist ein Standardwerk zu Grammatiken und Formalen Sprachen, [AB02] und [VW06] sind weitere Lehrbücher zu dieser Thematik. Die Originalarbeit zu (Chomsky-)Grammatiken findet sich in der Doktorarbeit von Noam Chomsky [Cho55]. Grammatiken werden dort als "generative Grammatiken" bezeichnet.

BNF hat seinen Ursprung in der Spezifikation der Sprache ALGOL 60 [NBB<sup>+</sup>63], die ihrerseits einen überaus wichtigen Einfluss auf die Entwicklung nahezu aller modernen Sprachen hatte. EBNF ist als ISO-Standard veröffentlicht [ISO96]. Syntaxdiagramme wurde erstmalig in der Spezifikation der Sprache Pascal verwandt [Wir73].

[ALSU08] ist ein umfassendes Standardlehrwerk zu Compilern. Weitere Bücher zu Compilern sind [Muc97] und [AK02].

Die Programmiersprache Java ist definiert in [Javc]. Dort ist im Anhang auch eine Grammatik zu Java in EBNF angegeben. In [Javb] sind die *Java Code Conventions* niedergeschrieben, die Formatierungs- oder "Schönschreibregeln" für Java Programme enthalten, an die man sich auch halten sollte.

## Verstehen

Sie sollten verstanden haben, aus welchen Grundbausteinen (Variablen, Ausdrücke, Anweisungen, ...) Programmiersprachen aufgebaut sind und wozu diese Grundbausteine genutzt werden. Weiterhin sollte erkannt worden sein, wie die Syntax von Programmiersprachen angegeben werden kann und welche Vor- und Nachteile diese verschiedenen Möglichkeiten haben. Die unterschiedlichen Ansätze der Ausführung von Programmen über Interpretierung und Übersetzung sollten verstanden sein.

## Kurz und knapp merken

- Programmiersprachen sind ein Kompromiss zwischen Menschen und Rechnern.
- Programmiersprachen existieren in verschiedenen Klassen, die auf einen bestimmten Zweck zielgerichtet sind (grob klassifiziert: maschinenorientiert, problemorientiert, universell).
- Die meisten höheren Programmiersprachen haben ähnliche Grundbestandteile.
- In Programmiersprachen gibt es Möglichkeiten zur Steuerung der Abfolge von Anweisungen und zur Manipulation von Daten/Werten.
- Bausteine, die in vielen Programmiersprachen vorkommen, sind:
  - Kommentare helfen mit umgangssprachlichen Formulierungen einem Leser dabei, das Programm besser und schneller zu verstehen. Sie haben keinen Einfluss auf die Bedeutung eines Programms.
  - Bezeichnern dienen dazu, verschiedenen Dingen in einem Programm einen Namen zu geben, um so darauf Bezug nehmen zu können.
  - Variablen speichern einen Wert, der sich im Laufe der Programmausführung ändern kann.
  - Ein Ausdruck beschreibt in formelmäßiger Form, wie genau ein Wert berechnet kann.
  - Eine Anweisung ist der Grundbaustein, mit den in vielen Programmiersprachen der Ablauf eines Programms beschrieben werden kann. Anweisungen können zum Beispiel in Form von Schleifen auch geschachtelt werden.
- Formale Sprachen sind eine Menge von Wörtern (Folge von Symbolen). Eine Sprache kann man über Regelwerke (Grammatiken) exakt definieren.
- Eine Ableitung ist eine Folge von Releanwendungen einer Grammatik. Kann man vom Startsymbol der Grammatik aus eine Ableitungsfolge zu einem Wirt nur aus Terminalsymbolen (Buchstaben) bestehend angeben, so ist dieses Wort in der Sprache enthalten.
- Programmiersprachen sind relativ einfach aufgebaut und sehr kompakt definiert. Sie lassen sich über eine Grammatik angeben. Zur Angabe einer solchen Grammatik stehen mehrere Möglichkeiten zur Verfügung, die letztendlich aber alle gleich ausdrucksstark sind (Chomsky-Grammatik, BNF, EBNF, Syntaxdiagramm).
- Zur Ausführung eines Programms gibt es zwei Ansätze, in denen jeweils die Analyse des Programms am Anfang steht. Ein Interpretierer geht Schritt für Schritt durch das Programm und führt Aktionen aus, die mit dem aktuell betrachteten Symbol im Zusammenhang stehen. Ein Compiler übersetzt einmalig den gesamten Programmtext in Maschinencode, der dann später direkt von der Zentraleinheit der Rechners ausgeführt werden kann.
- Virtuelle Maschinen ermöglichen es Compilern einheitlichen Code zu erstellen. Erst bei der Ausführung auf einem realen Rechner muss aber dann der Code für die virtuelle Maschine abgebildet werden auf den vorliegenden realen Rechner. Für die Sprache Java ist die Java Virtual Maschine definiert.

## Reflektion des Stoffs

- Geben Sie mehrere Gründe an, weshalb man maschinenorientierte Sprachen nur sehr selten zur Programmierung verwendet.
- Überlegen Sei sich eine beliebige Aufgabenstellung und überlegen anschließend, mit welcher Art von Werten Sie es dort zu tun haben könnten.
- Wozu dienen Bezeichner in Programmiersprachen? Wo haben Sie bereits außerhalb solcher Sprachen mit Bezeichnern zu tun gehabt?
- Können man folgende Sachen durch eine Typ-3-Grammatik angeben: Bezeichner, Ausdruck, Anweisung?
- Erläutern Sie die praktischen Vorteile der BNF und EBNF.
- Welche Vor- und Nachteile haben Syntaxdiagramme?
- Was sind die Vor- und Nachteile, wenn man in einem Compiler statt direktem Binärkode für eine bestimmte Zentraleinheitfamilie statt dessen Code für eine virtuelle Maschine wie die JVM erzeugt?



# Chapter 4

## Einfache Datentypen

In den nächsten Kapiteln werden die Grundbausteine von Programmiersprachen vorgestellt, aus denen letztendlich auch komplexe Programme aufgebaut sind. Einige dieser Aspekte wie Variablen, Ausdrücke und Anweisungen wurden zur Motivation bereits vorab in Kapitel 3.1 übersichtsartig vorgestellt. Diese Grundbausteine teilen sich einerseits auf in die Kontrollstrukturen, die den Ablauf eines Programms beeinflussen (in welcher Reihenfolge etwas getan werden soll) und andererseits in Möglichkeiten, wie man mit Daten in Programmen umgehen kann (Darstellung von Daten und mögliche Operationen auf diesen Daten); letzteres wird in diesem Kapitel zuerst behandelt.

Neben der Formulierung der Abfolge von Befehlen (Algorithmus) ist eine weitere Fragestellung die, wie im Algorithmus zu verarbeitende und vorkommende Daten behandelt werden können. Welche Daten gibt es, gibt es Gemeinsamkeiten zwischen Daten, wo sind die Unterschiede, wie lassen sich komplexe Daten behandeln? In den bisherigen einfachen Beispielen waren die Ein- und Ausgabedaten sowie auch Zwischenergebnisse einfache natürliche Zahlen, die mit ihrer allseits bekannten Bedeutung und den üblichen Operationen (Addition, Subtraktion usw.) genutzt wurden. Spätestens wenn komplexere Daten verarbeiten werden sollen stellt sich die Frage, was sind genau Daten, wie kann man Daten in einem Rechner darstellen und was kann man mit ihnen machen, das heißt welche Operationen lassen sich auf ihnen ausführen. Diese Fragen werden über den Begriff des **Datentyps** beantwortet.

### 4.1 Informationsdarstellung in Rechnern

Auch wenn sich in diesem Zusammenhang vornehmlich mit der Programmierung an sich beschäftigen werden soll, also losgelöst von einem bestimmten Computer, muss man sich doch über gewisse Randbedingungen im Klaren sein, die durch die Eigenschaften von Hardware allgemein vorgegeben sind, und diese Eigenschaften treten in allen Programmiersprachen auch zutage. Der folgende einführende Teil dient dazu, eine kurze Übersicht über die hier relevanten Eigenschaften von Computern zu geben und dient keineswegs einer vertieften Einführung in die Rechnerarchitektur (siehe dazu entsprechende Literaturhinweise).

Programme werden heute auf Digitalrechnern (im Gegensatz zu Analogrechnern) ausgeführt, die alle gewisse gemeinsamen Fähigkeiten haben, aber auch andererseits allgemeinen Einschränkungen unterliegen, die einem Programmierer bewusst sein müssen. Egal, ob es sich um ein Mobiltelefon, Waschmaschinensteuerung, Heimcomputer oder Multimillionen-Euro-Supercomputer handelt, alle diese Computer haben die Gemeinsamkeit, dass sie alle Daten als eine Folge von Bits und Bytes darstellen, und dies auch (heutzutage) in einer einheitlichen Form tun.

Ein **Bit** (Abkürzung des englischen Begriffs *binary digit*) ist eine atomare Informationseinheit, die genau zwei Zustände annehmen kann und damit zwei mögliche Werte darstellen kann, die mit den Ziffern 0 und 1

beschrieben werden sollen. Ein einziges Bit kann konkret dargestellt werden durch verschiedene physikalische Ausprägungen, die jeweils die beiden möglichen Zustände repräsentieren. Beispiele sind Licht an / Licht aus, magnetisiert / nicht magnetisiert, elektrische Ladung vorhanden / elektrische Ladung nicht vorhanden.

Mit einem Bit lassen sich also genau zwei verschiedene Zustände darstellen, die Information in einem Bit gibt an, welcher der beiden Zustände vorliegt. Man muss aber natürlich festlegen, welcher Zustand für welchen Informationsgehalt steht. Dies nennt man eine **Codierung**. Bekannte Codierungen allgemeiern Natur, die eher im Alltagsleben zu finden sind, sind etwa der Morsecode, in dem jedem Buchstaben eine Folge von Kurz- und Langsignalen (Punkt oder Strich) zugeordnet ist, die Braille-Schrift (Blindenschrift) und der Flaggencode in der Schiffahrt. Genauso ist aber auch das lateinische Alphabet für Buchstaben oder die arabischen Ziffern für Zahlenwerte eine Codierung. Wichtig ist an dieser Stelle zu verstehen, dass mit einem Bit zwei Zustände dargestellt werden; was diese Zustände *bedeuten*, legt eine Codierung fest. Dem gleichen Bit könnte man also verschiedene Bedeutungen zumessen, je nach Interpretation des Zustands.

#### **Definition 4.1 (Codierung):**

Eine Codierung ist eine Abbildung, die mögliche Zustände auf zugehörige Werte eines Wertebereichs abbildet. ♦

#### **Beispiel 4.1:**

Beispiele für Codierungen sind etwa  $f : \{0, 1\} \rightarrow \mathbb{N}$ ,  $g : \{0, 1\} \rightarrow \mathbb{N}$ ,  $h : \{0, 1\} \rightarrow \{\text{blau}, \text{grün}\}$  mit:

- $f(0) = 0, f(1) = 1$
- $g(0) = 1, g(1) = 0$
- $h(0) = \text{blau}, h(1) = \text{grün}$



Ist die Anzahl der möglichen Zustände grösser als zwei, so kommt man mit einem Bit zur Darstellung all dieser möglichen Zustände natürlich nicht mehr aus und man benötigt mehrere Bits. Hat man  $n$  Bits zur Verfügung, so kann man damit genau  $2^n$  verschiedene Zustände und damit Werte darstellen, nämlich über alle möglichen Permutationen der einzelnen Bitzustände.

#### **Beispiel 4.2:**

Samstag Abend, es ist Party-Zeit. Eingeladen sind Albert, Annette und Frank. Es soll dargestellt werden, wer zu einem bestimmten Zeitpunkt auf der Party anwesend ist und wer nicht. Dazu wird jedem potentiellen Teilnehmer ein Bit zugeordnet: Bit Albert oder  $b_0$ , Bit Annette oder  $b_1$  und Bit Frank oder  $b_2$ . Mit diesen drei Bit  $b_0, \dots, b_2$  lassen sich alle  $2^3 = 8$  möglichen Teilnahmezustände darstellen (Teilnehmer x ist anwesend oder nicht):

Frank	Annette	Albert	Bit-Folge $b_2b_1b_0$	Zustandsnummer
nicht da	nicht da	nicht da	000	$0 = 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$
nicht da	nicht da	da	001	$1 = 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
nicht da	da	nicht da	010	$2 = 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$
nicht da	da	da	011	$3 = 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$
da	nicht da	nicht da	100	$4 = 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$
da	nicht da	da	101	$5 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
da	da	nicht da	110	$6 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$
da	da	da	111	$7 = 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$

Faktor	Präfix	Zeichen	Faktor	Präfix	Zeichen	dezimal
$10^3$	Kilo	K	$2^{10}$	Kibi	Ki	1.024
$10^6$	Mega	M	$2^{20}$	Mebi	Mi	1.048.576
$10^9$	Giga	G	$2^{30}$	Gibi	Gi	1.073.741.824
$10^{12}$	Tera	T	$2^{40}$	Tebi	Ti	$\approx 1,099 \times 10^{12}$
$10^{15}$	Peta	P	$2^{50}$	Pebi	Pi	$\approx 1,125 \times 10^{15}$
$10^{18}$	Exa	E	$2^{60}$	Exbi	Ei	$\approx 1,152 \times 10^{18}$
$10^{21}$	Zetta	Z	$2^{70}$	Zebi	Zi	$\approx 1,180 \times 10^{21}$
$10^{24}$	Yotta	Y	$2^{80}$	Yobi	Yi	$\approx 1,208 \times 10^{24}$

Table 4.1: SI-Präfixe (links) und IEC-Präfixe (rechts)

Man kann daran aber schon erkennen, dass die Zuordnung eines Bits zu einem Teilnehmer wichtig ist. Die Bit-Folge 100 (nur Frank da) hat eine andere Bedeutung als 001 (nur Albert da).

In der Spalte "Zustandsnummer" ist eine Nummerierung der acht verschiedenen Zustände von 0-7 angegeben sowie ein Verfahren, wie sich diese Nummer eindeutig aus den Einzelzuständen der drei Bits bestimmen lässt. Das wird zu einem späteren Zeitpunkt noch genutzt. ♦

#### Definition 4.2 (höchst-,niederwertige Ziffer):

In einem Zahlensystem, in der die Reihenfolge von Ziffern eine Rolle spielt (Stellenwertsystem), nennt man die Ziffer, die für die geringste Wertigkeit steht (in unseren Zahlensystemen üblicherweise rechts stehend), die **niederwertigste Ziffer**, die Ziffer zur höchsten Wertigkeit nennt man **höchstwertige Ziffer**. ♦

Eine Folge von 8 Bits bezeichnet man als **Byte**. Alle Daten werden in Rechnern letztendlich in Vielfachen von Bytes gespeichert, zum Beispiel ganze Zahlen in 1, 2, 4 oder 8 Bytes, entsprechend 8, 16, 32 oder 64 Bits.

Bei großen Datenmengen (Festplattenkapazitäten, Hauptspeichergrößen,...) werden viele Bytes zur Speicherung dieser Daten benötigt. Die bekannten SI-Präfixe wie Kilo und Mega, wie sie in Tabelle 4.1 aufgelistet sind – das heißt bezogen auf die Basis 10 – führen leider zu einiger Verwirrung bei Größenangaben bezogen auf Byte. Die Verwendung dieser Präfixe im Zusammenhang mit Größenangaben wie zum Beispiel Hauptspeichergöße oder Kapazität von Festplatten (Festplatte hat eine Kapazität von xx Gigabyte) ist nicht immer einheitlich und bezieht sich bedauerlicherweise manchmal auf die Basis 2, das heißt eine Angabe wie 2 Megabyte bedeutet dann nicht etwa  $2 * 10^6$  (Basis 10) sondern  $2 * 2^{20} = 2 * 1.048.576$ . Um die Verwirrung noch größer zu machen, geben zum Beispiel Festplattenhersteller die Kapazität ihrer Festplatten in der üblichen Notation mit der Bedeutung Giga=10<sup>9</sup> an, einige Betriebssysteme melden die Größe dieser Festplatte aber in Gigabyte mit der Bedeutung Giga=2<sup>30</sup>.

Die *International Electrotechnical Commission (IEC)* hat sich um eine Klärung dieser Problematik bemüht und einen Standard akzeptiert, der explizit zusätzliche Präfixe für Größenangaben bezogen auf Zweierpotenzen definiert. Der IEC-Standard ist jedoch (noch) nicht weit verbreitet. Tabelle 4.1 gibt eine Übersicht über die IEC-Einheiten.

Wenn man sich im Folgenden Gedanken darüber machen muss, wie verschiedene Datenarten (Zahlen, Buchstaben, Vektoren, Matrizen, MP3-Musikstücke, Bestelllisten, Datenbankinhalte,...) in einem Rechner dargestellt werden können, muss man sich also jeweils adäquate Codierungen überlegen. Adäquat heißt in diesem Zusammenhang, dass die Codierung geeignet sein muss, mögliche Werte darzustellen (das wird aber leider nicht immer gelingen) und dass gängige Operationen auf diesen Werten ebenfalls möglich sein müssen.

Bei den nachfolgenden Diskussionen sollte im Hinterkopf behalten werden, dass mit  $n$  Bits genau  $2^n$  mögliche verschiedene Werte codiert werden können. Das heißt im Umkehrschluss, dass alle Werte unendlicher Zahlenmengen wie etwa  $\mathbb{N}, \mathbb{Z}$  oder  $\mathbb{R}$  *prinzipiell nicht* mit einer festen Anzahl von Bits darstellbar sind.

Weil unterschiedliche Datenarten unterschiedliche Anforderungen haben, wie zum Beispiel an den darstellbaren Bereich oder die Genauigkeit von Operationen, werden in Programmiersprachen unterschiedliche Basisdatentypen angeboten mit unterschiedlicher Codierung (interner Repräsentation) und unterschiedlichen darauf definierten Operationen. Eine solche Datenart nennt man in Programmiersprachen einen **Datentyp**.

### Definition 4.3 ((konkreter) Datentyp):

In einem (konkreten) Datentyp werden festgelegt:

- eine Wertemenge
- die auf dieser Wertemenge definierten Operationen
- eine Codierung der Werte



### Beispiel 4.3:

Betrachtet werden soll die Menge  $\mathbb{N}$  der natürlichen Zahlen.

- Die Wertemenge ist  $\mathbb{N} = \{0, 1, 2, \dots\}$ .
- Darauf seien folgende Operationen definiert:

$$\text{addieren} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$\text{subtrahieren} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

mit der üblichen Bedeutung der Addition und folgender Bedeutung für die Subtraktion (zu beachten ist, dass Operationen Werte innerhalb der Wertemenge produzieren müssen):

$$\text{subtrahieren}(x, y) := \begin{cases} x - y & \text{falls } x > y \\ 0 & \text{sonst} \end{cases}$$

- Die Codierung entspricht dem bekannten Dezimalsystem (Die Codierung mit Hilfe von Bits wird später betrachtet).



Wichtig ist also jetzt schon zu erkennen, dass es nicht "die Zahlen" in Programmiersprachen gibt, sondern sehr genau unterschieden werden muss, mit welchen Wertebereichen man es zu tun hat (wie klein / groß die Zahlen werden können), welche Eigenschaften die Werte haben sollen und welche Operationen möglich sein sollen. Die Unterscheidung in verschiedene Datentypen ist nicht ungewöhnlich, denn zum Beispiel in der Schulmathematik gibt es auch die Unterscheidung zwischen natürlichen Zahlen  $\mathbb{N}$ , ganzen Zahlen  $\mathbb{Z}$ , rationalen Zahlen  $\mathbb{Q}$  und reellen Zahlen  $\mathbb{R}$ . Diese Zahlenbereiche haben jeweils gewisse Wertebereiche und Operationen. Sie sind für bestimmte Berechnungen besser oder weniger gut geeignet sind. Wie man schnell anhand einiger einfacher Beispiele einsieht ist das Rechnen mit natürlichen Zahlen für einen Menschen oft sehr viel einfacher als das Rechnen mit reellen Zahlen. So kann man im Kopf schnell  $8 \cdot 7$  ausrechnen, aber  $8,3 \cdot 7,8$  fällt deutlich schwerer. Andererseits reichen für viele Berechnungen natürliche Zahlen aber nicht mehr aus.

Im Folgenden wird konkret untersucht, welche Grunddatentypen in der Programmiersprache Java vorhanden sind und wie die Werte als Bit-Folgen repräsentiert werden. Diese Grunddatentypen sind in mehr oder

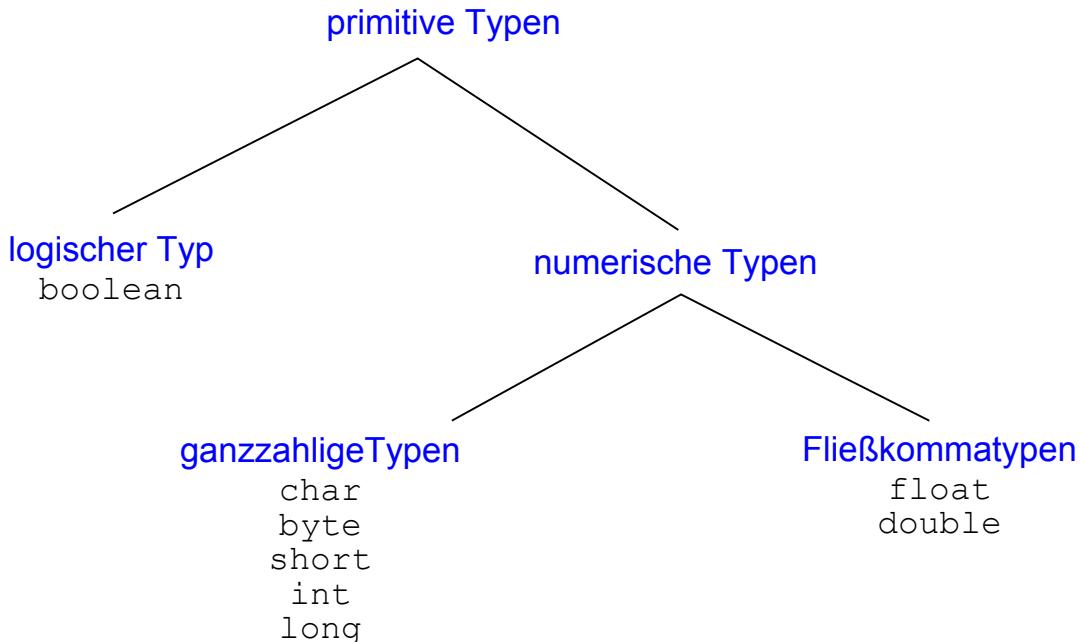


Figure 4.1: Primitive Datentypen in Java

weniger identischer Form in vielen anderen Programmiersprachen auch vorhanden, eventuell mit einer anderen Bezeichnung (Beispiel: `real` statt `float`). Die Aussagen, die hier für die Java-Basistypen gemacht werden, sind aber übertragbar auf andere Sprachen. In Java werden diese Grunddatentypen als **primitive Datentypen** bezeichnet. Abbildung 4.1 zeigt eine Übersicht und die Strukturierung dieser primitiven Datentypen in Java, auf die im Laufe des Kapitels eingegangen wird.

Alle später folgenden Datentypen sind im Endeffekt aus diesen wenigen Grunddatentypen zusammengesetzt. Neue komplexere Datentypen können aus diesen Basisdatentypen durch unterschiedliche Strukturierungsmöglichkeiten aufgebaut werden, was später im Kapitel 9 behandelt wird. Die nachfolgenden Unterkapitel stellen vorher alle Grunddatentypen von Java vor.

In der Betrachtung der einzelnen Datentypen wird jeweils zu diskutieren sein, was der entsprechende Wertebereich ist, wie die einzelnen Werte über Bit-Folgen dargestellt werden, wie Werte dieses Typs in einem Programm angegeben werden und wie sich mit Hilfe von Operatoren Ausdrücke formen lassen.

In Kapitel 3.1.5 wurde schon einführend diskutiert, dass Ausdrücke syntaktisch mit Hilfe von Operatoren und Operanden aufgebaut werden können. Weiterhin kann man zu jedem Ausdruck einen Wert berechnen – der Wert dieses Ausdrucks –, indem man die Operatoren sukzessive auf die Operanden anwendet. Beispielweise gilt:  $(3 + 4) * 5 = 35$ , weil  $(3 + 4) = 7$  und  $7 * 5 = 35$  gilt. Analog wird die mit den Operatoren der vorgestellten Datentypen sein. Auf einige Aspekte im Zusammenhang mit der Auswertung von Ausdrücken wird später noch im Detail eingegangen (Kapitel 7). Die nachfolgend in den einzelnen Unterkapiteln gezeigten Syntaxdiagramme geben nur einen Teil der syntaktisch möglichen Ausdrücke wieder. So fehlen etwa Methodenaufrufe, bestimmte Zugriffe auf Variablen oder Zuweisungsoperatoren, die größtenteils erst in späteren Kapitel behandelt werden. Ebenso werden Prioritäten in Operatoren in diesen Diagrammen hier nicht berücksichtigt. Für eine vollständige Referenz sei auf die Literatur verwiesen.

## 4.2 Wahrheitswerte

Es gibt genau zwei Wahrheitswerte: *wahr* und *falsch*: In einem Java-Programm werden diese beiden Wahrheitswerte mit den Namen `true` und `false` bezeichnet, der zugehörige Java-Datentyp heißt `boolean`. Wahrheitswerte werden in Programmen etwa dazu genutzt, nur in bestimmten Fällen, etwa wenn der Wahrheitswert `true` vorliegt, eine bestimmte Aktion durchzuführen; wenn der Wert `false` aber vorliegt, will man diese Aktion nicht durchführen. Über Wahrheitswerte werden etwa Selektionen und Iterationen gesteuert, die später noch im Detail (Kapitel 5.5 und Kapitel 5.4) behandelt werden.

Die rechnerinterne Repräsentation der beiden booleschen Werte ist in den gängigen Programmiersprachen unterschiedlich. Eigentlich bräuchte man zur Darstellung der beiden möglichen Zustände `true` und `false` nur ein Bit, zum Beispiel mit der Codierung, dass 1 dem Wert `true` entspricht und 0 dem Wert `false`. Aus verschiedenen Gründen (zum Beispiel effizienterer Zugriff auf der Hardware-Schicht) werden jedoch zur Repräsentation von booleschen Werten Bitfolgen aus acht (1 Byte) oder mehr Bit genutzt. Der Wert `false` wird dann bei Mehrbitdarstellungen als eine Folge von Null-Bit repräsentiert ( $0\dots 0_2$ ), der Wert `true` entweder durch eine 1 im niederwertigsten Bit dieser Bit-Folge ( $0\dots 01_2$ ; dies ist die Lösung in Java) oder aber jede Bit-Folge außer der Folge, die sich ausschließlich aus Null-Bits zusammensetzt, repräsentiert den Wert `true` ( $0001_2$  oder  $0011_2$  und so weiter; diese Lösung wird etwa in C und C++ genutzt).

Ausdrücke, die einen Wahrheitswert liefern, lassen sich mit den aus der Logik bekannten Operatoren Und  $\wedge$ , Oder  $\vee$ , Exklusives Oder  $\oplus$  und Nicht  $\neg$  aufbauen, die jeweils zwei Wahrheitsausdrücke (beziehungsweise einen bei der Negation) als Operanden benötigen. In Java werden diese Operatoren dargestellt durch die Symbole `&` und `&&` für Und, `|` und `||` für Oder, `^` für Exklusives Oder und `!` für Negation (siehe Tabelle 4.2). Auf die Unterschiede zwischen `&` und `&&` für das logische Und sowie `|` und `||` für das logische Oder wird weiter unten eingegangen. Auch lassen sich zwei Wahrheitswerte auf Gleichheit und Ungleichheit testen (in Java mit den Operatoren `==` und `!=`), das Resultat ist wiederum ein Wahrheitswert. Weiterhin liefert der Vergleich zweier Zahlenwerte mit den üblichen Vergleichsoperatoren wie `<` oder `<=` einen Wahrheitswert. Abbildung 4.2 zeigt die Syntaxdiagramme zu logischen Ausdrücken mit möglichen Operatoren.

In der folgenden Tabelle sind einige einfache Beispiele zu den Java-Operatoren des Java-Datentyps `boolean` aufgeführt.

Java-Operator	Beispiel	Wert des Beispiels	Wirkung
<code>==</code>	<code>true == false</code>	<code>false</code>	Test auf Gleichheit
<code>!=</code>	<code>true != false</code>	<code>true</code>	Test auf Ungleichheit
<code>!</code>	<code>!true</code>	<code>false</code>	Negation
<code>&amp;&amp;</code>	<code>true &amp;&amp; true</code>	<code>true</code>	Und
<code>&amp;</code>	<code>true &amp; true</code>	<code>true</code>	Und
<code>  </code>	<code>true    false</code>	<code>true</code>	Oder
<code>^</code>	<code>true ^ false</code>	<code>true</code>	Exklusives Oder

Aus dem Syntaxdiagramm ist ersichtlich, dass logische Ausdrücke mithilfe der verfügbaren Operatoren auch andere logische oder bei Verwendung relationaler Ausdrücke auch arithmetische Ausdrücke enthalten können. Komplexere logische Ausdrücke, die bei der Auswertung des Ausdrucks natürlich ebenfalls nur `true` oder `false` liefern, werden dann eingesetzt, wenn die Bedingung von mehreren Teilbedingungen abhängt. Seien `x`, `y` und `z` drei Variablen mit Zahlen als Inhalt, die die Zahlenwerte 3, 5 und 7 enthalten. Dann sind folgende logische Ausdrücke beispielsweise möglich:

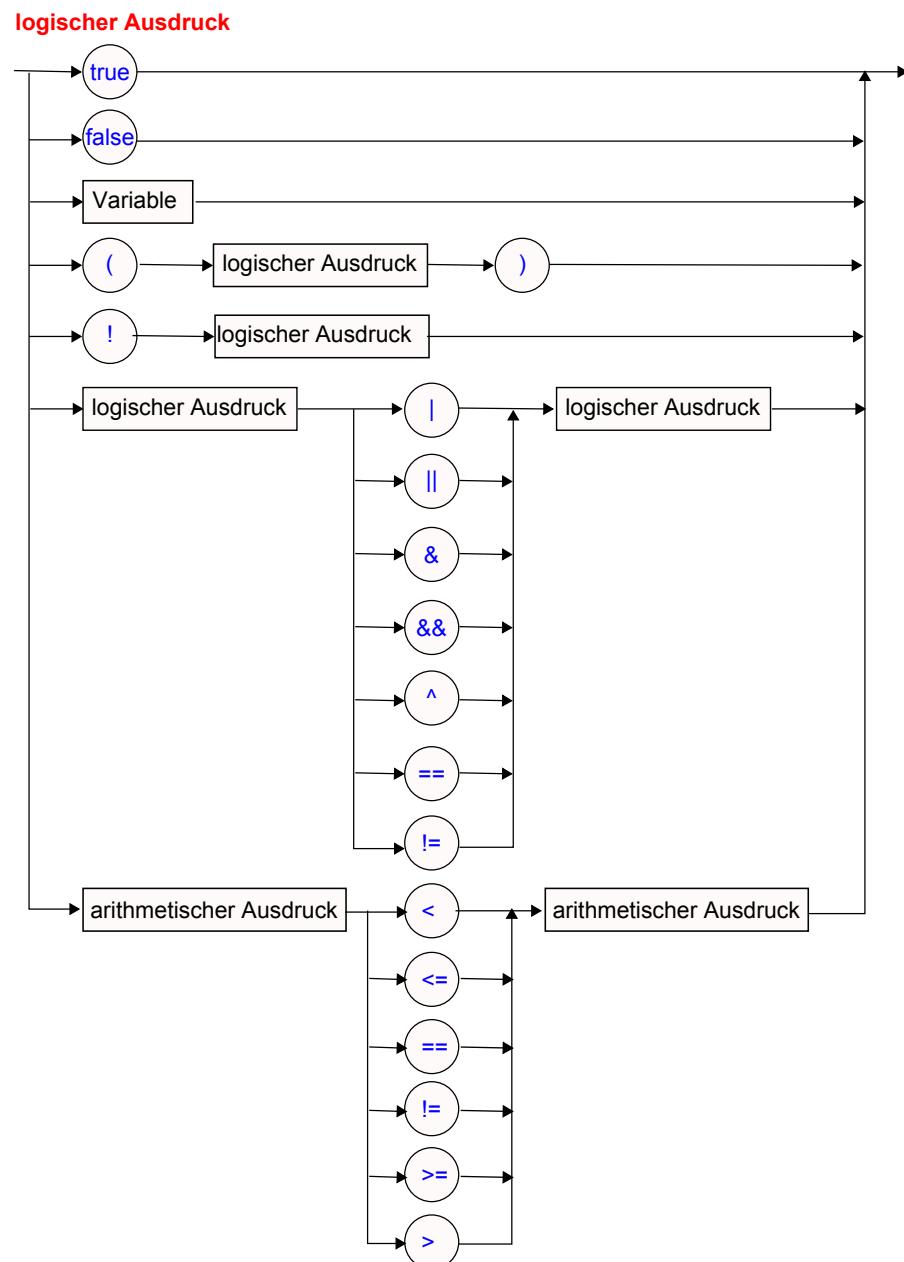


Figure 4.2: Syntaxdiagramm für logische Ausdrücke

Beispiel	Wert des Beispiels	Test worauf?
<code>x &lt; y</code>	<code>true</code>	x kleiner als y
<code>x &lt; y &amp;&amp; x &lt; z</code>	<code>true</code>	x gleichzeitig kleiner als y und z
<code>x &lt; y &amp;&amp; y &lt; z</code>	<code>true</code>	y liegt zwischen x und z
<code>0 &lt;= x &amp;&amp; x &lt;= 9</code>	<code>true</code>	x liegt im Intervall [0,9]
<code>!(0 &lt;= x &amp;&amp; x &lt;= 9)</code>	<code>false</code>	x liegt außerhalb des Intervalls [0,9]
<code>x &lt; 0    x &gt; 9</code>	<code>false</code>	x liegt außerhalb des Intervalls [0,9]

Beispielsweise würde der erste Ausdruck in der Tabelle `x < y` bei der Auswertung dieses Ausdrucks etwa testen, ob der aktuelle Inhalt der Variablen `x` kleiner als der aktuelle Inhalt der Variablen `y` ist.

Zu beachten ist, dass Formulierungen wie  $0 \leq x \leq 9$  aus der Mathematik in Programmiersprachen nicht direkt umsetzbar sind, sondern wie in den vorangegangenen Beispielen gezeigt mit Hilfe von zwei Teilformeln und der Konjunktion formuliert werden müssen.

Möchte man Wahrheitswerte speichern / zur späteren Weiterverwendung behalten, so benötigt man eine Variable, die einen Wahrheitswert aufnehmen kann (siehe Kapitel 3.1.4 zu einleitenden Bemerkungen über Variablen). Der entsprechende Datentyp heißt wie oben bereits angegeben `boolean`. Variablen vom Typ `boolean` sind nach dem Syntaxdiagramm überall dort verwendbar, wo auch logische Ausdrücke verwendbar sind.

#### Beispiel 4.4:

Die nachfolgende Diskussion setzt auf dem auf, was in Kapitel 2.4.5 bereits zum Programmgerüst von Java-Programmen und zu den allgemeinen Bestandteilen von Programmen in Kapitel 3.1 gesagt wurde. Dies soll zu Beginn aber noch einmal kurz zusammengefasst werden. Ein einfaches Java-Programm, wie es bis auf Weiteres betrachtet werden soll, hat immer den gleichen Aufbau. Ein solches Programm beginnt mit einer Klassendefinition der Form `public class Klassenname {`, wobei der Name der Klasse beliebig wählbar ist. Darin eingebettet ist die Definition einer main-Methode der Form `public static void main(String[] args) {` und darin dann wiederum eingebettet der eigentliche Programmtext. Beendet wird dies durch jeweils eine schließende geschweifte Klammer für die Zeile mit `main` und eine Klammer für die Zeile mit `class`.

Ein Programm wie das Gezeigte wird ausgeführt, indem ausgehend von dem, was in `main` eingebettet ist, Anweisung für Anweisung (oder vielleicht zu Beginn leichter verständlich: Zeile für Zeile) abgearbeitet wird. Das was in einer Anweisung / Zeile steht, wird entsprechend der Bedeutung ausgeführt und daran anschließend die nächste Anweisung / Zeile ausgeführt, bis die schließende geschweifte Klammer zu `main` erscheint.

Im Programm in Listing 4.1 ab Zeile 6 werden zwei neue Variablen deklariert mit Namen `b1` und `b2` vom Datentyp `boolean`. Zur Erinnerung: jede Variable muss in einem Programm vor ihrer Nutzung deklariert werden. Für jede deklarierte Variable wird im Hauptspeicher Platz reserviert, der immer den aktuellen Wert dieser Variablen enthält. Anschließend werden in Zeile 11 und 12 diesen beiden Variablen jeweils ein Wahrheitswert zugewiesen.

Bevor auf die Zeilen 15-17 eingegangen wird, vorher die Diskussion zu den Zeilen 20 und 24. In Zeile 20 wird in der Anweisung `b1 = (4 < 3);` zuerst der Ausdruck `(4 < 3)` ausgewertet, was als Ergebnis den Wert `false` liefert. Dieser Wert wird dann anschließend der Variablen `b1` zugewiesen, also die rechnerinterne Darstellung des Wertes `false` in dem Speicherplatz abgelegt, der der Variablen `b1` zugeordnet ist.

In Zeile 24 `b2 = !b1 && (3 < 4);` wird analog vorgegangen. Zuerst wird der Ausdruck auf der rechten Seite des Gleichheitszeichens ausgerechnet. Dazu wird zuerst der aktuelle Wert der Variable `b1` ermittelt (also der Wert, der in Zeile 20 in dieser Variablen gespeichert wurde). Dieser Wert (`false`) wird aufgrund des Negationsoperators `!` negiert, was als Ergebnis dieses Teilausdrucks `!b1` das Zwischenergebnis `true` liefert. Dann wird der Teilausdruck `(3 < 4)`, was als Ergebnis den Wert `true` liefert. Und zum Schluss werden diese beiden Teilergebnisse mit Hilfe des Und-Operators `&&` zu dem Gesamtergebnis `true` zusammengefasst. Dieses Gesamtergebnis des Ausdrucks wird dann der Variablen `b2` zugewiesen.

Listing 4.1: Beispiel zum Datentyp boolean.

```
41 /**
42 * Einige Beispieloperationen mit boolschen Werten
43 */
44 public class BoolBeispiel {
45     public static void main(String[] args) {
46
47         // zwei Variablen vom Typ boolean definieren
48         boolean b1, b2;
49
50         // diesen beiden Variablen jeweils einen Wert zuweisen
51         b1 = true;
52         b2 = false;
53
54         // Wir geben das Ergebnis einiger Operationen aus
55         System.out.println("true == false: " + (true == false));
56         System.out.println("true ^ false: " + (true ^ false));
57         System.out.println("b1 || b2: " + (b1 || b2));
58
59         // Test, ob 4 kleiner als 3 ist. Das Ergebnis speichern wir in b1
60         b1 = (4 < 3);
61
62         // Negation des Wertes, der in der Variablen b1 steht
63         // Und-Verknuepfung mit dem Test, ob 3 kleiner als 4 ist
64         b2 = !b1 && (3 < 4);
65
66         // Ausgabe des aktuellen Inhalts der Variablen auf dem Bildschirm
67         System.out.println("Der Wert von b1 ist jetzt " + b1);
68         System.out.println("Der Wert von b2 ist jetzt " + b2);
69     }
70 }
```

Also allgemein: in einer Zuweisung der Form `variablenname = Ausdruck;` wird zuerst der Ausdruck auf der rechten Seite des Gleichheitszeichens ausgewertet. Ist der Ausdruck etwas komplexer, so gibt es genaue Regeln, wie dies zu geschehen hat, worauf später auch eingegangen wird. Das Resultat des Ausdrucks ist aber immer genau ein Wert, egal wie komplexe der Ausdruck ist. Dieser Wert wird anschließend in Hauptspeicher dort abgelegt, der der Variablen zuordnet ist, die auf der linken Seite des Gleichheitszeichens steht.

Jetzt zu den Zeilen, die die Form haben `System.out.println(...)`. Zur Erinnerung: eine Zeile mit `System.out.println("beliebiger Text");` führt auf dem Bildschirm eine Ausgabe aus. Alles, was in Anführungszeichen steht, wird genau so auf dem Bildschirm ausgegeben (analog einem Zitat). Gegenüber dem früheren Beispiel in Kapitel 1.2 ist die Angabe dessen, was auszugeben ist, aber erweitert worden. Innerhalb der runden Klammern steht in der ersten Ausgabeanweisung etwa

```
4-1   "true == false: " + (true == false)
```

bestehend aus einem bekannten Textformat in Anführungszeichen "`true == false:`" gefolgt von einem Pluszeichen + gefolgt von einem Ausdruck in Klammern (`true == false`).

Die Abarbeitung dieser Ausgabeanweisung geschieht nun von links nach rechts gehend folgendermaßen (die allgemein gültigen Hintergründe und Details dazu in Kapitel 7.4). Zuerst wird der Text in Anführungszeichen als Zitat erkannt. Das nachfolgende Plus ist ein Operator, der in unterschiedlichen Kontexten auch unterschiedliche Bedeutungen haben kann, zum Beispiel die Addition zweier Werte, wenn die beiden Operanden arithmetische Ausdrücke sind. Ist ein Operand wie im vorliegenden Beispiel ein Text (in Java ist der entsprechende Datentyp `String`), so wird der andere Operand automatisch ebenfalls in einen Text umgewandelt. Der zweite, rechte Operand im Beispiel ist der logische Ausdruck (`true == false`). Dieser Ausdruck wird ausgewertet, was als Ergebnis `false` ergibt. Wie eben diskutiert, ist die interne Darstellung dieses Wertes eine Folge von Null-Bits. Aufgrund der Information, dass es sich um einen Wahrheitswert (Datentyp `boolean`) handelt und welche Bitfolge vorliegt, werden diese Bits nun entsprechend interpretiert und entweder der Text "`false`" (in unserem Fall) oder der Text "`true`" erzeugt. Der Plusoperator fügt in vorliegenden Zusammenhang die beiden Texte (`true == false:` und `false`) zu einem Gesamttext aneinander, so dass als Ausgabe auf dem Bildschirm erscheint:

```
4-1   true == false: false
```

Diese automatische Umwandlung des Wertes eines Ausdrucks in einen Text ist nicht nur mit Wahrheitswerten möglich, sondern mit *allen Werten* in Java! Versuchen Sie es! Die anderen Anweisungen mit `System.out.println` erzeugen nun ebenfalls jeweils eine Ausgabezeile. ♦

Oben wurden die beiden Operatorenpaare `&` und `&&` für Logisches Und sowie `|` und `||` für logisches Oder vorgestellt. Wo ist nun der Unterschied zwischen den beiden Varianten für Und beziehungsweise Oder? Bei der einfachen Version `&` und `|` werden die beiden Operandenausdrücke ausgewertet (der linke zuerst, dann der rechte), das Resultat für die beiden Operandenausdrücke ist jeweils `true` oder `false`. Anschließend werden zur Ermittlung des Wertes des Gesamtausdrucks diese beiden Wahrheitswerte miteinander entsprechend der Operation verknüpft, was als Gesamtergebnis `true` oder `false` liefert. Die Behandlung für die doppelten Symbolzeichen ist allerdings eine andere. Hier wird zuerst der linke Operand ausgewertet. Steht nach Ermittlung dieses Wertes das Gesamtresultat bereits fest (`true` bei Oder, `false` bei Und), so wird der zweite Operand *nicht mehr ausgewertet*. Steht das Gesamtergebnis nach Auswertung des ersten Operanden noch nicht fest, so wird der zweite Operand ebenfalls ausgewertet und die beiden Operandenwerte entsprechend verknüpft.

### Beispiel 4.5:

Im folgenden Programmausschnitt

4.1

(3 &lt; 4) || (4 &gt; 5)

liefert die Auswertung des linken Operandenausdrucks `(3 < 4)` den Wahrheitswert `true`. In einer Oder-Verknüpfung ist das Gesamtergebnis `true`, wenn einer der beiden Argumentwerte `true` ist. Insofern wäre es in diesem Fall gerechtfertigt, wenn auf die Auswertung des rechten Operandenausdrucks verzichtet wird, da das Gesamtergebnis durch den linken Operandenwert ja bereits feststeht (`true`). ♦

In Kapitel 7.2 wird auf diesen Sachverhalt nochmals im Detail eingegangen und auch auf mögliche Probleme beziehungsweise Vorteile mit den Doppelsymboloperatoren eingegangen.

## 4.3 Ganze Zahlen

Nachdem die Behandlung von Wahrheitswerten in Programmiersprachen diskutiert wurde stellt sich als nächste Frage: wie geht man in Programmiersprachen mit Zahlenwerten um? Hier lohnt sich zuerst ein Blick in die Mathematik, wo sehr wohl zwischen verschiedenen Zahlenarten (in der Mathematik heißen Zahlenmengen mit den darauf definierten Operationen Körper, Gruppen, Ringe,...) unterschieden wird. Die Menge der Natürlichen Zahlen  $\mathbb{N}$ , Ganzen Zahlen  $\mathbb{Z}$ , Rationalen Zahlen  $\mathbb{Q}$ , Reellen Zahlen  $\mathbb{R}$ , Komplexen Zahlen  $\mathbb{C}, \dots$  Jede dieser Zahlenarten hat bestimmte Eigenschaften und ist für bestimmte Sachen sehr gut geeignet, für andere eher weniger, weil bestimmte Sachen einerseits darin nicht ausgedrückt werden können oder andererseits zu komplex für einfache Sachverhalte sind. Oder haben Sie schon einmal gesehen, dass ein Verkäufer / eine Verkäuferin in einem Elektronikladen komplexe Zahlen bei der Rechnungserstellung nutzt? So ist dann beispielsweise das Kopfrechnen mit natürlichen Zahlen wesentlich einfacher als das Kopfrechnen mit reellen Zahlen. Dafür ist es aber andererseits in den Zahlenmengen  $\mathbb{N}, \mathbb{Z}$  und  $\mathbb{Q}$  überhaupt nicht möglich, den Wert  $\sqrt{2}$  korrekt darzustellen.

Bei Programmiersprachen ist die Problemstellung zuerst einmal ähnlich: was braucht man als Grundbausteine und was kann man gegebenfalls damit zusätzlich aufbauen? Beispielsweise lassen sich rationale Zahlen der Form  $x/y \in \mathbb{Q}$  als Paare von ganzen Zahlen  $x, y \in \mathbb{Z}$  (Zähler, Nenner) darstellen. Wenn man also ganze Zahlen darstellen kann und zusätzlich die Möglichkeit hätte Paare von Zahlen zu behandeln, so bräuchte man keinen "eingebauten" Datentyp für Rationale Zahlen. In Programmiersprachen kommt aber über diese Fragestellungen hinaus die zusätzliche zu lösende Problematik der Darstellung von möglichen Zahlenwerte durch eine Folge von Bits. Wie kann man die Zahl 2 durch eine bestimmte Bitkombination darstellen, wie die Zahlen  $-3, 1/2, \sqrt{2}, \pi, \dots$

Bereits in der Rechnerhardware wird eindeutig und sehr folgeschwer zwischen ganzen Zahlen und (einer Art von) reellen Zahlen unterschieden und diese Unterscheidung schlägt sich auch auf Programmiersprachen komplett durch. In fast allen Programmiersprachen wird unterschieden zwischen **ganzen Zahlen** (englisch: *integer*) und sogenannten Fließkommazahlen (englisch: *floating point*). Ganzzahlen können nur ganzzahlige Werte annehmen, und dies auch nur als Teilmenge aus  $\mathbb{Z}$ , und die Arithmetik mit Ganzzahlen ist (bis auf mögliche Bereichsüberschreitungen; siehe unten) exakt. Die Erwähnung, dass Berechnungen mit ganzzahligen Werten exakt ausgeführt werden, mag auf den ersten Blick verwundern, ist aber nicht selbstverständlich, wie die nachfolgende Kategorie zeigt. **Fließkommazahlen** sind eine Teilmenge der reellen Zahlen und die Arithmetik mit diesen Zahlen muss *nicht* exakt sein! Diese beiden Kategorien werden im Nachfolgenden im Detail betrachtet und ihre Darstellung in einem Rechner hergeleitet, die Operationen auf den Werten vorgestellt, aber auch die Beschränkungen diskutiert, die etwa im Fall von Fließkommazahlen sehr gravierend sein können und denen sich *jeder* Programmierer jederzeit bewusst sein muss.

### 4.3.1 Positive ganze Zahlen

Die Mengen der natürlichen Zahlen  $\mathbb{N}$  und ganzen Zahlen  $\mathbb{Z}$  sind unendlich groß und alle Werte lassen sich daher nicht mit einer endlichen Zahl von Binärziffern kodieren. Ein möglicher Ansatz zur Kodierung von natürlichen Zahlen wäre es, zur Kodierung eines Wertes soviele Bits zu nehmen, wie für diesen Wert notwendig sind. Für kleine Zahlen könnte man wenige Bits nehmen, für größere Zahlen entsprechend mehr Bits.

#### Beispiel 4.6:

Eine mögliche Kodierung nach diesem Schema wäre beispielsweise, allen Zahlen aufsteigend jeweils die nächste noch nicht vergebene Bitkombination zu vergeben. Hat man alle Kombination für  $n$  Bits vergeben, so nimmt man ein zusätzliches Bit dazu und vergibt nacheinander alle Kombination mit  $n+1$  Bits.

Wert	Kodierung
0	0
1	1
2	10
3	11
4	100
5	101
...	...



Dieser Ansatz hat den Nachteil, dass zur Darstellung von solchen Werten eine variable Anzahl von Bits notwendig ist, was aus mehreren Gründen gravierende Nachteile hätte (beschränkte Ressourcen in einem Prozessor, Dauer einer Operation hängt vom Wert und damit Anzahl der Bits ab, Länge der Bitfolge müsste gespeichert werden).

Ein anderer Ansatz ist, immer eine feste Anzahl  $n$  von Bits zu nehmen. Oben beispielhaft aufgeführte Nachteile wären damit gelöst, allerdings sind mit einer festen Anzahl  $n$  von Bits nur  $2^n$  verschiedene Werte darstellbar. Man wäre also bei einer Festlegung auf  $n$  Bits auf maximal  $2^n$  verschiedene Werte festgelegt. Sinnvollerweise kodiert man bei diesem Ansatz ein zusammenhängendes Intervall  $[p, \dots, q]$  aus  $\mathbb{N}$  (aus  $\mathbb{Z}$  wäre prinzipiell auch möglich) mit  $q - (p - 1) = 2^n$ . Jedem dieser Werte  $p$  bis  $q$  entspricht dann eine feste Bit-Kombination dieser  $n$  Bit.

#### Beispiel 4.7:

Mögliche Kodierungen mit 2 Bits für das Intervall  $[0, 3]$  beziehungsweise  $[123, 126]$  sind:

Wert	Kodierung	Wert	Kodierung
0	00	123	00
1	01	124	01
2	10	125	10
3	11	126	11



Normiert man das Intervall so, dass immer bei Null begonnen wird und nur positive Werte kodiert werden, so erhält man bei Nutzung von  $n$  Bits einen Wertebereich von  $[0, \dots, 2^n - 1]$ , entsprechend der Anzahl von

Möglichkeiten an Kombinationen der  $n$  Bit.

**Beispiel 4.8:**

Nachfolgende sind für einige ausgewählte Werte  $n$  die Intervalle aufgeführt, die mit dieser Kodierung überdeckt werden.

$n$	Intervall $[0, 2^n - 1]$
8	$[0, 255]$
16	$[0, 65.535]$
32	$[0, 4.294.967.295]$
64	$[0, \approx 1,8 \cdot 10^{19}]$
128	$[0, \approx 3,4 \cdot 10^{38}]$

Um diese Intervalle einordnen zu können, hier einige Zahlen aus verschiedenen Bereichen. Die Anzahl Studierender in einem Kurs ist (je nach Kurs) mit 8 Bits darstellbar. Alle Entfernungswerte zwischen zwei Orten auf der Erde (Äquatorumfang ca. 40.000 km) sind nicht mehr mit 8 Bit darstellbar, aber mit 16 Bit. Zur Darstellung der Einwohnerzahl Deutschlands (ca 80 Millionen) reichen 16 Bits nicht aus, man bräuchte die 32-Bit-Kategorie, die aber wiederum nicht zur Darstellung der Einwohnerzahl der Erde (circa 7-6 Milliarden) ausreicht. Der Schuldenstand der Bundesrepublik Deutschland beträgt circa 1.700.000.000.000 Euro, zur Darstellung bräuchte man also mindestens die Kategorie 64 Bits. Die Anzahl der Moleküle in einem Liter Wasser bei 4 Grad Celsius beträgt  $\approx 33 \times 10^{24}$  (Kategorie 128 Bit erforderlich). Die Wasservorräte der Erde werden auf circa 1,4 Milliarden Kubikkilometer geschätzt, also circa  $1,4 \cdot 10^{21}$  Liter (128 Bit erforderlich). Wollte man nach diesem Ansatz alle Wassermoleküle in den Weltmeeren zählen wären auch 128 Bit nicht ausreichend. Was ist mit der Anzahl aller Moleküle (Wasser, Salz, Erde,...) auf der Erde, aller Moleküle unseres Sonnensystems, der Milchstraße, des Weltalls,...? Jede Festlegung auf ein  $n$  legt zwangsläufig die Grenze der darstellbaren Werte fest. ♦

Nachdem also die Festlegung auf eine feste Anzahl von Bits vorgenommen wurde (wieviel das sind, wird noch offen gelassen), ist die Frage, wie einzelne Werte aus den Natürlichen Zahlen mit den zur Verfügung stehenden Bits kodiert werden können. Dazu soll zunächst das gewohnte Dezimalsystem zur Basis 10 angeschaut werden. In diesem Zahlensystem hat etwa die Zahl 1234 die Bedeutung:  $1234 = 1 \cdot 1000 + 2 \cdot 100 + 3 \cdot 10 + 4 = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$ . Ein solches Zahlensystem, in dem einer Ziffer entsprechend ihrer Position in der Kombination mit anderen Ziffern eine Wertigkeit zugeordnet ist, nennt man **Stellenwertsystem**. Das bekannte Dezimalsystem ist ein solches Stellenwertsystem mit den Ziffern 0, ..., 9 und der Basis 10. Allgemein in einem Stellenwertsystem kann man zu einer gegebenen Basis  $B$  und Ziffern  $0, \dots, B-1$  den Wert einer Zahl  $Z = (z_{n-1} \dots z_1 z_0)_B$  mit  $z_i \in \{0, \dots, B-1\}$  dann angeben durch:

$$\begin{aligned} \text{Wert}(z_{n-1} \dots z_0)_B &= z_{n-1} \cdot B^{n-1} + \dots + z_1 \cdot B^1 + z_0 \cdot B^0 \\ &= \sum_{i=0}^{n-1} z_i \cdot B^i \end{aligned}$$

Ist die Basis einer Zahlendarstellung nicht klar, so schreibt man diese auch als Suffix hinter die Ziffernfolge. Die Zahl 5 kann im Stellenwertsystem zur Basis 10 als  $5_{10}$  notiert werden, die gleiche Zahl im Stellenwertsystem zur Basis 2 wäre dann  $101_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$ .

**Beispiel 4.9:**

$$1234_{10} = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$$

also  $B = 10$ ,  $z_0 = 4$ ,  $z_1 = 3$ ,  $z_2 = 2$  und  $z_3 = 1$ .



Zum Verständnis sehr wichtig ist die Unterscheidung des mathematischen Wertes 1234 von der Darstellung dieses Wertes im Zehnersystem durch die Kombination der Ziffern 1, 2, 3 und 4 in dieser Reihenfolge. Dies ist nichts anderes als die Unterscheidung zwischen Syntax (Folge von Ziffern) und Semantik (Wert)!

Das heute gebräuchliche Stellenwertsystem zur Basis (10 Finger!) kommt ursprünglich aus Indien. Die Basis 10 ist relativ willkürlich, man könnte auch andere Basen nutzen. So nutzten die Babylonier etwa die Basis 60 (noch heute in unserer Zeitrechnung mit 60 Sekunden/Minuten vorhanden) und die Maya etwa die Basis 20.

Als Hinweis sei hier angegeben, dass sich ein Polynom der Form  $z_{n-1} \cdot B^{n-1} + \dots + z_1 \cdot B^1 + z_0 \cdot B^0 = \sum_{i=0}^{n-1} z_i \cdot B^i$  einfach und effizient mit Hilfe des **Horner-Schemas** auswerten lässt. Das Hornerschema ist auf der rechten Seite der folgenden Gleichung angegeben:

$$z_{n-1} \cdot B^{n-1} + \dots + z_1 \cdot B^1 + z_0 \cdot B^0 = (\dots (z_{n-1} \cdot B + z_{n-2}) \cdot B) \dots + z_1 \cdot B + z_0$$

Der Vorteil des Hornerschemas ist unter anderem, dass die Potenzen – eine relativ aufwändige Operationen auf einem Rechner – auf einfache Multiplikationen zurückgeführt werden können.

Ähnlich wie im Dezimalsystem lässt sich also auch im Dualsystem, das heißt in einem Stellenwertsystem zur Basis 2, einer Binärziffernfolge  $b_{n-1} \dots b_1 b_0$  mit  $b_i \in \{0, 1\}$  ein Wert zuordnen, nämlich die obige allgemeine Formel mit dem Spezialfall  $B = 2$ :

$$b_{n-1} \cdot 2^{n-1} + \dots + b_1 \cdot 2^1 + b_0 \cdot 2^0 = \sum_{i=0}^{n-1} b_i \cdot 2^i$$

**Beispiel 4.10:**

Die mit 4 Bit darstellbaren positiven Zahlen des Dualsystems sind:

Bit-Darstellung	Wert	Bit-Darstellung	Wert
0000	0	1000	8
0001	1	1001	9
0010	2	1010	10
0011	3	1011	11
0100	4	1100	12
0101	5	1101	13
0110	6	1110	14
0111	7	1111	15

Es gilt etwa:  $1011_2 = 11_{10}$ , weil  $1011_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8 + 0 + 2 + 1 = 11$ .



Will man eine Dualzahl in eine Dezimalzahl umwandeln, so muss man entsprechend der obigen Formel alle Dualziffern  $b_i$  mit der Stelligkeit  $2^i$  multiplizieren und diese Produkte aufaddieren.

**Beispiel 4.11:**

Der Wert der Dualzahl  $01010100_2$  ist also:

$$\begin{aligned}
 01010100_2 &= 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 \\
 &= 0 \cdot 128 + 1 \cdot 64 + 0 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 0 \cdot 1 \\
 &= 64 + 16 + 4 \\
 &= 84_{10}
 \end{aligned}$$

Will man umgekehrt eine Dezimalzahl in eine Dualzahl umwandeln, so muss man die Ziffern zur entsprechenden Stelligkeit bestimmen. Dazu soll als Motivation folgendes Beispiel dienen, in dem diese Bestimmung im bekannten Dezimalsystem durchgeführt wird.

**Beispiel 4.12:**

Im Dezimalsystem geschieht die Bestimmung der Stellenziffern der Zahl  $1234_{10}$ , indem man die Zahl sukzessive durch eine der Stelligkeit entsprechende Zehnerpotenz dividiert und den Rest weiter betrachtet. Also:

$$\begin{aligned}
 1234/10^3 &= 1 \text{ Rest } 234 \\
 234/10^2 &= 2 \text{ Rest } 34 \\
 34/10^1 &= 3 \text{ Rest } 4 \\
 4/10^0 &= 4
 \end{aligned}$$

Das heißt also, die Zahl 1234 besitzt (trivialerweise) im Dezimalsystem die Ziffernfolge 1, 2, 3, gefolgt von 4. Um die Zahl  $1234_{10}$  in eine Ziffernfolge des Dualsystems umzuwandeln, geht man ganz analog mit Zweier- statt Zehnerpotenzen vor und bestimmt jeweils eine Ziffer als den ganzzahligen Teil einer Division mit der entsprechenden (Zweier-)Potenz:

$$\begin{aligned}
 1234/2^{10} &= 1234/1024 = 1 \text{ Rest } 210 \\
 210/2^9 &= 210/512 = 0 \text{ Rest } 210 \\
 210/2^8 &= 210/256 = 0 \text{ Rest } 210 \\
 210/2^7 &= 210/128 = 1 \text{ Rest } 82 \\
 82/2^6 &= 82/64 = 1 \text{ Rest } 18 \\
 18/2^5 &= 18/32 = 0 \text{ Rest } 18 \\
 18/2^4 &= 18/16 = 1 \text{ Rest } 2 \\
 2/2^3 &= 2/8 = 0 \text{ Rest } 2 \\
 2/2^2 &= 2/4 = 0 \text{ Rest } 2 \\
 2/2^1 &= 2/2 = 1 \text{ Rest } 0 \\
 0/2^0 &= 0/1 = 0 \text{ Rest } 0
 \end{aligned}$$

Die Zahl  $1234_{10}$  besitzt also im Dualsystem eine Darstellung der Form  $10011010010_2$ , in der obigen Rechnung sind dies die Ziffern des Divisionsresultates von oben nach unten gelesen (höchste Stelligkeit steht vorne). ♦

Wie oben gesehen, lassen sich mit 4 Bit  $2^4 = 16$  unterschiedliche Werte darstellen. Wie man weiterhin leicht einsieht, reichen 16 unterschiedliche Werte für die meisten Rechneranwendungen nicht aus (Ihr Kontostand hatte vermutlich schon mehr als 16 verschiedene Zustände). Dementsprechend müssen auch mehr Bit zur Darstellung von Daten genommen werden. Üblich sind heute 32 oder 64 Bit, mit denen sich  $2^{32} = 4.294.967.296$  beziehungsweise  $2^{64} = 18.446.744.073.709.551.616$  verschiedene Werte darstellen lassen.

Dualdarstellung	Oktaldarstellung	Hexadezimaldarstellung	Dezimaldarstellung
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	a	10
1011	13	b	11
1100	14	c	12
1101	15	d	13
1110	16	e	14
1111	17	f	15

Table 4.2: Darstellung der Dezimalzahlen 0-15 in den verschiedenen Zahlensystemen

Als Bitfolgen sind 32 oder gar 64 Bit für einen menschlichen Betrachter nicht sehr übersichtlich. Bei der Betrachtung der Binärzahl  $0010011101011000000000111000110_2$  kann man leicht den Überblick verlieren. Aus diesem Grunde sind (neben der üblichen Dezimalnotation) zwei alternative Notationen üblich: die Oktal- und Hexadezimaldarstellung. Oktalzahlen sind Zahlendarstellungen zur Basis 8 mit den Ziffern  $0, \dots, 7$ , Hexadezimalzahlen sind Zahlendarstellungen zur Basis 16. Bei der Hexadezimaldarstellung reichen die gewohnten zehn Ziffern allerdings nicht mehr aus, man nimmt deshalb noch die Buchstaben A-F oder a-f als "Ziffern" mit der Wertigkeit 10-15 hinzu.

#### Beispiel 4.13:

Die Zahl  $17_{10}$  wird durch die Oktalzahl  $21_8 = 2 \cdot 8^1 + 1 \cdot 8^0$  dargestellt. In Hexadezimaldarstellung ist dies  $17_{10} = 11_{16} = 1 \cdot 16^1 + 1 \cdot 16^0$ .

Die Zahl  $27_{10}$  lautet in Oktaldarstellung  $33_8 = 3 \cdot 8^1 + 3 \cdot 8^0$  und in Hexadezimaldarstellung  $1B_{16} = 1 \cdot 16^1 + 11 \cdot 16^0$  (in der Hexadezimaldarstellung hat die Ziffer 11 die Darstellung B).  $\blacklozenge$

Eine Umwandlung zwischen Oktal-/Hexadezimaldarstellung und Dualdarstellung ist sehr einfach, da die Basen jeweils Zweierpotenzen sind: 3 beziehungsweise 4 Binärziffern werden jeweils in einer Oktal- beziehungsweise Hexadezimalziffer kodiert. Tabelle 4.2 gibt eine Tabelle der Darstellung der Dezimalzahlen 0-15 in den unterschiedlichen Zahlensystemen an.

Zur Umwandlung der Binärzahl  $0010011101011000000000111000110_2$  in eine Oktalzahl teilt man sinnvollerweise die Binärzahl beginnend bei der niedrigwertigsten Ziffer in Päckchen der Größe 3 ein, weil  $8 = 2^3$ , also 3 Binärziffern für eine Oktalziffer stehen. Das Resultat nach der Päckchenbildung ist dann  $00.100.111.010.110.000.000.111.000.110_2$ . Die einzelnen Päckchen lassen sich jetzt als Oktalziffern notieren, so dass die obige Zahl als Oktalzahl  $04726000706_8$  ist. Beispielsweise ist der Wert des letzten Päckchen  $110_2$  gleich  $1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 6_{10} = 6_8$ . Man kann das Ergebnis auch leicht überprüfen, indem man die Binärzahl in eine Dezimalzahl umwandeln (Polynom "ausrechnen") und dann anschließend diese

Dezimalzahl nach dem weiter unten beschriebenen Divisionsverfahren in eine Oktalzahl zerlegen. Welcher Weg war einfacher / schneller?

Zur Umwandlung der Binärzahl  $00100111010110000000000111000110_2$  in eine Hexadezimalzahl teilt man die Binärzahl in Päckchen der Grösse 4 ein ( $16 = 2^4$ ):  $0010.0111.0101.1000.0000.0001.1100.0110_2$ . Die einzelnen Päckchen lassen sich jetzt als Hexadezimalziffern notieren, so dass die obige Zahl als Hexadezimalzahl  $275801c6_{16}$  ist. Beispielsweise ist der Wert des zweitletzten Päckchens  $1100_2$  gleich  $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 10_{10} = c_{16}$ , also die Ziffern  $c$  im Hexadezimalsystem.

Um eine Zahl  $1234_{10}$  in eine Ziffernfolge des Oktalsystems umzuwandeln, geht man analog zur Umwandlung für das Dualsystem vor, indem man sukzessive den Restwert durch eine Potenz dividiert, damit eine Ziffer zur Basis erhält, und mit dem Rest der Division den Algorithmus weiterführt, bis der Restwert 0 ist.

$$\begin{aligned} 1234/8^3 &= 1234/512 = 2 \text{ Rest } 210 \\ 210/8^2 &= 210/64 = 3 \text{ Rest } 18 \\ 18/8^1 &= 18/8 = 2 \text{ Rest } 2 \\ 2/8^0 &= 2/1 = 2 \text{ Rest } 0 \end{aligned}$$

Also ist die Zahl  $1234_{10}$  im Oktalsystem darstellbar als  $2322_8$ .

Analog geschieht die Umwandlung der Zahl  $1234_{10}$  in eine Ziffernfolge des Hexadezimalsystems:

$$\begin{aligned} 1234/16^2 &= 1234/256 = 4 \text{ Rest } 210 \\ 210/16^1 &= 210/16 = d (= 13_{10}) \text{ Rest } 2 \\ 2/16^0 &= 2/1 = 2 \text{ Rest } 0 \end{aligned}$$

Also ist die Zahl  $1234_{10}$  im Hexadezimalsystem darstellbar als  $4d2_{16}$ .

Zur Umwandlung der Hexadezimalzahl in eine Zahl im gewohnten Dezimalsystem, also zur Basis 10, muss man wie gewohnt alle Ziffern mit der Stelligkeit der Position multiplizieren und das Ergebnis aufaddieren. Zum Beispiel lässt sich die Hexadezimaldarstellung  $275801c6$  "umrechnen" zu:

$$\begin{aligned} 275801c6_{16} &= 2 \cdot 16^7 + 7 \cdot 16^6 + 5 \cdot 16^5 + 8 \cdot 16^4 + 0 \cdot 16^3 + 1 \cdot 16^2 + c \cdot 16^1 + 6 \cdot 16^0 \\ &= 2 \cdot 268435456 + 7 \cdot 16777216 + 5 \cdot 1048576 + 8 \cdot 65536 + 0 \cdot 4096 \\ &\quad + 1 \cdot 256 + 12 \cdot 16 + 6 \cdot 1 \\ &= 536870912 + 117440512 + 5242880 + 524288 + 0 + 256 + 192 + 6 \\ &= 660078848_{10} \end{aligned}$$

Der Vorteil der Oktal- und Hexadezimaldarstellung ist die einfache Umwandlung von/nach der Binärdarstellung. Will man eine Zahl zwischen Dezimalsystem und Dualsystem umwandeln, so ist dies nicht mehr einfach durch Aufteilung in Päckchen und Umwandlung der Päckchen in eine entsprechende Ziffer des anderen Zahlensystems möglich (s.o. die Umwandlung von  $275801c6_{16}$  in  $391774662_{10}$ ).

Mit Binärzahlen, Oktal- und Hexadezimalzahlen lassen sich die bekannten Grundrechenarten ausführen. Hierzu kann man einfach die Algorithmen aus der Schulmathematik nehmen.

#### **Beispiel 4.14:**

Im Dezimalsystem geschieht die Addition der beiden Zahlen  $1001_{10}$  und  $11_{10}$  auf folgende Weise:

$$\begin{array}{r} 1 \ 0 \ 0 \ 1 \\ + \ 0 \ 0 \ 1 \ 1 \\ \hline 0 \ 0 \ 0 \ 0 \quad (\text{Übertrag im Dezimalsystem}) \\ = \ 1 \ 0 \ 1 \ 2 \end{array}$$

Im Dualsystem geschieht die Addition mit den Dualzahlen  $1001_2$  und  $11_1$  analog:

$$\begin{array}{r}
 1 & 0 & 0 & 1 & = 9_{10} \\
 + & 0 & 0 & 1 & 1 & = 3_{10} \\
 \hline
 0 & 1 & 1 & 0 & (\text{Übertrag im Dualsystem}) \\
 = & 1 & 1 & 0 & 0 & = 12_{10}
 \end{array}$$

Die Subtraktion den beiden Zahlen  $1100_{10}$  und  $110_{10}$  im Dezimalsystem geschieht wie folgt:

$$\begin{array}{r}
 1 & 1 & 0 & 0 \\
 - & 0 & 1 & 1 & 0 \\
 \hline
 1 & 1 & 0 & 0 & ("Borgen" im Dezimalsystem) \\
 = & 0 & 9 & 9 & 0
 \end{array}$$

Im Dualsystem geschieht dies für die Zahlen  $1100_2$  und  $110_1$  analog:

$$\begin{array}{r}
 1 & 1 & 0 & 0 & = 12_{10} \\
 - & 0 & 1 & 1 & 0 & = 6_{10} \\
 \hline
 1 & 1 & 0 & 0 & ("Borgen" im Dualsystem) \\
 = & 0 & 1 & 1 & 0 & = 6_{10}
 \end{array}$$

Für die Multiplikation der beiden Zahlen  $32_{10}$  und  $41_{10}$  im Dezimalsystem gilt:

$$\begin{array}{r}
 3 & 2 & \cdot & 4 & 1 \\
 \hline
 1 & 2 & 8 \\
 3 & 2 \\
 \hline
 = & 1 & 3 & 1 & 2
 \end{array}$$

Für die Multiplikation der beiden Zahlen  $6_{10} = 110_2$  und  $5_{10} = 101_2$  geschieht dies wiederum analog:

$$\begin{array}{r}
 1 & 1 & 0 & \cdot & 1 & 0 & 1 \\
 \hline
 1 & 1 & 0 \\
 0 & 0 & 0 \\
 1 & 1 & 0 \\
 \hline
 = & 1 & 1 & 1 & 1 & 0 & = 30_{10}
 \end{array}$$

Die Division lässt sich etwa durch fortgesetzte Subtraktion erzielen.



### 4.3.2 Positive und negative ganze Zahlen

Bis jetzt wurden nur positive Zahlen betrachtet. Nun stellt sich die Frage, wie man auch negative Zahlen darstellen kann und wie sich die Operationen auch auf negative Zahlen in der gewählten Darstellung ausweiten lassen. Wünschenswert wäre es zudem, wenn man eine einheitliche Darstellungsform für positive *und* negative Werte hätte. Aus der Anzahl  $2^n$  möglicher Werte für  $n$  Bits ergibt sich, dass bei der Hinzunahmen von negativen Werten der positive Wertebereich eingeschränkt werden muss (wenn man hinten etwas hinzufügen will, muss man vorne etwas wegnehmen). Es gibt zwei naheliegende Möglichkeiten negative Werte einzuführen: Vorzeichendarstellung und Komplementdarstellung.

In der **Vorzeichendarstellung** wird ein Bit dafür reserviert, das Vorzeichen der Zahl anzugeben. Die restlichen  $n - 1$  Bit stellen dann den Absolutwert der Binärzahl dar.

**Beispiel 4.15:**

Als Beispiel sei angenommen, dass man 4 Bit zur Verfügung hat und das höchstwertige / oberste Bit  $V$  das Vorzeichen kodiert (**Vorzeichenbit**). Ist das oberste Bit 0, so ist die Zahl insgesamt positiv. Ist das oberste Bit 1, so ist die Zahl insgesamt negativ. Die restlichen 3 Bit kodieren im positiven wie negativen Fall wie gewohnt den Zahlenwert.

$V$	$2^2$	$2^1$	$2^0$	Wert
0	0	0	1	$= +1 = +0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
1	0	0	1	$= -1 = -0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
0	1	0	1	$= +5 = +1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
1	1	0	1	$= -5 = -1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$



Diese Vorzeichendarstellung hat aber zwei Nachteile:

1. Die Zahl 0 hat zwei Darstellungen: +0 und -0. Dies ist unangenehm, da man beim Vergleich eines Wertes mit Null jeweils beide Versionen der Null berücksichtigen müsste.
2. Die Addition einer positiven Zahl  $x_1$  mit einer negativen Zahl  $x_2$  ist nicht mehr gleich der Subtraktion des Absolutwertes der zweiten Zahl  $x_1 - |x_2|$ . Dies hätte zur Folge, dass in einem Prozessor sowohl eine Additionseinheit als auch eine Subtraktionseinheit vorhanden sein müsste (Chipfläche ist kostbar!).

Zur rechnerinternen Darstellung sowohl positiver wie negativer Ganzzahlen (*Integer*) wird deshalb in heutigen Rechnern ausschließlich die **Zweierkomplementdarstellung** verwandt. Dabei wird eine Ganzzahl in  $n$  Bit  $b_{n-1} \dots b_0$  abgespeichert, wobei die Bits wie gewohnt jeweils den Wert 0 oder 1 annehmen können. Der Wert solch einer Bitfolge  $b_{n-1} \dots b_0$  mit  $n$  Bit in der Zweierkomplementkodierung ist aber gegenüber dem alten Ansatz zur Kodierung ausschließlich positiver Zahlen leicht verändert und entspricht

$$\text{Wert}(b_{n-1} \dots b_0) = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

Im Gegensatz zur reinen Vorzeichendarstellung von oben ist die Bedeutung des höchstwertigen Bits verschieden, nämlich nicht nur die Codierung, ob positiv oder negativ. Das höchstwertige Bit  $b_{n-1}$  kann aber trotzdem wie in der Vorzeichendarstellung als **Vorzeichenbit** interpretiert werden: Ist dieses Bit 0, so ist der Gesamtwert insgesamt positiv, ist dieses Bit 1, so ist der Gesamtwert insgesamt negativ. *Zusätzlich* geht bei der Zweierkomplementdarstellung aber auch der Wert dieser Stelligkeit in den Gesamtwert mit ein.

Um die Darstellung von Zahlen in Zweierkomplementdarstellung weiter herzuleiten, wird zuerst definiert, was das Einkomplement einer Ganzzahl ist.

**Definition 4.4 (Einkomplement, Zweierkomplement):**

Für eine Ganzzahl in Dualdarstellung  $b_{n-1} \dots b_0$  wird das **Einkomplement** dadurch gebildet, dass stellenweise jede Dualziffer durch ihr Komplement ersetzt wird. Das Komplement von 0 ist 1, das Komplement von 1 ist 0. Das **Zweierkomplement** einer Ganzzahl in Dualdarstellung mit  $n$  Bit ist definiert durch die Bildung des Einkomplements dieser Dualzahl und der anschließenden Addition des Wertes 1. Ein etwaiger Überlauf der resultierenden Zahl wird dabei ignoriert (es bleibt bei den  $n$  Bit). ◆

**Beispiel 4.16:**

Das Einkomplement von 00101 ist 11010, das Einkomplement von 10010 ist 01101.

Bit-Darstellung	Wert	Bit-Darstellung	Wert
0000	0	1000	-8
0001	1	1001	-7
0010	2	1010	-6
0011	3	1011	-5
0100	4	1100	-4
0101	5	1101	-3
0110	6	1110	-2
0111	7	1111	-1

Table 4.3: Mögliche Zweierkomplementzahlendarstellungen mit 4 Bit

Zur Bestimmung des Zweierkomplements von  $00101_2$  bestimmt man zuerst das Einerkomplement,  $11010_2$ , und addiert eine 1 hinzu, was insgesamt  $11011_2$  ergibt.

Das Zweierkomplement von  $10010_2$  ist  $01110_2$ , weil das Einskomplement  $01101_2$  ist und  $01101_2 + 00001_2 = 01110_2$ .  $\diamond$

**Anmerkung:** Addiert man zu einer beliebigen Dualzahl ihr Einerkomplement, so enthält die resultierende Dualzahl nur Einsen.

$$\begin{array}{r}
 & 0 & 0 & 1 & 0 & 1 & \text{Zahl} \\
 + & 1 & 1 & 0 & 1 & 0 & \text{Einskomplement} \\
 \hline
 = & 1 & 1 & 1 & 1 & 1 & \text{Resultat}
 \end{array}$$

Addiert man zu einer beliebigen Dualzahl ihr Zweierkomplement, so erhält man eine Zahl mit Nullen und einer Übertragsziffer mit Wert 1, das heißt das Zweierkomplement ist die Ergänzung zur nächsten Zweierpotenz.

$$\begin{array}{r}
 & 0 & 0 & 1 & 0 & 1 & \text{Zahl} \\
 + & 1 & 1 & 0 & 1 & 1 & \text{Zweierkomplement} \\
 \hline
 = & 1 & 0 & 0 & 0 & 0 & \text{Resultat}
 \end{array}$$

Ein weiterer Zusammenhang ergibt sich im Zusammenhang mit der Zweierkomplementbildung. Bildet man zu einer beliebigen Zahl  $x = b_{n-1} \dots b_1 b_0$  das Zweierkomplement, so ist der Wert der resultierenden Zahl gleich  $-x$ . In anderen Worten: mit der Operation einer Zweierkomplementbildung kann ein Vorzeichenwechsel durchgeführt werden. Die möglichen Zahlendarstellungen in Zweierkomplementdarstellung mit 4 Bit sind in Tabelle 4.3 angegeben.:

Generell lässt sich mit  $n$  Bit und der Zweierkomplementdarstellung ein Zahlenbereich von  $[-2^{n-1}, \dots, -1, 0, 1, \dots, 2^{n-1} - 1]$  abdecken. Der Vorteil der Zweierkomplementdarstellung ist die Tatsache, dass sich alle Zahlen des entsprechenden Zahlenintervalls *eindeutig* darstellen lassen (inklusive der Null) und eine Unterscheidung zwischen positiven und negativen Zahlen bei allen Grundoperationen wie Addition, Subtraktion etc. nicht notwendig ist. Weiterhin lässt sich die Subtraktion zweier Zahlen auf die Addition zurückführen.

Java-Typ	Speicherbedarf	Wertebereich
<code>byte</code>	1 Byte, 8 Bit	$-2^7$ bis $2^7 - 1 = -128$ bis $127$
<code>short</code>	2 Byte, 16 Bit	$-2^{15}$ bis $2^{15} - 1 = -32.768$ bis $32.767$
<code>int</code>	4 Byte, 32 Bit	$-2^{31}$ bis $2^{31} - 1 = -2.147.483.648$ bis $2.147.483.647$
<code>long</code>	8 Byte, 64 Bit	$-2^{63}$ bis $2^{63} - 1 = -9.223.372.036.854.755.808$ bis $9.223.372.036.854.775.807$

Table 4.4: Ganzzahltypen in Java

**Beispiel 4.17:**

Entsprechend lässt sich die Operation  $4 - 3$  auf  $4 + (-3)$  zurückführen, weil das Zweierkomplement von  $3_{10} = 0011_2$  der Darstellung  $1100_2 + 1 = 1101_2$  entspricht (siehe obiges Beispiel). Die Bildung eines Zweierkomplements ist wie gezeigt eine sehr einfache Operation.

$$\begin{array}{r}
 & 1 & 1 & 0 & 1 & = -3_{10} \\
 + & 0 & 1 & 0 & 0 & = 4_{10} \\
 \hline
 & 1 & 1 & & & \text{Übertrag} \\
 = & 0 & 0 & 0 & 1 & = 1_{10}
 \end{array}$$



### 4.3.3 Ganze Zahlen in Java

Wie eben kennen gelernt, lassen sich mit  $n$  Bit  $2^n$  verschiedene Werte darstellen. Mit der Zweierkomplementdarstellung ergibt sich gleichzeitig daraus auch ein festes Intervall  $[-2^{n-1}, +2^{n-1} - 1]$  darstellbarer Werte. In Kapitel 4.3.1 wurden beispielhaft einige Zahlenwerte aus dem alltäglichen Leben angegeben (Anzahl Studierende, Einwohner Deutschlands, Moleküle,...), wo klar wurde, dass es schwierig ist, ein festes  $n$  und damit Zahlenintervall anzugeben, in das alle Werte reinpassen, mit denen man es jemals zu tun bekommen könnte. Man könnte jetzt geneigt sein zu sagen, man nimmt einfach ein sehr großes  $n$ , beispielsweise  $n = 1024$  ( $2^{1024}$  ist eine ziemlich große Zahl!). Das würde andererseits unter der Annahme eines festen  $n$  für *alle* Zahlen aber auch bedeuten, dass zur Darstellung *jeder* Zahl 1024 Bits benötigt würden, was in den meisten Fällen eine sehr große Platzverschwendug wäre. Es wäre also nicht uninteressant einem Programmierer mehrere Möglichkeiten (kleines  $n$  / kleiner Zahlenbereich / wenig Speicherbedarf bis großes  $n$  / großer Zahlenbereich / großer Speicherbedarf) anzubieten, aus denen er ja nach Bedarf wählen kann.

In Java gibt es deshalb verschiedene Typen von Ganzzahlen in Zweierkomplementdarstellung, die sich durch ihren Speicherbedarf (wie viele Bit zur Verfügung stehen) und entsprechend damit ihrem Wertebereich unterscheiden. Während es in Java nur vorzeichenbehaftete Ganzzahlentypen gibt (das darstellbare Intervall ist immer  $[-2^{n-1}, +2^n - 1]$ ), kennen andere Programmiersprachen wie etwa C oder C++ die explizite Unterscheidung zwischen vorzeichenbehafteten Ganzzahlentypen (`signed int`) und vorzeichenlosen Ganzzahlentypen (`unsigned int`), also einem ausschließlich positivem Zahlenbereich  $[0, 2^n - 1]$ . Die Anzahl der insgesamt darstellbaren Zahlen ist immer natürlich auf  $2^n$  beschränkt, es stellt sich in diesen Sprachen also lediglich die Frage, ob positive und negative Zahlen im Intervall  $[-2^{n-1}, 2^{n-1} - 1]$  dargestellt werden sollen oder ausschließlich positive Zahlen im Intervall  $[0, 2^n - 1]$ .

Welchen dieser in Java zur Verfügung stehenden Datentypen man zur Aufnahme von Ganzzahlwerten zum Beispiel in einer Variablenklärung auswählt, muss man anhand von verschiedenen Kriterien beurteilen:

- Alle vorkommenden Werte müssen in diesem Ganzzahltyp darstellbar sein.
- Die Ergebnisse von Operationen, die auf diesen Werten ausgeführt werden, müssen in diesem Ganzzahltyp darstellbar sein oder müssen in den größeren Typ umgewandelt werden.

- Der Datentyp `long` benötigt doppelt soviel Speicherplatz wie der Datentyp `int`. Vergeudet man Speicherplatz? Dies ist insbesondere bei den später eingeführten Feldern (Kapitel 9.1.1) interessant, wo je nach gewählten Basistyp eines Feldes bei einer Milliarde Feldelementen 1 GB (Basistyp `byte`) oder 8 GB (Basistyp `long`) anfallen würden. Wenn man als Programmierer weiß, dass die vorkommenden Zahlen beispielweise nur zwischen 0 und 100 variieren können, wäre der kleinere Basistyp dann angebracht.

Als "stille Übereinkunft" gilt, dass der Datentyp `int` generell für alle Ganzzahlen genommen wird, wenn nicht besondere Gründe dagegen sprechen (begrenzter Speicher einerseits, sehr große Zahlen andererseits). Auf die Besonderheiten der Typen `byte` und `short` im Zusammenhang mit Ausdrücken wird in Kapitel 7.6 eingegangen.

Das Syntaxdiagramm zur Bildung ganzzahliger Werte (Konstanten) ist in Abbildung 4.3 zu sehen. Die einfachste und weitaus gebräuchlichste Form ist die dezimale Schreibweise in der gewohnten Form wie etwa `1234`. Ohne einen Suffix ist der so gebildete Wert vom Typ `int`. Durch Anhängen des Suffix `l` oder `L` wird dieser Wert explizit zu einem Wert des Datentyps `long` und nimmt demzufolge auch 64 Bit / 8 Byte Speicherplatz ein. Alternativ kann man Werte auch in Binär-, Oktal- und Hexadezimaldarstellung angeben (jeweils für `int` und `long`). Dies geschieht durch einen Präfix `0b` für eine Angabe in Binärdarstellung, einen Präfix `0` (führende Null) für Oktaldarstellung und einen Präfix `0x` für Hexadezimaldarstellung gefolgt von jeweils möglichen Ziffern der entsprechenden Darstellung.

#### **Beispiel 4.18:**

Beispiele für `int`-Werte sind: `3`, `03`, `0x3`, `0x4a`, `2128`, `0b110`

Beispiele für `long`-Werte sind: `3l`, `3L`, `03L`, `0x3L`, `2128L`, `0b110L`

Wie man leicht erkennt ist die Verwendung des Buchstabens `l` (Kleines L) problematisch, da es von einem Menschen leicht mit der Ziffer 1 verwechselt werden kann. Deshalb sollte man in den Fällen, wo ein solcher Suffix benötigt wird, nur `L` (großes L) verwenden. ♦

Wie man in Tabelle 4.5 der Operatoren auf ganzzahligen Ausdrücken sieht, werden in diesem Zusammenhang Operatorsymbole verwendet, die bereits auch schon bei den Operationen des Datentyps `boolean` verwandt wurden (Beispiel `&`) beziehungsweise jetzt in doppelter Bedeutung vorkommen (`+`). Die konkrete Bedeutung eines solchen Symbols ergibt sich also erst aus dem Zusammenhang der Verwendung (Kontext) und nicht nur aus dem Symbol alleine. Tabelle 4.4 gibt das Syntaxdiagramm zu ganzzahligen Ausdrücken an. Man beachte, dass ein Operator wie `eta<=` zwei ganzzahlige Operanden hat, als Ergebnis aber einen Wahrheitswert liefert.

Für die Java-Datentypen `int` und `long` sind eine Reihe von gleichlautenden Operationen definiert, die in Tabelle 4.5 aufgeführt sind. Bis auf Weiteres wird stillschweigend davon ausgegangen, dass alle Operanden eines Operators den gleichen Typ haben. Auf alle Aspekte der Mischung von Datentypen in solchen Operationen, die Behandlung von Operanden der Typen `byte` und `short` und weitere Punkte wird im Detail in Kapitel 7 eingegangen.

Die Bedeutung der Operationen sollte für die Grundrechenarten und die Vergleichsoperatoren (mit einem Wahrheitswert als Resultat der Ausdrucks) weitgehend selbsterklärend sein. Die Ausdruck mit Addition, Subtraktion und Multiplikation ganzzahliger Werte liefert als Resultat den entsprechenden ganzzahligen Wert in dem zugrundeliegenden Datentyp. Ist der Resultatwert nicht mehr mit  $n$  Bits darstellbar, so tritt ein **Überlauf** ein. In diesem Fall wird nur mit den  $n$  niederwertigsten Bits der Resultatdarstellung weiter gearbeitet, also *mit dem mathematisch falschen Wert!* Es ist an dieser Stelle sehr wichtig darauf hinzuweisen, dass weder die Rechnerhardware, noch der Compiler, noch das Java-Kaufzeitsystem in irgendeiner Weise auf diese Fehlersituation reagiert. Es steht schlicht der falsche Wert als Resultat der Operation fest, und

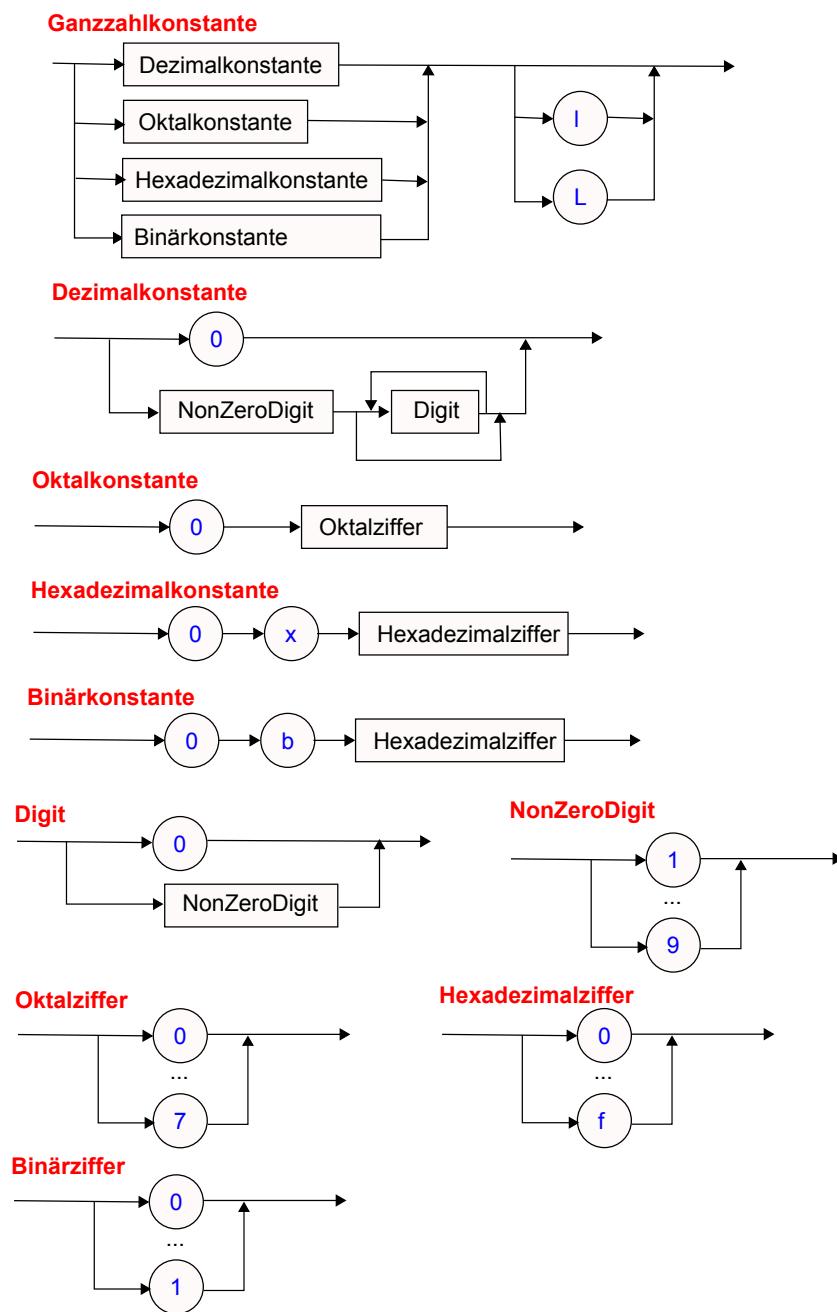


Figure 4.3: Syntaxdiagramm für ganzzahlige Konstanten

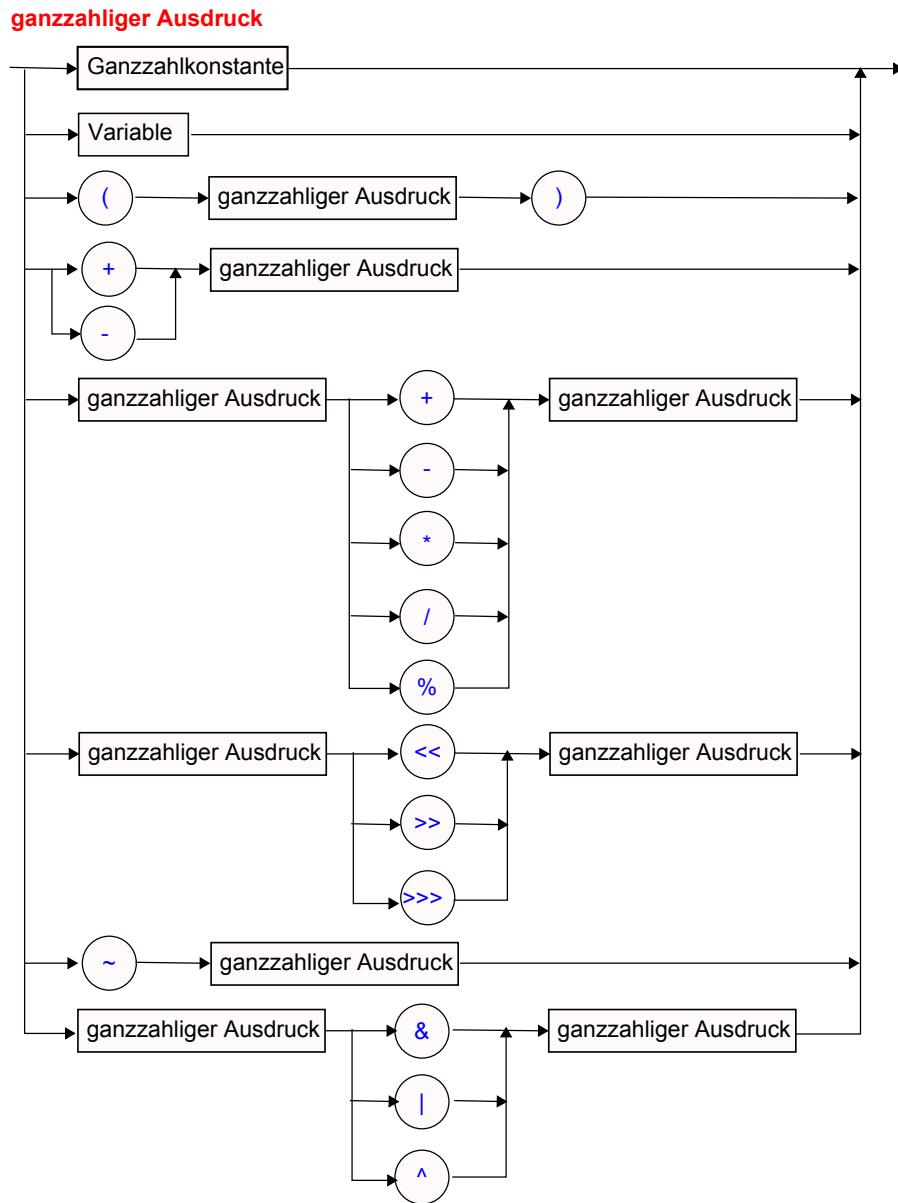


Figure 4.4: Syntaxdiagramm für ganzzahlige Ausdrücke.

Operator	Beispiel	Wert	Wirkung
+	+3	+3	Positivbildung (einstellige Operation)
-	-3	-3	Negativbildung (einstellige Operation)
+	3 + 2	5	Addition (zweistellige Operation)
-	3 - 2	1	Subtraktion (zweistellige Operation)
*	3 * 2	6	Multiplikation
/	3 / 2	1	Ganzzahlige Division
%	3 % 2	1	Modulo-Bildung
<<	1 << 2	4	Bitweises Shiften nach links
>>	5 >> 2	1	Bitweises Shiften nach rechts mit Vorzeichenbehandlung
>>>	5 >>> 2	1	Bitweises Shiften nach rechts
~	~0	-1	Bitweises Komplement
&	5&1	1	Bitweise Und-Operation
	5   2	7	Bitweise Oder-Operation
^	5 ^ 1	4	Bitweise Exclusive-Oder-Operation
==	3 == 2	false	Test auf Gleichheit
!=	3 != 2	true	Test auf Ungleichheit
<	3 < 2	false	Test auf kleiner
<=	3 <= 2	false	Test auf kleiner oder gleich
>	3 > 2	true	Test auf größer
>=	3 >= 2	true	Test auf größer oder gleich

Table 4.5: Java-Operatoren auf ganzzahligen Ausdrücken.

es liegt *ausschließlich in der Verantwortung des Programmierers*, dass solch eine Situation aufgrund der Operandenwerte nicht entstehen kann beziehungsweise dass man sich dessen bewusst ist.

Besonders hinzuweisen ist bei den Grundrechenarten auch auf die Division mit ganzzahligen Werten. Eine ganzzahlige Division bedeutet, dass beispielsweise bei  $7/2$  nicht  $3,5$  das Ergebnis ist, sondern  $3$  (mit einem Rest  $1$ ). Dies wird (insbesondere, aber nicht nur) von Programmieranfängern oft nicht berücksichtigt und falsch angewandt. Ein Vorteil dieser Operation ist, dass das Resultat der Operation wiederum ein ganzzahliger Wert ist und somit innerhalb des Datentyps verblieben wird. Ergänzend zur ganzzahligen Division gibt es die Modulooperation  $\%$ , die für zwei Werte  $x$  und  $y$  den Rest zur ganzzahligen Division liefert. Während also im Beispiel eben  $7/2$  das Resultat  $3$  liefert, liefert  $3 \% 5$  den Wert  $1$ . Eine Division oder Modulobildung durch Null führt zum einem Ausnahmefall (Exception; siehe späteres Kapitel 14), die ohne weitere Behandlung des Programmierers zu einem Abbruch des Programms führt. Dementsprechend sollte ein Programmierer entsprechend dafür sorgen, dass dieser Fall *nicht* eintrifft! Ganzzahlige Division und Modulobildung werden in der Informatik an vielen Stellen genutzt. Man erkennt also schon sehr diffizile Unterschiede in der Fehlererkennung beziehungsweise -behandlung, die man als Programmierer verstehen muss. Während manche Fehler stillschweigend übersehen werden (Überlauf), beenden anderen Fehler (Division durch 0) sofort die Programmausführung.

#### Beispiel 4.19:

In Listing 4.2 ist ein Beispielprogramm zum Datentyp `int` zu sehen. Als Ausgabe erscheint bei Ausführung des Programms auf dem Bildschirm:

41 | i1 \* i2 : 33

Listing 4.2: Beispiel zum Datentyp int.

```

4.1  /**
4.2   * Beispiel zu Operationen aus dem Datentyp int
4.3   */
4.4 public class IntBeispiel {
4.5     public static void main(String [] args) {
4.6
4.7         // zwei Variablen vom Typ int definieren
4.8         int i1, i2;
4.9
4.10        // diesen beiden Variablen Werte zuweisen
4.11        i1 = 3;
4.12        i2 = i1 + 8;
4.13
4.14        // Wir geben das Ergebnis einiger Operationen aus
4.15        System.out.println("i1 * i2: " + (i1 * i2));
4.16        System.out.println("11 / 3: " + (11 / 3));
4.17        System.out.println("11 % 3: " + (11 % 3));
4.18    }
4.19 }
```

```

4.2 11 / 3: 3
4.3 11 % 3: 2
```

**Beispiel 4.20:**

Die Aufgabenstellung ist für eine dreistellige natürliche Zahl die Quersumme zu berechnen. Zur Entwicklung einer Lösung hilft, dass für eine beliebige Zahl die Modulooperation mit 10 die letzte Dezimalziffer als Ergebnis liefert. Beispiel: 381 module 10 liefert 1, weil  $381 / 10 = 38$  ist mit einem Rest von 1. Damit kan man also die letzte Dezimalziffer einer *beliebigen* Dezimalzahl ermitteln. Eine weitere Überlegung führt die Behandlung der verbleibenden Teilzahl auf die Ursprungssituation zurück, für die ja schon eine Teillösung bekannt ist (wie man an die letzte Ziffer kommt). In einer Dezimalzahl lässt sich die letzte Dezimalziffer wertmäßig eliminieren, indem man die Zahl ganzzahlig durch 10 dividiert. Beispiel:  $321 \% 10$  ist 1,  $321 / 10$  liefert 32. Die Behandlung einer zweistelligen Zahl (nach Elimination der letzten Ziffer) kann man also wieder auf den Fall der letzten Ziffer plus Terstzahl zurückführen. Das entsprechende Programm für die Berechnung der Quersumme einer genau dreistelligen Dezimalzahl ist in Listing 4.3 zu sehen. ♦

Ganzzahlige Datentypen können auch für Bitfolgen "missbraucht" werden, also etwa eine Folge von 32 Bit im Zusammenhang mit dem Datentyp `int`. In diesem Zusammenhang werden die Operationen des nächsten Blocks der Tabelle 4.5 genutzt, die sogenannten Shift-Operationen. Diese nehmen die Bitdarstellung des linken Operanden und schieben die Bit um soviele Positionen in die eine oder andere Richtung, wie der zweite, rechte Operand angibt. Null-Bit werden auf der Einfügeseite nachgeschoben, auf der anderen Seite fallen entsprechend viele Bit wieder heraus. Bildlich wird also eine Bitfolge nach links oder rechts verschoben und Bits mit dem Wert 0 nachgeschoben. Der Operator `<<` schift die Bits von rechts nach links, also die niederwertigsten Bit bekommen wandern zu einer höherwertigen Position. Beim Shift-Operator `>>>` gilt entsprechend, dass die höherwertigen Bit zu niederwertigen Positionen wandern. Beim Shift-Operator `>>`

Listing 4.3: Berechnung der Quersumme einer dreistelligen Zahl.

```

4.1 /**
4.2 * Bilde die Quersumme einer dreistelligen Zahl
4.3 */
4.4 public class Quersumme3 {
4.5
4.6     public static void main(String [] args) {
4.7         int zahl = 321;      // Beispielzahl
4.8         int summe = 0;       // Quersumme (zu berechnen)
4.9
4.10        // letzte Ziffer extrahieren
4.11        summe = summe + (zahl % 10);
4.12        // letzte Ziffer der aktuellen Zahl eliminieren
4.13        zahl = zahl / 10;
4.14
4.15        // ursprünglich vorletzte Ziffer extrahieren
4.16        summe = summe + (zahl % 10);
4.17        // letzte Ziffer der aktuellen Zahl eliminieren
4.18        zahl = zahl / 10;
4.19
4.20        // ursprünglich drittletzte Ziffer extrahieren
4.21        summe = summe + (zahl % 10);
4.22
4.23        // Ergebnis / Quersumme auf Bildschirm ausgeben
4.24        System.out.println("Quersumme ist " + summe);
4.25    }
4.26}

```

werden negative Zahlen (höchstwertige Bit hat also den Wert 1) speziell behandelt, indem die Shift-Operation wie eben beschrieben angewandt wird (höher- nach niederwertig), aber der Wert der eingefügten Bit sich aus dem alten Wert des höchstwertigen Bit ergeben (1 statt 0, wenn der Ursprungswert negativ war).

### Beispiel 4.21:

Gegeben sei der Ausdruck `1 << 2`. Der linke Operand `1` hat die interne Darstellung `0...00012`. Der rechte Operand `2` besagt nun im Zusammenhang mit dem Linksshiftoperator `<<`, dass zwei Null-Bits von rechts reingeschoben werden, die zwei höchstwertigen Bits fallen raus. Das Resultat ist also `0...0001002`, also der int-Wert 4. Das Linksshiften entspricht einer Multiplikation mit einer Zweierpotenz ( $2^x$  falls der Wert des rechten Operand `x` ist).

Gegeben sei der Ausdruck `13 >>> 2`. Der linke Operand `13` hat die interne Darstellung `0...00011012`. Der rechte Operand `2` besagt nun im Zusammenhang mit dem Rechtsshiftoperator `>>>`, dass zwei Null-Bits von links reingeschoben werden, die zwei niederwertigsten Bits fallen heraus. Das Resultat ist also `0...000112`, also der int-Wert 3. Das Rechtsshiften entspricht einer ganzzahligen Division mit einer Zweierpotenz ( $2^x$  falls der Wert des rechten Operand `x` ist).

Gegeben sei der Ausdruck `-1 >> 2`. Der linke Operand `-1` hat die interne Darstellung `1111...11112`. Der rechte Operand `2` besagt nun im Zusammenhang mit dem Rechtsshiftoperator mit Vorzeichenbehandlung `>>`, dass zwei Bits von links reingeschoben werden, die zwei niederwertigsten Bits fallen heraus. Der Wert der eingefügten Bits ergibt sich aus dem ursprünglichen Wert des höchstwertige Bit (eine Eins im Beispiel). Das Resultat ist also `1111...11112`, also unverändert der int-Wert `-1`.

Listing 4.4: Beispiel für Bit-Operationen.

```

4.1  /**
4.2   * Ermitteln der Werte der untersten zwei Bits eines int-Wertes
4.3   */
4.4  public class Bitwerte {
4.5
4.6      public static void main(String [] args) {
4.7
4.8          // Beispielwert
4.9          int wert = -1234567;
4.10         // hier wird jeweils das Bit extrahiert
4.11         int bitWert;
4.12
4.13         //----- 1. Bit behandeln -----
4.14
4.15         // alle anderen Bit ausser unterstem Bit maskieren
4.16         bitWert = wert & 0x1;
4.17
4.18         // Bit-Wert ausgeben
4.19         System.out.println("1. Bit hat den Wert " + bitWert);
4.20
4.21         //----- 2. Bit behandeln -----
4.22
4.23         // nach rechts shiften
4.24         wert = wert >> 1;
4.25
4.26         // alle anderen Bit ausser unterstem Bit maskieren
4.27         bitWert = wert & 0x1;
4.28
4.29         // Bit-Wert ausgeben
4.30         System.out.println("2. Bit hat den Wert " + bitWert);
4.31     }
4.32 }
```

In dem letzten zu besprechenden Operatorblock aus Tabelle 4.5 sind die bitweisen Negations-, Und-, Oder- und Exklusives-Oder-Operationen angegeben. Bei ihnen werden die einzelnen Bits der beiden Operanden (des Operanden bei der Negation  $\sim$ ) entsprechend der Operation verknüpft; die resultierenden Bit ergeben insgesamt den Ergebniswert.

### Beispiel 4.22:

Für den Ausdruck  $5 \& 3$  werden die Bits mit der Operation Und verknüpft.  $5_{10} = 0000\dots0101_2 \& 3_{10} = 0000\dots0011_2 = 0000\dots0001_2 = 1_{10}$

Für den Ausdruck  $5 \mid 3$  werden die Bits mit der Operation Oder verknüpft.  $5_{10} = 0000\dots0101_2 \mid 3_{10} = 0000\dots0011_2 = 0000\dots0111_2 = 7_{10}$

Für den Ausdruck  $5 \wedge 3$  werden die Bits mit der Operation Exklusives Oder verknüpft.  $5_{10} = 0000\dots0101_2 \wedge 3_{10} = 0000\dots0011_2 = 0000\dots0110_2 = 6_{10}$  ♦

### Beispiel 4.23:

Für eine int-Variable beliebigen Inhalts soll ermittelt werden, welche Werte in den zwei niederwertigsten Bits enthalten sind (jeweils 0 oder 1 möglich). Die Überlegung für eine Lösung ist folgendermaßen. Wenn man

einen Wert  $z_{n-1}z_{n-2}\dots z_2z_1z_0$  hat und möchte den Wert des untersten Bits ermitteln, also ob dieser Wert 0 oder 1 ist, so kann man dieses Bit isolieren, indem man alle anderen Bits auf 0 setzt. Nach dieser Operation ist der Gesamtwert 0 oder 1, je nachdem, ob das unterste Bit des ursprünglichen Wertes 0 oder 1 war. Man muss allerdings beachten, dass man den ursprünglichen Wert noch weiter benötigt. Hat man das unterste Bit behandelt, so kann man durch einen Rechts-Shift um eine Position die Ausgangslage wieder herstellen, die man bei der Extraktion des untersten Bits hatte, nur, dass diesmal das ursprünglich zweite Bit auf der untersten Position steht. In Listing 4.4 ist das entsprechende Java-Programm zu sehen.

Wie man also sieht, lässt sich ein int-Wert leichzeitig als Bitfolge als auch als ganze Zahl ansehen und auch auf die eine oder andere Weise manipulieren. ♦

Sind Operanden zum Beispiel einer Additionsoperation unterschiedlichen Ganzahltyps, etwa Operand 1 vom Typ `int` und Operand 2 vom Typ `long`, so findet implizit eine Typumwandlung hin zum "größeren" Typ statt. Auf Typumwandlungen wird in Kapitel 7.6 detailliert eingegangen. Da die Thematik der Typumwandlungen nicht einfach ist (einige Aktionen findet implizit statt) wird insbesondere Programmieranfängern empfohlen, bei der Anwendung eines Operators darauf zu achten, dass beide Operanden gleichen Typs sind.

Auf einem Digitalrechner sind alle arithmetischen Operationen auf Ganzzahlen exakt, das heißt jede Operation liefert das mathematisch exakte Ergebnis. Ausnahmen von dieser Regel sind lediglich die oben schon angesprochenen Bereichsüberschreitungen, wenn das Ergebnis einer solchen arithmetischen Operation außerhalb des darstellbaren Bereiches dieses Datentyps liegt.

#### Beispiel 4.24:

Betrachtet wird die größte darstellbare positive Zahl im Datentyp `byte`:  $127_{10} = 0111111_2$ . Addieren man 1 zu dieser Zahl (Addition des Datentyps `byte`), so ergibt sich:

$$\begin{array}{r}
 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
 + & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
 \hline
 1 & 1 & 1 & 1 & 1 & 1 & 1 & \text{Übertrag} \\
 = & 1 & 0 & 0 & 0 & 0 & 0 & 0
 \end{array}$$

das heißt das Ergebnis dieser Operation ist  $10000000_2 = -128$ . Diese sogenannte **Bereichsüberschreitung** wird nicht als solche behandelt und *es wird mit dem falschen Wert weitergerechnet*. Anders ausgedrückt: alle Operationen sind im Endeffekt Modulo-Operationen respektive des darstellbaren Bereiches. Nochmals: es liegt *in der Verantwortung des Programmierers* dafür zu sorgen, dass solche Fälle nicht auftreten! ♦

## 4.4 Fließkommawerte

Für die Darstellung ganzer Zahlen wurde im letzten Kapitel eine Lösung gefunden. In vielen Anwendungsfällen reichen aber ganze Zahlen nicht aus, sondern rationale, reelle oder komplexe Zahlen sind notwendig. Hier bieten Programmiersprachen (und viele Prozessoren auch direkt in Hardware) Unterstützung für eine Teilmenge der reellen Zahlen an. Um auch hier zu einer Darstellungslösung für reelle Zahlen zu gelangen soll in einem ersten Schritt mit einer offensichtlichen Lösungsmöglichkeit aufbauend auf der Lösung für Ganzzahlen begonnen werden, die aber so nicht belassen wird, sondern in einem zweiten Schritt noch modifiziert wird.

#### 4.4.1 Darstellungsformate für Fließkommazahlen

Begonnen werden soll mit einem Beispiel zu der bekannten Darstellung von "Kommazahlen" im Dezimalsystem. Schaut man sich beispielsweise die Zahlendarstellung  $12,34_{10}$  an, so ist die Bedeutung  $12,34_{10} = 1 \cdot 10^1 + 2 \cdot 10^0 + 3 \cdot 0,1 + 4 \cdot 0,01 = 1 \cdot 10^1 + 2 \cdot 10^0 + 3 \cdot 10^{-1} + 4 \cdot 10^{-2}$ . Für eine beliebige solche Dezimaldarstellung  $z_{n-1} \dots z_0, d_1 \dots d_m$  mit Vorkommaanteil  $z_{n-1} \dots z_0$  und Nachkommaanteil  $d_1 \dots d_m$  ist der Wert dieser Darstellung  $\sum_{i=0}^n z_i \cdot 10^i + \sum_{j=1}^m d_j \cdot 10^{-j}$ . Dieses Vorgehen kann man auch verallgemeinern für beliebige Basen  $B \neq 0$ . Zur Darstellung von reellen Zahl bezogen auf eine Zahlenbasis  $B$  kann man den Vorkommaanteil und den Nachkommaanteil jeweils durch eine eigene Zahl kodieren:  $z_{n-1} \dots z_0$  für den Vorkommaanteil und  $d_1 \dots d_m$  für den Nachkommaanteil. Dies nennt man eine **Festkommazahl**. Der Wert solch einer Zahlendarstellung ergibt sich dann analog zu den bereits bekannten Ganzzahlen und dem obigen Dezimalbeispiel für  $B = 10$  durch:

$$wert(z_{n-1} \dots z_0, d_1 \dots d_m)_B = \sum_{i=0}^{n-1} z_i \cdot B^i + \sum_{j=1}^m d_j \cdot B^{-j}$$

##### Beispiel 4.25:

Es soll der Wert der Zahlendarstellung  $011.110_2$  ermittelt werden, also zur Basis 2.

$$\begin{aligned} 011.110_2 &= \sum_{i=0}^2 z_i \cdot 2^i + \sum_{j=1}^3 d_j \cdot 2^{-j} \\ &= (0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0) + (1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3}) \\ &= (0 + 2 + 1) + (0,5 + 0,25 + 0) \\ &= 3,75_{10} \end{aligned}$$

Der Nachteil solch einer fixen Aufteilung in einen Vorkomma- und Nachkommaanteil zur Darstellung reeller oder gebrochener Zahlen ist allerdings, dass der darstellbare Zahlenbereich größtmäßig sehr klein wird.

##### Beispiel 4.26:

Es soll diesem Beispiel davon ausgegangen werden, dass insgesamt 16 Bit zur Darstellung von Zahlen zur Verfügung. Für eine Ganzzahl in Zweierkomplementdarstellung bedeutet dies einen darstellbaren Bereich von  $[-2^{15}, \dots, 2^{15} - 1] = [-32768, \dots, 32767]$ . Teilt man diese 16 Bit nun für eine reelle Zahl in 8 Bit für den Vorkommaanteil und 8 Bit für den Nachkommaanteil ein, so ergibt sich damit ein Wertebereich, der in  $[-2^7 - 1, \dots, 2^7] = [-129, \dots, 128]$  enthalten ist und damit wesentlich kleiner als für Ganzzahlen ist. ♦

Das Problem der Festkommadarstellung liegt darin begründet, dass eine *feste* Aufteilung gefunden werden muss für die Genauigkeit (Nachkommaanteil) und der Größe einer Zahl (Vorkommaanteil). Dies sind bei der Festkommadarstellung zwei gegeneinander konkurrierende Ziele mit beschränkten Ressourcen (n Bits insgesamt zur Verfügung).

Ein ähnliches Problem tritt auch bei der Anzeige großer Zahlen auf einem (besseren) Taschenrechner auf. Übersteigt ein Zahlenwert eine bestimmte Größe (Anzahl Stellen auf dem Display reicht nicht aus), so wird in der Darstellungsform dieser Zahl gewechselt und das halblogarithmischen Format genommen. Beispielsweise bedeutet Anzeige  $2,897E12$  die Zahl 289700000000.  $E12$  steht dabei für  $10^{12}$ .

Reelle Zahlen werden deshalb intern in Rechnern auch in einem halbologarithmischen Format dargestellt bestehend aus einem **Vorzeichen**  $V$ , einer **Mantisze**  $M$  und einem **Exponenten**  $E$ . Ein Beispiel in Dezimalschreibweise wäre etwa  $+6,02 \cdot 10^{23}$ . Allgemein ergibt sich der Wert einer solchen halbologarithmischen Darstellung bestehend aus Vorzeichen  $V$ , Mantisze  $M$  und Exponent  $E$  zu einer Basis  $B$  mit:

$$wert(V, M, E)_B = (-1)^V M \cdot B^E$$

Der Exponent  $E$  kann sowohl positiv als auch negativ sein.

Man spricht von einer **normalisierten Mantisse**, wenn für eine Mantisse  $M \neq 0$  zu einer Basis  $B$  gilt:  $1 \leq M < B$ . Oder anders und etwas salopp ausgedrückt: vor dem Komma steht genau eine Ziffer. Eine Darstellung einer reellen Zahl im halbologarithmischen Format und normalisierter Mantisse nennt man **Fließkommazahl** oder auch **Gleitkommazahl**. Jede normalisierte Mantisse zur Basis 2 hat damit das Format  $1, m_2 m_3 \dots m_n$ , also immer als erste Ziffer eine 1.

#### Beispiel 4.27:

Die Zahl  $314_{10} = 314 \cdot 10^0$  hat eine nichtnormalisierte Mantisse. Man kann aber leicht die Mantisse normalisieren, indem man die Mantisse und entsprechend den Exponenten anpasst:  $314_{10} = 3,14 \cdot 10^2$  und es gilt  $1 \leq 3 < 10$ . Die genutzten Operationen in der Normalisierung waren "Komma verschieben" und Anpassen des Exponenten. ◆

Verwendet man  $m$  Bit für eine normalisierte Mantisse zur Basis 2, so ist damit die Zahl  $d_1 \cdot 2^{-1} + \dots + d_m \cdot 2^{-m}$  darstellbar. Durch die Beschränkung auf  $m$  Bit ist damit aber auch gleichzeitig die Genauigkeit beschränkt, mit der man eine reelle Zahl in solch einem Format darstellen kann. Im Mittel hat man damit einen **Rundungsfehler** von  $2^{-(m+1)}$ . Gleichzeitig ist daraus aber auch ersichtlich, dass sich nur reelle Zahlen exakt darstellen lassen, die der Summe von Zweierpotenzen (negativen Zweierpotenzen bei normalisierter Mantisse) entsprechen.

#### Beispiel 4.28:

Das bekannte Dezimalsystem soll als Beispiel genommen werden. Hat man 5 Ziffern zur Verfügung und möchte im Dezimalsystem die Zahl 0,1 darstellen, so ist das kein Problem (nämlich 0,10000), die Anzahl an Ziffern in der Mantisse ist für diese Zahl ausreichend. Möchte man aber die Zahl 0,123456789 mit einer solchen fünfstelligen Mantisse darstellen, so gibt es ein Problem. Darstellbar sind nur die Werte 0,12345 (abgeschnitten) und 0,12346 (aufgerundet) als nächste Näherungen, aber der eigentliche Wert 0,123456789 ist mit dieser Anzahl an Mantissenziffern nicht exakt darstellbar. Der Fehler der Darstellung beträgt damit 0,000006789 (abgeschnitten) beziehungsweise 0,000003211 (aufgerundet). ◆

Nimmt man statt der Basis 10 die Basis 2, so gibt es die gleichen Probleme der Darstellbarkeit und Genauigkeit, nur sind die Zahlen natürlich anders, die genau oder nicht genau darstellbar sind. Um zum Beispiel die für uns im Dezimalsystem als sehr einfach anzusehende Zahl 0,1 im Zahlensystem zur Basis 2 darzustellen, müsste man also Ziffern  $d_1 \dots d_m$  zu Zweierpotenzen finden, so dass gelten würde:  $\sum_{j=1}^m d_j \cdot 2^{-j} = 0,1_{10}$ . Versuchen Sie es! Leider hat aber die Zahl  $0,1_{10}$  als Summe von Dualbrüchen eine periodische Darstellung!  $0,1_{10} = 0,0001100110011\dots_2 = 0,\overline{00011}_2$  und kann damit *prinzipiell* nicht exakt in Dualdarstellung dargestellt werden, egal wie groß die Mantisse gewählt wird! Die sollten Sie sich merken!

Aus der Diskussion ergibt sich also insgesamt, dass sich bei einer Festlegung auf eine feste Zahl an Bit für die Mantisse (und Exponent) sich manche reellen Zahlen nicht exakt als Fließkommazahl zur Basis 2 darstellen

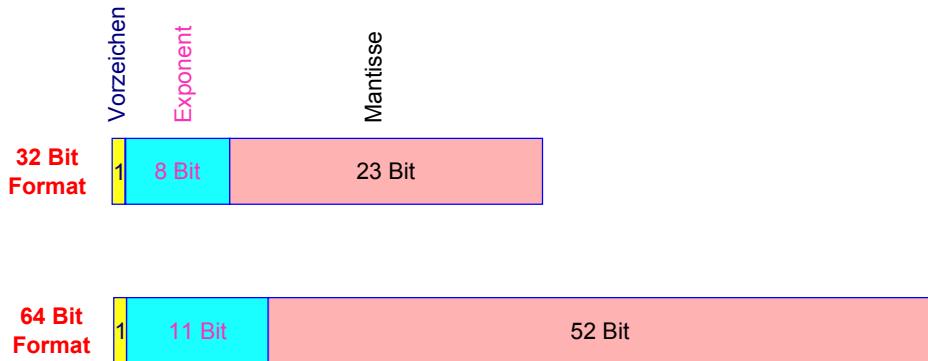


Figure 4.5: Fließkommadarstellung einer 32-Bit und 64-Bit Zahl nach IEEE 754-2008

lassen. Um genauer zu sein ist es eher die Ausnahme, dass eine Zahl aus dem unendlich großen Zahlenbereich der reellen Zahlen in dieser Form exakt darstellbar ist.

Nach der Festlegung auf negative Zweierpotenzen für die normalisierte Mantisse soll nun der Exponent einer Fließkommazahl genauer betrachtet werden. Zur Darstellung des Exponenten, der ja positiv oder negativ sein kann, könnte man analog zu der Darstellung von ganzen Zahlen eine Zweierkomplementdarstellung wählen. In der Praxis wird dies aber nicht getan und statt dessen ein **Excess-Code** verwendet. Der Hintergrund für diesen auf den ersten Blick eher ungewöhnlichen Weg liegt im dadurch möglichen sehr schnellen Vergleich von Fließkommazahlen in Prozessoren, der mit diesem Ansatz bit-weise geschehen kann und worauf hier nicht näher eingegangen wird. Ein Excess- $X$ -Code ist nichts anderes als eine Kodierung für Werte im Wertebereich  $A = [-X, +X]$  in dem Bereich  $B = [0, 2X]$ . Für einen gegebenen Wert  $a$  erhält man den kodierten Wert  $b$ , indem man den Wert  $X$  dazuaddiert:  $b = a + X$ . Umgekehrt erhält man zu einem kodierten Wert  $b$  den ursprünglichen Wert, indem man von  $b$   $X$  subtrahiert:  $a = b - X$ .

Alle kodierten Werte sind damit positiv im Bereich  $[0, 2X]$  und werden binär darstellt (Koeffizienten zu Zweierpotenzen, also die Binärkodierung). Wählt man beispielsweise eine Ursprungsbereich von  $A = [-127, +127]$ , so ist demzufolge  $X = 127$  und man benötigt insgesamt einen Exponenten mit 8 Bit.

#### Beispiel 4.29:

Der Ursprungsbereich ist  $A = [-127, +127]$ . Der Excesswert ist  $X = 127$ . Damit ergibt sich:

- Ursprungswert  $a = -1$ , kodierter Wert  $b = -1 + 127 = 126$ , Darstellung als Binärzahl  $1111110_2$ .
- Ursprungswert  $a = 0$ , kodierter Wert  $b = 0 + 127 = 127$ , Darstellung als Binärzahl  $1111111_2$ .
- Ursprungswert  $a = 1$ , kodierter Wert  $b = 1 + 127 = 128$ , Darstellung als Binärzahl  $10000000_2$ . ◆

Fast alle heutigen Prozessoren implementieren Fließkommazahlen nach dem Standard IEEE 754-2008 beziehungsweise dessen Vorgänger IEEE 754-1985. In dem Standard werden einerseits das genaue Speicherformat (wieviele Bit für Mantisse, Exponent) festlegt und andererseits aber auch Anforderungen an die Ausführung von Basisoperationen spezifiziert, zum Beispiel das Verhalten bei Bereichsüberlauf. Der Standard beschreibt unter anderem mehrere Formate zur Darstellung von Fließkommazahlen (siehe Abbildung 4.5):

- 32-Bit (4 Byte) Zahlen, in Programmiersprachen meist der Datentyp `float`. Die 32 Bit sind aufgeteilt in 1 Vorzeichen-Bit (S), 23 Bit Mantisse (M) und 8 Bit Exponent (E). Der Exponent wird im Excess-127-Code dargestellt. Mit  $m = 23$  ergibt sich damit eine Genauigkeit von  $2^{-(23+1)} = 5,96 \cdot 10^{-8}$  beziehungsweise 7 signifikante Dezimalnachkommastellen.

Format	Größe Mantissee	Größe Exponent	minimaler Exponent	maximaler Exponent	signifikante Dezimalstellen	kleinste Zahl	größte Zahl
32 Bit	23 Bit	8 Bit	-126	127	7	$\pm 1,175 \cdot 10^{-38}$	$\pm 3,403 \cdot 10^{38}$
64 Bit	52 Bit	11 Bit	-1022	1023	15	$\pm 2,225 \cdot 10^{-308}$	$\pm 1,798 \cdot 10^{308}$

Table 4.6: Eigenschaften von IEEE Fließkommamatypen

- 64-Bit (8 Byte) Zahlen, in Programmiersprachen meist der Datentyp `double`. Die 64 Bit sind aufgeteilt in 1 Bit Vorzeichen, 11 Bit Exponent und 52 Bit Mantisse. Der Exponent wird im Excess-1023-Code dargestellt. Hier gilt mit  $m = 52$  für die Genauigkeit  $2^{-(52+1)} = 9 \cdot 10^{-15}$ , das heißt 14 signifikante Dezimalnachkommastellen.
- 128-Bit (16 Byte) Zahlen mit einem Vorzeichenbit, 15 Bit Exponent und 112 Bit Mantisse. Der Exponent wird im Excess-16383-Code dargestellt. Dieser Typ ist selten in Hardware implementiert und ebenso in kaum einer Programmiersprache als Datentyp verfügbar. Für die Genauigkeit ergibt sich  $2^{-(112+1)} = 9,6 \cdot 10^{-35}$ , also 34 signifikante Dezimalnachkommastellen.

Weiterhin werden in der aktuellen Version des Standards auch zwei weitere Binärformate (16 Bit, 128 Bit) sowie Dezimalformate zur Basis 10 definiert, auf die aber nicht weiter eingegangen wird. Tabelle 4.6 fasst wesentliche Eigenschaften dieser Zahlen zusammen.

Um nochmals die Genauigkeit anzusprechen: wenn Sie beispielsweise das 32 Bit Format auswählen, womit die Mantisse auf 23 Bit festgelegt ist, so kann damit die Zahl  $0,5_{10} = 2^{-1} = 0,1_2$  exakt dargestellt werden, auch die Zahl  $0,0000011921_{10} = 2^{-23} = 0,000000000000000000000001_2$ , aber der Wert  $(2^{-1} + 2^{-23})$  ist nicht mehr exakt darstellbar, weil die Mantisse zu wenig Ziffern für die exakte Darstellung besitzt (zwei binäre Einsen weit voneinander entfernt).

Im IEEE-754 Standard bedeutet eine 1 im Vorzeichenbit einen negativen Wert und eine 0 entsprechend einen positiven Wert. Der Exponent wird als eine ganze Zahl im Excess-127-Code interpretiert. Der Wert der Mantisse  $M$  wird als  $(1.M)_2$  interpretiert, das heißt 1.0 plus dem Wert von  $M$  (es gibt Ausnahmen wie die Zahl 0, worauf hier nicht eingegangen wird). Bei dieser IEEE-Darstellung wird also die Mantisse nicht auf 0 normiert, sondern auf 1 (binär). Der Wert, der durch  $V$ ,  $M$  und  $E$  dargestellt wird, entspricht damit  $(-1)^V(1.M)(2^{E-127})$ . Um nun eine reelle Zahl in ein IEEE-Darstellungsformat zu bringen, sind mehrere Schritte notwendig:

1. Darstellung der reellen Zahl als Festkommadualzahl mit einem Vorkommaanteil und einem Nachkommaanteil
2. Normalisierung der Mantisse, so dass genau eine binäre 1 vor dem Komma steht (für alle Werte außer dem Wert 0 möglich)
3. Anpassung des Exponenten: durch die Normalisierung muss der Exponent so angepasst werden, dass insgesamt nach wie vor der gleiche Wert dargestellt wird
4. Kodierung des Exponenten im Excess-127 Format
5. Zusammenstellen der 32 Bit Zahl aus den einzelnen Komponenten (Vorzeichen, Exponent, Mantisse)

#### Beispiel 4.30:

Die Zahl 0,5 soll als Fließkommazahl dargestellt werden. Die Umwandlung geschieht in den oben beschriebenen Schritten:

1. Festkommazahl erzeugen: Die Dezimalzahl wird in eine Festkommazahl umgewandelt, wie dies oben bereits beschrieben wurde. Dann ergibt sich:  $0,5_{10} = 0,1000\dots_2$ , weil  $0_{10} = 0_2$  (Vorkommaanteil) und  $0,5 = 1 \cdot 0,5 = 1 \cdot 2^{-1}$  (Nachkommaanteil).
2. Normalisierung der Mantisse: dazu wird das Komma der Festkommazahl solange nach rechts oder links verschoben, bis nur genau eine 1 vor dem Komma steht. Für jede Zahl außer der 0 ist das möglich. Der Nachkommaanteil stellt dann die Mantisse dar, die 1 vor dem Komma wird ignoriert (diese 1 ist ja immer da). Im Beispiel bedeutet dies:  $0,1000\dots_2$  wird umgewandelt in  $1,000\dots_2$  ( $0,1_2 = 1,0_2 * 2^{-1}$ ; Rechtsschieben des Kommas ums eins).
3. Anpassung Exponent: Auf die Anzahl an Stellen, um die die Mantisse verschoben wurde, wird der Exponent gesetzt. Im Beispiel ist dies  $-1_{10}$  (Rechtsschieben des Kommas bedeutet negative Zahl, Linksschieben positive Zahl).
4. Kodierung im Excess-127 Format: Kodierung von  $-1$  im Excess-127-Format bedeutet  $127 + (-1) = 126_{10}$  oder  $01111110_2$ .
5. Zusammenstellung der 32 Bit Zahl: Insgesamt würde sich also als Darstellung der Zahl  $0,5$  im IEEE-754 Format für 32 Bit Zahlen die Zahl  $001111100000\dots00$  (1 Bit Vorzeichen, 8 Bit Exponent, 23 Bit Mantisse) ergeben. ◆

### Beispiel 4.31:

Die Zahl  $5,125$  soll als Fließkommazahl im 32 Bit IEEE-Format dargestellt werden. Die Umwandlung geschieht wieder in den Schritten:

1. Festkommazahl erzeugen:  $5,125_{10} = 101,001_2$ , weil  $5_{10} = 101_2$  und  $0,125 = 0 \cdot 0,5 + 0 \cdot 0,25 + 1 \cdot 0,125 (= 0 * 2^{-1} + 0 * 2^{-2} + 1 * 2^{-3})$ .
2. Normalisierung:  $101,001_2$  wird umgewandelt in  $1,01001_2$  (Rechtsschieben um zwei Stellen).
3. Anpassung Exponent: Exponent ist 2 (Rechtsschieben um zwei)
4. Kodierung im Excess-127 Format:  $127 + 2 = 129_{10}$  oder  $10000001_2$ .
5. Zusammenstellung der Zahl: Insgesamt würde sich also als Darstellung der Zahl  $5,125$  im IEEE-754 Format für 32 Bit Zahlen die Zahl  $0100000101001000\dots0$  (1 Bit Vorzeichen, 8 Bit Exponent, 23 Bit Mantisse) ergeben. ◆

Der IEEE-754 Standard kennt einige Bitmuster mit besonderer Bedeutung. So gibt es Darstellungen für  $+\infty = 1.0/0.0$  (Vorzeichen=0, Exponent=128, Mantisse=0) und  $-\infty = -1.0/0.0$  (Vorzeichen=1, Exponent=128, Mantisse=0) sowie eine "Zahl"  $NAN = 0.0/0.0$  (Not A Number), die als Fehlerwert zu betrachten ist. Bezuglich Details dazu sei auf die Originalliteratur und entsprechenden Fachbüchern zur Rechnerarithmetik verwiesen.

Das kleine Programm in Listing 4.5 bestimmt die Maschinengenauigkeit auf einem Rechner. Dabei ist die Maschinengenauigkeit definiert als der kleinste Wert  $\epsilon$ , so dass  $1 + \epsilon \neq 1$ .

Listing 4.5: Bestimmung des Maschinenepsilon.

```

4.1  /**
4.2   * Bestimme das Maschinen-Epsilon
4.3   */
4.4 public class MaschinenEpsilon {
4.5
4.6     public static void main(String [] args) {
4.7         double eps = 1.0;
4.8         while(1.0 + eps > 1.0) {
4.9             System.out.println("eps=" + eps);
4.10            eps = eps / 2.0;
4.11        }
4.12        System.out.println("Maschineneepsilon=" + eps);
4.13    }
4.14 }
```

#### 4.4.2 Mögliche Probleme bei arithmetischen Operationen

Für eine Darstellung von Fließkommazahlen mit einem Mantissen- und Exponentteil fester Länge ergeben sich wichtige Konsequenzen bei arithmetischen Operationen auf solchen Daten. Die Addition etwa zweier Fließkommazahlen geschieht in einem Prozessor nach einem festen Schema:

- 1) Anpassen der beiden Werte auf den gleichen (größeren) Exponenten
- 2) Addition der Werte
- 3) Normalisieren der Mantisse

##### Beispiel 4.32:

Dieses Beispiel wird zur einfacheren Nachvollziehbarkeit anhand von Dezimalzahlen durchgeführt, die analoge Aussage gilt aber natürlich auch für Binärzahlen. Addiert werden sollen die beiden Zahlen  $0,123 \cdot 10^1$  und  $0,999 \cdot 10^3$ , wobei eine Mantissenlänge von 3 Ziffern angenommen wird.

- 1) Anpassen der beiden Werte auf den gleichen (größeren) Exponenten:  $0,123 \cdot 10^1 + 0,999 \cdot 10^3 = 0,00123 \cdot 10^3 + 0,999 \cdot 10^3$ . Durch die feste Mantissenlänge kann der Wert  $0,00123 \cdot 10^3$  nur als  $0,001 \cdot 10^3$  gespeichert werden.
- 2) Addition der Werte:  $0,001 \cdot 10^3 + 0,999 \cdot 10^3 = 1,000 \cdot 10^3$  anstatt des korrekten Wertes  $1,00023 \cdot 10^3$
- 3) Normalisieren der Mantisse:  $1,000 \cdot 10^3 = 0,100 \cdot 10^4$



Zu beachten ist, dass durch das Verschieben der Mantisse im ersten Schritt bei einer festen Mantissenlänge Ziffern "herunterfallen" können, also verlorengehen. Dadurch wird ein prinzipielles Problem bei Operationen auf Fließkommazahlen ersichtlich. Wenn nämlich bei einer Operation das exakte (Zwischen-)Ergebnis aufgrund der begrenzten Mantissenlänge nicht mehr darstellbar ist, so werden die niedrigerwertigsten Ziffern (die also am rechten Rand der Darstellung stehen) weggelassen, da sie nicht mehr in die Darstellung mit begrenzter Zifferanzahl passen. In Prozessoren wird dieses Problem dadurch etwas abgemildert, dass intern bei der Operationen mit einer größeren Mantissenlänge gerechnet wird. Das prinzipielle Problem bleibt aber bestehen. Solche und ähnlich gelagerte Probleme können insbesondere bei Operationen mit Fließkommazahlen in den folgenden Fällen entstehen.

- Zwei Fließkommazahlen mit sehr unterschiedlicher Größe werden addiert. Dabei können Stellen der kleineren Zahl verloren gehen.
- Zwei Fließkommazahlen ähnlicher Größe voneinander subtrahiert. Dadurch kann es zu Auslöschungen

Listing 4.6: Beispiel zur Fließkommaproblematik.

```

4.1  /**
4.2   * Problem bei der Arithmetik mit Fliesskommawerten
4.3   */
4.4  public class Fliesskommaprobleme {
4.5
4.6      public static void main(String [] args) {
4.7          // Addition von Werten sehr unterschiedlicher Groessenordnung
4.8          float x = 0.123456789e-10f;
4.9          float y = 1.0f;
4.10         System.out.println(" " + x + " +" + y + " =" + (x+y));
4.11
4.12         // Subtraktion fast identischer Werte
4.13         x = 123456789.0f;
4.14         y = 123456788.0f;
4.15         System.out.println(" " + x + " -" + y + " =" + (x-y));
4.16
4.17         // Division mit einem Wert nahe bei 0
4.18         x = 1.23456789e37f;
4.19         y = 1e-38f;
4.20         System.out.println(" " + x + " /" + y + " =" + (x/y));
4.21     }
4.22 }
```

kommen.

- Division zweier Fließkommazahlen mit einem Divideng nahe bei 0. Dabei kann es zu einem Bereichsüberlauf kommen.

#### Beispiel 4.33:

Das Programm ist Listing 4.6 erzeugt die Ausgabe:

```

4.1 > java Fliesskommaprobleme
4.2 1.2345679E-11+1.0=1.0
4.3 1.23456792E8-1.23456784E8=8.0
4.4 1.2345678E37/1.0E-38=Infinity
```

### 4.4.3 Fließkommazahlen in Java

In Java werden die beiden IEEE-Formate für 32 und 64 Bit unterstützt. Der darstellbare Bereich von Fließkommazahlen für die beiden IEEE-Formate 32 Bit und 64 Bit und die entsprechenden Datentypen in Java sind in Tabelle 4.7 aufgeführt. An dieser Stelle nochmals der Hinweis, dass in den angegebenen Bereichen natürlich nicht alle reellen Werte exakt darstellbar sind.

Abbildung 4.6 zeigt die Syntaxdiagramme zu Fließkommakonstanten. In Programmiersprachen wird, genau wie im Englischen, der Dezimalpunkt als Trennzeichen zwischen Vor- und Nachkommaanteil verwendet, und nicht wie im Deutschen das Dezimalkomma. Eine Fließkommazahl hat nach diesem Syntaxdiagramm entweder einen Dezimalpunkt oder eine Exponentangabe oder einen Fließkommasuffix oder eine Kombination von diesen Angaben. Liegt keine dieser Angaben vor, so ist dies keine Fließkommakonstante, sondern (falls eine ansonsten korrekte Zahl angeben wurde) eine Konstante vom Typ `int` (vergleiche Syntaxdiagramm

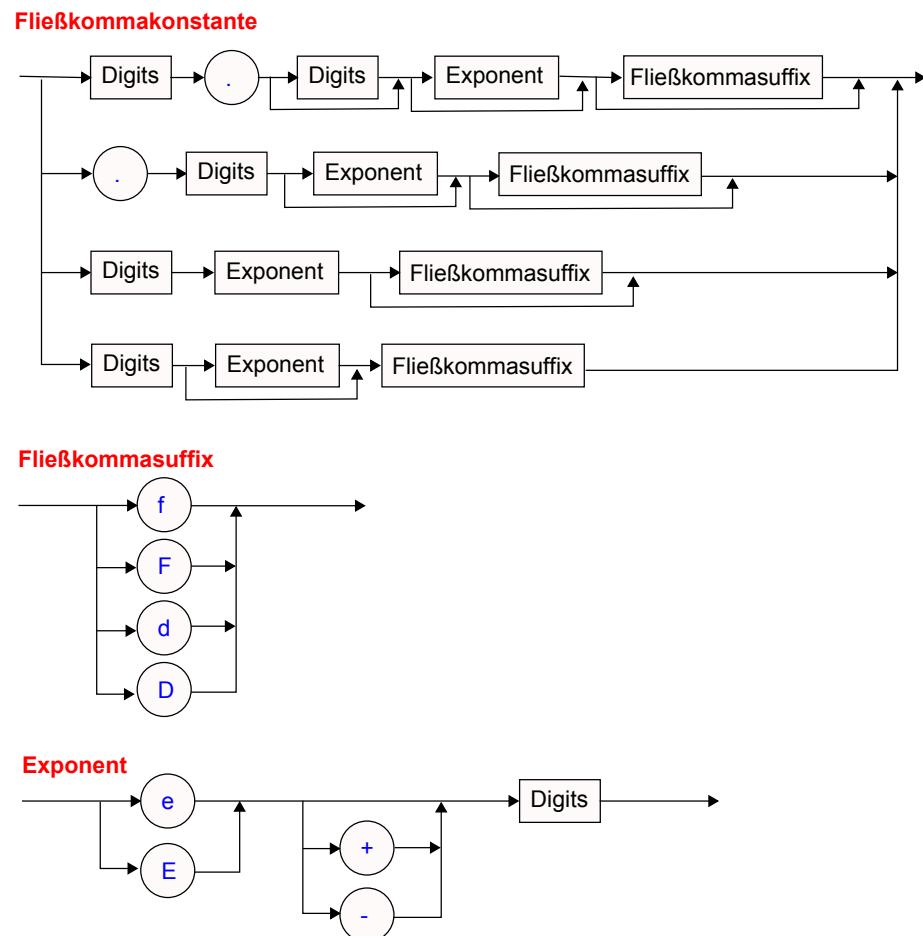


Figure 4.6: Syntax zu Fließkommakonstanten in Java

Typname	Speicherbedarf	Wertebereich
<b>float</b>	4 Byte	$-3,40282346 \cdot 10^{38}$ bis $3,40282347 \cdot 10^{38}$
<b>double</b>	8 Byte	$-1,7976931348623147e + 308$ bis $1,7976931348623147e + 308$

Table 4.7: Fließkommazahlen in Java

Listing 4.7: Beispiel zur Fließkommazahlen.

```

4.1 /**
4.2 * Beispiele zu Fließkommawerten
4.3 */
4.4 public class FloatBeispiel {
4.5     public static void main(String[] args) {
4.6
4.7         float f1, f2;
4.8         double d1, d2;
4.9
4.10        // Fließkommakonstanten des Typs float haben ein Suffix f
4.11        f1 = 0.1f;
4.12        f2 = 0.1f;
4.13        d1 = 0.1;
4.14        d2 = 0.1;
4.15
4.16        // Wir geben das Ergebnis einiger Operationen aus
4.17        System.out.println("f1 * f2: " + (f1 * f2));
4.18        System.out.println("d1 * d2: " + (d1 * d2));
4.19    }
4.20}

```

dazu in Abbildung 4.3)! Wird kein Fließkommasuffix angegeben oder wird einer der Fließkommasuffix **d** oder **D** angegeben, so ist die Fließkommakonstante vom Typ **double** und damit 64 Bit groß. Liegt der Fließkommasuffix **f** oder **F** vor, so ist der Typ **float**. Durch einen positiven oder negativen Exponenten kann der Wert weiter spezifiziert werden, wobei eine Angabe wie **E3** für den Exponentenwert  $10^3$  steht.

### Beispiel 4.34:

Die Zahl 3,14 wird als Fließkommazahl **3.14** geschrieben. Die Schreibweise **0.314e+1** entspricht der gleichen Zahl.

Weitere gültige Angaben für Fließkommakonstanten sind **43.**, **.43**, **.3f**, **.3e1f**.

Die Angabe der Konstanten **.3e1f** ist etwas anderes als die Angabe **.3e1d**. Die Angabe **.3e1f** ist aber äquivalent zur Angabe **3f**. ◆

### Beispiel 4.35:

Das Programm in Listing 4.7 zeigt die Definition von **float**- und **double**-Variablen sowie einfache Operationen mit diesen.

Als Ausgabe erscheint auf dem Bildschirm:

```

4.1 f1 * f2: 0.010000001
4.2 d1 * d2: 0.010000000000000002

```

Operator	Beispiel	Wert	Wirkung
+	+3.	+3.	Positivbildung (einstellige Operation)
-	-3.	-3.	Negativbildung (einstellige Operation)
+	3. + 2.	5.	Addition (zweistellige Operation)
-	3. - 2.	1.	Subtraktion (zweistellige Operation)
*	3. * 2.	6.	Multiplikation
/	3. / 2.	1.	Ganzzahlige Division
%	3. % 2.	1.	Ganzzahlige Division
==	3. == 2.	false	Test auf Gleichheit
!=	3. != 2.	true	Test auf Ungleichheit
<	3. < 2.	false	Test auf kleiner
<=	3. <= 2.	false	Test auf kleiner oder gleich
>	3. > 2.	true	Test auf größer
>=	3. >= 2.	true	Test auf größer oder gleich

Table 4.8: Fließkommaoperatoren in Java

Funktion	Beispielaufruf	Wirkung
<code>Math.abs()</code>	<code>Math.abs(-5.0)</code>	Absolutwert $ x $ einer Zahl
<code>Math.pow</code>	<code>Math.pow(2.0, 4.0)</code>	Potenz $x^y$
<code>Math.sqrt</code>	<code>Math.sqrt(2.0)</code>	Wurzel $\sqrt{x}$
<code>Math.sin()</code>	<code>Math.sin(0.2)</code>	Sinus
<code>Math.cos()</code>	<code>Math.cos(0.2)</code>	Cosinus
<code>Math.tan()</code>	<code>Math.tan(0.2)</code>	Tangens
<code>Math.exp()</code>	<code>Math.exp(1.0)</code>	Exponentialfunktion
<code>Math.log()</code>	<code>Math.log(0.1)</code>	natürlicher Logarithmus zu Basis $e$
<code>Math.log10()</code>	<code>Math.log10(0.1)</code>	Logarithmus zu Basis 10

Table 4.9: Auswahl an mathematischen Funktionen in Java

Zu beachten ist, dass als Ergebnisausgabe *nicht 0.01* erscheint, was das exakte Ergebnis gewesen wäre, sondern ein dazu leicht veränderter Wert. Wieso? Sie sollten diese Frage selbst beantworten können! Sie können dies auch selbst mit einem Taschenrechner nachvollziehen, wenn Sie von der Ausgangssituation in Java starten. ♦

Für Fließkommazahlen bietet Java die üblichen arithmetischen Operationen an. Tabelle 4.8 gibt einen Überblick dazu. Darauf hinaus gibt es eine Vielzahl von mathematischen Funktionen, die zusammen mit einer Java Entwicklung- und Laufzeitumgebung ausgeliefert werden und im Paket `java.Math` zur Nutzung bereitgestellt werden. Die Auswertung eines Funktionwertes geschieht wie in der Mathematik gewohnt durch Angabe des Funktionsnamens gefolgt von dem/den Funktionsargumenten in runden Klammern. Solche Funktionen (in Java werden sie Methoden genannt) werden im Detail in Kapitel 8 vorgestellt. Tabelle 4.9 zeigt eine Auswahl an solchen mathematischen Funktionen in Java. Zu beachten ist dabei, dass die trigonometrischen Funktionen eine Angabe in Radian erwarten. Die Umrechnung einer Gradzahl  $x$  nach Radian  $y$  ist aber einfach:  $y = \frac{x \cdot \pi}{180}$ . Weiterhin gibt es die beiden häufig genutzten Konstanten  $\pi$  und  $e$  Namen, die wie Variablen verwendet werden können: `Math.PI` und `Math.E`.

Listing 4.8: Berechnung zum Schiefen Wurf.

```

4.1  /**
4.2   * Berechnung des Bahngleichung beim Schiefen Wurf
4.3   */
4.4 public class SchieferWurf {
4.5     public static void main(String[] args) {
4.6
4.7     double v0 = 30.0;                      // Anfangsgeschwindigkeit
4.8     double alpha = 45.0 * (Math.PI / 180.0); // Abschusswinkel
4.9     double g = 9.81;                       // Erdschwerebeschleunigung
4.10    double x = 10.0;                      // gegebener x-Wert
4.11    double y;                           // gesuchter y-Wert
4.12
4.13    // Berechnung der Formel
4.14    y = Math.tan(alpha)*x
4.15    - ((g * x*x) / (2.0 * Math.pow(v0 * Math.cos(alpha), 2.0)));
4.16
4.17    // Ausgabe des Ergebnisses
4.18    System.out.println("Hoehe am Punkt " + x + " ist " + y);
4.19  }
4.20 }
```

**Beispiel 4.36:**

Das Programm in Listing 4.8 zeigt den Gebrauch einiger dieser mathematischen Funktionen anhand einer praktischen Anwendung. Aus der Schulphysik ist die Behandlung des schießen Wurfs bekannt. Dabei wird ein Körper mit einer Anfangsgeschwindigkeit  $v_0$  und einem Winkel  $\alpha$  nach oben weggestoßen (Luftwiderstand wird dabei ignoriert). Eine Fragestellung ist nun, was die Höhe dieses Körpers bei einer bestimmten Entfernung entlang der x-Richtung ist. Dazu kann die Formel genutzt werden:

$$y = \tan(\alpha) \cdot x - \frac{g \cdot x^2}{2 \cdot (v_0 \cdot \cos(\alpha))^2}$$

wobei  $g = 9,81 \frac{m}{s^2}$  die Erdschwerebeschleunigung ist. ♦

Anders als die Ganzzahlarithmetik ist die Arithmetik mit Fließkommazahlen *nicht immer exakt!* Die hängt natürlich auch damit zusammen, dass das Ergebnis zum Beispiel der Addition zweier exakt darstellbarer Zahlen nicht als Fließkommazahl darstellbar sein muss (Überlegen Sie sich ein Zahlenbeispiel zu dieser Aussage). David Goldberg hat in einem Artikel mit dem bezeichnenden Titel *What Every Computer Scientist Should Know About Floating-Point Arithmetic* [Gol91] die wichtigsten Eigenschaften von Fließkommazahlen und den Operationen darauf beschrieben.

**Beispiel 4.37:**

Ein weiteres Beispiel aus [Rum88] zur (Un-)Genauigkeit von Fließkommaberechnungen ist mit folgender Aufgabenstellung verbunden. Es soll der Wert  $f$  nach folgender Formel berechnet werden:

$$f = (333.75 - a^2)b^6 + a^2(11a^2b^2 - 121b^4 - 2) + 5.5b^8 + (a/2b)$$

für die Zahlen  $a = 77617$  und  $b = 33096$ . Alle vorkommenden Zahlen sind exakt als Fließkommazahlen darstellbar. Damit ergibt sich das Java-Programm in Listing 4.9, das diese Formel umsetzt. Als Ausgabe erscheint auf dem Bildschirm:

```

4.1 f = 1.1726039400531787
```

Listing 4.9: Beispiel zur Fließkommaproblematik (nach [Rum88]).

```

4-1  /**
4-2   * Fehlerbeispiel von Rump
4-3   */
4-4 public class FPFehler {
4-5     public static void main(String [] args) {
4-6
4-7     double a= 77617.0;
4-8     double b= 33096.0;
4-9     double f;
4-10
4-11    f = (333.75 - Math.pow(a,2)) * Math.pow(b,6)
4-12        + Math.pow(a,2) * (11 * Math.pow(a,2) * Math.pow(b,2) - 121 * Math.pow(b,4) -
4-13            2)
4-14        + (5.5 * Math.pow(b,8)) + (a / (2 * b));
4-15    System.out.println("f = " + f);
4-16 }
4-16 }
```

Dieses Beispiel ist für **double**-Zahlen angegeben (64 Bit Fließkommadarstellung). Alternativ kann man diese Formel auch analog mit dem Typ **float** angeben und (das geht dann aber nicht mehr in Java) als 128-Bit Fließkommaberechnung. Man käme dann zu den folgenden Ergebnissen:

Typ	Ergebniswert
<b>float</b>	1.172604
<b>double</b>	1.1726039400531787
128-Bit	1.17260394005311786318588349045201838

Da alle Werte in den ersten Nachkommastellen übereinstimmen könnte man vermuten, dass mit dem Programm auch annäherungsweise der korrekte Werte berechnet wird und man durch Auswahl eines geeigneten Fließkommatyps (32, 64 oder 128 Bit) die Genauigkeit erhalten kann, die man haben möchte. Dummerweise lautet das korrekte Resultat für diese Formel aber  $-0.827396059946821368141165095479816\dots$ ! Das falsche Ergebnis kommt unter anderem dadurch zustande, dass Rundungsfehler bei der Berechnung von Zwischenresultaten auftreten, die zu diesen komplett falschen Ergebniswerten führen. Dieses Beispiel zeigt eindrucksvoll, dass der Übergang zu einem Fließkommotyp mit höherer Bit-/ Stellenzahl nicht notwendigerweise ein "korrekteres" Resultat liefert, sondern dass ein Informatiker (und andere, die Programme und Algorithmen entwerfen), sich dieser potentiellen Probleme bewusst sein müssen. ♦

## 4.5 Zeichen

Ebenso wie Wahrheitswerte und Zahlen müssen auch Zeichen (Buchstaben, Interpunktionszeichen, Ziffern,...) rechnerintern dargestellt werden, wozu eine geeignete Kodierung notwendig ist. Eine seit langem bekannte Kodierung für Buchstaben und Ziffern ist der Morsecode, der in Tabelle 4.10 in Teilen zu sehen ist und mit dem jedem Buchstaben ein eindeutiger Wert als Folge von Punkten und Strichen zugeordnet wird.

Um eine geeignete Kodierung für Zeichen zu finden, muss man sich zuerst Gedanken darüber machen, was der Zeichenvorrat ist, der kodiert werden muss. Neben den Buchstaben (Klein- wie Großbuchstaben) und Ziffern müssen etwa auch Satzzeichen (Leerzeichen, Punkt, Semikolon etc.) kodiert werden. Zusätzlich hat man aber auch sogenannte Steuerzeichen definiert, mit denen man zum Beispiel das Einrücken von Text (Tabulator) oder das Zeilenende markieren kann. In einem **Zeichensatz** ist festgelegt, mit welchem Binärmuster Zeichen kodiert

a	•—	b	—•••	c	—•—•	d	—••	e	•	f	••—•
g	——•	h	••••	i	••	j	•——	k	—•—	l	•—••
m	——	n	—•	o	———	p	•——•	q	—•——	r	•—•
s	•••	t	—	u	••—	v	•••—	w	•——	x	—••—
y	—•—	z	——••								

Table 4.10: Teil des Morsecodes

untersten Bit	obersten Bit							
	000	001	010	011	100	101	110	111
0000	NUL	DLE	SPACE	0	@	P	—	o
0001	SOH	DC1	!	1	A	Q	a	p
0010	STX	DC2	"	2	B	R	b	q
0011	ETX	DC3	#	3	C	S	c	r
0100	EOT	DC4	\$	4	D	T	d	s
0101	ENQ	NAK	%	5	E	U	e	t
0110	ACK	SYN	&	6	F	V	f	u
0111	BEL	ETB	"`	7	G	W	g	v
1000	BS	CAN	(	8	H	X	h	w
1001	HT	EM	)	9	I	Y	i	x
1010	LF	SUB	*	:	J	Z	j	y
1011	VT	ESC	+	;	K	[	k	z
1100	FF	FS	,	<	L	\	l	{
1101	CR	GS	-	=	M	]	m	
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	←	o	DEL

Table 4.11: ASCII-Zeichensatz

werden, das heißt, welcher Binärwert zum Beispiel dem lateinischen Buchstaben "A" entspricht.

### 4.5.1 ASCII

Historisch haben sich mehrere Zeichensätze parallel beziehungsweise aufeinander aufbauend entwickelt. Der einfachste und gleichzeitig aber auch einer der ältesten ist der **ASCII-Zeichensatz**. ASCII steht für *American Standard Code for Information Interchange*. Im ASCII-Zeichensatz sind unter anderem die 26 Groß- und Kleinbuchstaben des lateinischen Alphabets, alle 10 Ziffern, Interpunktionszeichen, aber auch Steuerzeichen wie vertikaler oder horizontaler Vorschub definiert. Zur Kodierung stehen in diesem Code 7 Bit zur Verfügung, so dass maximal  $2^7 = 128$  Zeichen mit diesem Zeichensatz kodiert werden können. Tabelle 4.11 gibt den ASCII-Code wieder. Beginnend links oben in der Ecke sind die fortlaufenden Kodierungen jeweils spaltenweise nach unten gehend zu lesen.

Die ersten 32 Zeichen, das heißt die ersten beiden Spalten der Tabelle, sind Steuerzeichen wie Zeilenvorschub (LF) oder das Escape-Zeichen (ESC). Zum Beispiel die Bedeutung des Zeilenvorschubzeichens ist, dass das nachfolgende Zeichen in der nächsten Anzeigzeile angezeigt werden soll. Man sieht an dieser Tabelle bei näherer Betrachtung, dass alle Groß- und Kleinbuchstaben in einem zusammenhängenden Bereich abgelegt sind und dass Kleinbuchstaben versetzt (mit einem Offset von  $32_{10}$  beziehungsweise  $20_{16}$ ) zu den

Grossbuchstaben in der Tabelle zu finden sind. Weiterhin sind auch alle Ziffern in ihrer natürlichen Reihenfolge in der Tabelle enthalten. Dies ist bewusst so gewählt worden, um Folgendes einfach durchführen zu können:

- Der Test, ob ein kodiertes Zeichen ein Großbuchstabe ist, lässt sich einfach durchführen, indem getestet wird, ob der Code größer oder gleich dem Code für "A" ist und gleichzeitig kleiner oder gleich dem Code von "Z" ist (Test auf Kleinbuchstaben und Ziffern analog).
- Möchte man einen beliebigen Großbuchstaben in den entsprechenden Kleinbuchstaben umwandeln, muss man lediglich  $32_{10}$  auf den Code addieren.
- Möchte man einen beliebigen Kleinbuchstaben in den entsprechenden Großbuchstaben umwandeln, muss man  $32_{10}$  vom Code subtrahieren.
- Möchte man zu einem Buchstabenzeichen ermitteln, der wievielte Buchstabe im Alphabet dieses Zeichen ist, so muss man vom Code nur den Code von "A" abziehen (und eins addieren, falls man bei eins statt bei null zu zählen beginnt).
- Möchte man zu einem Ziffernzeichen den Wert ermitteln, also beispielsweise zum Zeichen "8" den numerischen Wert 8, so muss man vom Code nur den Code von "0" abziehen.

### 4.5.2 8-Bit Erweiterungen

Der ASCII-Zeichensatz benötigt 7 Bit zur Kodierung, ein Byte als Basiseinheit der Verarbeitung auf einem Computer enthält aber 8 Bit, ein Bit wird somit verschenkt. Wie man an der ASCII-Tabelle 4.11 aber sieht, sind zum Beispiel keine deutschen Umlaute darin kodiert. Dies führte dazu, dass man Erweiterungen des ASCII-Codes über dieses zusätzliche Bit einführt zur Kodierung von landesspezifischen Erweiterungen, in Deutschland etwa äöüß, in skandinavischen Ländern zum Beispiel øåæ. Da aber durch das zusätzliche Bit nur 128 zusätzliche Möglichkeiten offen stehen, der Bedarf aber weitaus höher war, hat u.a. die internationale Standardisierungsorganisation ISO (International Organization for Standardization; <http://www.iso.ch/>) regionsspezifische Kodierungen von 128 zusätzlichen Zeichen in den Kodierungen 128-255 definiert. Bei diesen Erweiterungscode hast also das achte Bit den Wert 1. Die im mitteleuropäischen Raum gebräuchliche Kodierung heißt **Latin-1** und wird in der Norm ISO 8859-1 definiert. Die später definierte Norm ISO 8859-15 ist dazu weitgehend identisch, hat aber als wichtige Ergänzung das Euro-Zeichen zusätzlich als Kodierung enthalten.

Einige Unternehmen setzen diese und teilweise aber auch zusätzlich eigene Kodierungen für die zusätzlichen 128 Möglichkeiten ein. Dazu zählen die verschiedenen Codepages bei Microsoft (z.B. 437, 850, 1252, 10000, 28605), Mac-Roman bei Apple und andere, die bei Nicht-ASCII-Zeichen nicht vollständig kompatibel zueinander sind.

### 4.5.3 Unicode und Universal Character Set

Überlegt man sich aber nun, dass zum Beispiel durch das weltweite Verschicken von Emails Texte, die beispielsweise mit der Kodierung Latin-1 kodiert sind und die landesspezifische Erweiterungen (zum Beispiel das deutsch ä) enthalten, in einem Land gelesen werden, dass eine andere Erweiterungskodierung benutzt, so wird klar, dass dieser Ansatz nach wie vor unbefriedigend ist. Überlegt man sich auch weiterhin Alphabete wie das chinesische oder japanische mit Hunderten oder Tausenden von unterschiedlichen Zeichen, so sieht man die Beschränkungen auch in diesem erweiterten ASCII-Zeichensatz.

Um einen umfassenderen Zeichensatz zu definieren, wurde der **Unicode**-Zeichensatz (<http://www.unicode.org/>) 1991 entworfen, der zum Beispiel der Zeichensatz ist, der in der Programmiersprache Java zur Kodierung von Zeichen verwendet wird. Unicode sah ursprünglich 65.536

verschiedene Zeichen vor, was mit 2 Bytes / 16 Bit für jedes Zeichen kodiert werden kann, statt der 7 Bit im ASCII-Zeichensatz beziehungsweise 8 Bit in den ASCII-Erweiterungen wie Latin-1.

Mit einer 16 Bit Unicode Kodierung können also maximal  $2^{16} = 65536$  Zeichen kodiert werden. Die Kodierung der ersten 128 Zeichen im Unicode-Zeichensatz ist identisch mit der ASCII-Kodierung, die Kodierung 128-255 entspricht ISO Latin-1, so dass hier für viele Anwender kein Unterschied im Kodierungswert besteht (bis auf den Unterschied in der Anzahl der benötigten Bits).

Mit einer späteren Erweiterung auf insgesamt 1.114.112 Zeichen (mindestens 21 Bits werden dafür benötigt) wurden weitere Kodierungsmöglichkeiten geschaffen. Weiter wurde damit einhergehend auch eine Strukturierung in 17 Ebenen, sogenannten *Planes* eingeführt. Eine solche Ebene kodiert  $2^{16} = 65.536$  Zeichen und benötigt dazu dementsprechend 16 Bits. Heute sind insgesamt ca. 150.000 Zeichen in mehreren Ebenen festgelegt. Die Ebene 0 (*Basic Multilingual Plane*) enthält dabei die wichtigsten Zeichen, inklusive den ASCII-Zeichen. Neben diesen festen Kodierungen gibt es in dieser Ebene aber auch einen Bereich mit 6.400 Codes, der als *Private Use Area* bezeichnet wird und z.B. für eigene Kodierungen innerhalb eines Unternehmens genutzt werden kann.

Neben den bekannten Schriftzeichen wie den Zeichen des lateinischen Alphabets sind aber auch eine Vielzahl weiterer Schriftzeichen aufgenommen worden. Beispiele sind hebräische oder arabische Zeichen, aber auch (aus unserer Sicht) exotischere wie die altpersische Keilschrift oder die phönizische Schrift. Daneben werden aber auch eine Vielzahl von Emojis damit kodiert.

Um einen Unicode-Codepunkt zu bezeichnen, wird die Notation genutzt `u+` gefolgt von mindestens vier hexadezimalen Ziffern. Beispiel: `u+0041` für das Zeichen A.

Die Arbeit des Unicode-Konsortiums wurde von der ISO als internationale Standardisierungsinstitution aufgegriffen. Die ISO hat in der Norm ISO/IEC 10646 die zur Zeit aktuelle Unicode-Version als **Universal Character Set** (UCS) übernommen. ISO und das Unicode-Konsortium haben vor, in Zukunft beide Zeichensatzdefinitionen synchronisiert zu halten. UCS unterscheidet auch unterschiedlich umfangreiche Kodierungen. UCS-2 mit einer 16 Bit Kodierung (2 Byte) entspricht dem Unicode mit 2 Byte (Ebene 0). Mit UCS-4 mit einer 31-Bit Kodierung (4 Byte) kann der gesamte Unicode-Umfang kodiert werden.

#### 4.5.4 Komprimierung gemäß UTF-8

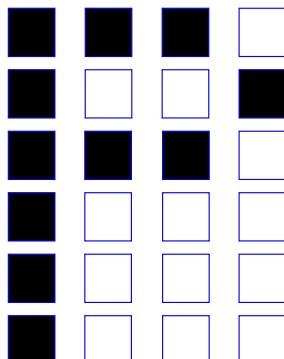
Speichert man Texte, die nur aus Buchstaben und Ziffern bestehen, statt in ASCII-Kodierung in UCS-2- oder UCS-4-Kodierung, so bedeutet dies einen doppelten beziehungsweise vierfachen Speicherbedarf bei gleichem Informationsgehalt. Gleichzeitig wären solche Texte nicht kompatibel mit rein ASCII-kodierten Texten, schon alleine aufgrund der genutzten Anzahl an Bytes pro Zeichen.

In UTF-8 (UCS Transformation Format), ebenfalls von der ISO als Anhang zur UCS-Norm normiert, wird eine kompakte Kodierung definiert (also quasi eine Kodierung/Komprimierung einer Kodierung), die alle UCS-4 Codes kodieren kann und gleichzeitig kompatibel zur ASCII-Kodierung ist. Die Vorteile der UTF-8 Kodierung sind zum einen die Kompatibilität mit existierenden ASCII-kodierten Texten und zum anderen die Reduktion der anfallenden Datenmenge, so dass bei typischen Texten aus der amerikanischen und europäischen Region faktisch ein Byte pro Zeichen benötigt wird und dies zudem der früher üblichen ASCII-Kodierung entspricht. Dazu wird in UTF-8 eine je nach zu kodierendem Zeichen unterschiedlich große Anzahl von Bytes benötigt, zwischen 1 und 4 Bytes.

Es sei erwähnt, dass es neben der UTF-8 Kodierung weiterhin auch UTF-16 (bis 4 Bytes) und UTF-32 (bis 6 Bytes) als weitere Kodierungen gibt.

Die allgemeinen Regeln zur (De-)Kodierung in UTF-8 sind:

- Beginnt das erste Byte mit einem 0-Bit, so stellen die restlichen 7 Bit dieses Bytes den ASCII-Code des Zeichens dar. Jede in reiner ASCII-Kodierung vorliegende Datei ist somit eine korrekt kodierte UTF-8 Datei, was Kompatibilität existierender ASCII-Dateien gewährleistet.

(a) P in einem 6x4 Bitmap  
Font

```
Zeichne_P (Referenzpunkt, Höhe, Breite):
```

Zeichne ausgehend vom Punkt "Referenzpunkt"  
nach oben gerichtet eine Linie der Länge  
"Höhe" und der Dicke "Breite"/20.

Zeichne vom Ende der gezeichneten Linie  
eine Halbellipse mit horizontalem Radius  
"Breite", vertikalem Radius "Höhe"/2  
und Dicke "Breite"/20 nach rechts

(b) P in einem Truetype Font

Figure 4.7: Gegenüberstellung eines Bitmap- und True Type Fonts

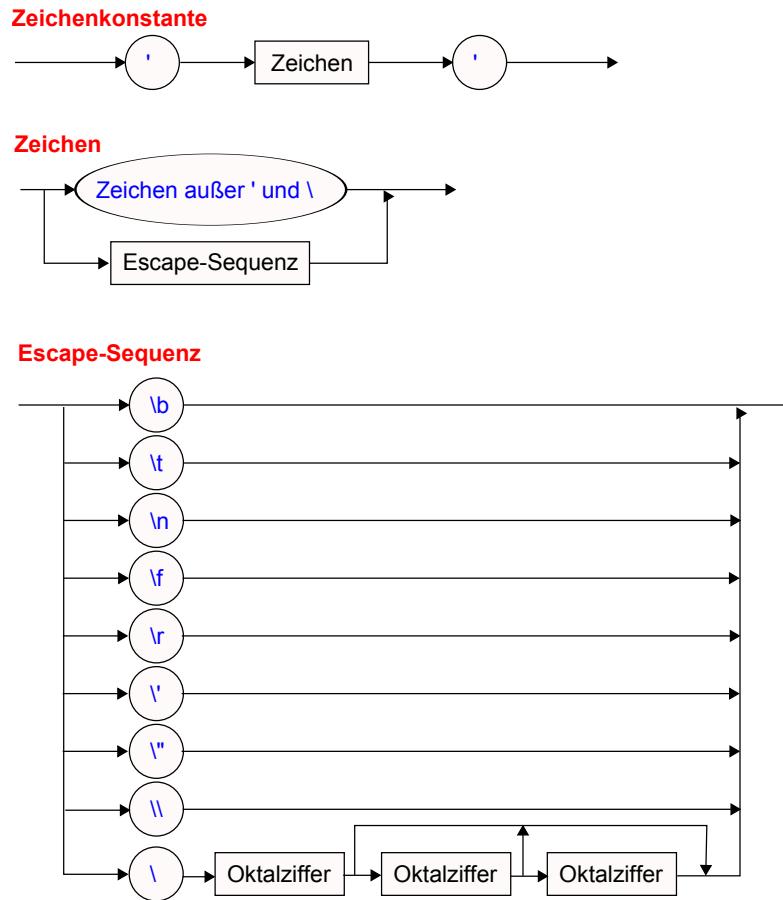
- Beginnt das erste Byte eines Zeichens mit dem Bitwert 1, so erfolgt in diesem und den nachfolgenden Bytes die Kodierung. Im ersten Byte wird kodiert, wieviele Byte  $n$  insgesamt zusätzlich für die Kodierung benötigt werden, indem die ersten  $n + 1$  Bit auf 1 gesetzt sind, das  $n + 2$ -te Bit muss 0 sein. Jedes der nachfolgenden  $n$  Byte enthält dann in den ersten 2 Bit das Bit-Muster 10. Nach diesem Format lassen sich also Folgen aus maximal 8 Byte spezifizieren (erstes Byte und maximal 7 Folgebyte).

Mit diesen beiden allgemeinen Regeln zum Aufbau von UTF-8 Codes geschieht nun die eigentliche Kodierung nach folgendem Schema, wobei die mit **x** markierten Bit-Stellen den eigentlichen Code enthalten, die mit 0 oder 1 explizit angegeben Bit dienen nur dazu, mögliche Folgebytes zu spezifizieren.

Anzahl Byte	Aufbau	Zeichen in xxx...
1	0xxx xxxx	U+0000 - U+007f (ASCII, 7 Bit)
2	110x xxxx 10xx xxxx	U+0080 - U+07ff (max. 11 Bit)
3	1110 xxxx 10xx xxxx 10xx xxxx	U+0800 - U+ffff (max. 16 Bit)
4	1111 0xxx 10xx xxxx 10xx xxxx 10xx xxxx	U+10000 - U+10ffff (max. 21 Bit)

Ein Vorteil der UTF-8-Kodierung ist, dass Texte als Folgen von Byte zum Beispiel über ein Netzwerk verschickt werden können, ohne dass ein Gesamtkontext erkannt werden muss. Kommt es zum Beispiel bei einer Übertragung eines UTF-8 Textes zu kleinen Aussetzern, so dass kleine Teile der Bytefolge verloren gehen, so kann die Empfangsseite trotzdem leicht den Großteil der übertragenen Daten erkennen, weil der Beginn eines jeden Zeichens entweder mit 0 anfängt (ASCII) oder mit 11 (Multibyte-Kodierung mit mindestens 2 Byte), so dass sehr einfach der Beginn eines neuen Zeichens und damit einer Aufsetzpunkt in der Erkennung gefunden werden kann.

Öffnet man zum Beispiel mit einem Editor eine Datei, so ist für den Editor nicht einfach erkennbar, welche Kodierung in dieser Datei überhaupt genutzt wird. Meist werden im Editor Heuristiken zur Erkennung eingesetzt. Um eine Datei als Unicode-basierte UTF-8 kodierte Datei zu kennzeichnen, hat das Unicode-Konsortium eine feste Byte-Folge definiert, die zu Beginn einer Datei auf eine Unicode-Kodierung hinweist. Dieses sogenannte *Byte Order Mark* ist die Byte-Folge **0xef 0xbb 0xbf**. Dieses Byte Order Mark wird von einigen Editoren automatisch beim Speichern in eine UTF-8 kodierte Datei eingefügt. Dies kann aber zu Problemen führen, wie zum Beispiel beim Java-Compiler, wenn dort ASCII erwartet wird.

Figure 4.8: Syntax zu Zeichenkonstanten (Typ `char`) in Java

#### 4.5.5 Darstellung von Zeichen

Zeigt man ein Zeichen auf einem Bildschirm an, zum Beispiel indem man in einem Programm eine entsprechende Ausgabeanweisung ausführt, so wird dem Betriebssystem der entsprechende Code dieses Zeichens übergeben. Das Betriebssystem (oder eventuell auch eine Zwischenschicht) sucht dann anhand des Codes in einer Tabelle, wie das Zeichen dargestellt werden kann. Diese Tabelle nennt man **Font**. In einem Font kann zum Beispiel abgelegt sein, welche Zeichen in einem rechteckigen Raster von  $n \times m$  Rasterpunkten geschwärzt werden sollen und welche weiß bleiben sollen. Solche Fonts (**Bitmap Fonts**) haben den Nachteil, dass sie optimal nur für Rechtecke dieser Größe angewandt werden können. Hat man zum Beispiel Rechtecke der Größe  $8n \times 8m$ , so muss entweder ein anderer Font mit dieser Größe definiert sein oder in der Vergrößerung sind deutliche Vierecke der Rasterung zu sehen, aus denen die Buchstaben zusammengesetzt sind. Ein Ausweg aus dieser Problematik sind die sogenannten **True-Type-Fonts**, die statt der Angabe von Pixeln vielmehr einen größenunabhängigen Algorithmus definieren, wie Buchstaben "gemalt" werden sollen. Abbildung 4.7 stellt die beiden Ansätze gegenüber. Es kann durchaus mehrere / viele Fonts parallel geben, die die Darstellung eines Zeichensatzes in Variationen (fett gedruckt, schräg gestellt, unterstrichen, und so weiter) angeben. Dies wird allerdings nicht direkt in einem Zeichencode kodiert, sondern muss zusätzlich gespeichert werden (zum Beispiel in den Dateiformaten von Textverarbeitungsprogrammen).

Listing 4.10: Beispiel zur Nutzung von Zeichen.

```

4-1 /**
4-2 * einfache Beispiel zur Nutzung von char
4-3 */
4-4 public class CharBeispiel {
4-5     public static void main(String [] args) {
4-6
4-7         // Definition einer Variablen vom Typ char
4-8         char c;
4-9
4-10        // Der Variablen wird der Wert a zugewiesen.
4-11        // Der Java-Compiler erkennt 'a' als Zeichenkonstante
4-12        // und wandelt diese Angabe automatisch in den
4-13        // entsprechenden Unicode um.
4-14        c = 'a';
4-15
4-16        // Ausgabe des Zeichens in der Variablen c.
4-17        // Hier wandelt der Java Compiler/ das Betriebssystem automatisch
4-18        // den Unicode in die Darstellung des entsprechende Zeichens um.
4-19        System.out.println("c: " + c);
4-20    }
4-21}

```

## 4.5.6 Zeichen in Java

In Java ist der Datentyp für ein Zeichen der Typ `char` (Abkürzung für `character`), als Kodierung eines Zeichens wird intern Unicode genutzt. In einer Variablen vom Typ `char` kann genau ein Zeichen in der Unicode-Kodierung (also 16 Bit) gespeichert werden. Konstanten des Datentyps `char` gibt man an, indem man das Zeichen in gewohnter Form (also nicht als Unicode-Zahl) in einfache Apostrophe einschliesst. Das Zeichen "a" würde man also in Java als "a" angeben. Der Java-Compiler wandelt dieses Zeichen dann implizit in den entsprechend Unicode um.

### Beispiel 4.38:

Das Programm in Listing 4.10 zeigt ein erstes Beispiel zur Nutzung des Datentyps `char`. Zu beachten ist, dass weder ein Programmierer noch ein Benutzer des Programms mit dem Unicode direkt in Berührung kommt, also einer bestimmten Zahl, die ein vorliegendes Zeichen darstellt. Als Ausgabe erscheint auf dem Bildschirm:

```
4-1 c: a
```

Abbildung 4.8 gibt das Syntaxdiagramm für eine Zeichenkonstante an. Darin ist auch zu sehen, dass sich alle (Unicode-)Zeichen bis auf ' und \ direkt innerhalb zweier Apostrophe angeben lassen. Das Zeichen \ spielt eine Sonderrolle, indem es sogenannte Escape-Sequenzen (Fluchtsequenzen) startet. Diese Escape-Sequenzen stellen ein einfaches Mittel dar, um einige wenige spezielle Zeichen in Form einer Umschreibung darzustellen. Weiterhin ist auch noch die Angabe von Zeichen in Form Escape-Sequenzen mit Oktalziffern möglich (Code 0-255). Für die Eingabe beliebiger Unicodezeichen existiert eine weitere Möglichkeit, auf die aber hier nicht eingegangen werden soll (siehe [Javc]).

Listing 4.11: Weiteres Beispiel zur Nutzung von Zeichen.

```

4.1 /**
4.2 * einfaches Beispiel zur Nutzung von char
4.3 */
4.4 public class CharBeispiel2 {
4.5     public static void main(String [] args) {

4.6
4.7     // Definition dreier Variablen vom Typ char
4.8     char c1 = 'a';
4.9     char c2 = '\n';
4.10    char c3 = 'b';

4.11
4.12    // Ausgabe der drei Zeichen
4.13    System.out.println(" " + c1 + c2 + c3);
4.14 }
4.15 }
```

Escape-Sequenz	Bedeutung
\b	vorangehendes Ausgabezeichen löschen (Backspace)
\t	horizontaler Tabulator
\n	Zeilenende (Line Feed)
\f	Seitenvorschub auf Drucker (Form Feed)
\r	Zeilenende (Carriage Return)
\"	Anführungszeichen
\'	Apostroph
\\	rückwärtsgerichteter Schrägstrich (Backslash)

**Beispiel 4.39:**

Im Programm in Listing 4.11 werden drei char-Variablen definiert und mit jeweils einem Zeichen initialisiert. Die Variable `c2` enthält anschließend das Zeichen für Zeilenende. Gibt man diese Zeichen hintereinander auf dem Bildschirm aus, so werden a und b auf zwei Ausgabezeilen dargestellt, weil die Ausgabe des mittleren Zeichens einen Seitenvorschub bewirkte. Als Ausgabe erscheint auf dem Bildschirm:

```

4.1 a
4.2 b
```

Weshalb im Programm die Ausgabe mit einem ""+ beginnt wird im Zusammenhang mit Operatorenpriorität erst später klar (Kapitel 7.4) und soll hier noch nicht behandelt werden. ♦

Wie oben bereits angesprochen sind die Zeichensätze für Rechner (egal in welcher Kodierung) so konzipiert, dass gewisse Eigenschaften gelten und diese auch ausgenutzt werden können. Ein Beispiel dafür ist die Umwandlung eines Zifferzeichens wie etwa "8" in den numerischen Wert 8 des Datentyps `int`. In Kapitel 5.4 ist diese Beispielanwendung in einem anderen Zusammenhang nochmals aufgegriffen und es ist dort eine Lösung präsentiert.

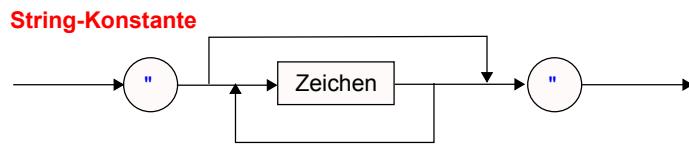
Etwas verwundern mag die Tatsache, dass der Datentyp `char` den numerischen Datentypen in Java zugeordnet wird (siehe Abbildung 4.1). Dies hängt damit zusammen, dass die interne Darstellung eines Zeichens als normale Zahl in Zweierkomplementdarstellung genommen wird. Weiterhin ist es auch zulässig, einen Zeichenwert über die Addition eines Zahlenwertes zu erhöhen, also einen nachfolgendes Zeichen damit angeben.

Listing 4.12: Beispiel zur Manipulation von Zeichencodes.

```

4-1 /**
4-2 * Umwandlungen von bestimmten Zeichenbereichen
4-3 */
4-4 public class Zeichenumwandlung {
4-5
4-6     public static void main(String [] args) {
4-7         char ziffer = '6';
4-8         int zifferwert = ziffer - '0';           // liefert den int-Wert 6
4-9         ziffer = (char)(zifferwert + '0');      // und wieder zurueck
4-10        System.out.println("Ziffernwert:" + zifferwert);
4-11        System.out.println("Ziffer zurueck:" + ziffer);
4-12
4-13        char gross = 'X';
4-14        char klein = (char)(gross + 0x20);       // aus gross mach klein
4-15        System.out.println("Kleinbuchstabe: " + klein);
4-16
4-17        klein = 'g';
4-18        gross = (char)(klein - 0x20);          // aus klein mach gross
4-19        System.out.println("Grossbuchstabe: " + gross);
4-20    }
4-21}

```

Figure 4.9: Syntax zu Zeichenketten (Typ **String**) in Java**Beispiel 4.40:**

```

4-1 char a = 'x';
4-2 char b = (char)(a + 1);
4-3 // b enthaelt das Nachfolzezeichen von 'x', also 'y'

```

Die Operation für den Nachfolgebuchstaben sieht etwas komplizierter aus als einfach nur `+1`. Auf die Notwendigkeit dazu und die Details wird in Kapitel 7.6 eingegangen. ♦

Durch Ausnutzung der Verhältniseigenschaften der Zeichen untereinander lassen sich einige Operationen leicht durchführen. So etwa die Umwandlung einer Ziffer in die entsprechende Zahl, eine Zahl zwischen 0 und 9 in die entsprechende Ziffer, die Umwandlung eines Groß- in einen Kleinbuchstaben und umgekehrt (siehe Listing 4.12).

## 4.6 Zeichenketten

Meist betrachtet man in einem Programm aber nicht nur genau ein Zeichen, sondern eine ganze Zeichenfolge. In Java und in anderen Programmiersprachen wird eine solche Zeichenfolge als **String** (oder **Zeichenkette**) bezeichnet. Eine konstante Folge von Zeichen wird in den meisten Programmiersprachen in Anführungszeichen eingeschlossen. Der String "Name" besteht demzufolge aus den vier Zeichen N", a", m" und e". Die Syntax für Zeichenfolgen in Java ist in Abbildung 4.9 gegeben.

Listing 4.13: Beispiel zur String-Verarbeitung.

```

4.1 /**
4.2 * einfache Beispiel zur Nutzung von String
4.3 */
4.4 public class StringBeispiel {
4.5     public static void main(String[] args) {
4.6
4.7         // Definition einer Variablen vom Typ String
4.8         String s;
4.9
4.10        // Der Variablen wird der Wert leere String zugewiesen.
4.11        s = "";
4.12        System.out.println("Inhalt s an Stelle 1: " + s);
4.13
4.14        // s erhaelt einen neuen String, der sich aus dem alten
4.15        // (leeren) String und "abc" zusammensetzt
4.16        s = s + "abc";
4.17        System.out.println("Inhalt s an Stelle 2: " + s);
4.18
4.19        // s erhaelt einen neuen String, der sich aus dem alten
4.20        // String "abc" und dem String "4711" zusammensetzt. Der
4.21        // String "4711" wurde automatisch aus dem int-Wert 4711 erzeugt
4.22        s = s + 4711;
4.23        System.out.println("Inhalt s an Stelle 3: " + s);
4.24    }
4.25 }
```

An dieser Stelle sei explizit auf den Unterschied zwischen dem Ausdruck A" und "a" hingewiesen. A", das heißt der Buchstabe A eingeschlossen in einfache Apostrophe, bezeichnet das einzelne Zeichen a und der Ausdruck ist vom Java-Typ `char`. Demgegenüber bezeichnet "a" den String, dessen einziges Zeichen der Buchstabe a ist. Der Java-Typ dieses Ausdrucks ist `String` und eben nicht `char`. Es ist auch nicht möglich zum Beispiel einer Variablen vom Typ `char` einen String zuzuweisen oder umgekehrt einer Variablen vom Typ `String` ein Zeichen in der Form a" zuzuweisen. Strings in Java sind keine Basistypen mehr, die in diesem Kapitel behandelt werden. Wie bereits bekannt, ist eine Programmiersprache sehr präzise und verlangt aber auch Gleches von einem Programmierer!

An dieser Stelle soll aber auch schon auf einige Operationen mit Strings eingegangen werden, auch wenn der Typ `String` kein Basistyp ist (was an einigen Stellen wichtige Konsequenzen hat, die später im Detail erläutert werden). Eine einfache String-Operation ist bereits bekannt: die Zusammenführung von zwei Strings zu einem zusammengesetzten String durch den Operator `+`. Diese Operation ist sogar allgemeiner anwendbar: wenn einer der beiden Operanden von `+` ein String ist, so wird der andere (erste/linke oder zweite/rechte) Operand automatisch in einem String umgewandelt und die beiden String dann zu einem neuen String zusammengeführt. Wichtig ist auch hier schon anzumerken, dass in Java dadurch weder der erste noch der zweite Operandenstring verändert wird; es entsteht ein komplett neuer String, dessen Inhalt (die Zeichen) insgesamt dem Inhalt aus dem ersten und zweiten Operandenstring entsprechen.

#### Beispiel 4.41:

Als Ausgabe des Programms in Listing 4.13 erscheint auf dem Bildschirm:

```

4.1 Inhalt s an Stelle 1:
4.2 Inhalt s an Stelle 2: abc
4.3 Inhalt s an Stelle 3: abc4711
```

Methodenname	Beispiel	Wert	Wirkung
<code>charAt()</code>	<code>s.charAt(0)</code>	a"	Zeichen an der i-ten Position
<code>indexOf()</code>	<code>s.indexOf('a')</code>	0	erstes Vorkommen eines Zeichens
<code>length()</code>	<code>s.length()</code>	3	Anzahl Zeichen im String
<code>substring()</code>	<code>s.substring(0,1)</code>	"ab"	Teilstring von-bis

Table 4.12: Ausgesuchte Java-Operationen auf Strings (Beispielstring `s="abc"`)

Listing 4.14: Beispiel zu String-Operationen.

```

4-1 /**
4-2 * einfache Beispiele zur Nutzung von String -Methoden
4-3 */
4-4 public class StringBeispiel2 {
4-5     public static void main(String[] args) {
4-6
4-7         // Definition einer Variablen vom Typ String
4-8         String s = "abc";
4-9
4-10        // hole char an i-ter Position
4-11        System.out.println("an Position 1: " + s.charAt(1));
4-12
4-13        // berechne Index eines Zeichens
4-14        System.out.println("b an Position : " + s.indexOf('b'));
4-15
4-16        // berechne Laenge des Strings
4-17        System.out.println("Laenge = " + s.length());
4-18
4-19        // Teilstring ohne 1. und letzten Buchstaben
4-20        // wir gehen davon aus, dass es diesen Teilstring auch wirklich gibt
4-21        System.out.println("Teilstring 1-2 ist " + s.substring(1,2));
4-22    }
4-23}

```

Stringverarbeitung wird oft in Anwendungen genutzt, um beispielsweise Ein- und Ausgabewerte zu verarbeiten, zwischen zwei Programmen über das Netzwerk zu kommunizieren, Webseiten abzurufen und vieles mehr. Insofern sind vielfältige Operationen auf Strings nötig, von denen Java eine Reihe in Form von Funktionen / Methoden in der Klasse `String` anbietet (woran Sie schon einen der Unterschiede zu den Basistypen sehen können). Die Nutzung dieser Operationen ist so, dass zu einem String `s` eine solche Operation aufgerufen werden kann.

#### Beispiel 4.42:

Das Programm in Listing 4.14 zeigt die Anwendung einiger der Stringoperationen. ♦

## 4.7 Einfache Ein- und Ausgabe von Daten

Dieses Unterkapitel nutzt einige Techniken, die bis jetzt noch nicht behandelt wurden und auch auf den nächsten Seiten noch nicht im Detail behandelt werden. Da erfahrungsgemäß aber sehr schnell die Frage danach aufkommt, wie man in einem Programm Daten ein- und ausgeben kann, soll diese Thematik schon an

dieser frühen Stelle pragmatisch und übersichtsartig behandelt werden, ohne alle diese Techniken im Detail zu erläutern sondern nur vorzustellen. Der Leser kann dieses Unterkapitel aber auch, wenn der Wissenshunger nach der Ein-/Ausgabe noch nicht zu stark ist, vorerst überspringen und später (frühestens nach Kapitel 8) diese Thematik wieder aufgreifen.

Bis jetzt wurden in den Beispielprogrammen Werte, mit denen gerechnet werden sollte (Eingabewerte des Algorithmus), zu Beginn des Programms entsprechenden Variablen explizit im Programm zugewiesen. Dieser Ansatz hat den Nachteil, dass man bei jeder Änderung eines Eingabewertes das Programm neu übersetzen muss. Dies ist für ein Programm absolut unüblich, denn man möchte einen Algorithmus / ein Programm so entwickeln, dass er möglichst allgemein ist und für alle erlaubten Eingabewerte eine sinnvolle Berechnung durchführt. Die Eingabewerte möchte man erst beim Start des Programms oder sogar erst zur Laufzeit des Programms angeben. Ein Beispiel ist ein Antiblockiersystem in einem Auto (ein Programm steuert das ABS), das permanent Messdaten wie etwa zur negativen Beschleunigung erhält und darauf entsprechend reagieren muss.

Aus diesem Grund gibt es flexiblere Möglichkeiten, um dem Programm Daten mitzuteilen. An dieser Stelle sollen nur pragmatisch einige einfache Techniken vorgestellt werden, die für die Problemstellungen in diesem Buch aber vollkommen ausreichen. Das Thema Ein-/Ausgabe ist insgesamt zwar sehr gut strukturiert, aber durchaus auch sehr komplex in den Details. Für einen tieferen Einstieg in die Thematik der Java Streams (der allgemeine Ansatz für die Ein-/Ausgabe in Java) sei zum Beispiel auf das entsprechende Kapitel im Java Tutorial verwiesen [Javd].

### 4.7.1 Daten beim Programmstart übergeben

Beim Programmstart können Daten angegeben werden, die während des Programmlaufs dann in das Java-Programm übernommen und verwendet werden können.

In eclipse müssen die Eingabewerte angegeben werden unter dem Menüpunkt "Run Configurations -> (x)=Arguments -> Program Arguments" und nach Eingabe der Werte anschließend der Punkt "Run" zum Start des Programms angeklickt werden. Die Eingabewerte werden durch Leerzeichen voneinander getrennt.

Die eingegebenen Werte kommen im Programm als "durchnummerierte" Strings an, die in den richtigen Datentyp umgewandelt werden müssen. Man muss im Programm also wissen, was man (im Sinne von Datentyp) als i-ten Eingabewert einlesen möchte. Zu beachten ist allerdings, dass auch ein zu dem gewünschten Datentyp korrekt geformelter Eingabewert vorliegen muss. Die Eingabestrings sind "durchnummeriert" mit `args[0]` bis `args[n-1]` bei n Eingabewerten.

Zu jedem primitiven Datentyp gibt es eine spezielle Methode zur Umwandlung eines beliebigen aber hinsichtlich des Datentyps korrekt geformten Strings in einen entsprechenden Wert von diesem Wunschdatentyp. Die nachfolgende Tabelle gibt für die primitiven Datentypen die entsprechenden Methoden in Form eines Beispiels an:

primitiver Datentyp	Beispiel
byte	<code>Byte.parseByte("23")</code>
short	<code>Short.parseShort ("23")</code>
int	<code>Integer.parseInt ("23")</code>
long	<code>Long.parseLong ("23")</code>
float	<code>Float.parseFloat ("23")</code>
double	<code>Double.parseDouble (23)</code>

Der Datentyp `char` sowie Strings können einfacher behandelt werden. Die praktische Nutzung ist im Beispiel in Listing 4.15 angegeben.

Listing 4.15: Einlesen von Werten aus der Kommandozeile.

```

4-1  /**
4-2   * Argumente unterschiedlichen Typs aus der Kommandozeile zu lesen
4-3   * Aufruf z.B. mit:
4-4   *      java Kommandozeilenargumente 1 2 3 4 5.7  3.1 x  str
4-5   * wobei die Eingabe in dieser Reihenfolge getaetigt werden muss:
4-6   *      byte, short, int, long, float, double, char, String
4-7   * Fehlerbehandlung wird nicht vorgenommen!
4-8   */
4-9
4-10 public class Kommandozeilenargumente {
4-11
4-12     public static void main(String[] args) {
4-13
4-14         // 1. Argument muss eine ganze Zahl im Wertebereich byte sein
4-15         byte b = Byte.parseByte(args[0]);
4-16
4-17         // 2. Argument muss eine ganze Zahl im Wertebereich short sein
4-18         short s = Short.parseShort(args[1]);
4-19
4-20         // 3. Argument muss eine ganze Zahl im Wertebereich int sein
4-21         int i = Integer.parseInt(args[2]);
4-22
4-23         // 4. Argument muss eine ganze Zahl im Wertebereich long sein
4-24         long l = Long.parseLong(args[3]);
4-25
4-26         // 5. Argument muss eine Fliesskommazahl im Wertebereich float sein
4-27         float f = Float.parseFloat(args[4]);
4-28
4-29         // 6. Argument muss eine Fliesskommazahl im Wertebereich double sein
4-30         double d = Double.parseDouble(args[5]);
4-31
4-32         // 7. Argument muss ein Character sein
4-33         char c = args[6].charAt(0);
4-34
4-35         // 8. Argument muss ein String sein
4-36         String str = args[7];
4-37     }
4-38 }
```

Listing 4.16: Einlesen von Werten über die Tastatur.

```

4-1 /**
4-2 * Beispiel zum Einlesen von Daten von der Tastatur
4-3 */
4-4 import java.util.*;
4-5
4-6 public class EingabeTastatur {
4-7
4-8     public static void main(String[] args) {
4-9
4-10         // Scanner von der Tastatur anlegen
4-11         Scanner sc = new Scanner(System.in);
4-12
4-13         System.out.println("Geben Sie nacheinander Daten folgender Typen "
4-14             + "auf der Tastatur ein: "
4-15             + "byte, short, int, long, float, double, "
4-16             + "ein Zeichen (in einer eigene Zeile), "
4-17             + "eine Zeichenkette in einer eigenen Zeile");
4-18
4-19         // Daten ueber den Scanner lesen
4-20         byte b = sc.nextByte();
4-21         short s = sc.nextShort();
4-22         int i = sc.nextInt();
4-23         long l = sc.nextLong();
4-24         float f = sc.nextFloat();
4-25         double d = sc.nextDouble();
4-26         String s1 = sc.next();
4-27         char c = s1.charAt(0);
4-28         String s2 = sc.next();
4-29
4-30         // eingelesene Daten ausgeben
4-31         System.out.println("Die eingelesenen Werte sind: " + b + " "
4-32             + s + " " + i + " " + l + " " + f + " " + d
4-33             + " " + c + " " + s2);
4-34
4-35         // Scanner abschliessen
4-36         sc.close();
4-37     }
4-38 }
```

Listing 4.17: Einlesen von Werten aus einer Datei.

```

4.1  /* Beispiel zum Einlesen von numerischen Daten aus einer Datei
4.2   */
4.3  import java.util.*;
4.4  import java.io.*;
4.5
4.6  public class EingabeDatei {
4.7
4.8      public static void main(String [] args) {
4.9
4.10         Scanner sc = null;
4.11
4.12         // Scanner von der Datei "test.txt" anlegen
4.13         try {
4.14             sc = new Scanner(new File("test.txt"));
4.15         } catch(FileNotFoundException e) {
4.16             System.out.println("Datei nicht vorhanden");
4.17             return;
4.18         }
4.19
4.20         System.out.println("In der Datei test.txt muessen nacheinander "
4.21                         + "Daten folgender Typen stehen: "
4.22                         + "byte, short, int, long, float, double");
4.23
4.24         // Daten vom Scanner lesen
4.25         byte b = sc.nextByte();
4.26         short s = sc.nextShort();
4.27         int i = sc.nextInt();
4.28         long l = sc.nextLong();
4.29         float f = sc.nextFloat();
4.30         double d = sc.nextDouble();
4.31
4.32         // eingelesene Daten ausgeben
4.33         System.out.println("Die eingelesenen Werte sind: " + b + " "
4.34                         + s + " " + i + " " + l + " " + f + " " + d);
4.35
4.36         // Scanner abschliessen
4.37         sc.close();
4.38     }
4.39 }
```

## 4.7.2 Daten während des Programms lesen

Noch flexibler wird man, wenn man Eingabedaten erst zur Laufzeit des Programms angeben kann (und nicht schon beim Start). Auch ohne hier wieder auf die exakten Details einzugehen wird das Beispielprogramm in Listing 4.16 angegeben, das für die primitiven Datentypen in Java zeigt, wie man zur Laufzeit des Programms von der Tastatur entsprechende Werte einlesen kann. Zu beachten ist hierbei, dass der Aufruf einer Methode wie etwa **sc.nextByte()** *blockierend* ist, was bedeutet, dass die Programmausführung an der Stelle anhält, bis eine Eingabe auf der Tastatur getätigkt wurde und die Taste **Return** ( $\leftarrow$ ) getätigkt wurde. Deshalb ist es oft für den Nutzer eines Programms hilfreich, wenn vorher eine Ausgabe auf dem Bildschirm erscheint, dass nun eine Eingabe zu erfolgen hat.

### 4.7.3 Daten aus einer Datei lesen

Will man sehr viele Daten verarbeiten so ist die Eingabe über eine Tastatur sehr mühsam wenn nicht sogar aufgrund der Menge unmöglich (Versuchen Sie einmal, 1 Terabyte Daten über die Tastatur einzugeben). Neben anderen Möglichkeiten wie etwa ein Datenstrom über ein Netzwerk ist eine weit verbreitete Art der Dateneingabe eine Datei. Dazu gibt man zu Beginn des Programms den Namen der Datei an und geht anschließend analog zum Einlesen von der Tastatur vor (siehe Listing 4.17).

### 4.7.4 Formatierte Ausgabe

Bis jetzt wurde die Möglichkeit der Bildschirmausgabe mit Hilfe von `System.out.println()` vorgestellt. Dabei werden die auszugebenden Werte in einem Standardformat dargestellt. Will man aber etwa Tabellen ausgeben oder Geldwerte mit genau zwei Nachkommastellen, so ist eine Beeinflussung dieser Ausgabe erforderlich. Java kennt unter anderem dazu die Methode `System.out.printf()`, wie sie in ähnlicher Form auch in anderen Programmiersprachen existiert und hier nur einführend mit ihren wesentlichen Möglichkeiten gezeigt wird. Die Wirkung dieser Methode unterscheidet sich dahingehend von `System.out.println()`, dass nun ein (Format-)String als erstes Argument angegeben wird, gefolgt von weiteren Argumenten, jeweils durch Komma getrennt. Die Anzahl der weiteren Argumente richtet sich nach dem Inhalt des Formatstrings! Ein Formatstring wird bei der Ausführung der Methode von links nach rechts abgearbeitet. Zeichen für Zeichen wird auf dem Bildschirm ausgegeben, wie bekannt. Der Unterschied ist allerdings, wenn ein Prozentzeichen im Formatstring erscheint. Dann wird dies als Beginn einer Formatangabe interpretiert, zu der das nächste noch nicht verarbeitete Argument als auszugebender Wert genommen / verarbeitet wird. Hinter dem Prozentzeichen wird ein Buchstabe angegeben, der die Art der Ausgabe angibt, und wozu das Argument typmäßig passen muss. Weiterhin kann dazwischen eine Zahl angeben werden, die die Weite in Zeichen der Ausgabe angibt. Bei Fließkommaangaben kann weiterhin noch die Anzahl an Nachkommastellen spezifiziert werden.

Als Formatbezeichner sind möglich (keine vollständige Aufzählung hier):

Formatbezeichner	Anwendung auf Argumente vom Typ
b	<code>boolean</code>
d	<code>int</code> (dezimale Ausgabe)
x	<code>int</code> (hexadezimale Ausgabe)
f	<code>double</code>
c	<code>char</code>
s	<code>String</code>

Listing 4.18 zeigt einige Beispiele, die entsprechende Ausgabe dazu ist:

```

4.1 abc123
4.2 abc12
4.3 abc 12
4.4 abc12,34
4.5 abcx12hallo

```

## 4.8 Zusammenfassung und Hinweise

### Literaturhinweise

Die Primärreferenz zu den primitiven Datentypen in Java ist [Javc]. In jedem guten Java-Lehrbuch werden diese Typen ebenfalls vorgestellt und anhand von einfachen Beispiel diskutiert.

Listing 4.18: Beispiele zur formatierten Ausgabe.

```

4.1 /**
4.2 * Beispiel zur formatierten Ausgabe
4.3 */
4.4 public class AusgabeFormatiert {
4.5
4.6     public static void main(String [] args) {
4.7         // keine Prozentzeichen
4.8         System.out.printf("abc123\n");
4.9         // ein int
4.10        System.out.printf("abc%d\n", 12);
4.11        // ein int mit 3 Zeichen Raum
4.12        System.out.printf("abc%3d\n", 12);
4.13        // ein float mit 5 Zeichen gesamt und 2 Nachkommastellen
4.14        System.out.printf("abc%5.2f\n", 12.34);
4.15        // ein char, int, String
4.16        System.out.printf("abc%c%d%s\n", 'x', 12, "hallo");
4.17    }
4.18 }
```

## Verstehen

Jede Information muss in einem (Digital-)Rechner in einer Folge von Bits/Bytes dargestellt werden. Die konkrete Darstellung von Werten der primitiven Typen ist verstanden. Diese Darstellung wird durch die interne Darstellung in Rechnern bereits vorgegeben. Aus der Art der rechnerinternen Darstellungen ergeben sich wichtige Konsequenzen.

## Kurz und knapp merken

- Kommentare helfen mit umgangssprachlichen Formulierungen einem Leser dabei, das Programm besser und schneller zu verstehen.
- Bezeichnern dienen dazu, verschiedenen Dingen in einem Programm einen Namen zu geben, um so darauf Bezug nehmen zu können.
- Ein Ausdruck beschreibt in formelmäßiger Form, wie genau ein Wert berechnet kann.
- Eine Anweisung ist der Grundbaustein, mit den in vielen Programmiersprachen der Ablauf eines Programms beschrieben werden kann.
- Jede Variable muss in Java deklariert werden.
- Variablen können während der Laufzeit eines Programms ihren Wert ändern (Zuweisung).
- Für logische Werte stehen eine kleine Zahl an Basisoperationen zur Verfügung (Negation, Und, Oder, Exlusives Oder).
- Ganzzahligen Werte aller ganzzahligen Typen werden intern in Zweierkomplementdarstellung dargestellt.
- Wenn keine Gründe dagegen sprechen, nutzt man `int` als ganzzahligen Typ.
- Innerhalb des entsprechenden Wertebereichs können ganze Zahlen exakt dargestellt werden und auch die Arithmetik ist exakt, wenn das Ergebnis innerhalb des Wertebereichs liegt. An Operationen stehen die arithmetischen Grundrechenarten sowie Bitoperationen zur Verfügung.
- Die ganzzahlige Division, Modulobildung und Bitoperationen sind wichtige Operationen der Informatik.
- Fließkommawerte werden intern in der IEEE754-Darstellung repräsentiert.
- Fließkommazahlen sind eventuell nur eine Annäherung an den exakten Wert. Bei der Arithmetik mit Fließkommazahlen ist große Sorgfalt geboten (Addition von großenmäßig sehr unterschiedlichen Werten, Subtraktion ähnlicher Werte, Division durch sehr kleine Zahlen, Vergleich auf 0).

- An Fließkommaoperationen stehen neben den Grundrechenarten viele weitere Operationen (ähnlich den Funktionen eines wissenschaftlichen Taschenrechners) in Form von Funktionen `Math.f()` zur Verfügung.
- Zeichen / Buchstaben werden in Java durch Unicode kodiert. Neben Buchstaben, Ziffern und Interpunktuation gibt es auch Kontrollcodes für zum Beispiel ein Zeilenende.
- Zeichencodes haben bestimmte Eigenschaften, die sich ausnutzen lassen.
- Ganzzahlige Werte und Operationen darauf können transparent genutzt werden, um mit Zahlen, Bits oder Zeichencodes zu rechnen.

## Häufige Fehler

- Eine ganzzahlige Division unterscheidet sich von einer Fließkommadivision.
- Man sollte immer genau wissen, ob man im ganzzahligen Bereich oder im Fließkommabereich arbeitet.
- Beim Umgang mit Fließkommawerten fallen eventuell keine exakten Resultate an, inklusive bereits der rechnerinternen Darstellung von Dezimalkonstanten wie 0,1.
- Der Test auf Gleichheit bei (insbesondere berechneten) Fließkommawerten ist problematisch.
- Häufig werden der Datentyp `char` und `string` durcheinander geworfen. Eine Konstante vom Typ `char` wird in Java zwischen zwei einfache Apostrophe eingeschlossen und enthält genau ein Zeichen (Beispiel: "a"). Eine String-Konstante wird in Anführungszeichen eingeschlossen und kann 0, 1, 2,... Zeichen enthalten (Beispiel: "abc"). "a" ist etwas anderes als "a"!.
- Strings mit `==` auf gleichen Inhalt vergleichen.

## Übungsfragen

- Wie werden in Java Wahrheitswerte intern dargestellt?
- Wo ist der Unterschied zwischen einfachem und doppeltem Und/Oder?
- Geben Sie eine logischen Ausdruck in Java an für (x,y,z sind ganzzahlige Variablen): x,y und z sind negativ
- Geben Sie eine logischen Ausdruck in Java an für (x,y,z sind ganzzahlige Variablen): x ist größer als y, aber nicht kleiner als z
- Geben Sie eine logischen Ausdruck in Java an für (x,y,z sind ganzzahlige Variablen): weder x noch y sind größer als z
- Geben Sie eine logischen Ausdruck in Java an für (x,y,z sind ganzzahlige Variablen): das 7. Bit von x ist 0
- Geben Sie eine logischen Ausdruck in Java an für (x,y,z sind ganzzahlige Variablen): x und y unterscheiden sich nicht in den untersten 3 Bit
- Geben Sie eine logischen Ausdruck in Java an für (x,y,z sind ganzzahlige Variablen): x,y und z sind alle negativ
- Geben Sie eine logischen Ausdruck in Java an für (x,y,z sind ganzzahlige Variablen): x ist gerade (exklusiv) oder ist doppelt so groß wie y
- Vereinfachen Sie: `!(x < y) & (x == y)`
- Vereinfachen Sie: `!(x > y) & (x == y)`
- Vereinfachen Sie: `!(x > y) & !(x == y)`
- Vereinfachen Sie: `!(x < y) & !(x == y)`
- Wieso gibt es überhaupt verschiedene Datentypen für Zahlen?
- Stellen Sie folgende Werte in Java dar: 7 in binär, 23 in oktal, 23 in Hexadezimal

- Stellen Sie folgende Werte in Java dar: 5 in binär, 17 in oktal, 27 in Hexadezimal
- Was ist die Zweierkomplementdarstellung des Wertes -128 im Datentyp byte?
- Was ist die Zweierkomplementdarstellung des Wertes -127 im Datentyp byte?
- Welche Möglichkeiten gibt es, eine Division von int x durch 2 zu notieren?
- Wie werden ganze Zahlen intern dargestellt?
- Welche Bedeutung hat das oberste Bit bei der Zweierkomplementdarstellung?
- Welche Java-Typen gibt es für ganzzahlige Werte und was ist ihr Wertebereich?
- Welche Darstellungsmöglichkeiten gibt es für ganzzahlige Konstanten?
- Welche Operationen gibt es auf ganzzahligen Werten?
- Welche Bitoperationen gibt es?
- Was ist eine Bit-Maske?
- Was zeichnet Operationen mit ganzzahligen Werten aus (im Gegensatz zu Fließkommawerten)?
- Was bedeutet ein Überlauf bei ganzzahligen Werten?
- Aus welchen Teilen besteht eine Fließkommazahl?
- Wie ist konkret eine 32 Bit IEEE754 Zahl aufgebaut?
- Beschreiben Sie das Verfahren, wie man aus einer Festkommadezimalzahl eine IEEE-Zahl macht!
- Welche Probleme können beim Rechnen mit Fließkommazahlen auftreten?
- Welche Java-Typen gibt es für reelle Zahlen?
- Wie ist die Notation für Fließkommakonstanten?
- Welche Operationen gibt es auf Fließkommawerten?
- Auf was muss man achten, wenn man die Sinus-Funktion in Java nutzt?
- Welche Eigenschaften hat der ASCII-Code?
- Was ist UTF-8? Wie ist der Zusammenhang mit ASCII, Unicode?
- Beschreiben Sie das Verfahren, nach dem Zeichencodes in UTF-8 kodiert werden!
- Wie werden Zeichen intern in Java dargestellt?
- Wie kann ich Zeichenkonstanten in Java angeben?
- Welche Escape-Sequenzen kennen Sie für Java Zeichenkonstanten?
- Wie kann man in Java mit Zeichencodes rechnen?
- Wie kann ich eine String-Konstante angeben?
- Welche Operationen gibt es auf Strings?
- Wie kann ich einen String(-Objekt) so verändern, dass eine 1 hinten angefügt wird (Kein neuer String darf entstehen)?
- Wie kann ich zwei Strings inhaltlich vergleichen?
- Wie kann ich testen, ob zwei Strings nicht den identischen Inhalt haben?
- Wie kann ich einen ganzzahligen Wert aus der Kommandozeile lesen?
- Wie kann ich einen ganzzahligen Wert zur Laufzeit von der Tastatur lesen?

## Reflektion des Stoffs

- Wenn man statt Bits mit zwei Zuständen stattdessen "Trips" mit drei möglichen Zuständen hätte: wieviele verschiedene Werte lassen sich dann mit n "Trips" darstellen?
- Überlegen Sie sich aus Ihrem privaten Umfeld, welche Informationen sich nicht direkt / als Ganzes mit den Kategorien der primitiven Typen in Java darstellen lassen.
- Was würde sich an welcher Stelle im Prozess der Programmerstellung für wen ändern (Programmier, Compiler, Ausführung auf einem Rechner), wenn man eine Variable nicht deklarieren müsste?
- Was sollte/könnte passieren, wenn man einer Variablen einen Wert zuweist, der nicht dem Typ der Variablen entspricht? Diskutieren Sie mögliche Fälle.

- Formulieren Sie allgemein die Bedingung, dass ein Wert  $x$  weder 11 noch 13 ist. Formulieren sind anschließend dies in Java.
- Wieviele verschiedene Werte lassen sich mit 3 Oktalziffern darstellen? Mit 2 Hexadezimalziffern?
- Wie könnte man zu einer beliebigen Zahl statt der Quersumme der Dezimalziffern die Quersumme der Binärziffern ermitteln?
- Kann aus zwei positiven Zahlen und einer Operation auf ihnen durch Bereichsüberlauf eine negative Zahl entstehen? Zwei negative Zahlen? Eine negative, eine positive? Beispiele?
- Welche Toleranzangabe würde bei dem Test `Math.abs(x-y) < toleranz` prinzipiell Sinn machen, welche nicht (wieso)? 1E10, 1E1, 1E-1, 1E-5, 1E-10, 1E-15, 1E-20, 1E-25, 1E-30, 1E-100, 1E-1000
- Würde ein Ausdruck wie '`A`'+0.5 auch Sinn machen? Wieso / wieso nicht?
- Strings sind keine primitiven Werte mehr. Auf ihnen sind eine Vielzahl von Operationen definiert.
- Welche Vor-/Nachteile könnte es haben, wenn beim + Operator ein neuer String entsteht? Alternativ könnte man ja auch das erste Argument verändern?

# Chapter 5

## Kontrollstrukturen

Kontrollstrukturen sind die Elemente einer Programmiersprache, die den Ablauf eines Programms beeinflussen. Dazu zählen etwa Konstrukte für das Hintereinanderausführen von Einzelaktionen, die Wiederholung von bestimmten Aktionen oder die Ausführung von Einzelaktionen nur, wenn eine bestimmte Bedingung zutrifft. Viele Programmiersprachen besitzen Kontrollstrukturen, die den allgemeinen algorithmischen Grundbausteinen aus Kapitel 2.2 entsprechen. Die Syntax mag manchmal verschieden sein, die Konzepte dahinter sind jedoch in den Programmiersprachen gleich.

### 5.1 Ablaufkontrolle über Anweisungen

Kontrollstrukturen gehören zu den Anweisungen, die bereits in Kapitel 3.2 im Zusammenhang mit Grammatiken behandelt wurden. Abbildung 5.1 zeigt das Syntaxdiagramm zu Anweisungen in Java, die in diesem Buch vorgestellt werden. Java kennt einige weitere Anweisungsarten (Beispiel try-catch-Block), und auch bezüglich Details sind bei einigen Konstrukten weitere Angaben möglich, als dies hier vorgestellt wird. Auf diese weiterführenden Aspekte wird in dieser Einführung nicht detailliert eingegangen und vielmehr auf entsprechende umfassene Referenzwerke beziehungsweise auf weiterführende Literatur verwiesen.

Die einzelnen Möglichkeiten einer Anweisung werden nachfolgend im Einzelnen beschrieben. Dabei ist das Vorgehen so, dass zuerst einfache grundlegende Anweisungen vorgestellt werden, die allerdings teilweise und auf den ersten Blick etwas ungewöhnlich erscheinen mögen (Beispiel leere Anweisung). Aufbauend auf diesen Grundbausteinen von Anweisungen werden anschließend dann komplexere Kontrollstrukturen vorgestellt.

### 5.2 Ausdrucksanweisung

Zu einem beliebigem Ausdruck  $e$  erhält man durch Anhängen eines Semikolons eine **Ausdrucksanweisung** (Abbildung 5.2). Auch hier stellt sich auf den ersten Blick die Frage nach dem Sinn einer solchen Möglichkeit.

#### Beispiel 5.1:

Die Anweisung

5-1       $3+4;$

berechnet  $3+4$ , was bekanntermaßen als Resultat den Wert 7 liefert. Dieser Wert wird nach der Berechnung aber nicht weiter verwertet! Das Interessante und Wichtige bei dieser Anweisungsform ist also die Auswertung des Ausdrucks an sich und nicht der Ergebniswert der Auswertung. ♦

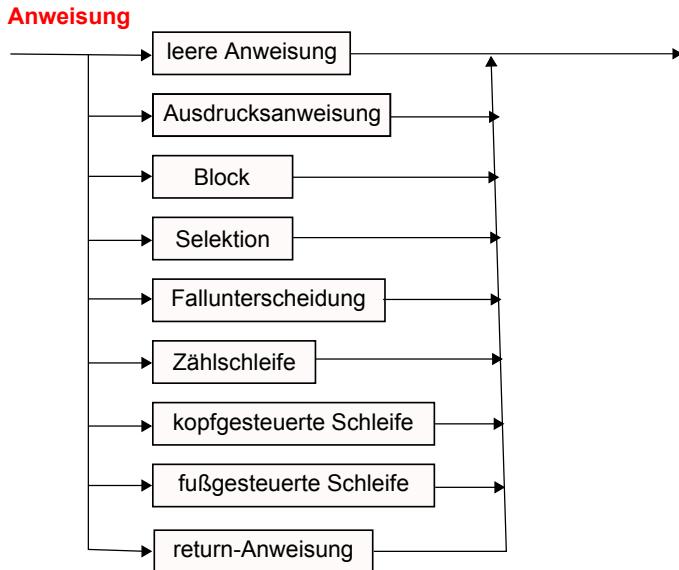


Figure 5.1: Syntaxdiagramm für Anweisungen



Figure 5.2: Syntaxdiagramm für die Ausdrucksanweisung

Eine Ausdrucksanweisung erlaubt syntaktisch jeden beliebigen Ausdruck, also auch obigen Programmausschnitt. Aber nur in wenigen Fällen macht dies auch Sinn (obiger Ausdruck eher weniger), nämlich dann, wenn in der *Auswertung des Ausdrucks* etwas Nachhaltiges, ein sogenannter *Nebeneffekt*, passiert. Nachfolgend sind die beiden wichtigsten und häufig vorkommenden Fälle dazu aufgeführt.

### 5.2.1 Zuweisung

In den meisten Fällen wird eine Ausdrucksanweisung im Zusammenhang mit Zuweisungen erfolgen. Eine **Zuweisung** hat die Aufgabe einer Variablen einen neuen Wert zuzuweisen, der durch einen Ausdruck angegeben wird. Eine einfache Form solch einer Zuweisung ist in Abbildung 5.3 gegeben, eine umfassendere Form wurde bereits in Abbildung 7.4 präsentiert. Die Seite links vom Gleichheitszeichen spezifiziert die Variable, deren Wert verändert werden soll. Die rechte Seite gibt in Form eines Ausdrucks an, wie sich der neue Wert der Variablen berechnet. Mit solch einer Zuweisung ist auch exakt definiert, in welcher Reihenfolge welche Aktionen erfolgen müssen. Wird ein solcher Ausdruck ausgewertet, so muss zuerst der Ausdruck auf der rechten Seite vollständig berechnet werden. Das Resultat ist ein Wert. Dieser Wert wird anschließend der Variablen auf der linken Seite des Gleichheitszeichens zugewiesen.

Der Vollständigkeit halber sei erwähnt, dass Java syntaktisch mehr Möglichkeiten auf der linken Seite des

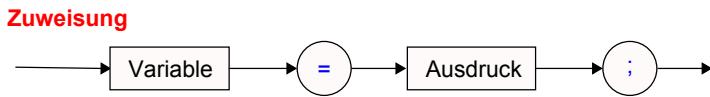


Figure 5.3: Syntaxdiagramm für eine Zuweisung

Gleichheitszeichens erlaubt, als es dieses Syntaxdiagramm darstellt. Einige Erweiterungen werden zum Beispiel später im Zusammenhang mit Feldern und Instanzen-/Klassenvariablen angegeben. Ein vollständiges Syntaxdiagramm für die Zuweisung, wie sie in diesem Buch behandelt wird, ist im Anhang aufgeführt (Anhang B). Für darüber hinaus gehende Möglichkeiten, die den Rahmen dieses Buches sprengen würden, sei der Leser auch hier auf die entsprechenden Literaturangaben zu Java verwiesen.

Als Merkregel gilt, dass auf der linken Seite in einer Zuweisung immer etwas stehen muss, was einen Speicherplatz zur Aufnahme eines Wertes bezeichnet, zum Beispiel ein Variablenbezeichner. Und auf der rechten Seite des Zuweisungsoperators muss ein Ausdruck stehen, in dem natürlich auch Variablenbezeichner auftreten können. Entsprechend dem Auftreten links und rechts vom Gleichheitszeichen einer Zuweisung spricht man auch von einem **L-Wert** – ein Speicherbereich wird angegeben – und **R-Wert** – der Inhalt eines Speicherbereichs wird genommen.

### 5.2.2 Methodenaufruf

Die erste sinnvolle Möglichkeit für eine Ausdrucksanweisung ist ein Methodenaufruf (oder auch Funktionsauswertung) der Form `f(x)`, wie er auch schon aus der Mathematik bekannt ist. Ein Methodenaufruf selbst ist ein Ausdruck. Alles Relevante zu Methoden wird im Kapitel 8 ausführlich behandelt, so dass an dieser Stelle aufgrund des Zusammenhangs nur auf die Möglichkeit hingewiesen wird.

#### Beispiel 5.2:

Die nachfolgende Anweisung wertet die uns bereits bekannte Methode `System.out.println` mit dem Argument `4711` aus. Das Resultat der Methodenauswertung in Form eines Ergebniswertes ist eigentlich irrelevant, wichtig ist aber, dass in der Auswertung der Methode die Ausgabe des Arguments (hier `4711`) auf dem Bildschirm erfolgt, also das eigentlich Interessante.

5-1      `System.out.println(4711);`

## 5.3 Sequenz und Block

Die Hintereinanderausführung von mehreren ausführbaren Anweisungen nennt man eine **Sequenz**. Die einzelnen Anweisungen werden im Programm auch hintereinander notiert. Die Bedeutung einer Sequenz ist, dass nacheinander, das heißt streng sequentiell, eine Anweisung nach der anderen in der vorgegebenen Reihenfolge ausgeführt wird. Vorgegebene Reihenfolge heißt bildlich von oben nach unten, wie man einen Text auch lesen würde.

Um auch mehrere Anweisungen an den Stellen verwenden zu können, an denen syntaktisch nur eine Anweisung erlaubt ist, ist es möglich, mehrere Anweisungen in Java und vielen anderen Programmiersprachen in einem **Block** zusammenzufassen, der syntaktisch genaue eine Anweisung ist. In einem Block sind weiterhin auch

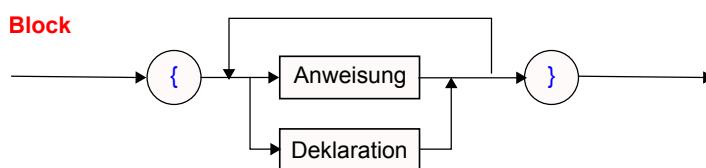


Figure 5.4: Syntaxdiagramm für einen Block

Listing 5.1: Umwandlung von Fahrenheit nach Celsius.

```

5-1  /**
5-2   * Umwandlung Grad Fahrenheit nach Grad Celsius
5-3   */
5-4  public class FahrenheitNachCelsius2 {
5-5      public static void main(String [] args) {
5-6
5-7          // Deklaration
5-8          double fahrenheit, celsius;
5-9
5-10         // Ausgangswert
5-11         fahrenheit = 80.0;
5-12
5-13         // Umrechnungformel anwendungen
5-14         celsius = 5.0 * (fahrenheit - 32.0) / 9.0;
5-15
5-16         // Ausgabe des Celsius-Wertes
5-17         System.out.println("Fahrenheit: " + fahrenheit + ", Celsius: " + celsius);
5-18     }
5-19 }
```

Deklarationen von Variablen erlaubt, die Auswirkungen werden in Kapitel 6 im dortigen Zusammenhang besprochen. Das Syntaxdiagramm für einen Block ist in Abbildung 5.4 zu sehen.

### Beispiel 5.3:

In dem Hauptprogramm des Beispiels in Listing 5.1 existieren vier Deklarationen beziehungsweise Anweisungen, die nacheinander ausgeführt werden. Zuerst werden in einer Deklaration zwei neue Variablen **fahrenheit** und **celsius** eingeführt. Anschließend wird in der Zuweisung **fahrenheit = 80.0** der Variablen **fahrenheit** der Wert 80 zugewiesen. In der nächsten Anweisung wird der Variablen **celsius** ein Wert zugewiesen, der sich durch Auswerten des Ausdrucks **5.0 \* (fahrenheit - 32.0) / 9.0** ergibt, also mathematisch exakt  $26,\overline{6}$ , was aber annäherungsweise in einer Fließkommavariablen dargestellt werden kann. Zuletzt wird in der vierten Anweisung die beiden Werte von **fahrenheit** und **celsius** zusammen mit einem Begleittext auf dem Bildschirm ausgegeben. ♦

Einen Block notiert man laut *Java Code Conventions*[Javb] so, dass die öffnende geschweifte Klammer an das Ende der laufenden Zeile geschrieben wird, der Inhalt des Block eingerückt in nachfolgenden Zeilen notiert wird und die schließende geschweifte Klammer alleine in einer Zeile bündig mit der Ursprungszeile angegeben wird. Das obige Beispiel zeigt die Formatierung an dem Block, der am Ende der Zeile, die **main** enthält, beginnt und in der zweitletzten Zeile aufhört.

## 5.4 Selektion

Die Selektion, also die gesteuerte Auswahl einer aus mehreren Handlungsalternativen, wird in Programmiersprachen unterschieden erstens in den Fall, dass eine Aktion nur in einem bestimmten Fall durchgeführt werden soll, zweitens, dass eine Aktion in einem bestimmten Fall und eine andere Aktion in allen anderen Fällen durchgeführt werden soll, und drittens, dass mehrere Fälle zur Auswahl stehen, von denen genau einer ausgewählt werden soll. Die ersten beiden Fälle werden durch die nachfolgende Einfachselektion

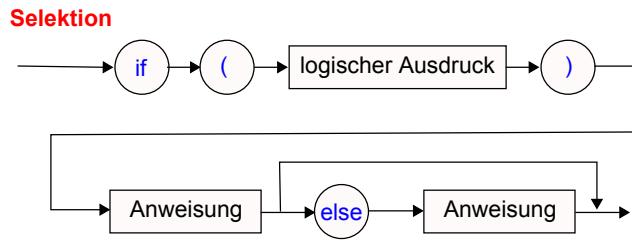


Figure 5.5: Syntaxdiagramm für eine Selektion

Listing 5.2: Berechnung des Maximums zweier Werte.

```

5-1  /**
5-2   * Berechnung des Maximums zweier Zahlen
5-3   */
5-4  public class MaxBerechnung {
5-5      public static void main(String[] args) {
5-6
5-7          // zwei Beispielwerte
5-8          int x = 5;
5-9          int y = 7;
5-10         // in der Variablen max steht anschliessend das Resultat
5-11         int max = 0;
5-12
5-13         // welcher Wert ist der groessere?
5-14         if(x > y)
5-15             max = x;
5-16         else
5-17             max = y;
5-18
5-19         // Ausgabe des gefundenen Wertes
5-20         System.out.println("Der groessere der beiden Werte ist " + max);
5-21     }
5-22 }
```

behandelt, der dritte Fall durch die später beschrieben Mehrfachselektion.

### 5.4.1 Einfachselektion

Die Selektion oder auch Auswahl dient dazu, anhand einer Bedingung einen von zwei möglichen Wegen in der weiteren Ausführung des Programms auszuwählen. Abbildung 5.5 zeigt das Syntaxdiagramm für eine Selektion in Java.

Zuerst wird der Bedingungsausdruck ausgewertet. Dieser Ausdruck muss einen Wahrheitswert liefern. Ist dieser Wert **true**, so wird die Anweisung hinter der schließenden Klammer ausgeführt (**then-Fall**), ist der Wert **false**, so wird – falls vorhanden – die zweite Anweisung hinter dem Schlüsselwort **else** ausgeführt (**else-Fall**), in jedem Fall aber die erste Anweisung nicht ausgeführt.

#### Beispiel 5.4:

Das Beispielprogramm in Listing 5.2 ermittelt in der Variablen **max** (dort steht anschließend das Ergebnis), welcher der beiden Werte, die in **x** und **y** stehen, der größere Wert ist. Für die angegebenen Beispielwerte 5 und 7 wäre das der Wert dementsprechend 7. ♦

Listing 5.3: Berechnung des Maximums und Minimums zweier Werte.

```

5-1  /**
5-2   * Berechnung des Maximums und Minimums zweier Zahlen
5-3   */
5-4  public class MaxMinBerechnung {
5-5      public static void main(String [] args) {
5-6
5-7          // zwei Beispielwerte
5-8          int x = 5;
5-9          int y = 7;
5-10         // in min und max stehen anschliessend die Resultatwerte
5-11         int max = 0;
5-12         int min = 0;
5-13
5-14         // welcher Wert ist der groessere?
5-15         // Daraus ergibt sich auch, wer der kleinere ist
5-16         if(x > y) {
5-17             max = x;
5-18             min = y;
5-19         } else {
5-20             max = y;
5-21             min = x;
5-22         }
5-23
5-24         // Ausgabe der gefundenen Werte
5-25         System.out.println("Der groessere der beiden Werte ist " + max);
5-26         System.out.println("Der kleinere der beiden Werte ist " + min);
5-27     }
5-28 }
```

Wie im Syntaxdiagramm zu sehen, besteht der then- und else-Fall jeweils syntaktisch aus genau einer Anweisung. Möchte man also mehrere Anweisungen dort ausführen lassen, muss man diese in einem Block zusammenfassen. Nach den Java Code Conventions schreibt man die öffnende geschweifte Klammer an das Ende der if-Zeile, während die schließende geschleifte Klammer des Blocks alleine in einer Zeile bündig unter dem `if` platziert wird. Analog gilt dies für die Platzierung der Symbole eines Blocks im else-zweig. Nachfolgend wird dies an einem Beispiel verdeutlicht.

### Beispiel 5.5:

Jetzt soll das Programm zur Ermittlung des Maximums von zwei Zahlen dahingehend erweitert werden (Listing 5.3), dass neben dem Maximum zusätzlich auch das Minimum der beiden Werte in der Variablen `min` ermittelt wird. Wenn man von zwei Zahlen das Maximum kennt, ist automatisch die andere Zahl das Minimum dieser beiden Zahlen. Das, was im if- und else-Fall zu tun ist, ist also keine einzelne Aktion mehr, sondern man muss sich jeweils zwei Sachen merken: was das Minimum in diesem Fall ist und was das Maximum ist. ♦

Von Programmieranfängern wird oft vergessen einen Block anzugeben, wenn mehr als eine auszuführende Anweisung vorliegt. In manchen solchen Fällen kann ein Compiler diesen Fehler aufgrund einer Syntaxverletzung erkennen, manchmal aber auch nicht.

**Beispiel 5.6:**

Der folgende *inkorrekte* Programm (versuchen Sie es zu übersetzen!) zeigt beispielhaft mögliche Problemfälle auf.

```

5-1  /** Fehlerhaftes Programm !!!
5-2  */
5-3  public class MaxMinBerechnungFehlerhaft {
5-4      public static void main( String [] args ) {
5-5
5-6          // zwei Beispielwerte
5-7          int x = 5;
5-8          int y = 7;
5-9          // in min und max stehen anschliessend die Resultatwerte
5-10         int max = 0;
5-11         int min = 0;
5-12
5-13         /* falsch: zwei Anweisungen sollten eigentlich im then-Fall
5-14             ausgefuehrt werden. Der Compiler erkennt beim else-Schlüsselwort ,
5-15             dass etwas syntaktisch falsch ist (wieso ?)
5-16             */
5-17         if (x > y)
5-18             // diese beiden Anweisungen sollten eigentlich im then-Fall
5-19             // ausgefuehrt werden
5-20             max = x;
5-21             min = y;
5-22         else
5-23             max = y;
5-24             min = x;
5-25
5-26         // gefaehrlicher Irrtum: Hier fehlt auch die Blockklammerung
5-27         // wie interpretiert dies ein Compiler? Was passiert?
5-28         if (x > y)
5-29             max = x;
5-30             min = y;
5-31     }
5-32 }
```

In der mit *falsch* gekennzeichneten Version gehört laut Syntaxdiagramm genau eine Anweisung zwischen **if** und **else**. Der Compiler wird also bei dem Schlüsselwort **else** einen Syntaxfehler erkennen und anzeigen, da die if-Anweisung bereits abgeschlossen ist (ohne else-Zweig).

Im davor gezeigten korrekten Beispielprogramm sind die beiden Anweisungen, die im then-Zweig ausgeführt werden sollen, wenn die Bedingung wahr ist, in einen Block eingeschlossen, der syntaktisch eine Anweisung ist und natürlich auch als solche syntaktisch korrekt erkannt wird.

Im Programmteil, der mit *gefährlicher Irrtum* gekennzeichnet ist, ist eine potentielle Fehlerquelle gezeigt, die ein Compiler *nicht* erkennen kann. Die eigentliche Intention war, die beiden Anweisungen **max = x;** und **min = y;** ausführen zu lassen, falls die Bedingung **x > y** erfüllt ist. Im Beispiel ist dies zwar durch Einrücken der beiden Anweisungen für einen menschlichen Leser kenntlich gemacht worden, aber diese Einrückung dient nur zur besseren Lesbarkeit des Programmes und suggeriert in diesem Fall leider das Falsche. Nur die erste Anweisung **max = x;** ist die Anweisung, die als then-Fall erscheint (syntaktisch gesehen die nächste Anweisung nach dem if-Test). Die zweite Anweisung **min = y;** wird hier in jedem Fall ausgeführt, also unabhängig von der Bedingung. Sie ist einfach die folgende Anweisung nach der if-Anweisung! Möchte man die eigentlich gedachte Version haben, so muss man die beiden Anweisungen in geschweiften Klammern als Block markieren, so dass sie syntaktisch als eine Anweisung erscheinen. ◆

Listing 5.4: Berechnung des Maximums und Minimums dreier Werte.

```

5-1  /**
5-2   * Bestimme das Maximum und Minimum von drei Zahlen
5-3   */
5-4  public class MaxMin3 {
5-5      public static void main(String[] args) {
5-6          int x, y, z, min, max;
5-7
5-8          // Beispiele fuer drei Werte 1,2,3
5-9          x = 1;
5-10         y = 2;
5-11         z = 3;
5-12
5-13         /* Vergleiche die ersten beiden Werte miteinander.
5-14          * Der kleinere Wert wird in min gespeichert,
5-15          * der groessere in max.
5-16          */
5-17         if (x < y) {
5-18             min = x;
5-19             max = y;
5-20         } else {
5-21             min = y;
5-22             max = x;
5-23         }
5-24
5-25         // nun wird das bisherige Minimum und Maximum
5-26         // mit dem dritten Wert verglichen.
5-27         if (z < min)
5-28             min = z;
5-29         if (z > max)
5-30             max = z;
5-31
5-32         // Ausgabe der Ergebnisse auf dem Bildschirm
5-33         System.out.println("Minimum=" + min + ", Maximum=" + max);
5-34     }
5-35 }
```

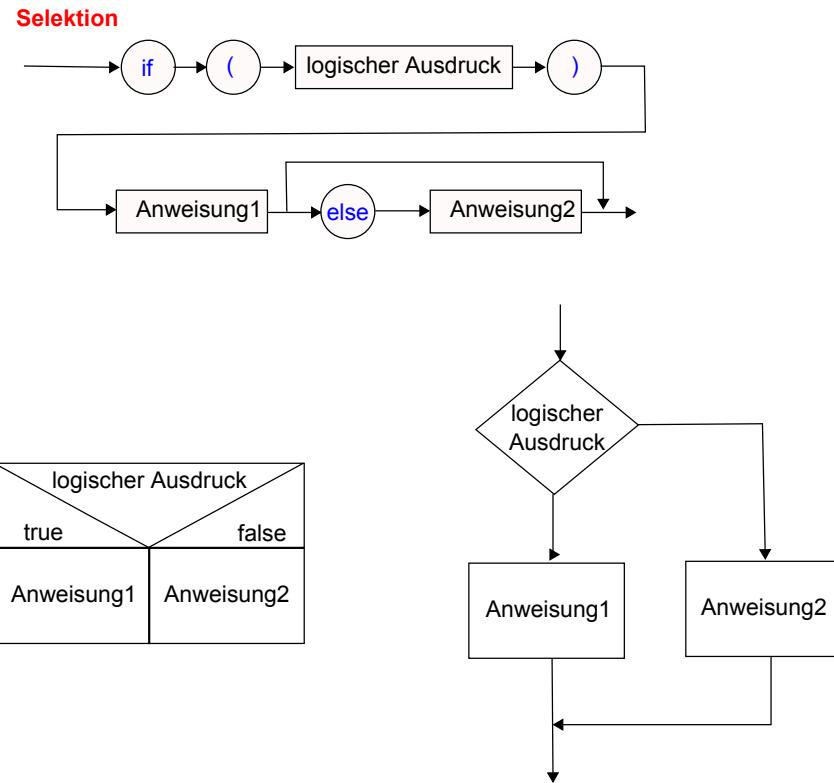


Figure 5.6: Syntaxdiagramm, Struktogramm und Flussdiagramm zur Selektion (zur Unterscheidung der beiden Anweisungen sind diese indiziert)

### Beispiel 5.7:

Im Programmbeispiel in Listing 5.4 wird das Minimum und Maximum von drei Zahlen bestimmt. Das Problem ist, dass nur der direkte Vergleich zweier Zahlen möglich ist. Kennt man allerdings die kleinere von zwei Zahlen, so kann dieses vorläufige Minimum mit der dritten Zahl verglichen werden, um so das Gesamtminimum zu erhalten. ♦

Eine Selektion lässt sich auch direkt in Form eines Flussdiagramms und eines Struktogramms angeben. Abbildung 5.6 gibt die entsprechende Übersetzung an für den Fall, dass der else-Zweig vorhanden ist. Ansonsten würde der entsprechende Teil auch in den Abbildungen leer bleiben (Struktogramm) beziehungsweise fehlen (Flussdiagramm).

Schaut man sich das Syntaxdiagramm einer Selektion an, so sieht man, dass im then- und else-Zweig genau eine Anweisung stehen kann / muss, und zwar eine beliebige Anweisung. Insofern kann man auch wiederum dort eine if-Anweisung benutzen, man kann also if-Anweisungen ineinander schachteln.

### Beispiel 5.8:

Nach dem gregorianischen Kalender ist ein Jahr ein Schaltjahr, wenn die Jahreszahl durch 4 teilbar ist, aber nicht durch 100. Ist sie aber durch 400 teilbar, so ist das Jahr doch ein Schaltjahr. Zum Beispiel sind die Jahre 2000, 2004, 2008, 2012, 2016 Schaltjahre, nicht aber 1900 und nicht 2100.

Die oben gezeigte Kombination aus geschachtelten if-else-Anweisungen lässt sich äquivalent dazu auch in einem kombinierten Wahrheitsausdruck angeben, wie er in Listing 5.6 zu sehen ist. ♦

Listing 5.5: Berechnung zu Schaltjahren über geschachtelte Selektionen.

```

5-1  /**
5-2   * Bestimme, ob ein Jahr des gregorianischen Kalenders ein Schaltjahr ist.
5-3   * Ein Jahr x ist Schaltjahr, wenn x durch 4 teilbar ist, aber nicht durch 100.
5-4   * Ist das Jahr aber durch 400 teilbar, so ist es doch ein Schaltjahr.
5-5  */
5-6 public class Schaltjahr2 {
5-7     public static void main(String[] args) {
5-8         // zu bestimmendes Jahr aus Kommandozeile
5-9         int jahr = Integer.parseInt(args[0]);
5-10        boolean istSchaltjahr;
5-11
5-12        // Ein Wert x ist durch y teilbar, wenn x % y == 0
5-13        if ((jahr % 4) == 0) {
5-14            if ((jahr % 100) == 0) {
5-15                if ((jahr % 400) == 0) {
5-16                    istSchaltjahr = true;
5-17                } else {
5-18                    istSchaltjahr = false;
5-19                }
5-20            } else {
5-21                istSchaltjahr = true;
5-22            }
5-23        } else {
5-24            istSchaltjahr = false;
5-25        }
5-26
5-27        if (istSchaltjahr)
5-28            System.out.println(jahr + " ist ein Schaltjahr");
5-29        else
5-30            System.out.println(jahr + " ist kein Schaltjahr");
5-31    }
5-32 }
```

Listing 5.6: Berechnung zu Schaltjahren über einen logischen Ausdruck.

```

5-1  /**
5-2   * Bestimme, ob ein Jahr des gregorianischen Kalenders ein Schaltjahr ist.
5-3   * Ein Jahr x ist Schaltjahr, wenn x durch 4 teilbar ist, aber nicht durch 100.
5-4   * Ist das Jahr aber durch 400 teilbar, so ist es doch ein Schaltjahr.
5-5  */
5-6 public class Schaltjahr {
5-7     public static void main(String[] args) {
5-8         // Beispiel fuer ein zu ueberpruefendes Jahr
5-9         int jahr = 2013;
5-10        boolean istSchaltjahr;
5-11
5-12        /* aquivalente Formulierung:
5-13           Eine Jahrzahl ist durch 4 und nicht durch 100 teilbar
5-14           oder sie ist durch 400 teilbar
5-15        */
5-16        istSchaltjahr = ( ( (jahr % 4) == 0)
5-17                      && (!((jahr % 100) == 0)))
5-18                      || ((jahr % 400) == 0);
5-19
5-20        System.out.println("Das Jahr " + jahr + " ist ein Schaltjahr? " + istSchaltjahr);
5-21    }
5-22 }
```

Bei einer Schachtelung von if-Anweisungen existiert jedoch ein *prinzipielles* Problem in der Zuordnung des else-Zweiges, was durch die Mehrdeutigkeit der grammatischen Regel entsteht. Dies soll an folgendem Beispiel eines Programmusschnitts erläutert werden:

```

5-1 if ( a > b )
5-2   if ( a > b )
5-3     c = a ;
5-4   else
5-5     c = b ;

```

Schaut man sich die Syntaxspezifikation in Abbildung 5.5 an, so gibt es in diesem Beispiel zwei Alternativen in der Zuordnung des else-Zweiges. Nach der Syntax ist es sowohl möglich, den else-Fall der ersten if-Anweisung als auch der zweiten if-Anweisung zuzuordnen. Im ersten Fall wäre der else-Zweig der zweiten Anweisung leer, im zweiten Fall der else-Zweig der ersten if-Anweisung leer. Diese Mehrdeutigkeit wird aufgelöst, indem in solchen Fällen der else-Zweig der letzten if-Anweisung zugeordnet wird, wie dies im Beispiel auch durch Einrücken bereits angedeutet wird. Es ist eine gute Programmierpraxis, immer einen Block im then- und else-Zweig zu verwenden, selbst wenn nur eine Anweisung vorliegt. Mit diesem Vorgehen gibt es keine Mehrdeutigkeit mehr und entspricht auch den Vorgaben der *Java Code Conventions*.

### 5.4.2 Mehrfachselektion

Der algorithmische Grundbaustein der Fallunterscheidung analog einer mehrfach geschachtelten Selektion ist ebenfalls in modernen Programmiersprachen vorhanden. Es bietet eine bequeme Notation für eine ansonsten nötige mehrfach geschachtelte Selektion. Eine (vereinfachte) Syntax ist in Abbildung 5.7 zu sehen. Die switch-Anweisung wird ausgeführt, indem der Ausdruck hinter dem Schlüsselwort **switch** zuerst ausgewertet wird. Der Resultatwert muss von einem ganzzahligen Typ sein (**byte**, **short**, **int**, **long**, **char**) sein. Anschließend wird dieser Wert nacheinander von oben nach unten mit den Werten aller Ausdrücke hinter den aufgeführten Schlüsselwörtern **case** verglichen, bis eine Übereinstimmung gefunden wird. Wird eine solche Übereinstimmung gefunden, so werden die Anweisungen zu diesem Fall ausgeführt, also was syntaktisch hinter dem Doppelpunkt zu diesem Fall steht. Sobald eine **break**-Anweisung ausgeführt wird, bricht die Bearbeitung der gesamten switch-Anweisung ab und die nächste Anweisung nach der switch-Anweisung wird ausgeführt.

#### Beispiel 5.9:

Das Beispielprogramm in Listing 5.7 zeigt eine Möglichkeit, eine Dezimalziffer in eine Zahl – den Wert der Ziffer – umzuwandeln. Als Anmerkung sei gesagt, dass dies a) einfacher zu realisieren ist unter Ausnutzung der Codeeigenschaften des Unicode und dass b) dies auch (vorzugsweise) mit Hilfe bereits vorhandener Methoden der Java Klassenbibliothek möglich ist.

Das Ziffernzeichen – im Programm beispielhaft die Ziffer **8** – wird nacheinander mit den Zeichen **0**, **1** und so weiter verglichen, bis im vorliegenden Beispieldfall bei dem Zeichen **8** eine Übereinstimmung gefunden wird. Deshalb wird die Anweisung hinter dem Doppelpunkt – hier die Anweisung **wert = 8;** – ausgeführt. Mit der anschließenden Ausführung der Anweisung **break** ist die Ausführung der gesamten Fallunterscheidung beendet und die nächste Ausführung des Programms wird mit der Anweisung der folgenden Anweisung **System.out.println(...)** fortgesetzt.

Zu beachten ist die Unterscheidung zwischen der Ziffer "8" und dem Wert 8. In alltäglichen Leben wird das synonym verwendet, aber dies ist nicht nur in Java ein Unterschied. An einer mehrstelligen Zahl wird dies klarer: die Darstellung **4711** enthält die vier Ziffern **4**, **7**, **1** und **1** in einer bestimmten Reihenfolge. Diese Zifferfolge interpretieren Menschen im Dezimalsystem als die Zahl 4711.

Als Ausgabe erscheint auf dem Bildschirm:

Listing 5.7: Umwandlung von Ziffern in Zahlen.

```

5-1  /**
5-2   * Umwandlung einer Dezimalziffer in die entsprechende Zahl
5-3   */
5-4  public class Dezimalziffernumwandlung {
5-5      public static void main(String[] args) {
5-6
5-7          // Beispiel fuer eine Ziffer
5-8          char ziffer = '8';
5-9
5-10         // in dieser Variablen steht anschliessend das Ergebnis
5-11         int wert = 0;
5-12
5-13         // Fallunterscheidung
5-14         switch(ziffer) {
5-15             case '0' : wert = 0;
5-16                 break;
5-17             case '1' : wert = 1;
5-18                 break;
5-19             case '2' : wert = 2;
5-20                 break;
5-21             case '3' : wert = 3;
5-22                 break;
5-23             case '4' : wert = 4;
5-24                 break;
5-25             case '5' : wert = 5;
5-26                 break;
5-27             case '6' : wert = 6;
5-28                 break;
5-29             case '7' : wert = 7;
5-30                 break;
5-31             case '8' : wert = 8;
5-32                 break;
5-33             case '9' : wert = 9;
5-34                 break;
5-35             default : System.out.println("hier stimmt was nicht");
5-36                 wert = -1;
5-37                 break;
5-38         }
5-39
5-40         System.out.println("Der Wert der Ziffer " + ziffer + " ist " + wert);
5-41     }
5-42 }
```

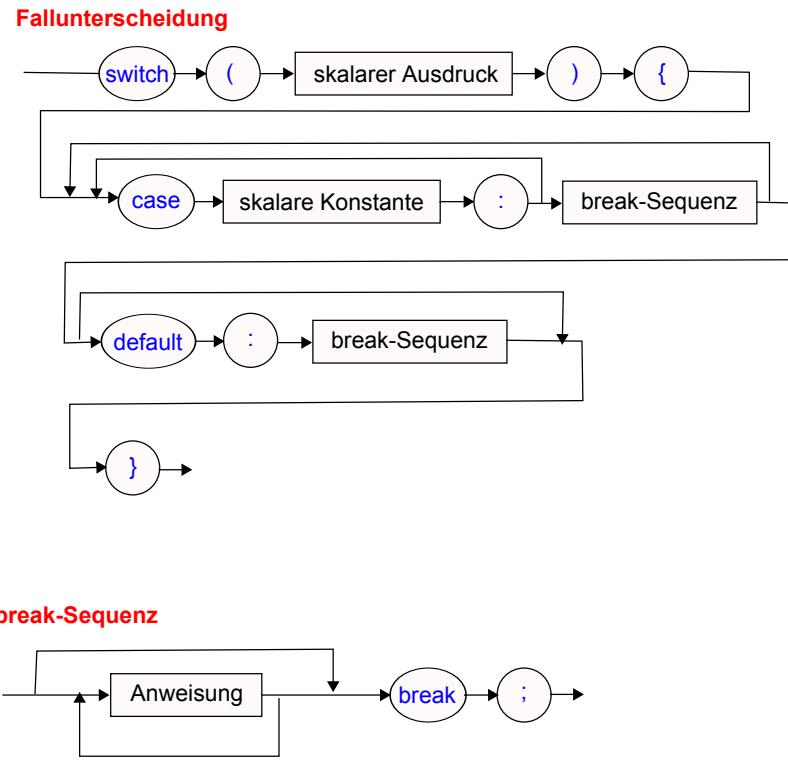


Figure 5.7: Syntaxdiagramm für eine Fallunterscheidung

5-1 | Der Wert der Ziffer 8 ist 8

### Beispiel 5.10:

Ein weiterführendes Beispiel ist die Berechnung der Anzahl der Tage in einem Monat. Entgegen dem angegebenen Syntaxdiagramm zur Fallunterscheidung in Abbildung 5.7, sieht man im Listing 5.8, dass auch zu mehreren Fällen die gleiche Aktion zugeordnet werden kann. Zur dieser erweiterten syntaktischen Möglichkeit sei auf den Java Sprachstandard verwiesen. ♦

Listing 5.8: Berechnung der Anzahl an Tagen in einem Monat.

```

5-1  /**
5-2   * Bestimmung der Anzahl der Tage zu einem Monat
5-3   */
5-4  public class Monatstage {
5-5      public static void main(String [] args) {
5-6
5-7          // Beispiel fuer einen Monat (1=Januar, 2=Februar,...)
5-8          int monat = 2;
5-9          // Beispiel fuer ein Jahr (Angabe wegen Schaltjahres noetig)
5-10         int jahr = 2020;
5-11
5-12         // in dieser Variablen steht anschliessend das Ergebnis
5-13         int anzahlTage = 0;
5-14
5-15         // Fallunterscheidung
5-16         switch(monat) {
5-17             // Monate mit 31 Tagen
5-18             case 1 :
5-19             case 3 :
5-20             case 5 :
5-21             case 7 :
5-22             case 8 :
5-23             case 10 :
5-24             case 12 : anzahlTage = 31;
5-25                 break;
5-26
5-27             // Monate mit 30 Tagen
5-28             case 4 :
5-29             case 6 :
5-30             case 9 :
5-31             case 11 : anzahlTage = 30;
5-32                 break;
5-33
5-34             // Februar ist etwas komplizierter wegen Schaltjahr
5-35             case 2 : if(((jahr%4==0) && (jahr%100 != 0))
5-36                     || (jahr % 400 == 0))
5-37                 anzahlTage = 29;      // Schaltjahr
5-38             else
5-39                 anzahlTage = 28;      // kein Schaltjahr
5-40                 break;
5-41
5-42             default : System.out.println("Monatsangabe nicht gueltig");
5-43                 anzahlTage = -1;
5-44                 break;
5-45         }
5-46
5-47         System.out.println("Anzahl der Tage ist " + anzahlTage);
5-48     }
5-49 }
```

## 5.5 Iteration

Bis jetzt wurden ausschließlich Kontrollstrukturen vorgestellt, die explizit den Ablauf eines Programms "von oben nach unten" steuern. Das soll heißen, dass mit dem jetzigen Kenntnisstand ein Programm so abläuft, wie es notiert wurde oder wie man es von oben nach unten lesen würde. Wollte man viele Aktionen durchführen, würde das Programm auch sehr lang werden, da jede einzelne Aktion notiert werden müsste, auch wenn es gleiche Aktion wäre, die vorher schon durchgeführt wurde. So sind bisher auch nur Berechnungen möglich mit einer festen Anzahl von Berechnungsschritten. Sind fünf Anweisungen angegeben, werden genau diese fünf Anweisungen durchgeführt. Wollte man sechs Anweisungen ausführen, müsste man das Programm entsprechend ergänzen.

### Beispiel 5.11:

$x^y$  ist definiert als  $x \cdot x \cdot \dots \cdot x$  ( $y$  mal) oder etwas anders notiert:  $1 \cdot x \cdot x \cdot \dots \cdot x$  ( $y$  mal) unter Ausnutzung des neutralen Elements der Multiplikation, so dass auch der Fall  $x^0$  Sinn macht. Wollte man ein Programm zur Berechnung der Potenz schreiben, so müsste man mit den bisher bekannten Mitteln etwa folgendes Vorgehen wählen:

```

5-1 public class ExplizitePotenzberechnung {
5-2     public static void main(String[] args) {
5-3         // Beispielwerte fuer x und y
5-4         int x = 5, y = 4;
5-5
5-6         // hier wird das Ergebnis aufgebaut
5-7         int potenz = 1;
5-8
5-9         if (y > 0) {
5-10             potenz = potenz * x;
5-11
5-12             if (y > 1) {
5-13                 potenz = potenz * x;
5-14
5-15                 if (y > 2) {
5-16                     potenz = potenz * x;
5-17                     ...
5-18                 }
5-19             }
5-20         }
5-21     }
5-22 }
```

Das Problem liegt also darin, dass man mit den bisherigen Mitteln keine Möglichkeit hat, gleiche oder ähnliche Anweisungen zu wiederholen, ohne die einzelnen Aktionen mehrfach hinschreiben zu müssen. Was also in der obigen Notation  $1 \cdot x \cdot x \cdot \dots \cdot x$  ( $y$  mal) die Aussage " $y$  mal" ist, wurde bis zu dieser Stelle für Programmiersprachen noch nicht eingeführt. Eine ähnliche Problematik liegt ebenfalls in der oft genutzten Summenbildung der Mathematik mit der Notation  $y = \sum_{i=1}^n i$  vor. Auch hier ist bis zu dieser Stelle noch kein Mittel bekannt, eine solche Summenbildung in Abhängigkeit eines Wertes  $n$  anzugeben.

Es wurde aber schon früher angegeben, dass die *Wiederholung* oder *Iteration* in verschiedenen Formen ein algorithmischer Grundbaustein ist. Im Folgenden werden drei Möglichkeiten der Wiederholung von Aktionen vorgestellt: einmal mit einer festen Anzahl an Wiederholungen und zwei Varianten, bei der die Anzahl der Wiederholungen nicht a priori feststehen muss, sondern weitere Wiederholungen von einer Bedingung

abhängig sind.

### 5.5.1 Zählschleife

Um das Beispiel der Potenzbildung  $x^y$  aufzugreifen würde man gerne notieren können:

```

5-1 int x = 5, y = 4;           // Beispielwerte
5-2 int potenz = 1;            // neutrales Element der Multiplikation
5-3 wiederhole y-mal          // Kontrolle der Wiederholung
5-4     potenz = potenz * x;   // zu wiederholende Aktion

```

Dies soll bedeuten, dass die Aktion `potenz = potenz * x;` insgesamt  $y$  mal ausgeführt werden soll, wobei der Wert  $y$  zu Beginn der Wiederholung feststehen muss.

Eine Summenbildung  $y = \sum_{i=1}^n i$  wie oben beispielhaft angegeben ließe sich mit diesem Ansatz ebenfalls analog formulieren:

```

5-1 int n = 100;              // Beispielwert Obergrenze Summe
5-2 int y = 0;                // neutrales Element der Addition
5-3 wiederhole i von 1,...,n  // Kontrolle der Wiederholung
5-4     y = y + i;            // zu wiederholende Aktion

```

Hier nutzt man also zusätzlich aus, dass mit jeder Wiederholung ein Zählerwert verbunden sein kann, wie es ja auch in der mathematischen Notation ausgenutzt wird.

Einen solchen Schleifentyp, bei dem die Anzahl der Wiederholungen zu Beginn der Schleife feststeht, nennt man **Zählschleife**. Der Grund für den Namen liegt darin, dass die "normale" Benutzung einer solchen Zählschleife sich so gestaltet, dass eine für diese Schleife zugeordnete Schleifenvariable von einem Startwert bis zu einem Endwert hochgezählt wird, das heißt mit jeder Wiederholung um eins erhöht wird. Im Beispiel oben würde die Variable `i` von 1 beginnend hochgezählt bis  $n$  und mit jedem neuen  $i$ -Wert die zu wiederholende Aktion ausgeführt.

Eine Schleife enthält dabei zwei Elemente, die in den beiden Beispielen auch entsprechend in den Kommentaren benannt wurden:

- einen Schleifenkopf, in dem alle Angaben zur Wiederholung stehen
- einen Schleifenrumpf, in dem die zu wiederholende Aktion notiert wird

Die Syntax für eine Zählschleife in Java, wie sie hier eingesetzt werden soll, ist in Abbildung 5.8 gegeben. Syntaktisch sind in Java (ebenso wie in C/C++) mehr Möglichkeiten vorhanden und diese Schleifenform ist auch allgemeiner anwendbar als hier angegeben. Diese Schleifenform soll aber vorerst nur als reine Zählschleife in dieser eingeschränkten Form verwendet werden.

Eine solche Zählschleife hat also eine feste Struktur bestehend aus einem Schleifenkopf und einem Schleifenrumpf syntaktisch bestehend aus genau einer Anweisung. Vergleichen Sie die Beschreibung mit dem nachfolgenden Beispiel, wo Sie die einzelnen Teilkonstrukte identifizieren können. Der Schleifenkopf besteht aus genau drei Teilen. Er beginnt mit einer Initialisierung in Form einer Variablen Deklaration der dieser Schleife zugeordneten Zählvariablen mit einem Initialisierungswert gegeben durch einen Ausdruck. Beispiele dazu sind `int i=1;` oder `char c ='A';`. Der zweite Teil ist ein logischer Ausdruck und gibt an, ob das Abbruchkriterium erfüllt oder nicht erfüllt ist. In der hier vorgeschlagenen Nutzungsweise wird sinnvollerweise die Zählvariable in das Abbruchkriterium eingehen. Der Schleifenrumpf wird solange wiederholt, solange der Wert dieses Ausdrucks `true` ergibt. Beispiele: `i < 100;` oder `c <= 'Z'`. Der dritte und letzte Teil des Schleifenkopfs gibt an, wie die Zählvariable verändert werden soll. Beispiele: `i++` oder `c=c+1`. Dieser Teil wird im Gegensatz zu den beiden anderen Teilen nicht durch ein Semikolon abgeschlossen!

Bezüglich des Schleifenrumpfs ist zu beachten, dass syntaktisch genau eine Anweisung angegeben werden muss. Möchte man dort nichts ausführen, so muss die leere Anweisung genutzt werden. Möchte man mehr als

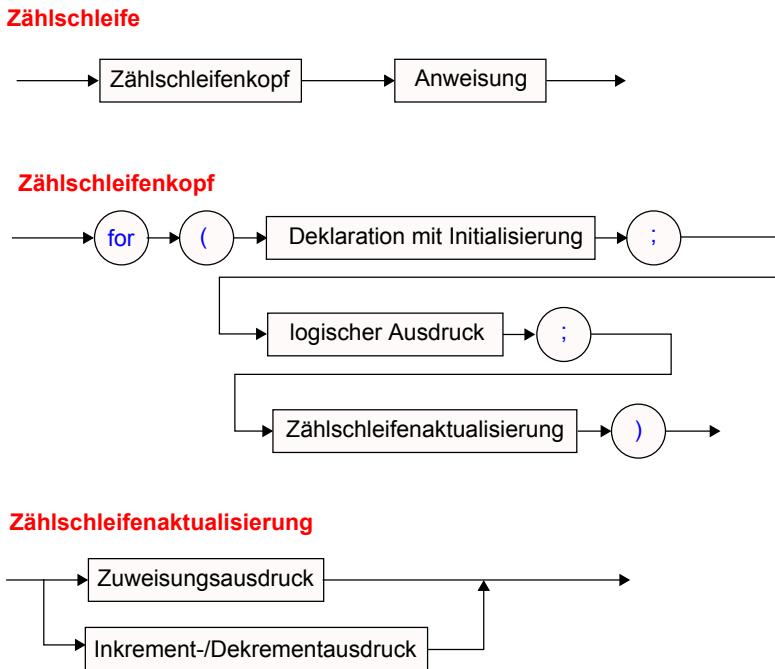


Figure 5.8: Syntaxdiagramm für Zählschleifen

eine Anweisung ausführen, so muss ein Block genutzt werden. Die *Java Code Conventions* schreiben vor, dass man in jedem Fall einen Block notieren soll, auch wenn der Schleifenrumpf nur aus einer Anweisung besteht. Dieses soll auch im Folgenden eingehalten werden.

Der Ablauf solch einer Zählschleife ist nun wie folgt: Die Schleife wird ausgeführt, indem die Initialisierung genau einmal ausgeführt wird. Anschließend wird das Abbruchkriterium, der logische Ausdruck, überprüft. Ist die Bedingung `false`, bricht die Schleife an dieser Stelle sofort ab und die Ausführung wird mit der nächsten Anweisung hinter der Schleife fortgeführt. Ist die Bedingung aber `true`, so wird der Schleifenrumpf (die Anweisung) einmal ausgeführt. Bevor die Bedingung erneut überprüft wird, wird jedoch vorher noch der dritte Teil des Schleifenkopfs (Zählschleifenaktualisierung) durchgeführt. Die Schleife wird solange iteriert (wiederholt), bis das Abbruchkriterium den Wert `false` ergibt.

### Beispiel 5.12:

In Listing 5.9 ist das Beispiel zur Potenzberechnung  $x^y$  für natürliche Zahlen  $x, y \in \mathbb{N}$  gegeben.

In der Initialisierung der Zählschleife `int i=1;` wird die Zählvariable `i` angelegt und auf den Startwert 1 gesetzt. Das Abbruchkriterium `i <= y` ist der Test, ob der Wert der Zählvariablen den Endwert `y` überschritten hat (Abbruch) oder noch nicht (weitermachen). Und in der Zählschleifenaktualisierung `i = i + 1` wird die Zählvariable `i` um den Wert 1 erhöht. Die wiederholt auszuführende Anweisung `potenz = potenz * x;` ist die Multiplikation des bisherigen Potenzwertes mit dem Wert, den die Variable `x` angibt.

Kontrollfrage: Was passiert, wenn zu Beginn `y` statt 4 den Wert 0 hat? Gehen Sie das Beispiel mit Papier und Bleistift durch, indem Sie zu jeder vorkommenden Variablen den aktuellen Wert auf dem Blatt notieren und bei jedem Schritt im Programm die entsprechende Änderung von Variablen vornehmen. ♦

Wie man in diesem Beispiel auch sieht, erhält man mit der Zählvariablen auch zu jeder Schleifeniteration eine "Nummer", nämlich den Wert der Zählvariablen. Dieser Wert kann in Berechnungen genutzt werden.

Listing 5.9: Potenzberechnung mit einer Zählschleife.

```

5-1  /**
5-2   * Potenzberechnung mit einer Zählschleife
5-3   */
5-4  public class PotenzberechnungZählschleife {
5-5      public static void main(String [] args) {
5-6
5-7          // Beispielwerte
5-8          int x = 5, y = 4;
5-9
5-10         // Ergebnisvariable mit neutralem Element der Multiplikation
5-11         int potenz = 1;
5-12
5-13         // zähle i von 1 bis y hoch
5-14         for(int i = 1; i <= y; i = i + 1) {
5-15             potenz = potenz * x;
5-16         }
5-17
5-18         System.out.println(x + " hoch " + y + "=" + potenz);
5-19     }
5-20 }
```

**Beispiel 5.13:**

Es soll die Summe  $\sum_{i=1}^n i$  berechnet werden. Der Java-Code dazu ist in Listing 5.10 zu sehen. In jeder Iteration wird also der aktuelle Wert der Zählvariablen zum Gesamtergebnis addiert. ♦

**Beispiel 5.14:**

Ebenso lässt sich für eine Zahl  $n \in \mathbb{N}_0$  die Fakultätsfunktion  $fakultaet(n) = 1 \cdot 2 \cdot \dots \cdot n = \prod_{i=1}^n i$  berechnen. Ein Spezialfall ist der Fall  $n = 0$ : in diesem Fall ist definiert  $fakultaet(0) = 1$ . Eine alternative Schreibweise der Fakultätsfunktion aus der Mathematik ist auch  $n!$ . Beispiel:  $3! = 3 \cdot 2 \cdot 1 = 6$ .

Die bisher multiplizierten Teilprodukte werden in der Variablen `fakultaet` gespeichert. Zu Beginn ist der Wert der Variablen 1, dem neutralen Element der Multiplikation (entsprechend  $0! = 1$ ). In der Schleife kann aus dem bisher ermittelten Wert von  $(n-1)!$  leicht der Wert  $n!$  durch Multiplikation mit dem Wert der Schleifenvariablen berechnet werden. ♦

Da eine Schleife selber wieder eine Anweisung ist, kann eine Schleife natürlich eine weitere Schleife als Schleifenrumpf – syntaktisch eine Anweisung – haben.

**Beispiel 5.15:**

Es soll die Summe  $\sum_{i=1}^n \sum_{j=1}^n (i + j)$  berechnet werden. Wie man an diesem Beispiel in Listing 5.12 sieht, hat jede Zählschleife ihre eigene Schleifenvariable, in der in gewisser Hinsicht der Stand der Schleifenbearbeitung gespeichert ist.

Wichtig zum Verständnis dieses Beispiels (aber auch ganz allgemein von Schleifen) ist, dass für jeden Wert von `i` in der äußeren Schleife die innere `j`-Schleife entsprechend von vorne durchlaufen wird. Der Schleifenrumpf der `i`-Schleife ist die gesamte `j`-Schleife. Wenn also das Abbruchkriterium der `i`-Schleife den Wert `true` ergibt (das heißt kein Abbruch der `i`-Schleife), so wird der Schleifenrumpf der `i`-Schleife komplett "von vorne beginnend" ausgeführt. Also jeweils eine neue Variable `j` angelegt, diese mit dem Wert 1 initialisiert, der Abbruchtest `j <= n` ausgewertet und so weiter. Bricht irgendwann die innere `j`-Schleife ab, so wird ganz normal

Listing 5.10: Summenberechnung.

```

5-1 /**
5-2 * Summenbildung mit Schleifenvariable
5-3 */
5-4 public class SummeUeberI {
5-5
5-6     public static void main(String [] args) {
5-7         // Beispielwert
5-8         int n = 10;
5-9
5-10        // Ergebnisvariable mit neutralem Element der Addition
5-11        int summe = 0;
5-12
5-13        for(int i=1; i <= n; i = i + 1) {
5-14            summe = summe + i;
5-15        }
5-16
5-17        System.out.println("Summe ueber " + n + " ist " + summe);
5-18    }
5-19 }
```

Listing 5.11: Fakultätsberechnung mit einer Zählschleife.

```

5-1 /**
5-2 * Fakultaetsberechnung ueber Zahlschleife
5-3 */
5-4 public class FakultaetZahlschleife {
5-5     public static void main(String [] args) {
5-6         // Beispielwert
5-7         int n = 10;
5-8
5-9         // Ergebnisvariable mit neutralem Element der Fakultaet
5-10         int fakultaet = 1;
5-11
5-12         for(int i=1; i <= n; i = i + 1) {
5-13             fakultaet = fakultaet * i;
5-14         }
5-15
5-16         System.out.println("fakultaet(" + n + ")=" + fakultaet);
5-17     }
5-18 }
```

Listing 5.12: Berechnung einer Doppelsumme.

```

5-1  /*
5-2   * Summe über i und j bilden
5-3   */
5-4 public class SummeUeberIUndJ {
5-5     public static void main(String [] args) {
5-6       // Beispielwert
5-7       int n = 10;
5-8
5-9       // Ergebnisvariable mit neutralem Element der Operation
5-10      int summe = 0;
5-11
5-12      for(int i=1; i <= n; i = i + 1) {
5-13        for(int j=1; j <= n; j = j + 1) {
5-14          summe = summe + (i + j);
5-15        }
5-16      }
5-17
5-18      System.out.println("Doppelsumme ueber " + n + " ist " + summe);
5-19    }
5-20  }

```

mit der äußeren i-Schleife weitergarbeitet, also die Zählschleifenaktualisierung der i-Schleife durchlaufen und anschließend das Abbruchkriterium überprüft. Der Schleifenrumpf einer (Zähl-)Schleife ist also quasi als Black Box anzusehen, die gesteuert über den Schleifenkopf entsprechend oft ausgeführt wird, was immer der Schleifenrumpf auch sein mag. ♦

### Beispiel 5.16:

Ein weiteres Beispiel für geschachtelte Schleifen ist die Erzeugung aller dreistelligen Binärzahlen in Zweierkomplementdarstellung durch Permutation der Ziffern (siehe Listing 5.13). Jede Ziffer kann dabei den Wert 0 oder 1 annehmen. Zur Erinnerung: der Wert einer Zahl  $b_{n-1} \dots b_0$  in Zweierkomplementdarstellung war:

$$\text{Wert}(b_{n-1} \dots b_0) = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

Die drei Zählschleifen stehen für die drei Binärziffern. Jede Schleife "erzeugt" alle Möglichkeiten für die jeweilige Binärziffer. Die äußere i2-Schleife ist also dafür zuständig, dass alle möglichen Werte (0 und 1) für die höchstwertige Binärziffer erzeugt werden. Die mittlere i1-Schleife produziert alle möglichen Werte der mittleren Ziffer und die innerste i0-Schleife variiert das niederwertigste Bit der Darstellung.

Als Ausgabe des Programms erfolgt:

```

5-1 000=0
5-2 001=1
5-3 010=2
5-4 011=3
5-5 100=-4
5-6 101=-3
5-7 110=-2
5-8 111=-1

```

Bevor Sie weiterlesen sollte Ihnen der exakte Ablauf einer geschachtelten Schleife im Detail klar sein. ♦

Listing 5.13: Erzeugung aller dreistelligen Binärzahlen.

```

5-1 /**
5-2 * gebe alle moeglichen 3-stelligen Zweierkomplementzahlen aus
5-3 */
5-4 public class Binaerzahlen {
5-5     public static void main(String [] args) {
5-6         // Erzeugen aller moeglichen Kombination von drei Binaerziffern
5-7         for(int i2=0; i2 <= 1; i2++) {
5-8             for(int i1=0; i1 <= 1; i1++) {
5-9                 for(int i0=0; i0 <= 1; i0++) {
5-10                     // dies ist die Formel einen Zweierkomplementwert
5-11                     int wert = ((-i2*4) + i1*2 + i0*1);
5-12                     System.out.println(i2 + " " + i1 + " " + i0 + "=" + wert);
5-13                 }
5-14             }
5-15         }
5-16     }
5-17 }
```

Listing 5.14: Erzeugung von Quadrat- und Kubikzahlen.

```

5-1 /**
5-2 * Quadratzahlen und Kubikzahlen zwischen 10 und 20 erzeugen
5-3 */
5-4 public class Quadratzahlen {
5-5     public static void main(String [] args) {
5-6
5-7         // Ueberschrift ausgeben
5-8         System.out.println("n n^2 n^3");
5-9
5-10        // von 10 beginnend bis einschlieslich 20 zaehlen
5-11        for(int i=10; i <= 20; i++) {
5-12            // i*i ist die Quadratzahl zu i, i*i*i die Kubikzahl
5-13            System.out.println(i + " " + (i*i) + " " + (i*i*i));
5-14        }
5-15    }
5-16 }
```

Bis jetzt wurde die Zählvariable einer Schleife immer mit einem Startwert 1 belegt. Der Startwert kann aber beliebig gewählt werden.

### Beispiel 5.17:

Es sollen für  $n = 10$  bis  $20$  die Werte  $n^2$  und  $n^3$  ausgegeben werden. Listing 5.14 zeigt ein entsprechendes Programm dazu. Die Ausgabe ist in Listing 5.15 angegeben. ♦

Und genauso kann man natürlich auch rückwärts zählen.

### Beispiel 5.18:

Diesmal sollen die Zahlen  $n^2$  und  $n^3$  im umgekehrter Reihenfolge zum letzten Beispiel, also von  $n = 20$  bis

Listing 5.15: Ausgabe zur Programmausführung des Programms in Listing 5.14.

```

5-1 n  n^2  n^3
5-2 10 100 1000
5-3 11 121 1331
5-4 12 144 1728
5-5 13 169 2197
5-6 14 196 2744
5-7 15 225 3375
5-8 16 256 4096
5-9 17 289 4913
5-10 18 324 5832
5-11 19 361 6859
5-12 20 400 8000

```

Listing 5.16: Erzeugung von Quadrat- und Kubikzahlen.

```

5-1 /**
5-2 * Quadratzahlen und Kubikzahlen zwischen 10 und 20 rückwärts erzeugen
5-3 */
5-4 public class Quadratzahlen2 {
5-5     public static void main(String[] args) {
5-6
5-7         // Überschrift ausgeben
5-8         System.out.println("n  n^2  n^3");
5-9
5-10        // von 20 beginnend bis einschließlich 10 runterzählen
5-11        for(int i=20; i >= 10; i--) {
5-12            // i*i ist die Quadratzahl zu i, i*i*i die Kubikzahl
5-13            System.out.println(i + " " + (i*i) + " " + (i*i*i));
5-14        }
5-15    }
5-16 }

```

$n = 10$ , ausgegeben werden. Listing 5.16 zeigt die Änderung dazu.

Zu beachten dabei ist, dass das Abbruchkriterium zu der Initialisierung und der Zählschleifenaktualisierung passen muss. Will man vorwärts zählen, muss man die Schleifenvariable in der Zählschleifenaktualisierung erhöhen und der Schleifenrumpf muss wiederholt werden, solange die Obergrenze noch nicht überschritten wurde ( $\leq$  **Obergrenze**). Will man rückwärts zählen, muss die Schleifenvariable in der Zählschleifenaktualisierung reduziert werden und der Schleifenrumpf wird ausgeführt, solange die Schleifenvariable die Untergrenze nicht unterschritten hat ( $\geq$  **Untergrenze**). ♦

### Beispiel 5.19:

Ein letztes Beispiel zu Zählschleifen ist die (vereinfachte) Berechnung von Annuitätendarlehen, wie sie zum Beispiel in einer Hausfinanzierung oft genutzt wird. Der Darlehensnehmer möchte ein Darlehen in einer Höhe von  $K$  Euro zu einem Zinssatz von  $p$  Prozent aufnehmen und ist bereit monatlich  $r$  Euro zu bezahlen. Die Laufzeit des Darlehens beträgt  $n$  Monate. Der konstante monatliche Betrag  $r$  wird aufgeteilt in eine Zinsteil  $r_1$  und eine Tilgungsanteil  $r_2$  mit  $r = r_1 + r_2$  konstant. Durch die monatliche Tilgung nimmt langsam das Kapital ab und damit verändert sich entsprechend der Zinsanteil, der im Laufe der Monate langsam abnimmt. Aufgrund der konstanten monatlichen Rate wird damit der Tilgungsanteil kontinuierlich größer.

Für jeden Monat muss also neu bestimmt werden, wie sich aus dem vorherigen Monat der Zinsanteil  $r_1$  und der Tilgungsanteil  $r_2$  berechnet und wie sich dadurch das Kapital verändert.

In Listing 5.18 ist die Ausgabe des Programmes aus Listing 5.17 gezeigt. Man würde sich an dieser Stelle natürlich eine schön formatierte Tabelle wünschen, in der Geldbeträge genau zwei Stellen hinter dem Komma haben, Zahlen untereinander stehen und so weiter. In Kapitel 4.7.4 wurden dazu Möglichkeiten aufgezeigt. Der Leser kann als Übung das Programm selbst so modifizieren, dass eine solche Ausgabe erfolgt.

In 12 Monaten wurden also im Beispiel 12000 Euro bezahlt (1000 Euro monatlich), die sich aufteilten in insgesamt 5832.22 Euro Zinsen und 6167.78 Euro Tilgung. Nach 12 Monaten beträgt die Restschuld noch 93832.22 Euro.

Als Hinweis sei hier angemerkt, dass aufgrund der Verwendung des Datentyps `double` es, wie man an der Ausgabe sieht, für die meisten Werte zu (Zwischen-)Resultaten kommt, die nicht mit exakten Euro- oder Centbeträgen übereinstimmen. Würde die Verwendung eines ganzzahligen Datentyps wie zum Beispiel `int` das Problem lösen? ♦

Auf ein oft gemachten Programmierfehler von Anfängern im Zusammenhang mit zusammengesetzten Anweisungen sei an dieser Stelle hingewiesen. Gegeben ist das Programm im Listing 5.19 mit einer leicht erweiterten Syntax für Zählschleifen als bisher vorgestellt, um den Fehler darstellen zu können.

Dies ist ein syntaktisch korrektes Programm! Nur wird das Programm nicht das gewünschte Resultat liefern. Sehen Sie den Fehler? Das Problem ist das Semikolon hinter dem Schleifenkopf. Ein Schleifenrumpf besteht aus genau der Anweisung, die hinter dem Schleifenkopf angegeben ist. Im Beispiel ist dies die leere Anweisung, die durch das Semikolon gegeben ist. Das gleiche Programm entsprechend der Struktur des Programms formatiert sähe wie in Listing 5.20 aus. Es würde also 10 mal nichts gemacht (leere Anweisung) und dann anschließend genau ein mal der Block mit der Ausgabeanweisung ausgeführt.

Listing 5.17: Berechnung von Annuitätendarlehen.

```

5-1  /**
5-2   * Berechnung eines Annuitätendarlehens
5-3   */
5-4  public class Annuitätendarlehen {
5-5      public static void main(String[] args) {
5-6
5-7          // Beispielwerte
5-8          double kapital = 100000.0;           // Darlehnssumme
5-9          double zinssatz = 0.06;            // Zinssatz 6% p.a.
5-10         double zahlung = 1000.0;          // monatliche Zahlung
5-11         int laufzeit = 12;              // Laufzeit in Monaten
5-12
5-13         // Variablen fuer zu berechnende Werte
5-14         double tilgung, zinsen, summe_zinsen = 0.0, summe_tilgung = 0.0;
5-15
5-16         // Ueberschrift ausgeben
5-17         System.out.println("Monat Restschuld Zinsen Tilgung");
5-18
5-19         // wir laufen ueber alle Monate
5-20         for(int monat = 1; monat <= laufzeit; monat++) {
5-21             // Zinsen auf das aktuell verbliebene Kapital
5-22             zinsen = kapital * zinssatz / 12.0;
5-23             // Tilgung ist die konstante monatliche Zahlung abzueglich der Zinsen
5-24             tilgung = zahlung - zinsen;
5-25             // Restschuld reduziert sich um Tilgung
5-26             kapital = kapital - tilgung;
5-27             // wir berechnen Gesamtzinsen und -tilgung
5-28             summe_zinsen = summe_zinsen + zinsen;
5-29             summe_tilgung = summe_tilgung + tilgung;
5-30
5-31             // gebe aktuelle Stand fuer den Monat aus
5-32             System.out.println(monat + " " + kapital + " "
5-33                         + zinsen + " " + tilgung);
5-34         }
5-35
5-36         // Endergebnis ausgeben
5-37         System.out.println("Summe Zahlungen: "
5-38                         + (summe_zinsen + summe_tilgung));
5-39         System.out.println("Summe Zinsen: " + summe_zinsen);
5-40         System.out.println("Summe Tilgung: " + summe_tilgung);
5-41         System.out.println("Restschuld: " + kapital);
5-42     }
5-43 }
```

Listing 5.18: Ausgabe zur Annuitätenberechnung.

```

5-1 Monat Restschuld Zinsen Tilgung
5-2 1 99500.0 500.0 500.0
5-3 2 98997.5 497.5 502.5
5-4 3 98492.4875 494.98749999999995 505.01250000000005
5-5 4 97984.9499375 492.4624375 507.5375625
5-6 5 97474.8746871875 489.9247496875 510.0752503125
5-7 6 96962.24906062344 487.3743734359375 512.6256265640625
5-8 7 96447.06030592656 484.8112453031172 515.1887546968828
5-9 8 95929.2956074562 482.23530152963275 517.7646984703672
5-10 9 95408.94208549347 479.646478037281 520.353521962719
5-11 10 94885.98679592094 477.04471042746735 522.9552895725326
5-12 11 94360.41672990053 474.42993397960464 525.5700660203954
5-13 12 93832.21881355003 471.80208364950266 528.1979163504973
5-14 Summe Zahlungen: 12000.0
5-15 Summe Zinsen: 5832.218813550043
5-16 Summe Tilgung: 6167.781186449957
5-17 Restschuld: 93832.21881355003

```

Listing 5.19: Fehlerhaftes Beispiel.

```

5-1 /**
5-2 * Eigentlich: Gebe alle Zahlen zwischen 0 und 9 auf dem Bildschirm aus
5-3 */
5-4 public class FehlerLeereAnweisung {
5-5     public static void main(String [] args) {
5-6         int i;
5-7
5-8         for(i=0; i<10; i++) {
5-9             System.out.println("i hat den Wert " + i);
5-10        }
5-11    }
5-12 }

```

Listing 5.20: Fehlerhaftes Beispiel, richtig formatiert.

```

5-1 /**
5-2 * Eigentlich: Gebe alle Zahlen zwischen 0 und 9 auf dem Bildschirm aus
5-3 */
5-4 public class FehlerLeereAnweisung2 {
5-5     public static void main(String [] args) {
5-6         int i;
5-7
5-8         for(i=0; i<10; i++) ;
5-9         {
5-10             System.out.println("i hat den Wert " + i);
5-11         }
5-12     }
5-13 }

```

### 5.5.2 Kopfgesteuerte Schleife

Zählschleifen führen in der oben eingeführten Form eine Anzahl von Iterationen aus, wobei diese Anzahl zu Beginn der Schleife feststehen muss. Dort, wo man diese Iterationsanzahl a priori bestimmen kann sollte man dann auch eine Zählschleife nehmen, da diese Schleifenform an einer Stelle alle wesentlichen Informationen zur Schleifenkontrolle zusammenzieht und so für den Leser übersichtlicher ist als die nachfolgend beschriebenen Schleifenformen.

Nun sind aber durchaus Fälle denkbar, in denen die Anzahl der Iterationen a priori nicht feststeht, sondern vom Verlauf der Schleife selbst abhängt. Ein bereits behandeltes Beispiel ist die Suche eines Telefonbucheintrages in einem nichtsortierten Telefonverzeichnis, indem man das Telefonverzeichnis von vorne nach hinten durchsucht, bis man den gewünschten Eintrag gefunden hat. Zu Beginn der Schleife steht also nicht fest, wie viele Vergleichsoperationen durchgeführt werden müssen, dies ergibt sich erst durch den Inhalt des Telefonbuchs und nach welchem Namen man darin sucht.

Benötigt wird also eine Notation für den bereits identifizierten Grundbaustein:

5-1	<b>while</b> Bedingung gilt <b>do</b>
5-2	Anweisung

Da die Schleifenbedingung zu Beginn (am Kopf) der Schleife steht, spricht man auch von einer **kopfgesteuerten Schleife**. Die Java-Syntax für solch eine kopfgesteuerte Schleife ist in Abbildung 5.9 zu sehen. Im *Schleifenkopf* wird die Bedingung angegeben, die den *Schleifenrumpf* in Form einer Anweisung kontrolliert. Die Ausführung solch einer Schleife geschieht, indem die Bedingung, die durch den logischen Ausdruck spezifiziert ist, ausgewertet wird. Ist der Wert des Ausdrucks `true`, so wird die Anweisung des Schleifenrumpfs ausgeführt und mit der Auswertung des Testausdrucks fortgefahrene. Dies wird solange wiederholt, bis der Wert des logischen Ausdrucks `false` ist. In diesem Fall bricht die Schleife ab und die Ausführung des Programms wird nach der Schleife fortgesetzt. Auch hier der Hinweis, dass syntaktisch genau eine Anweisung im Schleifenrumpf steht. Möchte man mehrere Anweisungen dort wiederholt ausführen, so muss ein Block gewählt werden, der syntaktisch genau diese eine Anweisung darstellt.

#### Beispiel 5.20:

Ein Beispiel für eine wiederholte Ausführung, bei der man nicht die Anzahl der Wiederholungen vorgeben kann, ist die Berechnung des größten gemeinsamen Teilers zweier positiver Zahlen  $x, y \in \mathbb{N}$ , wie dies schon in Kapitel 2.4 als Beispielprogramm gebracht wurde. Der ggT kann in etwas veränderter Form angegeben werden:

$$ggT(x, y) = \begin{cases} x & \text{falls } x = y \\ ggT(x - y, y) & \text{falls } x > y \\ ggT(x, y - x) & \text{falls } x < y \end{cases}$$

Für den Fall  $x = y$  bricht also die Berechnung ab und das Ergebnis ist  $x$  (oder  $y$ , das ja in dem Fall den gleichen Wert hat). Ansonsten können zwei Fälle auftreten: entweder ist  $x < y$  oder  $x > y$ . In beiden Fällen wird dann analog verfahren, indem das Verfahren / diese Berechnungsformel nochmals angewandt wird allerdings mit veränderten Werten für  $x$  beziehungsweise  $y$ . Überträgt man diese rekursive Definition entsprechend der gerade erfolgten Beschreibung, so erhält man ein iteratives Verfahren, bei dem zu Beginn *nicht* feststeht, wie

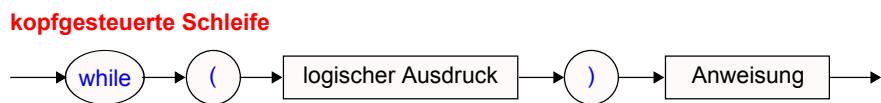


Figure 5.9: Syntaxdiagramm für kopfgesteuerte Schleifen

Listing 5.21: ggT-Berechnung.

```

5-1  /**
5-2   * groesster gemeinsamer Teiler ggT
5-3   */
5-4  public class ggT {
5-5      public static void main (String [] args) {
5-6          // Beispielwerte
5-7          int x = 43158;
5-8          int y = 26364;
5-9
5-10         // gebe die Werte von x und y aus
5-11         System.out.print ("Der ggT von " + x + " und " + y + " ist ");
5-12
5-13         if (x == 0) {
5-14             // gebe das Ergebnis / den ggT aus
5-15             System.out.println (y);
5-16         } else {
5-17             while (y != 0) {
5-18                 if (x > y) {
5-19                     x = x - y;
5-20                 } else {
5-21                     y = y - x;
5-22                 }
5-23             }
5-24             // das Verfahren brach ab und in x (und y) steht das Resultat
5-25             System.out.println (x);
5-26         }
5-27     }
5-28 }
```

oft die Iterationsvorschrift angewandt werden muss; das hängt von den Zahlen  $x$  und  $y$  ab. Das Programm zu diesem Verfahren ist in Listing 5.21 abgebildet. Als Ausgabe kommt bei diesem Beispielprogramm für  $x = 43158$  und  $y = 26364$ :

5.1 ggT von 43158 und 26364 ist 6

Vergleichen Sie die Definition der mathematischen Funktion mit dem Programm: exakt die gleichen Fälle werden im Programm wie in der Funktionsdefinition behandelt. Der Rekursion in der Funktionsdefinition entspricht im Programm die wiederholte Ausführung in Form einer kopfgesteuerten Schleife. Wie werden später noch behandeln, ob solche eine Umwandlung einer REkursion in eine Iteration grundsätzlich immer möglich ist. ♦

Wo Zählschleifen ebenfalls nicht genutzt werden können, sind numerische Näherungsverfahren zur näherungsweisen Bestimmung eines Wertes (im letzten Satz stand *näherungsweiser Wert* und nicht exakter Wert!). Ausgehend von einem Startwert kommt man in jedem Berechnungsschritt dem korrekten Wert näher. Man wiederholt den Berechnungsschritt für einen besseren Näherungswert solange, bis eine geforderte Genauigkeit erreicht ist, die man selber vorgeben kann. Der Unterschied zur ggT-Berechnung ist also, dass man nicht den exakten Wert bekommen muss (das würde eventuell viel zu lange dauern), sondern man möchte eine gute Annäherung an den exakten Wert in akzeptabler Zeit.

### Beispiel 5.21:

Das Verfahren von Heron zur Bestimmung der Quadratwurzel zu einer Zahl  $x \in \mathbb{R}, x > 0$  ist ein Beispiel für dieses Vorgehen. Ausgehend von einem beliebigen Startwert  $y_0 > 0$  gewinnt man eine bessere Näherung  $y_{n+1}$  an den gesuchten Wert  $\sqrt{x}$  über die Berechnungsvorschrift:

$$y_{n+1} = \frac{y_n + \frac{x}{y_n}}{2}$$

Diese Vorschrift besagt also, dass man aus dem Originalwert  $x$  und dem alten Annäherungswert  $y_n$  nach der angegebenen Vorschrift einen neuen Wert  $y_{n+1}$  berechnen kann. Dieses Verfahren *garantiert*, dass die Näherung an den korrekten Wert mit jedem Schritt besser wird, man kommt also dem korrekten Wert immer näher. Für eine vorgegebene Genauigkeit  $\epsilon$  muss man demnach solange diese Berechnungsvorschrift anwenden, bis  $|y_n^2 - x| < \epsilon$ . Oder anders ausgedrückt: wenn der Abstand zwischen dem Originalwert  $x$ , zu dem die Quadratwurzel bestimmt werden soll, und der aktuellen Annäherung  $y_n$  an den Wert  $\sqrt{x}$  zum Quadrat so klein wird, wie man es gerne hätte ( $\epsilon$  kann vorgegeben werden). Je kleiner man  $\epsilon$  wählt, umso genauer kann man werden, aber um so länger muss man auch rechnen (mehr Iterationen). Je größer man  $\epsilon$  vorgibt, umso kürzer wird man rechnen müssen (weniger Iterationen), umso ungenauer wird aber auch die Näherungslösung sein.

Die Wahl des Startwertes hat natürlich auch einen gewissen Einfluss darauf, wieviele Iterationen benötigt werden. Im beschriebenen Verfahren von Heron nähert man sich aber sehr schnell dem richtigen Wert an (gutes Konvergenzverhalten), so dass die Wahl des Startwertes weniger problematisch ist. Ein "guter" Startwert ist zum Beispiel die Zahl  $x$  selber, zu der man  $\sqrt{x}$  berechnen will. Die Ausgabe des Programms aus Listing 5.22 ist in Listing 5.23 angegeben.

Entwickler von solchen iterativen numerischen Verfahren haben nicht nur die Aufgabe, sich solch ein Verfahren für eine gegeben Problemstellung auszudenken, sondern sie müssen auch *beweisen*, dass für beliebige Eingabewerte erstens dieses Verfahren immer zum korrekten Ergebnis konvergiert, zweitens wie schnell es konvergiert und drittens wie große die Fehlerabweichung nach  $n$  Iterationen höchstens ist. Nur dann kann sich

Listing 5.22: Iterative Berechnung einer Quadratwurzel.

```

5-1  /**
5-2   * Heron Verfahren zur Bestimmung der Quadratwurzel
5-3   */
5-4  public class Heron {
5-5      public static void main(String [] args) {
5-6
5-7          // Beispielwert, zu dem die Quadratwurzel bestimmt werden soll
5-8          double x = 8351.0;
5-9          // gewuenschte Genauigkeit
5-10         double epsilon = 0.000001;
5-11         // Fehlerabschaetzung zum korrekten Wert
5-12         double fehler;
5-13         // derzeitiger Annaeherungswert y_n
5-14         double y;
5-15
5-16         // Startwert angeben
5-17         // Dies koennte ein beliebiger Wert sein. Probieren Sie es aus!
5-18         y = x;
5-19
5-20         // Absolutbetrag fuer Fehlerabschaetzung
5-21         fehler = Math.abs(x - y*y);
5-22
5-23         while(fehler > epsilon) {
5-24             // Berechnungsvorschrift fuer bessere Naehnung
5-25             y = (y + x/y) / 2.0;
5-26
5-27             // Absolutbetrag fuer Fehlerabschaetzung
5-28             fehler = Math.abs(x - y*y);
5-29
5-30             // wir geben zur Kontrolle die derzeitige Naehnung aus
5-31             System.out.println("y=" + y);
5-32         }
5-33     }
5-34 }
```

Listing 5.23: Ausgabe zur Berechnung der Quadratwurzel.

```

5-1 y=4176.0
5-2 y=2088.9998802681994
5-3 y=1046.4987435038079
5-4 y=527.2393430977079
5-5 y=271.53922469803194
5-6 y=151.14676459891066
5-7 y=103.1988495800834
5-8 y=92.06014714295677
5-9 y=91.38629045342645
5-10 y=91.38380603899313
5-11 y=91.38380600522174
```

ein Programmierer auf solch einen Algorithmus verlassen und hat auch Maße dafür, wie lange das Verfahren in einer bestimmten Problemstellung höchstens benötigt und wie groß der Fehler höchstens sein kann, mit dem er rechnen muss.

### 5.5.3 Fußgesteuerte Schleife

Im Beispiel des Heron-Verfahrens zur näherungsweisen Bestimmung der Quadratwurzel einer Zahl musste man zweimal den Programmcode zur Berechnung der Fehlerabschätzung (Absolutwert berechnen) angeben. Das lag daran, dass der Test zur Fehlerabschätzung zu Beginn der Schleife erfolgte. Einmal musste beim ersten Eintritt in die Schleife diese Berechnung vor der Schleife stattfinden und weiterhin musste innerhalb des Schleifenrumpfs diese Berechnung erfolgen, um den Test der nachfolgenden Iteration vorzubereiten. Man kann sich den doppelten Programmcode aber sparen, wenn der Abbruchtest erst nach Ausführung des Schleifenrumpfes erfolgen würde. Allerdings würde dann der Schleifenrumpf mindestens einmal durchlaufen, was bei der Variante mit dem Test zu Beginn der Schleife nicht notwendigerweise der Fall sein muss, nämlich wenn der Startwert bereits die Genauigkeitsanforderungen erfüllt.

Diese Schleifenvariante wurde bereits als allgemeinen algorithmischen Grundbaustein angegeben:

5-1	Wiederhole
5-2	Anweisung
5-3	Solange Bedingung gilt

Diese Schleifenform nennt man aufgrund des Kontrollausdrucks **fußgesteuerte Schleife** und hat in Java die Syntax, wie sie in Abbildung 5.10 gezeigt ist.

#### Beispiel 5.22:

Das Heron-Verfahren mit einer do-while-Schleife sieht dann wie in Listing 5.24 angegeben aus. Zu beachten ist gegenüber der while-Version, dass der Schleifenrumpf in jedem Fall mindestens einmal ausgeführt wird. ♦

Während in einer Zählschleife der Schleifenrumpf eine zu Beginn der Schleife feststehende Anzahl von Iterationen ausgeführt wird, hängt die Terminierung der kopf- und fußgesteuerten Schleife nur von der Testbedingung ab. Damit die Schleife irgendwann einmal terminiert, muss also sichergestellt werden, dass der Testausdruck irgendwann zu einem Wert ausgewertet wird (`false`), der die Schleife terminiert. Ist dies nicht der Fall – die Abbruchbedingung ist nie erfüllt – so spricht man von einer **Endlosschleife**.

#### Beispiel 5.23:

Ein Beispiel für die sinnvolle Anwendung einer Endlosschleife ist die Steuerung eines Herzschrittmachers, wie sie in einer (drastisch) vereinfachten Form in Listing 5.25 gezeigt ist. ♦

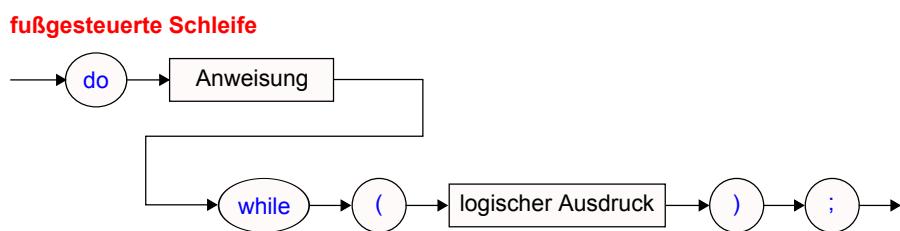


Figure 5.10: Syntaxdiagramm für fußgesteuerte Schleifen

Listing 5.24: Iterative Berechnung einer Quadratwurzel (Version 2).

```

5-1  /**
5-2   * Heron Verfahren zur Bestimmung der Quadratwurzel (Version 2)
5-3   */
5-4  public class Heron2 {
5-5      public static void main(String [] args) {
5-6
5-7          // Beispielwert, zu dem die Quadratwurzel bestimmt werden soll
5-8          double x = 8351.0;
5-9          // gewuenschte Genauigkeit
5-10         double epsilon = 0.000001;
5-11         // Fehlerabschaetzung zum korrekten Wert
5-12         double fehler;
5-13         // derzeitiger Annaeherungswert y_n
5-14         double y;
5-15
5-16         // Startwert angeben
5-17         // Dies koennte ein beliebiger Wert sein. Probieren Sie es aus!
5-18         y = x;
5-19
5-20         do {
5-21             // Berechnungsvorschrift fuer bessere Naehnung
5-22             y = (y + x/y) / 2.0;
5-23
5-24             // Absolutbetrag fuer Fehlerabschaetzung
5-25             fehler = Math.abs(x - y*y);
5-26
5-27             // wir geben zur Kontrolle die derzeitige Naehnung aus
5-28             System.out.println("y=" + y);
5-29
5-30         } while(fehler > epsilon);
5-31
5-32     }
5-33 }
```

Listing 5.25: Zentrale Schleife eines Herzschrittmachers.

```

5-1  /**
5-2   * Herzschrittmacher
5-3   */
5-4  public class Herzschrittmacher {
5-5
5-6      public static void main(String [] args) {
5-7
5-8          while (true) {
5-9              // Gebe Impuls
5-10             // Sammle relevante Messwerte
5-11             // Berechne Wartezeit
5-12             // Warte
5-13         }
5-14     }
5-15 }
```

### 5.5.4 Überführen von Schleifenformen

Kopf- und fußgesteuerte Schleifen lassen sich ineinander überführen. Eine Zählschleife lässt sich ebenfalls in eine kopf- und fußgesteuerte Schleife überführen, der umgekehrte Weg gilt allerdings nicht für die reinen Zählschleifen, wie sie hier eingeführt wurden, bei denen also bei Schleifenbeginn die Anzahl der Iterationen feststeht. Die for-Schleifen in Java und C/C++ sind allerdings allgemeiner als die hier vorgestellten Zählschleifen (eine Obermenge) und man kann jede while-Schleife in Java und C/C++ in eine äquivalente for-Schleife umwandeln.

Eine Zählschleife der Form

```
5-1   for ( Initialisierung ; logischer Ausdruck ; Aktualisierung )
5-2       Anweisung ;
```

lässt sich als kopfgesteuerte Schleife darstellen:

```
5-1   Initialisierung ;
5-2   while ( logischer Ausdruck ) {
5-3       Anweisung ;
5-4       Aktualisierung ;
5-5   }
```

Eine kopfgesteuerte Schleife der Form

```
5-1   while ( logischer Ausdruck )
5-2       Anweisung ;
```

lässt sich als fußgesteuerte Schleife darstellen:

```
5-1   if ( logischer Ausdruck ) {
5-2       do
5-3           Anweisung ;
5-4       while ( logischer Ausdruck );
5-5   }
```

und eine fußgesteuerte Schleife der Form

```
5-1   do
5-2       Anweisung ;
5-3   while ( logischer Ausdruck );
```

lässt sich als kopfgesteuerte Schleife darstellen:

```
5-1   Anweisung ;
5-2   while ( logischer Ausdruck )
5-3       Anweisung ;
```

#### Beispiel 5.24:

Das Programm aus Listing 5.26 kann in ein äquivalentes Programm umgeformt werden, das in Listing 5.27 zu sehen ist. ♦

Es sei angemerkt, dass ein entsprechend obiger Regeln umgeformtes Programm eventuell kleine Unterschiede hinsichtlich Semantik zum Ursprungssprogramm haben kann. Ein Beispiel ist die Deklaration einer Variablen im Initialisierungsausdruck einer for-Schleife. Wandelt man eine solche for-Schleife in eine while-Schleife um, so ist die Semantik leicht anders, da nun die Variable außerhalb der Schleife deklariert ist (Details zu den Konsequenzen in Kapitel 6).

Listing 5.26: Iterative Berechnung einer Quadratwurzel (do-while).

```

5-1  /**
5-2   * Heron Verfahren zur Bestimmung der Quadratwurzel (Version 2)
5-3   */
5-4 public class Heron2 {
5-5     public static void main(String[] args) {
5-6
5-7         // Beispielwert, zu dem die Quadratwurzel bestimmt werden soll
5-8         double x = 8351.0;
5-9         // gewuenschte Genauigkeit
5-10        double epsilon = 0.000001;
5-11        // Fehlerabschaetzung zum korrekten Wert
5-12        double fehler;
5-13        // derzeitiger Annaeherungswert y_n
5-14        double y;
5-15
5-16        // Startwert angeben
5-17        // Dies koennte ein beliebiger Wert sein. Probieren Sie es aus!
5-18        y = x;
5-19
5-20        do {
5-21            // Berechnungsvorschrift fuer bessere Naehnung
5-22            y = (y + x/y) / 2.0;
5-23
5-24            // Absolutbetrag fuer Fehlerabschaetzung
5-25            fehler = Math.abs(x - y*y);
5-26
5-27            // wir geben zur Kontrolle die derzeitige Naehnung aus
5-28            System.out.println("y=" + y);
5-29
5-30        } while(fehler > epsilon);
5-31    }
5-32 }
5-33 }
```

Listing 5.27: Iterative Berechnung einer Quadratwurzel (while).

```

5-1  /**
5-2   * Heron Verfahren zur Bestimmung der Quadratwurzel (Version 3)
5-3   */
5-4  public class Heron3 {
5-5      public static void main(String[] args) {
5-6
5-7          // Beispielwert, zu dem die Quadratwurzel bestimmt werden soll
5-8          double x = 8351.0;
5-9          // gewuenschte Genauigkeit
5-10         double epsilon = 0.000001;
5-11         // Fehlerabschaetzung zum korrekten Wert
5-12         double fehler;
5-13         // derzeitiger Annaeherungswert y_n
5-14         double y;
5-15
5-16         // Startwert angeben
5-17         // Dies koennte ein beliebiger Wert sein. Probieren Sie es aus!
5-18         y = x;
5-19
5-20         // Berechnungsvorschrift fuer bessere Naeherung
5-21         y = (y + x/y) / 2.0;
5-22
5-23         // Absolutbetrag fuer Fehlerabschaetzung
5-24         fehler = Math.abs(x - y*y);
5-25
5-26         // wir geben zur Kontrolle die derzeitige Naeherung aus
5-27         System.out.println("y=" + y);
5-28
5-29         while (fehler > epsilon) {
5-30             // Berechnungsvorschrift fuer bessere Naeherung
5-31             y = (y + x/y) / 2.0;
5-32
5-33             // Absolutbetrag fuer Fehlerabschaetzung
5-34             fehler = Math.abs(x - y*y);
5-35
5-36             // wir geben zur Kontrolle die derzeitige Annaeherung aus
5-37             System.out.println("y=" + y);
5-38         }
5-39     }
5-40 }
```

## 5.6 Abweichung von der normalen Schleifenausführung

Manchmal kann es günstig sein, eine Schleife schon frühzeitig abzubrechen, zum Beispiel, weil eine bestimmte Bedingung innerhalb des Schleifenrumpfs eingetreten ist. Eine entsprechende Abbruchbedingung könnte man in den Schleifenkontrollausdruck mit einbeziehen.

### Beispiel 5.25:

Eine Primzahl ist eine natürliche Zahl  $\geq 2$ , die nur durch die Zahl 1 und sich selber teilbar ist. Um alle Primzahlen bis zu einer Obergrenze  $n$  zu bestimmen, kann man sukzessive alle Zahlen  $i$  von 2 bis  $n$  darauf untersuchen, ob Sie durch eine Zahl  $j$  mit  $2 \leq j \leq n$  teilbar ist. Findet sich keine solche Zahl  $j$ , so ist die Zahl  $i$  eine Primzahl. Findet man aber auch nur eine Zahl  $j$ , die ein Teiler von  $i$  ist, so kann  $i$  keinen Primzahl mehr sein. Das Programm in Listing 5.28 setzt diese Idee um, indem die innere  $j$ -Schleife eine Abbruchbedingung aus zwei Teilbedingungen bekommt, die diese Idee umsetzt. ♦

Im vorangegangenen Beispiel muss an drei Stellen des Programms auf diese Abbruchbedingung eingegangen werden (siehe Vorkommen der Variablen `zuende`). Einfacher wäre es, diese Abbruchteilbedingung (Teiler gefunden) als Ausnahmefall zu behandeln und lediglich dies entsprechend im Schleifenrumpf angeben zu können. Dazu besitzt Java (und andere Sprachen) die `break`-Anweisung, deren Syntax in Abbildung 5.11 gegeben ist und die innerhalb einer Schleife (for, while, do-while) oder einer switch-Anweisung genutzt werden kann. Die Bedeutung ist wie folgt. Stößt man in der Ausführung auf eine `break`-Anweisung, so wird an der Stelle die innerste Schleife / innerste switch-Anweisung sofort abgebrochen und die Ausführung hinter dieser Schleife / switch-Anweisung fortgesetzt.

### Beispiel 5.26:

Das modifizierte Programm unter Nutzung einer `break`-Anweisung ist dann wie in Listing 5.29 zu sehen. ♦

### Beispiel 5.27:

Die Aufgabenstellung ist es, Zahlen einlesen und aufsummieren, bis die nächste eingelesene Zahl negativ ist. Die Lösung dazu ist in Listing 5.30 zu sehen. ♦

Weiterhin gibt es die `continue`-Anweisung, die nur im Zusammenhang mit einer Schleife genutzt werden kann. Die Ausführung einer `continue`-Anweisung bewirkt, dass die aktuelle Schleifeniteration sofort beendet wird und bei einer for-Schleife der Update-Teil des Schleifenkopfes gefolgt vom Schleifentest ausgeführt wird. Bei einer while- oder do-while-Schleife wird sofort der Schleifentest ausgeführt. Der Gemeinsameit einer `break`-

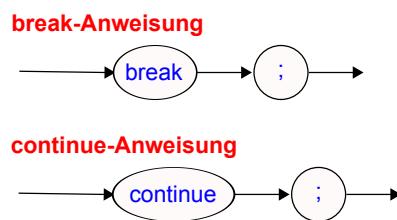


Figure 5.11: Syntaxdiagramme für die `break`- und `continue`-Anweisung

Listing 5.28: Berechnung von Primzahlen.

```

5.1  /**
5.2   * Primzahlen bestimmen
5.3   */
5.4  public class Primzahlen {
5.5
5.6      public static void main(String[] args) {
5.7          // Obergrenze als Programmargument
5.8          int n = Integer.parseInt(args[0]);
5.9
5.10         int i,j;
5.11         System.out.println("Primzahlen zwischen 2 und " + n + " sind: ");
5.12         for (i=2; i<n; i++) {
5.13             boolean zuende = false;
5.14             for (j=2; j<i && !zuende; j++) {
5.15                 if(i % j == 0) {
5.16                     // Teiler gefunden, es kann keine Primzahl mehr sein
5.17                     zuende = true;
5.18                 }
5.19             }
5.20             if(i == j) {
5.21                 // Primzahl gefunden
5.22                 System.out.print(" " + i);
5.23             }
5.24         }
5.25
5.26         // Zeile abschliessen
5.27         System.out.println();
5.28     }
5.29 }
```

Listing 5.29: Berechnung von Primzahlen (mit break).

```
5-1  /**
5-2   * Primzahlen bestimmen (mit break)
5-3   */
5-4  public class Primzahlen2 {
5-5
5-6      public static void main(String[] args) {
5-7          // Obergrenze als Programmargument
5-8          int n = Integer.parseInt(args[0]);
5-9
5-10         int i,j;
5-11         System.out.println("Primzahlen zwischen 2 und " + n + " sind: ");
5-12         for (i=2; i<n; i++) {
5-13             for (j=2; j<i; j++) {
5-14                 if(i % j == 0) {
5-15                     // Teiler gefunden, es kann keine Primzahl mehr sein
5-16                     break;
5-17                 }
5-18             }
5-19             if(i == j) {
5-20                 // Primzahl gefunden
5-21                 System.out.print(" " + i);
5-22             }
5-23         }
5-24
5-25         // Zeile abschliessen
5-26         System.out.println();
5-27     }
5-28 }
```

Listing 5.30: Einlesen und aufsummieren von Zahlen.

```

5-1  /**
5-2   * Lese Zahlen von Tastatur ein , bsi negative Zahl eingegeben wird
5-3   */
5-4  import java.util.*;
5-5
5-6  public class ZahlenEinlesen {
5-7      public static void main(String[] args) {
5-8
5-9          // Scanner von der Tastatur anlegen
5-10         Scanner sc = new Scanner(System.in);
5-11         int summe = 0;
5-12
5-13         System.out.println("Geben Sie ganze Zahlen ein. -1 beendet die Eingabe");
5-14         while(true) {
5-15             int zahl = sc.nextInt();
5-16             if(zahl < 0)
5-17                 break;
5-18             summe = summe + zahl;
5-19         }
5-20
5-21         // eingelesene Daten ausgeben
5-22         System.out.println("Die Summe der Werte ist: " + summe);
5-23
5-24         // Scanner abschliessen
5-25         sc.close();
5-26     }
5-27 }
```

und continue-Anweisung in einer Schleife ist also, dass die aktuelle Iteration in beiden Fällen abgebrochen wird. Der Unterschied innerhalb einer Schleife ist der, dass bei einer break-Anweisung die komplette Schleife beendet wird, eine continue-Anweisung aber bewirkt, mit der nächsten Iteration dieser Schleife fortzufahren.

### Beispiel 5.28:

Die Aufgabenstellung ist, genau 10 Zahlen einzulesen und deren Quadratwurzelwerte aufzusummen. Für negative Zahlen ist die Wurzelfunktion nicht definiert und solche Zahlen sollen einfach übersprungen werden. Das Programm in Listing 5.31 zeigt dieses Verhalten mit einer continue-Anweisung bei negativen Zahlen. Zu beachten ist, dass man in diesem einfachen und kurzen Beispiel auch mit einer if-Anweisung die negativen Zahlen hätte entsprechend behandeln können. ♦

Eine break-oder continue-Anweisung setzt man dann sinnvoll ein, wenn eine bestimmte Bedingung, die den Abbruch der Schleife beziehungsweise der aktuellen Iteration bedeutet, einen größeren Aufwand zur Behandlung bewirken würde. Wie im ersten Beispiel gezeigt, müsste man an drei Stellen diese Ausnahmefall behandeln, während die Nutzung der break-Anweisung diesen Fall auch als Ausnahmefall kennzeichnet. Als Hinweis sei hier angemerkt, dass sowohl die break- als auch die continue-Anweisung der Idee eines vollkommenen strukturierten Ablauf wiedersprechen, da sie quasi als Ausbruchsmechanismus den sofortigen Abbruch einer Schleife bewirken.

Java erlaubt weiterhin die Nutzung von Sprungmarken im Zusammenhang mit `continue`, `break` und `switch`, worauf hier aber nicht weiter eingegangen wird.

Listing 5.31: Einlesen und verarbeiten von 10 Zahlen.

```
5-1  /**
5-2   * Lese 10 nichtnegative Zahlen von Tastatur ein und berechen Summe der Wurzelwerte
5-3   */
5-4  import java.util.*;
5-5
5-6  public class ZahlenEinlesen2 {
5-7
5-8      public static void main(String[] args) {
5-9
5-10         // Scanner von der Tastatur anlegen
5-11         Scanner sc = new Scanner(System.in);
5-12         double summe = 0.0;
5-13
5-14         System.out.println("Geben Sie 10 ganze Zahlen ein.");
5-15         for(int i=0; i<10; i=i+1) {
5-16             double zahl = sc.nextDouble();
5-17             if(zahl < 0.0)
5-18                 continue;
5-19             summe = summe + Math.sqrt(zahl);
5-20         }
5-21
5-22         // eingelesene Daten ausgeben
5-23         System.out.println("Die Summe der Wurzelwerte ist: " + summe);
5-24
5-25         // Scanner abschliessen
5-26         sc.close();
5-27     }
5-28 }
```

## 5.7 Leere Anweisung

Die einfachste Form einer Anweisung ist die **leere Anweisung** (siehe Abbildung 5.12). Diese Anweisungsform bewirkt nichts! Dies ist sicherlich auf den ersten Blick irritierend. Der Sinn einer leeren Anweisung liegt aber darin, in den Situationen, in denen die Syntax genau eine Anweisung verlangt, diese geforderte Anweisung syntaktisch korrekt auch formuliert werden kann, ohne etwas Inhaltliches zu tun. Beispielhaft kann im Zusammenhang mit Schleifen eine solche Situation auftreten, wo im Schleifenrumpf genau eine Anweisung verlangt ist (siehe Kapitel 5.5 später für Details zu Schleifen).

### Beispiel 5.29:

Das Programm in Listing 5.32 ermittelt mit Hilfe einer Schleife (Details dazu gleich), welche Zahl beginnend von 1 die erste Zahl größer als 1000 ist, die sich durch Verdopplung ihrer Vorgängerzahlen ergibt. Die mathematische Formulierung des funktionalen Zusammenhangs wäre also:

$$f(x) := \begin{cases} x \cdot x & \text{falls } x \cdot x > 1000 \\ f(x \cdot x) & \text{sonst} \end{cases}$$

◆

## 5.8 Zusammenfassung und Hinweise

### Verstehen

Anweisungen dienen der Ablaufsteuerung eines Programms. Der zentrale Begriff dabei ist der Begriff der Anweisung. Es gibt einer überschaubare Anzahl unterschiedlicher Anweisungen.

### Kurz und knapp merken

- Alle erkannten algorithmischen Grundbausteine finden sich in Form von Anweisungen wieder.
- Eine Ausdrucksanweisung und als Spezialfall eine Zuweisung sind wichtige Einzelanweisungen.
- Aus mehreren Anweisungen lässt sich ein Block angeben, der syntaktisch wiederum genau eine Anweisung ist. Will man zum Beispiel in einem Schleifenrumpf mehr als eine Anweisung ausführen, so muss man diese Anweisungen in einem Block zusammenfassen.
- Es gibt drei verschiedene Formen der Iteration: Zählschleifen mit einer bei Schleifenbeginn festgelegten Anzahl an Iterationen und zwei Formen von Schleifen, wo über eine Abbruchbedingung erst während der Ausführung der Schleife über einen Abbruch entschieden werden kann.

**Leere Anweisung**

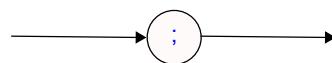


Figure 5.12: Syntaxdiagramm für die leere Anweisung

Listing 5.32: Beispiel zur leeren Anweisung.

```

5-1  /**
5-2   * ermittle die erste natuerliche Zahl groesser als 1000,
5-3   * die von 1 beginnend sich jeweils durch Verdopplung
5-4   * der Vorgaengerzahl ergibt
5-5   */
5-6 public class LeereAnweisung {
5-7     public static void main(String[] args) {
5-8       int zahl;
5-9
5-10      for(zahl=1; zahl<=1000; zahl=zahl+zahl)
5-11        // hier muss nichts mehr getan werden
5-12        ;
5-13
5-14      System.out.println("Die gesuchte Zahl ist " + zahl);
5-15    }
5-16 }
```

## Häufige Fehler

- Oft werden ausschließlich while-Schleife für jegliche Form der Iteration verwendet. Dort, wo die Anzahl der Iterationen zu Beginn der Schleife feststeht beziehungsweise dort berechnet werden kann, sollte auch eine Zählschleife verwenden werden, da diese Schleifenform in solchen Fällen weitaus übersichtlicher ist.
- Das Abbruchkriterium in Zählschleifen ist genau verkehrt herum angegeben. Eine Schleife wird solange wiederholt, wie das Kriterium *erfüllt* ist, also der Wert `true` vorliegt. Der Abbruch erfolgt, wenn der Testausdruck den Wert `false` hat.
- Bei Zählschleifen, in denen die Zählvariable heruntergezählt werden soll, ist entweder das Abbruchkriterium nicht entsprechend (Beispiel: `i < n`) oder der Aktualisierungsausdruck passt nicht (Beispiel: `i++`).
- Ein Semikolon ist eine leere Anweisung und wird auch als solche interpretiert. Siehe gebrachtes Beispiel in Listing 5.19. Manchmal wird ein Semikolon vergessen oder eines zuviel gesetzt, was in beiden Fällen die Bedeutung des Programms verändern kann.

## Übungsfragen

- Welche Anweisungsarten kennen Sie in Java?
- Was ist eine Ausdrucksanweisung? Semantik?
- Was ist ein Block? Besondere Eigenschaften / Vorteile?
- Was ist die Semantik der if-else Anweisung?
- Wie wird das Problem des dangling else aufgelöst?
- Was ist die Semantik der switch-Anweisung?
- Was ist die Semantik der for-Schleife?
- Formulieren Sie mit einer for-Schleife die Iteration über all ungeraden Zahlen von 1 bis 99!
- Formulieren Sie mit einer for-Schleife die Iteration über alle Zweierpotenzen bis 1000!
- Formulieren Sie mit einer for-Schleife die Iteration über alle durch 4 teilbaren Werte bis 1000!
- Formulieren Sie mit einer for-Schleife die Iteration von 40 bis -40 in umgekehrter Richtung!

- Geben Sie Java-Code an, um mit zwei geschachtelten Schleifen alle gültigen 2-stelligen Hexadezimalzahlen als String zu erzeugen (00, 01,...,FF)!
- Welche Schleifenformen lassen sich (automatisiert) in welche anderen Formen überführen? Wie?
- Welche Möglichkeiten gibt es, von der normalen Schleifenausführung abzuweichen?
- Wozu dient die leere Anweisung?

## Reflektion des Stoffs

- Kann man innerhalb eines Blocks einen weiteren Block verwenden? Wenn ja, wieso und wie? Wenn nein, wieso nicht?
- Wie kann man (mit den bis jetzt verfügbaren Mitteln) das Minimum und Maximum von drei Werten bestimmen?
- Wie müsste man den Kopf einer Zählschleife angeben, so dass man in der Zählvariablen die Werte 1,2,4,8,16,32,64,...,2n für ein gegebenes n generiert?
- Könnte man das Verfahren zur Berechnung einer Quadratwurzel (ohne das Verfahren an sich zu verändern) auch mit Hilfe einer Zählschleife angeben? Wenn ja, wie? Wenn nein, wieso nicht?
- Überlegen Sie sich selbst ein (praktisches) Beispiel, bei dem eine fußgesteuerte Schleife vorteilhaft wäre gegenüber einer kopfgesteuerten Schleife?
- Wie könnte man es erreichen, in einer zweifach geschachtelten Schleife aus dem innersten Schleifenrumpf heraus die gesamte Schleife abzubrechen?
- Mit while- und do-while-Schleifen sind auch Endlosschleifen möglich. Bei (richtig formulierten) Zählschleifen nicht.

# Chapter 6

## Variablen

In diesem Kapitel sollen Variablen nochmals ausführlicher als bisher betrachtet werden. Variablen haben einen Namen zur Identifikation und sind in der Lage zu einem Zeitpunkt genau einen Wert zu speichern. Variablen dienen in einem Programm dazu, einen berechneten Wert gegeben durch einen Ausdruck zwischenspeichern, der an späterer Stelle im Programmablauf benötigt wird. Mit Variablen lassen sich damit auch Mehrfachberechnungen vermeiden, wenn der Wert eines Ausdrucks mehrfach benötigt wird.

### Beispiel 6.1:

Im Beispiel in Listing 5.1, das schon früher vorgestellt wurde, findet die Umrechnung von 80 Grad Fahrenheit und 72 Grad Fahrenheit nach Celsius an zwei Stellen im Programm statt.

Speichert man den einmalig berechneten Celsiuswert, so kann man über die entsprechende Variable diesen Umrechnungswert nutzen, ohne ihn neu zu berechnen, wie in der modifizierten Programmversion in Listing 6.2 gezeigt. Als Ausgabe erscheint genauso wie vorher auf dem Bildschirm:

```
6-1 Fahrenheit: 80.0, Celsius: 26.66666666666668  
6-2 Fahrenheit: 72.0, Celsius: 22.22222222222222  
6-3 Die Differenz der beiden Werte ist 4.44444444444446 Grad Celsius
```

Listing 6.1: Umrechnung von Fahrenheit nach Celsius.

```

6-1 /**
6-2 * Umwandlung Grad Fahrenheit nach Grad Celsius
6-3 */
6-4 public class FahrenheitNachCelsius {
6-5     public static void main(String[] args) {
6-6
6-7         // Umrechnung 80 Grad Fahrenheit
6-8         System.out.println("Fahrenheit: " + 80.0 +
6-9             ", Celsius: " + 5.0 * (80.0 - 32.0) / 9.0);
6-10
6-11        // Umrechnung 72 Grad Fahrenheit
6-12        System.out.println("Fahrenheit: " + 72.0 +
6-13            ", Celsius: " + 5.0 * (72.0 - 32.0) / 9.0);
6-14
6-15        // Bestimmung der Differenz in Celsius
6-16        // (waere auch in Fahrenheit moeglich)
6-17        double diff = (5.0 * (80.0 - 32.0) / 9.0)
6-18            - (5.0 * (72.0 - 32.0) / 9.0);
6-19        System.out.println("Die Differenz der beiden Werte ist " + diff
6-20            + " Grad Celsius");
6-21    }
6-22 }

```

Listing 6.2: Umrechnung von Fahrenheit nach Celsius.

```

6-1 /**
6-2 * Umwandlung Grad Fahrenheit nach Grad Celsius
6-3 */
6-4 public class FahrenheitNachCelsius3 {
6-5     public static void main(String[] args) {
6-6
6-7         double diff, fahrenheit1, fahrenheit2, celsius1, celsius2;
6-8
6-9         // Konsistente Verwendung:
6-10        // an einer Stelle werden die Eingabewerte angegeben
6-11        fahrenheit1 = 80.0;
6-12        fahrenheit2 = 72.0;
6-13
6-14        // Umrechnung erster Wert
6-15        celsius1 = 5.0 * (fahrenheit1 - 32.0) / 9.0;
6-16        System.out.println("Fahrenheit: " + fahrenheit1 +
6-17            ", Celsius: " + celsius1);
6-18
6-19        // Umrechnung zweiter Wert
6-20        celsius2 = 5.0 * (fahrenheit2 - 32.0) / 9.0;
6-21        System.out.println("Fahrenheit: " + 72.0 +
6-22            ", Celsius: " + celsius2);
6-23
6-24        // Bestimmung der Differenz in Celsius
6-25        // (waere auch in Fahrenheit moeglich)
6-26        diff = celsius1 - celsius2;
6-27        System.out.println("Die Differenz der beiden Werte ist " + diff
6-28            + " Grad Celsius");
6-29    }
6-30 }

```

Jede Variable hat vier Kennzeichen:

1. **Variablenname:** Über ihren Namen ist jede Variable ansprechbar (referenzierbar). In den meisten Programmiersprachen ist ein Variablenname ein **Bezeichner** bestehend aus einem Buchstaben gefolgt von beliebig vielen Buchstaben oder Ziffern. Meist gilt auch das Zeichen \_ (Unterstrich) als ein Buchstabenersatz. Als Variablenbezeichner darf kein reserviertes Schlüsselwort genommen werden. Die *Java Code Conventions* geben an, dass Variablennamen mit einem Kleinbuchstaben beginnen und der erste Buchstabe eines Teilwortes groß geschrieben wird. Beispiel: `int anzahlZiegenImAuto;`. Der Variablenname muss eindeutig einer Variablen zugeordnet sein. Wie später zu sehen sein wird, kann es in verschiedenen Kontexten mehrere Variablen mit dem gleichen Namen geben, aber durch den Kontext ist jeweils sichergestellt welche Variable bei Verwendung des Namens jeweils gemeint ist.
2. **Typ:** Durch den Typ einer Variablen ist festgelegt, welche Werte die Variable annehmen kann und welche Operationen möglich sind. Weiterhin wird damit aber auch festgelegt, wie viele Bit zur Darstellung von Werten benötigt werden und wie diese Bit interpretiert werden, das heißt die Repräsentation von Werten. Jede Variable hat genau einen Typ, der in Form einer einer Deklaration angegeben wird. Beispiel: `int i;`
3. **Wert:** Mit jeder Variablen ist zu jedem Zeitpunkt immer genau einen Wert aus der Wertemenge des Typs assoziiert, der sich im Laufe der Programmabarbeitung aber verändern kann, was über eine Zuweisung möglich ist.
4. **Speicheradresse:** Der assozierte Wert einer Variablen muss jederzeit abrufbar sein, das heißt irgendwo gespeichert werden. Im von Neumann Rechnermodell geschieht dies im Hauptspeicher, wo vom Compiler zu jeder Variablen ausreichend Speicherplatz reserviert wird, der zur Aufnahme des aktuellen Wertes dieser Variablen dient. Dieser Speicherplatz hat eine eindeutige Hauptspeicheradresse.

Fasst man die obigen Punkte zusammen, so ist eine Variable in einer Programmiersprache deshalb nichts anderes als ein (zusammenhängender) Speicherbereich im Hauptspeicher des von Neumann Rechnermodells, der über den Variablennamen (typisiert) ansprechbar ist!

Die Inhalte aller Variablen zu einem bestimmten Stand im Programmablauf beschreiben den *Zustand*, in dem sich das Programm gerade befindet. Es wäre damit etwa prinzipiell möglich, den Inhalt aller Variablen zum Beispiel in eine Datei zu speichern und das Programm zu einem späteren Zeitpunkt an der gleichen Programmstelle und mit den gleichen Inhalten aller Variablen fortzusetzen.

Das Variablenkonzept ist für Programmiersprachen charakteristisch, die dem **imperativen Programmierparadigma** folgen. Nach diesem Paradigma ist ein Gesamtprogramm aus Einzelanweisungen zusammengesetzt, die den Hauptspeicher der von Neumann Architektur über Zuweisungen verändern (Zustandstransformation). Java ist eine Programmiersprache, die sowohl Programmstrukturen für das imperativ als auch für das objektorientierte Programmierparadigma (später dazu mehr) enthält.

Schaut man sich eine Zuweisung der Form

6-1      `a = 3 * a + 1;`

an, so sieht man, dass die Verwendung des Variablenbezeichners `a` auf der rechten Seite der Zuweisung – im Ausdruck – eine andere Bedeutung hat als auf der linken Seite der Zuweisung. Im Ausdruck auf der rechten Seite der Zuweisung bedeutet die Benutzung des Variablenamens `a`, dass der **Wert** der Variablen `a` genommen werden soll, auf der linken Seite der Zuweisung bedeutet die Verwendung von `a`, dass der **Speicherbereich** gemeint ist, in dem der neue Wert – der Wert des Ausdrucks – abgespeichert werden soll. Man kann sich den obigen Ausdruck also vorstellen als:

6-1      `speicher(a) = 3 * wert(a) + 1;`

## 6.1 Deklaration

Bekannt ist bereits, dass es Konstanten unterschiedlichen Typs gibt (`int`, `long`, `float`, ...), mit unterschiedlicher Wertemenge, unterschiedlicher interner Repräsentation von Werten der Wertemenge und auch unterschiedlichen Operationen auf den Werten. Ebenso haben Variablen genau einen unveränderbaren Typ, den man in Form einer **Deklaration** bekannt macht. Mit einer Variablen-deklaration sind verschiedene Dinge verbunden:

- Eine neue Variable wird mit dem spezifizierten Namen angelegt. Falls dieser Name bereits vorher für eine Variable verwendet wurde, so wäre dies ein Fehler. Später (Kapitel 6.3) wird auf bestimmte Situationen eingegangen, wo dies doch möglich ist.
- Für diese Variable wird in der Deklaration der Typ festgelegt, der nicht mehr veränderbar ist. Mit dem Typ ist gleichzeitig die mögliche Wertemenge und die interne Darstellung von Werten festgelegt.
- Basierend auf der Typangabe reserviert der Compiler für die neue Variable ausreichend großen Speicherplatz im Hauptspeicher. Dieser Speicherplatz ist ebenfalls fest mit der neuen Variablen verbunden und wird sich nicht mehr ändern.
- Wurde ein Initialisierungsausdruck in der Deklaration angegeben, so wird der Wert dieses Ausdrucks berechnet und in der Variablen gespeichert. Ansonsten ist der Wert der Variablen undefiniert! Undefiniert heißt, dass irgendein zufälliger Wert in der Variablen steht. Der Compiler erkennt den Fall, dass man eine nicht initialisierte Variable in einer späteren Berechnung in einem Ausdruck nutzen will, ohne dass es zu einer sinnvollen Zuweisung vorher gekommen ist. Später (Kapitel 11) wird gezeigt, dass in bestimmten Situationen der Wert einer bestimmten Variable auch ohne Initialisierungsausdruck sehr wohl definiert ist.

### Beispiel 6.2:

Mit einer Deklaration der Form

6-1

```
int x = 5+2;
```

geschieht also Folgendes:

1. Unter dem Bezeichner `x` ist ab dieser Stelle im Programm eine Variable bekannt.
2. Diese Variable hat aufgrund der Deklaration den Typ `int`, womit Wertebereich, mögliche Operationen und interne Repräsentation des Wertes festgelegt sind.
3. Der Compiler legt 4 Byte (32 Bit) Speicherplatz im Hauptspeicher an. Der Variablenname (und damit natürlich auch die Variable) wird mit diesem Speicherplatz assoziiert.
4. In diesem Speicherplatz wird der Wert der Ausdrucks `5+2`, also 7 in der durch den Typ vorgegebenen internen Repräsentation abgelegt. Im Beispiel also die 32 Bit Zweierkomplementdarstellung des Wertes 7. ◆

Bei den bisher behandelten Java-Konstrukten wurden zwei Formen von Variablen-deklarationen vorgestellt:

1. Die Deklaration innerhalb eines Blocks. In diesem Fall spricht man auch von einer *blocklokalen Variablen*. Beispiel siehe Listing 6.3.
2. Die Deklaration innerhalb einer for-Schleife. In diesem Fall spricht man von einer *schleifenlokalen Variablen*. Beispiel siehe Listing 6.4.

Listing 6.3: Blocklokale Variablen.

```

6-1 /**
6-2 * Beispiel fuer blocklokale Variablen
6-3 */
6-4 public class BlockLokal {
6-5
6-6     public static void main(String [] args) {
6-7
6-8         int x = 1;           // lokal zum main-Block
6-9
6-10        {
6-11            int y = 2;       // lokal zum inneren Block
6-12        }
6-13    }
6-14 }
```

Listing 6.4: Schleifenlokale Variable.

```

6-1 /**
6-2 * Beispiel fuer schleifenlokale Variable
6-3 */
6-4 public class SchleifenLokal {
6-5
6-6     public static void main(String [] args) {
6-7
6-8         for(int i=0; i<5; i++) {           // Schleife 1
6-9
6-10             for(int j=0; i<2; i++) {           // Schleife 2
6-11
6-12                 int k = i + j;
6-13                 System.out.println("Summe ist " + k);
6-14
6-15             } // Ende Schleife 2
6-16         } // Ende Schleife 1
6-17     }
6-18 }
```

Diese Unterscheidung ist für die nachfolgend eingeführten Begriffe Gültigkeit, Sichtbarkeit und Lebensdauer von Bedeutung. In Kapitel 8 und 11 werden auch weitere Variablenarten eingeführt, die auch außerhalb eines Blocks beziehungsweise einer for-Schleife deklariert werden können. In Anhang D werden dann alle Variablenarten mit einer entsprechenden Kategorisierung der nachfolgenden Begriffe aufgeführt.

### Beispiel 6.3:

Zu dieser Thematik gibt es eine vielzitierte Geschichte. In früheren Versionen der Programmiersprache Fortran, die ihre Ursprünge in den 50er Jahren des letzten Jahrhunderts hat, waren (und sind zum Teil noch bis heute) einige Besonderheiten:

1. Variablen müssen nicht explizit deklariert werden. Erscheint ein unbekannter Variablenbezeichner im Programmtext, so wird diese Variable implizit nach ihrem Anfangsbuchstaben mit einem bestimmten Typ deklariert: Fängt die Variable mit einem Buchstaben zwischen I und N an, so ist dies eine ganzzahlige Variable, ansonsten eine Fließkommavariablen.
2. Im Programmtext kann man beliebig Zwischenräume einfügen, auch innerhalb eines Variablenbezeichners.

1962 wurde eine Raumsonde im damaligen Wert von 18,5 Millionen Dollar in eine Erdumlaufbahn geschossen. Im Kontrollprogramm – in Fortran geschrieben – sollten Anweisungen stehen:

```
6-1 DO 9 J=1,3
6-2 ...
6-3 9 CONTINUE
```

was bedeutet, dass in einer Schleife die mit ... gekennzeichneten Anweisungen drei mal ausgeführt werden sollen. Dem Programmierer unterlief aber ein folgenschwerer Fehler. Er vertippte sich und gab statt des Kommas ein Punkt ein. Es stand also im Programm:

```
6-1 DO 9 J=1.3
6-2 ...
6-3 9 CONTINUE
```

Die Bedeutung änderte sich nun gegenüber dem eigentlich beabsichtigtem Programm. Der Fortran-Compiler eliminiert zuerst (korrekterweise) alle Leerzeichen: `DO9J=1.3` und erkennt dann (korrekterweise) eine Zuweisung der Form `DO9J = 1.3`. Entsprechend obiger Regeln deklariert der Compiler implizit dies als eine Fließkommavariablen mit Namen `DO9J` und weist dieser Variablen den Wert `1.3` zu. Die nachfolgende Anweisung wurde nur einmal und nicht dreimal ausgeführt. Der Effekt dieser kleinen Änderung war, dass die Raumsonde nach Erreichen ihrer Umlaufbahn nicht korrekt arbeitete und vernichtet werden musste. ♦

Hinweis: Seit Java 10 gibt es die sogennante *Local Variable Type Inference* für die Deklaration von lokalen Variablen (blocklokale Variablen und Variablen in for-Schleifen) zusammen mit einer Initialisierung. Durch die Ermittlung des Typs des Initialisierungsausdrucks ist der Compiler in der Lage, den daraus resultierenden Typ der Variablen zur Übersetzungszeit selbst zu erschließen. Anstatt einer expliziten Typangabe verwendet man `var`, um dies zu kennzeichnen.

### Beispiel 6.4:

Im Listing 6.5 wird die Variable `i` explizit mit einer Typangabe deklariert. In der nachfolgenden Zeile wird die Variable `j` über Typinterferenz automatisch mit dem Typ `int` deklariert, da die Zuweisung des initialen int-Wertes 58 diesen Typ festlegt. ♦

Listing 6.5: Variablen Deklaration mir var.

```

6-1 public class VarKlasse {
6-2     public static void main(String [] args) {
6-3         int i = 57;
6-4         var j = 58;
6-5
6-6         //var k;      // Fehler: Typ von k nicht herleitbar
6-7     }
6-8 }
```

Listing 6.6: Beispiel zum Gültigkeitsbereich blocklokaler Variablen.

```

6-1 /**
6-2 * Beispiel zum Gueltigkeitsbereich einer Variablen
6-3 */
6-4 public class Gueltigkeitsbereich {
6-5     public static void main(String [] args) { // Hier beginnt ein Block
6-6
6-7     double d1, d2;
6-8
6-9     d1 = 10.0;
6-10    d2 = 0.1;
6-11    System.out.println("d1 * d2: " + (d1 * d2));
6-12
6-13 } // Hier endet der Block
6-14 }
```

Verwenden Sie diese Möglichkeit *nicht im 1.Semester*, damit Ihnen mit jeder Deklaration selber bewusst ist, welcher Typ an dieser Stelle gefragt ist. Dieses Konstrukt wird interessant, wenn komplexe Typen genutzt werden (Stoff des 2.Semesters).

## 6.2 Gültigkeitsbereich

Jeder Bezeichner einer Variablen hat einen **Gültigkeitsbereich**, der durch den Deklarationsort im Programm festgelegt wird. Der Gültigkeitsbereich legt den Programmreich (also den Programmcode) fest, für den die Deklaration gültig ist und in dem diese Variable mit diesem einfachen Namen bezeichnet werden kann.

Für eine **blocklokale Variable** beginnt der Gültigkeitsbereich am Deklarationspunkt im Programm und endet am Ende des umschließenden Blocks. Er enthält auch alle inneren Blöcke dieses Blocks. Eine **schleifenlokale Variable** hat einen Gültigkeitsbereich, der die gesamte for-Schleife umfasst.

Außerhalb ihres Gültigkeitsbereichs ist eine Variable mit ihrem einfachen Namen nicht bekannt und eine Verwendung eines ungültigen Variablenamens würde vom Compiler als Fehler angezeigt. Nicht nur Variablen haben einen Gültigkeitsbereich, sondern – wie später zu sehen sein wird – auch Methoden und Klassen.

### Beispiel 6.5:

Gegeben ist das Programm in Listing 6.6. Die Variablen **d1**, **d2** haben einen Gültigkeitsbereich, der genau am Deklarationspunkt im Programm beginnt (exakt nach dem letzten Buchstaben des Variablenbezeichners in der Deklaration) und am Ende des umschließenden **Blocks** endet, der durch die geschweiften Klammern gegeben ist und im Beispiel durch den Kommentar **//Hier endet der Block** gekennzeichnet ist. ♦

Listing 6.7: Beispiel 2 zum Gültigkeitsbereich blocklokaler Variablen.

```

6-1  /**
6-2   * Beispiel zum Gueltigkeitsbereich mit geschachtelten Bloecken
6-3   */
6-4 public class Gueltigkeitsbereich2 {
6-5     public static void main(String[] args) { // Hier beginnt ein Block B1
6-6       double d0;
6-7
6-8       { // Hier beginnt ein innerer Block B2
6-9         double d1 = 1.0;
6-10        System.out.println("d1=" + d1);
6-11       } // Hier endet der Block B2
6-12
6-13       { // Hier beginnt ein weiterer innerer Block B3
6-14         double d2 = 2.0;
6-15         System.out.println("d2=" + d2);
6-16       } // Hier endet der Block B3
6-17
6-18     // Hier sind weder d1 noch d2 bekannt
6-19
6-20   } // Hier endet der Block B1
6-21 }
```

Wie bereits bekannt, können Blöcke ineinander geschachtelt werden, um zum Beispiel den Gültigkeitsbereich von Bezeichnern bewusst zu beeinflussen (einzuschränken).

### **Beispiel 6.6:**

Im Beispielprogramm in Listing 6.7 ist die Variable `d1` nur innerhalb des Blocks B2 bekannt, die Variable `d2` nur innerhalb von Block B3. Die Variable `d0` ist in Block B1 bekannt, der auch Block B2 und B3 umfasst. ♦

Durch den eingeschränkten Gültigkeitsbereich von Variablenbezeichnern ist eine weitere Frage interessant: Kann ich den gleichen Variablenbezeichner in verschiedenen nicht überlappenden Gültigkeitsbereichen mehrfach verwenden für *unterschiedliche* Variablen gleichen Namens? Die Antwort muss lauten: Ja!

### **Beispiel 6.7:**

Das Beispiel in Listing 6.8 erzeugt die Ausgabe:

```

6-1 d1=1.0
6-2 d1=2.0
```

Für schleifenlokale Variable ist wie bereits gesagt der Gültigkeitsbereich die gesamte for-Schleife.

### **Beispiel 6.8:**

Im Beispielprogramm in Listing 6.9 hat die Variable `i` der äußeren Schleife 1 einen Gültigkeitsbereich vom Deklarationspunkt im Schleifenkopf bis zum Ende der Schleife 1, also auch innerhalb der inneren Schleife 2. Die Variable `j` ist nur innerhalb der inneren Schleife gültig, also vom Deklarationspunkt `int j=0;` bis zur

Listing 6.8: Beispiel 3 zum Gültigkeitsbereich blocklokal Variablen.

```

6-1 /**
6-2 * Beispiel zum Gültigkeitsbereich mit Namenswiederverwendung
6-3 */
6-4 public class Gültigkeitsbereich3 {
6-5     public static void main(String [] args) { // Hier beginnt ein Block B1
6-6         double d0;
6-7
6-8         { // Hier beginnt ein innerer Block B2
6-9             double d1 = 1.0;
6-10             System.out.println("d1=" + d1);
6-11         } // Hier endet der Block B2
6-12
6-13         { // Hier beginnt ein weiterer innerer Block B3
6-14             double d1 = 2.0;
6-15             System.out.println("d1=" + d1);
6-16         } // Hier endet der Block B3
6-17
6-18         // Hier ist d1 nicht mehr bekannt
6-19
6-20     } // Hier endet der Block B1
6-21 }
```

Listing 6.9: Beispiel zum Gültigkeitsbereich schleifenlokaler Variablen.

```

6-1 /**
6-2 * Beispiel fuer schleifenlokale Variable
6-3 */
6-4 public class SchleifenLokal {
6-5
6-6     public static void main(String [] args) {
6-7
6-8         for(int i=0; i<5; i++) { // Schleife 1
6-9
6-10             for(int j=0; i<2; i++) { // Schleife 2
6-11
6-12                 int k = i + j;
6-13                 System.out.println("Summe ist " + k);
6-14
6-15             } // Ende Schleife 2
6-16         } // Ende Schleife 1
6-17     }
6-18 }
```

schließenden Klammer der Schleife 2. Die Variable `x` ist eine blocklokale Variable des Schleifenrumpfes der inneren j-Schleife.



## 6.3 Sichtbarkeit

Wie eben gesehen, ist die mehrfache Verwendung des gleichen Variablenbezeichners in nicht überlappenden Gültigkeitsbereichen möglich. Was aber ist, wenn die Gültigkeitsbereiche sich überlappen? Mit den bis jetzt bekannten Java Konstrukten ist dies nicht möglich, der Compiler würde einen Fehler melden (probieren Sie es selbst aus).

### Beispiel 6.9:

Nachfolgendes Java-Programm ist fehlerhaft (andere Sprachen erlauben dies).

```

6-1  /* Beispiel zur Sichtbarkeit, das fehlerhaft ist
6-2  */
6-3  public class Sichtbarkeit {
6-4
6-5      public static void main(String[] args) { // Beginn Block 1
6-6          int x; // erste Variable mit Namen x
6-7
6-8          { // Beginn Block 2
6-9              int x; // zweite Variable mit Namen x
6-10
6-11              // welches x?
6-12              System.out.println("x=" + x);
6-13
6-14      } // Ende Block 2
6-15
6-16  } // Ende Block 1
6-17 }
```

Der Compiler gibt folgende Fehlermeldung aus:

```

6-1 Sichtbarkeit.java:9: x is already defined in main(java.lang.String[])
6-2     int x; // zweite Variable mit Namen x
6-3             ^
6-4 1 error
```

In Java gibt es aber weitere Sprachkonstrukte, auf die später in Kapitel 11 eingegangen wird, die es erlauben, dass sich Gültigkeitsbereiche von Variablen mit gleichem Variablenbezeichner überlappen (es wird später verschiedene Kategorien von Variablen geben).

### Beispiel 6.10:

Als Ausgabe des Programms in Listing 6.10 erscheint:

```

6-1 1. Stelle d=2.0
6-2 2. Stelle d=1.0
```

Später wird in einem anderen Zusammenhang eingeführt, dass der Gültigkeitsbereich der Variablen 1 mit Namen `a` die gesamte Klasse / das gesamte Programm ist. Der Gültigkeitsbereich der Variablen 2 ebenfalls mit

Listing 6.10: Beispiel zur Sichtbarkeit von Variablen.

```

6-1  /**
6-2   * Beispiel zur Sichtbarkeit bei ueberlappenden Gueltigkeitsbereichen
6-3   */
6-4  public class Sichtbarkeit2 {
6-5
6-6      static double d = 1.0;    // Variable 1 mit Namen d
6-7
6-8      public static void main(String [] args) {
6-9          {
6-10              double d = 2.0;    // Variable 2 mit Namen d
6-11
6-12              System.out.println("1. Stelle d=" + d);
6-13          }
6-14
6-15          System.out.println("2. Stelle d=" + d);
6-16      }
6-17 }

```

Namen `d` ist wie eben gelernt der Block von `main`. Insofern gibt es eine Überschneidung der Gültigkeitsbereiche und damit die Frage, wie damit umgegangen wird. ♦

Die Antwort auf die Frage nach der Behandlung überlappender Gültigkeitsbereiche ist ein weiteres Konzept in Programmiersprachen genannt *Sichtbarkeit*. Gibt es zu einer Variablen keinen überlappenden Gültigkeitsbereich mit einer anderen Variablen gleichen Namens, so fallen Gültigkeitsbereich und Sichtbarkeitsbereich zusammen. Gibt es jedoch einen überlappenden Gültigkeitsbereich, so wird die Sichtbarkeit einer der beiden Variablen eingeschränkt. Dies bedeutet, dass die Variable zwar in diesem Bereich nach wie vor gültig ist, aber von einer anderen Variablen überdeckt wird, also quasi versteckt wird. Mit dem Variablenbezeichner alleine kann man die nicht-sichtbare Variable in diesem Überlappungsbereich nicht mehr ansprechen. Welche Variable sichtbar bleibt und welche Variable für einen Teil ihres Gültigkeitsbereiches unsichtbar wird entscheidet die Regel, dass die innere Variable (aus Sicht einer Blockschachtelung) sichtbar bleibt und die äußere Variable teilweise unsichtbar.

Im obigen Beispiel bedeutet dies, dass der Gültigkeitsbereich der ersten Variablen `d` zwar das gesamte Programm ist, aber diese Variable mit ihrem einfachen Namen `d` in dem Teil unsichtbar ist, in dem die zweite Variable `d` ihren Gültigkeitsbereich hat. Verwendet man in dem `main`-Block den Namen `d`, so ist damit durch die Sichtbarkeitsregel eindeutig das innere `d` des `main`-Blocks gemeint.

## 6.4 Lebensdauer

Gerade in Bezug auf Blöcke und den darin deklarierten Variablen taucht die Frage auf, ob erstens eine Variable weiter existiert, wenn der Block in der Ausführung verlassen wird oder zweitens, ob eine Blockvariable jedes mal neu geschaffen wird, wenn man während der Ausführung des Programms in den Block eintritt und die Variable wieder gelöscht wird, wenn der Block verlassen wird.

Beide Ansätze sind in Java (und anderen Sprachen) möglich und werden durch einen zusätzlichen Schlüsselwort bei der Deklaration gesteuert. Für blocklokale und schleifenlokale Variablen ist in Java allerdings nur die Variante möglich, dass die Variable neu erzeugt wird, wenn in der Programmausführung der Block / die Schleife begonnen wird. Die Variable wird wieder gelöscht, wenn in der Programmausführung der Block / die

Listing 6.11: Beispiel zur Lebensdauer von Variablen.

```

6-1  /**
6-2   * Beispiel zur Lebensdauer
6-3   */
6-4  public class Lebensdauer {
6-5      public static void main(String [] args) {
6-6
6-7          for (int i=1; i < 5; i++) {
6-8              // hier beginnt ein Block mit einer Variablen j
6-9              int j;
6-10             j = i;
6-11         }
6-12     }
6-13 }
```

Schleife verlassen wird. Später wird auch im Zusammenhang mit Klassen (Kapitel 11) die zweite Möglichkeit besprochen.

### Beispiel 6.11:

Mit jedem Eintritt in den Block in Listing 6.11, das heißt insgesamt vier mal, wird eine neue Variable mit Namen j geschaffen, die mit Verlassen des Blocks – am Ende jeder Iteration – wieder gelöscht wird. ♦

Den Zeitraum in der Programmausführung zwischen der Erzeugung einer Variablen (oder später auch Objektes) und ihrem Vernichten nennt man die *Lebensdauer* der Variablen (oder des Objektes). Während der Lebensdauer einer Variablen ist der reservierte Speicherplatz an diese Variable gebunden.

## 6.5 Zusammenfassung und Hinweise

### Verstehen

Sie sollten verstanden haben, wie Variablen deklariert werden, welche Konzepte mit ihnen verbunden sind (Typ, Gültigkeitsbereich, Sichtbarkeit, Lebensdauer) und wozu Variablen in einem Programm genutzt werden.

### Kurz und knapp merken

- Der *Gültigkeitsbereich* einer Variablen gibt an, in welchem Bereich des Programms eine Variable mit ihrem einfachen Namen angesprochen werden kann.
- Die *Sichtbarkeit* ist von Bedeutung, wenn die Gültigkeitsbereich zweier namensgleicher Variablen nicht überschneidungsfrei sind. Die innere / nächste Variable verdeckt in einem solchen Fall die äußere Variable.
- Die *Lebensdauer* einer Variablen ist der Zeitraum vom Erzeugen des mit dieser variablen assoziierten Speichers bis zum Löschen dieses Speichers. Für blocklokale Variablen ist dies die Ausführungszeit des Blocks, für schleifenlokale Variablen ist dies die Ausführungszeit der Schleife.

## Häufige Fehler

- Viele kompliziertere Sachverhalte in der (Java-)Programmierung lassen sich einfach mit den in diesem Kapitel eingeführten Begriffen herleiten. Erfahrungsgemäß werden diese Begriffe aber sehr schnell wieder vergessen beziehungsweise deren Bedeutung unterschätzt.

## Übungsfragen

- Welche Eigenschaften haben Variablen, die durch eine Deklaration vergeben werden?
- Was versteht man unter einem Gültigkeitsbereich? Sichtbarkeit?
- Was versteht man unter der Lebensdauer einer Variablen?
- Geben Sie ein Beispiel an, dass der Gültigkeitsbereich nicht gleich der Sichtbarkeit ist!

## Reflektion des Stoffs

- Ist es mit den derzeit bekannten Java-Konstrukten möglich, eine Variable zu deklarieren, die einen Gültigkeitsbereich über das gesamte Programm hat (vom ersten bis zum letzten Zeichen des Programmcodes)?



# Chapter 7

## Ausdrücke

Ausdrücke wurden bereits an mehreren Stellen vorgestellt und genutzt. In diesem Kapitel werden einige Besonderheit im Zusammenhang mit Ausdrücken behandelt. Dazu zählt, dass in Java, anders als in vielen anderen Programmiersprachen, die Auswertung von Ausdrücken (aus guten Gründen) strengen Regeln unterliegt, wozu verschiedene Aspekte beitragen. Weiterhin wird der wichtige aber auch schwierige Komplex der Typumwandlungen besprochen, also ob und wie Werte eines Datentyps auch in einem anderen Datentyp sinnvoll darstellbar sind (Beispiel `int ↔ float`).

Das Syntaxdiagramm zu einem Ausdruck ist in Abbildung 7.1 gegeben.

### 7.1 Wert eines Ausdrucks

Bereits mehrfach wurden arithmetische Ausdrücke behandelt, und zwar:

1. In Kapitel 3.2.1 die Syntax von arithmetischen Ausdrücken, definiert durch Angabe von Produktionen einer Grammatik.
2. Die Umwandlung von arithmetischen Ausdrücken in verschiedene äquivalente Darstellungen: Binärbaum, Präfix-, Infix- und Postfixnotation (Kapitel 10.3).
3. Die Auswertung von arithmetischen Ausdrücken durch die Funktion `eval`, die über der Struktur von Binärbäumen definiert war (Kapitel 10.3) und die Auswertung von arithmetischen Ausdrücken in

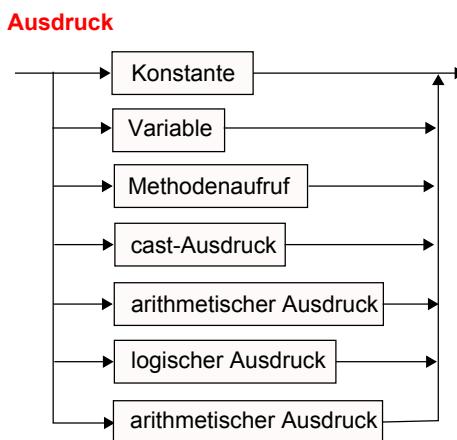


Figure 7.1: Syntax zu einem Ausdruck

Postfixnotation mit Hilfe eines Stacks (Kapitel 10.4).

Rechnet man einen arithmetischen Ausdruck aus, wie man dies zum Beispiel über die Funktion *eval* machen kann, so bekommt man als Ergebnis eine Zahl, den Wert des Ausdrucks. Jedem Ausdruck lässt sich also ein Wert zuordnen, der sich durch das Ausrechnen des Ausdrucks ergibt.

### Beispiel 7.1:

Der Wert des Ausdrucks **3** ist 3. Der Wert des Ausdrucks **3+4** ist 7. Der Wert des Ausdrucks **(3+4)\*5** ist 35. ♦

In einem Ausdruck werden genauso allgemein Operanden mit Hilfe von Operatoren verknüpft. Die Zahl der Operanden zu einem Operator ergibt sich aus der Stelligkeit des Operators. Als Operanden können selber wieder Teilausdrücke auftreten. Letztendlich kann ein Ausdruck aber neben Operatoren heruntergebrochen werden auf Konstanten / Literale und Variablen. Für eine Konstante ist der Wert der direkt damit verbundene Konstantenwert, für Variablen wird jeweils der aktuelle Inhalt dieser Variablen (was natürlich auch ein Wert ist) ermittelt und für die Berechnung genommen.

### Beispiel 7.2:

Im folgenden Programmausschnitt wird dieses Vorgehen nochmals explizit dargestellt. In der zweiten Anweisung (**int b = ...**) wird der Ausdruck auf der rechten Seite des Gleichheitszeichens wie folgt ausgewertet: zuerst wird der aktuelle Wert der Variablen **a** ermittelt. Dann wird zu diesem Wert der Wert 5 addiert (der Wert der Konstanten 5). Anschließend wird wieder der aktuelle Wert der Variablen **a** ermittelt und die eben ermittelte Summe durch diesen Wert dividiert, was den Wert des Gesamtausdrucks ergibt.

```
7-1   int a = 5;
7-2   int b = (a + 5) / a;
```

Neben den arithmetischen Ausdrücken, die bei ihrer Auswertung eine Zahl ergeben, sind aber auch Ausdrücke anderer Typisierung möglich. So etwa Ausdrücke zu Wahrheitswerten in der bereits besprochenen Form in Java (Beispiel: **true && false** ergibt den boolschen Wert **false**), oder aber Ausdrücke über Zeichenfolgen, wie sie ebenso bereits vorher diskutiert wurden. Neben dem Wert eines Ausdrucks ist also auch der **Typ** des Ausdrucks durch die im Ausdruck angewandten Operationen und Operanden bestimmt.

### Beispiel 7.3:

1. Der Java-Ausdruck **3+4** ist vom Java-Typ **int**. 3 und 4 sind Konstanten vom Typ **int** und der Infixoperator **+** liefert für zwei int-Argumente ein int-Resultat. Damit ist der Wert des Gesamtausdrucks also vom Typ **int**.
2. Der Java-Ausdruck **false && true** ist vom Typ **boolean**. **false** und **true** sind jeweils vom Typ **boolean** und der Infixoperator **&&** liefert angewandt auf zwei boolean-Argumente ein boolean-Resultat.
3. Der Ausdruck **3 + true** ist in Java nicht erlaubt, weil der Infixoperator **+** für einen linken Operanden vom Typ **int** und einen rechten Operanden vom Typ **boolean** nicht definiert ist. ♦

Listing 7.1: Umwandlung von Fahrenheit nach Celsius.

```

7-1 /**
7-2 * Umwandlung Grad Fahrenheit nach Grad Celsius
7-3 */
7-4 public class FahrenheitNachCelsius {
7-5     public static void main(String [] args) {
7-6
7-7         // Umrechnung 80 Grad Fahrenheit
7-8         System.out.println("Fahrenheit: " + 80.0 +
7-9             ", Celsius: " + 5.0 * (80.0 - 32.0) / 9.0);
7-10
7-11        // Umrechnung 72 Grad Fahrenheit
7-12        System.out.println("Fahrenheit: " + 72.0 +
7-13            ", Celsius: " + 5.0 * (72.0 - 32.0) / 9.0);
7-14
7-15        // Bestimmung der Differenz in Celsius
7-16        // (waere auch in Fahrenheit moeglich)
7-17        double diff = (5.0 * (80.0 - 32.0) / 9.0)
7-18            - (5.0 * (72.0 - 32.0) / 9.0);
7-19        System.out.println("Die Differenz der beiden Werte ist " + diff
7-20            + " Grad Celsius");
7-21    }
7-22}

```

In der Beschreibung der ganzzahligen und Fließkommawerte in Java sind ja schon einige Operationen mit Zahlenwerten vorgestellt worden. Anhand nachfolgenden Beispiels werden einige Aspekte von Ausdrücken diskutiert.

#### Beispiel 7.4:

In einigen Ländern (zum Beispiel in den USA) werden nach wie vor im Alltag die Temperatur in Grad Fahrenheit statt in Grad Celsius angegeben. Möchte man in Grad Celsius rechnen, so muss man die Temperaturangabe  $f$  in Fahrenheit jeweils umrechnen in einen Wert  $c$  in Grad Celsius, was über die Formel

$$c = 5 \cdot (f - 32) / 9$$

erreicht werden kann. Das Programm in Listing 7.1 gibt die Umrechnung von 80 und 72 Grad Fahrenheit sowie die Differenz in Celsius an. Als Ausgabe erscheint auf dem Bildschirm:

```

7-1 Fahrenheit: 80.0, Celsius: 26.66666666666668
7-2 Fahrenheit: 72.0, Celsius: 22.22222222222222
7-3 Die Differenz der beiden Werte ist 4.44444444444446 Grad Celsius

```



In dem Beispiel ist zu sehen, dass der Wert eines Ausdrucks (zum Beispiel der Wert für die Umrechnungsformel für 72 Grad Fahrenheit) "verbraucht" wird, also nur an dieser Stelle temporär zur Verfügung steht. Man benötigt den Celsius-Wert für 72 Grad Fahrenheit aber zweimal (in der Ausgabe und in der Differenzbildung), genauso den Wert für 80 Grad Fahrenheit. Im Beispiel wurde dann jeweils explizit die Umrechnungsformel als Ausdruck notiert. Dieses Vorgehen ist unübersichtlich, aufwändig und mit unnötiger Berechnung behaftet. Ebenso kann es zu **Inkonsistenzen** führen, wenn man statt 80 Grad Fahrenheit 82 Grad Fahrenheit nehmen

### Bedingungsausdruck

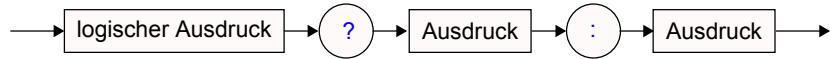


Figure 7.2: Syntaxdiagramm für einen Bedingungsausdruck

möchte und aus Versehen nur an einer Stelle und nicht an beiden Stellen, an denen dieser Wert genutzt wird, die entsprechende Änderung vornehmen würde.

Ein Hilfsmittel ist aber bereits bekannt, mit dem Werte in Programmiersprachen gespeichert werden können: Variablen. Ebenso wird später in Kapitel 8 noch ein Hilfsmittel eingeführt, mit dem sich die Berechnung eines Ausdrucks kapseln lässt, um so die Formel nur einmal notieren zu müssen, aber beliebig oft auswerten zu können: Funktionen oder wie sie in Java heißen: Methoden.

## 7.2 Weitere Operatoren

Operatoren sind die Verknüpfungselemente in einem Ausdruck, die zusammen mit den Operanden für die Berechnung eines Wertes stehen. Operatoren lassen sich allgemein klassifizieren:

- bezüglich ihrer Stelligkeit (auf wieviele Operanden sie angewandt werden): es gibt einstellige oder unäre Operatoren (Beispiel: `+3`), zweistellige oder binäre Operatoren (Beispiel: `3 + 4`) und dreistellige oder ternäre Operatoren (`(3 < 4) ? 5 : 6`)
- bezüglich ihrer Stellung zu den Operanden: Prefixoperatoren stehen vor den Operanden (Beispiel: `+3`), Infixoperatoren stehen zwischen den Argumenten (Beispiel: `3 + 4`) und Postfixoperatoren stehen nach den Operanden (Beispiel: `a++`)

Neben den bereits vorgestellten Operatoren gibt es in Java weitere Operatoren, die von ihrer Schreieweise und Wirkungsweise etwas aus dem bekannten Rahmen fallen und deshalb erst hier eingeführt werden.

Der erste dieser Operatoren ist der dreistellige Bedingungsoperator `? :`, der die Angabe einer Selektion in Form eines Ausdruck erlaubt. Die Syntax ist in Abbildung 7.2 gegeben. Die Bedeutung ist wie folgt: erst wird der logische Ausdruck vor dem Fragezeichen ausgewertet. Ergibt die Auswertung dieses Ausdrucks den Wert `true`, so wird der Ausdruck nach dem Fragezeichen ausgewertet; dieses dort gewonnene Resultat ist dann auch das Resultat des Gesamtausdrucks. Ansonsten wird der Ausdruck nach dem Doppelpunkt ausgewertet; analog ist das Resultat des Gesamtausdrucks dann der Wert dieses Ausdrucks. Der Vorteil dieses Operators ist der, dass man eine Selektion in Form eines Ausdrucks notieren kann, also beispielsweise in größeren Ausdrücken verwenden kann.

### Beispiel 7.5:

Um den größeren von zwei Werten zu bestimmen, kann man neben einer if-Anweisung auch den Bedingungsoperator verwenden. Listing 7.2 zeigt ein Beispiel zur Verwendung in diesem Zusammenhang.

Weitere etwas aus dem üblichen Rahmen fallende (und von der Mehrzahl der Programmieranfänger oft falsch genutzte) Operatoren sind die einstelligen Inkrement- und Dekrement-Operatoren `++` und `--`, die man vor oder nach einem syntaktisch eingeschränkten Ausdruck schreiben kann (siehe Abbildung 7.3). Inkrementieren heißt das Erhöhen eines Wertes um eins, dekrementieren das Erniedrigen eines Wertes um eins. Syntaktisch muss

Listing 7.2: Beispiel zur Nutzung des Bedingungsoperators.

```

7-1 int a=1, b=2, max;
7-2
7-3 // ueber eine Selektionsanweisung (dazu spaeter Details)
7-4 if(a > b)
7-5     max = a;
7-6 else
7-7     max = b;
7-8
7-9 // ueber einen Bedingungsausdruck
7-10 max = (a > b) ? a : b;

```

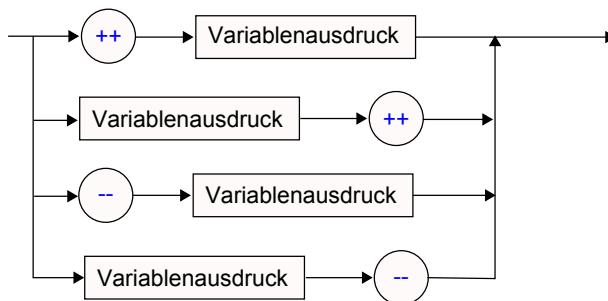
**Inkrement-/Dekrementausdruck**

Figure 7.3: Syntaxdiagramm für Inkrement- und Dekrementausdrücke

der Operand ein Variablenausdruck sein (also etwas, das eine Speicherstelle bezeichnet wie ein Variablenname). Der Operator kann rechts oder links von diesem Operanden stehen, was auf die Semantik des Ausdrucks Auswirkungen hat. Die Bedeutung ist nun wie folgt: Der Wert des Variablenausdrucks wird berechnet (also der Inhalt der Variablen bestimmt), dieser Wert wird je nach Operator um eins hoch- oder runtergezählt, und anschließend wird dieser neue gewonnene Wert wieder in die Variable gespeichert. Da dies ein Ausdruck ist ist die Frage, was der Wert des Gesamtausdrucks sein soll. Das hängt nun von der Stellung des Operators ab: steht der Operator vor dem Operanden (Beispiel: `++a`), so ist der Wert des Gesamtausdrucks der Wert der Variablen nach der Modifikation. Steht der Operator nach dem Operanden (Beispiel: `a++`), so ist der Wert des Gesamtausdrucks der Wert der Variablen vor der Operation. Auf jeden Fall wird also die Variable verändert ( $+/- 1$ ), der Wert des Gesamtausdrucks kann aber der alte oder neue Wert der Variablen sein, je nach Stellung des Operators zum Operanden.

**Beispiel 7.6:**

Listing 7.3 zeigt Beispiele zur Nutzung der Inkrement-/Dekrementoperatoren.

Weiterhin gibt es in Java die Zuweisungsoperatoren. Das sind zusammengesetzte Operatoren, die in verkürzenden Scheibweise eine Operation und anschließend eine Zuweisung mit einer Variablen syntaktisch als eine Operation zusammenfassen. Die möglichen Operatoren sind in Abbildung 7.4 angegeben. Ein Spezialfall ist das Zuweisungszeichen `=` alleine.

Listing 7.3: Beispiele zur Nutzung von Inkrement-/Dekrementoperatoren.

```

7-1 int a ,b ;
7-2
7-3 a = 1;
7-4 b = ++a; // a hat anschliessend den Wert 2
7-5 // b hat anschliessend den Wert 2
7-6
7-7 a = 1;
7-8 b = a++; // a hat anschliessend den Wert 2
7-9 // b hat anschliessend den Wert 1
7-10
7-11 a = 1;
7-12 b = --a; // a hat anschliessend den Wert 0
7-13 // b hat anschliessend den Wert 0
7-14
7-15 a = 1;
7-16 b = a--; // a hat anschliessend den Wert 0
7-17 // b hat anschliessend den Wert 1

```

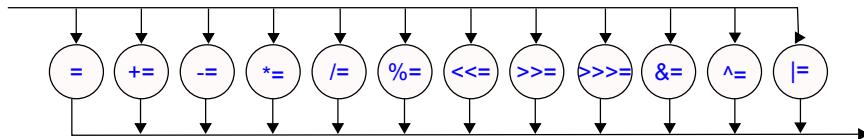
**Zuweisungsausdruck****Zuweisungsoperator**

Figure 7.4: Syntaxdiagramm eines Zuweisungsausdrucks

Listing 7.4: Beispiele zur Nutzung von Zuweisungsoperatoren.

```

7-1 int a, b;
7-2
7-3 a = 1;
7-4 b = 2;
7-5 a = a + b; // in a steht anschliessend 3
7-6
7-7 a = 1;
7-8 b = 2;
7-9 a += b; // in a steht anschliessend 3

```

**Beispiel 7.7:**

Im Beispielprogramm in Listing 7.4 sind die beiden durch Kommentare markierten Anweisungezeilen äquivalent.

Zu beachten ist die Priorität in Zuweisungsausdrücken; die Zuweisung selber hat eine sehr niedrige Priorität.

**Beispiel 7.8:**

In dem nachfolgenden Beispielcode wird zuerst `b+c` ausgewertet. Anschließend wird der Wert von `a` mit diesen Wert multipliziert und in `a` gespeichert. Die eigentlich höhere Priorität des Multiplikationsoperators spielt hier also keine Rolle.

```

7-1 int a = 1, b=2, c=3;
7-2 a *= b + c;

```

## 7.3 Überladen von Operatoren

In früheren Beispielen wurde gezeigt, dass das gleiche Operatorsymbol in unterschiedlichen Kontexten und mit unterschiedlicher Bedeutung verwandt werden konnte. Ein gutes Beispiel dazu ist das Plus-Symbol `+`, das in mehreren Bedeutungen existiert:

- Als Vorzeichenplus. Beispiel: `int x = +3;`
- Als Additionssymbol bei ganzen Zahlen. Beispiel: `int a,b,c; a = b + c;`
- Als Additionssymbol bei Fließkommazahlen. Beispiel: `float a,b,c; a = b + c;`
- Als Symbol für die Verkettung von Strings. Beispiel: `String s = "abc"+ "xyz";`

Ein anderes Beispiel ist das Symbol `&`, das sowohl für das bitweise Und als auch für das logisches Und steht. Sind mit einem Operatorsymbol mehrere Bedeutungen (Addieren, Verketten von Strings) oder mehrere Nutzungsmöglichkeiten mit gleicher Bedeutung (Addieren mit `int` und `float`) verbunden, so spricht man von einem *überladenen Operator*. Die konkrete Bedeutung des Operatorsymbols ergibt sich dann aus dem Zusammenhang (Kontext), oft durch die Anzahl oder den Typ der Operanden.

## 7.4 Priorität und Assoziativität von Operatoren

Ein weiterer Punkt im Zusammenhang mit Ausdrücken beziehungsweise den darin genutzten Operatoren ist die **Priorität** von Operatoren für den Fall, dass in einem Ausdruck mehr als ein Operator angewandt wird. Aus

der Schule ist die einfache Regel "Punkt- vor Strichrechnung" bekannt, die besagt, dass die Operatoren \* und / eine höhere Priorität (oder auch eine stärkere Bindung) haben als die Operatoren + und -.

### Beispiel 7.9:

$3 + 4 * 5$  bedeutet mit expliziter Klammerung  $3 + (4 * 5)$ . ◆

Programmiersprachen folgen dieser einfachen Regel. Aber zusätzlich wird auch in jeder Programmiersprache zu jedem Operator genau die Priorität definiert, um so eindeutig den Wert eines Ausdrucks bestimmen zu können. In einem Ausdruck mit mehreren Operatoren werden die Operatoren in der Reihenfolge angewandt, die die Priorität vorgibt. Im Anhang ist in Tabelle C.1 die Priorität für alle Java-Operatoren angegeben. Obere Tabelleneinträge bedeuten höhere Priorität, also stärkere Bindung. Operatoren zwischen Trennstrichen haben gleiche Priorität.

### Beispiel 7.10:

Der Ausdruck

7-1 `! true && 3 < 4 || 5 + 6 > 7`

hat entsprechend obiger Prioritätstabelle mit expliziter Klammerung folgendes Aussehen:

7-1 `((! true) && (3 < 4)) || ((5 + 6) > 7)`

Es gibt Wichtigeres zu behalten als diese Tabelle. Oder wissen Sie jetzt noch die genau Reihenfolge aller in der Tabelle aufgeführten Operatoren? Deshalb ist eine einfache zu merkende Regeln: Klammern haben oberste Priorität. Wenn Sie sich also nicht sicher sind, welcher Operator in einem Ausdruck nun stärker bindet, so setzen Sie einfach Klammern um die Teilausdrücke!

Neben der Priorität von Operatoren ist weiterhin auch die Assoziativität wichtig, falls mehrere Operatoren gleicher Priorität genutzt werden. Für eine Addition im Datentyp `int` der Form  $3 + 4 + 5$  ist es unerheblich, ob die Auswertung in der Reihenfolge  $(3 + 4) + 5$  oder  $3 + (4 + 5)$  geschieht, man erhält in beiden Fällen für die angegebenen Werte das gleiche Ergebnis. Führt man diese Addition aber mit Fließkommazahlen und den entsprechenden Fließkommaoperatoren durch, so ist die Reihenfolge aufgrund der bereits oben besprochenen Eigenschaften der Darstellung von Fließkommazahlen durchaus wichtig, denn Fließkommanoperatoren sind aufgrund der Problematik der exakten Darstellbarkeit von Zwischenresultaten nicht grundsätzlich assoziativ.

### Beispiel 7.11:

Im Programm in Listing 7.5 werden die Zahlen 1, 1, 1, 2 und 1, 3 addiert, einmal von links nach rechts und ein anderes Mal von rechts nach links, was durch explizite Klammerung erzwungen werden kann. Als Ausgabe erscheint auf dem Bildschirm:

7-1 `(a + b) + c = 3.6000001`  
7-2 `a + (b + c) = 3.6`

Unbedarf könnte man nach diesem Beispiel argumentieren, dass die beiden Werte sich ja nur geringfügig unterscheiden. Deshalb ein weiteres Beispiel, das die Problematik und die möglichen Auswirkungen

Listing 7.5: Reihenfolge von Operationen.

```

7-1  /**
7-2   * Beispiel zur Assoziativitaet von Operatoren
7-3   */
7-4  public class Auswertungsreihenfolge {
7-5      public static void main(String [] args) {
7-6
7-7          float a = 1.1f;
7-8          float b = 1.2f;
7-9          float c = 1.3f;
7-10
7-11         System.out.println("(a + b) + c = " + ((a + b) + c));
7-12         System.out.println("a + (b + c) = " + (a + (b + c)));
7-13     }
7-14 }
```

Listing 7.6: Reihenfolge von Operationen.

```

7-1  /**
7-2   * Beispiel zur Assoziativitaet von Operatoren
7-3   */
7-4  public class Auswertungsreihenfolge2 {
7-5      public static void main(String [] args) {
7-6
7-7          float a = -67108864f;
7-8          float b = 67108865f;
7-9          float c = 1f;
7-10
7-11         System.out.println("(a + b) + c = " + ((a + b) + c));
7-12         System.out.println("a + (b + c) = " + (a + (b + c)));
7-13     }
7-14 }
```

hoffentlich jedem klar macht. Auch hierbei ist der eigentlich Ursprung des Problems die Restriktionen der Fließkommadarstellung.

### Beispiel 7.12:

Zum Programm in Listing 7.6 erscheint als Ausgabe auf dem Bildschirm:

```

7-1 (a + b) + c = 1.0
7-2 a + (b + c) = 0.0
```

Wiederholungsfrage zu diesem Beispielprogramm: Wieso sind diese Zahlen so gewählt? Könnte man auch andere float-Zahlen wählen? Könnten das beliebige Kombinationen von float-Zahlen sein? ♦

In Tabelle C.1 im Anhang ist deshalb weiterhin in der letzten Spalte die Assoziativität der Operatoren angegeben. Linkssassoziativität bedeutet, dass ein Ausdruck der Form  $a + b + c$  ausgerechnet wird in der Form  $(a + b) + c$ . Rechtsassoziativität bedeutet, dass ein Ausdruck der Form  $a = b = c$  ausgewertet wird wie  $a = (b = c)$ .

In einem Java-Ausdruck werden zuerst Prioritätsregeln angewandt und erst, wenn Operatoren gleicher Priorität

zur Wahl stehen, deren Assoziativität berücksichtigt.

Während in manchen Sprachen die Reihenfolge der Auswertung abgesehen von den Prioritäten nicht zwingend vorgeschrieben ist (zum Beispiel in der Programmiersprache C, um möglichst weitgehende Optimierungen des Compilers zu ermöglichen), ist in Java die Reihenfolge also genau festgelegt: Die Auswertung beispielsweise des Ausdrucks  $1.1 + 1.2 + 1.3$  würde auf jeden Fall und somit reproduzierbar (vergleiche Eigenschaften eines Algorithmus) als  $(1.1 + 1.2) + 1.3$  ausgewertet.

Ebenfalls könnte ein Compiler geneigt sein, den Ausdruck  $4.0 * x * 0.5$  durch  $2.0 * x$  zu ersetzen. Auch dies ist in Java (km Gegensatz zu den meisten anderen Programmiersprachen) aus den gleichen Gründen nicht erlaubt, der Compiler muss also Code erzeugen, der zuerst  $4.0 * x$  multipliziert und anschließend dieses Zwischenergebnis mit 0.5 multipliziert.

Sieht man sich die Definition des logischen Und an, so wird klar, dass die Operation `false` liefert, wenn einer der beiden Operanden `false` ist, unabhängig vom Wahrheitswert des anderen Operanden. Man könnte also die Ausführung eines Programmes beschleunigen, wenn man nur einen (zum Beispiel den ersten) Operanden des logischen Und auswertet und, falls dieser den Wert `false` hat, auf die Auswertung des zweiten Operanden verzichtet. Analoges gilt für das logische Oder, wenn einer der beiden Operanden `true` ist. Diese Regelung gilt auch in Java (und C/C++) für die boolschen Operatoren `&&` und `||`. Steht nach Auswerten des linken Operanden das Ergebnis fest, so wird der zweite Operand nicht mehr ausgewertet (verkürzte Auswertung, *Shortcut*). Dass dieses Vorgehen durchaus diskussionswürdig und nicht selbstverständlich ist, liegt darin, dass die Auswertung des zweiten Operanden zum Beispiel nicht terminieren kann (Endlosschleife) und somit strenggenommen auch das Ergebnis der Gesamtoperation nicht definiert wäre, weil dann ebenfalls die Auswertung des Gesamtausdrucks nicht terminieren würde. Ebenso kann die Auswertung des zweiten Operanden Seiteneffekte erzeugen, zum Beispiel den Wert einer Variablen verändern (Beispiel: `++a`). In Java und andere Programmiersprachen gibt es deshalb eine Möglichkeit der Unterscheidung, welche Version man haben möchte. Die logischen Operatoren `&&` und `||` sind in Java so definiert, dass das Gesamtergebnis schon mit dem ersten Operanden feststehen muss, wenn das Gesamtergebnis im obigen Sinne schon festgelegt ist. Java bietet als Alternative die Operatoren `&` und `|` (das heißt nur ein Symbolzeichen jeweils), die auf jeden Fall beide Operanden auswerten. Werden auf jeden Fall alle Operanden einer Operation ausgewertet, so spricht man von **strikter Auswertung**, ansonsten von nichtstrukterter Auswertung.

## 7.5 Konstante Werte

Manchmal möchte man in einem Programm einem bestimmten Zahlenwert einen sinnvollen Namen geben, der besser den Sachverhalt ausdrückt. Ein Beispiel wäre es, dem Wert  $3,1415\dots$  den Namen `PI` zu geben und überall dort, wo dieser Wert genutzt wird, stattdessen `PI` zu verwenden. Mit den jetzigen Mitteln könnte man dies etwa durch eine Deklaration `double PI = 3.1415;` erreichen. Ein Nachteil wäre, dass dadurch eine *Variable*, also etwas Veränderliches angelegt wird. Der Wert Pi wird aber nach menschlichem Ermessen nie einen anderen Wert haben. Um dies auch explizit auszudrücken, gibt es in Java das Schlüsselwort `final`, das im Rahmen einer Variablen Deklaration genutzt werden kann, um obigen Sachverhalt explizit anzugeben. Würde man im Gültigkeitsbereich/sichtbaren Bereich dieser so deklarierten Variablen versuchen im Programm den Wert (zum Beispiel in einer Zuweisung) zu ändern, so erkennt dies der Compiler und markiert dies als Fehler. Nach den *Java Code Conventions* werden die Namen von als `final` deklarierten Variablen ausschließlich aus Großbuchstaben gewählt. Über die Markierung einer Variablen als `final` und die Einhaltung dieser Konvention kann man damit einerseits einerseits eine erhöhte Lesbarkeit erreichen, da in einem Programm die Verwendung einer Variablen aus ausschließlich Großbuchstaben (vereinbarungsgemäß!) einen nicht-veränderlichen Wert darstellt.

Listing 7.7: Beispiel zu einer final-Deklaration einer Variablen/Konstanten.

```

7-1  /**
7-2   * Umwandlung von Fahrenheit nach Celsius (ueber Konstante)
7-3   */
7-4 public class FahrenheitNachCelsiusTest {
7-5
7-6     public static void main(String [] args) {
7-7         final double FAKTOR = 5.0 / 9.0;
7-8         double fahrenheit = 72;
7-9         double celsius = FAKTOR * (fahrenheit - 32.0);
7-10        System.out.println(fahrenheit + " Grad F sind in Celsius " + celsius);
7-11    }
7-12 }
```

**Beispiel 7.13:**

In Listing 7.7 ist das Beispiel zur Umrechnung von Fahrenheit nach Celsius so modifiziert worden, dass der konstante Umrechnungsfaktor 5/9 als eine final-markierte Variable / Konstante angegeben wurde. ♦

Einige Anmerkungen dazu:

- In anderen Programmiersprachen wird statt `final` oft das Schlüsselwort `const` verwendet. Beispiel: C/C++.
- In Java kann `final` auch in anderen Zusammenhängen (zum Beispiel Klassendefinitionen) und mit einer dann anderen Semantik verwendet werden.
- Mit einer `final`-Deklaration einer Variablen gibt man einem Compiler auch semantische Zusatzinformationen, nämlich, dass diese Variable definitiv niemals ihren Wert ändern wird. Diese Information kann ein Optimierer im Compilers zum Beispiel nutzen.

## 7.6 Typumwandlung

Jeder Ausdruck in Java hat einen Typ, der sich aus der Struktur und dem Inhalt des Ausdrucks sowie den oben formulierten Regeln (Priorität, Assoziativität) eindeutig ermitteln lässt. Grundsätzlich gibt es in Java die Möglichkeit der Umwandlung eines Wertes von einem Typ in einen anderen Typ. Diese Umwandlung kann sowohl explizit durch den Programmierer als auch implizit ohne Zutun des Programmierers möglich sein. In manchen Fällen sind solche Typumwandlungen sinnvoll anwendbar, manchmal abhängig von den umzuwandelnden Werten und manchmal können sie auch keinen Sinn machen.

**Beispiel 7.14:**

Nachfolgend sind beispielhaft einige mögliche Fälle aufgeführt, die auftreten können:

- Die Umwandlung des Wertes `5` im Datentyp `byte` zum Wert `5` im Datentyp `int` ist problemlos möglich; beide Werte in den jeweiligen Datentypen stellen den gleichen mathematischen Wert `5` exakt dar.
- Die Umwandlung des Wertes `3.0` vom Typ `double` in den Wert `3` des Datentyps `int` ist möglich.
- Die Umwandlung des Wertes `3.5` vom Typ `double` in einen Wert vom Typ `int` ist nicht möglich. Eine Annäherung in `int` an den exakten Wert wäre `3` oder `4`. ♦

Die oben beispielhaft aufgeführten Probleme lassen sich wie folgt klassifizieren:

- Nicht jeder Typ kann in jeden anderen Typ sinnvoll umgewandelt werden. Beispiel: `double` nach `boolean`
- Nicht jeder Wert des einen Typs ist im anderen Typ darstellbar. Beispiel: `double` nach `int`.
- Nicht jeder Wert des einen Typs ist im anderen Typ *exakt* darstellbar. Beispiel: `int` nach `double`.

Typumwandlungen sind ein sehr komplexes Thema und werden an dieser Stelle nur mit ihren für die Praxis wichtigsten Aspekten behandelt. Für eine vollständige Diskussion sei auf die Literatur verwiesen. Das Referenzwerk zu Java ist [Javc], wo in Kapitel 5 umfassend Typumwandlungen in Java beschrieben sind.

Da einige der Typumwandlungen teilweise aus Bequemlichkeit für den Programmierer implizit, also ohne explizite Formulierung des Programmierers, durchgeführt werden, erkennt ein Programmieranfänger eventuell nicht, was an manchen Stellen im Programm wirklich passiert. In dieser Beschreibung erfolgt eine Beschränkung auf die wesentlichen Aspekte zu den bis jetzt bekannten primitiven Datentypen. Die später in Kapitel 9 beschriebenen Referenztypen besitzen eigene Regeln der Typumwandlung, auf die an dieser Stelle nicht eingegangen wird.

### **Beispiel 7.15:**

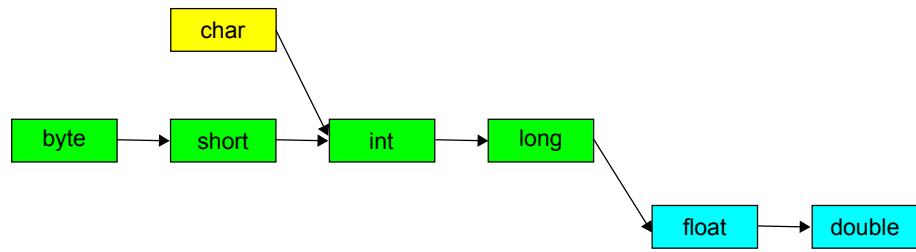
In allen gängigen Programmiersprachen ist es erlaubt zu schreiben: `3.0 + 1`, das heißt der Additionsoperator angewandt auf zwei Werte unterschiedlichen Typs, `3.0` vom Typ `double` und `1` vom Typ `int`. Bekannt ist ja bereits aus den vorangehenden Kapiteln, dass ganzzahlige Werte ganz anders gespeichert werden als Fließkommawerte (Zweierkomplementdarstellung versus IEEE-Darstellung). Eine binäre Addition, wie diese etwa bei den ganzen Zahlen vorgestellt wurde, ist also nicht ohne weiteres möglich. Wenn man den obigen Ausdruck so schreibt, so findet durch den Compiler automatisch eine Typumwandlung statt. In diesem Fall würde der Compiler Code erzeugen, der `3.0` auswertet, dann `1` auswertet, anschließend das Resultat des zweiten Teilausdrucks (`int 1`) in einen Fließkommawert `1.0` umwandelt und dann die Fließkommaaddition auf die beiden Werte Fließkommawerte anwendet. ♦

In Java lassen sich alle numerischen Datentypen, also alle primitiven Datentypen außer dem Typ `boolean`, prinzipiell in einen anderen umwandeln.

Man spricht bei primitiven Datentypen von erweiternder Typumwandlung, wenn einer der in Abbildung 7.5 im oberen Teil durch Pfeile angegebenen Wege beschritten wird. Beispiel: `short` nach `int` oder `char` nach `double`. Solche erweiternden Typumwandlungen finden exakt statt, solange die Typumwandlung innerhalb der integralen Typen stattfindet. Der Übergang von integralem Typ zu Fließkommotyp kann allerdings zu Problemen der Genauigkeit führen (siehe Übungsaufgabe). Eine erweiternde Typumwandlung innerhalb der integralen Typen führt zu einer Erweiterung der Zweierkomplementdarstellung auf den größeren Typ unter Beachtung des Vorzeichens. Beispiel: eine `-3` im Datentyp `byte` bleibt nach Umwandlung in den Datentyp `int` nach wie vor eine `-3`.

Man spricht bei primitiven Datentypen von verengender Typumwandlung, wenn einer der in Abbildung 7.5 im unteren Teil durch Pfeile angegebenen Wege beschritten wird. Beispiel: `long` nach `int` oder `double` nach `char`. Bei diesen verengenden Umwandlungen kann es zur Problemen der Darstellbarkeit und Genauigkeit kommen. Die möglichen Probleme einer einengenden Typumwandlung sind vielfältiger. Bei der Umwandlung eines integralen Typs der Größe  $n$  Bits in einen kleineren integralen Typ der Größe  $m$  Bits werden einfach die untersten  $m$  Bits erhalten und die oberen  $n - m$  Bit ignoriert. Dadurch kann es durch die Zweierkomplementdarstellung zu einer Vorzeichenumkehr kommen. Zur Erinnerung: bei negativen Zahlen ist das höchste Bit immer 1, bei positiven Zahlen immer 0.

### erweiternde Typumwandlung



### verengende Typumwandlung

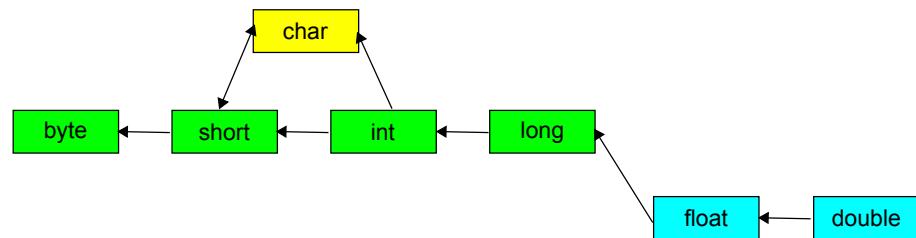


Figure 7.5: Erweiternde und verengende Typumwandlungen

### Beispiel 7.16:

Im Programm in Listing 7.8 wird der Variablen `i` ein großer negativer Wert zugewiesen ( $-2^{30}$  oder hexadezimal `c0000000`). Die anschließende Umwandlung zu einem Byte-Wert streicht die obersten 24 Bit (also die ersten 6 Ziffern der Hexadezimalzahl) und lässt die unteren 8 Bit übrig (die untersten 2 Hexadezimalziffern). Das Programm liefert als Ausgabe:

7-1	<code>i : -1073741824</code>
7-2	<code>b : 0</code>

Implizite Typumwandlungen sind ausschließlich erweiternde Typumwandlungen und finden statt:

- wenn in einer Zuweisung der primitive Typ des Ausdrucks auf der rechten Seite nicht mit dem primitiven Typ der Variablen auf der linken des Gleichheitszeichens übereinstimmt
- wenn in einem arithmetischen Ausdruck unterschiedliche numerische Typen vorkommen
- zur Anpassung von Operanden auf Operatoren
- in einem Methodenaufruf (später mehr dazu), wenn der Typ eines aktuellen Parameters nicht mit dem Typ des formalen Parameters übereinstimmt

### Beispiel 7.17:

Der Programmausschnitt in Listing 7.9 zeigt Beispiele zu diesen prinzipiellen Möglichkeiten für implizite Typumwandlungen. ♦

Sind Operanden einer arithmetischen Operation vom Typ `byte`, `short` oder `char` (also kleiner als `int`, so werden die Operanden grundsätzlich zuerst auf den Datentyp `int` erweitert, die Operation findet dann also auch im Datentyp `int` statt.

Listing 7.8: Einengende Typumwandlung.

```

7-1 /**
7-2 * Informationsverlust durch einengende Typumwandlung
7-3 */
7-4 public class EinengendeTypumwandlung {
7-5
7-6     public static void main(String [] args) {
7-7         // i bekommt einen grossen negativen Wert
7-8         int i = - (1<<30);
7-9         System.out.printf("i: " + i);
7-10
7-11         // hier ist die einengende Typumwandlung
7-12         // die oberen Bits werden einfach abgeschnitten
7-13         byte b = (byte)i;
7-14         System.out.println("b: " + b);
7-15     }
7-16 }
```

Listing 7.9: Einige prinzipielle Möglichkeiten für Typumwandlungen.

```

7-1 /**
7-2 * Beispiele zu den verschiedenen impliziten Typumwandlungen
7-3 */
7-4 public class Typumwandlung {
7-5
7-6     public static void main(String [] args) {
7-7         byte b = 1;
7-8         // Zuweisungsumwandlung
7-9         int a = b;
7-10
7-11         // unterschiedliche Typen fuer a und b
7-12         a = a + b;
7-13
7-14         b = 127;
7-15         byte b2 = 126;
7-16         // auch wenn beide Operanden b und b2 vom Typ byte sind,
7-17         // werden die Operanden zuerst nach int umgewandelt
7-18         // und im Typ int findet auch die Addition statt.
7-19         // Deshalb ist das Resultat auch das korrekte Ergebnis 253,
7-20         // was im Typ byte nicht darstellbar ist
7-21         a = b + b2;
7-22         System.out.println(a);
7-23
7-24         // Methodenaufruf. Math.sqrt erwartet double-Argument
7-25         double d = Math.sqrt(2);
7-26     }
7-27 }
```

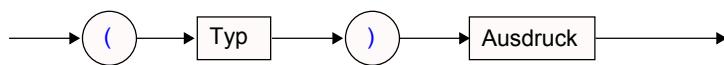
**cast-Ausdruck**

Figure 7.6: Syntax für eine explizite Typumwandlung

Explizite Typumwandlungen sind sowohl als erweiternde als auch verengende Typumwandlungen möglich. Die Syntax dafür ist in Abbildung 7.6 gegeben. Der Wunschtyp wird in Klammern vor den Ausdruck geschrieben, wobei die Umwandlung des Ursprungstyps des Ausdrucks in den Wunschtyp prinzipiell möglich sein muss. Eine solche Operation nennt man eine **cast-Operation**.

### Beispiel 7.18:

An dieser Stelle soll auf das Beispiel im Zusammenhang mit dem Datentyp `char` (Kapitel 4.5) zurück gegriffen werden.

```
7-1  char a = 'x';
7-2  char b = (char)(a + 1);
7-3  // b enthaelt das Nachfolzeichen von 'x', also 'y'
```

In dem Ausdruck `a+1` ist `a` vom Typ `char`, `1` vom Typ `int`. Deshalb findet eine implizite Umwandlung des `char`-Wertes in einen `int`-Wert statt, die Addition wird als `int`-Addition durchgeführt und das Resultat dieses Ausdrucks ist ebenfalls vom Typ `int`. Da die zuzuweisende Variable vom Typ `char` ist, ist eine Typumwandlung des `int`-Wertes in einen `char`-Wert notwendig. Dies ist eine einengende Typumwandlung, wird also nicht implizit durchgeführt. Insofern muss eine explizite Typumwandlung in Form einer cast-Operation durch den Programmierer erfolgen. Fehlt diese cast-Operation, so ist dies ein Programmierfehler, den der Compiler erkennt und beanstandet. ♦

## 7.7 Zusammenfassung und Hinweise

### Verstehen

Es sollte verstanden sein, dass es (in Java) feste Regeln zur Anwendung von Operatoren gibt. Typumwandlungen können Werte verändern. Manche Typumwandlungen finden implizit statt.

### Kurz und knapp merken

- Jeder Ausdruck liefert bei seiner Berechnung einen Wert.
- Jeder Ausdruck hat einen Typ, der sich aus den Operatoren und Operanden eindeutig ergibt.
- Zuweisungsoperatoren verändern den Wert einer Variablen (Update-Operation).
- Operatoren können überladen sein. Damit kann das gleiche Symbol in mehreren Zusammenhängen, evtl. sogar in unterschiedlicher Bedeutung genutzt werden.
- Alle Operatoren haben eine definierte Priorität und Assoziativität.
- `|` und `||` sowie `&` und `&&` zeigen Unterschiede in der Auswertung des zweiten Operanden.
- Über das Schlüsselwort `final` kann man nicht mehr zu veränderne Variablen (Konstanten) deklarieren, was die Lesbarkeit und Sicherheit in einem Programm erhöhen kann.
- Mit Typumwandlungen lassen sich Werte in einen anderen Typ umwandeln. Dies ist prinzipiell nicht immer möglich.
- Man unterscheidet erweiternde und einengende Umwandlungen.
- Implizite Umwandlungen sind nur erweiternd möglich, explizite Umwandlungen in Form eines cast-Ausdrucks sowohl erweiternd als auch einengend.

## Häufige Fehler

- Typumwandlungen sind ein häufiger Quell von Fehlern. Viele Typumwandlungen finden implizit statt. Sowohl bei impliziten als auch expliziten Typumwandlungen muss man sich im Klaren darüber sein, was intern bei der Umwandlung passiert.
- Die Priorität und Assoziativität von Operatoren wird falsch oder gar nicht eingeschätzt. Wenn Sie sich im unklaren sind, verwenden Sie explizite Klammerung.
- Inkrement- und Dekrementoperatoren sind sehr einfach in der Anwendung, wenn man denn genau ihre Wirkung kennt. Oft wird dabei aber mit Halbwissen operiert.

## Übungsfragen

- Wie kann ich in Java angeben, dass sich der Wert einer Variablen nach der ersten Zuweisung nicht mehr ändert?
- Was sind die Vorteile, wenn man eine Variable mit final deklariert?
- Erläutern Sie den Unterschied zwischen `++` und `+`
- Erläutern Sie den Unterschied zwischen `&&` und `&`
- Was bedeutet: `int x = (a >> 1 != 5) ? 'A' : 'B' + 1;`
- Welchen Wert hat x:  
`int x = 1; x = ((x = (x++ << 1))) + ++x;`
- Welchen Wert hat x:  
`int x = 1; x = x++ + ++x;`
- Welchen Wert hat x:  
`int x = 1; x = (x++ << ++x) >> x++;`
- Welchen Wert hat x:  
`int x = 10; x = (x++ / (++x >> 2));`
- Welchen Wert hat x:  
`int x = 10; x = (++x / (x++ >> 2));`
- Welchen Wert hat x:  
`int x = 10; x = x++ % x--;`
- Welchen Wert hat x:  
`int x = 10; x = (x = x++) + 5;`
- Welchen Wert hat x:  
`int x = 10; x = ((++x > 0) & (++x > -10)) ? x+5 : x-5;`
- Welchen Wert hat x:  
`int x = 10; x = ((++x > 0) && (++x > -10)) ? x+5 : x-5;`
- Was ist der Typ und Wert des Ausdrucks: `3L + 4.0F`
- Was ist der Typ und Wert des Ausdrucks: `'A' + 1`
- Was ist der Typ und Wert des Ausdrucks: `(int)3.9`
- Was ist der Typ und Wert des Ausdrucks: `(long)-1 * 3.0`
- Was ist der Typ und Wert des Ausdrucks: `(int)(1.5 * (int)1.5)`
- Was ist der Typ und Wert des Ausdrucks: `(byte)-1, (byte)255`
- Was ist der Typ und Wert des Ausdrucks: `(int)3.5 - (byte)3.5`
- Was ist der Typ und Wert des Ausdrucks: `(byte)5, (byte)255, (byte)256`
- Was ist der Typ und Wert des Ausdrucks: `(byte)(1 << 7), (byte)(1 << 8), (int)(byte)-1`
- Was ist der Typ und Wert des Ausdrucks: `(byte)4.9, (byte)4.499E0`
- Was ist der Typ und Wert des Ausdrucks: `(byte)1E2`

- Was ist der Typ und Wert des Ausdrucks: `(char) ('R' +32-1)`

## Reflektion des Stoffs

- Welche Werte haben die Variablen `x` und `y` nach folgender Anweisungsfolge und wieso?

```
7.1 int x = 5, y = 5;  
7.2 x = ++x + y++;
```

- Geben Sie ein Beispiel an, wo aus einem positiven Wert durch eine cast-Operation ein negativer Wert entsteht.
- Darf man innerhalb des Rumpfes einer Zählschleife eine Konstante (`final`) deklarieren? Wenn nein, wieso nicht? Wenn ja, wie und was ist dann der Gültigkeitsbereich und die Lebensdauer?



## Chapter 8

# Programmstrukturierung durch Methoden

Die bis jetzt gezeigten Java-Programme waren alle sehr kurz (wenige Zeilen lang) und enthielten den kompletten Programmcode in einem umfassen Block von `main`. Da auch bis jetzt die Aufgabenstellungen nicht allzu komplex waren, ist das auch verständlich und die gezeigten Programme bereiteten in dieser Form auch keine (Verständnis-)Probleme. Sicherlich ebenfalls verständlich ist aber auch, dass nicht alle Probleme der Welt so einfach wie die bis jetzt behandelten sind und auch nicht alle Programmlösungen der Welt in wenige Zeilen Code passen. Beispielsweise der Programmcode für gängige Betriebssysteme umfasst derzeit etwa jeweils 10-100 Millionen Zeilen Programmcode[SLO]. Große betriebliche Anwendungssysteme wie SAP liegen in der gleichen Größenordnung. Und damit ist natürlich auch klar, dass diese Programmzeilen nicht alle untereinander in einem Block namens `main` stehen. Bereits wenige hundert zusammenhängende Zeilen Programmcode sind für einen Menschen meist kaum noch lesbar, erweiterbar und wartbar, wenn dieser Programmcode nicht weiter strukturiert wird. Eine Strukturierung dient dem systematischen Aufbau von Programmen, der Kapselung von Funktionalität und Daten sowie der Wiederverwendung von Programmcode. Solche eine Strukturierung von Programmen kann auf verschiedenen Ebenen erfolgen. Die Programme / Anwendungssysteme, die die oben erwähnte Größenordnung von Millionen Codezeilen umfassen, sind aufgeteilt in einzelne Teilprogramme oder Subsysteme, die ganz bestimmte Aufgaben erfüllen. Aber selbst ein einziges abgeschlossenes Programm wird üblicherweise noch weiter unterteilt. Je nach Programmierparadigma und Programmiersprache ist das Vorgehen und sind die Möglichkeiten allerdings unterschiedlich. In Java etwa gibt es als Strukturierungselemente Pakete, darunter Klassen, innerhalb von Klassen die Methoden und darin wiederum Blöcke. Mit diesen Methoden werden wir uns nun in diesem Kapitel auseinandersetzen. Klassen kommen dann in einem späteren Kapitel (Kapitel 11).

Im einführenden Kapitel wurde schon gezeigt, dass es wünschenswert ist wiederverwendbare (Teil-)Algorithmen als Funktionen zu kapseln. Dort wurde das Beispiel die Flächenberechnung eines beliebigen Rechtecks als Funktion von zwei Parametern angeführt.

### Beispiel 8.1:

Berechnet werden sollte die Fläche zweier Rechtecke  $a$  und  $b$  mit Längen- und Breitenangaben von  $10 \times 10$  m beziehungsweise  $20 \times 5$  m. Der Algorithmus unter Nutzung einer entsprechenden Funktion ist:

```
8-1  Flaecheninhalt von Rechteck a = Flaechenberechnung(10,10)  
8-2  Flaecheninhalt von Rechteck b = Flaechenberechnung(20,5)
```

wobei die Funktion `Flaechenberechnung` wie folgt gegeben war:

Listing 8.1: Programm zur Flächenberechnung.

```

8-1  /**
8-2   * Flaechenberechnung von Rechtecken
8-3   */
8-4  public class Flaechenberechnung {
8-5      public static void main(String [] args) {
8-6
8-7          // Hier erfolgen zwei Aufrufe der Methode zur Flaechenberechnung
8-8          System.out.println("Die Flaeche von Rechteck a ist " + flaeche(10.0, 10.0));
8-9          System.out.println("Die Flaeche von Rechteck b ist " + flaeche(20.0, 5.0));
8-10     }
8-11
8-12     // Hier steht der Algorithmus zur Berechnung der Flaeche
8-13     public static double flaeche(double laenge, double breite) {
8-14         return laenge * breite;
8-15     }
8-16 }
```

```
8-1  Flaechenberechnung (Laenge, Breite) := Laenge * Breite
```

Diese Ansatz, Modul- oder auch Unterprogrammtechnik genannter, ist in Programmiersprachen ebenfalls vorhanden und stellt ein sehr wichtiges Konzept zur strukturierten ebenso wie zur später behandelten Objektorientierten Programmierung dar. In den verschiedenen Programmiersprachen ist die Bezeichnungsweise für Module dann **Methoden** (Java), **Funktionen** (C und C++), Prozeduren / Funktionen (Pascal) oder **Unterprogramme** (Fortran).

### Beispiel 8.2:

Obiges Beispiel sieht in Java wie in Listing 8.1 aus. Die Ausgabe des Programms ist:

```

8-1  Die Flaeche von Rechteck a ist 100
8-2  Die Flaeche von Rechteck b ist 100
```

Nutzt man die Unterprogrammtechnik, so müssen – wie im Beispiel zu sehen – zwei Dinge erfolgen: erstens die Deklaration einer Methode inklusive der Angabe des Algorithmus' und zweitens deren Aufruf, der beliebig oft erfolgen kann. Diese beiden Aufgaben sollen im Folgenden nacheinander betrachtet werden. Dabei sind auch Fragen der Parameterübergabe zu klären.

## 8.1 Definition einer Methode

Vergleichen Sie die nachfolgende Diskussion zu Methoden mit der Definition einer mathematischen Funktion *flaeche*, deren Definitions- und Wertebereich gegeben ist durch:

$$flaeche : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

mit der Funktionsdefinition

$$flaeche(laenge, breite) := laenge \cdot breite$$

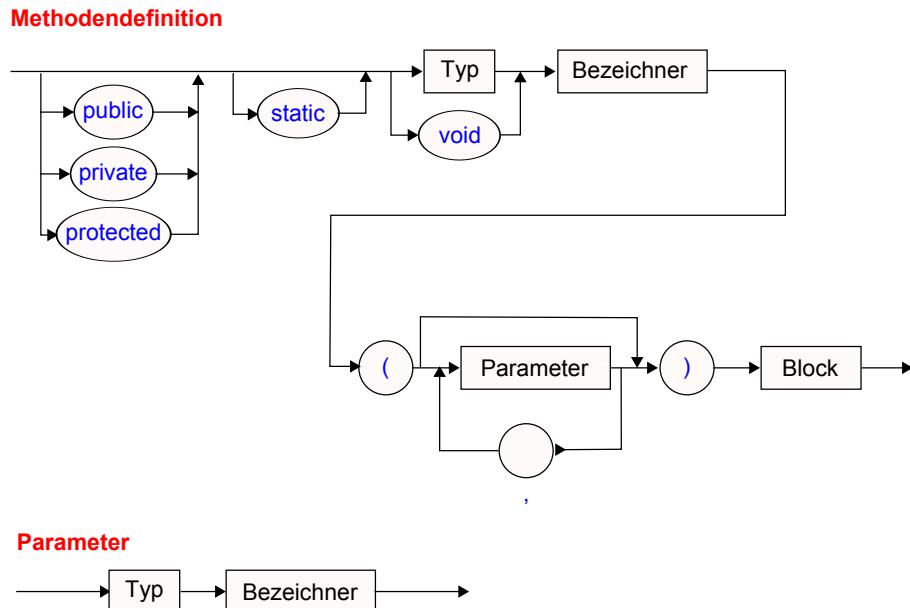


Figure 8.1: Syntaxdiagramm für eine Methodendefinition

sowie ihrer (doppelten) Nutzung an anderer Stelle in der Form

$$\text{gesamtflaeche} \leftarrow \text{flaeche}(10, 10) + \text{flaeche}(20, 5)$$

Eine Java-Methode muss genau einmal *definiert* werden. In einer *Methodendefinition* wird der Algorithmus konkret beschrieben zusammen mit einer formalen Beschreibung, wie diese Methode an anderer Stelle genutzt werden kann (Schnittstelle). Diese beiden Teile einer Methodendefinition heißen Methodenkopf (Algorithmusimplementierung) und Methodenrumpf (Beschreibung der Aufrufschwittstelle).

### Beispiel 8.3:

Im Programmausschnitt

```

8-1  public static double flaeche(double laenge, double breite) {
8-2      return laenge * breite;
8-3  }
```

besteht der Methodenkopf aus dem ersten Teil

```
8-1  public static double flaeche(double laenge, double breite)
```

und der Methodenrumpf aus dem nachfolgenden restlichen Programmteil (die öffnende geschweifte Klammer hier der Übersicht halber nach vorne gezogen)

```

8-1  {
8-2      return laenge * breite;
8-3  }
```

Der Methodenkopf gibt die **Schnittstelle** an, über die diese Methode referenzierbar ist. Dort ist der Name der Methode angegeben (**flaeche**), die Anzahl, der Typ und die Namen der *formalen Parameter* (**laenge** vom Typ **double** und **breite** vom Typ **double**), sowie der Typ des Ergebnisses (**double**), vor dem Methodennamen

angegeben. Im Beispiel wird also eine Methode definiert, deren Name `flaeche` ist, die zwei Parameter `laenge` und `breite` vom Typ `double` besitzt und die ein Ergebniswert vom Typ `double` liefert.

Die Syntax für eine Methodendefinition in Java ist in Abbildung 8.1 gegeben. Java kennt weitere Möglichkeiten, auf die aber im Rahmen dieses Buches nicht eingegangen wird (weitere Schlüsselwörter, Exceptions, abstrakte Methoden,...). Diese Details sind in [Javc] zu finden.

Im Methodenkopf, der Schnittstelle zwischen Implementierer und Nutzer dieser Methode, wird formal angegeben, wie die Methode aufgerufen werden muss, jedoch nicht, wie die Methode ihre Berechnung durchführt. Die Java-Syntax erlaubt einige optionale Angaben zu Beginn eines Methodenkopfs. Dazu gehört eine optionale Angabe zur Publizierung der Methode, was eine Art Zugriffsberechtigung für die Nutzung in anderen Klassen ist (später mehr). Solange nur eine Klasse programmiert wird, ist diese Angabe nicht wichtig. Bis auf Weiteres soll `public` dort geschrieben werden oder diese Angabe weggelassen werden. Eine weitere optionale Angabe ist das Schlüsselwort `static`, das eine wichtige Rolle spielt. Details dazu werden ebenfalls später in Kapitel 11 behandelt. Bis dahin soll immer `static` zu einer Methode geschrieben werden.

Die restlichen Angaben des Methodenkopfs ähneln der Definition einer mathematischen Funktion, wenn auch in der Reihenfolge etwas anders notiert. Vergleichen sie nachfolgende folgende Beschreibung mit der Angabe der mathematischen Funktionsangabe  $flaeche : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ . Zuerst folgt die Angabe des Ergebnistyps, in der Mathematik wird dieser ans Ende geschrieben. Als Ergebnistyp ist jeder Java-Typ zugelassen. Neben den bis jetzt bekannten primitiven Datentypen auch die noch später folgenden Referenztypen (Kapitel 9). Weiterhin ist ein besonderer "Typ" an dieser Stelle erlaubt: `void` besagt, dass das Ergebnis leer ist, also kein Ergebnis geliefert wird. Dies macht zum Beispiel dann Sinn, wenn die Aufgabe einer Methode darin besteht eine Bildschirmausgabe durchzuführen; ein Ergebniswert wird also garnicht produziert.

Nach dem Ergebnistyp wird der Name der Methode angegeben. Hier ist ein beliebiger Java Bezeichner zugelassen (Folge von Buchstaben oder Ziffern, erstes Zeichen ein Buchstabe), die Java Code Conventions[Javb] geben den Hinweis, dass der Bezeichner mit einem Kleinbuchstaben beginnen soll, Teilwörter fangen mit einem Großbuchstaben an und nach Möglichkeit sollte der Name von einem Verb abgeleitet sein (hier wird etwas aktiv getan; unsere Beispilmethode sollte also eigentlich besser `flaecheBerechnen` heißen).

Nach dem Methodennamen muss eine runde öffnende Klammer folgen. Vor der schließenden Klammer, die den Methodenkopf beendet, können optional Angaben zu formalen Parametern erfolgen. Werden mehrere formale Parameter angegeben, so werden die Angaben durch Komma getrennt. Die formalen Parameter sind Platzhaltersymbole für die in einem Methodenaufruf an anderer Stelle übergebenen aktuellen Parameterwerte. Der Name eines formalen Parameters ist beliebig, da es halt nur ein Platzhalter ist. Der Sinn eines formalen Parameters ist, dass man innerhalb einer Methode auf einen aktuellen Parameterwert Bezug nehmen kann. Jede formale Parameter entspricht einer Variablen Deklaration, allerdings ohne den ansonsten möglichen Initialisierungsausdruck. Das heißt: eine Typangabe gefolgt von einem Bezeichner. Die so im Methodenkopf eingeführten formalen Parameter werden wie Variablen behandelt, die mit dem Wert des aktuellen Übergabeparameter bei einem Methodenaufruf (siehe Kapitel 8.2) initialisiert werden. Jeder formale Parameter hat einen Gültigkeitsbereich vom Deklarationspunkt bis zum Ende des Methodenrumpfs (der ein Block ist). Die Lebensdauer ist die Dauer des Methodenaufrufs. Aus der Tatsache, dass formale Parameter wie Variablen behandelt werden, ergibt sich auch, dass innerhalb der Methode (anders als in der Mathematik) der Wert eines formalen Parameters über eine Zuweisung auch verändert werden könnte.

Die Details der Implementierung der Methode werden im Rumpf der Methode angegeben. Dort steht der exakte Algorithmus zur Berechnung des Gewünschten. Für den Nutzer der Methode ist nur die Schnittstelle (wie kann ich die Methode ansprechen) und die Funktionalität (was leistet die Methode) wichtig. Die eigentlich Implementierung der Methode, also wie konkret die Berechnung im Rumpf erfolgt, ist eigentlich für einen Nutzer der Methode irrelevant, ja sogar beliebig austauschbar, solange die Schnittstelle eingehalten und die Funktionalität erfüllt wird! Dies ist ja auch gerade so gewollt, dass im Verfeinerungsprozess an dieser Stelle die

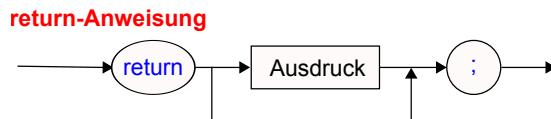


Figure 8.2: Syntaxdiagramm für die return-Anweisung

Detaillierung abbricht und man sich für die weiteren Details nicht mehr interessieren muss. Die Schnittstelle gibt an, wie man diese Methode nutzen kann, nicht, wie sie im Detail im Inneren funktioniert. Schaut man sich einen Taschenrechner an, so kann man durch Drücken der `sin`-Taste den Sinuswert für ein Argument erhalten. Wie dies intern im Taschenrechner erfolgt, ist für den Nutzer uninteressant, solange der Sinuswert mit hinreichender Genauigkeit und in akzeptabler Zeit berechnet wird. Oder haben Sie sich schon einmal Gedanken darüber gemacht, wie genau der Taschenrechner den Sinuswert berechnet?

Soll die Methode einen Wert als Ergebnis liefern, so muss dies einerseits im Methodenkopf durch die Angabe eines entsprechenden Ergebnistyps gekennzeichnet werden, also etwas anderes als `void` angegeben werden. Weiterhin muss im Methodenrumpf kenntlich gemacht werden, was der Ergebniswert ist. In Java geschieht dies in Form einer return-Anweisung, in der man nach dem Schlüsselwort `return` einen Ausdruck notiert, mit dem das Methodenergebnis erzeugt werden kann. Abbildung 8.2 zeigt die Syntax für die return-Anweisung in Java. Ist als Ergebnistypangabe im Funktionskopf `void` angegeben worden, so muss der Ausdruck nach dem Schlüsselwort `return` entfallen. Weiterhin ist zu beachten, dass eine return-Anweisung neben der Kenntlichmachung des Ergebniswerts auch das sofortige Beenden der Programmausführung in dieser Methode bedeutet. Kommt es im Kontrollfluss einer Methode zur Abarbeitung einer return-Anweisung, so werden nachfolgende Anweisungen nicht mehr ausgeführt und der Kontrollfluss würde an der aufrufenden Stelle dieser Methode weitergeführt (Details in nachfolgenden Kapiteln). Der Compiler erkennt auch eine Situation, dass nach einer return-Anweisung weitere Anweisungen stehen, und würde dies als Programmierfehler markieren. Nachfolgend einige Beispiele zu einfachen Methodendefinitionen. Arbeiten Sie diese Beispiele durch und verstehen den Zusammenhang zwischen Methodenkopf und Methodenrumpf.

#### Beispiel 8.4:

Die Funktion `quadratBerechnen`

```

8-1  public static double quadratBerechnen(double x) {
8-2      return x*x;
8-3  }
  
```

erwartet ein double-Argument und liefert als Ergebnis einen double-Wert zurück, die Quadratzahl zum übergebenen Argument. ♦

#### Beispiel 8.5:

Die Funktion `kubikBerechnen`

```

8-1  public static double kubikBerechnen(double x) {
8-2      return x * x * x;
8-3  }
  
```

erwartet ein double-Argument und liefert als Ergebnis einen double-Wert zurück, die Kubikzahl zum übergebenen Argument. ♦

Listing 8.2: Beispiel zur korrekten Definition von Methoden in einer Klasse.

```

8-1 public class MethodenBeispiele {
8-2     public static void methode1(int x) {
8-3         // ...
8-4     }
8-5
8-6     public static void main(String [] args) {
8-7         // ...
8-8     }
8-9
8-10    public static void methode2(int x) {
8-11        // ...
8-12    }
8-13 }
```

**Beispiel 8.6:**Die Funktion `maximumBerechnen`

```

8-1     public static int maximumBerechnen(int x, int y) {
8-2         if(x > y) {
8-3             return x;
8-4         } else {
8-5             return y;
8-6         }
8-7     }
```

oder auch kürzer:

```

8-1     public static int maximumBerechnen(int x, int y) {
8-2         return (x > y) ? x : y;
8-3     }
```

erwartet zwei int-Argumente und liefert als Ergebnis einen int-Wert zurück, nämlich den größeren der beiden übergebenen Werte. ♦

Methoden können in Java nur außerhalb der `main`-Methode und außerhalb jeder anderen Methode definiert werden. Eine Methode muss aber innerhalb einer Klasse (`public class XYZ { ... }`) definiert werden, die durch die beiden äußersten geschweiften Klammern begrenzt wird. Werden mehrere Methoden definiert, so ist die Reihenfolge untereinander beliebig, in der sie angegeben werden. In machen Programmiersprachen anders als in Java muss allerdings eine Methode im Programmtext definiert sein, *bevor* sie genutzt werden kann. Der *Gültigkeitsbereich* eines Methodennamens ist die gesamte Klasse.

**Beispiel 8.7:**

Listing 8.2 enthält Beispiele für korrekt definierte Methoden in einer Klasse.

Listing 8.3 enthält Beispiele für *nicht korrekt* definierte Methoden.

In den Beispielen zum Quadartwurzelverfahren wurde zu Beginn jeder Methode eine spezielle Form von Kommentar verwendet, ein Dokumentationskommentar. Mit Hilfe von solchen speziell aufgebauten Kommentaren [Java] und dem Werkzeug `javadoc`, das Teil des Java Entwicklungsumgebung ist, lässt sich aus solchen Dokumentationskommentaren ein HTML-Dokument mit Dokumentation zur Software erstellen.

Listing 8.3: Beispiel zu möglichen Fehlern bei Methodendefinitionen.

```

8-1 public class MethodenBeispiele {
8-2     public static void methode1(int x) {
8-3         // ...
8-4
8-5         // Fehler: Schachtelung von Methoden nicht erlaubt
8-6         public static void methode2(int x) {
8-7             // ...
8-8         }
8-9     }
8-10
8-11     public static void main(String [] args) {
8-12         // ...
8-13     }
8-14 }
8-15
8-16 // Fehler: Methoden ausserhalb einer Klasse nicht erlaubt
8-17 public static void methode3(int x) {
8-18     // ...
8-19 }
```

Das hat den Vorteil, dass innerhalb des Source Codes die Dokumentation enthalten ist und nicht in einem externen Dokument verwaltet wird (Die Erfahrung der letzten 50 Jahre Softwareentwicklung hat gezeigt, dass der Ansatz über ein externes Dokument nicht richtig funktioniert hat). Die gesamte Dokumentation zu allen Java Klassen, die Teil der Java Entwicklungsumgebung sind, wird auf diese Weise erstellt und ist unter [Java] zu finden.

Diese speziellen Dokumentationskommentare können vor Klassen und vor Methoden notiert werden. Weiterhin sind sie möglich vor Instanzen- und Klassenvariablen, auf die aber erst in Kapitel 11 eingegangen wird.

Der Aufbau solcher Kommentare ist so[Java], dass diese spezielle Kommentarform durch `/**` eingeleitet wird. Üblicherweise (das ist aber mit neueren Java-Versionen optional) werden nachfolgende Zeilen jeweils mit einem Stern begonnen. Abgeschlossen wird die letzte Kommentarzeile wie bei mehrzeiligen Kommentaren gewohnt mit der schließenden Kommentarkennzeichnung `*/`. Diese Kommentare sind also nur eine Spezialform der allgemeinen Kommentarform `/* ... */`. Die erste Zeile eines solchen Kommentars soll kurz und knapp (als Einzeiler halt) den Sinn und Zweck des Nachfolgenden beschreiben. Weitere Zeilen können dies weiter detaillieren. Über eine Reihe vordefinierter Tags der Form `@name` lassen sich spezielle Sachen beschreiben. Dazu gehören unter anderem:

- `@author`: Name des Autors (nur bei Klassen)
- `@param`: Parameterbeschreibung (nur bei Methoden)
- `@return`: Beschreibung des Ergebniswertes (nur bei Methoden)

Für weitere Details sei auf die Literatur verwiesen [Java].

## 8.2 Aufruf einer Methode

Nachdem eine Methode definiert ist kann sie beliebig oft im Gültigkeitsbereich der Methode in Form eines (Teil-)Ausdrucks **ausgewertet** werden. Ein anderer Begriff dafür ist auch eine Methode **aufrufen**. Der Gültigkeitsbereich eines Methodennamens ist die gesamte Klasse. Später wird vorgestellt (Kapitel 11), dass

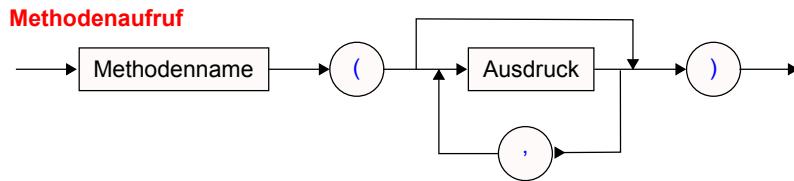


Figure 8.3: Syntaxdiagramm eines Methodenaufrufs

man in einer erweiterten Bezeichnungsweise auch über eine Klasse hinaus einen Methodennamen verwenden kann.

Ein Methodenaufruf ist syntaktisch ein Ausdruck. Abbildung 8.3 zeigt die Syntax eines Methodenaufrufs. Zu jedem Methodenaufruf gehört der Name der Methode direkt gefolgt von einer öffnenden und später schließenden runden Klammer. Zwischen den runden Klammern können jeweils in Form eines Ausdrucks und durch Kommata getrennt die **aktuellen Parameter** (andere Bezeichnung dafür: Argumente) angegeben werden. In jedem Aufruf muss die durch die Methodendefinition festgelegte Schnittstelle vollständig eingehalten werden: Die Anzahl, der Typ und die Reihenfolge der formalen Parameter der Methodendefinition und der aktuellen Parameter jedes Methodenaufrufs müssen übereinstimmen (auf kompatible Typanpassungen von Argumenten wird gleich noch eingegangen).

Zur Auswertung eines Methodenaufruf werden drei Schritte durchgeführt:

1. Die aktuellen Parameter werden jeweils in Form eines Ausdrucks angegeben. In einem ersten Schritt werden diese Ausdrücke der Reihe nach von links nach rechts ausgewertet. Das Ergebnis ist für jeden aktuellen Parameter jeweils ein Wert.
2. Anschließend wird die Methode mit diesen Parameterwerten aufgerufen. Dabei werden die formalen Parameter, die ja innerhalb der Methode wie lokale Variablen behandelt werden, mit den Werten der aktuellen Parameter initialisiert. Diese Paarung von aktuellen mit formalen Parametern findet positionsweise statt: der erste aktuelle Parameter wird mit dem ersten formalen Parameter assoziiert, der zweite aktuelle mit dem zweiten formalen und so weiter. Der Methodenrumpf wird ausgeführt, bis eine return-Anweisung ausgeführt wird oder bis das Ende der Methodenrumpfs erreicht wird (Eine Rückkehr ohne explizite return-Anweisung ist nur möglich, falls die Methode als `void` deklariert wurde). In der return-Anweisung einer nicht als `void` definierten Methode wird der Ausdruck hinter dem `return` ausgewertet. Das Ergebnis ist genau ein Wert, der als Ergebnis des Methodenaufrufs zurückgegeben wird.
3. Der Wert des Methodenaufrufs, also der im zweiten Schritt ermittelte Wert, wird an der aufrufenden Stelle statt des Methodenaufrufs zur weiteren Berechnung genutzt.

### Beispiel 8.8:

Gegeben ist der Programmausschnitt aus Listing 8.4.

Die Auswertung geschieht nun in den beschriebenen drei Schritten:

1. Die Werte der aktuellen Parameter werden bestimmt. In unserem Beispiel würde also `f(3*x, x*x)` reduziert zu `f(15, 25)` weil  $3 * x = 3 * 5 = 15$  und  $x * x = 5 * 5 = 25$  für den relevanten Wert  $x = 5$  ist.
2. Anschließend wird die Methode mit diesen beiden Parameterwerten aufgerufen: `f(15, 25)`. Im konkreten Fall bedeutet dies, dass der formale Parameter `x` mit dem Wert 15 initialisiert wird und der formale Parameter `y` mit dem Wert 25. Innerhalb der Methode kann auf diese beiden Werte über den Namen der formalen Parameter Bezug genommen werden. Dies geschieht in dem Ausdruck `x+y`, das

Listing 8.4: Beispiel zu Auswertung eines Methodenaufrufs.

```

8-1   ...
8-2   {
8-3     ...
8-4     int x = 5;
8-5     // Methodenaufruf
8-6     int y = f(3*x, x*x) + 10;
8-7     ...
8-8   }
8-9
8-10  // Methodendefinition
8-11  public static int f(int x, int y) {
8-12    return x+y;
8-13  }

```

Ergebnis dieses Ausdrucks ist also im konkreten Fall  $x + y = 15 + 25 = 40$ . Der Wert 40 wird also als Ergebnis der Methodenauswertung an die aufrufende Stelle zurück geliefert.

3. An der aufrufenden Stelle wird dieser Resultatwert 40 anstelle des Methodenaufrufs genutzt und damit der Wert des Gesamtausdrucks berechnet:  $f(3*x, x*x) + 10 = 40 + 10 = 50$ . Der Variablen **y** wird also der Wert 50 zugewiesen. ♦

Wird eine Methode mit aktuellen Parametern aufgerufen, so liefert diese Methode – falls sie terminiert – einen Wert vom spezifizierten Ergebnistyp. Dieser Wert kann dann zum Beispiel innerhalb eines umfassenden Ausdrucks weiterverwendet werden. Beispiel: `3.0 + flaeche(10, 10)`. Man kann sich also einen Methodenaufruf so vorstellen, dass dieser Aufruf innerhalb eines Ausdrucks durch den Ergebniswert dieses Methodenaufrufs im Ausdruck (textuell) ersetzt wird. Im Beispiel würde dies also bedeuten: der Ausdruck `3 + flaeche(10, 10)` würde vereinfacht zu `3.0 + 100.0` und das in einem weiteren Schritt wiederum zu `103.0`.

### Beispiel 8.9:

Eine quadratische Gleichung  $ax^2 + bx + c = 0$  hat die beiden reellen Lösungen  $x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$  und  $x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$ , falls  $b^2 - 4ac > 0$ . Zur Umsetzung dieser Formeln benötigt man die beiden Funktionen zur Quadrierung einer Zahl und zur Bestimmung der Wurzel zu einer positiven Zahl. Die Quadratzahl einer Zahl kann man direkt angeben, die Quadratwurzel einer Zahl könnte man mit dem schon beschriebenen Heron-Verfahren bestimmen. Als Ausgabe zum entsprechenden Programm in Listing 8.5 erscheint aus dem Bildschirm:

```

8-1 x1 = -0.25834261322605867
8-2 x2 = -7.741657386773941

```

Im Hauptprogramm (`main`) stehen Aufrufe der Methoden `quadratBerechnen` und `wurzelBerechnen`. Die Definition dieser beiden Methoden erfolgt dann weiter unten. Der Methodenaufruf `quadratBerechnen(b)` etwa liefert als Ergebnis den Wert  $b^2$  für den aktuellen Parameter **b**. Dieser Quadratwert wird an der Aufrufstelle in einem umfassenen Ausdruck weiter verwendet zur Berechnung der Bedingung  $b^2 - 4ac > 0$ .

Interessant ist auch der weiter unten angegebenen Aufruf `wurzelBerechnen(quadratBerechnen(b) - 4.0 * a * c)`, in dem innerhalb eines Methodenaufrufs von `wurzelBerechnen` ein weiterer Methodenaufruf von `quadratBerechnen` stattfindet. Zur Erinnerung: ein Methodenaufruf ist ein Ausdruck und ein Ausdruck kann natürlich auch zur Bildung komplexerer Ausdrücke verwendet werden. ♦

Listing 8.5: Quadratische Gleichung lösen.

```

8-1  /**
8-2   * Loesen einer quadratischen Gleichung
8-3   */
8-4  public class QuadratischeGleichung {
8-5
8-6      // hier beginnt die Programmausfuehrung
8-7      public static void main(String [] args) {
8-8          // Beispielwerte
8-9          double a = 1.0;
8-10         double b = 16.0;
8-11         double c = 2.0;
8-12
8-13         double x1, x2, w;
8-14
8-15         // Nur falls  $b^2 - 4ac > 0$  ist gibt es Loesungen
8-16         if(quadratBerechnen(b)-4.0*a*c > 0) {
8-17             // hier kommen die Formeln zur Berechnung der Loesung
8-18             w = wurzelBerechnen(quadratBerechnen(b) - 4.0 * a * c);
8-19             x1 = (-b + w) / (2.0 * a);
8-20             x2 = (-b - w) / (2.0 * a);
8-21             System.out.println("x1 = " + x1);
8-22             System.out.println("x2 = " + x2);
8-23         } else {
8-24             System.out.println("Keine reellen Loesungen vorhanden");
8-25         }
8-26     }
8-27
8-28     /**
8-29      * Die Methode berechnet die Quadratzahl zu einer Zahl
8-30      * @param x Argument
8-31      * @result liefert  $x^2$  als Ergebnis
8-32      */
8-33     public static double quadratBerechnen(double x) {
8-34         return x * x;
8-35     }
8-36
8-37     /**
8-38      * Quadratwurzelberechnung nach dem Heron-Verfahren
8-39      * @param x Argument
8-40      * @result liefert Wurzel(x) als Ergebnis
8-41      */
8-42     public static double wurzelBerechnen(double x) {
8-43         double epsilon = 1e-15;
8-44         double fehler;
8-45         // Startwert
8-46         double y = x;
8-47
8-48         do {
8-49             y = 0.5 * (y + x/y);
8-50             if(x > quadratBerechnen(y))
8-51                 fehler = x - quadratBerechnen(y);
8-52             else
8-53                 fehler = quadratBerechnen(y) - x;
8-54         } while (fehler > epsilon);
8-55
8-56         return y;
8-57     }
8-58 }
```

An dieser Stelle wird die Bedeutung des Ausdruckskonzepts klar. Wie bereits bekannt, kann ein Ausdruck in Verbindung mit Operatoren selbst wieder Teilausdrücke enthalten. Schauen Sie sich das Syntaxdiagramm zu Ausdrücken nochmals an, um diesen Zusammenhang und auch die Bedeutung eines Ausdrucks zu erkennen.

**Beispiel 8.10:**

Der Abstand zweier Punkte  $P_1 = (x_1, y_1)$  und  $P_2 = (x_2, y_2)$ , gegeben in kartesischen Koordinaten, ist  $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ . Die Ausgabe des Programms aus Listing 8.6 ist:

8-1	Abstand (2.0 ,5.0) und (1.0 ,3.0)=2.23606797749979
8-2	Abstand (1.0 ,3.0) und (0.0 ,2.0)=1.4142135623730951

Da die korrekte Nutzung von Methoden und hier insbesondere die Bedeutung der formalen und aktuellen Parameter für Programmieranfänger oft nicht einfach ist hier eine Zusammenfassung der bis jetzt behandelten wichtigsten Aspekte im Zusammenhang mit Methoden. Wichtig zum grundlegenden Verständnis von Methoden insgesamt ist die Unterscheidung zwischen der Methodendefinition und einem oder mehreren Methodenaufrufen an anderer Stelle. In einer Methodendefinition, die für eine Methode genau ein Mal vorkommt, wird im Methodenkopf die Schnittstelle definiert, über die die Nutzung dieser Methode erfolgen muss. Innerhalb des Methodenrumpfs wird der Algorithmus angegeben, der zur Berechnung des Ergebniswertes notwendig ist. Die formalen Parameter in der Methodendefinition spielen nur Platzhalterrollen für die übergebenen Werte in einem Methodenaufruf, ihr Name ist beliebig wählbar und hat nichts aber auch garnichts zu tun mit eventuell gleich lautenden Variablen in einem Methodenaufruf (siehe Beispiel gleich).

In einem Methodenaufruf wird der Name der aufgerufenen Methode angegeben gefolgt von den aktuellen Parametern. Die aktuellen Parameter sind jeweils Werte, die in Form eines Ausdrucks angegeben werden. Solch ein Ausdruck für einen Parameterwert eines Methodenaufrufs wird ausgewertet, das Resultat des Ausdrucks ist wie bekannt genau ein Wert. Mit den Werten für alle aktuellen Parameter wird dann die Methode aufgerufen. Jeder formale Parameter wird mit dem übergebenen Wert des entsprechenden aktuellen Parameters initialisiert und innerhalb des Methodenrumpfs kann auf diese Werte dann über den Namen des formalen Parameters Bezug genommen werden.

**Beispiel 8.11:**

Im Beispiel in Listing 8.7 soll nochmals verdeutlichen, dass die Namen von formalen Parametern beliebig sind. In beiden Methodenaufrufen wird  $x^2 = 5^2 = 25$  berechnet. Anschließend werden die beiden Resultatwerte der Methodenaufrufe addiert. Das **x** in **methode1**, also der Name des formalen Parameters dieser Methode, hat absolut nichts mit dem Variablenname **x** in **main** zu tun! Machen Sie sich diesen Zusammenhang klar! Wie man in **methode2** sieht, die die gleiche mathematische Funktion wie **methode1** berechnet nämlich  $x \mapsto x^2$  (oder äquivalent  $a \mapsto a^2$  oder äquivalent  $knuffi \mapsto knuffi^2$ ), könnte der formale Parameter auch **y**, **a**, **knuffi**, **einstein**, ... heißen, wenn man im Methodenrumpf der Methode den Namen auch entsprechend konsistent verwendet. ♦

Listing 8.6: Abstandsberechnung.

```

8-1  /**
8-2   * Abstand zweier Punkte berechnen
8-3   */
8-4  public class Abstand {
8-5
8-6      // hier beginnt unsere Programmausfuerhung
8-7      public static void main(String [] args) {
8-8          // Abstand berechnen zwischen den Punkten (2,5) und (1,3)
8-9          System.out.println("Abstand (2.0,5.0) und (1.0,3.0)="
8-10             + abstandBerechnen(2.0, 5.0, 1.0, 3.0));
8-11
8-12          // Abstand berechnen zwischen (1,3) und (0,2)
8-13          System.out.println("Abstand (1.0,3.0) und (0.0,2.0)="
8-14             + abstandBerechnen(1.0, 3.0, 0.0, 2.0));
8-15      }
8-16
8-17      /**
8-18       * Abstand zweier Punkte (x1,y1) und (x2,y2) berechnen
8-19       * @param x1 x-Koordinate Punkt 1
8-20       * @param y1 y-Koordinate Punkt 1
8-21       * @param x2 x-Koordinate Punkt 2
8-22       * @param y2 y-Koordinate Punkt 2
8-23       * @return Abstand dieser beiden Punkte
8-24       */
8-25      public static double abstandBerechnen(double x1, double y1,
8-26                                     double x2, double y2) {
8-27          return wurzel(quadrat(x2-x1) + quadrat(y2-y1));
8-28      }
8-29
8-30      /**
8-31       * Quadratzahl zu einer Zahl
8-32       * @param x Argument
8-33       * @return Quadratzahl zum Argument
8-34       */
8-35      public static double quadrat(double x) {
8-36          return x * x;
8-37      }
8-38
8-39      /**
8-40       * Quadratwurzel zu einer Zahl
8-41       * @param x Argument
8-42       * @return Wurzel des Arguments
8-43       */
8-44      public static double wurzel(double x) {
8-45          // hier machen wir es uns einfacher
8-46          // und benutzen die Java-eigene Wurzelmethode
8-47          return Math.sqrt(x);
8-48      }
8-49  }

```

Listing 8.7: Beliebige Namen von formalen Parametern.

```
8-1  /**
8-2   * Beispiel zu Parameternamen
8-3   */
8-4  public class Parameternamen {
8-5
8-6    /**
8-7     * berechnet den Quadratwert eines Wertes
8-8     * @param x Eingabewert
8-9     * @return liefert  $x^2$  als Ergebnis
8-10    */
8-11   public static int methode1(int x) {
8-12     return x*x;
8-13   }
8-14
8-15   /**
8-16    * berechnet den Quadratwert eines Wertes
8-17    * @param y Eingabewert
8-18    * @return liefert  $y^2$  als Ergebnis
8-19    */
8-20   public static int methode2(int y) {
8-21     return y*y;
8-22   }
8-23
8-24   public static void main(String [] args) {
8-25     int x = 5;
8-26
8-27     // hier erfolgt der Aufruf der beiden Methoden
8-28     int ergebnis = methode1(x) + methode2(x);
8-29   }
8-30 }
```

## 8.3 Überladen von Methoden

Ebenso wie bei Operatoren in Kapitel 7.3 gibt es in Java auch bei Methoden die Möglichkeit der Überladung eines Methodennamens. Dies bedeutet, dass mehrere Methoden mit gleichem Methodennamen in einer Klasse existieren können, allerdings muss es weitere Unterscheidungsmerkmale geben. Dazu dient der Begriff der **Signatur** eines Methode. Zu einer Methodensignatur gehören:

- der Name der Methode
- die Anzahl der Parameter
- die Typen der Parameter inklusive der Parameterreihenfolge

In Java gehört also weder der Ergebnistyp noch die Namen der formalen Parameter zur Methodensignatur. Methoden sind verschieden, wenn ihre Signatur unterschiedlich ist. Alle Methoden einer Klasse müssen verschieden sein, also eine unterschiedliche Signatur haben. Bei einem Methodenaufruf zu einem überladenen Methodennamen entscheidet alleine der Kontext, welche der Methoden angewandt wird.

### Beispiel 8.12:

Die Signaturen folgender Methodenköpfe sind aller verschieden:

- `public static int f(int x)`
- `public static int f(float x)` (anderer Parametertyp)
- `public static int f(int x, int y)` (Anzahl der Parameter unterschiedlich)
- `public static int g(int x, int y)` (Methodename verschieden)
- `public static int g(float x, int y)` (anderer Parametertyp)
- `public static int g(int x, float y)` (anderer Parametertyp)

Die Signaturen folgender Methoden sind gleich und dürfen damit nicht gleichzeitig in einer Klasse definiert werden (ansonsten würde der Compiler einen Fehler melden):

- `public static int f(int x, int y)`
- `public static int f(int y, int x)` (Parameternamen spielen keine Rolle)
- `public static float f(int x, int y)` (Ergebnistyp geht nicht in Signatur ein)



Das Überladen von Methoden kann man zum Beispiel dann sinnvoll einsetzen, wenn die gleiche Funktionalität für unterschiedliche Parametertypen oder für eine unterhsciedliche Anzahl an Argumenten bereit gestellt werden soll. Zu beachten ist hierbei allerdings auch, dass Java ja bereits für viele primitive Datentypen automatisch eine Typanpassung vornimmt (erweiternde Typumwandlung).

### Beispiel 8.13:

Das Beispielprogramm in Listing 8.8 zeigt das Überladen des Methodennamens `ausgeben`. Dieser Name wird in zwei Versionen definiert, einmal im Zusammenhang mit einem `int`-Argument und ein zweites mit einem `double`-Argument. Bei den Aufrufen in `main()` entscheidet der Kontext, welche Methode mit dem Namen `ausgeben` gewählt wird. In diesem Beispiel ist der Kontext die Anzahl und der Typ des Arguments / der Argumente. Zu beachten ist in dem Beispiel auch, dass für den Aufruf `ausgeben(5.0f)` zwar keine explizite Methode definiert ist, aber durch eine erweiternde Typumwandlung, die vom Compiler automatisch vorgenommen wird, das Argument in den Typ `double` umgewandelt werden kann und damit die `double`-Variante der Methode aufgerufen wird.



Listing 8.8: Überladen von Methodennamen.

```
8.1  /**
8.2   * Ueberladen einer Mtehode
8.3   */
8.4  public class Ueberladen {
8.5
8.6      public static void ausgeben(int x) {
8.7          System.out.println("int-Wert ist " + x);
8.8      }
8.9
8.10     public static void ausgeben(double x) {
8.11         System.out.println("double-Wert ist " + x);
8.12     }
8.13
8.14     public static void ausgeben(int x, double y) {
8.15         System.out.println("int-Wert ist " + x + ", double-Wert ist " + y);
8.16     }
8.17
8.18     public static void main(String[] args) {
8.19         // Aufruf mit einem int-Argument
8.20         ausgeben(5);
8.21         // Aufruf mit einem double-Argument
8.22         ausgeben(5.0);
8.23         // Aufruf mit einem float-Argument.
8.24         // Hier wird durch eine erweiternde Typanpassung die double-Variante aufgerufen
8.25         ausgeben(5.0f);
8.26         // Aufruf mit einem int- und double-Argument
8.27         ausgeben(3, 5.0);
8.28     }
8.29 }
```

**Beispiel 8.14:**

Das Programm in Listing 8.9 zeigt die korrekte Definition einer Funktion sowie auch schon die Nutzung / den Aufruf dieser Funktion, die im Detail im folgenden Kapitel behandelt wird. Die Funktion `quadratWurzelBerechnen` erwartet ein double-Argument und liefert als Ergebnis einen double-Wert zurück, die Quadratwurzel zum übergebenen Argument. Dabei wird das Verfahren von Heron genommen, das bereits früher vorgestellt wurde.

Im letzten Beispiel zum Quadratwurzelverfahren stellt sich bezüglich dieses angegebenen Heron-Verfahrens eine wichtige Frage: der Wert, zu dem der Quadratwurzelwert berechnet werden soll, ist als Parameter der Methode angegeben, was leicht einsichtig ist. Was ist aber mit epsilon, das in dem Algorithmus Einfluss darauf hat, wie genau der Ergebniswert berechnet werden soll. Je kleiner epsilon gewählt ist (Beispiel  $10^{-14}$ ), umso genauer werde ich in der Berechnung, umso länger muss ich aber in der fußgesteuerten Schleife des Algorithmus auch iterieren. Wähle ich epsilon dagegen groß (Beispiel  $10^{-1}$ ), umso weniger genau werde ich eventuell im Ergebniswert, aber umso schneller bekomme ich das Ergebnis. Man könnte also epsilon als weiteren Parameter dieser Methode einführen, mit dem der Nutzer dieser Methode die Genauigkeit und damit verbunden den Berechnungsaufwand steuern könnte. Die erste Methodenversion zur Quadratwurzelberechnung mit "eingebautem" epsilon wäre damit eher für einen unerfahrenen Nutzer geeignet, der sich also nicht mit den Details des Algorithmus auseinandersetzen will und wo in der Methode ein epsilon ohne Zutun des Nutzers gewählt würde, durch das die maximal erreichbare (oder spezifizierte) Genauigkeit auf jeden Fall erreicht wird. Die zweite Methodenversion, bei der der Nutzer einen epsilon-Wert mit angeben kann und muss, wäre dann eher für den erfahrenen Nutzer gedacht, der die Bedeutung und die Auswirkungen von epsilon in diesem Algorithmus kennt. In Java gibt es die Möglichkeit, beide Methoden gleichzeitig angeben zu können, um so einem Nutzer die Alternative geben zu können. Die entsprechende Programmversion mit beiden Alternativen zusammen wäre dann wie in Listing 8.10 zu sehen.

Grundsätzlich sollten als Parameter einer Methode die Variablen gewählt werden, mit denen von außen die Methode gesteuert werden soll. Im Beispiel hätte man demnach die Auswahl, ob man eine Methode mit "eingebauter" Genauigkeit wählen möchte (ohne Einflussmöglichkeiten auf die Genauigkeit) oder ob man die Genauigkeit mit spezifizieren möchte. Auf ihrem Taschenrechner werden Sie vermutlich nur die Version auf einer Taste haben, die ohne äußere Einflussmöglichkeiten zur Genauigkeit den Wurzelwert berechnet, da der "normale" Nutzer eines Taschenrechners sich keine Gedanken über solche Sachen machen möchte beziehungsweise qualifiziert machen kann.

Listing 8.9: Berechnung der Quadratwurzel.

```

8-1  /**
8-2   * Methode zur Quadratwurzelberechnung nach dem Heron-Verfahren
8-3   */
8-4  public class QuadratWurzelBerechnung {
8-5
8-6      public static void main(String [] args) {
8-7
8-8          // berechne Quadratwurzel von 8157
8-9          double wurzel = quadratWurzelBerechnen(8157.0);
8-10     }
8-11
8-12     /**
8-13      * Methode zur Berechnung der Quadratwurzel einer positiven Zahl.
8-14      * Der Ergebniswert ist innerhalb einer Genauigkeit von 10^-6
8-15      * zum exakten Wert.
8-16      * @param x Wert, zu dem der Wurzelwert berechnet werden soll
8-17      * @return Quadratwurzelwert zu x
8-18      */
8-19      public static double quadratWurzelBerechnen(double x) {
8-20          // gewuenschte Genauigkeit
8-21          double epsilon = 0.000001;
8-22          // Fehlerabschaetzung zum korrekten Wert
8-23          double fehler;
8-24          // derzeitiger Annaeherungswert y_n
8-25          double y;
8-26
8-27          // Startwert angeben
8-28          // Dies koennte ein beliebiger Wert sein. Probieren Sie es aus!
8-29          y = x;
8-30
8-31          do {
8-32              // Berechnungsvorschrift fuer bessere Annaeherung
8-33              y = 0.5 * (y + x/y);
8-34
8-35              // Absolutbetrag fuer Fehlerabschaetzung
8-36              if(x > y*y)
8-37                  fehler = x - y*y;
8-38              else
8-39                  fehler = y*y - x;
8-40
8-41          } while(fehler > epsilon);
8-42
8-43          // Ergebnis liefern
8-44          return y;
8-45      }
8-46  }

```

Listing 8.10: Berechnung der Quadratwurzel überladen.

```

8-1  /**
8-2   * Methode zur Quadratwurzelberechnung nach dem Heron-Verfahren
8-3   */
8-4  public class QuadratWurzelBerechnung2 {
8-5
8-6      public static void main(String [] args) {
8-7
8-8          // berechne Quadratwurzel von 8157 mit einer Genauigkeit von 10^-6
8-9          double wurzell = quadratWurzelBerechnen(8157.0, 1e-6);
8-10
8-11         // berechne Quadratwurzel von 8157 mit "eingebauter" Genauigkeit
8-12         double wurzel2 = quadratWurzelBerechnen(8157.0);
8-13     }
8-14
8-15     /**
8-16      * Methode zur Berechnung der Quadratwurzel einer positiven Zahl.
8-17      * Der Ergebniswert ist innerhalb einer Genauigkeit von 10^-6
8-18      * zum exakten Wert.
8-19      * @param x Wert, zu dem der Wurzelwert berechnet werden soll
8-20      * @return Quadratwurzelwert zu x
8-21      */
8-22      public static double quadratWurzelBerechnen(double x) {
8-23          // gewuenschte Genauigkeit
8-24          double epsilon = 0.000001;
8-25
8-26          // wir nutzen die andere Methode mit unserer Genauigkeitsvorgabe
8-27          return quadratWurzelBerechnen(x, epsilon);
8-28      }
8-29
8-30     /**
8-31      * Methode zur Berechnung der Quadratwurzel einer positiven Zahl.
8-32      * @param x Wert, zu dem der Wurzelwert berechnet werden soll
8-33      * @param epsilon Vorgabe der Berechnungsgenauigkeit
8-34      * @return Quadratwurzelwert zu x
8-35      */
8-36      public static double quadratWurzelBerechnen(double x, double epsilon) {
8-37          // Fehlerabschaetzung zum korrekten Wert
8-38          double fehler;
8-39          // derzeitiger Annaehlerungswert y_n
8-40          double y;
8-41
8-42          // Startwert angeben
8-43          // Dies koennte ein beliebiger Wert sein. Probieren Sie es aus!
8-44          y = x;
8-45
8-46          do {
8-47              // Berechnungsvorschrift fuer bessere Annaehlerung
8-48              y = 0.5 * (y + x/y);
8-49
8-50              // Absolutbetrag fuer Fehlerabschaetzung
8-51              if(x > y*y)
8-52                  fehler = x - y*y;
8-53              else
8-54                  fehler = y*y - x;
8-55
8-56          } while(fehler > epsilon);
8-57
8-58          // Ergebnis liefern
8-59          return y;
8-60      }
8-61  }

```

## 8.4 Rekursiv definierte Methoden

Bevor die Details diskutiert werden, die bei einem Methodenaufruf intern passieren, soll ein Spezialfall von Methodenaufrufen betrachtet werden, nämlich dass die aufgerufenen Methode die eigene Methode ist. Man spricht in einem Fall auch von einer rekursiv definierten Methode (lateinisch *recurrere* zurücklaufen).

In der Mathematik (und wie gleich zu sehen ist in der Informatik genauso) ist es nicht ungewöhnlich, dass man zur Definition einer mathematischen Funktion auf diese selber zurückgreift. Beispielsweise kann man die bereits früher vorgestellte Fakultätsfunktion  $fakultaet : \mathbb{N} \rightarrow \mathbb{N}$  mit  $fakultaet(n) = \prod_{i=1}^n i = 1 \cdot 2 \cdot 3 \cdots n$  auch angeben als:

$$fakultaet(n) = \begin{cases} 1 & \text{falls } n < 2 \\ n \cdot fakultaet(n-1) & \text{sonst} \end{cases}$$

In dieser Funktionsdefinition ist in sehr knapper und auch natürlicher Weise das Gesamtproblem auf die Berechnung eines kleineren aber gleich gearteten Teilproblems zurückgeführt. In Kapitel 16.5 wird in anderem Zusammenhang nochmals auf solche rekursiv definierte Methoden eingegangen und man sieht dort, wie man das Konzept der Rekursion in bestimmten Fällen als allgemeine Problemlösungsstrategie einsetzen kann.

Viele Studierende haben beim ersten Anblick einer derart definierten rekursiven Funktion allerdings Verständnisprobleme. Diese sind unbegründet (was hoffentlich auch im Folgenden klar wird), wenn man bei einer Funktionsauswertung einfach *ignoriert*, dass die aufgerufene Funktion auf der rechten Seite der Definitionsgleichung den gleichen Namen hat wie die Funktion, die auf der linken Seite definiert wird.

Denn will man den Funktionswert wie beispielsweise bei der oben definierten Fakultätsfunktion an der Stelle  $n = 4$  ermitteln, muss man die definierende formelmäßige Beziehung auflösen, indem man einen Funktionsaufruf *jeweils* durch die rechte Seite dieser Funktion ersetzt und dabei alle formalen Parameter darin durch die aktuellen Parameterwerte ersetzt. Sollten wie in unserem Beispiel mehrere Fälle in der Definitionsgleichung auf der rechten Seite möglich sein, so werden diese streng sequentiell von oben nach unten durchlaufen, bis der erste zutreffende Fall eintrifft. Sollten in der daraus entstehenden Formel weitere Funktionsaufrufe vorhanden sein, so muss man diese zuerst weiter auflösen.

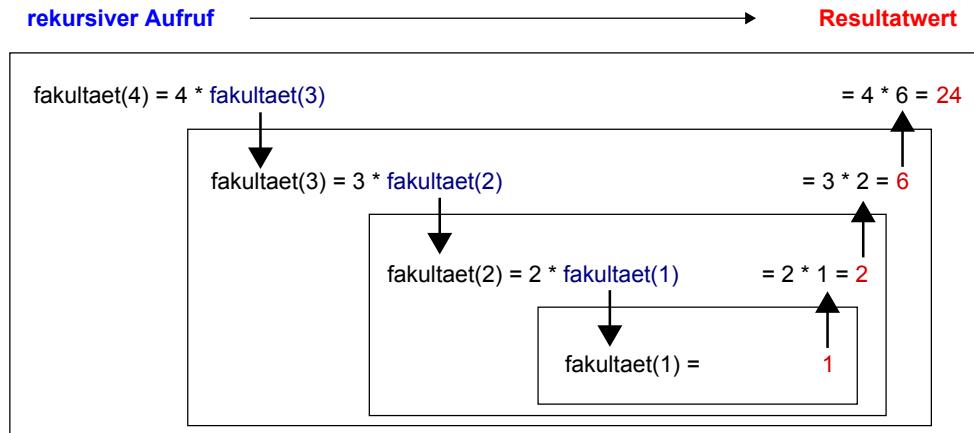
### Beispiel 8.15:

Damit ergibt sich beispielsweise für den Aufruf von  $fakultaet(4)$ :

$$\begin{aligned} fakultaet(4) &= 4 \cdot fakultaet(3) \\ &= 4 \cdot 3 \cdot fakultaet(2) \\ &= 4 \cdot 3 \cdot 2 \cdot fakultaet(1) \\ &= 4 \cdot 3 \cdot 2 \cdot 1 \\ &= 24 \end{aligned}$$

In Abbildung 8.4 ist die Auswertung des Methodenaufrufs `fakultaet(4)` schematisch abgebildet. Wie man dort sieht, entsteht eine Hierarchie von noch in Bearbeitung befindlichen Methodenaufrufen, die bei `fakultaet(1)` abbricht und in jeder Stufe einen Wert liefert, der auf den nächsthöheren Stufe verwendet wird. Bevor der Wert der Stufe  $n$  nicht vorliegt kann Stufe  $n - 1$  nicht weiterarbeiten! Ein nachfolgendes Kapitel wird sich mit der damit zusammenhängenden Problematik widmen.

Weitere Beispiele für rekursive Funktionsdefinitionen sind:

Figure 8.4: Aufrufhierarchie bei Auswertung von `fakultaet(4)`

- Die Fibonacci-Funktion  $fib : \mathbb{N} \rightarrow \mathbb{N}$  ist wie folgt definiert:

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n) &= fib(n-2) + fib(n-1) \text{ für } n > 1 \end{aligned}$$

- Der Binomialkoeffizient  $\binom{n}{k} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  mit  $\binom{n}{k} := \frac{n!}{k!(n-k)!}$  kann ebenfalls rekursiv definiert werden durch:

$$\binom{n}{k} = \begin{cases} 0 & \text{falls } n < k \\ 1 & \text{falls } k = 0 \text{ oder } n = k \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{sonst} \end{cases}$$

Bevor Sie weitermachen rechnen Sie auf einem Blatt Papier streng schematisch aus, was der Wert von  $fib(3)$  und  $\binom{3}{1}$  ist!

Solche rekursiven Funktionsdefinitionen lassen sich in modernen Programmiersprachen direkt umsetzen in eine entsprechende Methode.

### Beispiel 8.16:

Wie Sie an den Methoden in Listing 8.11 sehen können, kann die mathematische Definition Eins zu Eins übernommen werden. ♦

Eine Methode wird rekursiv aufgerufen, wenn entweder die Methode – wie in den obigen Beispielen – sich selber direkt aufruft (**direkte Rekursion**) oder aber in Form einer **indirekten Rekursion**, dass eine Methode `g` eine Methode `f` aufruft und diese (eventuell auf weiteren Umwegen) wiederum Methode `f`.

### Beispiel 8.17:

Dass Rekursion auch geschachtelt werden kann, zeigt die Ackermann-Funktion (aus der Theoretischen

Listing 8.11: Direkte Umsetzung rekursiv definierter Funktionen in Java.

```

8-1  /**
8-2   * Beispiele rekursive definierter Methoden
8-3   */
8-4  public class RekursiveMethoden {
8-5
8-6      public static void main(String[] args) {
8-7          // in einer Schleife Methodenwerte berechnen
8-8          for(int i=0; i<10; i++) {
8-9              // Aufruf Fakultaetsfunktion
8-10             System.out.println("fakultaet(\"+i\")== " + fakultaet(i));
8-11
8-12              // Aufruf Fibonacci-Funktion
8-13             System.out.println("fibonacci(\"+i\")== " + fibonacci(i));
8-14
8-15              // Aufruf von n ueber k
8-16             System.out.println(""+i+" ueber 1=" + nUeberK(i, 1));
8-17         }
8-18     }
8-19
8-20     /**
8-21      * Fakultaetsfunktion
8-22      * @param n Argument
8-23      * @return n!
8-24      */
8-25      static int fakultaet(int n) {
8-26          if (n == 0) {
8-27              return 1;
8-28          } else {
8-29              return n * fakultaet(n-1);
8-30          }
8-31      }
8-32
8-33     /**
8-34      * Fibonacci-Funktion
8-35      * @param n Argument
8-36      * @return Wert der Fibonacci-Folge an der Stelle n
8-37      */
8-38      static int fibonacci(int n) {
8-39          if (n == 0) {
8-40              return 0;
8-41          } else if (n == 1) {
8-42              return 1;
8-43          } else {
8-44              return fibonacci(n-2) + fibonacci(n-1);
8-45          }
8-46      }
8-47
8-48     /**
8-49      * n ueber k
8-50      * @param n Argument 1
8-51      * @param k Argument 2
8-52      * @return (n ueber k)
8-53      */
8-54      static int nUeberK(int n, int k) {
8-55          if (n < k) {
8-56              return 0;
8-57          } else if ((k == 0) || (n == k)) {
8-58              return 1;
8-59          } else {
8-60              return nUeberK(n-1, k-1) + nUeberK(n-1, k);
8-61          }
8-62      }
8-63  }

```

Listing 8.12: Ackermann-Funktion.

```

8-1  /**
8-2   * Ackermann Funktion
8-3   */
8-4  public class Ackermann {
8-5
8-6      public static void main(String [] args) {
8-7          for (int n=0; n<5; n++)
8-8              for (int m=0; m<5; m++)
8-9                  System.out.println("ack("+n+", "+m+")=" + ack(n,m));
8-10     }
8-11
8-12     /**
8-13      * Ackermann Funktion
8-14      * @param m erster Parameterwert
8-15      * @param n zweiter Parameterwert
8-16      * @return Wert der Ackermann Funktion an der Stelle m,n
8-17     */
8-18     public static int ack(int m, int n) {
8-19         if (m == 0) {
8-20             return n+1;
8-21         } else if (n == 0) {
8-22             return ack(m-1, 1);
8-23         } else {
8-24             return ack(m-1, ack(m, n-1));
8-25         }
8-26     }
8-27 }
```

Informatik). Die Ackermannfunktion  $Ackermann : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  ist definiert durch:

$$\begin{aligned}
Ackermann(0, n) &= n + 1 \\
Ackermann(m, 0) &= Ackermann(m - 1, 1) \\
Ackermann(m, n) &= Ackermann(m - 1, Ackermann(m, n - 1))
\end{aligned}$$

Auch diese geschachtelte Rekursion lässt sich direkt umsetzen in einer Methode (siehe Listing 8.12). Weshalb diese unscheinbare Ackermann-Funktion eine wichtige Rolle in der Theoretischen Informatik spielt erkennen Sie eventuell durch das "Rumspielen" mit dieser Funktion. Rufen Sie die Ackermann einmal, wie im Beispielprogramm angegeben, für verschiedene Werte für  $m$  und  $n$  auf. Beginnen Sie mit kleinen Werten wie etwa  $(0,0)$  und schauen was passiert, wenn der Wert für  $m$  größer als 3 wird (was ja auch keine so große Zahl ist). ♦

## 8.5 Strategien der Parameterübergabe

In diesem Kapitel sollen die Detail betrachtet werden, wie die Übergabe von aktuellen Parameterwerten an eine aufgerufene Methoden vonstatten geht. Diese Diskussion soll an einem konkreten Beispiel geführt werden, das schon vorher in etwas ausführlicherer Form vorgestellt wurde.

In der Methode `abstandBerechnen` in Listing 8.13 zur Berechnung des Abstands zweier Punkte findet ein geschachtelter Aufruf der Methoden `wurzel` und `quadrat` statt. Spätestens an dieser Stelle stellt sich für

Listing 8.13: Abstandsberechnung.

```

8-1  /**
8-2   * Abstand zweier Punkte berechnen (Version 2)
8-3   */
8-4  public class Abstand2 {
8-5
8-6      // hier beginnt unsere Programmausfuerhung
8-7      public static void main(String[] args) {
8-8          // Abstand berechnen zwischen den Punkten (2,5) und (1,3)
8-9          double a = abstandBerechnen(2.0, 5.0, 1.0, 3.0);
8-10     }
8-11
8-12 /**
8-13  * Abstand zweier Punkte (x1,y1) und (x2,y2) berechnen
8-14  */
8-15  public static double abstandBerechnen(double x1, double y1,
8-16          double x2, double y2) {
8-17      return wurzel(quadrat(x2-x1) + quadrat(y2-y1));
8-18  }
8-19
8-20 /**
8-21  * Quadratzahl zu einer Zahl
8-22  */
8-23  public static double quadrat(double x) {
8-24      return x * x;
8-25  }
8-26
8-27 /**
8-28  * Quadratwurzel zu einer Zahl
8-29  */
8-30  public static double wurzel(double x) {
8-31      // hier machen wir es uns einfacher
8-32      // und benutzen die Java-eigene Wurzelmethode
8-33      return Math.sqrt(x);
8-34  }
8-35 }
```

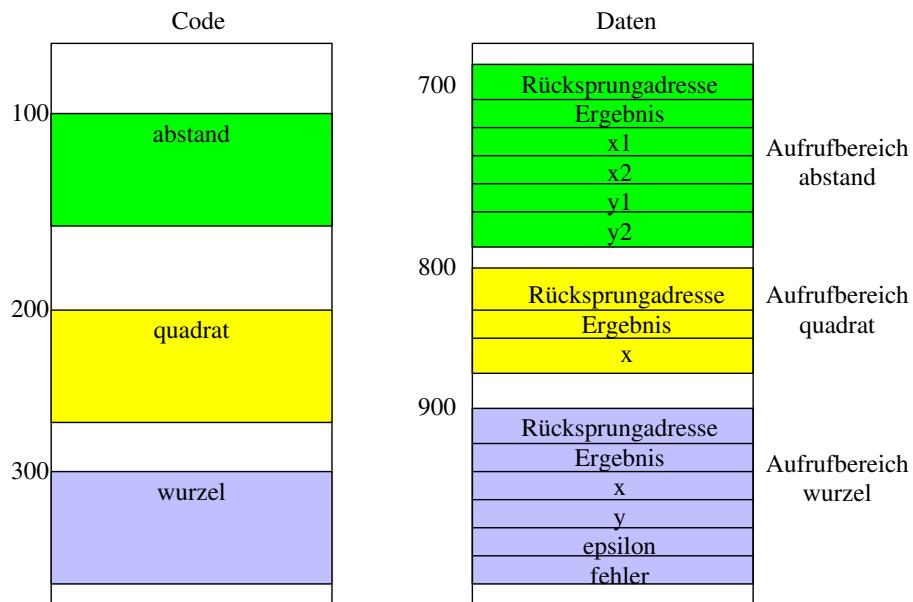


Figure 8.5: Statische Aufrufbereiche

eine Auswertung auf einem konkreten Rechner die Frage, wie denn nun diese Auswertung erfolgen kann, das heißt mit welchen Mechanismen ein Compiler Maschinencode erzeugen kann, der diese Berechnung vornimmt beziehungsweise ein Interpreter die Ausführung des Programms vornimmt.

Wird eine Methode aufgerufen, so muss klar definiert sein, wie die aktuellen Parameter übergeben werden, wie die Methode ihr Ergebnis berechnet, wie dieses Ergebnis an den Aufrufer übergeben wird und wo mit der Ausführung der Berechnung fortgefahrene werden muss.

Dazu soll ein Ausschnitt aus dem letzten Beispielprogramm betrachtet werden. Hinter der return-Anweisung in der Methode `abstandBerechnen` steht der Ausdruck, über den das Ergebnis der Methode berechnet wird. In dem Ausdruck sind weitere Methodenaufrufe von `wurzel` und `quadrat` enthalten. In Java ist genau definiert, wie diese Auswertung zu erfolgen hat: von innen nach außen und von links nach rechts. Bei geschachtelten Methodenaufrufen muss also zuerst der innere Aufruf vollständig ausgewertet werden, bevor der äußere Aufruf ausgewertet wird. Ebenso werden, falls ein Methodenaufruf mehrere Argumente hat, die Argumentausdrücke streng sequentiell von links nach rechts vollständig ausgewertet. Um im Beispiel die Wurzelmethode aufzurufen, werden also zuerst die Argumente ausgewertet, im Beispiel genau ein Argument gegeben durch den Ausdruck `quadrat(x2-x1) + quadrat(y2-y1)`. In unserem Beispiel bedeutet dies, dass zuerst `quadrat(x2-x1)` ausgewertet wird, anschließend `quadrat(y2-y1)` und dann die beiden resultierenden Werte addiert.

Hier gilt wieder die gleiche Auswertestrategie, zuerst werden die Argumente ausgewertet. Angenommen, das Ergebnis des Ausdrucks `x2-x1`, also des Arguments von `quadrat`, sei die Zahl 5, im Rechner als 32 Bit Zahl in Zweierkomplementdarstellung repräsentiert.

In einem Rechner liegt neben den Daten auch der Code aller Methoden als Teil des gesamten ausführbaren Programmes im Hauptspeicher (von Neumann Modell). In Abbildung 8.5 ist dies grafisch dargestellt. Der Code der Methode `abstand` liege zum Beispiel ab Adresse 100 im Speicher, der Code von `quadrat` ab Adresse 200 und der Code von `wurzel` ab Adresse 300. Kommt man in der Ausführung des Programmcodes in der Methode `abstand` nun an diese Stelle, wo der Wert der Methode `quadrat` angewandt auf den Wert 5 berechnet werden soll (im Beispiel sei dies der Code an der Speicheradresse 127), so hat der Compiler an dieser Stelle Maschinencode erzeugt, der die Programmausführung an anderer Stelle weitermachen lässt, nämlich zu Beginn

der Methode `quadrat`, im Beispiel an Adresse 200. Es wird also an der Aufrufstelle in der aufrufenden Methode `abstand` *nicht* der Code der aufgerufenen Methode eingefügt, sondern lediglich ein Sprung an den Anfang des Codes der Methode `abstand`. Bei der URM hatte man schon den URM-Befehl `goto 1` für eine Speicheradresse *l* kennengelernt. Ähnliche Befehle gibt es auch in Maschinensprachen realer Prozessoren. Dieses Vorgehen hat u.a. den Vorteil, dass bei mehreren Aufrufen der gleichen Methode der Code dieser Methode nur einmal vorhanden sein muss.

Bevor aber mit der Ausführung des Programmcodes für `quadrat` begonnen werden kann, müssen noch zwei Sachen geklärt werden:

1. Woher weiß die aufgerufene Methode `quadrat`, was ihr als Argument übergeben wurde und wo sie das Ergebnis der Methode an die aufrufende Methode übergeben soll?
2. Woher weiß die aufgerufene Methode oder der Rechner, wo in der Programmausführung nach Beendigung der Methode `quadrat` fortgefahren werden soll?

Zur Lösung beider Probleme wird zu jeder Methode Speicherplatz reserviert, in dem diese Informationen übergeben werden: Der **Aufrufbereich** oder *activation record* einer Methode (siehe Abbildung 8.5). An einer Stelle im Speicher, die sowohl der aufrufenden Methode als auch der aufgerufenen Methode bekannt ist, werden die aktuellen Übergabeparameter abgespeichert. Im Beispiel ist dies die mit `x` gekennzeichnete Speicherstelle des Aufrufbereichs von `quadrat`, in der die Zahl 5 vor Ausführung des Sprungbefehls abgespeichert wird. Die gleiche Speicherstelle ist innerhalb der Funktion `quadrat` unter dem Parameternamen `x` bekannt.

Das zweite oben angesprochene Problem tritt dann auf, wenn der Code der aufgerufenen Methode `quadrat` abgearbeitet ist und der Ergebniswert zurückgegeben werden soll. Wo soll die Programmausführung fortgesetzt werden? Sicherlich nicht hinter dem Code von `quadrat`, sondern eigentlich muss man ja dort fortfahren, wo man durch den Methodenaufruf unterbrochen wurde. Um dort mit der Programmausführung weiterzumachen, muss man die entsprechende Speicheradresse wissen. Aus diesem Grund enthält jeder Aufrufbereich einen Eintrag **Rücksprungadresse**, in den vor dem Aufruf der Methode die Speicheradresse abgelegt wird, an der später mit der Ausführung fortgefahrene werden soll.

Ebenfalls muss der Ergebniswert an die aufrufende Methode übergeben werden. Dies geschieht ebenfalls im Aufrufbereich, wo die aufgerufene Methode in einem reservierten Speicherplatz den Ergebniswert abspeichert und die aufrufende Methode anschließend den Ergebniswert abhol. Der Aufrufbereich ist also gewissermassen ein Postkasten, in den eine Partei etwas einwirft und die andere Partei abholt. Weiterhin werden neben der Rücksprungadresse, den Aufrufparametern und dem Ergebniswert in einem Aufrufbereich auch Speicher für alle lokalen Variablen einer Methode angelegt.

In Abbildung 8.6 ist für den Aufruf der Methode `quadrat` mit dem aktuellen Argument 5 innerhalb der Methode `abstand` gezeigt, welche Schritte gemacht werden müssen. Bevor die Methode `quadrat` aufgerufen wird, wird das aktuelle Argument 5 in den Aufrufbereich an die Stelle des ersten (und einzigen) Parameters `x` kopiert. Dann wird die Rücksprungadresse im Aufrufbereich abgelegt, in diesem Fall die nächste Adresse nach dem Aufrufbefehl. An dieser Stelle soll nach Ausführung der Methode `quadrat` mit der Programmausführung fortgefahrene werden. Nachdem die Vorarbeiten getan sind, wird die Methode aufgerufen, indem an die Adresse verzweigt wird, an der der Code der Methode abgelegt ist. Nachdem die Methode `quadrat` einen Wert berechnet hat (hier  $5*5=25$ ), wird der Ergebniswert im Aufrufbereich in dem dafür vorgesehenen Bereich abgespeichert. Anschließend muss noch zur aufrufenden Methode zurückgesprungen werden, was über die Rücksprungadresse geschieht, indem ein Register mit der Adresse geladen wird und indirekt zu dieser Adresse gesprungen wird (im Beispiel: `goto r1`, das heißt springe zu der Speicheradresse, die in Register R1 angegeben ist).

Die Parameterübergabe wurde im letzten Abschnitt so beschrieben, dass für jeden aktuellen Parameter in einem Methodenaufruf der Wert berechnet wird, dieser Wert an die entsprechende Position im Aufrufbereich

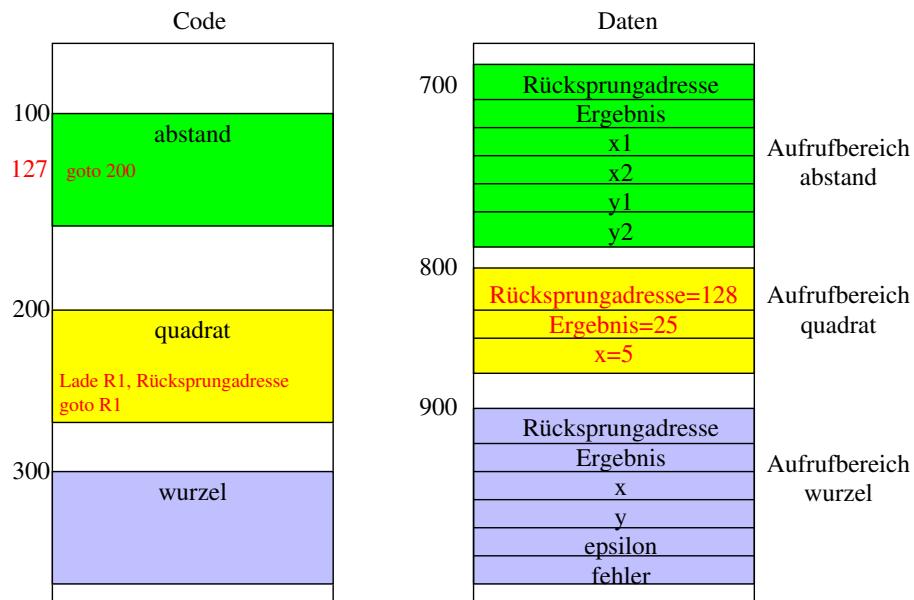


Figure 8.6: Aufrufbereiche bei Aufruf von quadrat

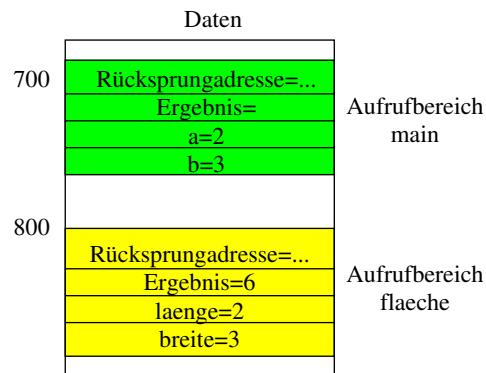


Figure 8.7: Aufrufbereiche bei Aufruf von flaeche(a,b)

kopiert wird und dann die Methode aufgerufen wird, indem zum Anfang des Maschinencodes der aufgerufenen Methode verzweigt wird. Diese Art der Parameterübergabe nennt man **call-by-value** oder **Wertübergabe**.

### Beispiel 8.18:

Das Beispiel der Flächenberechnung wird so abgeändert, dass statt Zahlen zwei Variablen als aktuelle Parameter genommen werden, die an die Methode `flaeche` übergeben werden (siehe Listing 8.14). Die Ausgabe des Programms ist:

8-1 Die Flaeche des Rechtecks ist 6

In Abbildung 8.7 ist der Aufrufbereich für diesen Aufruf von `flaeche` gezeigt. Entsprechend der *call-by-value*-Strategie werden also die *Werte* der Variablen ermittelt und *Kopien* dieser Werte an die dafür vorgesehenen Stellen im Aufrufbereich von `flaeche` kopiert. ◆

Listing 8.14: Beispiel zu Übergabestrategien.

```

8-1 /**
8-2 * Uebergabestrategien
8-3 */
8-4 public class Uebergabestrategien {
8-5     public static void main(String [] args) {
8-6         int a=2, b=3;
8-7
8-8         // Flaechenberechnung fuer a und b
8-9         System.out.println("Die Flaeche des Rechtecks ist " + flaeche(a,b));
8-10    }
8-11
8-12    // Hier ist die Methode flaeche
8-13    public static int flaeche(int laenge, int breite) {
8-14        return laenge * breite;
8-15    }
8-16 }
```

Listing 8.15: Beispiel 2 zu Übergabestrategien.

```

8-1 /**
8-2 * Uebergabestrategien (Version 2)
8-3 */
8-4 public class Uebergabestrategien2 {
8-5     public static void main(String [] args) {
8-6         int a=2, b=3;
8-7
8-8         // Vertauschen des Inhalts der beiden Variablen a und b
8-9         System.out.println("a=" + a + ", b=" + b);
8-10        swap(a,b);
8-11        System.out.println("a=" + a + ", b=" + b);
8-12    }
8-13
8-14    public static void swap(int x, int y) {
8-15        int tmp;
8-16
8-17        tmp = x;
8-18        x = y;
8-19        y = tmp;
8-20        System.out.println("in swap: a=" + x + ", b=" + y);
8-21        return;
8-22    }
8-23 }
```

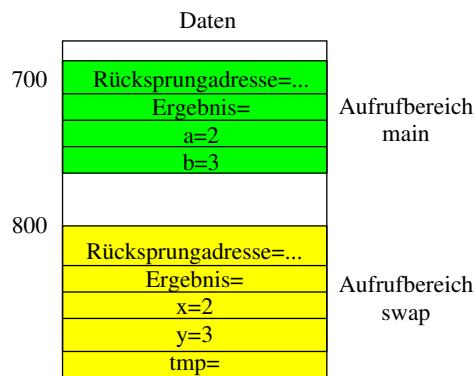


Figure 8.8: Aufrufbereiche bei Aufruf von swap(a,b)

**Beispiel 8.19:**

Nun soll ein zweites Beispiel betrachtet werden (Listing 8.15), in dem eine Methode entwickelt werden soll, die den Inhalt zweier beliebiger Variablen vertauscht. Die Idee ist dabei, dass man über eine dritte Hilfsvariable den Inhalt der beiden Variablen zirkulär vertauscht, das heißt nach dem Muster `tmp = x; x = y; y = tmp`. Die Ausgabe des Programms ist:

```

8-1 a=2, b=3
8-2 in swap: a=3, b=2
8-3 a=2, b=3

```

Hier ist also nicht das geschehen, was man sich gewünscht hätte: das Vertauschen des Inhalts der beiden Variablen `a` und `b`. Der Grund liegt darin, dass – entsprechend der *call-by-value*-Strategie – die Werte der beiden Variablen `a` und `b` ermittelt werden und *Kopien der Werte* übergeben werden. Innerhalb der Methode `swap` werden dann die Kopienwerte verändert (wie die mittlere Ausabezeile auch belegt), allerdings nicht die ursprünglichen Variablen `a` und `b` des aufrufenden Methode `main`.

In Abbildung 8.8 ist der Aufrufbereich für den Aufruf von `swap` gezeigt. Innerhalb der Methode `swap` werden zwar die beiden Werte vertauscht, aber diese Änderung wirkt sich nicht auf die ursprünglichen Variablen `a` und `b` aus, da eine *Kopie* der Variableninhalte übergeben wurde. Mit der Übergabestrategie für Parameter *call-by-value* kann man also nicht den Inhalt der beiden Variablen vertauschen.

Als Ursache des Problems ist also erkannt worden, dass Kopien der Werte der Variablen `a` und `b` übergeben wurden. Um eine Methode `swap` programmieren zu können, die den Inhalt der Variablen verändert, müssten aber eigentlich die Variablen selbst übergeben werden. Eine Lösung dieser Problematik für die bisher bekannten einfachen Java-Datentypen wie `int` oder `float` ist prinzipiell machbar, bedürfte an dieser Stelle aber erheblicher Zusatzerläuterungen, die vom eigentlichen Problem ablenken würden. Wir werden später die Java-Lösung sehen, an dieser Stelle soll die Programmiersprache Fortran (steht für *Formula Translator*) als Anschaubungsbeispiel dienen.

**Beispiel 8.20:**

Die Syntax für das swap-Beispiel ist in Fortran etwas anders, aber die Bedeutung des Programms ist sicherlich einfach durch die Zuweisungen und Schlüsselworte zu erkennen.

```

8-1 program main
8-2   integer a, b
8-3   a = 2
8-4   b = 3

```

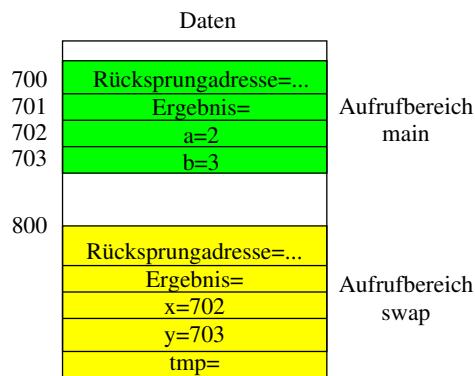


Figure 8.9: Aufrufbereiche bei Aufruf von swap(a,b) mit der Strategie call-by-reference

```

8-5   print a,b          ! erste Zeile der Ausgabe
8-6   call swap(a,b)
8-7   print a,b          ! zweite Zeile der Ausgabe
8-8 end
8-9
8-10 subroutine swap(x,y)
8-11   integer x, y
8-12   integer tmp
8-13   tmp = x
8-14   x = y
8-15   y = tmp
8-16   return
8-17 end

```

Die Ausgabe des Programms ist:

```

8-1 2 3
8-2 3 2

```

Wie man an der Ausgabe sehen kann, sind also tatsächlich die Inhalte der beiden Variablen **a** und **b** vertauscht worden. ♦

Die Frage stellt sich jetzt: Was ist in Fortran anders als in Java? Denn die beiden Programme in Fortran und Java sehen sehr ähnlich aus. Wie schon vorher gesehen, wird zur Übergabe von Methodenargumenten in Java das Verfahren *call-by-value* genommen. In Fortran ist dies anders, das Verfahren zur Parameterübergabe in Fortran heißt **call-by-reference**. Während bei *call-by-value* eine Kopie des Parameterwertes in den Aufrufbereich abgelegt wird, wird bei *call-by-reference* eine **Referenz (Zeiger)** auf den eigentlichen Wert in den Übergabebereich der aufgerufenen Methode abgelegt. Solch ein Verweis gibt die Hauptspeicheradresse an, an der der eigentliche Wert zu finden ist. Das heißt also, statt direkt den Wert selber anzugeben, wird die Hauptspeicheradresse übergeben, an der der Wert zu finden ist (indirekte Angabe).

Abbildung 8.9 zeigt den Aufrufbereich, wie er für den Methodenaufruf von **swap** in Fortran aussieht. Anstatt wie im Java-Programm die Werte 2 und 3 zu übergeben, werden hier die Adressen 702 und 703 übergeben, an denen die Werte zu finden sind. Will man auf die Werte zugreifen, muss man also indirekt über diese Adressen gehen. Wie im obigen Fortran-Programm zu sehen ist, muss sich der Fortran-Programmierer um diese Indirektion nicht kümmern. Der Fortran-Compiler sorgt dafür, dass in der Methode **swap** (in Fortran heißt eine Methode *Subroutine*) bei Verwendung einer Parameters (zum Beispiel **x** für den ersten formalen Parameter) der eigentliche Wert nur über den Verweis ermittelt werden kann. Dazu gibt es spezielle Maschinenbefehle, die

Listing 8.16: Fakultätsberechnung.

```

8-1  /**
8-2   * Fakultaet rekursiv
8-3   */
8-4  public class Fakultaet {
8-5
8-6      /**
8-7       * Fakultaetsfunktion
8-8       * @param n Argument
8-9       * @return n!
8-10      */
8-11     static int fakultaet(int n) {
8-12         int wert = 0;
8-13         if (n == 0) {
8-14             return 1;
8-15         } else {
8-16             wert = fakultaet(n-1);
8-17             return n * wert;
8-18         }
8-19     }
8-20
8-21     public static void main(String [] args) {
8-22         System.out.println("fakultaet(2)=" + fakultaet(2));
8-23     }
8-24
8-25 }
```

für eine vorgegebene Adresse  $a$  den Inhalt der Adresse  $a$  ermitteln. Im Beispiel würde der Compiler für jeden Zugriff auf den Parameter  $\mathbf{x}$  Maschinencode erzeugen, der den Wert 702 in ein Register lädt und anschließend den indirekten Ladebefehl anwendet, so dass der Wert aus der Adresse 702 ausgelesen wird. Weil dies für jeden Zugriff auf  $\mathbf{x}$  und  $\mathbf{y}$  in der Methode `swap` gilt, bedeutet dies aber auch, dass in einer Zuweisung der Form  $\mathbf{x} = \mathbf{y}$  und  $\mathbf{y} = \mathbf{tmp}$  der Wert der *ursprünglichen Variablen*  $\mathbf{x}$  beziehungsweise  $\mathbf{y}$  verändert wird. Die Erklärung für die Tatsache, dass in Java die `swap`-Methode nicht den gewünschten Effekt hat, in Fortran die Methode jedoch das Gewünschte leistet, liegt in den unterschiedlichen Übergabestrategien für Methodenargumente: *call-by-value* in Java und *call-by-reference* in Fortran. Die übergebenen Adressen, im Beispiel 702 und 703, verweisen auf die eigentlichen Werte. Man spricht deshalb von **Referenz** oder **Zeigern**.

In C/C++ und Java ist eine indirekte Parameterübergabe auch möglich (wird aber hier nicht weiter verfolgt), so dass das `swap`-Beispiel in einer modifizierten Form ebenfalls das gewünschte Ergebnis liefern würde.

Für rekursive Aufrufe ist das bisher gewählte Schema zur Übergabe von Aufrufparametern und zur Speicherung der Rückkehradresse in einem statischen Aufrufbereich für jede Methode nicht mehr möglich. Würde dieses Schema beibehalten, so würden bei einem rekursiven Aufruf die alten Werte (inkorrekt erweise) überschrieben. Schon 1960 mit der Sprache Algol60 hatte man sich Gedanken über diese Problematik gemacht und die Lösung darin gefunden, dass man einen **Laufzeitstack** verwenden kann. Anstatt wie bisher die Verwaltungsinformationen in einem statischen Bereich abzulegen, verwendet man nun einen Stack, der zu jedem Funktionsaufruf – und nicht mehr zu jeder Funktion selber – die Verwaltungsinformationen auf einem Stack abspeichert: einen **dynamischen Aufrufbereich**.

Als Beispiel zur Veranschaulichung soll die Fakultätsfunktion dienen, wie sie bereits in leicht veränderter Form verwendet wurde.

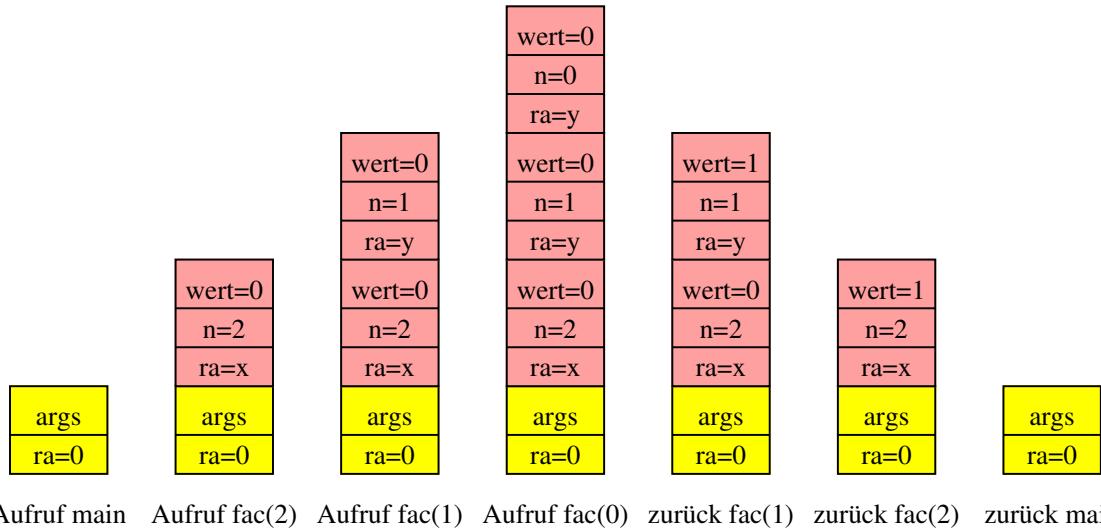
**Beispiel 8.21:**

Figure 8.10: Laufzeitstack für fac(2)

In Abbildung 8.10 ist der dynamisch wachsende und abnehmende Laufzeitstack zur Programmausführung vom Programm in Listing 8.16 gezeigt. Das Hauptprogramm `main` wird dabei vom Betriebssystem (oder Laufzeitsystem), das das Programm insgesamt startet, wie ein ganz normaler Methodenaufruf behandelt. Das Betriebssystem legt auch für das Hauptprogramm die Übergabeparameter (bei `main` ein Argument `args`, was hier nicht weiter betrachtet werden soll, und die Rücksprungadresse (mit `ra` gekennzeichnet) auf dem Laufzeitstack ab.

In `main` wird dann die Fakultätsfunktion `fac` mit dem Argument 2 aufgerufen. Dies bewirkt, dass auf dem Laufzeitstack ein neuer dynamischer Aufrufbereich angelegt wird mit der Rücksprungadresse `ra=x` (für Position x) und dem Aufrufwert `n=2`. Zusätzlich wird für die lokale Variable `wert` auf dem Laufzeitstack Speicher angelegt. In `fac(2)` wird rekursiv an Position y `fac(1)` aufgerufen, was wiederum einen dynamischen Aufrufbereich auf dem Laufzeitstack anlegt. Die Rückkehradresse ist diesmal die Position y, das Aufrufargument ist `n=1` und es wird wiederum eine neue lokale Variable `wert` auf dem Stack angelegt. In `fac(1)` wird `fac(0)` aufgerufen, entsprechend wieder Rückkehradresse, Aufrufargument und lokale Variable auf dem Stack abgelegt und im Aufruf von `fac(0)` bricht die Rekursion ab. Dieser Aufruf liefert den Wert 1 als Ergebniswert. Durch die return-Anweisung wird der oberste Aufrufbereich vom Stack gelöscht, zu der Rücksprungadresse verzweigt und dort mit der Ausführung weitergemacht. Dies bedeutet, dass die Variable `wert` dieses `fac`-Aufrufs den Ergebniswert 1 erhält und in der return-Anweisung den Wert `1*1` als Funktionsergebnis liefert. Auch hier wird wieder der dynamische Aufrufbereich vom Stack gelöscht, zur Rücksprungadresse verzweigt und dort mit der Ausführung weitergemacht. Für den Aufruf von `fac(2)` liefert die Funktion demzufolge `2*1` als Ergebnis zurück, so dass nach Rückkehr im Hauptprogramm das Ergebnis von `fac(2)=2` und ein bereinigter Stack vorliegt. ♦

Was geschieht beim Beispielprogramm in Listing 8.17? Der Aufruf `aufruf(1)` im Hauptprogramm bewirkt einen weiteren Aufruf von `aufruf(2)`, dieser einen Aufruf von `aufruf(3)` und so weiter, also eine nie abbrechende Rekursion. Man erkennt die Problematik auch bei der Formulierung der mathematischen Funktion, die dieses Java-Programm beschreibt:

$$aufruf(n) = aufruf(n+1)$$

Listing 8.17: Was passiert?

```

8-1  /**
8-2   * Testprogramm
8-3   */
8-4 public class WasPassiert {
8-5     static int aufruf( int n ) {
8-6       return aufruf(n+1);
8-7     }
8-8
8-9     public static void main( String [ ] args ) {
8-10       System.out.println("aufruf(1) = " + aufruf(1));
8-11     }
8-12 }
```

Man muss also bei einer rekursiven Definition einer Funktion beziehungsweise bei der Programmierung einer rekursiven Methode dafür sorgen, dass die Rekursion abbricht. Rekursive Beschreibungen dienen ja in erster Linie dazu, komplexe Fälle auf einfache Fälle zurückzuführen. Man muss also für den Basisfall eine explizite Lösung angeben und komplexere Fälle auf einfachere Fälle zurückführen, die letztendlich auf den (oder die) Basisfall zurückgeführt werden können. Die Fakultätsfunktion war dafür ein gutes Beispiel, wo die Rekursion  $fac(n - 1)$  abbricht, wenn  $n = 0$ :

$$fac : \mathbb{N} \rightarrow \mathbb{N} = \begin{cases} fac(0) &= 1 \\ fac(n) &= n * fac(n-1) \text{ falls } n \geq 1 \end{cases}$$

Für die Fakultätsfunktion kann man einfach über vollständige Induktion zeigen, dass die Rekursion abbricht und wie viele rekursive Aufrufe die Methode `fac` mit dem Aufrufargument  $n$  erzeugt. Bei der Ackermann-Funktion ist es durch blosses Hinsehen schon sehr schwierig, die Anzahl der rekursiven Aufrufe für beliebige Argumente  $n$  und  $m$  anzugeben. Es ist jedoch nicht möglich, für jede beliebige rekursiv definierte Methode (oder Funktion) und für beliebige Aufrufargumente  $x_1, \dots, x_n$  die (maximale) Anzahl der rekursiven Aufrufe a priori anzugeben (vergleiche Theoretische Informatik).

Aus diesem Grund ist es auch nicht möglich, beim Starten eines beliebigen Programms die maximale Größe des Laufzeitstacks für dieses Programm anzugeben. Das Betriebssystem muß aber den für ein Programm zur Verfügung stehenden Speicher (beziehungsweise korrekter den virtuellen Adressraum des Prozesses) aufteilen. Unter Unix geschieht dies, indem der Speicher in vier Bereiche aufgeteilt wird, von denen zwei eine feste Größe haben (statisch) und zwei dynamisch wachsen beziehungsweise schrumpfen können (Abbildung 8.11):

1. Im **Code-Bereich** wird der Maschinencode des Programms abgelegt. Für jedes Programm hat der Code-Bereich eine andere Größe, diese Größe ist aber für jedes Programm fest, es wird ja zur Laufzeit des Programms kein neuer Code erzeugt oder weggenommen.
2. Im **statische Datenbereich** – wie der Name schon sagt ist dieser Bereich ebenfalls in der Größe für ein Programm fixiert – werden Daten abgelegt, die zu Beginn des Programmlaufs angelegt werden und bis zum Ende des Programmlaufs existieren. Das Schlüsselwort `static`, das vorher schon oft im Zusammenhang mit Methodendefinitionen in einem Java-Programm benutzt wurde, hat im Zusammenhang mit Variablen u.a. Einfluss darauf, ob eine Variable in diesem statischen Datenbereich oder im folgenden Laufzeitstack abgelegt wird.
3. Der schon angesprochene **Laufzeitstack** muss in der Größe variabel sein, da ja zu Programmstart nicht bekannt ist, wie das dynamische Aufrufverhalten des Programms sein wird, das heißt wie viele dynamische Aufrufbereiche abgelegt werden müssen. Der Laufzeitstack wächst und schrumpft deshalb mit den Methodenaufrufen.

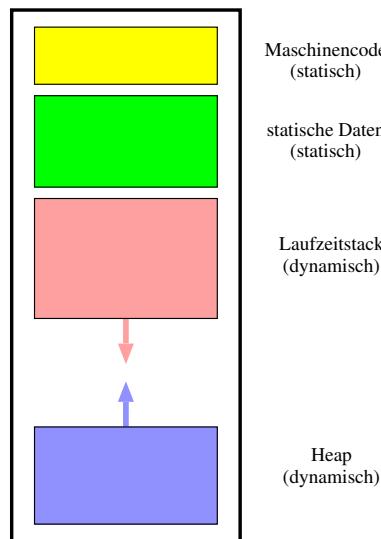


Figure 8.11: Speicheraufteilung unter Unix

4. Die uns bis jetzt bekannten Daten (zum Beispiel int, float, char, boolean) sind einfache Objekte, deren Wert direkt in einer Variablen abgespeichert wird, das heißt im statischen Datenbereich oder in einem dynamischen Aufrufbereich auf dem Laufzeitstack. Später werden auch komplexere Objekte vorgestellt, die nicht mehr direkt in Variablen gespeichert werden, sondern die dynamisch – bei Bedarf – neu geschaffen werden und für die in Variablen dann ein indirekter Verweis (Zeiger) abgespeichert wird. Für diese Objekte ist ein weiterer Speicherbereich nötig, der entsprechend der Erzeugung und des Löschens dieser Objekte wachsen und schrumpfen kann. Den dafür nötigen Speicherbereich nennt man **Heap**.

## 8.6 Umwandlung von Rekursion in Iteration

Rekursive Aufrufe sind aufgrund der Speicherung von Verwaltungsinformationen auf dem Laufzeitstack relativ aufwändig. Für jeden rekursiven Methodenaufruf muss auf dem Laufzeitstack ein neuer Aufrufbereich angelegt werden, die Werte der aktuellen Parameter dorthin kopiert werden, die Methode aufgerufen werden und so weiter. Für bestimmte Rekursionsmuster lässt sich eine äquivalente iterative Formulierung finden, die weitaus effizienter abgearbeitet werden kann.

### Beispiel 8.22:

Oben war bereits die rekursive Funktionsdefinition für die Summe  $s_n = \sum_{i=1}^n i$  gezeigt worden:

$$\begin{aligned} S(0) &= 0 \\ S(n) &= S(n-1) + n \end{aligned}$$

Die rekursive Funktion als Java-Methode notiert sieht folgendermassen aus:

```

8-1  static int S( int n ) {
8-2      if (n == 0)
8-3          return 0;
8-4      else
8-5          return S(n-1) + n;
8-6  }
```

Die Berechnung des Funktionswertes für ein Argument  $k = 4$  war die Umsetzung des Rekurrenzformel:

$$\begin{aligned}
 S(4) &= S(3) + 4 \\
 &= S(2) + 3 + 4 \\
 &= S(1) + 2 + +4 \\
 &= S(0) + 1 + 2 + 3 + 4 \\
 &= 0 + 1 + 2 + 3 + 4 \\
 &= 10
 \end{aligned}$$

In der rekursiven Berechnung wird der Wert "von hinten nach vorne" ausgerechnet. Eine gleichwertige Berechnung lässt sich aber auch "von vorne nach hinten" abgeben, was einer iterativen Version des Algorithmus entspricht:

```

8-1   static int S(int n) {
8-2     int i = 0;
8-3     int s = 0;
8-4
8-5     while (i < n) {
8-6       s = s + i;
8-7       i = i + 1;
8-8     }
8-9     return s;
8-10    }
```

Schaut man sich hier den Verlauf des Algorithmus an, so erkennt man, wie sich die Gesamtsumme "von vorne nach hinten" aufbaut.

Wert i	Wert s	entspricht
0	0	S(0)
1	1	S(1)
2	3	S(2)
3	6	S(3)
4	10	S(4)



### Beispiel 8.23:

Die Eulersche Zahl  $e$  ist als Grenzwert der Folge  $a_n$  definiert:

$$e = \lim_{n \rightarrow \infty} a_n$$

mit

$$a_n = 1 + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!}$$

oder als Rekurrenzformel:

$$\begin{aligned}
 a_0 &= 1 \\
 a_n &= a_{n-1} + \frac{1}{n!}
 \end{aligned}$$

Für eine vorgegebene Genauigkeit  $\varepsilon$  lässt sich über diese Formel eine Näherung  $\bar{e}$  angeben. Dazu muss man solange Folgeglieder aufaddieren, bis für zwei Folgeglieder  $a_n$  und  $a_{n-1}$  gilt:  $a_n - a_{n-1} < \varepsilon$ .

Eine rekursive Lösung sähe also wie in Listing 8.18 angegeben aus. Die Ausgabe des Programms ist:

Listing 8.18: Annäherungsweise Berechnung von e (rekursiv).

```

8-1  /**
8-2   * Berechnung von e
8-3   */
8-4  public class AnnaeherungE {
8-5      /** Berechnung von n!
8-6       * @param n Basiswert
8-7       * @return n!
8-8       */
8-9      static int fac( int n) {
8-10         if (n == 0)
8-11             return 1;
8-12         else
8-13             return n * fac(n-1);
8-14     }
8-15
8-16     /** Berechnung von a(n)
8-17      * @param n Wert, bis zu dem berechnet werden soll
8-18      * @result a(n)
8-19      */
8-20     static double reihe(int n) {
8-21         if (n == 0)
8-22             return 1.0;
8-23         else
8-24             return (1.0 / fac(n)) + reihe(n-1);
8-25     }
8-26
8-27     /** Berechnung einer Approximation an e
8-28      * @param epsilon Genauigkeitsparameter
8-29      * @return Annaeherung an e
8-30      */
8-31     static double e(double epsilon) {
8-32         int n = 2;
8-33         double summe = 0.0;
8-34
8-35         while(reihe(n) - reihe(n-1) >= epsilon) {
8-36             System.out.println(n + ". Annaeherung an e ist " + reihe(n));
8-37             n = n + 1;
8-38         }
8-39         return reihe(n-1); // wir haben eben eins zu weit gezaehlt
8-40     }
8-41
8-42     public static void main(String[] args) {
8-43         double epsilon = 0.00001;
8-44         System.out.println("Annaeherung an e ist " + e(epsilon));
8-45     }
8-46 }
```

```

8-1 prompt> javac Annaeherung_e.java
8-2 prompt> java Annaeherung_e
8-3 2. Annaeherung an e ist 2.5
8-4 3. Annaeherung an e ist 2.6666666666666665
8-5 4. Annaeherung an e ist 2.708333333333333
8-6 5. Annaeherung an e ist 2.7166666666666663
8-7 6. Annaeherung an e ist 2.718055555555554
8-8 7. Annaeherung an e ist 2.7182539682539684
8-9 8. Annaeherung an e ist 2.71827876984127
8-10 Annaeherung an e ist 2.71827876984127

```

Man sieht an diesem Beispielprogramm, das eine direkte Umsetzung der Rekurrenzformel ist, dass einige Berechnungen wiederholt stattfinden (zum Beispiel `reihe(n)`) und die Rekursion ebenfalls eliminiert werden könnte.

Die iterative Version sieht wie in Listing 8.19 angegeben aus. Die Ausgabe des Programms ist:

```

8-1 prompt> javac Annaeherung_iterativ_e.java
8-2 prompt> java Annaeherung_iterativ_e
8-3 2. Annaeherung an e ist 2.0
8-4 3. Annaeherung an e ist 2.5
8-5 4. Annaeherung an e ist 2.6666666666666667
8-6 5. Annaeherung an e ist 2.708333333333335
8-7 6. Annaeherung an e ist 2.7166666666666667
8-8 7. Annaeherung an e ist 2.718055555555556
8-9 8. Annaeherung an e ist 2.7182539682539684
8-10 9. Annaeherung an e ist 2.71827876984127
8-11 Annaeherung an e ist 2.71827876984127

```

Bei der iterativen Programmversion sieht man, wie man die Rekursion in der Fakultätsfunktion und der Reihenberechnung eliminiert sowie unnötige Berechnungen der Reihenglieder durch das Zwischenspeichern in Variablen optimieren kann.

Das Programm lässt sich noch weiter vereinfachen, was an aber an dieser Stelle nicht gemacht werden soll. ♦

Nach diesen Beispielen stellt sich die Frage: Kann man jede rekursiv definierte Methode (Funktion) automatisch in eine äquivalente Schleife, das heißt Iteration, umwandeln und umgekehrt?

Zuerst soll die umgekehrte Fragestellung betrachtet werden: Kann man eine beliebige Iteration in Rekursion umwandeln? Die Antwort auf diese Frage ist: Ja! Generell lässt sich *jede* while-Schleife in eine äquivalente Rekursion umwandeln und dies kann man einfach nach folgendem Schema machen. Für eine while-Schleife der Form

```

8-1 while Bedingung
8-2     Anweisung

```

kann man eine äquivalente Funktion **schleifeRekursiv** formulieren:

```

8-1 SchleifeRekursiv() {
8-2     if (Bedingung) {
8-3         Anweisung;
8-4         SchleifeRekursiv();
8-5     }
8-6 }

```

Zu beachten sind hierbei allerdings Sichtbarkeit und Lebensdauer von Variablen. Alle rekursiven Aufrufe müssen bei diesem einfachen Umsetzungsschema auf die gleichen Variablen zugreifen.

Listing 8.19: Annäherungsweise Berechnung von e (iterativ).

```

8-1  /**
8-2   * Berechnung von e (iterativ)
8-3   */
8-4  public class AnnaeherungEIterativ {
8-5      /** Berechnung von n!
8-6       * @param n Basiswert
8-7       * @return n!
8-8       */
8-9      static int fac( int n ) {
8-10         int resultat = 1;
8-11         while(n > 0) {
8-12             resultat = resultat * n;
8-13             n = n - 1;
8-14         }
8-15         return resultat;
8-16     }
8-17
8-18     /** Berechnung von a(n)
8-19      * @param n Wert, bis zu dem berechnet werden soll
8-20      * @result a(n)
8-21      */
8-22     static double reihe(int n) {
8-23         double summe = 1.0;
8-24         while(n > 0) {
8-25             summe = summe + (1.0 / fac(n));
8-26             n = n - 1;
8-27         }
8-28         return summe;
8-29     }
8-30
8-31     /** Berechnung einer Approximation an e
8-32      * @param epsilon Genauigkeitsparameter
8-33      * @return Annaeherung an e
8-34      */
8-35     static double e(double epsilon) {
8-36         int n = 2;
8-37         double reihe_n , reihe_n1 , summe = 0.0;
8-38
8-39         reihe_n1 = reihe(0);
8-40         reihe_n = reihe(1);
8-41         while(reihe_n - reihe_n1 >= epsilon) {
8-42             System.out.println(n + ". Annaeherung an e ist " + reihe_n);
8-43             reihe_n1 = reihe_n;
8-44             reihe_n = reihe(n);
8-45             n = n + 1;
8-46         }
8-47         return reihe_n1; // wir haben oben eins zu weit gezaehlt
8-48     }
8-49
8-50     public static void main(String[] args) {
8-51         double epsilon = 0.00001;
8-52         System.out.println("Annaeherung an e ist " + e(epsilon));
8-53     }
8-54 }
```

Listing 8.20: Beispiel zur Umwandlung von Iteration nach Rekursion.

```

8-1  /**
8-2   * Beispiel zur Umwandlung Rekursion Iteration
8-3   */
8-4  public class SchleifenRekursiv {
8-5      // Da wir nach dem Schema keine Variablen an die Funktion uebergeben,
8-6      // brauchen wir diese Variablen an dieser Stelle.
8-7      static int i = 0;
8-8      static int summel = 0, summe2 = 0;
8-9
8-10     static void SchleifeR() {
8-11         if (i < 10) {
8-12             summe2 = summe2 + i;
8-13             i = i + 1;
8-14             SchleifeR();
8-15         }
8-16     }
8-17
8-18     public static void main(String[] args) {
8-19         i = 0;
8-20         while(i < 10) {
8-21             summel = summel + i;
8-22             i = i + 1;
8-23         }
8-24         i = 0;
8-25         SchleifeR();
8-26         System.out.println("summel= " + summel + " , summe2=" + summe2);
8-27     }
8-28 }
```

**Beispiel 8.24:**

Die Ausgabe des Programms aus Listing 8.20 ist:

```

8-1 prompt> javac SchleifeRekursiv.java
8-2 prompt> java SchleifeRekursiv
8-3 summel= 45 , summe2=45
```

Nachdem (ohne formalen Beweis) gezeigt wurde, dass jede Schleife in eine äquivalente rekursive Funktion überführt werden kann, stellt sich jetzt die ursprüngliche Frage, ob der umgekehrte Weg auch möglich ist: Die Umwandlung einer beliebigen rekursiven Funktion in eine iterative Form. Und hier ist die Antwort: Nein! Nur eine spezielle Klasse von rekursiven Funktionen, die endrekursiven Funktionen (*tail recursive*), ist generell in ein äquivalentes Iterationskonstrukt umwandelbar.

Endrekursive Funktionen haben als letzte Aktion im Programmcode den rekursiven Aufruf. Man kann dies für eine Funktion  $f : \mathbb{N} \rightarrow \mathbb{N}$  auch mit einem Rekursionsschema angeben:

$$f(x) = \begin{cases} g(x) & \text{falls } P(x) \\ f(h(x)) & \text{sonst} \end{cases}$$

wobei  $g$  und  $h$  beliebige Funktionen sind mit  $g, h : \mathbb{N} \rightarrow \mathbb{N}$  und  $P : \mathbb{N} \rightarrow \mathbb{B}$  ein Prädikat. Da  $h(x)$  im sonst-Fall zuerst ausgewertet wird (angenommen mit Ergebniswert  $y \in \mathbb{N}$ ), erfolgt der Aufruf  $f(y)$  als letzte Aktion im sonst-Fall. Solch eine endrekursive Funktion lässt sich übersetzen in eine äquivalente iterative Version:

```

8-1 public static int f (x) {
8-2     while ! P(x)
```

```

8-3     x = h(x);
8-4     return g(x);
8-5 }
```

Dies ist zum Beispiel bei dem Ersetzungsschema der while-Schleife durch eine rekursive Funktion sichtbar. Als letzte Aktion in der Funktion **SchleifeRekursiv** erfolgt der rekursive Aufruf.

### Beispiel 8.25:

Die rekursiv definierte Methode

```

8-1 static int ulam(int n) {
8-2     if (n == 1)
8-3         return n;
8-4     else {
8-5         if (n % 2 == 0)
8-6             n = n / 2;
8-7         else
8-8             n = 3 * n + 1;
8-9     }
8-10    return ulam(n);
8-11 }
```

lässt sich nach obigem Schema umwandeln in:

```

8-1 static int ulam2(int n) {
8-2     while (!(n == 1)) {
8-3         if (n % 2 == 0)
8-4             n = n / 2;
8-5         else
8-6             n = 3 * n + 1;
8-7     }
8-8     return n;
8-9 }
```

Als Konsequenz daraus folgt sofort, dass Rekursion als Kontrollstruktur mächtiger ist als Iteration, da jede while-Schleife – wie gesehen – durch eine rekursive Funktion ersetzt werden kann, jedoch der umgekehrte Weg nicht generell möglich ist.

Die Fibonacci-Funktion, die als Beispiel für eine rekursiv definierte Funktion im letzten Abschnitt angeführt wurde, lässt sich ebenfalls als iterative Lösung angeben, wenn auch nicht mit den oben angegebenen automatischen Umsetzungsmustern.

Rekursive Verfahren sind oft – wenn man die Rekursion an sich verstanden hat – einfacher zu verstehen als ein entsprechendes iteratives Verfahren und die Lösungen sind oft auch "eleganter". Der Nachteil rekursiver Lösungen ist meist die erhöhte Laufzeit, da die Ausführung rekursiver Methoden mit höherem Verwaltungsaufwand (Erzeugen des Aufrufbereichs etc.) verbunden ist. Ebenso haben wir aber auch am Beispiel der Berechnung der Fibonacci-Funktion gesehen, dass eine einfache elegante Lösung über Rekursion nicht notwendigerweise die bessere (schnellere) sein muss.

## 8.7 Zusammenfassung und Hinweise

### Verstehen

Methoden sind ein wichtiges Strukturierungsmittel in Programmen. In Methoden werden Teilaufgaben gelöst, die mehrfach verwendet werden können. Die Umsetzung von Methodenaufrufen geschieht mit Hilfe des Laufzeitstacks, auf dem pro Aufruf ein Aufrufbereich als Verwaltungsinformation angelegt wird.

### Kurz und knapp merken

- Methoden stellen ein Strukturierungsmittel für Programme bereit.
- Die nach außen / für einen Nutzer sichtbare Schnittstelle einer Methode wird im Methodenkopf angegeben.
- Die meist nur für den Implementierer relevante Realisierung der Methode / Berechnung wird im Methodenrumpf angegeben.
- In einem Methodenrumpf kann an jeder Stelle durch eine return-Anweisung die Methodenberechnung beendet werden und ein Ergebnis dieses Methodenaufrufs angegeben werden.
- Bei einem Methodenaufruf werden die Argumentausdrücke von links nach rechts ausgewertet und dann erst die Methode mit diesen Werten aktiviert.
- Nach Beenden einer Methodenberechnung wird die Ausführung an der Stelle fortgesetzt, von der der Aufruf erfolgte
- Methodennamen können überladen werden. Die Signatur aller Methoden muss sich dabei unterscheiden.
- Rekursive Funktionsdefinitionen sind in Java möglich, ohne dass die Rekursion explizit behandelt werden müsste. Die Auswertung geschieht analog einer "normalen" Methodenauswertung.
- Rekursive Lösungen können einen (erheblichen) Verwaltungsaufwand bei der Auswertung eines Aufrufs nach sich ziehen.
- Jede iterativ formulierte Lösung lässt sich nach einem festen Verfahren in eine äquivalente rekursive Lösung überführen. Der umgekehrte Weg ist allgemein nicht möglich. Nur für endrekursive Formulierungen lässt sich ein festes Übersetzungsschema angeben.
- Es existieren für Daten drei Speicherbereiche, die gewisse Eigenschaften haben (insbesondere wie sie verwaltet werden, ...). Programmiersprachen enthalten Konstrukte, um eine Aufteilung von Daten / Variablen auf diese Bereiche zu beeinflussen.
- Mit jedem Methodenaufruf werden Verwaltungsinformationen auf dem Laufzeitstack abgelegt.
- Call-by-value und call-by-reference sind Strategien der Parameterübergabe mit unterschiedlicher Auswirkung.
- Java nutzt *immer* die Call-by-Value Strategie für die Parameterübergabe.

### Häufige Fehler

- Bei Verwendung von Rekursion muss durch entsprechende Definition der Methode dafür gesorgt werden, dass jede Berechnung irgendwann einmal abbricht.

## Übungsfragen

- Wieso hat man Methoden eingeführt?
- Welche Möglichkeiten der Strukturierung von Programmen gibt es in Java?
- Wie ist eine Methode in Java aufgebaut?
- Was ist der Gültigkeitsbereich eines Methodenamens?
- Was ist die Gültigkeit, Sichtbarkeit und Lebensdauer von formalen Parametern einer Methode?
- Wann macht als Ergebnistyp einer Methode void Sinn?
- Was ist die Ausgabe: `void f(long v){ System.out.println(v); } ... f(3); f((byte)3); f((byte)3.5);`
- Was ist die Ausgabe: `void f(int v){ System.out.println(v); } ... int x=3; f(x++); f(++x); f(++x + x++);`
- Was ist die Ausgabe:  
`void f(int v){ Systemn.out.println(v); } ... int x=255; f((byte)(++x + x++));`
- Was gibt man in einem Dokumentskommentar einer Methode an?
- Was versteht man unter dem Überladen eines Methodenamens? Wozu nutzt man es? Beispiel!
- Welche Variante eines überladenen Methodenamens wird beim Aufruf ausgewählt?
- Was versteht man unter einer rekursiven Funktion / Methode?
- Wie sind normalerweise rekursive Methode (inhaltlich) angegeben?
- Geben Sie ein Beispiel für eine rekursive Methodedefinition an!
- Wie erfolgt intern die Auswertung: `int f(int x){ int y = x; return f(++x + y++); } ... f(3)`
- Vor-/Nachteile rekursiver Lösungen?
- Kann ich jede iterative Lösung in eine rekursive umwandeln? Wenn ja, wie?
- Kann ich jede rekursive Lösung in eine iterative umwandeln? Wenn ja, wie?
- In welche Bereiche ist der Hauptspeicher bzgl. einer Programmausführung aufgeteilt?
- Wozu dient der statische Datenbereich?
- Wozu dient der Laufzeitstack?
- Wozu dient der Heap?
- Was sind die Eigenschaften der Daten, die im statischen Datenbereich abgelegt sind?
- Was sind die Eigenschaften der Daten, die im Laufzeitstack abgelegt sind?
- Was sind die Eigenschaften der Daten, die im Heap abgelegt sind?
- Welche Speichersegmente sind in ihrer Größe fest und welche dynamisch?
- Was ist ein Aufrufbereich?
- Wann wird ein neuer Aufrufbereich angelegt?
- Wie kann es z.B. vorkommen, dass die Größe eines Laufzeitstacks nicht mehr ausreicht? Geben Sie ein konkretes Beispiel an!
- Was ist call-by-value?
- Was ist call-by-reference?
- Wo ist der Unterschied zwischen call-by-value und call-by-reference?
- Geben Sie ein Beispiel für ein Problem an, was mit call-by-reference, aber nicht mit call-by-value lösbar ist!

## Reflektion des Stoffs

- Geben Sie einen analogen Java-Methodenkopf zur mathematischen Funktion  $f : \mathbb{N} \times B \rightarrow \mathbb{R}$  ( $B$  soll für die Wahrheitswerte stehen).

- Geben Sie eine vollständige Java-Methode an zur mathematischen Funktion  $f : \mathbb{N} \times B \rightarrow \mathbb{R}$  mit

$$f(x, b) = \begin{cases} x & \text{falls } b \\ x + 0.5 & \text{sonst} \end{cases}$$

- Werten Sie auf dem Papier aus: **nÜberK(2, 1)**
- Rekursion ist also mächtiger als Iteration und jede iterative Formulierung lässt sich in eine rekursive Formulierung übersetzen, aber nicht umgekehrt. Wieso lässt man dann die Iteration als Programmierkonzept nicht komplett aus Programmiersprachen raus?
- Gibt es Programme, für die man eine maximale Stack-Größe angeben kann?
- Könnte man für jedes Programm eine maximale Stack-Größe angeben?
- Könnte man eine Stack-Größe angeben, die für alle Programme reichen würde?

## **Chapter 9**

# **Komplexere Datentypen**

In diesem Kapitel werden die bereits bekannten primitiven Datentypen um weitere Datentypen erweitert. Abbildung 9.1 zeigt die vollständige Struktur aller Java-Datentypen, wovon Feldtypen in diesem Kapitel behandelt werden.

Es sei schon vorab angemerkt, dass alle neu dazukommenden nicht-primitiven Datentypen nicht mehr zu der Kategorie "primitiv" zählen, und damit auch erhebliche Konsequenzen verbunden sind, die nachfolgend dann auch diskutiert werden.

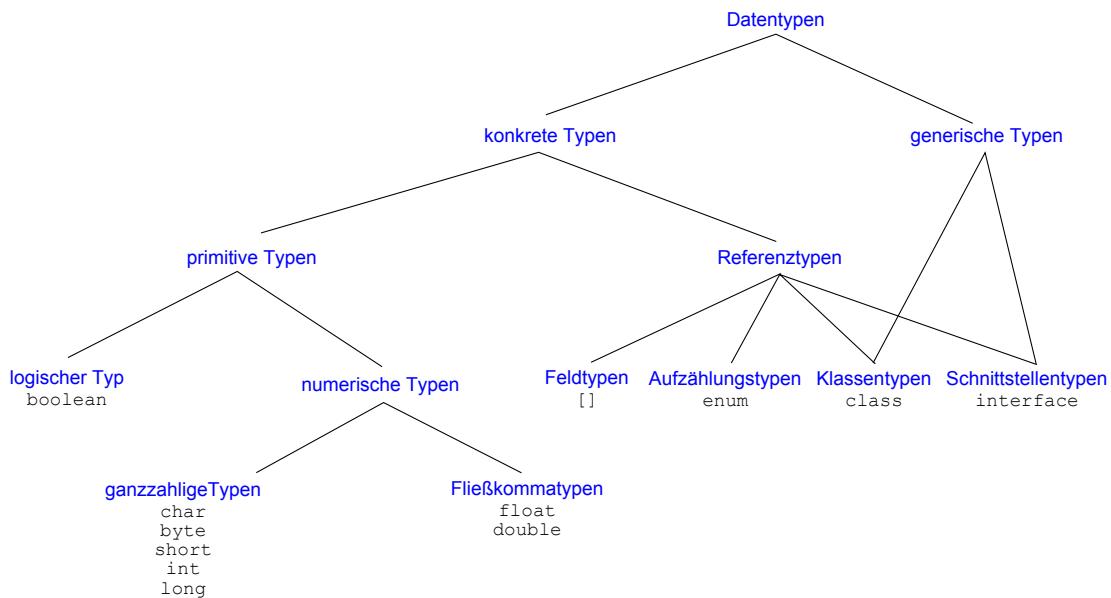


Figure 9.1: Struktur der Datentypen in Java

## 9.1 Felder

In der Mathematik und vielen Anwendungsbereichen spielen Vektoren und Matrizen eine sehr wichtige Rolle. Ein Beispiel ist etwa das Skalarprodukt  $skalarProdukt(x, y)$  zweier gleich großer Vektoren  $x, y \in \mathbb{R}^n$  der Länge  $n > 0$ , das wie folgt definiert ist  $skalarProdukt(x, y) = \sum_{i=1}^n x_i \cdot y_i$ .  $x$  und  $y$  sind die Vektoren,  $x_i$  ist ein Vektorelement in Form einer reellen Zahl, das durch den Index  $i$  innerhalb des Vektors ausgewählt wird. Ein weiteres Beispiel für eine Vektoroperation ist zu einem gegebenen Vektor  $x = (x_1, \dots, x_n)$  die Angabe eines Durchschnittsvektors  $y = (y_1, \dots, y_n)$  mit:

$$y_i = \begin{cases} x_i & \text{falls } i \in \{1, n\} \\ (x_{i-1} + x_i + x_{i+1})/3 & \text{falls } i \in \{2, \dots, n-1\} \end{cases}$$

Jedes Element des Ergebnisvektors bis auf die Randelemente ( $i = 1$  oder  $i = n$ ) ergeben sich jeweils aus der Durchschnittsbildung von drei benachbarten Elementen des Ausgangsvektors. Zu beachten ist hierbei die Trennung der allgemeinen Formel  $(x_{i-1} + x_i + x_{i+1})/3$  von der Wiederholungsvorschrift  $\forall i \in 2, \dots, n-1$  im zweiten Fall, worauf später im Fall von Java-Feldern wieder eingegangen wird.

In Programmiersprachen existieren diese Vektoren in einer ähnlichen Form als **Felder** (englisch: array), die Elemente eines Grundtyps  $T$  bis zu einer beim Erzeugen eines Feldes vorgegebenen maximalen Anzahl  $n$  aufnehmen können. Hat man ein Feld erzeugt, so kann man auf die einzelnen Feldelemente analog zum Beispiel des Skalarprodukts über einen Index relativ zum Feld in eckigen Klammern notiert zugreifen.

### Beispiel 9.1:

Das Beispiel zum Skalarprodukt sieht in Java wie in Listing 9.1 gegeben aus. Eine Diskussion der Details dazu erfolgt auf den nächsten Seiten. Das Beispiel zum Vektor durchschnitt sieht in Java wie in Listing 9.2 gegeben aus. ♦

Listing 9.1: Skalarprodukt.

```

9-1  /**
9-2   * Skalarprodukt als Beispiel zur Feldverwendung
9-3   */
9-4  public class Skalarprodukt {
9-5
9-6      public static void main(String[] args) {
9-7          // zwei Felder x und y anlegen
9-8          double[] x = new double[4];
9-9          double[] y = new double[4];
9-10         // unsere Ergebnisvariable
9-11         double skalarprodukt = 0.0;
9-12
9-13         // Skalarprodukt berechnen
9-14         for(int i=0; i<x.length; i++) {
9-15             skalarprodukt += x[i] * y[i];
9-16         }
9-17
9-18         System.out.println("Skalarprodukt der Vektoren ist " + skalarprodukt);
9-19     }
9-20 }
```

Listing 9.2: Vektordurchschnitt.

```

9-1  /**
9-2   * Durchschnitt von Vektorelementen
9-3   */
9-4  public class VektorDurchschnitt {
9-5
9-6      public static void main(String[] args) {
9-7          double[] x = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0};
9-8          double[] y = new double[6];
9-9
9-10         y[0] = x[0];
9-11         y[5] = x[5];
9-12         for(int i=1; i<x.length-1; i++) {
9-13             y[i] = (x[i-1] + x[i] + x[i+1]) / 3.0;
9-14         }
9-15     }
9-16 }
```

### 9.1.1 Eindimensionale Felder

Als Einstieg in die Thematik ist die Syntax für eine einfache Java-Felddeklaration in Abbildung 9.2 gegeben. Basierend auf einem Basistyp konstruiert man einen neuen Feldtyp dadurch, dass man zwei eckige Klammern `[]` an den Basistyp anhängt. Dies zusammen ist ein neuer Datentyp, der überall dort verwendet werden kann, wo einer der bisher bekannten Typen genutzt werden konnte, beispielsweise in der im Syntaxdiagramm angegebenen Variablen-deklaration. Der Basistyp kann zum Beispiel einer der uns bereits bekannten Typen wie `int` oder `double` sein, aber auch selber wieder ein Feldtyp. Hinter der Angabe des Feldtyps erscheint der Name der zu deklarierenden Feldvariablen, unter diesem Variablennamen ist ab diesem Punkt das Feld bekannt (später mehr Details dazu). Hinter der Angabe des Variablen-namens folgt ein Gleichheitszeichen und nachfolgend Initialisierungsausdruck zu einem Feld. In dem Ausdruck zwischen den beiden eckigen Klammern wird angegeben, wieviele Elemente das Feld besitzen soll.

#### Beispiel 9.2:

Im Beispielprogramm in Listing 9.3 werden zwei Felder `i_feld` und `d_feld` vom Basistyp `int` beziehungsweise `double` mit 10 beziehungsweise 100 Elementen deklariert. ♦

Nach einer Felddeklaration lassen sich die einzelnen Elemente eines Feldes direkt ansprechen durch den Feldnamen und einen Indexausdruck in eckigen Klammern, der angibt, das wievielte Element des Feldes gemeint ist (analog der Indexausdrücke zu Vektoren der Mathematik). Das erste Feldelement hat in Java den Index 0, das zweite Feldelement den Index 1 und so weiter. Das letzte Element hat den Index *laenge* – 1. Oft werden an dieser Stelle Fehler gemacht, dass zum Beispiel nach einer Deklaration der Form `int[] x = new int[4];` auf `x[4]` versucht zuzugreifen, was kein gültiger Index wäre.

Die Java-Syntax zur Angabe eines Feldelementes ist in Abbildung 9.3 gegeben. Jedes Feldelement kann wie eine eigene Variable aufgefasst werden. Zum Beispiel kann man Feldelementen einen Wert zuweisen (linke Seite einer Zuweisung) oder in einem Ausdruck verwenden (zum Beispiel auf der rechten Seite einer Zuweisung). Also beispielsweise ist erlaubt:

```
9-1  int[] a = new int[4]; // Feld mit 4 Elementen
9-2  a[0] = a[0] + 1;      // erhöhe den Wert des ersten Feldelements um 1
```

In Java wird zu jedem Feld intern auch die Länge des Feldes in Form der Anzahl der Datenelemente gespeichert, die über den Ausdruck `Feldvariable.length` erfragbar ist. Es ist ein guter Programmierstil, wo immer möglich anstatt einer festen Zahl für die Anzahl der datenelemente in einem Feld (zum Beispiel in einer for-Schleife) einen solchen Längenausdruck zu verwenden.

Alternativ zum Anlegen eines Feldes mit den `new`-Operator kann man auch ein Feld mit expliziten Werten direkt angeben. Die Größe des Feldes ergibt sich aus der Anzahl der angegebenen Werte. Dazu muss man

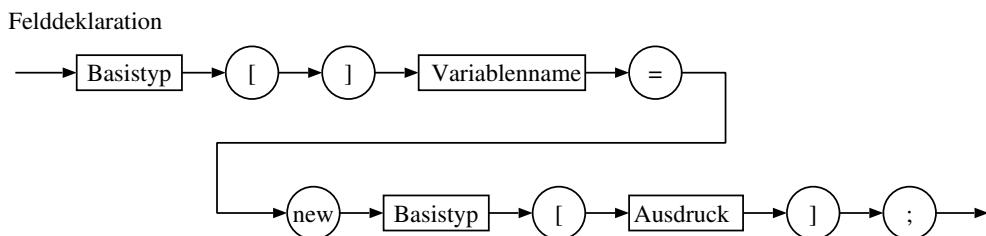


Figure 9.2: Syntaxdiagramm für eine einfache Felddeklaration

Listing 9.3: Einfaches Beispiel zu Felddeklarationen.

```

9.1  /**
9.2   * Testprogramm zur Felddeklaration
9.3   */
9.4  public class FeldTest {
9.5
9.6      public static void main(String [] args) {
9.7          int [] i_feld = new int [10];
9.8          double [] d_feld = new double [100];
9.9      }
9.10 }
```

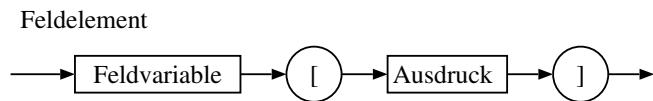


Figure 9.3: Syntaxdiagramm zur Indizierung eines Feldelementes

in einer Deklaration die Werte durch geschweifte Klammern eingeschlossen angeben. Als Hinweis sei hier angegeben, dass sich auch anonyme Felder anlegen lassen, wie im Beispiel in Listing 9.4 gezeigt wird.

Eine Aufgabe im Zusammenhang mit Feldern ist die Beantwortung der Frage, ob ein bestimmter Wert in einem Feld vorkommt oder nicht. Also beispielweise, ob in einem Feld, das die Feiertage von  $n$  Mitarbeitern enthält, der Wert 0 vorkommt. Als Ergebnis einer solchen Suche möchte man die erste Position im Feld haben, an der dieser Wert aufgetreten ist beziehungsweise eine Kennzeichnung im Ergebniswert, dass der gesuchte Wert nicht aufgetreten ist. Das Verfahren dazu kann so ablaufen, dass nach und nach jedes Feldelement mit dem gesuchten Wert verglichen wird und diese Suche abgebrochen wird, sobald der gesuchte Wert gefunden wurde. Das Verfahren nennt man aufgrund dieses Vorgehens auch **sequentielles Suchen** (siehe Listing 9.5). Dies wird später in Kapitel 15.2.1 nochmals aufgegriffen.

Die boolesche Variable `gefunden` dient dazu, nach Besuch aller Feldelemente zu signalisieren, ob in dem Feld der Suchwert gefunden wurde oder nicht. Findet man den Suchwert, so hat diese Variable den Wert `true`. Würde der Suchwert mehrfach im Feld auftreten, so würde man nur den ersten Wert erkennen. Das Verfahren lässt sich aber leicht erweitern, dass aller Vorkommen nach und nach erkannt werden können.

In Listing 9.6 ist das gleiche Verfahren als Methode realisiert. Hierbei muss bei der Überlegung zu dieser Methode entschieden werden, welche Information wie an einen Aufrufer übermittelt werden kann. Es kann erstens der Fall auftreten, dass der gesuchte Wert in dem Feld vorhanden ist. Dann ist man an der ersten Position interessiert, an der dieser Wert in dem Feld zu finden ist. Zweitens kann aber auch der Fall auftreten, dass das gesuchte Element nicht in dem Feld vorkommt. In diesem Fall gibt es auch keine relevante Position als Ergebnis. Eine Problematik ist, dass eine Methode immer nur genau einen Ergebniswert liefern kann. In der folgenden Methode werden die beiden möglichen Fälle dadurch gelöst, dass der Ergebniswert der Methode vom Typ `int` ist und die Position des gefundenen Elementes für den ersten Fall angibt. In diesem ersten Fall kann der Ergebniswert dann nur zwischen 0 und der Länge des Feldes minus 1 einschließlich liegen, andere Positionen für ein Finden des Wertes sind nicht möglich. Für den zweiten Fall, dass der Wert also nicht im Feld gefunden wurde, liefert die Methode einen Wert als Ergebnis, der kein gültiger Positionsindex sein kann, um diesen Fall dem Aufrufer zu signalisieren. Beispielweise ist die Zahl `-1` ein solcher Wert. Ein Aufrufer der Methode kann (und muss) anhand des Ergebniswertes damit erkennen, ob der gesuchte gefunden wurde ( $1 \leq x < feld.length$ ) oder nicht ( $x = -1$ ) und wenn er gefunden wurde, was die entsprechende Position im Feld ist.

Listing 9.4: Einfaches Beispiel zu Felddeklarationen über Initialisierung.

```

9-1 /**
9-2 * Felder anlegen ueber Angabe von Werten
9-3 */
9-4 public class FeldAnlegen {
9-5
9-6     public static void main(String[] args) {
9-7         // int-feld der Laenge 5
9-8         int[] a = {1,2,3,4,5};
9-9
9-10        // double-Feld der Laenge 3
9-11        double[] b = {1.0, 2.0, 3.0};
9-12
9-13        // int-Feld der Laenge 3 an die Methode uebergeben
9-14        methode(new int[]{1,2,3});
9-15    }
9-16
9-17    // Methode erwartet ein int-Feld als Argument
9-18    public static void methode(int[] a) {
9-19        return;
9-20    }
9-21}

```

Listing 9.5: Sequentielles Suchen in einem Feld.

```

9-1 /**
9-2 * Sequentielles Suchen in einem Feld
9-3 */
9-4 public class SequentiellesSuchen {
9-5
9-6     public static void main(String[] args) {
9-7         int x = 5;                      // zu suchender Wert
9-8         int[] a = new int[10];          // Feld mit Daten
9-9
9-10        // Belegung des Feldes mit Beispielwerten
9-11        for(int i=0; i<a.length; i++) {
9-12            a[i] = i;
9-13        }
9-14
9-15        // Suchen
9-16        boolean gefunden = false;
9-17        int position = -1;
9-18        for(int i=0; i<a.length; i++) {
9-19            if(a[i] == x) {
9-20                gefunden = true;
9-21                position = i;
9-22                break;
9-23            }
9-24        }
9-25
9-26        if(gefunden) {
9-27            System.out.println("gefunden an Position " + position);
9-28        } else {
9-29            System.out.println("nicht gefunden");
9-30        }
9-31    }
9-32}

```

Listing 9.6: Sequentielles Suchen in einem Feld als Methode.

```

9-1 /**
9-2 * Sequentielles Suchen in einem Feld als Methode
9-3 */
9-4 public class SequentiellesSuchen2 {
9-5
9-6     /** sequentielles Suchen in einem Feld
9-7     * @param a Feld mit Werten
9-8     * @param x gesuchter Wert
9-9     * @result Position (Index), an der Wert gefunden wurde. Ansonsten -1
9-10    */
9-11    public static int suchen(int[] a, int x) {
9-12        int position = -1;
9-13        // durchlaufe alle Feldelemente
9-14        for(int i=0; i<a.length; i++) {
9-15            if(a[i] == x) {
9-16                // Wert gefunden, wir koennen aufhoeren
9-17                position = i;
9-18                break;
9-19            }
9-20        }
9-21        return position;
9-22    }
9-23
9-24    public static void main(String[] args) {
9-25        int x = 5;                      // zu suchender Wert
9-26        int[] a = {1,2,3,4,5,6,7,8,9}; // Feld mit Daten
9-27
9-28        // suchen
9-29        int position = suchen(a, x);
9-30
9-31        if(position >= 0) {
9-32            System.out.println("gefunden an Position " + position);
9-33        } else {
9-34            System.out.println("nicht gefunden");
9-35        }
9-36    }
9-37}

```

Listing 9.7: Zugriff auf Feldindizes außerhalb des erlaubten Bereichs.

```

9-1 public class Indexverletzung {
9-2     public static void main(String[] args) {
9-3         int [] feld = new int [2];
9-4         feld[2] = 4;
9-5     }
9-6 }
```

Nutzt man beim Feldzugriff einen Indexwert, der außerhalb der Feldgrenzen liegt, so tritt bei der Ausführung des Programms ein Laufzeitfehler auf, den das Java-Laufzeitsystem immer erkennt und das Programm abbricht.

### Beispiel 9.3:

Startet man das Programm aus Listing 9.7 , so bekommt man als Fehlermeldung:

```

9-1 prompt> java Indexverletzung
9-2 Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
9-3         at Indexverletzung.main(Indexverletzung.java:6)
```

Das Laufzeitsystem von Java, das die Ausführung des Programms im gewissen Sinne überwacht, stellt fest, dass ein Zugriff auf ein nicht-vorhandenes Feldelement stattfinden sollte und bricht das gesamt Programm daraufhin ab. ♦

Im Beispiel des sequentiellen Suchens sieht man auch den bereits vorher erwähnten Zusammenhang zwischen Zählschleifen und Feldern. Zum **Durchlaufen** eines Feldes nutzt man eine Zählschleife, die die Indizes "erzeugt", um auf alle Feldelemente zuzugreifen. Ebenso wie es bei Binäräbäumen eine Durchlaufstrategie zum Besuch aller Knoten des Binärbaumes gab (preorder, inorder, postorder), gibt es auch hier eine Durchlaufstrategie zum Besuch aller Feldelemente, das sequentielle Durchlaufen des Feldes. In Durchlaufstrategien spiegelt sich also auch in gewisser Weise die **Struktur** von Datentypen wider. Das Thema wird in Kapitel 16.2 nochmals aufgegriffen.

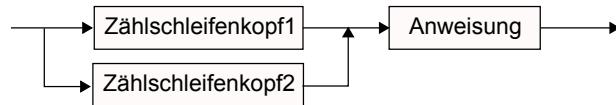
Den engen Zusammenhang zwischen Feldern und Zählschleifen speziell in dem zuletzt angesprochenen Zusammenhang sieht man auch daran, dass die Syntax für for-Schleifen auch in erweiterter Form möglich ist (siehe Abbildung 9.4). Dazu wird ein Feld angegeben in Form eines Feldausdrucks (Beispiel: der Name einer Feldvariablen) und eine Variable zum Basistyp des Feldes. Die Bedeutung dieses Konstrukts ist es, dass die Variable des Basistyps, die Zählvariable, alle Werte des Feldes annehmen wird und damit der Schleifenrumpf dann jeweils ausgeführt wird.

### Beispiel 9.4:

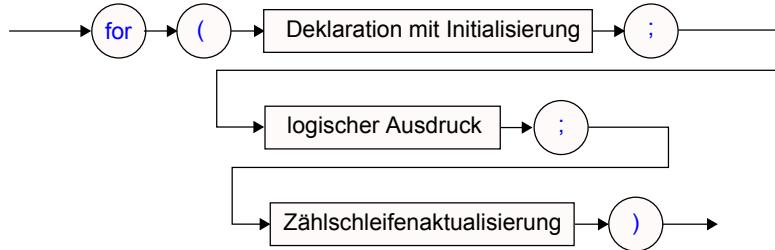
Das Programm in Listing 9.8 gibt unter Nutzung der erweiterten Syntax alle Elemente eines Feldes auf dem Bildschirm aus.



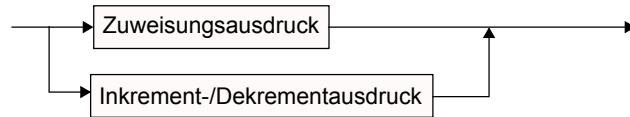
### Zählschleife



### Zählschleifenkopf1



### Zählschleifenaktualisierung



### Zählschleifenkopf2



Figure 9.4: Erweitertes Syntaxdiagramm für Zählschleifen

Listing 9.8: Beispiel für die erweiterte Schleifensyntax.

```

9-1 /**
9-2 * for als Aufzaehlung
9-3 */
9-4 public class Foreach {
9-5
9-6     public static void main(String[] args) {
9-7         int[] a = new int[10];
9-8
9-9         // iteriere ueber alle Feldelemente von a
9-10        // x nimmt hierbei alle Werte von x[0,...,x.length-1] ein
9-11        for(int x : a) {
9-12            // gebe das i-te Feldelement aus
9-13            System.out.println(x);
9-14        }
9-15    }
9-16 }
```

### 9.1.2 Mehrdimensionale Felder

Mit der eben eingeführten Syntax ist es möglich, basierend auf einem Basistyp einen neuen Feldtyp zu konstruieren, der wiederum selber ein Typ ist. Nichts hindert uns also daran, diesen neuen Feldtyp als Basistyp einen wiederum neuen Feldtyp anzugeben, wie etwa `double[][] a` oder mit expliziter Klammerung, um dies deutlich zu machen: `(double[][]) a`. Die Elemente eines solchen zweidimensionalen Feldes sind in Java dann jeweils selber eindimensionale Felder des ursprünglichen Datentyps, mit allen Konsequenzen, die sich daraus ergeben (siehe auch Kapitel 9.2).

#### Beispiel 9.5:

Eine Matrix  $a \in \mathbb{R}^{n \times m}$  ist eine zweidimensionale Matrix mit  $n$  Zeilen und  $m$  Spalten. Eine Beispiel mit der entsprechenden Notation für eine Matrix aus  $\mathbb{R}^{2 \times 3}$  ist:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

In Java würde eine entsprechende Deklaration für  $n = 2, m = 3$  aussehen:

```
9-1 int n = 2, m = 3;
9-2 double[][] a = new double[n][m];
9-3 a[1][2] = 4711.0;
```

In der letzten Programmzeile wird dem Matrixelement  $a_{1,2}$  ein Wert zugewiesen. Zu beachten ist, dass die Indizierung in jeder Dimension bei 0 beginnt und dass die Indexwerte der jeweiligen Dimensionen in eckigen Klammern jeweils hintereinander (in der Reihenfolge der Dimensionen) geschrieben werden. ♦

Ebenso wie im eindimensionalen Fall ist auch ein explizites Anlegen einer zwei- oder mehrdimensionalen Matrix möglich. Hierbei sein nochmals darauf hingewiesen, dass eine  $n$ -dimensionales Feld in Java ein Feld von  $(n-1)$ -dimensionalen Feldern ist. Die durch Komma getrennten Werte sind also selber wiederum Feldinitialisierungsausdrücke.

#### Beispiel 9.6:

Nachfolgend wird die obige Matrix mit zwei Zeilen und drei Spalten in Java definiert als Feld mit zwei Elementen, die jeweils Felder mit drei Elementen sind.

```
9-1 int [][] a = { {1,2,3}, {4,5,6} };
9-2 // aequivalent und besser lesbar
9-3 int [][] b = { {1,2,3},
9-4                 {4,5,6} };
```

Ebenso wie im eindimensionalen Fall kann man auch die Feldlänge in jeder Dimension eines mehrdimensionalen Feldes erfragen. Auch hier hilft die Rückführung eines  $n$ -dimensionalen Feldes auf ein Feld von  $(n-1)$ -dimensionalen Feldern.

#### Beispiel 9.7:

Im Beispiel in Listing 9.9 wird ein zweidimensionales Feld angelegt und in einer Methode der Inhalt des übergebenen Feldes auf dem Bildschirm ausgegeben.

Listing 9.9: Bildschirmausgabe einer Matrix.

```

9.1  /**
9.2   * Ausgabe eines zweidimensionalem Feldes
9.3   */
9.4  public class MatrixAusgabe {
9.5
9.6      public static void main(String[] args) {
9.7          int [][] a = { {1,2,3}, {4,5,6} };
9.8          zeigeMatrix(a);
9.9      }
9.10
9.11     // zeige den Inhalt einer Matrix auf dem Bildschirm
9.12     public static void zeigeMatrix(int [][] a) {
9.13         // iteriere ueber Zeilen
9.14         for(int i=0; i<a.length; i++) {
9.15             // iteriere ueber Spalten
9.16             for(int j=0; j<a[i].length; j++) {
9.17                 // Ausgabe Element i,j
9.18                 System.out.print(" " + a[i][j]);
9.19             }
9.20             // Ausabezeile nach einer Matrixzeile beenden
9.21             System.out.println();
9.22         }
9.23     }
9.24 }
```

Zu beachten ist, dass für das zweidimensionale Feld `a` der Wert des Ausdrucks `a[i]` im Schleifenkontrollausdruck ein eindimensionales Feld ist, das damit auch eine Länge hat. ♦

Matrizen spielen eine wichtige Rolle in vielen Bereichen. Dabei basieren oft viele anwendungsbezogene Operationen mit Matrizen auf einfachen Basisoperationen, die auf Matrizen definiert sind (tiefergehende Details auch zur numerischen Umsetzung etwa in [GL96]). Dazu zählen die Matrixaddition zweier gleichgroßer Matrizen, die Matrixmultiplikation oder auch die Multiplikation einer Matrix mit einem Vektor.

### Beispiel 9.8:

Im Beispiel in Listing 9.10 addiert die Methode `addieren` zwei beliebige gleichgroße Matrizen und liefert als Ergebnis die Ergebnismatrix. ♦

Auf einige tiefergehende Aspekte wurde hier nicht eingegangen. Zum Beispiel, dass bei einem zweidimensionalen Feld `double[][] a` in Java, das ja bekanntermaßen ein Feld von Felder ist, die eindimensionalen Felder `a[i]` eine unterschiedliche Länge haben können und auch einzeln angelegt werden können.

Listing 9.10: Addition zweier Matrizen.

```

9-1  /**
9-2   * Matrixaddition
9-3   */
9-4  public class MatrixAddition {
9-5
9-6      // addiere zwei gleich grosse Matrizen
9-7      public static double[][] addieren(double[][] a, double[][] b) {
9-8          // Achtung: wir ueberpruefen hier nicht,
9-9          // ob beide Matrizen gleich gross sind!
9-10         // Ergebnismatrix erzeugen
9-11         double[][] c = new double[a.length][a[0].length];
9-12
9-13         // iteriere ueber Zeilen
9-14         for(int i=0; i<a.length; i++) {
9-15             // iteriere ueber Spalten
9-16             for(int j=0; j<a[i].length; j++) {
9-17                 c[i][j] = a[i][j] + b[i][j];
9-18             }
9-19         }
9-20
9-21         // gebe Matrix c als Ergebnis zurueck
9-22         return c;
9-23     }
9-24
9-25     // zeige den Inhalt einer Matrix auf dem Bildschirm
9-26     public static void zeigeMatrix(double[][] a) {
9-27         for(int i=0; i<a.length; i++) {
9-28             for(int j=0; j<a[i].length; j++) {
9-29                 System.out.print(" " + a[i][j]);
9-30             }
9-31             System.out.println();
9-32         }
9-33     }
9-34
9-35     public static void main(String[] args) {
9-36         double[][] a = {{1,2,3}, {4,5,6}, {7,8,9}};
9-37         double[][] b = {{9,8,7}, {6,5,4}, {3,2,1}};
9-38
9-39         double[][] c = addieren(a,b);
9-40         zeigeMatrix(c);
9-41     }
9-42 }
```

## 9.2 Referenzen

Bei der Betrachtung von Feldern wurde bisher ein sehr wesentlicher Aspekt nicht diskutiert, dass nämlich Feldtypen zur Kategorie Referenztypen gehören. In Java besteht nämlich ein wesentlicher Unterschied zwischen den vor diesem Kapitel eingeführten primitiven Datentypen `boolean`, `byte`, `short`, `int`, `long`, `char`, `float` und `double` (im Java-Sprachstandard werden diese auch *einfache Typen* genannt) und allen anderen Datentypen, zu denen auch die eben besprochenen Feldtypen gehören. Eine Variable eines primitiven Typs kann genau einen Wert des entsprechenden Typs aufnehmen. Alle nicht-primitiven Typen in Java, so auch Feldtypen, werden als **Referenztypen** bezeichnet. Eine Variable eines Referenztyps nimmt jetzt nicht mehr einen Wert des entsprechenden Datentyps *direkt* auf, also beispielsweise ein Feld von `int` für einen Datentyp `int[]`, sondern lediglich eine Referenz (Verweis, Zeiger) auf ein solches Feld. Eine Referenz ist eine Hauptspeicheradresse, unter der dann beispielsweise der eigentliche Feldinhalt zu finden ist. Eine Feldvariable ist also damit auch nur ein indirekter Verweis auf das eigentliche Feld. Bei der Nutzung einer Feldvariablen zum Beispiel in einem Ausdruck wird dieser Sachverhalt transparent verdeckt. Bei einem Ausdruck `a[i]` ist zum Beispiel für den Programmierer zuerst einmal nicht erkennbar, dass die Feldvariable `a` nicht das Feld selber ist, sondern lediglich nur eine Referenz auf ein Feld ist. Dieser kleine Detailunterschied zieht beträchtliche Konsequenzen nach sich, die in diesem Kapitel grundsätzlich diskutiert werden. Diese für einen Anfänger zu Beginn nicht ganz einfachen Sachverhalte sollen in diesem Kapitel eingehend behandelt werden.

### 9.2.1 Deklaration von Refenzvariablen

In der Deklaration einer Feldvariablen konnte man schon einen Unterschied zu der gewohnten Deklaration mit einem primitiven Typ, etwa einer `int`-Variablen sehen:

9-1	<code>int i = 5;</code>	<i>// i ist eine Variable vom Typ int</i>
9-2	<code>int [] feld = new int [4];</code>	<i>// feld ist eine Variable vom Typ Feld von int</i>

Die Bedeutung der ersten Zeile dieses Programmausschnitts wurde bereits vorher erläutert. Darin wird eine Variable mit Namen `i` neu angelegt. Der Ausdruck auf der rechten Seite des Zuweisungsausdrucks wird ausgewertet, das Ergebnis ist die Zahl 5 (oder genauer die Zweierkomplementdarstellung der Zahl 5 mit 32 Bits), die in der Variablen `i` dann gespeichert wird.

In der zweiten Zeile mit der Deklaration der Feldvariablen `feld` passieren ebenfalls zwei Dinge, die man auch getrennt voneinander angeben könnte (allerdings zwingend in der obigen Reihenfolge):

1. Mit `int [] feld` wird eine Variable `feld` deklariert, die eine **Referenz** auf ein Feld aufnehmen kann. Mit dieser Deklaration ist *kein* Feld angelegt worden, sondern nur eine vergleichsweise kleine Referenzvariable, die genau eine Hauptspeicheradresse, eine Referenz als Wert aufnehmen kann.
2. Mit `new int [4]` auf der rechten Seite des Zuweisungsausdrucks wird ein (anonymes) Feld der angegebenen Größe angelegt. Dieser Ausdruck liefert bei Auswertung als Ergebniswert eine Referenz auf ein neu geschaffenes namenloses Feld der entsprechenden Größe. Über den `new`-Operator wird intern auf dem Heap nach einem zusammenhängenden freien Speicherbereich gesucht, der hinreichend groß für das neu anzulegende Feld ist. Die Größe bestimmt sich aus der Größe des Basistyps (im Beispiel 4 Byte für `int`) und der Anzahl von Elementen (im Beispiel 4), also im Beispiel insgesamt  $4 \cdot 4 = 16$  Byte sowie zusätzlich einige Bytes als Verwaltungsoverhead. Der freie Speicherbereich wird mit Nullen initialisiert. Der `new`-Operator liefert als Ergebnis die Startadresse im Hauptspeicher des neu angelegten Speicherbereichs. Ist im Heap kein freier Speicherbereich der angeforderten Größe mehr vorhanden, tritt ein Laufzeitfehler auf und das Programm bricht ab.
3. Durch das Gleichheitszeichen, also eine Zuweisung, wird im Beispiel der Wert des `new`-Ausdrucks, also die Referenz auf das neu geschaffene Feld, in der Referenzvariablen abgelegt.

Listing 9.11: Anlegen eines sehr großen Feldes.

```

9-1 public class VielSpeicher {
9-2     public static void main(String[] args) {
9-3         int [] feld = new int [1000000000];
9-4     }
9-5 }
```

Speicheradresse	Inhalt	Variablenname	Speicheradresse	Inhalt	Variablenname
3836	?	i	3836	5	i
3840	?	feld	3840	4712	feld

...	...
4712	0
4716	0
4720	0
4724	0

(a) Anlegen der Referenzvariablen

(b) Anlegen des Feldes

Figure 9.5: Referenzen

**Beispiel 9.9:**

Das Beispielprogramm in Listing 9.11 versucht ein sehr großes Feld anzulegen. Das Programm erzeugt nach dem Start als Ausgabe:

```

9-1 prompt> java VielSpeicher
9-2 Exception in thread main '' java.lang.OutOfMemoryError: Java heap space
9-3         at VielSpeicher.main(VielSpeicher.java:3)
```

das heißt, das Java Laufzeitsystem konnte nicht soviel Heap-Speicher besorgen, wie nötig gewesen wäre. ♦

Abbildung 9.5 zeigt nochmals schrittweise den Unterschied zwischen dem Anlegen einer Referenzvariablen und einer Variablen eines einfachen Typs sowie der Zuweisung eines Wertes an diese jeweiligen Variablen. Nach dem ersten Schritt, der Deklaration der beiden Variablen (Abbildung 9.5a; linker Teil des gesamten Zuweisungsausdrucks), existieren die beiden Variablen und haben zuerst einmal einen undefinierten Wert. Im zweiten Schritt (Abbildung 9.5b; rechter Teil des Zuweisungsausdrucks) bekommt die int-Variablen den skalaren Wert 5 zugewiesen. Bei der Feldvariablen wird über den new-Operator Speicher für ein neues, mit Nullen initialisiertes Feld von int angelegt. Das Resultat des new-Ausdrucks ist eine Referenz (die Hauptspeicheradresse) auf dieses anonyme Feld und genau diese Referenz wird in der Referenzvariablen abgelegt. In Abbildung 9.6 ist eine schematische Darstellung angegeben, wo die konkreten Speicheradressen weggelassen sind und lediglich die Referenz als Pfeil gezeigt ist, was anschaulicher ist als konkrete Hauptspeicheradressen.

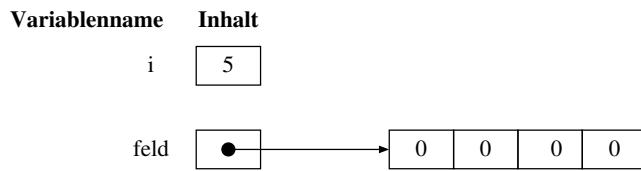


Figure 9.6: Schematische Darstellung einer Referenz

### 9.2.2 Operationen mit Referenzen

Referenzvariablen enthalten Referenzen in Form von Hauptspeicheradressen. Kann man mit diesen Werten arbeiten, beispielsweise rechnen wie dies mit int-Werten möglich ist? Was ist möglich an Operationen und was passiert dabei? In Java, anders als etwa in anderen Programmiersprachen wie C/C++, sind nur wenige Operationen mit Referenzen möglich. Dazu zählt die Zuweisung eines Referenzwertes an eine Variable eines kompatiblen Typs<sup>1</sup> sowie der Vergleich zweier typkompatibler Referenzwerte.

Begonnen wird mit der Zuweisung von Referenzwerten. Es ist möglich, einer Referenzvariablen den Wert eines typkompatiblen Referenzausdrucks etwa in Form einer Referenzvariablen gleichen oder kompatiblen Typs zuzuweisen. Die Bedeutung dieser Operation ist, dass der Referenzwert / die Hauptspeicheradresse des Referenzausdrucks einfach kopiert wird und in der Referenzvariablen abgelegt wird. Und hier kommt jetzt der entscheidende und überaus wichtige Punkt: anschließend verweisen nämlich *zwei Referenzvariablen auf das gleiche Referenzobjekt!*

#### Beispiel 9.10:

Im Programm in Listing 9.12 werden drei Referenzvariablen `feld1`, `feld2`, `feld3` angelegt. Die Variablen `feld1` und `feld2` erhalten jeweils eine Referenz auf neu geschaffene Felder zugewiesen. `feld3` bekommt bei der Deklaration keinen Wert zugewiesen und hat damit zuerst einmal einen nicht definierten Wert.

Nach dem Deklarationsteil werden den beiden Feldelementen `feld1[0]` und `feld2[0]` Werte zugewiesen. Die nachfolgende Zeile ist nun aber interessant. Was bewirkt die Zuweisung `feld3 = feld1`?

Führt man nun das Programm aus, so ist die Ausgabe:

```
9.1 feld1[0]=4711, feld2[0]=31415, feld3[0]=4711
9.2 feld1[0]=4711, feld2[0]=31415, feld3[0]=31415
```

Wie man sieht, entspricht der Wert in `feld3[0]` dem Wert von `feld1[0]`. Das könnte zwei Ursachen haben: das Feld, auf das `feld1` verweist, wurde kopiert und das so neu geschaffene kopierte Feld `feld3` zugewiesen. Dies ist aber nicht der Fall! Nicht das Feld wurde kopiert, sondern vielmehr wurde lediglich die Referenz auf das Feld, also der direkte Inhalt der Variablen `feld1` kopiert, womit nun zwei Referenzvariablen `feld1` und `feld3` auf das gleiche anonyme Feld verweisen. Analog gilt das für die zweite Zuweisung `feld3 = feld2`. Abbildung 9.7 zeigt schematisch die Situation jeweils vor den beiden Ausgabeanweisungen im Programm. ♦

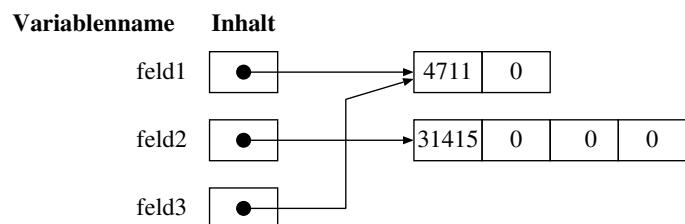
Während in einigen Fällen sicherlich die Zuweisung in der erläuterten Bedeutung auch so gewünscht ist (Referenz wird kopiert, aber nicht das referenzierte Objekt), gibt es sicherlich aber auch andere Fälle, in denen ein Programmierer nicht nur die Referenz kopieren und zuweisen will, sondern wirklich das komplette Objekt kopieren und zuweisen möchte. Für diese Fälle gibt es eine Methode `clone`, die das Gewünschte leistet und auf jedes Referenzobjekt angewandt werden kann. Die Methode kann nur im Zusammenhang mit einem

<sup>1</sup>An dieser Stelle wird nicht weiter auf Typkompatibilität bei Referenztypen eingegangen und im Folgenden bei Operationen auf Referenzen stets auch stillschweigend von einer Typkompatibilität ausgegangen.

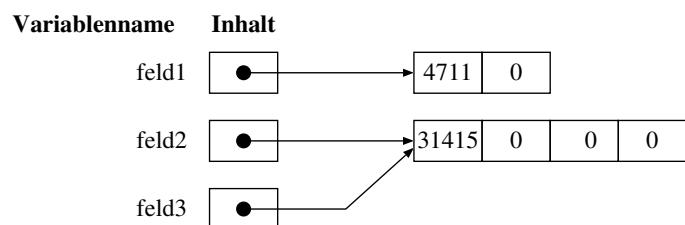
Listing 9.12: Zuweisung von Referenzen.

```

9-1  /**
9-2   * Zuweisung von Referenzvariablen
9-3   */
9-4  public class Referenzzuweisung {
9-5      public static void main(String [] args) {
9-6          int [] feld1 = new int [2];
9-7          int [] feld2 = new int [4];
9-8          int [] feld3;
9-9
9-10
9-11         feld1[0] = 4711;
9-12         feld2[0] = 31415;
9-13         feld3 = feld1;
9-14         System.out.println("feld1[0]="+ feld1[0] + " , feld2[0]="+ feld2[0]
9-15                     + " , feld3[0]="+ feld3[0]);
9-16         feld3 = feld2;
9-17         System.out.println("feld1[0]="+ feld1[0] + " , feld2[0]="+ feld2[0]
9-18                     + " , feld3[0]="+ feld3[0]);
9-19     }
9-20 }
```



(a) Zustand vor der ersten Ausgabeanweisung



(b) Zustand nach der zweiten Ausgabeanweisung

Figure 9.7: Referenzzuweisung

Listing 9.13: Kopieren / Clonen eines Feldes.

```

9.1  /**
9.2   * Clonen eines Feldes
9.3   */
9.4  public class FeldClone {
9.5
9.6      public static void main(String[] args) {
9.7          // Referenzvariable, die auf das Feld verweist
9.8          int[] a = {1,2,3};
9.9          // Referenzvariable, die auf nichts verweist
9.10         int[] b;
9.11
9.12         // Version 1: kopieren der Referenz
9.13         b = a;
9.14         // hiermit aendert sich auch a[0],
9.15         // weil a und b auf das gleiche Feld verweisen
9.16         b[0] = 4;
9.17         System.out.println("a[0] = " + a[0] + ", b[0] = " + b[0]);
9.18
9.19         // Version 2: Clonen des Feldes
9.20         b = a.clone();
9.21         // hiermit aendert sich nicht a[0],
9.22         // weil a und b auf verschiedene Felder verweisen
9.23         b[0] = 5;
9.24         System.out.println("a[0] = " + a[0] + ", b[0] = " + b[0]);
9.25     }
9.26 }
```

referenzierten Objekt genutzt werden, was in Kapitel 11 erst im Detail eingeführt wird, hier aufgrund des Zusammenhangs aber schon erwähnt werden soll.

### Beispiel 9.11:

Im Programm in Listing 9.13 wird in Version 1 die Referenz in `a` kopiert und `b` zugewiesen. Nach `b[0]=4;` ist bei einem späteren Zugriff auf `a[0]` dort ebenfalls ein veränderter Wert zu sehen, weil beide Variablen das gleiche Feld referenzieren.

In der zweiten Version wird das Feld geclont, also eine inhaltsgleiche Kopie des Feldes erzeugt und `b` zugewiesen. Wenn dann `b[0]` verändert wird, hat das keine Auswirkungen auf `a[0]`.

Die Ausgabe ist:

```

9.1 a[0]=4, b[0]=4
9.2 a[0]=4, b[0]=5
```

Neben der Zuweisung ist die zweite mögliche Operation mit Referenzwerten der Vergleich zweier Referenzwerte. Für zwei Referenzwerte `ref1` und `ref2` können diesen beiden Werte verglichen werden mit `ref1 == ref2`. Die Bedeutung ist hierbei, dass die beiden Referenzen (Hauptspeicheradressen) verglichen werden (`true` für gleich und `false` für ungleich ist das Ergebnis) und *nicht* die Inhalte der Objekte, auf die diese Referenzen verweisen.

Listing 9.14: Vergleich von Referenzen.

```

9.1 /**
9.2 * Vergleich mit Referenzen
9.3 */
9.4 public class Referenzvergleich {
9.5
9.6     public static void main(String[] args) {
9.7         int[] feld1 = {1,2,3};
9.8         int[] feld2 = {1,2,3};
9.9
9.10        // Vergleich 1
9.11        System.out.println("Vergleich 1: " + (feld1 == feld2));
9.12
9.13        // Referenzzuweisung
9.14        feld1 = feld2;
9.15        // Vergleich 2
9.16        System.out.println("Vergleich 2: " + (feld1 == feld2));
9.17    }
9.18 }
```

**Beispiel 9.12:**

Im Programm in Listing 9.14 werden zwei Felder gleichen Inhalts angelegt und die Referenzen darauf jeweils zwei Referenzvariablen zugewiesen. Die Ausgabe des Programms ist:

```

9.1 Vergleich 1: false
9.2 Vergleich 2: true
```

Beim ersten Vergleich werden die beiden Referenzen verglichen, die auf die beiden Felder verweisen. Auch wenn der Inhalt der Felder, also die int-Werte, alle gleich sind, so sind es doch zwei verschiedene Felder. Nach der Zuweisung `feld1 = feld2;` verweisen beide Referenzvariablen auf das gleiche Feld. Damit werden im zweiten Vergleich auch zwei identische Referenzen (gleiche Hauptspeicheradressen) verglichen, weshalb das Ergebnis des Vergleich wahr ist. ♦

Es gibt einen speziellen Referenzwert `null`, der angibt, dass kein Verweis auf ein Objekt vorliegt. Weist man einer Referenzvariablen diesen null-Wert zu, so kann man natürlich über die Referenzvariable auch nicht mehr auf zum Beispiel Feldelemente zugreifen. Einen Referenzwert kann man weiterhin mit `null` vergleichen.

**Beispiel 9.13:**

Die in Listing 9.15 gezeigte Änderung am ursprünglichen Programm in Listing 9.12 bewirkt, dass nach der Zuweisung `feld3 = null;` ein Zugriff über die Referenzvariable `feld3` zu einem Laufzeitfehler führt und das Programm beendet wird. Die Ausgabe des Programms ist:

```

9.1 feld1[0]=4711, feld2[0]=31415, feld3[0]=4711
9.2 feld1[0]=4711, feld2[0]=31415, feld3[0]=31415
9.3 Exception in thread "main" java.lang.NullPointerException
9.4         at Referenzzuweisung2.main(Referenzzuweisung2.java, Compiled Code)
```

Listing 9.15: Zuweisung von Referenzen (Beispiel 2).

```

9.1  /**
9.2   * Zuweisung Referenzvariablen (Version 2)
9.3   */
9.4  public class Referenzzuweisung2 {
9.5      public static void main(String[] args) {
9.6          int [] feld1 = new int [2];
9.7          int [] feld2 = new int [4];
9.8          int [] feld3;
9.9
9.10         feld1[0] = 4711;
9.11         feld2[0] = 31415;
9.12         feld3 = feld1;
9.13         System.out.println("feld1[0]="+ feld1[0] + ", feld2[0]="+ feld2[0]
9.14             + ", feld3[0]="+ feld3[0]);
9.15         feld3 = feld2;
9.16         System.out.println("feld1[0]="+ feld1[0] + ", feld2[0]="+ feld2[0]
9.17             + ", feld3[0]="+ feld3[0]);
9.18         feld3 = null;
9.19         System.out.println("feld1[0]="+ feld1[0] + ", feld2[0]="+ feld2[0]
9.20             + ", feld3[0]="+ feld3[0]);
9.21     }
9.22 }
9.23

```

**Beispiel 9.14:**

Der nachfolgende Programmausschnitt zeigt die Möglichkeit, wie man **null** auch in Bedingungen nutzen kann.

```

9.1  int [] feld1 = {1,2,3};
9.2  int [] feld2 = null;
9.3
9.4  // Test auf leere Referenz
9.5  if((feld1 == null) || (feld2 == null)) {
9.6      ...
9.7  }

```

Eine weitere interessante Frage taucht mit Referenzen auf, wenn auf ein Objekt keine Referenz mehr existiert. Das Programm in Listing 9.16 zeigt diesen Fall. Nach der Zuweisung **feld1 = feld2;** zeigen beide

Listing 9.16: Zuweisung von Referenzen (Beispiel 3).

```

9.1  /**
9.2   * Zuweisung Referenzvariablen (Version 3)
9.3   */
9.4  public class Referenzzuweisung3 {
9.5      public static void main(String[] args) {
9.6          int [] feld1 = {1,2};
9.7          int [] feld2 = {3,4};
9.8
9.9          feld1 = feld2;
9.10         // was ist jetzt mit dem Feld {1,2}?
9.11     }
9.12 }

```

Referenzvariablen auf das Feld mit den Werten 3 und 4. Aber was ist mit dem Feld passiert, das die beiden Werte 1 und 2 enthält? Es gibt keine Referenz mehr darauf. Dies ist eine Situation, die durchaus häufig in Java-Programmen vorkommen kann. Eine solche Situation bedeutet auch, dass im Hauptspeicher ein (oder mehrere) Objekt liegt, das nicht mehr referenziert werden kann und somit auch nicht mehr aus dem Programm heraus erreichbar ist. Würde man untätig bleiben und solche Fälle auch mit großen Objekten vermehrt auftreten, wäre irgendwann auf dem Heap kein freier Platz mehr für neue Objekte vorhanden, weil nur noch "Leichen" auf dem Heap liegen. Für diese Fälle wird im Java Laufzeitsystem ein sogenannter *garbage collector* aktiviert, der alle Objekte, auf die keine direkten oder indirekten Referenzen aus den Programmvariablen existieren, aus dem Heap entfernt und den damit verbundenen Heap-Speicher wieder als unbelegt markiert. Anders als zum Beispiel in C++ muss sich der Programmierer nicht um das Aufräumen des Speichers kümmern, der *garbage collector* besorgt dies automatisch.

### 9.2.3 Referenzen bei Methodenaufrufen

Nach Einführung der Referenzen soll nun die Methode `swap` aus Kapitel 8.5 nochmals betrachtet werden. Zur Erinnerung: es war nicht möglich gewesen, den Inhalt zweier Variablen eines primitiven Typs über einen Methodenaufruf zu tauschen, weil die Übergabestrategie call-by-value innerhalb der Methode Kopien der Werte tauschte, aber nicht die Originalwerte.

Hier soll nun zusätzlich eine Variante der swap-Methode betrachtet werden, bei der die beiden zu vertauschenden Werte in einem Feld abgelegt sind und dieses Feld (oder besser gesagt eine Referenz darauf) als Parameter übergeben wird (Listing 9.17). Die Ausgabe des Programms ist:

```

9.1 vorher: a=2, b=3
9.2 in swap: a=3, b=2
9.3 nachher: a=2, b=3
9.4 in swap: a=3, b=2
9.5 mit Feld: a=3, b=2

```

Wie man also sieht, liefert die zweite auf einem Feld basierte Methodenversion das gewünschte Resultat, nämlich das Vertauschen der Werte. Um diesen Unterschied zu verstehen, muss man sich die Details der Parameterübergabe nochmals anschauen. In beiden Fällen wird beim Methodenaufruf der Wert des Ausdrucks ausgewertet und auf dem Laufzeitstack abgelegt. Im ersten Fall sind das *Kopien* der Werte 2 und 3, im zweiten Fall ist dies eine *Kopie* der Referenz auf das Feld, in beiden Fällen also Wertkopien (call-by-value).

Innerhalb der swap-Methode mit zwei int-Argumenten werden diese Kopien in `x` und `y` vertauscht, aber nicht die Originalwerte in `a` und `b` von `main`, weil kein Zugang zu den Originalwerten innerhalb der swap-Methode mehr vorhanden ist.

In der zweiten swap-Methode mit einer Referenz hat man auch nur eine Kopie der Referenz übergeben bekommen. Wenn jetzt aber die Tauschaktion von `x[0]` und `x[1]` durchgeführt wird, so werden damit die Originalwerte des Feldes vertauscht, auf die man auch über die Referenzkopie Zugriff hat.

Dies hat schwerwiegende Konsequenzen. Denn alle Objekte, die per Referenz an eine Methode übergeben werden, können innerhalb der aufgerufenen Methode verändert werden.

Listing 9.17: Wiederbetrachtung von Übergabestrategien.

```

9.1 /**
9.2  * Test zu Uebergabestrategien
9.3 */
9.4 public class Uebergabestrategien {
9.5
9.6     public static void main(String[] args) {
9.7         // Uebergabe von Werten eines primitiven Typs
9.8         int a=2, b=3;
9.9         System.out.println("vorher: a=" + a + ", b=" + b);
9.10        swap(a,b);
9.11        System.out.println("nachher: a=" + a + ", b=" + b);
9.12
9.13         // Uebergabe eines Referenzwertes
9.14         int[] a2 = {2,3};
9.15         swap(a2);
9.16         System.out.println("mit Feld: a=" + a2[0] + ", b=" + a2[1]);
9.17     }
9.18
9.19     // Methode mit Parametern primitiven Typs
9.20     public static void swap(int x, int y) {
9.21         int tmp = x;
9.22         x = y;
9.23         y = tmp;
9.24         System.out.println("in swap: a=" + x + ", b=" + y);
9.25     }
9.26
9.27     // Methode mit Parameter eines Referenztyps
9.28     public static void swap(int[] x) {
9.29         int tmp = x[0];
9.30         x[0] = x[1];
9.31         x[1] = tmp;
9.32         System.out.println("in swap: a=" + x[0] + ", b=" + x[1]);
9.33     }
9.34 }
```

### 9.2.4 Strings genauer angeschaut

In Kapitel 4.6 wurden Strings (Zeichenketten) eingeführt und gleichzeitig auch schon darauf hingewiesen, dass `String` kein primitiver Datentyp in Java ist. `String` ist eine Klasse, darauf wird im Kapitel 11 im Detail eingegangen. An dieser Stelle ist wichtig, dass Klassen ebenfalls zu den Referenztypen gehören und damit Variablen vom Typ `String` nur Referenzen auf die eigentlichen Daten (Folge von Zeichen) enthalten.

Was bedeutet dies jetzt? Es soll mit dem Vergleich zweier Strings angefangen werden. Nach einer Deklaration (siehe Abbildung 9.8)

```

9-1  String s1 = "abc";
9-2  String s2 = "ab";
9-3  String s3 = "c";
9-4  String s4 = s2 + s3;

```

ist der Vergleich `s1 == s4` syntaktisch zulässig. Dieser Vergleich liefert aber nicht den vielleicht erwarteten Wert `true`. Denn es wird nicht der Inhalt der beiden Strings verglichen (der gleich ist), sondern es werden die Referenzen auf diese beiden Strings verglichen, die auf zwei verschiedene (aber inhaltsgleiche) Strings verweisen. Insofern liefert der Test `s1 == s4` für diesen Fall den Wert `false`. Möchte man zwei Strings beziehungsweise String-Variablen auf gleichen Inhalt testen, so muss man dazu die entsprechenden Stringfunktionen nutzen (`s1.equals(s4)`; siehe Dokumentation der Klasse `String` [Java]).

**Anmerkung:** Es sei an dieser Stelle darauf hingewiesen, dass das letzte Beispiel aufgrund einer Besonderheit der Implementierung von Java-Strings etwas aufwändiger wurde. Anstatt einfach `String s4 = "abc";` zu schreiben, wurde der Code `String s4 = s2 + s3;` mit entsprechenden `s2` und `s3` angegeben, der intern anders behandelt wird. Wird in Java nämlich ein String mit einem konstanten Wert angegeben (sprich in doppelten Anführungszeichen), so wird dieser String-Wert intern im sogenannten *String Pool* im Heap abgelegt. Werden im Java-Programm zwei Strings mit dem gleichen *konstanten* angegebenen Inhalt angelegt, so wird intern im String Pool dafür nur ein String-Wert angelegt, auf den beide String-Referenzen dann verweisen. Das hat Einfluss auf den `==` Operator, denn in diesem Fall ist sowohl der Inhalt der Strings gleich, als auch die beiden Referenzen. Dies ist aber nicht mehr der Fall, wenn der String-Wert nicht-konstant angegeben wird, z.B. als Ergebnis einer `+` Operation (siehe genutzter Code) oder etwa über einen Konstruktor (siehe später in Kap.11). Im Zusammenhang mit Strings sind einige Besonderheiten zu beachten, die nach der Einführung von Referenzen einfacher zu erläutern sind. Objekte vom Typ `String` werden in Java (anders als in anderen Programmiersprachen) *niemals* verändert. Eine Operation der Form `"abc" + "def"` hängt nicht an den ersten String den zweiten an, sondern es wird ein neuer String erzeugt, der die beiden String-Inhalte aneinander gekettet als Inhalt hat, also eine neue String `"abcdef"`.

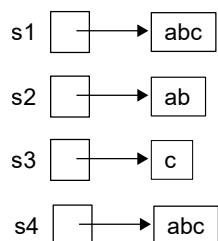


Figure 9.8: Situation nach der Deklaration von `s1` bis `s4`.

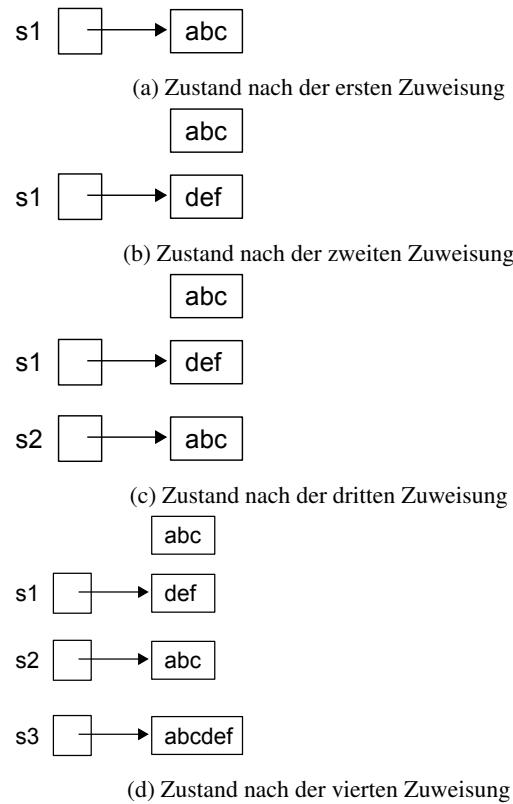


Figure 9.9: Ablauf der vier Anweisungen.

**Beispiel 9.15:**

Gegeben sind folgende vier Anweisungen:

```

9-1  String s1 = "abc";
9-2  s1 = "def";
9-3  String s2 = "abc";
9-4  String s3 = "abc"+"def";
  
```

Abbildung 9.9 zeigt die Situation nach der Ausführung der einzelnen Anweisungen. ♦

## 9.3 Zusammenfassung und Hinweise

### Verstehen

Neben der Kategorie der primitiven Datentypen gibt es in Java auch die Kategorie der Referenztypen. Die (interne) Behandlung von Werten und Variablen dieser beiden Kategorien ist sehr unterschiedlich. Durch Referenzen ergeben sich neue Möglichkeiten aber auch neue potentielle Probleme.

### Kurz und knapp merken

- Felder sind eine Anordnung von n Elementen eines Basistyps. In Java ist `T[]` ein Feldtyp zu einem beliebigen Basistyp `T`.
- Die Länge eines Feldes `a` kann man dynamisch über `a.length` ermitteln.
- Feldobjekte werden mit `new` beziehungsweise implizit über den Feldinhalt angelegt.

- Auf ein Feldelement greift man über einen ganzzahligen Index `i` zu: `a[i]`.
- Mehrdimensionale Felder sind in Java Felder von Feldern, die Elemente in der ersten Dimension sind also selber wieder Felder und werden so behandelt.
- Referenzen sind Speicheradressen im (virtuellen) Adressraum.
- Eine Variable eines Referenztyps enthält immer eine Referenz.
- `null` ist ein spezieller Referenzwert.
- Auf ein Objekt können 0, 1 oder mehrere Objekte zeigen.
- Der `==`-Operator angewandt auf zwei Referenzen vergleicht die Referenzen auf Gleichheit, aber nicht den Inhalt der Objekte, auf die die Referenzen verweisen.
- Durch die Übergabe einer Referenz (call-by-value!) hat man in aufgerufenen der Methode indirekt Zugriff auf den Originalwert.

## Häufige Fehler

- Felder der Länge `n` beginnen mit dem Index `0` und haben als letzten Index `n-1`, und *nicht n*.
- Der Umgang mit Referenzen ist für Anfänger fast immer sehr gewöhnungsbedürftig und es treten zu Beginn oft viele Fehler im Verständnis ihrer Anwendung, ihrer Möglichkeiten und der potentiellen Problemen im Umgang mit ihnen auf.
- Man muss jederzeit dafür sorgen, dass beim Zugriff auf ein Objekt die entsprechende Referenz nicht `null` ist.
- Sind `x` und `y` Referenzvariablen (oder allgemein Referenzausdrücke), so bewirkt `x == y` ein Vergleich der Referenzen und nicht der Objekte hinter den Referenzen.

## Übungsfragen

- Was sind alle Referenztypen in Java (aufzählen)?
- Was ist der Unterschied zwischen einem Feldtyp und einem Feldwert? Beispiel!
- Wie kann man eine Matrix mit Wahrheitswerten der Größe `n x m` anlegen?
- Kann man ein Feldobjekt vom Typ `boolean[]` nachträglich vergrößern oder verkleinern?
- Was ist der Wert des Ausdrucks `new boolean[23]`?
- Was ist (ungefähr) der Speicherbedarf des Feldes `new boolean[23]`?
- Gegeben ist: `int[] a = new int[10];` Was macht diese Deklaration exakt? Was ist anschließend der Wert des Ausdrucks `a`?
- Gegeben ist: `int[] a = new int[10];` Was macht diese Deklaration exakt? Was kommt in der Methode `f` als Wert an: `f(a, a[0]);`?
- Was ist das Ergebnis von `int[] a = {1, 2, 3};` Was passiert genau in dieser Deklaration?
- Wieso heißt es `a.length` für ein Feld, aber `s.length()` für einen String?
- Gegeben: `int[] a = new int[10]; int i= 3;`  
Ist `a[++i]++`; erlaubt? Was ist der Wert der Variablen vor und nach dieser Anweisung?
- Wie kann man ein int-Feld `a` initialisieren, in dem der `i`-te Wert des Feldes den Wert `Feldlänge-i` erhält?
- Was sehen Sie: `int[] a = {1, 2, 3}; System.out.println(a[3]);`
- Suche nach dem Index des ersten Vorkommens eines Wertes in einem Feld?

- Suche nach dem Index des letzten Vorkommens eines Wertes in einem Feld?
- Kann ich in Java ein 2D-Feld anlegen, in dem die Zeilen unterschiedliche Länge haben? Wenn ja, wie?
- Wie kann ich die Anzahl an Zeilen und Spalten eines 2D-Feldes a bekommen?
- Wie lege ich ein 2D-Feld der Dimension  $n \times m$  an mit Inhalt  $a_{i,j} = i + j$ ?
- Wie gehe ich vor, wenn ich alle Permutationen des Inhalts eines Feldes erzeugen will?
- Was passiert genau: `int[] a = {1, 2, 3}; a[1];`
- Was passiert genau: `String s1 = "A"; String s2 = s1; String s3 = s1 + ';`  
`String s1 = "A"; String s2, s3;`  
 s2 soll auf den gleichen String verweisen, s3 auf einen anderen String gleichen Inhalts.
- `String s1 = "A", s2=s1, s3 = `` + s1;`  
 Ergebnis von s1==s2, s1==s3, s1.equals(s2), s1.equals(s3)?
- Kann es mehrere Referenzvariablen geben, die auf das gleiche Objekt verweisen? Wenn ja, Beispiel!
- `int[][] a = {{1, 2}, {3, 4}};`  
 Geben Sie eine Deklaration einer Variablen b an, die eine Referenz auf die zweite Zeile enthält!
- `int[][] a = {{1, 2}, {3, 4}}; int[] b = a[1]; b[0] = 7;`  
 Ist das erlaubt? Wenn ja, was bewirkt es?
- Wofür steht null? Wie genutzt?
- Wie kann man eine Methode swap für int-Werte in Java programmieren?

## Reflektion des Stoffs

- Wäre das möglich: `, 2, 3 (new int[1]).length`? Wenn nein, wieso nicht? Wenn ja, was wäre das Ergebnis?
- Ergänzen Sie den folgenden Programmausschnitt so, dass verschiedene Werte in der Programmausgabe erscheinen, ohne dass die Variable **a** auf der linken Seite einer Zuweisung erscheint:

```

9.1 int[] a = {1};
9.2 System.out.println(a[0]);
9.3 System.out.println(a[0]);

```



# Chapter 10

## Abstrakte Datentypen

Dieses Kapitel ist eher grundlagenorientiert und soll dabei helfen, die nachfolgende Einführung in konkrete Basisdatentypen und später auch in die Objektorientierung vorzubereiten.

In diesem Kapitel werden Datentypen von einem grundsätzlichen Standpunkt aus betrachtet, in folgenden Kapitel dann konkret in der Umsetzung auf konkreten Rechnern und in Konstrukten konkreter Programmiersprachen. Auf komplexe Daten wird dann in einem späteren Kapitel eingegangen.

### 10.1 Abstrakter Datentyp

In einer ersten Näherung ist ein Datentyp eine Wertemenge mit einer Menge von Operationen auf dieser Wertemenge.

#### Beispiel 10.1:

Die Boolschen Werte  $\mathbb{B} = \{\text{wahr}, \text{falsch}\}$  mit folgenden Operationen sind bekannt:

$$\begin{aligned}\neg & : \mathbb{B} \rightarrow \mathbb{B} \\ \wedge & : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B} \\ \vee & : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B} \\ & \dots\end{aligned}$$

Genau wie im Problemlösungsprozess sollen auch bei der Betrachtung von Datentypen zwischen der Spezifikation und der konkreten Realisierung unterschieden werden. Dies führt zum Begriff des abstrakten Datentyps, durch den ein Datentyp spezifiziert wird, jedoch noch keine Angaben über eine konkrete Implementierung gemacht werden.

#### Definition 10.1 (Abstrakter Datentyp):

Sei  $S$  eine Menge von Sorten (Wertebereiche) und  $Ops$  eine Menge von Operationssymbolen. Ein abstrakter Datentyp  $T = \langle S^T, Ops^T \rangle$  besteht aus einer Familie von Trägermengen  $\{s_i^T \mid s_i \in S\}$  und einer Familie von Operationen  $\{Ops_i^T \mid Ops_i \in Ops\}$  auf diesen Trägermengen. ♦

Diese auf den ersten Blick sehr formale Definition lässt sich wie folgt interpretieren. In der Menge  $S$  sind Namen von Wertemengen enthalten, beispielsweise  $X, Y, \mathbb{N}$ . Dies sind zuerst einmal reine Namen, ohnen ihnen eine bestimmte Bedeutung zu geben. Ebenso ist die Menge  $Ops$  eine Menge von Namen von Operationen, auch hier ohne anzugeben, was diese Namen bedeuten sollen. Ein abstrakter Datentyp gibt jedem Namen nun eine Bedeutung, indem jeweils zu einer Sorte (also einem Namen) gesagt wird, was für eine Wertemenge dahintersteht, und zu jeder Operation (also auch hier dem Namen der Operation) angibt, was diese Operation inhaltlich tun soll.

### Beispiel 10.2:

Es soll ein abstrakter Datentyp *boolean* definiert werden. Die Menge  $S$  der Sorten wird mit  $\{ \text{bool} \}$  festgelegt, die Menge der Operationssymbole sei  $Ops = \{ \text{not}, \text{and}, \text{or} \}$ . Das sind damit die *Namen* für Wertemengen und Operationen.

Die Trägermenge zu *bool* wird mit  $\mathbb{B} = \{ \text{wahr}, \text{falsch} \}$  definiert und die Operationen in der Menge  $Ops^{boolean}$  zu den Operationssymbolen sind:

- $\text{not}^{boolean} : \mathbb{B} \rightarrow \mathbb{B}$  mit  $x \mapsto \neg x$
- $\text{and}^{boolean} : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$  mit  $(x, y) \mapsto x \wedge y$
- $\text{or}^{boolean} : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$  mit  $(x, y) \mapsto x \vee y$

Damit wurde den Namen eine *Bedeutung* gegeben.

Der abstrakte Datentyp *boolean* kann mit den obigen Definitionen dann komplett angegeben werden als  $\text{boolean} = < \{ \mathbb{B} \}, Ops^{boolean} >$ . ◆

### Beispiel 10.3:

Als zweites Beispiel soll die Menge der natürlichen Zahlen als abstrakten Datentyp *nat* angeben werden. Die Menge der Sorten ist  $S = \{ \text{bool}, \text{NZahlen} \}$  und die Menge der Operationssymbole (hier zur Vereinfachung nur eine kleine Auswahl) ist  $Ops = \{ \text{add}, \text{sub}, \text{gleich}, \text{not}, \text{and}, \text{or} \}$ . Auch hier wurde damit lediglich zwei Mengen mit Namen angegeben, ohne auf die Bedeutung der Namen einzugehen.

Die Trägermengen sind  $\mathbb{B} = \{ \text{wahr}, \text{falsch} \}$  zu *bool* und  $\mathbb{N} = \{ 0, 1, 2, \dots \}$  zu *NZahlen*. Die Realisierung der Operationssymbole  $Ops^{nat}$  ist:

- $\text{add}^{nat} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  mit  $(x, y) \mapsto x + y$
- $\text{sub}^{nat} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  mit  $(x, y) \mapsto x - y$   
(modifizierte Subtraktion wie bei der URM, um in der Menge  $\mathbb{N}$  zu bleiben).
- $\text{gleich}^{nat} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$  mit  $(x, y) \mapsto \text{wahr}$ , falls  $x$  gleich  $y$ , und  $\text{falsch}$  sonst
- $\text{not}^{nat} : \mathbb{B} \rightarrow \mathbb{B}$  mit  $x \mapsto \neg x$
- $\text{and}^{nat} : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$  mit  $(x, y) \mapsto x \wedge y$
- $\text{or}^{nat} : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$  mit  $(x, y) \mapsto x \vee y$

Damit lässt sich der abstrakte Datentyp zu den natürlichen Zahlen angeben als  $\text{nat} = < \{ \mathbb{B}, \mathbb{N} \}, Ops^{nat} >$ . ◆

Im Folgenden soll zur einfacheren Darstellung eines abstrakten Datentyps die Angabe der Menge  $S$  der Sorten und der Menge  $Ops$  der Operationssymbole stillschweigend weglassen werden. Wie man an den beiden Beispielen sehen kann sind die Trägermengen und die Operationen der bedeutsame Teil.

Unterscheiden muss man zwischen einem **abstrakten Datentypen** und einer **konkreten Implementierung** eines Datentyps auf einem bestimmten Rechner. Der obige Datentyp *nat* der natürlichen Zahlen ist als abstrakter Datentyp angegeben worden, man kann mit den dort definierten Werten und Operationen "rechnen", über eine konkrete Realisierung ist aber nichts ausgesagt worden. Für eine konkrete Realisierung müsste man

zum Beispiel festlegen, wie intern eine Zahl aus  $n \in \mathbb{N}$  repräsentiert werden soll. Zum Beispiel repräsentieren Menschen die Zahl 1 als ein halber schräger Strich hoch und ein senkrechter Strich herunter (Amerikaner haben eine andere Repräsentation, nämlich nur einen senkrechten Strich herunter). Ein Rechner würde die Zahl durch ein bestimmtes Bitmuster darstellen. Zum Beispiel ist es möglich, den oben definierten abstrakten Datentyp *nat* der natürlichen Zahlen in einer Mikroprozessorsteuerung für eine Waschmaschine (*Embedded Controller*) konkret zu implementieren und genauso den gleichen Datentyp auf einem Supercomputer auf eine andere Weise, d.h. mit einer anderen Darstellung für Zahlenwerte konkret zu implementieren. Da Implementierungen nur eine endliche Zahl von Zahlen darstellen können, wird die Wertemenge allerdings nur eine Teilmenge der unendlichen Menge der natürlichen Zahlen sein. Der abstrakte Datentyp bliebe aber in beiden Fällen der gleiche. Zuerst soll deshalb in diesem Kapitel ausschließlich nur mit abstrakten Datentypen gearbeitet werden, um später zu konkreten Implementierungen zu kommen.

Abstrakte Datentypen sind sehr hilfreich in einer frühen Phase des Entwurfsprozesses. Ähnlich dem Vorgehen bei der schrittweisen Verfeinerung lässt man die Details zuerst außen vor und konzentriert sich auf die wesentlichen Eigenschaften des Datentyps. Dabei hat man die Freiheit, bei der Umsetzung in einen konkreten Datentyp in einer späteren Phase des Entwurfsprozesses durchaus zwischen mehreren Implementierungsalternativen wählen zu können, ja sogar diese austauschen zu können, solange nur die Wertemenge dargestellt werden kann und die Operationen mit der definierten Wirkungsweise implementiert sind.

#### **Beispiel 10.4:**

Gegeben ist folgende Problemstellung. Zu vier Punkten  $P_1, \dots, P_4$  im zweidimensionalen euklidischen Raum  $\mathbb{R}^2$  soll die Weglänge  $l$  des Weges  $P_1 - P_2 - P_3 - P_4$  berechnet werden.

Die Berechnung der Gesamtweglänge kann zurückgeführt werden auf die Berechnung des Abstandes zwischen zwei Punkten. Der Lösungsalgorithmus sieht damit wie folgt aus:  $l = \text{Abstand}(P_1, P_2) + \text{Abstand}(P_2, P_3) + \text{Abstand}(P_3, P_4)$ .

Um den Abstand zwischen zwei Punkten berechnen zu können muss man wissen, in welcher Form die Punkte angegeben sind: kartesische Koordinaten der Form  $(x, y)$  oder Polarkoordinaten der Form  $(r, \varphi)$ ? Von dieser Darstellungsform der Einzelpunkte hängt auch die Berechnungsformel ab, mit der man den Abstand zweier Punkte berechnen kann. Für kartesische Koordinaten berechnet sich der Abstand  $d$  zweier Punkte  $P_1 = (x_1, y_1)$  und  $P_2 = (x_2, y_2)$  mit  $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ . Der Abstand  $d$  für zwei Punkte in Polarkoordinaten,  $P_1 = (r_1, \varphi_1)$  und  $P_2 = (r_2, \varphi_2)$ , kann berechnet werden mit der Formel  $d = \sqrt{r_1^2 + r_2^2 - 2r_1r_2 \cos(\varphi_2 - \varphi_1)}$ . Unabhängig von der konkreten Realisierung, kartesische Koordinaten oder Polarkoordinaten, lässt sich aber obiger Algorithmus angeben als die Summation der Abstände zwischen zwei Punkten:  $l = \text{Abstand}(P_1, P_2) + \text{Abstand}(P_2, P_3) + \text{Abstand}(P_3, P_4)$ . ◆

Im Folgenden sollen nun einige Beispiele für abstrakte Datentypen vorgestellt werden. Ein wesentlicher Erkenntnispunkt sollte dabei sein zu erkennen, wie man ausgehend von den Basisoperationen weitere darauf aufbauende Operationen definieren kann. Die Definition und Nutzung neuer Operationen basierend auf bekannten Grundoperationen ist natürlich nichts anderes als Programmierung!

## **10.2 Folge**

Begonnen werden soll die Betrachtung mit einem abstrakten Datentyp Folge. Dazu wird zu Beginn erst einmal präzise definiert, was im mathematischen Sinn eine Folge ist. Seien  $M_1, M_2, \dots, M_n, n \geq 0$  Mengen. Jedes Element  $(x_1, \dots, x_n) \in M_1 \times \dots \times M_n$  heißt Tupel der Länge  $n$ .  $x_i$  heißt  $i$ -te Komponente des Tupels. Die

eben betrachteten Punkte in Form von kartesischen Koordinaten sind beispielsweise Tupel der Länge 2 aus  $\mathbb{R} \times \text{ReelleZahlen}$ .

**Definition 10.2 (Folge):**

Zu einer Basismenge  $M$  heißt ein Tupel  $(m_1, \dots, m_n)$  der Länge  $n, n \leq 0$  aus  $M^n$  eine **Folge** über  $M$ . ♦

Als Schreibweisen werden benutzt:

- Man notiert üblicherweise spitze statt runde Klammern für Folgen:  $x = < x_1, \dots, x_n >, x_i \in M, 1 \leq i \leq n, n \geq 0$
- Für  $n = 0$  gilt:  $x = <> =: \varepsilon$  (leere Folge)

**Definition 10.3 ( $M^*, M^+$ ):**

Sei  $M$  eine Menge. Dann ist  $M^n$  für  $n \geq 0$  rekursiv definiert durch:

$$\begin{aligned} M^0 &= \{\varepsilon\} \\ M^1 &= M \\ M^{n+1} &= M^n \times M \end{aligned}$$

Weiterhin ist definiert:

$$\begin{aligned} M^* &= M^0 \cup M^1 \cup M^2 \cup \dots = \bigcup_{i \geq 0} M^i && (\text{Menge aller Folgen über } M) \\ M^+ &= M^1 \cup M^2 \cup \dots = \bigcup_{i > 0} M^i && (\text{Menge aller nichtleeren Folgen über } M) \end{aligned}$$

**Beispiel 10.5:**

Sei  $M = \{a, b\}$ .

$$\begin{aligned} M^0 &= \{\varepsilon\} \\ M^1 &= \{< a >, < b >\} \\ M^2 &= \{< a, a >, < a, b >, < b, a >, < b, b >\} \\ M^3 &= \{< a, a, a >, < a, a, b >, < a, b, a >, < a, b, b >, \dots, < b, b, b >\} \end{aligned}$$

Sei  $M$  nun eine beliebige nichtleere Basismenge. Betrachtet wird der abstrakte Datentyp

$$\text{folge} = < \{M, M^*, \mathbb{B}\}, \text{Ops}^{\text{folge}} >,$$

wobei die Operationen

$$\text{Ops}^{\text{folge}} = \{\text{emptyfolge}, \text{makefolge}, \text{first}, \text{rest}, \text{concat}, \text{isemptyfolge}, \text{isequal}\}$$

wie folgt definiert sind:

1.  $\text{emptyfolge} : \emptyset \rightarrow M^*$  mit:

$$\text{emptyfolge}() = \varepsilon$$

Erzeugt die leere Folge  $\varepsilon$ .

2.  $\text{makefolge} : M \rightarrow M^*$  mit:

$$\text{makefolge}(m) = \langle m \rangle$$

Erzeugt eine Folge, deren einziges Element  $m$  (ein Element der Basismenge) ist.

3.  $\text{first} : M^* \rightarrow M$  mit:

$$\text{first}(f) = \begin{cases} \perp & \text{falls } f = \varepsilon \\ x_1 & \text{falls } f = \langle x_1, \dots, x_n \rangle \text{ mit } n > 0 \end{cases}$$

Ergibt das erste Element der Folge. Zu beachten ist, dass diese Operation nur definiert ist, falls die Folge nichtleer ist.  $\perp$  bedeutet dabei "nicht definiert".

4.  $\text{rest} : M^* \rightarrow M^*$  mit:

$$\text{rest}(f) = \begin{cases} \perp & \text{falls } f = \varepsilon \\ \langle x_2, \dots, x_n \rangle & \text{falls } f = \langle x_1, \dots, x_n \rangle \text{ mit } n > 0 \end{cases}$$

Ergibt den Rest der Folge ohne das erste Element. Für  $n = 1$ , d.h.  $f = \langle x_1 \rangle$ , ergibt entsprechend obiger Definition  $\text{rest}(f) = \varepsilon$ . Auch hier muss die Folge nichtleer sein.

5.  $\text{concat} : M^* \times M^* \rightarrow M^*$  mit:

$$\text{concat}(f_1, f_2) = f_1 \circ f_2$$

wobei für zwei Folgen  $f_1 = \langle x_1, \dots, x_n \rangle$  und  $f_2 = \langle y_1, \dots, y_m \rangle$  ( $n, m \geq 0$ ) der  $\circ$ -Operator definiert ist durch:  $\langle x_1, \dots, x_n \rangle \circ \langle y_1, \dots, y_m \rangle = \langle x_1, \dots, x_n, y_1, \dots, y_m \rangle$ .  $\text{concat}$  erzeugt aus zwei Folgen eine neue Folge, die durch das Aneinanderfügen der ersten mit der zweiten Folge entsteht.

6.  $\text{isemptyfolge} : M^* \rightarrow \mathbb{B}$  mit:

$$\text{isemptyfolge}(f) = \begin{cases} \text{wahr} & \text{falls } f = \varepsilon \\ \text{falsch} & \text{sonst} \end{cases}$$

Testet eine Folge auf eine leere Folge.

7.  $\text{isequal} : M^* \times M^* \rightarrow \mathbb{B}$  mit:

$$\text{isequal}(f_1, f_2) = \begin{cases} \text{wahr} & \text{falls } (f_1 = \varepsilon) \wedge (f_2 = \varepsilon) \\ \text{falsch} & \text{falls } (f_i = \varepsilon) \wedge (f_j \neq \varepsilon), \\ & i, j \in \{1, 2\}, i \neq j \\ (f_1 = \varepsilon) \wedge & \\ \text{isequal}(\text{rest}(f_1), \text{rest}(f_2)) & \text{sonst} \end{cases}$$

Testet, ob zwei Folgen gleich sind, d.h. gleiche Länge haben und komponentenweise gleich sind. Hierbei wird der Vergleich zweier beliebiger Folgen letztendlich (über Rekursion) auf den Vergleich zweier Elemente aus  $M$  zurückgeführt.

In der Definition der Funktion  $\text{isequal}$  werden "komplizierte" Fälle, in denen beide Folgen mehr als ein Element haben ("sonst"-Fall), auf einfache Fälle zurückgeführt. Dies ist eine sogenannte rekursive Definition und dies wird vielfach noch verwendet. **Rekursiv** bedeutet in der Informatik so viel wie **selbstbezüglich** oder **auf sich selbst verweisend**.

**Beispiel 10.6:**

Sei  $M = \{a, b\}$ .

1. Die Folge  $f = \langle a, b \rangle$  lässt sich mit Hilfe der Basisoperationen wie folgt erzeugen:

$$f = concat(makefolge(a), makefolge(b))$$

mit der üblichen Bedeutung von Funktionsauswertungen.

2. Der Test, ob die beiden Folgen  $f_1 = \langle a, b, c \rangle$  und  $f_2 = \langle a, b, a \rangle$  gleich sind, findet Schritt für Schritt wie folgt statt:

$$isequal(\langle a, b, c \rangle, \langle a, b, a \rangle)$$

$$\begin{aligned} &= (first(\langle a, b, c \rangle) = first(\langle a, b, a \rangle)) \\ &\quad \wedge isequal(rest(\langle a, b, c \rangle), rest(\langle a, b, a \rangle))) \\ &= (a = a) \wedge isequal(\langle b, c \rangle, \langle b, a \rangle) \\ &= (first(\langle b, c \rangle) = first(\langle b, a \rangle)) \wedge isequal(rest(\langle b, c \rangle), rest(\langle b, a \rangle)) \\ &= (b = b) \wedge isequal(\langle c \rangle, \langle a \rangle) \\ &= (first(\langle c \rangle) = first(\langle a \rangle)) \wedge isequal(rest(\langle c \rangle), rest(\langle a \rangle)) \\ &= (c = a) \wedge isequal(\langle \rangle, \langle \rangle) \\ &= falsch \end{aligned}$$



Folgen können durch die sukzessive Ausführung von concat-Operationen beliebig lang werden. Alle nichtleeren Folgen haben dabei aber einen gemeinsamen Aufbau: sie bestehen aus einem ersten Folgeglied und einem Rest, der wiederum eine Folge ist. Um nun als Grundoperationen nicht eine Vielzahl von Operationen bereitzustellen zu müssen (zum Beispiel erzeuge Folge mit 2,3,...,n Gliedern; gib Restfolge ohne die ersten 2,3,...,n Glieder), wird folgendes Prinzip angewandt, das dem oben beschriebenen (rekursiven) Aufbau von Folgen Rechnung trägt. Möchte man eine Operation für beliebige Folgen definieren, so ist das allgemeine Vorgehen so, dass man die Operation auf das erste Folgeglied anwendet und ggf. auf den Rest der Folge.

**Beispiel 10.7:**

Gegeben ist die Aufgabenstellung, in einer Folge von natürlichen Zahlen nach dem Vorkommen des Wertes 1 zu suchen. Ist der Wert 1 in der Folge enthalten, soll *wahr* das Resultat der Operation sein, ansonsten *falsch*. Die Lösung kann man nun so formulieren, dass für eine gegebene Folge  $f$  drei Fälle auftreten können:

- 1) Die Folge ist leer, die Folge enthält damit auch keine 1 und damit ist das Resultat der Operation *falsch*.
- 2) Das erste Folgeglied hat den Wert 1 und damit ist das Ergebnis *wahr*.
- 3) Andernfalls – eine nichtleere Folge, deren ersten Glied ungleich der gesuchten 1 ist – wird die Operation auf den Rest der Folge angewandt.

Diese drei Punkte lassen sich in als Grundgerüst für eine Funktionsdefinition nutzen:

$hat1 : M^* \rightarrow \mathbb{B}$  mit:

$$hat1(f) = \begin{cases} falsch & \text{falls } f = \epsilon \\ wahr & \text{falls } first(f) = 1 \\ hat1(rest(f)) & \text{sonst} \end{cases}$$



Mit Hilfe der Operationen dieses Datentyps lassen sich nun weitere nützliche Funktionen definieren / programmieren. Ein Palindrom ist ein Wort oder Satz, in dem der i-te Buchstabe von vorne gleich dem i-ten Buchstaben von hinten gesehen ist. Beispiel (ohne Beachtung der Groß-/Kleinschreibung): *Otto*.

Umgangssprachlich beschrieben lautet ein möglicher Algorithmus für *palindrom* also: Falls die Folge leer ist oder nur aus einem Folgeglied besteht, so ist die Folge ein Palindrom. Ansonsten besteht die Folge aus mindestens zwei Folgegliedern. Falls das erste und letzte Folgeglied nicht übereinstimmen, kann die Folge kein Palindrom mehr sein. Ansonsten wende den Algorithmus rekursiv auf die um das erste und letzte Folgeglied gekürzte Folge an.

Um die Funktion (Operation) *palindrom* zu definieren, braucht man einige (Hilfs-) Funktionen, da man mit den bis jetzt zur Verfügung stehenden Funktionen zum Beispiel nicht das letzte Folgeglied für eine beliebige Folge anschauen kan.

1. *spiegel* :  $M^* \rightarrow M^*$  mit:

$$\text{spiegel}(f) = \begin{cases} \epsilon & \text{falls } f = \epsilon \\ \text{concat}(\text{spiegel}(\text{rest}(f)), \text{makefolge}(\text{first}(f))) & \text{sonst} \end{cases}$$

Die Funktion *spiegel* liefert für eine Folge  $\langle x_1, \dots, x_n \rangle$  die gespiegelte Folge  $\langle x_n, \dots, x_1 \rangle$ .

### Beispiel 10.8:

$$\begin{aligned} \text{spiegel}(\langle a, b, c \rangle) &= \text{concat}(\text{spiegel}(\text{rest}(\langle a, b, c \rangle)), \text{makefolge}(\text{first}(\langle a, b, c \rangle))) \\ &= \text{concat}(\text{spiegel}(\langle b, c \rangle), \langle a \rangle) \\ &= \text{concat}(\text{concat}(\text{spiegel}(\text{rest}(\langle b, c \rangle)), \text{makefolge}(\text{first}(\langle b, c \rangle))), \langle a \rangle) \\ &= \text{concat}(\text{concat}(\text{spiegel}(\langle c \rangle), \langle b \rangle), \langle a \rangle) \\ &= \text{concat}(\text{concat}(\text{concat}(\text{spiegel}(\text{rest}(\langle c \rangle)), \text{makefolge}(\text{first}(\langle c \rangle))), \langle b \rangle), \langle a \rangle) \\ &= \text{concat}(\text{concat}(\text{concat}(\text{spiegel}(\epsilon), \langle c \rangle), \langle b \rangle), \langle a \rangle) \\ &= \text{concat}(\text{concat}(\text{concat}(\epsilon, \langle c \rangle), \langle b \rangle), \langle a \rangle) \\ &= \text{concat}(\text{concat}(\langle c \rangle, \langle b \rangle), \langle a \rangle) \\ &= \text{concat}(\langle c, b \rangle, \langle a \rangle) \\ &= \langle c, b, a \rangle \end{aligned}$$

2. *head* :  $M^+ \rightarrow M^*$  mit:

$$\text{head}(f) = \begin{cases} \text{spiegel}(\text{rest}(\text{spiegel}(f))) & \text{falls } f = \langle x_1, \dots, x_n \rangle \text{ mit } n > 0 \\ \perp & \text{sonst} \end{cases}$$

Die Funktion *head* liefert für eine Folge  $\langle x_1, \dots, x_n \rangle$  die Folge  $\langle x_1, \dots, x_{n-1} \rangle$ , d.h. die ursprüngliche Folge ohne das letzte Folgeglied.

### Beispiel 10.9:

$$\begin{aligned} \text{head}(\langle a, b, c \rangle) &= \text{spiegel}(\text{rest}(\text{spiegel}(\langle a, b, c \rangle))) \\ &= \text{spiegel}(\text{rest}(\langle c, b, a \rangle)) \\ &= \text{spiegel}(\langle b, a \rangle) \\ &= \langle a, b \rangle \end{aligned}$$

3.  $last : M^* \rightarrow M$  mit:

$$last(f) = \begin{cases} \perp & \text{falls } f = \epsilon \\ first(f) & \text{falls } rest(f) = \epsilon \\ last(rest(f)) & \text{sonst} \end{cases}$$

Die Funktion  $last$  liefert für eine Folge  $\langle x_1, \dots, x_n \rangle$  das letzte Element  $x_n$ .

**Beispiel 10.10:**

$$\begin{aligned} last(\langle a, b, c \rangle) &= last(rest(\langle a, b, c \rangle)) \\ &= last(\langle b, c \rangle) \\ &= last(rest(\langle b, c \rangle)) \\ &= last(\langle c \rangle) \\ &= first(\langle c \rangle) \\ &= c \end{aligned}$$

Mit Hilfe der definierten Funktionen lässt sich nun die Funktion  $palindrom : M^* \rightarrow \mathbb{B}$  definieren, die für eine gegebene Folge von Buchstaben  $\langle x_1, \dots, x_n \rangle$  entscheidet, ob die Folge ein Palindrom darstellt oder nicht.

$palindrom : M^* \rightarrow \mathbb{B}$  mit

$$palindrom(f) = \begin{cases} wahr & \text{falls } (f = \epsilon) \vee (rest(f) = \epsilon) \\ falsch & \text{falls } first(f) \neq last(f) \\ palindrom(rest(head(f))) & \text{falls } first(f) = last(f) \end{cases}$$

**Beispiel 10.11:**

$$\begin{aligned} palindrom(\langle o, t, t, o \rangle) &= palindrom(rest(head(\langle o, t, t, o \rangle))) \\ &= palindrom(rest(\langle o, t, t \rangle)) \\ &= palindrom(\langle t, t \rangle) \\ &= palindrom(rest(head(\langle t, t \rangle))) \\ &= palindrom(rest(\langle t \rangle)) \\ &= palindrom(\epsilon) \\ &= wahr \end{aligned}$$

**Beispiel 10.12:**

$$\begin{aligned} palindrom(\langle o, r, t, o \rangle) &= palindrom(rest(head(\langle o, r, t, o \rangle))) \\ &= palindrom(rest(\langle o, r, t \rangle)) \\ &= palindrom(\langle r, t \rangle) \\ &= falsch \end{aligned}$$

Eine andere (wesentlich einfache) Möglichkeit der "Programmierung" von *palindrom* wäre

$$\text{palindrom}(f) := \text{isequal}(f, \text{spiegel}(f))$$

Jetzt soll eine weitere nützliche Operation auf Folgen definiert werden. Sei  $M$  total geordnet. Dann lässt sich eine Funktion  $\text{istsortiert} : M^* \rightarrow \mathbb{B}$  definieren mit:

$$\text{istsortiert}(f) = \begin{cases} \text{wahr} & \text{falls } f = \varepsilon \vee \text{rest}(f) = \varepsilon \\ \text{falsch} & \text{falls } \text{first}(f) > \text{first}(\text{rest}(f)) \\ \text{istsortiert}(\text{rest}(f)) & \text{sonst} \end{cases}$$

Die Funktion liefert den Wert *wahr*, wenn die Folge  $f$  aufsteigend sortiert ist, ansonsten *falsch*. Die Menge  $M_{<}^* := \{f \in M^* \mid \text{istsortiert}(f) = \text{wahr}\}$  bezeichnet dann die Menge aller Folgen, die aufsteigend sortiert sind. Anhand einer einfachen Funktion soll nun eine weitere Fragestellung erörtert werden. Die Funktion ist *ersetzen* :  $M^* \times M \times M \rightarrow M^*$ , die für einen Aufruf *ersetzen*( $f, a, b$ ) eine Folge liefert, die der ursprünglichen Folge  $f$  gleicht, jedoch – soweit vorhanden – das erste Vorkommen von  $a$  in  $f$  durch  $b$  ersetzt. Als Beispiel also: *ersetzen*( $<1, 2, 3, 3, 4>, 3, 6$ ) =  $<1, 2, 6, 3, 4>$ .

Umgangssprachlich könnte man eine Lösung wie folgt formulieren, wobei wiederum die Struktur einer Folge den Rahmen vorgibt. Falls die Folge leer ist, so ist das Resultat die leere Folge, in diesem Fall ist auch nichts zu ersetzen. Ansonsten ist die Folge nicht leer. Falls das erste Element der Folge gleich dem gesuchten  $a$  ist, so konstruieren man die Resultatfolge, indem man die Resultatfolge mit  $b$  beginnen lassen und die Restfolge (ohne  $a$ ) anhängen. Falls das erste Element der Folge jedoch ungleich  $a$  ist, so konstruiert man die Resultatfolge, indem man das erste Element unverändert in die Resultatfolge einfügt, die Ersetzungsoperation auf die Restfolge anwendt und die daraus resultierende Folge anhängt.

*ersetzen* :  $M^* \times M \times M \rightarrow M^*$  mit:

$$\text{ersetzen}(f, a, b) = \begin{cases} \varepsilon & \text{falls } f = \varepsilon \\ \text{concat}(\text{makefolge}(b), \text{rest}(f)) & \text{falls } \text{first}(f) = a \\ \text{concat}(\text{makefolge}(\text{first}(f)), \text{ersetzen}(\text{rest}(f), a, b)) & \text{sonst} \end{cases}$$

Interessant an dieser Stelle ist die Technik, wie man bereits bearbeitete Daten – die bereits bearbeiteten Folgeglieder – mit noch zu bearbeitenden Daten – die Restfolge – verknüpft. Dies wird auch im folgenden Beispiel deutlich.

### Beispiel 10.13:

$$\text{ersetzen}(<1, 2, 3, 3, 4>, 3, 6)$$

$$\begin{aligned} &= \text{concat}(<1>, \text{ersetzen}(<2, 3, 3, 4>, 3, 6)) \\ &= \text{concat}(<1>, \text{concat}(<2>, \text{ersetzen}(<3, 3, 4>, 3, 6))) \\ &= \text{concat}(<1>, \text{concat}(<2>, \text{concat}(<6>, \text{ersetzen}(<3, 4>, 3, 6)))) \\ &= \text{concat}(<1>, \text{concat}(<2>, \text{concat}(<6>, \text{concat}(<6>, \text{ersetzen}(<4>, 3, 6)))))) \\ &= \text{concat}(<1>, \text{concat}(<2>, \text{concat}(<6>, \text{concat}(<6>, \text{concat}(<4>, \text{ersetzen}(<>, 3, 6))))))) \\ &= \text{concat}(<1>, \text{concat}(<2>, \text{concat}(<6>, \text{concat}(<6>, \text{concat}(<4>, \varepsilon)))))) \\ &= \text{concat}(<1>, \text{concat}(<2>, \text{concat}(<6>, \text{concat}(<6>, \text{concat}(<4>, <4>))))) \\ &= \text{concat}(<1>, \text{concat}(<2>, \text{concat}(<6>, <6, 4>))) \\ &= \text{concat}(<1>, \text{concat}(<2>, <6, 6, 4>)) \\ &= \text{concat}(<1>, <2, 6, 6, 4>) \\ &= <1, 2, 6, 6, 4> \end{aligned}$$

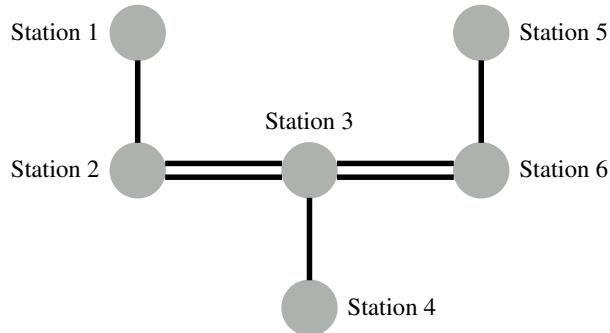


Figure 10.1: Beispiel für eine Schemazeichnung eines Bahnnetszes

Ein weiterer an dieser Stelle anzumerkender Punkt ist der funktionale Gedanke bei der Definition neuer Operationen, wie er bereits bei der Motivation zur Primitiven Rekursion angesprochen wurde. Entscheidend ist *was* ausgeführt wird, nicht das *wie*. Später wird man sich darüber Gedanken machen müssen, wie (genau!) auf einem Computer die rekursive Ausführung einer Funktion verwaltet werden kann, zum Beispiel wo Zwischenergebnisse abgelegt werden, wie die Folge abgespeichert werden muss usw. An dieser Stelle ist man vollkommen frei von solchen Gedanken, die funktionale Programmierung befreit von solchem Ballast.

### 10.3 Binärbaum \*

Oft ist man in der Informatik an Objekten und den Zusammenhängen zwischen den Objekten interessiert. Die schematische Darstellung eines Straßenbahnnetzes ist ein Beispiel dafür. Dort gibt es werden Haltestellen durch Kreise dargestellt. Manche Kreise sind durch Linien verbunden, was einer direkten Gleisverbindung der beiden Haltestellen entspricht. Abbildung 10.1 zeigt ein Beispiel eines solchen Bahnnetzes.

Solche Zusammenhänge werden in der Informatik (und in anderen Disziplinen) durch Graphen dargestellt, die aus Knoten, die unseren Kreisen im Straßenbahnnetz entsprechen, und Kanten, die den Verbindungen im Beispiel entsprechen, bestehen. Graphen sind wichtige Hilfsmittel zur Repräsentation von Zusammenhängen sowie deren Informationsverarbeitung. Eine mögliche Fragestellung ist zum Beispiel, ob es überhaupt möglich ist, von Station 1 nach Station 5 zu gelangen oder welche Stationen ich von Station 1 aus alle erreichen kann. Eine ganze Teildisziplin der Informatik/Mathematik, die Graphentheorie, beschäftigt sich mit dieser Thematik. Im Folgenden findet eine Beschränkung auf eine Unterklasse von Graphen statt, die eine ganz bestimmte Struktur haben. Diese speziellen Graphen, werden Binäräume genannt, haben u.a. die Eigenschaft, dass ein Knoten maximal mit zwei anderen Knoten verbunden ist.

#### Definition 10.4 (Binärbaum):

Sei  $M$  eine Menge. Die Menge  $M^\Delta$  der **Binäräume** über  $M$  ist induktiv definiert durch:

- 1)  $\varepsilon \in M^\Delta$  (leerer Binärbaum)
- 2)  $a \in M, x, y \in M^\Delta \Rightarrow < x, a, y > \in M^\Delta$ .  $x$  heißt in diesem Fall linker **Teilbaum**,  $a$  **Wurzel** und  $y$  rechter Teilbaum des Binärbaumes.

Ein Knoten ohne Teilbäume (d.h. ein Knoten der Form  $< \varepsilon, a, \varepsilon >$ ) heißt **Blatt**. Wurzeln, auch von Teilbäumen, nennt man **Knoten**.



#### Beispiel 10.14:

Sei  $M = \{a, b, c, d\}$ . Dann sind folgende Binäräume in  $M^\Delta$ :

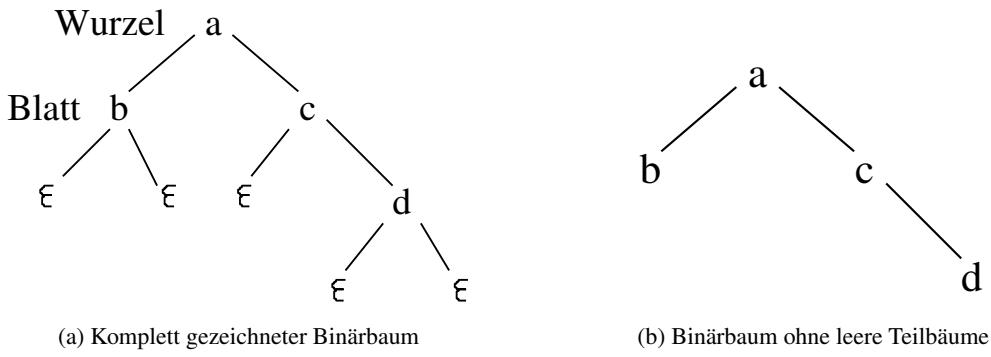


Figure 10.2: Grafische Darstellung von Binärbäumen

- 1)  $\epsilon$
- 2)  $\langle \epsilon, a, \epsilon \rangle$
- 3)  $\langle \langle \epsilon, b, \epsilon \rangle, a, \epsilon \rangle$
- 4)  $\langle \epsilon, a, \langle \epsilon, b, \epsilon \rangle \rangle$
- 5)  $\langle \langle \epsilon, b, \epsilon \rangle, b, \langle \epsilon, b, \epsilon \rangle \rangle$
- 6)  $\langle \langle \epsilon, b, \epsilon \rangle, a, \langle \epsilon, c, \langle \epsilon, d, \epsilon \rangle \rangle \rangle$
- 7)  $\langle \langle \epsilon, b, \langle \epsilon, b, \epsilon \rangle \rangle, a, \langle \epsilon, c, \langle \epsilon, d, \epsilon \rangle \rangle \rangle$

◆

Man kann sich Binärbäume anschaulich in grafischer Form als baumartiges Gebilde vorstellen (daher der Name), mit der Wurzel nach oben zeigend und nach unten sich zu den Blättern verzweigend. Abbildung 10.2 zeigt in grafischer Form den Binärbaum  $\langle \langle \epsilon, b, \epsilon \rangle a \langle \epsilon, c, \langle \epsilon, d, \epsilon \rangle \rangle \rangle$ . Dabei wird die Wurzel des Baumes oben gezeichnet und rechts und links jeweils der rechte beziehungsweise der linke Teilbaum eingezeichnet. Abbildung 10.2a zeigt den vollständigen Binärbaum. Üblicherweise lässt man aber der Übersichtlichkeit wegen leere Teilbäume weg, wie in Abbildung 10.2b zu sehen ist.

Betrachtet werden soll nun der Datentyp

$$\text{binbaum} = \langle \{M, M^\Delta, \mathbb{B}\}, \text{Ops}^{\text{binbaum}} \rangle,$$

wobei die Operationen

$$\text{Ops}^{\text{binbaum}} = \{\text{emptytree}, \text{maketree}, \text{root}, \text{lefttree}, \text{righttree}, \text{isemptytree}, \text{isequal}\}$$

wie folgt definiert sind:

1.  $\text{emptytree} : \emptyset \rightarrow M^\Delta$  mit:

$$\text{emptytree}() = \epsilon$$

Erzeugt einen leeren Binärbaum.

2.  $\text{maketree} : M^\Delta \times M \times M^\Delta \rightarrow M^\Delta$  mit:

$$\text{maketree}(l, a, r) = \langle l, a, r \rangle$$

Erzeugt eine Binärbaum mit Wurzel  $a$  und linkem und rechten Teilbaum  $l$  beziehungsweise  $r$ .

3.  $\text{root} : M^\Delta \rightarrow M$  mit:

$$\text{root}(t) = \begin{cases} \perp & \text{falls } t = \epsilon \\ a & \text{falls } t = \langle l, a, r \rangle \end{cases}$$

Liefert die Wurzel eines Binärbaumes  $t = \langle l, a, r \rangle$ .

4.  $\text{lefttree} : M^\Delta \rightarrow M^\Delta$  mit:

$$\text{lefttree}(t) = \begin{cases} \perp & \text{falls } t = \epsilon \\ l & \text{falls } t = \langle l, a, r \rangle \end{cases}$$

Liefert den linken Teilbaum eines Binärbaumes  $t = \langle l, a, r \rangle$ .

5.  $\text{righttree} : M^\Delta \rightarrow M^\Delta$  mit:

$$\text{righttree}(t) = \begin{cases} \perp & \text{falls } t = \epsilon \\ r & \text{falls } t = \langle l, a, r \rangle \end{cases}$$

Liefert den rechten Teilbaum eines Binärbaumes  $t = \langle l, a, r \rangle$ .

6.  $\text{isemptytree} : M^\Delta \rightarrow \mathbb{B}$  mit:

$$\text{isemptytree}(t) = \begin{cases} \text{wahr} & \text{falls } t = \epsilon \\ \text{falsch} & \text{sonst} \end{cases}$$

Testet einen Binärbaum auf den leeren Binärbaum.

7.  $\text{isequal} : M^\Delta \times M^\Delta \rightarrow \mathbb{B}$  mit:

$$\text{isequal}(t_1, t_2) = \begin{cases} \text{wahr} & \text{falls } \text{isemptytree}(t_1) \wedge \text{isemptytree}(t_2) \\ \text{falsch} & \text{falls } \text{isemptytree}(t_i) \wedge \neg \text{isemptytree}(t_j) \\ & (i, j \in \{1, 2\}, i \neq j) \\ (\text{root}(t_1) = \text{root}(t_2)) \wedge & \\ (\text{isequal}(\text{lefttree}(t_1), \text{lefttree}(t_2)) \wedge & \\ \text{isequal}(\text{righttree}(t_1), \text{righttree}(t_2))) & \text{sonst} \end{cases}$$

Testet, ob zwei Binärbäume (komponentenweise) gleich sind.

Im Folgenden wird der Übersichtlichkeit halber ein Blatt eines Binärbaums  $\langle \epsilon, a, \epsilon \rangle$  kurz als  $\langle a \rangle$  notiert. Auch hier spiegeln die Auswahl und Art der Operationen die Struktur eines Binärbaums wider: ein Binärbaum besteht aus einer Wurzel und einem linken und rechten Teilbaum, die wieder Binärbäume sind.

**Anwendung 1:** Der Binärbaum aus Abbildung 10.2a lässt sich konstruieren durch Komposition aus Grundfunktionen. Sei:

$$\begin{aligned} bbaum &:= \text{maketree}(\epsilon, b, \epsilon) \\ dbaum &:= \text{maketree}(\epsilon, d, \epsilon) \end{aligned}$$

Dann lässt sich dieser Binärbaum erzeugen durch:

$$\text{maketree}(bb Baum, a, \text{maketree}(\epsilon, c, db Baum))$$

Bemerkung: Die Bezeichnung von Zwischenberechnungen ( $bb Baum$ ,  $db Baum$ ) werden bei der Betrachtung von Programmiersprachen wieder zu finden sein. Dort übernehmen Variablen die Rolle der Zwischenspeicherung.

**Anwendung 2:** Der Test, ob ein Wert aus der Grundmenge  $M$  im Binärbaum enthalten ist, lässt sich wie folgt aus den Grundfunktionen erzeugen.  $\text{contains} : M^\Delta \times M \rightarrow \mathbb{B}$  mit:

$$\text{contains}(t, a) = \begin{cases} \text{falsch} & \text{falls } t = \varepsilon \\ \text{wahr} & \text{falls } a = \text{root}(t) \\ \text{contains}(\text{lefttree}(t), a) \vee \text{contains}(\text{righttree}(t), a) & \text{sonst} \end{cases}$$

**Beispiel 10.15:**

$$\begin{aligned} \text{contains}(< a >, b, << c >, d, \varepsilon >, d) \\ = \text{contains}(< a >, d) \vee \text{contains}(<< c >, d, \varepsilon >, d) \\ = (\text{contains}(\varepsilon, d) \vee \text{contains}(\varepsilon, d)) \vee \text{wahr} \end{aligned}$$

An dieser Stelle kann man abbrechen, weil  $(a \vee b) \vee \text{wahr}$  wahr ist, unabhängig von  $a$  und  $b$ . (Dies gilt jedoch nur, wenn wie in diesem Fall die Berechnung von  $a$  und  $b$  abbricht!)  $\blacklozenge$

**Anwendung 3:** Zu jedem Knoten kann man die **Tiefe** (oder den **Level**) des Knotens angeben. Die Tiefe des Wurzelknotens ist 0. Für jeden Knoten ungleich der Wurzel ist seine Tiefe gleich der Tiefe des Vaterknotens plus 1. Die Höhe eines Binärbaumes ist dann definiert als die maximale Tiefe eines Knotens des Baumes, wobei der leere Baum definitionsgemäß die Höhe 0 haben soll.  $\text{baumhoehe} : M^\Delta \rightarrow \mathbb{N}$  mit:

$$\text{baumhoehe}(t) = \begin{cases} 0 & \text{falls } t = \varepsilon \text{ oder } t = < a > \text{ mit } a \in M \\ 1 + \max(\text{baumhoehe}(\text{lefttree}(t)), \text{baumhoehe}(\text{righttree}(t))) & \text{sonst} \end{cases}$$

**Beispiel 10.16:**

$$\begin{aligned} \text{baumhoehe}(< a >, b, << c >, d, \varepsilon >) \\ = 1 + \max(\text{baumhoehe}(< a >), \text{baumhoehe}(<< c >, d, \varepsilon >)) \\ = 1 + \max(0, (1 + \max(\text{baumhoehe}(< c >), \text{baumhoehe}(\varepsilon)))) \\ = 1 + \max(1, 1 + \max(0, 0)) \\ = 1 + \max(1, 1) \\ = 2 \end{aligned}$$

**Anwendung 4:** Die Anzahl der Knoten in einem Binärbaum lässt sich mit der folgenden Funktion ermitteln.  $\text{anzahlknoten} : M^\Delta \rightarrow \mathbb{N}$  mit:

$$\text{anzahlknoten}(t) = \begin{cases} 0 & \text{falls } t = \varepsilon \\ 1 + \text{anzahlknoten}(\text{lefttree}(t)) + \text{anzahlknoten}(\text{righttree}(t)) & \text{sonst} \end{cases}$$

**Beispiel 10.17:**

$$\text{anzahlknoten}(< a >, b, << c >, d, \varepsilon >)$$

$$\begin{aligned}
&= 1 + \text{anzahlknoten}(< a >) + \text{anzahlknoten}(<< c >, d, \varepsilon >) \\
&= 1 + (1 + \text{anzahlknoten}(\varepsilon) + \text{anzahlknoten}(\varepsilon)) \\
&\quad + (1 + \text{anzahlknoten}(< c >) + \text{anzahlknoten}(\varepsilon)) \\
&= 1 + (1 + 0 + 0) + (1 + (1 + \text{anzahlknoten}(\varepsilon) + \text{anzahlknoten}(\varepsilon)) + 0) \\
&= 1 + 1 + (1 + (1 + 0 + 0) + 0) \\
&= 4
\end{aligned}$$

◆

Die Funktion *anzahlknoten* im obigen Beispiel zeigt eine Anwendung, in der der gesamte Binärbaum durchlaufen werden muss. Zur Bestimmung der Gesamtzahl der Knoten muss – aufgrund der rekursiven Struktur von Binärbäumen – jeder Knoten (beziehungsweise Teilbaum) aufgesucht werden. Das Aufsuchen aller Knoten einer (Binär-) Baumes ist eine oft angewandte Technik. Aufgrund der Kommutativität ( $a + b = b + a$ ) und Assoziativität ( $((a + b) + c = a + (b + c))$ ) des + Operators ist die Reihenfolge des Aufsuchens der Knoten im Beispiel *anzahlknoten* unerheblich. In anderen Fällen ist sie jedoch von Relevanz.

### 10.3.1 Arithmetische Ausdrücke

Sei  $M = \mathbb{Z} \cup \{(,), +, -, *, /\}$ . Betrachtet werden die **arithmetischen Ausdrücke**  $\text{expr}_M$  über  $M$ , die wie folgt definiert sind:

- 1)  $x \in \text{expr}_M$  für  $x \in \mathbb{Z}$
- 2) Für  $x, x_1, x_2 \in \text{expr}_M$  gilt:
  - $(x) \in \text{expr}_M$
  - $x_1 + x_2 \in \text{expr}_M$
  - $x_1 - x_2 \in \text{expr}_M$
  - $x_1 * x_2 \in \text{expr}_M$
  - $x_1 / x_2 \in \text{expr}_M$

D.h. arithmetische Ausdrücke sind entweder einfache Ausdrücke bestehend aus einer Konstanten (eine ganze Zahl) oder komplexe arithmetische Ausdrücke, die mit Hilfe von Operationssymbolen aus arithmetischen Ausdrücken zusammengesetzt werden können.

Bemerkungen:

- Durch die rekursive Definition lassen sich beliebig komplexe Ausdrücke erzeugen.
- Es lassen sich nur wohlgeformte Ausdrücke erzeugen. Man kann mit der obigen Definition zum Beispiel nicht erzeugen:  $(3+4$ ).
- Die Operationssymbole sind maximal zweistellig, d.h. benötigen maximal zwei Operanden. Für ein Operationssymbol ist die Stelligkeit eindeutig festgelegt.
- Konstanten kann man auch als nullstellige Operationsymbole auffassen. Beispiel: Die Konstante 3 könnte man auch als Funktion  $drei : \emptyset \rightarrow \mathbb{Z}$  mit  $drei() := 3$  angeben.

#### Beispiel 10.18:

- $5 \in \text{expr}_M$
  - $3 + 4 \in \text{expr}_M$
  - $3 + (4 * 5) \in \text{expr}_M$
  - $3 + 4 * 5 \in \text{expr}_M$
- ◆

Durch die Beschränkung der Stelligkeit auf maximal 2 und die feste Stelligkeit der Operationssymbole lassen sich arithmetische Ausdrücke auf einfache Weise durch Binärbäume darstellen. Knoten eines Binärbaums

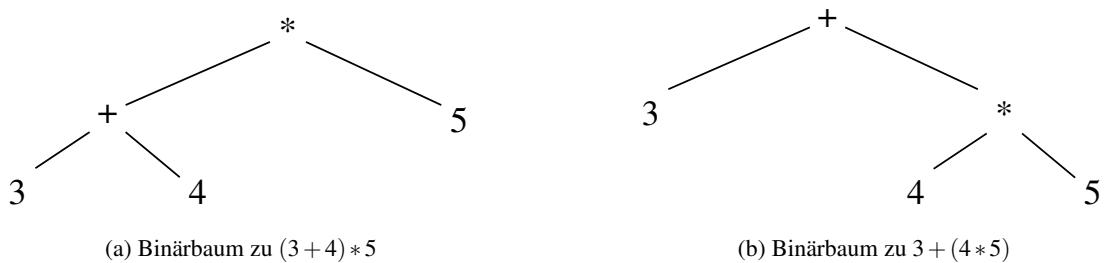


Figure 10.3: Repräsentation von arithmetischen Ausdrücken durch Binärbäume

entsprechen den Konstanten beziehungsweise den Operationsymbolen. Argumente der Operationssymbole sind die Söhne des Knotens, der das Operationsymbol repräsentiert. Abbildung 10.3 zeigt die Binärbäume zu zwei arithmetischen Ausdrücken. Zur besseren Lesbarkeit des Baumes sollte man Operationssymbole nicht als Infixoperation (Beispiel:  $a + b$ ) ansehen, sondern als Präfixoperation (Beispiel:  $+(a, b)$ ).

Arithmetische Ausdrücke lassen sich mit Binärbäumen darstellen und die Transformation eines arithmetischen Ausdrucks in einen Binärbaum kann automatisch mit Hilfe eines Algorithmus erfolgen. Dazu definiert man die Funktion  $\text{trans} : \text{expr}_M \rightarrow M^\Delta$  wie folgt:

$$\text{trans}(e) = \begin{cases} < \varepsilon, x, \varepsilon > & \text{falls } e = x \text{ und } x \in \mathbb{Z} \\ \text{trans}(e_1) & \text{falls } e = (e_1) \text{ und } e_1 \in \text{expr}_M \\ < \text{trans}(e_1), +, \text{trans}(e_2) > & \text{falls } e = e_1 + e_2 \text{ und } e_1, e_2 \in \text{expr}_M \\ < \text{trans}(e_1), -, \text{trans}(e_2) > & \text{falls } e = e_1 - e_2 \text{ und } e_1, e_2 \in \text{expr}_M \\ < \text{trans}(e_1), *, \text{trans}(e_2) > & \text{falls } e = e_1 * e_2 \text{ und } e_1, e_2 \in \text{expr}_M \\ < \text{trans}(e_1), /, \text{trans}(e_2) > & \text{falls } e = e_1 / e_2 \text{ und } e_1, e_2 \in \text{expr}_M \end{cases}$$

Auch hier sei wieder angemerkt, dass die Definition der Funktion wiederum der Struktur der Daten folgt.

### Beispiel 10.19:

1)  $e = (3 + 4) * 5 \in \text{expr}_M$ . Dann ist:

$$\begin{aligned} \text{trans}( (3 + 4) * 5 ) &= < \text{trans}( (3 + 4) ), *, \text{trans}(5) > \\ &= < \text{trans}(3 + 4), *, < 5 > > \\ &= << \text{trans}(3), +, \text{trans}(4) >, *, < 5 > > \\ &= <<< 3 >, +, < 4 > >, *, < 5 > > \end{aligned}$$

2)  $e = 3 + (4 * 5) \in \text{expr}_M$ . Dann ist:

$$\begin{aligned} \text{trans}( 3 + (4 * 5) ) &= < \text{trans}(3), +, \text{trans}( (4 * 5) ) > \\ &= << 3 >, +, \text{trans}(4 * 5) > > \\ &= << 3 >, +, < \text{trans}(4), *, \text{trans}(5) > > \\ &= << 3 >, +, << 4 >, *, < 5 > > > \end{aligned}$$

Abbildung 10.3 zeigt die beiden Binärbäume in grafischer Form. Wie man sieht, spiegelt sich die unterschiedliche Klammerung in einer anderen Struktur im entsprechenden Binärbaum wider. ♦

Bemerkung: Die eben beschriebene Funktion *trans* ist ein "‘Mini-Übersetzer’", der eine Programmiersprache (arithmetische Ausrücke) in eine andere Sprache oder Darstellung (Binärbäume) übersetzt. In Kapitel 3.3.2 wurde bereits ein Übersetzer für die Übersetzung eines Programmes in die Maschinensprache eines Rechners borgestellt. Dieser Übersetzer arbeitet von der Funktionsweise her ähnlich wie dieser Mini-Übersetzer hier.

Man kann ebenfalls eine Funktion angeben, die beliebige arithmetische Ausdrücke auswertet:  $\text{trans} : \text{expr}_M \rightarrow \mathbb{Z}$  mit:

$$\text{eval}(e) = \begin{cases} x & \text{falls } e = x \text{ und } x \in \mathbb{Z} \\ \text{eval}(e_1) & \text{falls } e = (e_1) \text{ und } e_1 \in \text{expr}_M \\ \text{eval}(e_1) + \text{eval}(e_2) & \text{falls } e = e_1 + e_2 \text{ und } e_1, e_2 \in \text{expr}_M \\ \text{eval}(e_1) - \text{eval}(e_2) & \text{falls } e = e_1 - e_2 \text{ und } e_1, e_2 \in \text{expr}_M \\ \text{eval}(e_1) * \text{eval}(e_2) & \text{falls } e = e_1 * e_2 \text{ und } e_1, e_2 \in \text{expr}_M \\ \text{eval}(e_1) / \text{eval}(e_2) & \text{falls } e = e_1 / e_2 \text{ und } e_1, e_2 \in \text{expr}_M \end{cases}$$

Um im Zahlenkörper der ganzen Zahlen zu bleiben, muss man die Division  $/$  als ganzzahlige Division definieren. Dabei muss unterschieden werden zwischen dem Operationssymbol  $+ \in M$  und dem Symbol  $+$  (zum Beispiel bei  $\text{eval}(e_1) + \text{eval}(e_2)$ ), das für die mathematische Operation Addition steht.

### Beispiel 10.20:

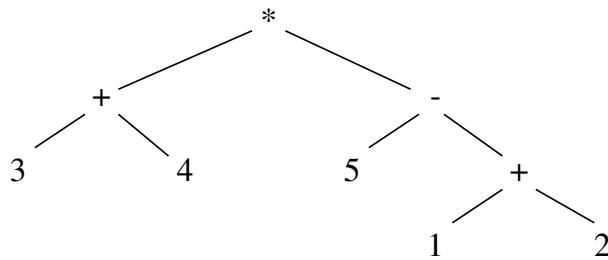
1)  $e = (3 + 4) * 5 \in \text{expr}_M$ . Dann ist:

$$\begin{aligned} \text{eval}( (3 + 4) * 5 ) &= \text{eval}( (3 + 4) ) * \text{eval}(5) \\ &= \text{eval}(3 + 4) * 5 \\ &= (\text{eval}(3) + \text{eval}(4)) * 5 \\ &= (3 + 4) * 5 \\ &= 7 * 5 \\ &= 35 \end{aligned}$$

2)  $e = 3 + (4 * 5) \in \text{expr}_M$ . Dann ist:

$$\begin{aligned} \text{eval}( 3 + (4 * 5) ) &= \text{eval}(3) + \text{eval}( (4 * 5) ) \\ &= 3 + \text{eval}(4 * 5) \\ &= 3 + (\text{eval}(4) * \text{eval}(5)) \\ &= 3 + (4 * 5) \\ &= 3 + 20 \\ &= 23 \end{aligned}$$

Es sollen nun Funktionen definiert werden, die aus einem Binärbaum eine Folge erzeugen. Bei unseren Zeichnungen von Binärbäumen waren dies stets zweidimensionale Gebilde, Folgen hingegen eindimensionale. Wir "‘plätzen’" also gewissermassen einen zweidimensionalen Binärbaum zu einer eindimensionalen Folge, ohne dass Informationen zum Inhalt oder der Struktur des Baumes verloren gehen sollen. Dazu bedienen man sich der bereits angesprochenen Technik des Durchlaufens durch einen Binärbaum. Dabei werden drei verschiedene Durchlaufstrategien diskutiert, die jeweils in einer Funktion angegeben sind. Die Funktionen

Figure 10.4: Binärbaum zum arithmetischen Ausdruck ( $<3> + <4>$ ) \* ( $<5> - (<1> + <2>)$ )

*preorder, inorder, postorder :  $M^\Delta \rightarrow M^*$*  sind definiert durch:

$$\text{preorder}(t) = \begin{cases} \varepsilon & \text{falls } t = \varepsilon \\ \text{makefolge}(a) \circ \text{preorder}(l) \circ \text{preorder}(r) & \text{falls } t = <l, a, r> \end{cases}$$

$$\text{inorder}(t) = \begin{cases} \varepsilon & \text{falls } t = \varepsilon \\ \text{inorder}(l) \circ \text{makefolge}(a) \circ \text{inorder}(r) & \text{falls } t = <l, a, r> \end{cases}$$

$$\text{postorder}(t) = \begin{cases} \varepsilon & \text{falls } t = \varepsilon \\ \text{postorder}(l) \circ \text{postorder}(r) \circ \text{makefolge}(a) & \text{falls } t = <l, a, r> \end{cases}$$

Zur Erinnerung: *makefolge(a)* erzeugt eine Folge, deren einziges Element *a* ist. Der Operator  $\circ$  fügt zwei Folgen zu einer Folge zusammen (Konkatenation).

Prinzipiell fallen bei allen Strategien drei Aufgaben an, die je nach Strategie in unterschiedlicher Reihenfolge bearbeitet werden:

- 1) Bearbeite die Wurzel
- 2) Bearbeite (rekursiv) linken Teilbaum
- 3) Bearbeite (rekursiv) rechten Teilbaum

Für die einzelnen Strategien gilt folgende Reihenfolge:

- 1) **Preorder:** Bearbeite Wurzel, bearbeite linken Teilbaum, bearbeite rechten Teilbaum
- 2) **Inorder:** bearbeite linken Teilbaum, bearbeite Wurzel, bearbeite rechten Teilbaum
- 3) **Postorder:** bearbeite linken Teilbaum, bearbeite rechten Teilbaum, bearbeite Wurzel

### Beispiel 10.21:

Gegeben sei der Binärbaum

$\text{trans}((3+4)*(5-(1+2))) = <<<3>, +, <4>>, *, <<5>, -, <<1>, +, <2>>>$

wie er in grafischer Form auch in Abbildung 10.4 zu sehen ist. Für die einzelnen Strategien ergeben sich (Trennkommas in den resultierenden Folgen wurde der Übersichtlichkeit wegen weggelassen):

1)  $\text{preorder}(<<< 3 >, +, < 4 >>, *, << 5 >, -, << 1 >, +, < 2 >>>)$

$$\begin{aligned}
 &= \text{makefolge}(*) \circ \text{preorder}(<< 3 >, +, < 4 >>) \circ \\
 &\quad \text{preorder}(<< 5 >, -, << 1 >, +, < 2 >>) \\
 &= < * > \circ (\text{makefolge}(+) \circ \text{preorder}(< 3 >) \circ \text{preorder}(< 4 >)) \\
 &\quad \circ (\text{makefolge}(-) \circ \text{preorder}(< 5 >) \circ \text{preorder}(<< 1 >, +, < 2 >>)) \\
 &= < * > \circ (< + > \circ (< 3 > \circ \varepsilon \circ \varepsilon) \circ (< 4 > \circ \varepsilon \circ \varepsilon)) \\
 &\quad \circ (< - > \circ (< 5 > \circ \varepsilon \circ \varepsilon)) \circ (\text{makefolge}(+) \circ \text{preorder}(< 1 >) \circ \text{preorder}(< 2 >)) \\
 &= < * + 3 4 - 5 + > \circ < 1 > \circ \varepsilon \circ \varepsilon \circ < 2 > \circ \varepsilon \circ \varepsilon)) \\
 &= < * + 3 4 - 5 + 1 2 >
 \end{aligned}$$

2)  $\text{inorder}(<<< 3 >, +, < 4 >>, *, << 5 >, -, << 1 >, +, < 2 >>>)$

$$\begin{aligned}
 &= \text{inorder}(<< 3 >, +, < 4 >>) \circ \text{makefolge}(*) \circ \\
 &\quad \text{inorder}(<< 5 >, -, << 1 >, +, < 2 >>) \\
 &= \text{inorder}(< 3 >) \circ \text{makefolge}(+) \circ \text{inorder}(< 4 >) \circ < * > \circ \\
 &\quad \text{inorder}(< 5 >) \circ \text{makefolge}(-) \circ \text{inorder}(< 1 >, +, < 2 >) \\
 &= (\varepsilon \circ < 3 > \circ \varepsilon) \circ < + > \circ (\varepsilon \circ < 4 > \circ \text{epsilon}) \circ < * > \circ \\
 &\quad (\varepsilon \circ < 5 > \circ \varepsilon) \circ < - > \circ \text{inorder}(< 1 >) \circ \text{makefolge}(+) \circ \text{inorder}(< 2 >) \\
 &= < 3 + 4 * 5 - > \circ (\varepsilon \circ < 1 > \circ \varepsilon) \circ < + > \circ (\varepsilon \circ < 2 > \circ \varepsilon) \\
 &= < 3 + 4 * 5 - 1 + 2 >
 \end{aligned}$$

3)  $\text{postorder}(<<< 3 >, +, < 4 >>, *, << 5 >, -, << 1 >, +, < 2 >>>)$

$$\begin{aligned}
 &= \text{postorder}(<< 3 >, +, < 4 >>) \circ \\
 &\quad \text{postorder}(<< 5 >, -, << 1 >, +, < 2 >>) \circ \text{makefolge}(*) \\
 &= (\text{postorder}(< 3 >) \circ \text{postorder}(< 4 >) \circ \text{makefolge}(+)) \circ \\
 &\quad (\text{postorder}(< 5 >) \circ \text{postorder}(< 1 >, +, < 2 >) \circ \text{makefolge}(-)) \circ \text{makefolge}(*) \\
 &= (\varepsilon \circ \varepsilon \circ < 3 >) \circ (\varepsilon \circ \varepsilon \circ < 4 >) \circ < + > \circ \\
 &\quad ((\varepsilon \circ \varepsilon \circ < 5 >) \circ (\text{postorder}(< 1 >) \circ \text{postorder}(< 2 >) \circ \\
 &\quad \text{makefolge}(+))) \circ < - > \circ < * > \\
 &= < 3 4 + 5 > \circ (\varepsilon \circ \varepsilon \circ \text{makefolge}(1)) \circ (\varepsilon \circ \varepsilon \circ \text{makefolge}(2)) \circ < + - * > \\
 &= < 3 4 + 5 1 2 + - * >
 \end{aligned}$$

Am Beispiel kann man sehen, wie die drei Strategien zur Bearbeitungsreihenfolge (Knoten, linker Teilbaum, rechter Teilbaum) sehr unterschiedliche Resultate liefern können. Bezogen auf die Umwandlung von Binärbäumen, die arithmetische Ausdrücke darstellen, zu Wörtern mit Hilfe der Funktionen *preorder*, *inorder*, *postorder* lässt sich Folgendes sagen:

- Durch *inorder* wird die uns gewohnte **Infixnotation** erzeugt, in der ein binärer (2-stelliger) Operator zwischen den Operanden notiert wird. Allerdings werden bei der automatischen Umwandlung von

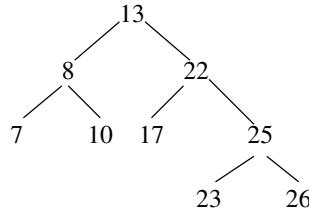


Figure 10.5: Beispiel eines binären Suchbaums

Binärbäumen zu Wörtern keine Prioritäten von Operatoren beachtet. Eine Klammerung wäre zusätzlich nötig, fehlt aber in der automatischen Transformation.

- Durch *preorder* wird die Präfixnotation, durch *postorder* die **Postfixnotation** erzeugt. In einer Präfixnotation werden die Operationsymbole vor den Operanden notiert, in einer Postfixnotation nach den Operanden. Beispiel: Der Infixausdruck  $3 + 4$  wird in Präfixschreibweise notiert als  $+34$  und in Postfixschreibweise  $34+$ .
- Präfix- und Postfixnotationen benötigen keine Klammerung, durch die Reihenfolge der Operanden und Operatoren ist dies implizit gegeben.
- Die Postfixnotation ist auch als **UPN (Umgekehrt Polnische Notation)** bekannt.
- Über die Postfixnotation ist ein einfaches Auswerten von arithmetischen Ausdrücken mit Hilfe eines Stacks (s.u.) möglich.

### 10.3.2 Suchbäume

#### Definition 10.5 (Suchbaum):

Sei  $M$  eine total geordnete Menge.  $t \in M^\Delta$  heißt (binärer) Suchbaum über  $M$ , falls für  $t$  gilt:

- 1) All Knoten im linken Teilbaum von  $t$  sind kleiner als die Wurzel von  $t$ .
- 2) Alle Knoten im rechten Teilbaum von  $t$  sind größer als die Wurzel von  $t$ .
- 3) Der linke und rechte Teilbaum von  $t$  sind jeweils Suchbäume.

◆

In Abbildung 10.5 ist ein Beispiel eines binären Suchbaums über  $\mathbb{N}$  gegeben. Für einen Binärbaum  $t \in M^\Delta$  lässt sich über die Funktion  $istSuchbaum : M^\Delta \rightarrow \mathbb{B}$  entscheiden, ob der Binärbaum auch ein binärer Suchbaum ist. Zur Entwicklung dieser Funktion nutzt man die in der Definition angegebenen Bedingungen aus. Zuerst benötigt man zwei Hilfsfunktionen, die das Minimum beziehungsweise Maximum in einem Binärbaum bestimmen.

$$\min, \max : M^\Delta \rightarrow M$$

mit

$$\min(t) = \begin{cases} \perp & \text{falls } t = \varepsilon \\ \text{root}(t) & \text{falls } lefttree(t) = righttree(f) = \varepsilon \\ \min(\min(lefttree(t)), \text{root}(t)) & \text{falls } lefttree(t) \neq \varepsilon \wedge righttree(t) = \varepsilon \\ \min(\text{root}(t), \min(righttree(t))) & \text{falls } lefttree(t) = \varepsilon \wedge righttree(t) \neq \varepsilon \\ \min(\min(lefttree(t)), \\ \quad \text{root}(t), \\ \quad \min(righttree(t))) & \text{sonst} \end{cases}$$

Die Funktion  $\max$  wird analog definiert.  $\min, \max : \mathbb{N}^n \rightarrow \mathbb{N}$  geben für ein festes  $n$  Zahlen den kleinsten beziehungsweise größten dieser Werte an.

Bemerkung: Besitzt die Menge  $M$  ein maximales beziehungsweise minimales Element, so lässt sich die Funktionsdefinition kürzer angeben. Wie?

Mit diesen Hilfsfunktionen lässt sich nun die Funktion  $\text{istSuchbaum}$  angeben:

$$\text{istSuchbaum}(t) = \begin{cases} \text{wahr} & \text{falls } t = \epsilon \\ \text{wahr} & \text{falls } \text{isempty}(lefttree(t)) \\ & \wedge \text{isempty}(righttree(t)) \\ \text{wahr} & \text{falls } \text{isempty}(righttree(t)) \\ & \wedge \neg \text{isempty}(lefttree(t)) \\ & \wedge \text{istSuchbaum}(lefttree(t)) \\ & \wedge (\min(lefttree(t)) < \text{root}(t)) \\ \text{wahr} & \text{falls } \text{isempty}(lefttree(t)) \\ & \wedge \neg \text{isempty}(righttree(t)) \\ & \wedge \text{istSuchbaum}(righttree(t)) \\ & \wedge (\max(righttree(t)) > \text{root}(t)) \\ \text{wahr} & \text{falls } \neg \text{isempty}(lefttree(t)) \wedge \neg \text{isempty}(righttree(t)) \\ & \wedge \text{istSuchbaum}(lefttree(t)) \wedge \text{istSuchbaum}(righttree(t)) \\ & (\min(lefttree(t)) < \text{root}(t)) \wedge (\max(righttree(t)) > \text{root}(t)) \\ \text{falsch} & \text{sonst} \end{cases}$$

Enthält die Menge  $M$  ein kleines und größtes Element  $(-\infty, +\infty)$ , so lassen sich die Funktionen einfacher angeben:

$$\min, \max : M^\Delta \rightarrow M$$

mit

$$\min(t) = \begin{cases} +\infty & \text{falls } t = \epsilon \\ \min(\min(lefttree(t)), \text{root}(t), \min(righttree(t))) & \text{sonst} \end{cases}$$

$\max$  analog.

Die Funktion  $\text{istSuchbaum}$  lässt sich dann ebenfalls einfacher angeben und spiegelt direkt die Bedingungen der Definition wider:

$$\text{istSuchbaum}(t) = \begin{cases} \text{wahr} & \text{falls } \text{isempty}(t) \\ \text{falsch} & \text{falls } \min(lefttree(t)) > \text{root}(t) \vee \max(righttree(t)) < \text{root}(t) \\ \text{istSuchbaum}(lefttree(t)) \wedge \text{istSuchbaum}(righttree(t)) & \text{sonst} \end{cases}$$

Bezeichne  $M_{<}^\Delta := \{t \in M^\Delta \mid \text{istSuchbaum}(t)\}$  die Menge der binären Suchbäume über  $M$ . Um in einem Suchbaum einen vorgegebenen Wert zu finden, kann man die Funktion  $\text{binSuchen} : M_{<}^\Delta \times M \rightarrow \mathbb{B}$  anwenden:

$$\text{binSuchen}(t, a) = \begin{cases} \text{falsch} & \text{falls } t = \epsilon \\ \text{wahr} & \text{falls } a = \text{root}(t) \\ \text{binSuchen}(lefttree(t), a) & \text{falls } a < \text{root}(t) \\ \text{binSuchen}(righttree(t), a) & \text{falls } a > \text{root}(t) \end{cases}$$

Für Suchbäume gilt:  $\text{inorder}(t)$  ist aufsteigend sortiert für alle  $t \in M_{<}^\Delta$ , d.h.  $\forall t \in M_{<}^\Delta : \text{inorder}(t) \in M_{<}^*$ .

## 10.4 Kellerspeicher

Ein Kellerspeicher (oder Stapel oder Stack) ist ein Datenspeicher, auf den man Daten ablegen kann und jeweils den obersten Eintrag, d.h. die zuletzt abgelegten Daten wieder entfernen kann oder lesend darauf zugreifen kann. Ein gutes Beispiel aus dem alltäglichen Leben ist ein Tablettsender in einer Mensa, wo oben auf den Stapel saubere Tablets abgelegt werden und man immer nur das oberste Tabletts des Stacks entnehmen kann. Der Datentyp  $stack = < \{M, M^*, \mathbb{B}\}, Ops^{stack} >$  wird über einer Menge  $M$  definiert, dem Grunddatentyp für unseren Stack.  $M$  kann beliebig sein. Zum Beispiel mit  $M = \mathbb{N}$  erhält man einen Stack von natürlichen Zahlen, mit  $M = \{rot, gelb, blau\}$  erhält man einen Stack, auf dem Daten abgelegt werden können, die den Wert *rot*, *gelb* oder *blau* haben können. Ein Stack hat folgende Operationen  $Ops^{stack}$ :

1.  $emptystack : \emptyset \rightarrow M^*$  mit:

$$emptystack() = <>$$

Erzeugt einen leeren Stack, d.h. einen Stack ohne Elemente.

2.  $isemptystack : M^* \rightarrow \mathbb{B}$  mit:

$$isemptystack(s) = \begin{cases} \text{wahr} & \text{falls } s = <> \\ \text{falsch} & \text{sonst} \end{cases}$$

Testet, ob ein Stack leer ist.

3.  $push : M^* \times M \rightarrow M^*$  mit:

$$push(s, x) = < x, x_1, \dots, x_n > \text{ falls } s = < x_1, \dots, x_n >, n \geq 0$$

Fügt das Element  $x$  vorne an den Stack an.

4.  $top : M^* - \rightarrow M$  mit:

$$top(s) = \begin{cases} x_1 & \text{falls } s = < x_1, \dots, x_n >, n > 0 \\ \perp & \text{sonst} \end{cases}$$

Liefert das erste Element des Stacks.

5.  $pop : M^* - \rightarrow M^*$  mit:

$$pop(s) = \begin{cases} < x_2, \dots, x_n > & \text{falls } s = < x_1, \dots, x_n >, n > 0 \\ \perp & \text{sonst} \end{cases}$$

Löscht das erste Element des Stacks.

Abbildung 10.6 zeigt in grafischer Form die Wirkung der beiden Operationen *push* und *pop*. In der Literatur ist oft die *pop*-Operation als Kombination der beiden hier vorgestellten Operationen *pop* und *top* definiert.

### Beispiel 10.22:

Als Anwendung des Datentyps *stack* und des Datentyps *folge* soll nun das Spiegeln eines Wortes betrachtet werden. Ein Wort soll als eine Folge von Buchstaben ansehen werden. Für ein Wort  $< x_1, \dots, x_n >$  soll dann das Ergebnis des Spiegelns das Wort  $< x_n, \dots, x_1 >$  sein.

**Modul** *spiegeln(wort)*

$S_1 : s := emptystack()$

$S_2 : \text{solange nicht isemptyfolge(wort)}$

$S_3 : push(s, first(wort))$

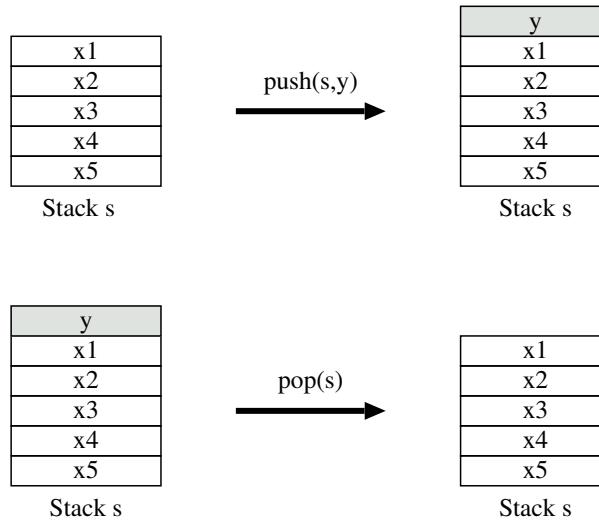


Figure 10.6: push- und pop-Operationen auf einem Stack

$S_4 : \text{wort} := \text{rest}(\text{wort})$   
 $S_5 : \mathbf{end}$   
 $S_6 : \text{spiegelwort} = <>$   
 $S_7 \text{ solange nicht } \text{isemptystack}(s)$   
 $S_8 : \text{spiegelwort} := \text{concat}(\text{spiegelwort}, \text{makefolge}(\text{top}(s)))$   
 $S_9 : s := \text{pop}(s)$   
 $S_{10} : \mathbf{end}$   
 $S_{11} : \text{spiegeln} := \text{spiegelwort}$

### Modulende

Es ist zu beachten, dass  $s$  vom Datentyp *stack* ist und das Wort *wort* vom Datentyp *folge*. Der Algorithmus soll anhand des Wortes  $\langle a, b, c \rangle$  durchgegangen werden. Dazu notiert man sich in einer Tabelle den Inhalt der Daten an bestimmten Punkten in der Ausführung des Algorithmus.

Stelle im Algorithmus	wort	spiegelwort	s
$S_1$	$\langle a, b, c \rangle$	undef.	undef.
$S_5$ (1. mal)	$\langle b, c \rangle$	undef.	$\langle a \rangle$
$S_5$ (2. mal)	$\langle c \rangle$	undef.	$\langle b, a \rangle$
$S_5$ (3. mal)	$\langle \rangle$	undef.	$\langle c, b, a \rangle$
$S_{10}$ (1. mal)	$\langle \rangle$	$\langle c \rangle$	$\langle b, a \rangle$
$S_{10}$ (2. mal)	$\langle \rangle$	$\langle c, b \rangle$	$\langle a \rangle$
$S_{10}$ (3. mal)	$\langle \rangle$	$\langle c, b, a \rangle$	$\langle \rangle$

### Beispiel 10.23:

Als zweite Anwendung der abstrakten Datenstruktur Stack soll die bereits angesprochene Auswertung von arithmetischen Ausdrücken in Postfixnotation angeschaut werden. Der Algorithmus dazu lautet:

### Modul auswerten(ausdruck)

$S_1 : s := \text{emptystack}()$   
 $S_2 : \text{solange nicht } \text{isemptyfolge(ausdruck)}$   
 $S_3 : x := \text{first(ausdruck)}$   
 $S_4 : \text{Fallunterscheidung nach } x :$   
 $S_5 : \text{Falls } x \in \mathbb{Z} :$   
 $S_6 : \text{push}(s, x)$   
 $S_7 : \text{Falls } x = + :$   
 $S_8 : x_1 := \text{top}(s); s := \text{pop}(s)$   
 $S_9 : x_2 := \text{top}(s), s := \text{pop}(s)$   
 $S_{10} : x := x_2 + x_1$   
 $S_{11} : \text{push}(s, x)$   
 $S_{12} : \text{Falls } x = - :$   
 $S_{13} : x_1 := \text{top}(s); s := \text{pop}(s)$   
 $S_{14} : x_2 := \text{top}(s), s := \text{pop}(s)$   
 $S_{15} : x := x_2 - x_1$   
 $S_{16} : \text{push}(s, x)$   
 $S_{17} : \text{Falls } x = * :$   
 $S_{18} : x_1 := \text{top}(s); s := \text{pop}(s)$   
 $S_{19} : x_2 := \text{top}(s), s := \text{pop}(s)$   
 $S_{20} : x := x_2 * x_1$   
 $S_{21} : \text{push}(s, x)$   
 $S_{22} : \text{Falls } x = / :$   
 $S_{23} : x_1 := \text{top}(s); s := \text{pop}(s)$   
 $S_{24} : x_2 := \text{top}(s), s := \text{pop}(s)$   
 $S_{25} : x := x_2 / x_1$   
 $S_{26} : \text{push}(s, x)$   
 $S_{27} : \text{Ende Fallunterscheidung}$   
 $S_{28} : \text{ausdruck} := \text{rest(ausdruck)}$   
 $S_{29} : \text{end}$   
 $S_{30} : \text{auswerten} := \text{top}(s)$

### Modulende

Anhand der Folge  $<3\ 4\ +\ 5\ 1\ 2\ +\ -\ *>$  in Postfixnotation soll die Anwendung des Algorithmus nachvollzogen werden.

Stelle im Algorithmus	ausdruck	x	s
$S_4(1. \text{ mal})$	$<3\ 4\ +\ 5\ 1\ 2\ +\ -\ *>$	3	$<>$
$S_4(2. \text{ mal})$	$<4\ +\ 5\ 1\ 2\ +\ -\ *>$	4	$<3>$
$S_4(3. \text{ mal})$	$<+5\ 1\ 2\ +\ -\ *>$	+	$<4\ 3>$
$S_4(4. \text{ mal})$	$<5\ 1\ 2\ +\ -\ *>$	5	$<7>$
$S_4(5. \text{ mal})$	$<1\ 2\ +\ -\ *>$	1	$<5\ 7>$
$S_4(6. \text{ mal})$	$<2\ +\ -\ *>$	2	$<1\ 5\ 7>$
$S_4(7. \text{ mal})$	$<+\ -\ *>$	+	$<2\ 1\ 5\ 7>$
$S_4(8. \text{ mal})$	$<-\ *>$	-	$<3\ 5\ 7>$
$S_4(9. \text{ mal})$	$<*>$	*	$<2\ 7>$
$S_{30}$	$<>$	*	$<14>$

Der Stack ist eine der wichtigsten Datenstrukturen in der Informatik. Er findet Anwendung in solchen Fällen, wo folgendes umgangssprachlich formuliertes Prinzip angewandt wird: *Behalte ein Zwischenergebnis im Hinterkopf und mache eine Zwischenrechnung an einer anderen Stelle. Wenn die andere Berechnung fertig ist, soll an der alten Stelle weitergemacht werden.*

## 10.5 Schlange \*

Beim Stack wurde nur die Spitze des Stacks manipuliert, entweder das oberste Element entnommen oder ein neues Element auf dem Stack abgelegt. Bei einer **Schlange** (Queue) geschieht die Manipulation in veränderter Form.

Der Datentyp  $queue = < \{M, M^*, \mathbb{B}\}, Ops^{queue} >$  wird wiederum über einer Menge  $M$  definiert. Eine Schlange hat folgende Operationen  $Ops^{queue}$ :

1.  $emptyqueue : \emptyset \rightarrow M^*$  mit:

$$emptyqueue() = <>$$

Erzeugt eine leere Schlange.

2.  $isemptyqueue : M^* \rightarrow \mathbb{B}$  mit:

$$isemptyqueue(q) = \begin{cases} \text{wahr} & \text{falls } q = <> \\ \text{falsch} & \text{sonst} \end{cases}$$

Testet, ob eine Schlange leer ist.

3.  $enqueue : M^* \times M \rightarrow M^*$  mit:

$$enqueue(q, x) = < x_1, \dots, x_n, x > \quad \text{falls } q = < x_1, \dots, x_n >, n \geq 0$$

Fügt das Element  $x$  hinten an die Schlange an.

4.  $head : M^* - \rightarrow M$  mit:

$$head(q) = \begin{cases} x_1 & \text{falls } q = < x_1, \dots, x_n >, n > 0 \\ \perp & \text{sonst} \end{cases}$$

Liefert das erste Element (den Kopf) der Schlange.

5.  $dequeue : M^* - \rightarrow M^*$  mit:

$$dequeue(q) = \begin{cases} < x_2, \dots, x_n > & \text{falls } q = < x_1, \dots, x_n >, n > 0 \\ \perp & \text{sonst} \end{cases}$$

Löscht das erste Element der Schlange.

Abbildung 10.7 zeigt in grafischer Form die Wirkung der beiden Operationen  $enqueue$  und  $dequeue$ .

### Beispiel 10.24:

Als Anwendung des Datentyps  $queue$  soll folgende Problemstellung betrachtet werden. Gegeben sei eine Menge  $M_A$  von Zeichen (Alphabet). Betrachte die Menge  $M = M_A \cup \{/, !\}$ , wobei gelten soll, dass  $M_A \cap \{/, !\} = \emptyset$ . In einer Folge von Zeichen  $< x_1, \dots, x_n >$  existiert genau eine Teilfolge der Form  $< /, y_1, \dots, y_m, / >$ . In der weiteren Folge kommt genau einmal das Zeichen  $!$  vor. Beispiel:  $< a, b, /, 1, 2, /, c, d, !, e >$ . Die Aufgabenstellung ist nun, die Zeichenfolge so zu verarbeiten, dass alle Zeichen unverändert ausgegeben werden. Die Zeichen zwischen den Schrägstrichen werden jedoch nicht ausgegeben. Trifft man aber auf das Zeichen  $!$ , so soll dieses durch die Teilfolge zwischen den Schrägstrichen ersetzt werden. Im Beispiel wäre also das Ergebnis  $< a, b, c, d, 1, 2, e >$ .

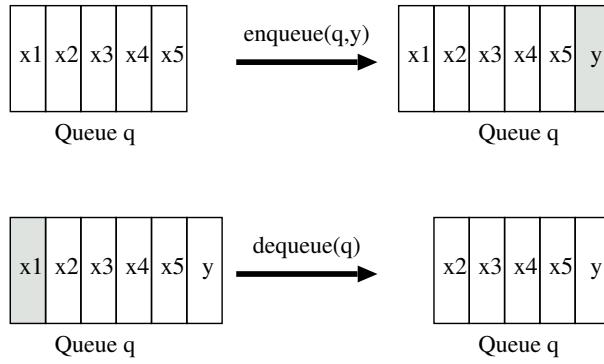


Figure 10.7: enqueue- und dequeue-Operationen auf einer Schlange

**Modul** ersetzen(wort)

```

 $S_1 : q := \text{emptyqueue}()$ 
 $S_2 : \text{ersatzwort} := <>$ 
 $S_3 : \text{solange } \text{first}(w\text{ort}) \neq /$ 
     $S_4 : \text{ersatzwort} := \text{concat}(\text{ersatzwort}, \text{makefolge}(\text{first}(w\text{ort})))$ 
     $S_5 : w\text{ort} := \text{rest}(w\text{ort})$ 
 $S_6 : \text{end}$ 
 $S_7 : w\text{ort} := \text{rest}(w\text{ort})$ 
 $S_8 : \text{solange } \text{first}(w\text{ort}) \neq /$ 
     $S_9 : \text{enqueue}(q, \text{first}(w\text{ort}))$ 
     $S_{10} : w\text{ort} := \text{rest}(w\text{ort})$ 
 $S_{11} : \text{end}$ 
 $S_{12} : w\text{ort} := \text{rest}(w\text{ort})$ 
 $S_{13} : \text{solange } \text{first}(w\text{ort}) \neq !$ 
     $S_{14} : \text{ersatzwort} := \text{concat}(\text{ersatzwort}, \text{makefolge}(\text{first}(w\text{ort})))$ 
     $S_{15} : w\text{ort} := \text{rest}(w\text{ort})$ 
 $S_{16} : \text{end}$ 
 $S_{17} : \text{solange nicht } \text{isemptyqueue}(q)$ 
     $S_{18} : \text{ersatzwort} := \text{concat}(\text{ersatzwort}, \text{makefolge}(\text{head}(q)))$ 
     $S_{19} : q := \text{dequeue}(q)$ 
 $S_{20} : \text{end}$ 
 $S_{21} : w\text{ort} := \text{rest}(w\text{ort})$ 
 $S_{22} : \text{solange nicht } \text{isemptyfolge}(w\text{ort})$ 
     $S_{23} : \text{ersatzwort} := \text{concat}(\text{ersatzwort}, \text{makefolge}(\text{first}(w\text{ort})))$ 
     $S_{24} : w\text{ort} := \text{rest}(w\text{ort})$ 
 $S_{25} : \text{end}$ 
 $S_{26} : \text{ersetzen} := \text{ersatzwort}$ 

```

**Modulende**

Der Algorithmus ist aufgeteilt in verschiedene Phasen, die jeweils für einen Teil der Eingabefolge zuständig sind. In  $S_3$  bis  $S_6$  wird der erste Teil der Zeichenfolge bis zum ersten Vorkommen eines Schrägstrichs bearbeitet, in  $S_8$  bis  $S_{11}$  der Ersatztext, in  $S_{13}$  bis  $S_{16}$  der Text zwischen Ersatztext und !-Zeichen, in  $S_{17}$  bis  $S_{20}$  das !-Zeichen und schließlich in  $S_{22}$  bis  $S_{25}$  der Rest der Zeichenfolge.

Anhand der obigen Folge  $\langle a, b, /, 1, 2, /, c, d, !, e \rangle$  soll der Algorithmus nachvollzogen werden.

Stelle im Algorithmus	wort	ersatzwort	q
vor $S_3$	$\langle a, b, /, 1, 2, /, c, d, !, e \rangle$	$\langle \rangle$	$\langle \rangle$
vor $S_8$	$\langle 1, 2, /, c, d, !, e \rangle$	$\langle a, b \rangle$	$\langle \rangle$
vor $S_{13}$	$\langle c, d, !, e \rangle$	$\langle a, b \rangle$	$\langle 1, 2 \rangle$
vor $S_{17}$	$\langle !, e \rangle$	$\langle a, b, c, d \rangle$	$\langle 1, 2 \rangle$
vor $S_{22}$	$\langle e \rangle$	$\langle a, b, c, d, 1, 2 \rangle$	$\langle \rangle$
vor $S_{26}$	$\langle \rangle$	$\langle a, b, c, d, 1, 2, e \rangle$	$\langle \rangle$

◆

## 10.6 Zusammenfassung und Hinweise

### Verstehen

In abstrakte Datentypen werden Wertemengen mit den darauf definierten Operationen definiert (programmiert). Dadurch hat man an *einer* Stelle alle relevanten Informationen zum Umgang mit Werten solchen Typs.

## Chapter 11

# Grundlagen Objektorientierter Programmierung

In der Realität ist es eher der Normalfall, dass man sich nicht nur ausschließlich mit Informationen befasst, die sich durch eine Zahl oder über ein homogenes Feld von Zahlen adäquat darstellen lassen. Vielmehr muss man sich häufig mit komplexeren Dingen befassen. Möchte man etwa ein Musikstück, ein Bankkonto oder die Bundesligatabelle in einem Programm darstellen, so ist die Struktur dafür eben komplexer. Programmiersprachen liefern für die gerade aufgezählten Beispiele eben nicht mehr einen fertigen Datentyp mit, da man beliebig viele weitere Beispiele bringen könnte, für die man das auch verlangen könnte. Der Ansatz ist vielmehr ein anderer. Aufbauend auf den bereits eingeführten primitiven Datentypen und den weiteren Typkonstrukten (Feld, Referenz) lassen sich neue Typen *selbst programmieren*, die bereits bekannte Typen wiederum nutzen können. Dies führt zu den Begriffen der Klasse und des Objekts, was in diesem Kapitel behandelt wird. Damit verbunden ist auch eine neue und andere Herangehensweise an das Programmieren an sich: das *Objektorientierte Programmieren*.

Eine weitere Idee in diesem Zusammenhang wird es nun auch sein, dass alle Funktionalität zu einem neu geschaffenen Typ an einer Stelle gesammelt wird, was die Software-Entwicklung bei richtigem Gebrauch wesentlich übersichtlicher und weniger fehleranfällig werden lässt, anstatt an vielen Stellen in einem großen Programm etwas zu manipulieren.

Bei der **objektorientierten Software-Entwicklung** denkt und arbeitet man mit Objekten der realen Welt, die als solche in Software modelliert werden und die miteinander kommunizieren (können). Denken Sie nur an eine Menschenmenge, in der Gruppen von Personen zusammenstehen und miteinander reden. Im objektorientierten Programmieransatz wird ein System als eine Menge miteinander kooperierender und kommunizierender **Objekte** aufgefasst, die untereinander **Nachrichten** austauschen. Das Denken in miteinander kommunizierenden Objekten hat den Vorteil, dass sich so sehr realitätsnah einfache wie auch komplexe Systeme in einem Software-System abbilden lassen und die Denkweise des Anwenders (also nicht unbedingt die des Entwicklers) unterstützt wird.

Es sei hier angemerkt, dass insbesondere für komplexere Problemstellungen mit einem objektorientierten Ansatz schon in frühen Phasen des Problemlösungsprozesses begonnen werden muss. Es macht keinen Sinn, eine Spezifikations- und Entwurfsphase nach einem anderen Ansatz zu betreiben, um später eine rein objektorientierte Sprache zur Implementierung zu wählen. Es gibt viele Entwicklungssysteme, die den objektorientierten Ansatz durchgängig in allen Phasen unterstützen.

Im Rest des Kapitels werden einige grundlegende Konzepte des objektorientierten Ansatz aus Programmiersprachen- und Programmentwicklungssicht vorgestellt. Dazu sollen an einem durchgehenden Beispiel nach und nach die verschiedenen Konzepte eingeführt werden. Nachfolgende Kapitel stellen darauf

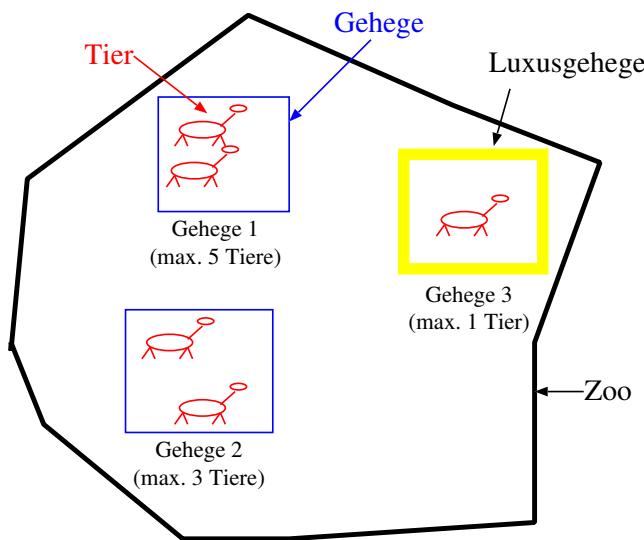


Figure 11.1: Schematische Darstellung eines Zoos

aufbauende, weiterführende Konzepte vor.

### Beispiel 11.1:

Es soll ein Zoo (Abbildung 11.1) als Software-System modelliert werden. Ein Zoo besteht – aus abstrakter Sicht – aus einer Menge von Gehegen und einer Menge von Tieren, die in den Gehegen leben. Jedes Gehege hat eine eindeutige Nummer zur Identifikation und für jedes Gehege gibt es eine Maximalbelegung an Tieren, die nicht überschritten werden darf. Zum Beispiel dürfen im Löwenkäfig nicht mehr als drei Tiere leben. Wenn ein Gehege gebaut wird, so vergibt man eine neue eindeutige Gehegenummer und gibt die Maximalbelegungszahl aufgrund der Größe des Geheges vor. In ein Gehege können Tiere einzahlen und auch wieder ausziehen.

Jedes Tier hat eine eindeutige Nummer zur Identifikation und lebt, bis auf vielleicht eine kurze Phase nach der Geburt, in einem Gehege. Da unser Zoo notorisch an Geldmangel leidet, hat sich die klvere Zoodirektorin weitere Finanzierungsquellen einfallen lassen: zahlungskräftige Besucher können eine Patenschaft für ein Tier übernehmen und verpflichten sich damit, einen jährlichen Betrag zu spenden. Ist ein Tier ein Patentier, so kommt es statt in ein Standardgehege mit Mehrfachbelegung in ein Luxuseinzelgehege, an dem in goldenen Lettern der Name des Paten angebracht wird (natürlich nur solange der Pate zahlt). ♦

## 11.1 Modellierung der Realität

Im Zoobeispiel können Objekte unterschiedlicher Art identifiziert werden (vergleiche obige umgangssprachliche Beschreibung):

1. *Tiere* haben eine Identifikationsnummer und leben in einem Gehege. *Patentiere* sind normale Tiere, beziehen aber zusätzlich ein Jahreseinkommen vom Paten und leben in Luxusgehegen.
2. *Gehege* haben eine Nummer und können bis zu einer Maximalbelegung Tiere aufnehmen. Luxusgehege nehmen nur ein Tier auf.
3. Ein *Zoo* besteht aus einer Menge von Gehegen und Tieren.

Schaut man sich diese Objekte im Beispiel an, so haben diese gewisse *Eigenschaften* (Nummer des Geheges, Maximalbelegung des Geheges) und gewisse *Aktionen/Aktivitäten* können ausgeführt werden (zum Beispiel ein Tier zieht in Gehege ein). Die Eigenschaften eines Objekts nennt man **Attribute** des Objekts und die möglichen Aktionen auf einem Objekt nennt man **Methoden** des Objekts. Jedes Objekt wird also durch seine Attribute (inklusive der Attributwerte) und Methoden beschrieben. Attribute speichern den inneren Zustand des Objektes, Methoden sind die Schnittstelle eines Objekts nach außen. Die Attribute von Objekten ergeben sich oft aus Nomen (Hauptwörtern) einer textuellen Problembeschreibung. Bei Tieren sind dies etwa die Identifikationsnummer oder das Gehege, in dem ein Tier lebt. Methoden ergeben sich eher aus den Verben der textuellen Beschreibung.

Zum Beispiel könnte ein Gehege-Objekt folgende Attribute und Methoden haben:

Attribute	Methoden
eindeutige Nummer zur Identifikation	Gehege anlegen
Maximalbelegung	Gehege abreissen
aktuelle Belegung	Gehege bekommt eine neues Tier zugewiesen Aus Gehege zieht ein Tier aus Ist das Gehege voll?

Ein Beispiel für ein konkretes Gehege-Objekt, das die Nummer 1 hat, maximal 3 Tiere aufnehmen kann und zur Zeit 2 Tiere enthält, wäre demzufolge:

Attribute	Methoden
eindeutige Nummer zur Identifikation = 1	Gehege anlegen
Maximalbelegung = 3	Gehege abreissen
aktuelle Belegung = 2	Gehege bekommt eine neues Tier zugewiesen Aus Gehege zieht ein Tier aus Ist das Gehege voll?

Ein Tier-Objekt hat beispielsweise folgende Attribute und Methoden:

Attribute	Methoden
Identifikationsnummer	Tier wird geboren
Name	Tier stirbt
In welchem Gehege zu finden	Tier zieht in Gehege ein/um
Jahreseinkommen (bei Patentieren)	Tier bekommt Patenschaft (bei Patentieren)

Ein Beispiel für ein konkretes Tier-Objekt mit Namen Gerti und mit der Nummer 17 in Gehege 1 lebend und das ein Patentier mit einem Jahreseinkommen von 100 Euro ist, wäre demzufolge:

Attribute	Methoden
Nummer = 17	Tier wird geboren
Name = Gerti	Tier stirbt
In welchem Gehege zu finden = 1	Tier zieht in Gehege ein/um
Jahreseinkommen (bei Patentieren) = 100	Tier bekommt Patenschaft (bei Patentieren)

Die Methode "Tier wird geboren" könnte zum Beispiel dem Attribut Nummer des konkreten Tieres die eindeutige Identifikationsnummer zuweisen.

## 11.2 Klassenbildung

Um das Zoomodell zu entwerfen, könnte man für *jedes* Tier zum Beispiel eine Methode schreiben, die das Einziehen in einen Gehege beschreibt. Wie man leicht einsieht, ist dies unnötig, da *alle* Tiere diesen

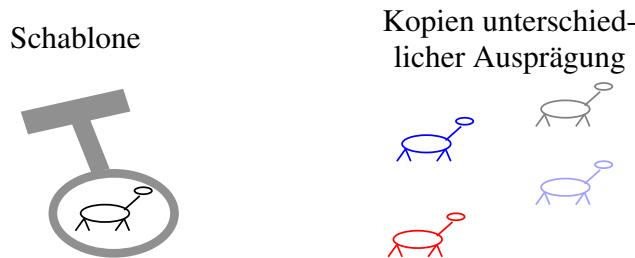


Figure 11.2: Schablone (Klasse) und Kopien mit unterschiedlicher Ausprägung (Objekte)

Vorgang ausführen können und der Vorgang auch (fast) identisch sein wird. Ebenfalls gilt dies für die anderen identifizierten Methoden von Tierobjekten und ebenso für Gehege. Ebenfalls haben *alle* Tier- und Gehegeobjekte die gleichen Attribute, lediglich die Ausprägung der Attribute (der Wert eines Attributs) ist eventuell verschieden, zum Beispiel in *welchem* Gehege das Tier lebt, *welche* Nummer ein Tier hat.

Die Beschreibung von Objekten gleicher Art fasst man zu **Klassen** zusammen (Abstraktion). Eine Klasse besitzt alle wesentlichen Attribute und Methoden der Objekte, die unter dieser Klasse zusammengefasst werden sollen. Im Gegensatz zu einem Objekt haben in einer Klasse die Attribute allerdings keine Werte. Es wird also beispielsweise in einer Klasse angegeben, *dass* ein Tier / jedes Tier eine Nummer hat, aber nicht welche Nummer. Dieser Abstraktionsvorgang wurde implizit schon in der Angabe der Attribute und Methoden der Tier- und Gehegeobjekte im letzten Abschnitt gemacht. Zuerst wurden allgemein Gehege und Tiere beschrieben und anschließend wurde jeweils ein konkretes Beispiel für ein Gehege bzw. ein Tier angegeben, in dem jedem Attribut ein Attributwert zugeordnet war. Eine Klasse (mit den offengelassenen Attributwerten!) ist also eine Art Blaupause für eine ganze Tierklasse, Gehegeklasse usw., in der von dem konkreten Tier, von dem konkreten Gehege abstrahiert wird (Abbildung 11.2). Erzeugt man aus der Blaupause eine Kopie und füllt die Attributwerte aus (siehe Beispieldier), erhält man aus der Tierklasse ein konkretes Tierobjekt. Ein solches Objekt zu einer Klasse nennt man auch eine **Instanz** der Klasse.

An dieser Stelle soll nun die Java-Umsetzung zur Tier-Klasse diskutiert werden, die die wesentlichen Eigenschaften und Methoden aller Tiere zusammenfassen soll. Im letzten Abschnitt wurden diese Attribute und Methoden ja bereits identifiziert. Patentierte sollen zuerst einmal aussen vor gelassen werden, diese werden später in Kapitel 12.1 betrachtet. In vereinfachter Form wird in Java eine Klasse mit dem Schlüsselwort `class` eingeleitet (siehe Java Language Specification [Javc] für alle Möglichkeiten der Syntax), gefolgt von dem Namen der Klasse. Anschließend kommen in geschweiften Klammern eingeschlossen die Attribute und Methoden der Klasse, die von der Java-Sprache her in der Reihenfolge beliebig gemischt werden können. Die Java Code Conventions fordern aber, zuerst alle Attribute und dann alle Methoden anzugeben. Attributdeklarationen sind analog den bereits bekannten Variablendefinitionen, allerdings außerhalb einer Methode, vereinbarungsgemäß am Anfang einer Klasse. Der Variablenname entspricht dem Attributnamen, der Variablenwert dem Attributwert. In einer *Klassendefinition* ist entsprechend den obigen Anmerkungen zu Klassen und Objekten der Attributwert offen gelassen (Blaupause), bei konkreten *Objekten* hat jedes Attribut aber einen ganz konkreten Wert (gleich mehr dazu). Methodendefinitionen wurden im Prinzip bereits früher in Kapitel 8 vorgestellt. In Abbildung 11.3 ist die Java-Syntax für eine einfache Form einer Klassendefinition angegeben.

Durch eine Klassendefinition wird ein neuer Typ eingeführt, der den Namen der Klasse besitzt. Ein solcher Typ kann überall dort genutzt werden, wo ein Typ verlangt ist (Variablendeclaration, Parameterspezifikation bei Methoden,...).

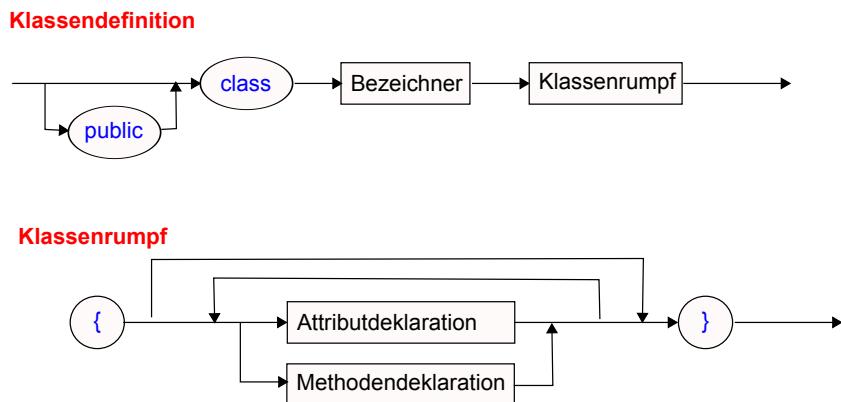


Figure 11.3: Syntaxdiagramm für eine einfache Form der Klassendefinition in Java.

Listing 11.1: Klasse Tier (Version 1, unvollständig).

```

11-1 /**
11-2 * Tier ( Version 1, unvollstaendig )
11-3 */
11-4 class Tier {
11-5
11-6     // Angabe von Attributen
11-7
11-8     // Angabe von Methoden
11-9 }
```

Listing 11.2: Klasse Gehege (Version 1, unvollständig).

```

11-1 /**
11-2 * Gehege ( Version 1, unvollstaendig )
11-3 */
11-4 class Gehege {
11-5
11-6     // Angabe von Attributen
11-7
11-8     // Angabe von Methoden
11-9 }
```

**Beispiel 11.2:**

Listing 11.1 zeigt eine unvollständige Klasse zu einem Tier und Listing 11.2 die entsprechende unvollständige Gehegeklasse.



Klassennamen beginnen vereinbarungsgemäß mit einem Großbuchstaben. Eine Klasse **Abc** muss in einer Datei **Abc.java** abgelegt sein, also Klassenname und der Präfix (1. Teil) des Dateinamens müssen absolut identisch sein, inklusive Groß-Kleinschreibung. Diese Regelung wird später wieder etwas aufgeweicht werden im Zusammenhang mit inneren/lokalen Klassen (Kapitel 13.4) und Zugriffsrechten (Kapitel 12.4).

## 11.3 Instanzvariablen und Instanzmethoden

Mit Attributdeklarationen als Teil einer Klassendefinition werden Variablen deklariert, die überall im Programmtext der Klasse als "normale" Variablen unter ihrem einfachen Namen genutzt werden können. Man spricht dabei auch von **Instanzvariablen**. Jede *konkrete Instanz* einer Klasse hat nun **einen eigenen Satz** dieser Instanzvariablen. Während in einer Klassendefinition einmalig angegeben wird, dass ein / jedes Tier zum Beispiel eine Nummer hat, so besitzt *jede Instanz* dieser Klasse genau eine eigene Instanzvariable **nummer** mit einem konkreten und individuellen Wert. Wird innerhalb einer Klasse auf eine Instanzvariable mit ihrem einfachen Namen Bezug genommen / sie verwendet, so muss dies immer relativ zu einem Bezugsobjekt erfolgen, worauf später noch eingegangen wird.

In Listing 11.3 ist die erweiterte, aber weiterhin noch unvollständige Klassendefinition in Java zur Tierklasse angegeben<sup>1</sup>. In der Klasse wird also definiert, dass jedes Tier ein Attribut für seine Identifikationsnummer besitzt (der Attributname ist **nummer** und der Typ dieses Attributs ist **int**), einen Namen (der Attributname ist **name** und der Typ dieses Attributs ist **String**) und ein weiteres Attribut für das Gehege, in dem das Tier lebt. Der Attributname dazu ist **gehege** und der Typ dieses letzten Attributs ist **Gehege**. Der Typ **Gehege** ist kein vordefinierter Java-Typ wie **int** oder **String**, sondern dieser Typ entspricht dem Klassennamen unserer Gehegeklasse. Auf Instanzmethoden, die im Beispiel schon aufgeführt sind, wird gleich eingegangen.

Analog zur Tierklasse lässt sich nun ebenso die Klasse **Gehege** angeben, die die Blaupause für alle Gehege sein soll (siehe Listing 11.4). Jedes Gehege hat also die Attribute (**nummer**, **maximalBelegung** und **aktuelleBelegung**, alle vom Typ **int**) sowie **bewohner**, ein Feld von Tieren. Ebenso wie in der Tierklasse werden auch hier einige der Methodenrümpfe der Übersichtlichkeit halber noch leer gelassen.

Es ist allerdings bei Instanzvariablen zu beachten, dass diese andere Eigenschaften als die bis jetzt bekannten blocklokalen Variablen haben, es sind eine andere Art von Variablen. Der **Gültigkeitsbereich** der Instanzvariablen erstreckt sich über die gesamte Klasse und die **Lebensdauer** ist an die Lebensdauer des Objekts gekoppelt. Zur Erinnerung: der Gültigkeitsbereich von blocklokalen Variablen ist der die Deklaration umfassende Block und die Lebensdauer ist auf die Ausführung dieses Blocks beschränkt. Es kann damit jetzt auch der gleiche Name in doppelter Bedeutung *gleichzeitig* vorkommen: einmal als Name einer blocklokalen Variablen und *gleichzeitig* mit einem sich überschneidenden Gültigkeitsbereich als Instanzvariable. Bei solchen Überscheidungen des Gültigkeitsbereichs wird die Sichtbarkeit der Instanzvariablen eingeschränkt (siehe Kapitel 6.2 und 6.3 und das dort angegebene Beispiel). Es ist jedoch allgemein gesprochen keine gute Programmierpraxis, den gleichen Namen für verschiedene Dinge zu vergeben.

Die Methoden einer Klasse werden entsprechend der bereits vorgestellten Methodensyntax angegeben. In den Beispielklassen **Tier** und **Gehege** gibt es aber einen Unterschied zur Methodendefinition, wie sie in Kapitel 8 in den Beispielen angewandt wurde: es fehlt das Schlüsselwort **static** in Methodenkopf. Methoden ohne das Schlüsselwort **static** sind **Instanzmethoden**, Methoden mit dem Deklarationszusatz **static** sind **Klassenmethoden**; auf den Unterschied wird später noch gesondert im Zusammenhang mit

---

<sup>1</sup> Von diesem Zeitpunkt weichen wir von den Java Code Conventions ab, indem kleine Methoden in einer Zeile angegeben werden, um die Darstellung kompakt zu halten.

Listing 11.3: Klasse Tier mit Instanzvariablen und Instanzmethoden (Version 2, unvollständig).

```

11-1 /** Tier V2 (Version 2, unvollstaendig)
11-2  */
11-3 class Tier {
11-4     // Instanzvariablen
11-5     int nummer;           // eindeutige Nummer des Tieres
11-6     String name;        // Name des Tieres (evtl. nicht vorhanden)
11-7     Gehege gehege;      // Gehege, in dem es wohnt
11-8
11-9     // Methoden der Klasse
11-10    // liefere Name des Tieres
11-11    String getName() { return name; }
11-12
11-13    // aendere Namen des Tieres
11-14    void setName(String neuerName) { name = neuerName; }
11-15
11-16    /** ein Tier zieht in ein anderes Gehege um
11-17    * @param gehege das neue Gehege
11-18     */
11-19    void umziehen(Gehege gehege) {
11-20        // hier noch offen gelassen
11-21    }
11-22
11-23    /** liefere String-Darstellung eines Tiers
11-24     */
11-25    public String toString() {
11-26        return "Tier " + getName()
11-27        + ", Gehege=" + ((gehege == null) ? "ohne Gehege" : nummer);
11-28    }
11-29}

```

Listing 11.4: Klasse Gehege mit Instanzvariablen und Instanzmethoden (Version 2, unvollständig).

```

11-1  /** Gehege Version 2 (noch unvollstaendig)
11-2  */
11-3  class Gehege {
11-4      // Instanzvariablen
11-5      int nummer;           // eindeutige Nummer
11-6      int maximalBelegung; // max. Anzahl an Tieren in Gehege
11-7      int aktuelleBelegung; // derzeitige Anzahl
11-8      Tier[] bewohner;    // derzeitige Tiere
11-9
11-10     // Methoden
11-11     // liefere Nummer des Geheges
11-12     int getNummer() { return nummer; }
11-13
11-14     // liefere Maximalbelegung
11-15     int getMaximalBelegung() { return maximalBelegung; }
11-16
11-17     // liefere aktuelle Belegung
11-18     int getAktuelleBelegung() { return aktuelleBelegung; }
11-19
11-20     // Ein Tier zieht ein. Es muss noch Platz im Gehege sein!
11-21     void einziehen(TierV2 tier) {
11-22         // hier noch freigelassen
11-23     }
11-24
11-25     // Ein Tier zieht aus. Es muss im Gehege vorhanden sein!
11-26     void ausziehen(TierV2 tier) {
11-27         // hier noch freigelassen
11-28     }
11-29
11-30     // liefere String-Darstellung eines Geheges
11-31     public String toString() {
11-32         return "Gehege " + nummer + ", maximal=" + maximalBelegung
11-33             + ", aktuell=" + aktuelleBelegung;
11-34     }
11-35 }
```

Klassenmethoden eingegangen (Kapitel 11.5). Die Anwesenheit beziehungsweise Abwesenheit von `static` in einer Methodendeklaration spezifiziert also, ob es sich um Klassenmethode oder Instanzmethode handelt.

## 11.4 Instanziierung, Instanzen, Konstruktoren und Bezugsobjekt

Bis jetzt wurden nur Klassen definiert, die gemeinsame Eigenschaften von Objekten gleicher Art zusammenfassen. Dieser Ansatz ist also ausschließlich *beschreibender / deklarativer Art*. Der eigentliche Sinn der Software-Modellierung ist es aber natürlich, irgendwann auch mit konkreten Objekten zu arbeiten (in einem Zoo leben 12 Tiere in 4 Gehege), die Klassen dienen also nur zur Arbeitserleichterung, um Objekte gleicher Art zusammenfassend beschreiben zu können (zu einer allgemeinen Tierbeschreibung zu abstrahieren).

### 11.4.1 Instanziierung und Instanzen

Über eine Klasse lassen sich nun aber *beliebig viele Objekte erzeugen*, die alle genau die Eigenschaften und Fähigkeiten dieser Klasse besitzen. Man spricht bei diesem Vorgang von **Instanziierung**, ein erzeugtes Objekt nennt man **Instanz** der Klasse. Siehe auch nochmals Abbildung 11.2 zu dieser Idee.

Als Folgerung daraus haben alle instanzierten Objekte einer Klasse die gleichen Attribute und Methoden, nämlich genau die der Klasse. Während eine Klasse eine Blaupause für alle Objekte dieser Klasse ist, ist ein Objekt eine spezielle Ausprägung dieser Klasse. Im Gegensatz zu einer deklarativen Beschreibung in einer Klasse, wo die Existenz von Attributen angegeben wird, aber nicht deren Werte, haben in Objekten die Attribute jetzt auch zusätzlich konkrete Werte, jeweils spezifisch zu einem Objekt. Man nimmt also bildlich eine Kopie von der Blaupause und füllt die leeren Attributwerte aus.

Während in einer Tierklasse also gesagt wird, *dass* ein Tier eine Nummer hat (Attribut Nummer) und wie man es allgemein verändern kann (in der Methode `setName`), hat nun eine Instanz dieser Klasse, also ein Tierobjekt, zu diesem Attribut auch einen konkreten Wert, zum Beispiel den Wert 1, und bei Ausführung der Methode `setName` würde sich für genau ein Objekt dessen Name dadurch ändern (und nicht für alle Tiere). Verschiedene Tierobjekte werden dann auch jeweils einen anderen Wert zu diesem Attribut haben (1,2,3,...)

#### Beispiel 11.3:

Im Tierbeispiel sagt die Tierklasse aus, *dass* jedes Tier eine Nummer hat (Attribut Nummer). Es steht aber in dieser Deklaration keine konkrete Nummer dabei, weil die Klasse ja *alle möglichen* Tierobjekte abstrakt beschreiben will. Jedes Tier soll eine Nummer haben, aber diese Nummern werden sich ja zwischen allen Tieren unterscheiden, so dass eine konkrete Nummernangabe in einer Klasse wenig Sinn machen würde (wie sollte das dann auch für mehr als ein Objekt überhaupt angegeben werden?).

Ein *konkretes Tierobjekt* hat, wie jedes andere Tierobjekt auch, nun aber eine konkrete Nummer, zum Beispiel 4711, ein anderes die Nummer 17. Alle Tierobjekte haben also ein Attribut Nummer, der *Wert* des Attributs wird sich aber zwischen den Tierobjekten unterscheiden. ♦

### 11.4.2 Konstruktoren

Um den Instanziierungsprozess beeinflussen zu können, kann in einer Klasse ein oder auch mehrere **Konstruktoren** definiert werden, eine spezielle Art von Methode. Ein Konstruktor wird analog zu einer (Instanz-)Methode angegeben, mit zwei Besonderheiten:

1. Es wird kein Ergebnistyp angegeben, diese Stelle der Methodendefinition bleibt leer.
2. Der Name des Konstruktors muss mit dem Namen der Klasse übereinstimmen.

Ein Konstruktornname kann damit also analog zu Methodennamen überladen werden (siehe Kapitel 8.3), die Signaturen müssen sich dabei wie bekannt unterscheiden. Wird in einer Klasse kein Konstruktor angegeben (*und nur dann!*), so ist in dem Fall und nur in dem Fall implizit ein parameterloser Konstruktor definiert, der **Default-Konstruktor**, dessen Methodenrumpf leer ist, also nichts macht.

In einem Konstruktor wird man üblicherweise Aktionen durchführen, die beim Erzeugen eines Objekts anfallen würden, etwa einen selbst definierten Anfangszustand eines Objekts herzustellen. Ein Beispiel dazu wären etwa die Vergabe eines Namens und einer Identifikationsnummer für ein konkretes Tierobjekt, wenn ein Tierobjekt erzeugt / instanziert wird.

Listing 11.5 und 11.6 zeigen in einer weiteren Version die Tier-Klasse mit zwei Konstruktoren und die Gehege-Klasse, ebenfalls mit zwei Konstruktoren. Einige Methodenrumpfe sind in dieser Version noch leer.

### 11.4.3 Operator new

Über einen Konstruktor lässt sich also der Instanziierungsprozess beeinflussen. Dies geschieht genau dann, wenn der Konstruktor aufgerufen wird, was an beliebiger Stelle in einem Programm geschehen kann, auch außerhalb der eigenen Klasse (was sogar dem Normalfall entspricht). Ein Konstruktor wird aufgerufen, wenn ein Objekt aktiv erzeugt wird. Die Instanziierung eines anonymen Objektes einer Klasse geschieht über den **new-Operator**, dem die Angabe eines Konstruktorauftrufs folgt (vergleiche auch Kapitel 9.1.1 zur Erzeugung von Feldern). Also beispielsweise für einen parameterlosen Konstruktor: `new Klassenname()`, beziehungsweise konkret für die Tier-Klasse: `new Tier()`. Über die Parameterangaben wird bei überladenen Konstruktornamen der konkrete Konstruktor ausgewählt, wie dies auch schon bei überladenen normalen Methodenauftrufen besprochen wurde.

Zu Beginn der Auswertung des `new`-Operators wird vom Java-Laufzeitsystem im Heap Speicher angelegt für das neue Objekt inklusive aller Instanzvariablen, die alle jeweils mit dem entsprechenden Nullwert des jeweiligen Datentyps belegt werden. Also beispielsweise `(int) 0` oder `(float) 0` oder `null` bei Referenzen. Dies ist ein anderes Verhalten als für blocklokale Variablen, die einen undefinierten Wert nach einer Deklaration ohne Initialisierungsausdruck haben! Nach der Speicherallokation wird der durch die Parameterangaben spezifizierte Konstruktor aufgerufen und ausgeführt. Dementsprechend wird man oft in einem Konstruktor Aktionen durchführen, die ein Objekt in einen definierten Ausgangszustand bringen, der dann auch sich von ausschließlich Null-Werten unterscheidet; diese Initialisierung mit Nullwerten wird ja implizit immer getan und müsste demzufolge auch nicht explizit noch getan werden.

Der `new`-Operator gefolgt von einem Konstruktorauftruf ist ein Ausdruck. Der Wert dieses Ausdrucks ist eine Referenz auf das neu geschaffene Objekt im Heap, der Resultattyp des Ausdrucks ergibt sich aus dem Typ des neu geschaffenen Objekts. Da ein anonymes Objekt geschaffen wurde, ist diese Referenz damit aber auch nach dem Erzeugungsprozess die *einige Möglichkeit*, um dieses Objekt referenzieren zu können. Oft wird man diese Referenz deshalb in einer Variablen eines entsprechenden (Referenz-)Typs speichern (siehe nachfolgendes Beispiel).

#### Beispiel 11.4:

Ein konkreter Zoo soll nun "gebaut" werden. Dieser Zoo enthält zwei Tiere und zwei Gehege. Listing 11.7 zeigt die entsprechenden Programmlogik. Zu Beginn werden in `main` in zwei Deklarationen mit Initialisierungsausdruck zwei Tiere instanziert und die Referenzen auf die beiden neuen Objekte in zwei Variablen `t1` und `t2` vom Typ `Tier` gespeichert. Rechts vom Gleichheitszeichen, im Initialisierungsausdruck der Deklaration, steht der Instanziierungsausdruck, der als Ergebnis eine Referenz auf das neu geschaffene Objekt liefert. Links vom Gleichheitszeichen steht die Typspezifikation und die Angabe des Namens der Variablen. Man beachte, dass der Klassenname `Tier` als Typangabe in der Variablen Deklaration zur Variablen `t1` und `t2` verwendet wird. Die beiden Zeilen zur Erzeugung der Gehege sind analog. ♦

Listing 11.5: Klasse Tier (Version 3 mit Konstruktoren, unvollständig).

```

11-1  /** Tier (Version 3, unvollständig)
11-2  */
11-3  class Tier {
11-4      // Instanzvariablen
11-5      int nummer;           // eindeutige Nummer des Tieres
11-6      String name;        // Name des Tieres (evtl. nicht vorhanden)
11-7      Gehege gehege;      // Gehege, in dem es wohnt
11-8
11-9      // parameterloser Konstruktor
11-10     Tier() {
11-11         // vergabe automatisch eine neue Nummer. Wie???
11-12         nummer = 1;
11-13         this.name = "unbekannt";
11-14         // gehege ist bereits null
11-15     }
11-16
11-17     // Erzeuge ein neues Tier mit Namen
11-18     Tier(String name) {
11-19         // vergabe automatisch eine neue Nummer. Wie???
11-20         nummer = 1;
11-21         this.name = name;
11-22         // gehege ist bereits null
11-23     }
11-24
11-25     // liefere Name des Tieres
11-26     String getName() { return name; }
11-27
11-28     // aendere Name des Tieres
11-29     void setName(String name) { this.name = name; }
11-30
11-31     // ein Tier zieht in ein anderes Gehege um
11-32     void umziehen(Gehege gehege) {
11-33         // hier noch offen gelassen
11-34     }
11-35
11-36     // liefere String-Darstellung eines Tiers
11-37     public String toString() {
11-38         return "Tier " + getName()
11-39         + ", Gehege=" + ((gehege == null) ? "ohne Gehege" : nummer);
11-40     }
11-41 }
```

Listing 11.6: Klasse Gehege (Version 3 mit Konstruktoren, unvollständig).

```

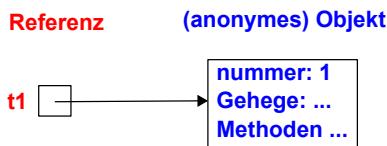
11-1  /** Gehege Version 3 (noch unvollstaendig)
11-2  */
11-3  class Gehege {
11-4      // Instanzvariablen
11-5      int nummer;           // eindeutige Nummer
11-6      int maximalBelegung; // max. Anzahl an Tieren in Gehege
11-7      int aktuelleBelegung; // derzeitige Anzahl
11-8      Tier[] bewohner;    // derzeitige Tiere
11-9
11-10     // Konstruktoren
11-11     // Konstruktor fuer Standardgehege
11-12     Gehege() {
11-13         // nutze anderen Konstruktor mit Defaultgroesse
11-14         this(5);
11-15     }
11-16     // Konstruktor fuer Gehege mit einer vorgegebenen Maximalbelegung
11-17     Gehege(int maximalBelegung) {
11-18         // vergabe Nummer und erhoehe den Zaehler fuer vergeben Nummern (Klassenvariable).
11-19         Wie???
11-20         nummer = 1;
11-21         this.maximalBelegung = maximalBelegung;
11-22         bewohner = new Tier[maximalBelegung];
11-23     }
11-24
11-25     // Methoden
11-26     // liefere Nummer des Geheges
11-27     int getNummer() { return nummer; }
11-28
11-29     // liefere Maximalbelegung
11-30     int getMaximalBelegung() { return maximalBelegung; }
11-31
11-32     // liefere aktuelle Belegung
11-33     int getAktuelleBelegung() { return aktuelleBelegung; }
11-34
11-35     // Ein Tier zieht ein. Es muss noch Platz im Gehege sein!
11-36     void einziehen(Tier tier) {
11-37         // hier noch offen gelassen
11-38     }
11-39
11-40     // Ein Tier zieht aus. Es muss im Gehege vorhanden sein!
11-41     void ausziehen(Tier tier) {
11-42         // hier noch offen gelassen
11-43     }
11-44
11-45     // liefere String-Darstellung eines Geheges
11-46     public String toString() {
11-47         return "Gehege " + nummer + ", maximal=" + maximalBelegung
11-48             + ", aktuell=" + aktuelleBelegung;
11-49     }

```

Listing 11.7: Klasse Zoo (Version 1, unvollständig).

```

11-1 /**
11-2 * Zoo (Version 1, noch unvollständig)
11-3 */
11-4 public class Zoo {
11-5
11-6     public static void main(String [] args) {
11-7
11-8         // zwei Tiere erzeugen
11-9         Tier t1 = new Tier();
11-10        Tier t2 = new Tier("Gerti");
11-11
11-12         // Standardgehege erzeugen
11-13         Gehege g1 = new Gehege();
11-14         // Grossgehege erzeugen
11-15         Gehege g2 = new Gehege(10);
11-16     }
11-17 }
```

Figure 11.4: Erzeugen eines anonymen Objektes und Zuweisung der Referenz an die Variable `t1`.

In der Deklarationszeile

```
11-1 Tier t1 = new Tier();
```

werden – wie bereits früher besprochen – zwei Schritte zusammengefasst: Das Erzeugen eines neuen anonymen Objektes vom Typ `Tier` und die Definition einer Variablen `t1` und deren Initialisierung mit der Speicheradresse des neuen anonymen Objektes. Abbildung 11.4 zeigt schematisch das Resultat dieser Programmzeile. Mit dieser Zeile wird also zusammengefasst ein Objekt vom Typ (oder Klasse) `Tier` erzeugt und dieses Objekt ist dann anschließend unter dem Variablenamen `t1` referenzierbar.

#### 11.4.4 Bezugsobjekt

Arbeitet man mit Instanzvariablen und Instanzmethoden, so stellt sich die Frage, auf welches Objekt man sich dabei bezieht. Wie eben angemerkt, gibt eine Attributdeklaration in einer Klasse an, dass *alle* Objekte dieser Klasse dieses Attribut haben sollen. Aber jedes instanzierte Objekt dieser Klasse hat ja einen *eigenen Satz an Instanzvariablen mit objektspezifischen Werten*. Wenn also jetzt im Programmcode in Listing 11.3 etwa in der Methode `setName` steht `name = neuerName`, so stellt sich die Frage, zu welcher Instanz der Wert der Instanzvariablen `name` verändert werden soll. Die Antwort ist, dass dies eine Instanz ist, die nachfolgend **Bezugsobjekt** genannt wird, und die sich aus der Nutzung ergibt. Wird eine Instanzmethode von außerhalb der Klasse aufgerufen, so muss *explizit* ein Bezugsobjekt angegeben werden. Die Syntax dafür ist `Bezugsobjekt .Methodenaufruf`, wobei `Bezugsobjekt` oft eine Referenzvariable sein wird, die eine Referenz auf ein Objekt des entsprechenden Typs enthält. Ohne die Angabe eines solches Bezugsobjekt macht ein Aufruf einer Instanzmethode *außerhalb der eigenen Klasse* keinen Sinn und ist auch syntaktisch nicht erlaubt. Ruft man *innerhalb einer Instanzmethode* eine weitere Instanzmethode der gleichen Klasse auf (auch Rekursion ist dabei

Listing 11.8: Beispiel zur Nutzung von Instanzmethoden.

```

11-1 public class Test {
11-2     public static void main(String[] args) {
11-3         Tier t1 = new Tier();
11-4         // rufe Methode auf mit Bezugsobjekt t1
11-5         t1.setName("Helmut");
11-6         // rufe weitere Methode auf mit Bezugsobjekt t1
11-7         String s = t1.getName()
11-8         // rufe toString auf und damit innerhalb der toString Methode
11-9         // die Methode getName()
11-10        System.out.println("Tier t1 ist " + t1.toString());
11-11
11-12        // erzeuge weiteres Tier
11-13        Tier t2 = new Tier();
11-14        // rufe Methode auf mit Bezugsobjekt t2
11-15        t2.setName("Gerti");
11-16    }
11-17}

```

möglich), so bleibt das Bezugsobjekt das gleiche.

### Beispiel 11.5:

Gegegen ist der Beispielcode in Listing 11.8 sowie die Tierklasse aus Listing 11.5.

In der Zeile `t1.setName("Helmut");` wird also die Methode `setName` der Tierklasse aufgerufen. Mit dem Präfix `t1.` wird explizit das Bezugsobjekt für genau diesen Aufruf angegeben.

Beim Aufruf der Methode `toString` mit dem expliziten Bezugsobjekt `t1` wird innerhalb der Instanzmethode `toString()` die Instanzmethode `getName()` der gleichen Klasse aufgerufen. Dabei bleibt das Bezugsobjekt das gleiche. ♦

Innerhalb einer Instanzmethode lässt sich das Bezugsobjekt über das Schlüsselwort `this` referenzieren. `this` ist also eine Objektreferenz. Typische Beispiele zur Nutzung von `this` sind:

- Treten Überschneidungen im Gültigkeitsbereich von Variablen auf (blocklokale Variablen / formale Parameter einer Methode und Attribute), so ist die bereits eben angegebene Regel, dass Attribute für den Überschneigungsbereich unsichtbar werden. Blocklokale Variablen sind also unter ihrem einfachen Namen weiterhin referenzierbar. Möchte man trotzdem die gleichlautende Instanzvariable ansprechen, so kann man dies explizit über `this.attributname` erreichen.

```

11-1     // Instanzvariable
11-2     String name;
11-3
11-4     // Erzeuge ein neues Tier mit Namen
11-5     Tier(String name) {
11-6         this.name = name; // Ueberschneidung der Gueltigkeitsbereiche
11-7     }

```

- Analog dazu kann man *explizit* eine Instanzmethode des eigenen Objekts ansprechen:

```

11-1     public String toString() {
11-2         return "Tier " + getName();
11-3     }

```

und

```
11-1  public String toString() {
11-2      return "Tier " + this.getName();
11-3  }
```

sind in diesem Beispiel äquivalent.

- Soll das aktuelle Bezugsobjekt über die Referenz auf das Objekt an eine weitere Methode als aktueller Parameterwert übergeben werden, so kann man dies mittels `this` erreichen.

```
11-1  // rufe zu "meinem" Gehege die Methode ausziehen() mit mir als Argument auf
11-2  this.gehege.ausziehen(this);
```

- Innerhalb eines Konstruktors lässt sich ein weiterer Konstruktor der eigenen Klasse `this()` aufrufen, wenn dies als *erste* Anweisung im Konstruktor geschieht. Analog ist dies auch mit Konstruktoren möglich, die Parameter erwarten.

```
11-1 class Tier {
11-2     ...
11-3     Tier() {
11-4         // rufe anderen Konstruktor auf, der alles weitere macht
11-5         this("unbekannt");
11-6     }
11-7
11-8     Tier(String name) {
11-9         this.name = name;
11-10    }
11-11 }
```

## 11.4.5 Lebensdauer von Objekten

Die **Lebensdauer eines Objektes** beginnt mit seiner Instanziierung und endet, wenn keine aktive Referenz mehr auf dieses Objekt existiert. Eine Referenz gilt als aktiv, wenn sie über eine Referenz auf dem Laufzeitstack direkt oder indirekt (zum Beispiel über ein weiteres referenziertes Objekt) erreichbar ist.

Damit ist es aber nun auch möglich, dass nicht mehr referenzierbare (also nicht mehr aktive) Objekte existieren können. Diese Objekte sind ab dem Zeitpunkt des Programmablaufs, wo die letzte aktive Referenz verschwindet, nicht mehr referenzierbar und belegen Speicherplatz im Heap. Würde man es bei diesem Zustand belassen, würde immer mehr Heap-Speicher/Hauptspeicher belegt, der nicht mehr referenzierbar wäre und irgendwann kein Speicher mehr für neue Objekte zur Verfügung stehen. Das Java Laufzeitsystem nutzt aus diesem Grund einen **Garbage Collector** (Müllsampler), der von Zeit zu Zeit alle "'Objektleichen'" aus dem Heap entfernt und damit wieder neuen freien Speicher schafft. In anderen Programmiersprachen wie etwa C++ muss ein Programmierer sich explizit um das Aufräumen des Speichers kümmern.

### Beispiel 11.6:

Im Listing 11.9 ist ein Beispiel angegeben zu dem Fall, dass Objektleichen auf dem Heap entstehen können.

Listing 11.9: Beispiel zur Lebensdauer von Objekten.

```

11-1 public class LebensdauerObjekt {
11-2     public static void main(String[] args) {
11-3         /* Die Referenz t1 wird auf dem Laufzeitstack angelegt
11-4         * und bleibt dort aktiv bis zum Ende des main-Blocks
11-5         */
11-6         Tier t1 = new Tier();
11-7
11-8     {
11-9         /* Die Referenz t2 wird auf dem Laufzeitstack angelegt
11-10        * und wird geloescht, wenn dieser Block beendet wird.
11-11        * Danach ist das ueber t2 referenzierte Objekt nicht mehr erreichbar,
11-12        * da t2 vom Laufzeitstack geloescht wird..
11-13        */
11-14         Tier t2 = new Tier();
11-15     }
11-16
11-17     /* t2 existiert an dieser Stelle nicht mehr.
11-18     * Damit ist das urspruenglich ueber t2 referenzierte Tierobjekt
11-19     * zu einer Leiche im Heap geworden.
11-20     */
11-21 }
11-22 }
```

## 11.5 Klassenvariablen und -methoden

Im bisherigen Konzept ist der Attributwert (im Gegensatz zum Attribut selber) an ein konkretes Objekt gebunden, an eine Instanz der Klasse. Die Variablen, die ja die Umsetzung der Attribute in einer Programmiersprache wie Java sind, heißen folgerichtig auch **Instanzvariablen** und können erst einen Wert zugewiesen bekommen, wenn auch das Objekt mittels `new` erzeugt wurde.

Manchmal kann es aber auch sinnvoll sein, ein Attribut *und* dessen Attributwert mit der Klasse und nicht mit einem einzelnen Objekt zu assoziieren. Ein Beispiel dazu wäre etwa, die Anzahl *aller* erzeugten Objekte einer Klasse zu zählen. Es würde wenig Sinn machen, einen solchen Zähler an einem (welchem?) Objekt zu verankern. Und auch allen Objekten einer Klasse einen solchen Zähler zu geben, macht wenig Sinn. Statt dessen soll ja gerade über alle Objekte hinweg gezählt werden, inklusive der Möglichkeit, dass der Zähler 0 ist und überhaupt kein Objekt der Klasse existiert.

In Java wird eine Bindung einer Variablen an die Klasse (statt an eine Instanz) dadurch erreicht, dass man der Attributdefinition das Schlüsselwort `static` voranstellt. Dies bewirkt, dass dieses Attribut nun eine **Klassenvariable** ist.

### Beispiel 11.7:

Im Beispiel in Listing 11.10 wird bei der Erzeugung eines neuen Tieres im Konstruktor *automatisch* durch das Programm eine neue eindeutige Tiernummer vergeben. Über ein Attribut der *Klasse tier* (und nicht einer Instanz dieser Klasse) erhält man damit die nächste noch nicht vergebene Nummer. Mit jedem neuen erzeugten Tier wird diese Nummer erhöht.

In Abbildung 11.5 existieren die *Instanzvariablen* `nummer`, `name`, `gehege` pro Objekt einmal und haben dort individuelle Werte (Tier Nummer 1 mit Namen Gerti,...). Die *Klassenvariable* `neueNummer` existiert nur einmal und ist der Klasse zugeordnet, und nicht irgendeinem Objekt. Diese Klassenvariable `neueNummer` ist andererseits aber auch jedem Objekt dieser Klasse "bekannt". ♦

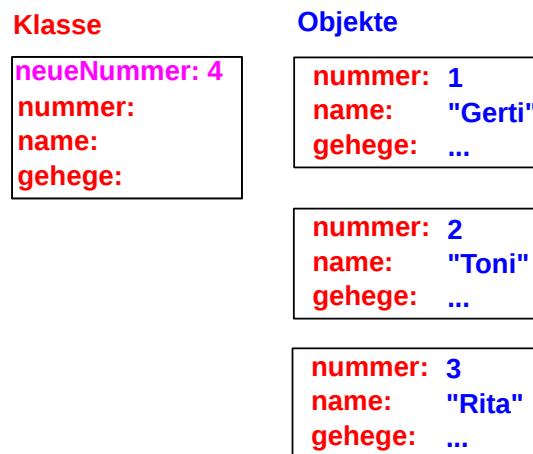


Figure 11.5: Klassen- und Instanzvariablen

Listing 11.10: Beispiel zu Klassenvariablen.

```

11-1 class Tier {
11-2
11-3     // Klassenvariablen
11-4     static int neueNummer = 0;    // Zaehler fuer Tiere
11-5     ...
11-6
11-7     Tier(String name) {
11-8         // vergabe automatisch eine neue Nummer
11-9         nummer = ++neueNummer;
11-10        this.name = name;
11-11    }
11-12    ...
11-13 }
```

Listing 11.11: Beispiel zur Nutzung von Klassenvariablen und -methoden.

```

11-1 // Frage naechste freie Gehegenummer an (Klassenmethode wegen static)
11-2 static int getNaechsteFreieNummer() { ... }
11-3
11-4 // Veraendere naechste freie Gehegenummer (Klassenmethode wegen static)
11-5 static void setNaechsteFreieNummer(int nummer) { ... }

```

Ebenso wie man zwischen Instanz- und Klassenvariablen unterschiedet, kann man auch zwischen **Klassen-** und **Instanzmethoden** unterscheiden. Eine Instanzmethode ist an die Existenz eines (des) Objektes geknüpft, eine Klassenmethode ist unabhängig von der Existenz eines Objektes dieser Klasse. Analog wie bei den Klassenattributen geschieht dies in Java auch hier durch das Voranstellen des Schlüsselwortes **static** in der Methodendefinition innerhalb einer Klasse. Eine Methode mit dem Schlüsselwort **static** (und damit Klassenmethode unabhängig von der Existenz eines Objekts dieser Klasse) wurde in früheren Beispielen schon oft verwendet: Die Methode

```

11-1 public static void main(String [] args)

```

ist ein Beispiel dafür.

Ein weiteres Beispiel ist die Methode **Math.sin** und weitere Methoden in der Klasse **Math**, die unabhängig von einer Objektinstanz der Klasse **Math** genutzt werden können (und sollen). In dieser Klasse **Math** sind auch Beispiele für Klassenvariablen zu finden, etwa **Math.PI** für eine bezüglich der Darstellungsgenauigkeit gute Annäherung an den Wert  $\pi$ .

Klassenmethoden machen zum Beispiel dann Sinn, wenn man den Wert von Klassenattributen abfragen oder verändern will. Ein Beispiel dazu ist in Listing 11.11 gegeben.

Es gibt jetzt also drei verschiedene Arten von Variablen mit jeweils eigener Charakteristik: Die Eigenschaften und Unterschiede von Instanz- und Klassenvariablen / -methoden sowie blocklokalen Variablen lassen sich wie folgt zusammenfassen.

- **Blocklokale Variablen** werden innerhalb eines Blocks deklariert. Sie haben einen Gültigkeitsbereich des umfassenden Blocks und eine Lebensdauer, die auf die Laufzeit des Blocks beschränkt ist (vergleiche Lebensdauer in Kapitel 6.4). Sie werden implizit nicht initialisiert. Im Programm werden sie zur temporären Speicherung von Werten genutzt (zum Beispiel Zwischenresultate). Eine blocklokale Variable ist weder mit einem Objekt noch mit einer Klasse assoziiert.
- **Instanzvariablen** werden innerhalb einer Klasse und außerhalb einer Methode ohne das Schlüsselwort **static** deklariert (üblicherweise zu Beginn einer Klasse). Sie haben einen Gültigkeitsbereich der Klasse und die Lebensdauer ist an die Lebensdauer des Bezugsobjekts gekoppelt. Sie werden implizit mit dem Nullwert des Datentyps initialisiert. Instanzvariablen speichern den Zustand eines Objekts. Eine deklarierte Instanzvariable existiert für jedes erzeugte Objekt.
- **Klassenvariablen** werden innerhalb einer Klasse und außerhalb einer Methode mit dem Schlüsselwort **static** deklariert (üblicherweise zu Beginn einer Klasse). Sie haben einen Gültigkeitsbereich der Klasse und eine Lebensdauer, die sich vom Laden der Klasse durch die Java Virtual Machine bis zum Programmende erstreckt. Der Fakt ist dies die gesamte Programmlaufzeit. Sie werden implizit mit dem Nullwert des Datentyps initialisiert. Sie werden genutzt, um Instanz-übergreifende Daten zu speichern. Klassenvariablen existieren nur genau einmal und sind unabhängig von einem Objekt.
- **Instanzmethoden** werden ohne das Schlüsselwort **static** deklariert.
- **Klassenmethoden** werden mit dem Schlüsselwort **static** deklariert.

- Instanzvariablen und -methoden brauchen *immer* ein Bezugsobjekt. Dies kann explizit angegeben werden. Beispiel: `new Klassenname().y` oder `t1.getName()` bzw. ergibt sich implizit aus einem Bezugsobjekt (zum Beispiel innerhalb einer Instanzmethode).

Auf Klassenvariablen und Klassenmethoden hat man unabhängig von einem Bezugsobjekt Zugriff. Die Referenzierung kann über den einfachen Namen innerhalb des Klasse erfolgen oder über den Klassennamen als Präfix (Beispiel: `Klassenname.klassenmethode()`) oder auch über ein Objekt dieser Klasse (Beispiel: `new Klassenname().klassenmethode()`). Weiterhin sind aber auch Zugriffsrechte zu beachten, wenn von außerhalb einer Klasse auf Variablen und Methoden zugegriffen wird (später mehr dazu).

### **Beispiel 11.8:**

Zum letzten Punkt sind in Listing 11.12 eine Auswahl an Möglichkeiten und Beschränkungen aufgezeigt. Weitere Auswirkungen darauf (Einschränkungen) haben Schlüsselwörter zur Zugriffskontrolle, auf die später in Kapitel 12.4 eingegangen wird.

1. Innerhalb der Klasse `Unterschied` hat man in der Instanzmethode `methodeInstanz` über den einfachen Namen `variableInstanz` Zugriff auf die entsprechende Instanzvariable. Innerhalb der Klassenmethode `methodeKlasse` ist das *nicht* möglich, weil das entsprechende Bezugsobjekt fehlt. Auch etwa die Verwendung von `this` innerhalb der Klassenmethode macht keinen Sinn und ist nicht erlaubt.
2. Innerhalb der Klasse `Unterschied` hat man sowohl in der Instanzmethode `methodeInstanz` als auch in der Klassenmethode `methodeKlasse` über den einfachen Namen `variableKlasse` Zugriff auf die entsprechende Klassenvariable.
3. Analog und aus den gleichen Gründen ließe sich in der Instanzmethode die Klassenmethode über ihren einfachen Namen aufrufen, aber nicht umgekehrt.
4. In einer anderen Klasse `AndereKlasse` muss man über ein Objekt auf eine Instanzvariable zugreifen, indem man `Objektreferenz.name` angibt, also zum Beispiel `u.variableInstanz = 7`.
5. Über eine Objektreferenz oder den Klassennamen kann man auf eine Klassenvariable zugreifen; im Beispiel `u.variableKlasse = 8` oder `Unterschied.variableKlasse = 6`
6. Analog kann man eine Instanzmethode in einer anderen Klasse nur über eine Instanz referenzieren (Beispiel `u.methodeInstanz()`). Eine Klassenmethode kan man sowohl über eine Instanz (Beispiel `u.methodeKlasse()`) als auch über den Klassennamen aufrufen (Beispiel `Unterschied.methodeKlasse()`). ◆

Listing 11.12: Beispiel zu Zugriffsmöglichkeiten von Instanz-/Klassenvariablen/-methoden.

```

11-1 /**
11-2 * Unterschied Klassen-/Instanzvariable
11-3 */
11-4 public class Unterschied {
11-5
11-6     public static int variableKlasse = 1;      // Klassenvariable
11-7     public       int variableInstanz = 2;      // Instanzvariable
11-8
11-9     public void methodeInstanz() {             // Instanzmethode
11-10        int blockLokal;                      // blocklokale Variable
11-11
11-12        variableInstanz = 3;                  // Zugriff auf Instanzvariable
11-13        variableKlasse = 4;                  // Zugriff auf Klassenvariable
11-14
11-15        methodeInstanz();                   // Zugriff auf Instanzmethode
11-16        methodeKlasse();                   // Zugriff auf Klassenmethode
11-17    }
11-18
11-19    public static void methodeKlasse() {       // Klassenmethode
11-20        int blockLokal;                      // blocklokale Variable
11-21
11-22        // hier ist kein Zugriff auf Instanzvariable/-methode moeglich!
11-23
11-24        variableKlasse = 5;                  // Zugriff auf Klassenvariable
11-25        methodeKlasse();                   // Zugriff auf Klassenmethode
11-26    }
11-27}
11-28
11-29
11-30 // andere Klasse
11-31 class AndereKlasse {
11-32     public static void main(String[] args) {
11-33         // ueber Klassennamen
11-34         Unterschied.variableKlasse = 6;
11-35         Unterschied.methodeKlasse();
11-36
11-37         // ueber Instanz
11-38         Unterschied u = new Unterschied();
11-39         u.variableInstanz = 7;
11-40         u.variableKlasse = 8;
11-41         u.methodeInstanz();
11-42         u.methodeKlasse();
11-43     }
11-44 }
```

## 11.6 Datenkapselung

Eine Klasse mit ihren Attributen und Methoden entspricht im Grunde einem Abstrakten Datentyp, wie er bereits in Kapitel 10 vorgestellt wurde. In einer Klasse werden nun alle Informationen zu gleichartigen Objekten im Rahmen des Abstraktionsprozesses gebündelt. Man spricht auch von **Datenkapselung** (Englisch: *Data Encapsulation*), da alle Informationen zu einer Klasse – Attribute und Methoden entsprechend Daten und Operationen – an einer Stelle zusammengefasst sind. Ist der interne Zustand von Objekten einer Klasse (die Attributwerte) und die interne Repräsentation von Daten nach außen nicht sichtbar, spricht man von **Information Hiding**.

Nun ist es aber durchaus sinnvoll, ein Objekt nach einem Attributwert zu fragen oder ein Objekt zu veranlassen, einen Attributwert zu modifizieren. Im Beispiel möchte man von einem Gehegeobjekt dessen Nummer wissen oder man möchte eventuell die Nummer eines Gehegeobjektes nachträglich ändern. Da Attribute bzw. Attributwerte den schützenswerten inneren Zustand eines Objektes darstellen und deshalb anderen Objekten kein direkter Zugriff auf diese Attribute gewährt werden soll, müssen solche Anfragen über Methoden geschehen, die ja die Schnittstelle eines Objektes / einer Klasse nach außen darstellen (Abbildung 11.6).

Im Zusammenhang mit der Datenkapselung gibt es eine Übereinkunft bezüglich der Namensgebung von Methoden, die Attributwerte von solchen Attributen erfragen und modifizieren, die nach außen über Methoden sichtbar sein sollen (sogenannte **Properties**). Um den Attributwert eines Attributs  $P$  vom Typ  $T$  zu erfragen, definiert man in der Klasse eine Methode

```
11-1 public T getP()
```

Ebenfalls kann man eine Methode

```
11-1 public void setP(T name)
```

zur Verfügung stellen, wenn der Attributwert verändert werden soll. Die Namensgebung ist also den Java Code Conventions folgend ein kleingeschriebenes **get**/**set** gefolgt von dem Attributnamen, allerdings beginnend mit einem Großbuchstaben. Diese Namensgebung wird zum Beispiel intensiv genutzt bei **Java Beans**, einer Java Komponententechnologie.

In der Klasse **Gehege** geschieht dies zum Beispiel mit den Methoden

```
11-1 public int getNummer() { ... }
11-2 public void setNummer(int neuenummer) { ... }
```

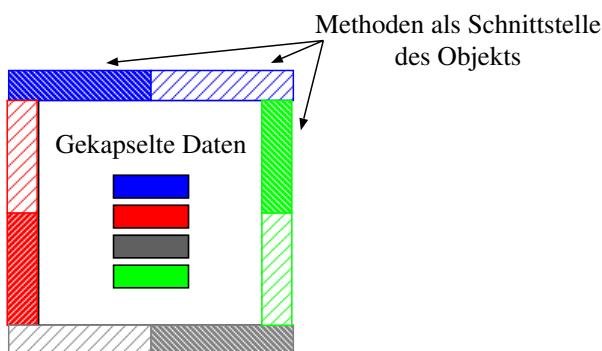


Figure 11.6: Datenkapselung

## 11.7 Komplette Beispiele

An dieser Stelle werden zum zusammenhängenden Verständnis des bisher Gesagten zwei komplette Beispiele angegeben.

### 11.7.1 Komplettes Zoo-Beispiel

In den Listings 11.13 bis 11.15 wird das vollständige Java-Programm für das Zoobeispiel gezeigt.

Listing 11.13: Klasse Zoo.

```

11-1 /**
11-2 * Zoo
11-3 */
11-4 public class Zoo {
11-5     public static void main(String [] args) {
11-6
11-7         Tier t1 = new Tier();
11-8         Tier t2 = new Tier("Gerti");
11-9
11-10        // Standardgehege
11-11        Gehege g1 = new Gehege();
11-12        // Grossgehege
11-13        Gehege g2 = new Gehege(10);
11-14
11-15        // Tier 1 zieht in Gehege 1 ein
11-16        t1.umziehen(g1);
11-17        t2.umziehen(g1);
11-18
11-19        System.out.println(t1);
11-20        System.out.println(t2);
11-21        System.out.println(g1);
11-22        System.out.println(g2);
11-23    }
11-24 }
```

Listing 11.14: Klasse Gehege.

```

11-1 /**
11-2 */
11-3 public class Gehege {
11-4     // Klassenvariablen
11-5     // Zaehler fuer Gehege
11-6     private static int vergebeneNummern = 0;
11-7     // Instanzvariablen
11-8     private int nummer;           // eindeutige Nummer
11-9     private int maximalBelegung; // max. Anzahl an Tieren in Gehege
11-10    private int aktuelleBelegung; // derzeitige Anzahl
11-11    private Tier[] bewohner;     // derzeitige Tiere
11-12
11-13    // Konstruktor fuer Standardgehege
11-14    public Gehege() {
11-15        // nutze anderen Konstruktor mit Defaultgroesse
11-16        this(5);
11-17    }
11-18
11-19    // Konstruktor fuer Gehege mit einer vorgegebenen Maximalbelegung
11-20    public Gehege(int maximalBelegung) {
11-21        // vergabe Nummer und erhoehe den Zaehler fuer vergeben Nummern (Klassenvariable)
```

```

11-22     nummer = ++vergebeneNummern;
11-23     this.maximalBelegung = maximalBelegung;
11-24     bewohner = new Tier[maximalBelegung];
11-25 }
11-26
11-27 // liefere Nummer des Geheges
11-28 public int getNummer() { return nummer; }
11-29
11-30 // liefere Maximalbelegung
11-31 public int getMaximalBelegung() { return maximalBelegung; }
11-32
11-33 // liefere aktuelle Belegung
11-34 public int getAktuelleBelegung() { return aktuelleBelegung; }
11-35
11-36 // Ein Tier zieht ein. Es muss noch Platz im Gehege sein!
11-37 public void einziehen(Tier tier) {
11-38     // freien Platz suchen
11-39     for(int i=0; i<maximalBelegung; i++) {
11-40         if(bewohner[i] == null) {
11-41             bewohner[i] = tier;
11-42             break;
11-43         }
11-44     }
11-45     // ein Tier mehr
11-46     ++aktuelleBelegung;
11-47     // beim Tier muss diese Information konsistent gehalten werden
11-48 }
11-49
11-50 // Ein Tier zieht aus. Es muss im Gehege vorhanden sein!
11-51 public void ausziehen(Tier tier) {
11-52     // Tier suchen
11-53     for(int i=0; i<maximalBelegung; i++) {
11-54         if(bewohner[i] == tier) {
11-55             bewohner[i] = null;
11-56             break;
11-57         }
11-58     }
11-59     // ein Tier weniger
11-60     --aktuelleBelegung;
11-61     // beim Tier muss diese Information konsistent gehalten werden
11-62 }
11-63
11-64 // liefere String-Darstellung eines Geheges
11-65 public String toString() {
11-66     return "Gehege " + nummer + ", maximal=" + maximalBelegung
11-67     + ", aktuell=" + aktuelleBelegung;
11-68 }
11-69 }
```

Listing 11.15: Klasse Tier.

```

11-1 /**
11-2 */
11-3 public class Tier {
11-4     // Klassenvariablen
11-5     private static int neueNummer = 0; // Zaehler fuer Tiere
11-6     // Instanzvariablen
11-7     private int nummer;           // eindeutige Nummer des Tieres
11-8     private String name;          // Name des Tieres (evtl. nicht vorhanden)
```

```

11-9   private Gehege gehege;           // Gehege, in dem es wohnt
11-10
11-11   // Konstruktoren
11-12   // parameterloser Konstruktor fuer ein Tier ohne Namen
11-13   public Tier() {
11-14       // rufe anderen Konstruktor auf, der alles weitere macht
11-15       this("unbekannt");
11-16   }
11-17
11-18   // Erzeuge ein neues Tier mit Namen
11-19   public Tier(String name) {
11-20       // vergabe automatisch eine neue Nummer
11-21       nummer = ++neueNummer;
11-22       this.name = name;
11-23       // gehege ist bereits null
11-24   }
11-25
11-26   // Methoden der Klasse
11-27   // liefere Name des Tieres
11-28   public String getName() { return name; }
11-29
11-30   // aendere Name des Tieres
11-31   public void setName(String name) { this.name = name; }
11-32
11-33   // ein Tier zieht in ein anderes Gehege um
11-34   public void umziehen(Gehege gehege) {
11-35       // altem Gehege Auszug mitteilen
11-36       if(this.gehege != null)
11-37           this.gehege.ausziehen(this);
11-38
11-39       // neuem Gehege Einzug mitteilen
11-40       if(gehege != null)
11-41           gehege.einziehen(this);
11-42
11-43       // aktuelles Gehege merken
11-44       this.gehege = gehege;
11-45   }
11-46
11-47   // Klassenmethode: liefere Anzahl bis jetzt erzeugter Tiere
11-48   public static int anzahlTiere() { return neueNummer; }
11-49
11-50   // liefere String-Darstellung eines Tiers
11-51   public String toString() {
11-52       return "Tier " + getName()
11-53           + ", Gehege=" + ((gehege == null) ? "ohne Gehege" : nummer);
11-54   }
11-55 }
```

### 11.7.2 Komplettes Beispiel zu einfachen geometrischen Objekten

In den Listings 11.16 bis 11.18 ist ein weiteres vollständiges Java-Programm für einfache geometrische Objekte (Punkt und Linie im zweidimensionalen Raum) angegeben.

Listing 11.16: Klasse Punkt.

```

11-1 /**
11-2 * Punkt in 2D
11-3 */
```

```

11-4 public class Punkt {
11-5     // Instanzvariablen: Koordinaten des Punktes
11-6     private double x;
11-7     private double y;
11-8     // Klassenvariable
11-9     private static int anzahlPunkte;
11-10
11-11     // parameterloser Konstruktor
11-12     public Punkt() {
11-13         // rufe anderen Konstruktor mit Default-Werten auf
11-14         this(0.0, 0.0);
11-15     }
11-16
11-17     // Konstruktor
11-18     public Punkt(double x, double y) {
11-19         this.x = x;
11-20         this.y = y;
11-21         // ein Punkt mehr
11-22         anzahlPunkte++;
11-23     }
11-24
11-25     public double getX() {
11-26         return x;
11-27     }
11-28
11-29     public double getY() {
11-30         return y;
11-31     }
11-32
11-33     // verschiebe Punkt
11-34     public void move(Punkt p) {
11-35         x += p.x;
11-36         y += p.y;
11-37     }
11-38
11-39     // erzeuge String-Darstellung
11-40     public String toString() {
11-41         return "(" + x + "," + y + ")";
11-42     }
11-43
11-44     // Klassenmethode
11-45     public static int getAnzahlPunkte() {
11-46         return anzahlPunkte;
11-47     }
11-48 }
```

Listing 11.17: Klasse Gerade.

```

11-1 /**
11-2 * Gerade
11-3 */
11-4 public class Gerade {
11-5     // 2 Endpunkte der Geraden
11-6     private Punkt p1;
11-7     private Punkt p2;
11-8
11-9     // Konstruktor
11-10    public Gerade(Punkt p1, Punkt p2) {
11-11        this.p1 = p1;
```

```

11-12     this.p2 = p2;
11-13 }
11-14
11-15 // verschiebe Gerade
11-16 public void move(Punkt p) {
11-17     p1.move(p);
11-18     p2.move(p);
11-19 }
11-20
11-21 // erzeuge String-Darstellung
11-22 public String toString() {
11-23     return p1.toString() + " - " + p2.toString();
11-24 }
11-25 }
```

Listing 11.18: Klasse GeometrieTest (Version 1).

```

11-1 /**
11-2 * Testprogramm fuer geometrische Objekte (Version 1)
11-3 */
11-4 public class GeometrieTest {
11-5     public static void main(String[] args) {
11-6         // 2 Punkte erzeugen
11-7         Punkt p1 = new Punkt(0.0, 0.0);
11-8         Punkt p2 = new Punkt(1.0, 1.0);
11-9         Punkt p3 = new Punkt(3.0, 3.0);
11-10
11-11         // eine Gerade erzeugen
11-12         Gerade g = new Gerade(p1, p2);
11-13         System.out.println(g);
11-14
11-15         // Gerade verschieben
11-16         g.move(p3);
11-17         System.out.println(g);
11-18
11-19         // Klassenmethode aufrufen
11-20         System.out.println("Anzahl erzeugter Punkte: " + Punkt.getAnzahlPunkte());
11-21     }
11-22 }
```

## 11.8 Zusammenfassung und Hinweise

### Literaturhinweise

Es gibt zahlreiche Bücher mit einführendem Charakter zur Objektorientierten Programmierung, etwa [Sch10]. Umfassende Lehrbücher zu Java mit Beispielen sind [HMHG11] und [Ull12]. In [SW11] werden viele praktische Beispiele zu Klassen aufgeführt und eine Sammlung von Reflektionsfragen (mit Antworten) angegeben.

Im Zusammenhang mit Klassen sei ein **record** nur erwähnt. Dies ist eine Art Typdeklaration analog einer sehr beschränkten Klasse. Ein **record** wird dazu genutzt, Daten zu kapseln, auf die man nur lesend zugreift.

### Beispiel 11.9:

Gegeben sei folgende Klasse (das Beispiel ist aus der Java Dokumentation entnommen und nimmt einige Sprachkonstrukte vorweg, die später erst erläutert werden):

```

11-1 final class Rectangle implements Shape {
11-2     final double length;
11-3     final double width;
11-4
11-5     public Rectangle(double length, double width) {
11-6         this.length = length;
11-7         this.width = width;
11-8     }
11-9
11-10    double length() { return length; }
11-11    double width() { return width; }
11-12 }
```

Diese Klasse kann in Form eines *record* wie folgt notiert werden:

```

11-1 record Rectangle(float length, float width) {
11-2 }
```

Durch diesen **record** werden automatisch ein entsprechender Konstruktor sowie pro Instanzvariable eine Getter-Funktion generiert (allerdings ohne get im Namen).

## Verstehen

Klassen dienen der Abstraktion und beschreiben deklarativ gleichartige Objekte (der realen Welt). Man muss also einmal eine Klasse beschreiben / programmieren und kann dann daraus beliebig viele Objekte / Instanzen dieser Art erzeugen.

## Kurz und knapp merken

- In der Objektorientierten Modellierung identifiziert man beteiligte Objekte. Daraus ermittelt man gleichartige Objekte, die man über eine Klasse beschreiben kann.
- Die Eigenschaften in einer Klasse sind entweder direkt durch Java-Typen spezifizierbar oder man muss wiederum geeignete Beschreibungen finden, zum Beispiel über eine weitere beschreibende Klasse.
- In einer Java-Klasse werden Instanzvariablen und -methoden angegeben, die den (aktuellen) Zustand eines Objekts angeben / speichern und manipulieren.
- Klassenvariablen und -methoden dienen der Speicherung und Verwaltung von nicht-objektspezifischen Daten, die aber aufgrund des Information Hiding dort angesiedelt werden, wo sie logisch hingehören.
- Instanzvariablen und -methoden benötigen immer ein Bezugsobjekt.
- Aus einer Klasse lassen sich mit Hilfe des **new**-Operators Objekte einer Klasse erzeugen.
- In den Erzeugungsprozess kann man über Konstruktoren eingreifen.

## Häufige Fehler

- In einer Klassenmethode (also mit **static** angegeben) versucht man auf eine Instanzvariable zuzugreifen oder eine Instanzmethode aufzurufen.

- Klassenvariablen dienen der Aufnahme eines Wertes, der unabhängig von einer Instanz ist. Instanzdaten kann man also nicht in einer Variablen abspeichern, die mit `static` deklariert ist.
- Sobald man selber mindestens einen Konstruktor angegeben hat, gibt es den Default-Konstruktor dieser Klasse nicht mehr.
- Man muss zwischen einem (anonymen) Objekt und einer Referenz auf ein Objekt unterscheiden.

## Übungsfragen

- Welche Rolle spielen Klassen in der Softwareentwicklung?
- Wofür stehen Attribute in der Software-Modellierung?
- Wofür stehen Methoden in der Software-Modellierung?
- Was kann alles in einer Klasse stehen?
- Wie kommt man beim Software-Entwurf auf Attribute einer Klasse?
- Wie kommt man beim Software-Entwurf aus Textbeschreibung auf Methoden einer Klasse?
- Wozu dient eine Instanzvariable und wie gebe ich sie an?
- Wozu dient eine Klassenvariable und wie gebe ich sie an?
- Was ist der Gültigkeitsbereich und die Lebensdauer einer Instanz- / Klassenvariablen?
- Was bedeutet genau: `class K { int x = 3; K2 y = new K2(); float[] z = new float[0]; }`
- Was ist eine Instanzmethode und wie gebe ich eine an?
- Was ist eine Klassenmethode und wie gebe ich eine an?
- Was ist eine Instanziierung? Beispiel!
- Was ist das Ergebnis von `new Klasse1()`?
- Was geschieht genau bei `new K()`?
- Wann werden Instanzvariablen wie initialisiert, wann/wie Klassenvariablen?
- Was ist die Lebensdauer des Objekts `new K()`?
- Wozu dient ein garbage collector?
- Wozu dienen Konstruktoren?
- Wie kann ich Konstruktoren angeben?
- Muss / kann ich 0, 1, mehre Konstruktore angeben?
- Wenn ich keinen Konstruktor angebe, was bedeutet das?
- Kann ein Konstruktor überladen werden? Überschrieben?
- Wenn ich mehrere Konstruktoren angebe: worauf muss ich achten?
- Was ist ein Default-Konstruktor?
- Wenn eine Klasse mehrere Konstruktoren hat: Wie wird der richtige ausgewählt bei `new`?
- Was passiert genau bei: `class K { int x; main(...){ new K(); }}`
- Welche prinzipiellen Möglichkeiten gibt es eine Klassenmethode aufrufen?
- Welche prinzipiellen Möglichkeiten gibt es eine Instanzmethode aufrufen?
- Auf was in der eigenen Klasse habe ich mit einfacherem Namen in einer Instanzmethode Zugriff?
- Auf was in der eigenen Klasse habe ich mit einfacherem Namen in einer Klassenmethode Zugriff?
- Kann ich (und wie) von außerhalb der Klasse auf Instanz- / Klassen- / block-lokale Variablen zugreifen?
- Welche Variablen(-arten) haben nach der Deklaration welche Werte?

## Reflektion des Stoffs

Modellieren Sie folgendes Problem zuerst abstrakt und darauf aufbauend anschließend konkret in Java. Die abstrakte Modellierung sollte also ergeben, was die Objekte sind und was für Attribute und Aktivitäten dabei

aufreten.

Modellieren Sie eine Obsthandlung, die Bananen und Äpfel verkauft. Bananen haben einen Krümmungsradius, ein Gewicht und einen Preis, der vom Gewicht abhängig ist. Der Kilopreis für Bananen ist 2 Euro. Äpfel haben einen Durchmesser, eine Farbe, ein Gewicht und einen gewichtsabhängigen Preis. Der Kilopreis für Äpfel ist 3 Euro. In der Obsthandlung liegen 3 Bananen mit unterschiedlicher Krümmung und Gewicht (suchen Sie sich Werte dafür aus) und 4 Äpfel aus, auch mit unterschiedlichem Durchmesser und Gewicht. Ermitteln Sie den Verkaufspreis aller Bananen und Äpfel in der Obsthandlung.



# Chapter 12

## Vererbung

In diesem Kapitel wird das wichtige Konzept der Vererbung mit damit verbundene Begriffe erläutert.

### 12.1 Vererbung

Ein Patentier ist ein normales Tier mit einer zusätzlichen Eigenschaft: ein Jahreseinkommen. Man könnte jetzt eine Klasse `Patentier` entwerfen und alle Attribute und Methoden der Klasse `Tier` dort reinkopieren und anschließend die Attribute und Methoden hinzufügen, die ein Patentier *zusätzlich* gegenüber einem normalen Tier hat. Würde man allerdings später einmal die Tierklasse ändern, zum Beispiel durch Hinzufügen von Attributen oder Methoden, so müsste man dies jedesmal genau so auch in der Klasse `Patentier` nachtragen, um die Konsistenz zwischen den beiden Klassen zu gewährleisten.

#### 12.1.1 Das Prinzip der Vererbung

Der gerade beschriebene Ansatz wäre ein aufwändiger und noch schlimmer ein sehr fehleranfälliger Ansatz. Objektorientierte Sprachen wie Java haben einen anderen Mechanismus, um solche Erweiterungen zu handhaben: das Prinzip der **Vererbung**. Schaut man sich obige Beschreibung eines Patentieres an, so ist ein Patentier nichts anderes als ein normales Tier mit einigen *zusätzlichen* Eigenschaften, also eine Erweiterung des Tieres. Dementsprechend könnte man jetzt auch eine Klasse `Patentier` als Erweiterung der Klasse `Tier` ansehen. Und genauso wird dies auch konzeptionell und syntaktisch in objektorientierten Sprachen gehandhabt.

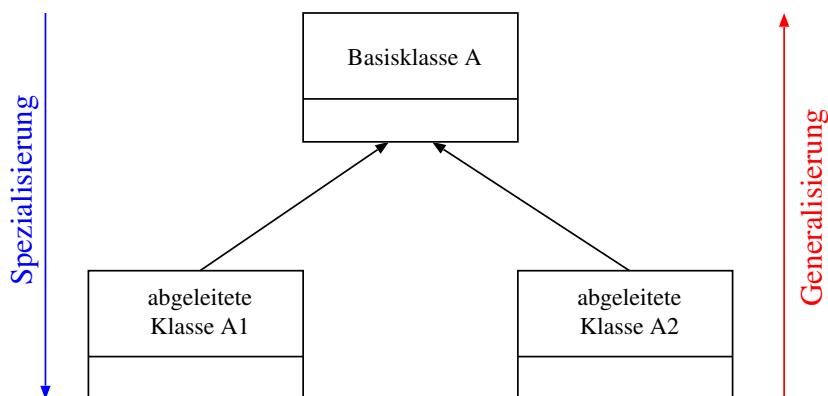


Figure 12.1: Prinzip der Vererbung.

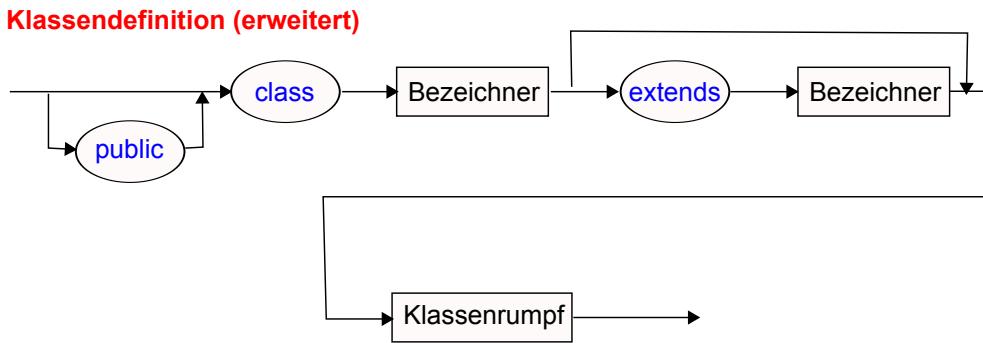


Figure 12.2: Erweitertes Syntaxdiagramm für eine Klasse.

Listing 12.1: Erste Version des Patentiers basierend auf der Tierklasse.

```

12-1 /**
12-2 * Patentier Version 1 (noch unvollstaendig)
12-3 */
12-4 class Patentier extends Tier {
12-5
12-6     // zusaetzliche Attribute und Methoden kommen hier hin
12-7 }
  
```

Das allgemeine Prinzip der Vererbung ist in Abbildung 12.1 zusehen. Aus einer beliebigen Basisklasse kann man durch eine **Spezialisierung** eine (oder mehrere) abgeleitete Klassen erhalten, die spezieller sind als die Basisklasse. In ihnen ist also mehr festgelegt als in der Basisklasse, die Funktionalität der Basisklasse wird in der neuen Klasse erweitert.

Der umgekehrte Weg von einer abgeleiteten Klasse zur Basisklasse nennt man **Generalisierung**. Man abstrahiert etwa im Beispiel von zwei abgeleiteten Klassen A1 und A2 zu einer gemeinsamen Basisklasse A, indem man unterschiedliche Merkmale der abgeleiteten Klassen weglässt und nur die *gemeinsamen* Attribute betrachtet, man also allgemeiner wird.

Das Unterscheidungsmerkmal, durch das die Hierarchiebildung zustande kommt, nennt man **Diskriminator**. Die Ursprungsklasse heißt **Basisklasse** oder **Oberklasse**, die erweiterte Klasse heißt **abgeleitete Klasse** oder **Unterklasse**.

### 12.1.2 Vererbung in Java

In Java wird eine abgeleitete Klasse in der Klassendefinition durch das Schlüsselwort `extends` gefolgt von dem Klassenbezeichner der Basisklasse gekennzeichnet. Die erweiterte Syntax zur Definition einer Klasse ist in Abbildung 12.2 zu sehen.

#### Beispiel 12.1:

Listing 12.1 zeigt eine erste Version für ein Patentierklassen, die von der Tierklasse abgeleitet ist. ♦

#### Beispiel 12.2:

Listing 12.2 zeigt eine Klasse `Punkt3D`, die von der Klasse `Punkt` (als 2D-Punkt) abgeleitet ist. ♦

Listing 12.2: Klasse Punkt3D.

```

12-1  /**
12-2   * Punkt in 3D
12-3   */
12-4 public class Punkt3D extends Punkt {
12-5     // zusätzliche Instanzvariable: dritte Koordinate
12-6     private double z;
12-7
12-8     // Default-Konstruktor
12-9     public Punkt3D() {
12-10        // Konstruktor der Oberklasse wird hier automatisch aufgerufen
12-11        z = 0;
12-12        // alternativ hier: this(0,0,0);
12-13    }
12-14
12-15     // Konstruktor
12-16     public Punkt3D(double x, double y, double z) {
12-17        // (richtigen) Konstruktor der Oberklasse aufrufen
12-18        super(x,y);
12-19        this.z = z;
12-20    }
12-21
12-22     public double getZ() {
12-23         return z;
12-24     }
12-25
12-26     // verschiebe Punkt
12-27     public void move(Punkt3D p) {
12-28        // verschiebe 2D Anteil
12-29        super.move(p);
12-30        // verschiebe zusätzlichen 3D Anteil
12-31        z += p.z;
12-32    }
12-33
12-34     // erzeuge String-Darstellung
12-35     public String toString() {
12-36         return "(" + super.getX() + "," + super.getY() + "," + z + ")";
12-37     }
12-38 }
```

Listing 12.3: Beispiel zum Gültigkeitsbereich bei Vererbung.

```

12-1 class Test1 {
12-2     int meineVariable1;
12-3     int meineVariable2;
12-4
12-5     Test1() {
12-6         meineVariable1 = 1;
12-7         meineVariable2 = 2;
12-8     }
12-9 }
12-10
12-11 class Test2 extends Test1 {
12-12     // eine Instanzvariable gleichen Namens wurde von Basisklasse geerbt
12-13     int meineVariable1;
12-14
12-15     Test2() {
12-16         /* an der Stelle ist die eigene Variable sichtbar (mit einfachen Namen ansprechbar)
12-17             und nicht die ererbte
12-18             */
12-19         meineVariable1 = 3;
12-20         // eine andere ererbte Variable kann aber ueber den einfachen Namen veraendert
12-21             werden
12-22         meineVariable2 = 4;
12-23     }
12-24 }
```

In Java besitzt / erbt die abgeleitete Klasse alle Methoden und Attribute der Basisklasse, die sogar über ihren einfachen Namen in der abgeleiteten Klasse referenzierbar sind. In Kapitel 12.4 werden dazu Einschränkungen angegeben (Zugriffsrechte). Die abgeleitete Klasse wird aber *erweitert* um die eigenen Attribute und Methoden der abgeleiteten Klasse, die der Basisklasse allerdings umgekehrt nicht bekannt sind ("‘Einbahnstrasse’").

Der Gültigkeitsbereich von Variablen und Methoden der Basisklasse wird also erweitert auf die abgeleitete Klasse. Damit ist es dann auch möglich, dass sowohl in der Basisklasse als auch in der abgeleiteten Klasse der gleiche Name verwendet wird (Variable oder Methode), wodurch die Sichtbarkeit festgelegt werden muss, da zwei Variablen / Methoden mit dem gleichen einfachen Namen *gleichzeitig* zwar gültig sein können, aber nur eine sichtbar sein kann. In einem solchen Kollisionsfall ist über den einfachen Namen immer die Variable / Methode der abgeleiteten Klasse gemeint (diese ist also nicht nur gültig, sondern auch sichtbar).

### Beispiel 12.3:

Im Programm in Listing 12.3 erbt die Klasse `Test2` von der Klasse `Test1`, in der zwei Instanzvariablen deklariert sind. Die Verwendung des einfachen Namens `meineVariable1` in `Test2` führt zu einem Problem, da sowohl die ererbte Variable der Basisklasse als auch eine neu deklarierte Variable der abgeleiteten Klasse diesen Namen besitzt. In diesem Fall wird die Variable der Basisklasse in der abgeleiteten Klasse unsichtbar (nicht mehr über den einfachen Namen ansprechbar) und nur die Variable der abgeleiteten Klasse ist sichtbar.♦

Von einem Objekt einer abgeleiteten Klasse kann man Bezug nehmen auf den Objektteil der (in Java eindeutigen) Basisklasse über das Schlüsselwort `super` (siehe Abbildung 12.3). `super` bezeichnet also ein Objekt. Eine Methode `f` der Basisklasse ist zum Beispiel explizit durch `super.f()` aufrufbar, zum Beispiel um bei Gleichheit der Signatur in Basisklasse und abgeleiteter Klasse die Methode der Basisklasse aufzurufen.

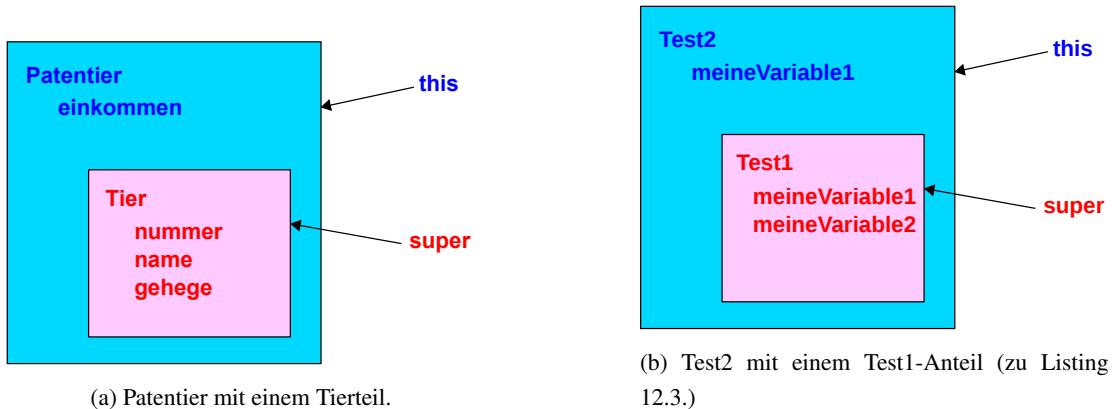


Figure 12.3: Objekte abgeleiteter Klassen mit Objektanteil der Basisklasse.

Analoges gilt für Instanz- und KlassenvARIABLEN. Im obigen Beispiel kann man also in der Klasse `test2` trotz Unsichtbarkeit (aber natürlich trotzdem Gültigkeit!) auf die geerbte Instanzvariable der Basisklasse Bezug nehmen mit `super.meineVariable1`. Während also `this` das gesamte Bezugsobjekt bezeichnet, so bezeichnet `super` nur den geerbten Teil des Objekts (ein Objekt), der sich aus der Basisklasse ergibt.

In Java kann man durch Voranstellen spezieller Schlüsselwörter in einer Attribut- oder Methodendefinition in der Basisklasse erreichen, dass in der abgeleiteten Klasse diese so gekennzeichneten Attribute und Methoden nicht sichtbar sind, was manchmal erwünscht sein kann. Die prinzipielle Möglichkeit dazu wurde bereits im Kapitel 11.6 vorgestellt.

In einem Konstruktor der abgeleiteten Klasse kann man selbst den parameterlosen Konstruktor der Oberklasse über `super()` bzw. mit Parameter über `super(...)` aufrufen. Dies muss dann die *erste* Anweisung in einem Konstruktor der abgeleiteten Klasse sein. Ruft man selbst als erste Anweisung in einem Konstruktor nicht einen Konstruktor der Basisklasse auf, so wird *implizit* der parameterlose Konstruktor der Basisklasse aufgerufen. Existiert dieser nicht (weil man zum Beispiel selbst einen anderen Konstruktor angegeben hat), so ist das dann ein Fehler.

#### Beispiel 12.4:

Betrachtet man das Tierbeispiel wieder, so kann man also eine Klasse `Patentier` als Erweiterung der Klasse `Tier` ansehen, ein Patentier ist eine Spezialisierung eines normalen Tieres. Und dementsprechend kann man aus einer Basisklasse `Tier` eine Klasse `Patentier` ableiten. Ein Patentier benötigt gegenüber der Definition eines normalen Tieres ein *zusätzliches* Attribut, in dem das Jahreseinkommen des Patentieres gespeichert ist.

In Listing 12.4 ist die Patentier-Klasse angegeben. Beim Erzeugen eines Patentieres wird jetzt zusätzlich das Jahreseinkommen angegeben. Im Konstruktor eines Patentieres muss also zuerst ein normales Tier erzeugt werden und zusätzlich muss das Jahsgehalt abgespeichert werden. Der Aufruf des Konstruktors der Basisklasse geschieht im ersten Patentier-Konstruktor (ohne Namen) implizit durch Weglassen eines expliziten Konstruktorauftrags der Basisklasse, während der Konstruktor der Basisklasse im zweiten Konstruktor explizit aufgerufen wird, um so dort den Namen der Tiere übergeben zu können.

Ähnlich wie beim Konstruktor kann auch bei der Erzeugung einer textuellen Darstellung in `toString` verfahren werden. Die Methode `toString()` eines Patentieres ist gleich der Methode `toString()` eines normalen Tieres und weiterhin der *zusätzlichen* Angabe des Jahreseinkommens. An der Stelle wird die Methode `toString` überschrieben (Erläuterung im nächsten Abschnitt) und die Methode der Basisklasse explizit mit `super.toString()` aufgerufen. Reflektionsfrage an der Stelle: Was passiert, wenn dort nur der Aufruf `toString()` ohne `super` stehen würde? ♦

Listing 12.4: Klasse Patentier.

```

12-1  /**
12-2   * Patentier
12-3   */
12-4 class Patentier extends Tier {
12-5
12-6     // alle Instanz- und Klassenvariablen der Basisklasse werden geerbt
12-7     int einkommen; // jaehrliches Einkommen
12-8
12-9     Patentier(int einkommen) {
12-10        // parameterloser Konstruktor super() der Oberklasse wird hier implizit aufgerufen
12-11        this.einkommen = einkommen;
12-12    }
12-13
12-14    Patentier(String name, int einkommen) {
12-15        // einen bestimmten Konstruktor der Oberklasse explizit aufrufen
12-16        super(name);
12-17        this.einkommen = einkommen;
12-18    }
12-19
12-20    // alle Methoden der Basisklasse werden geerbt
12-21
12-22    // Ueberschreiben der Methode toString der Basisklasse
12-23    public String toString() {
12-24        return super.toString() + ", Einkommen=" + einkommen;
12-25    }
12-26 }
```

### 12.1.3 Klassenhierarchien

Sieht man sich die Motivation für Vererbung und damit abgeleiteten Klassen noch einmal an, so ist eine logische Schlussfolgerung daraus, dass man von einer abgeleiteten Klasse wiederum andere Klassen ableiten kann, diese können wieder als Basisklasse weiterer Klassen dienen usw. Es entsteht also eine Hierarchie von Klassen von allgemeinen Objektbeschreibungen hin zu Beschreibungen von spezialisierten Objekten. Da eine abgeleitete Klasse in Java nur genau eine Basisklasse besitzen kann, entsteht damit ein Baum von abgeleiteten Klassen. Im Unterschied zu Binärbäumen – vergleiche Kapitel 10.3 – können die Knoten in allgemeinen Bäumen auch mehr, aber auch weniger als zwei Söhne haben.

Abbildung 12.4 zeigt ein Beispiel für eine Klassenhierarchie. Übliche Kraftfahrzeuge kann man (vereinfacht) unterteilen in LKW und Fahrzeuge zur Beförderung von Personen, der Diskriminator ist also in diesem Fall die Unterscheidung, ob Güter oder Personen transportiert werden sollen. Bei Personentransport wird anhand des Diskriminators "Anzahl Räder" noch weiter unterschieden in PKW und Motorräder. In der vorliegenden Unterteilung sind zum Beispiel die PKW noch weiter unterteilt in Sportwagen, Limousine, Coupe und Van. Je nach Aufgabenstellung können natürlich auch ganz andere Diskriminatoren und damit Ableitungshierarchien für Kraftfahrzeuge oder PKW Sinn machen. Alle Kraftfahrzeuge besitzen einen Motor, der eine bestimmte Leistung erbringen kann, die von Objekt zu Objekt verschieden sein kann (Attribut dieser Klasse). Für alle Kraftfahrzeuge gibt es eine Methode **MotorStarten()**, die den Motor startet. Bei den Unterklassen Motorrad, PKW und LKW sind jetzt ganz unterschiedliche Angaben von Relevanz. Für alle drei Unterklassen ist nach wie vor die Motorleistung als Attribut interessant und natürlich die Methode zum Starten des Motors (aus der Basisklasse geerbt), aber für einen LKW ist das Ladevolumen als Attribut wichtig, für ein Motorrad

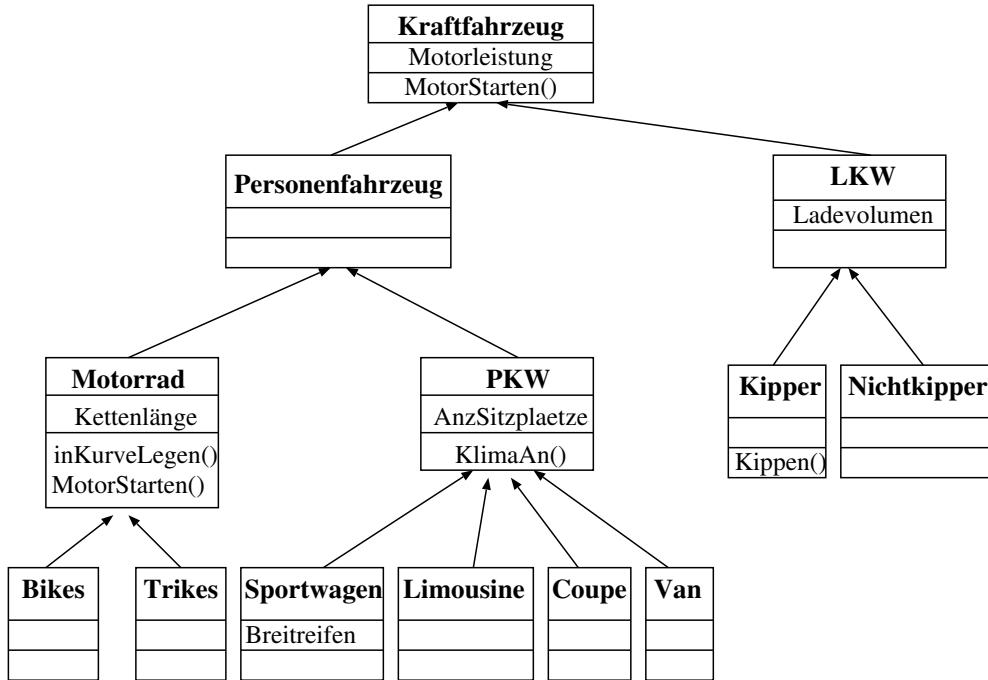
**Beispiel 12.5:**

Figure 12.4: Beispiel für eine Klassenhierarchie

eher weniger. Dafür kennt ein Motorrad die Methode `inKurveLegen()`, was eher uninteressant für einen LKW und einen PKW sein wird. Genauso macht die Methode `Kippen` nur Sinn bei einem Kipper-LKW. In der Motorradklasse wird die Methode `MotorStarten()` der Basisklasse überschrieben, zum Beispiel weil bei Motorrädern durch Anreten gestartet wird. ♦

Für eine Hierarchiebildung durch Spezialisierung gelten die folgenden Regeln:

- Is-A-Beziehung:** Ein Objekt der spezielleren Klasse ist aufgrund des Diskriminators ein besonderer Repräsentant der generelleren Klasse.
- Substitutionsregel:** Ein Objekt der spezielleren Klasse kann jederzeit anstelle eines Objekts der generelleren Klasse stehen. Die Umkehrung gilt nicht! Dieses Stellvertreterprinzip hat für objektorientierte Sprachen eine große Bedeutung und wird in Kapitel 12.5 eingehend besprochen.
- Klassifikationsprinzip:** Eine Spezialisierung führt zu Klassen von Objekten mit gleichem Verhalten und Eigenschaften, aber unterschiedlichen Eigenschaftswerten.

In Java (und weiteren Sprachen wie etwa Smalltalk) sind alle Klassen direkt oder indirekt von der Klasse `Object` abgeleitet, es entsteht ein Baum von Klassen mit einer eindeutigen Wurzel, die für *alle* Objekte gleich ist (Abbildung 12.6). Leitet man eine eigene Klasse nicht von einer weiteren Klasse ab, so ist die eigene Klasse *implizit* von der Klasse `Object` abgeleitet. Alle solchen Klassen haben also ein implizites `extends Object` in der Klassendefinition stehen.

Die Klasse `Object` (und damit *jede* Klasse in Java) besitzt einige Methoden, aus denen hier zwei näher vorgestellt werden:

- Die Methode `public String toString()` liefert für ein Objekt eine String-Darstellung. Die Methode in `Object` liefert per Default einen String der Form `java.lang.Object@3be67280`, wobei der Teilstring

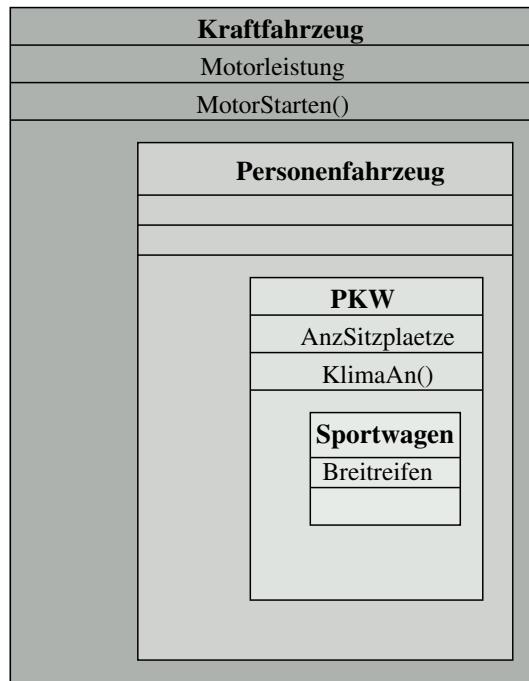


Figure 12.5: Veranschaulichung der Substitutionsregel

hinter `@` der Adresse im Hauptspeicher entspricht. Die Methode `toString` wird implizit aufgerufen, wenn zu einem Objekt ein String generiert werden muss, etwa bei `System.out.println("Ausgabe ist " + referenz)`, wobei `referenz` eine Referenz auf ein beliebiges Objekt ist. Sinnvollerweise wird man (falls diese Funktionalität genutzt wird) die Methode `toString` in einer abgeleiteten Klasse überschreiben (s.u.)

- Die Methode `clone()` erzeugt aus einem Objekt ein inhaltsgleiches weiteres Objekt. Dazu muss die Klasse allerdings die Schnittstelle `Cloneable` realisieren (siehe Kapitel 13.3). Eine Anwendung der `clone`-Funktionalität ist das Klonen von Feldern. Ein einfaches Beispiel dazu ist:

```

12-1   int [] a = {1,2,3};
12-2   int [] b = a.clone();
  
```

Es wird der Inhalt des Feldes `a` in ein weiteres neues Feld gleicher Größe kopiert. Anschließend gibt es also zwei Felder, die inhaltsgleich sind.

Eine einheitliche Basisklasse, von der letztendlich alle Klassen abgeleitet sind, ist für eine objektorientierte Sprache aber nicht zwingend, zum Beispiel gibt es in C++ keine "‘Mutter aller Klassen’" wie in Java.

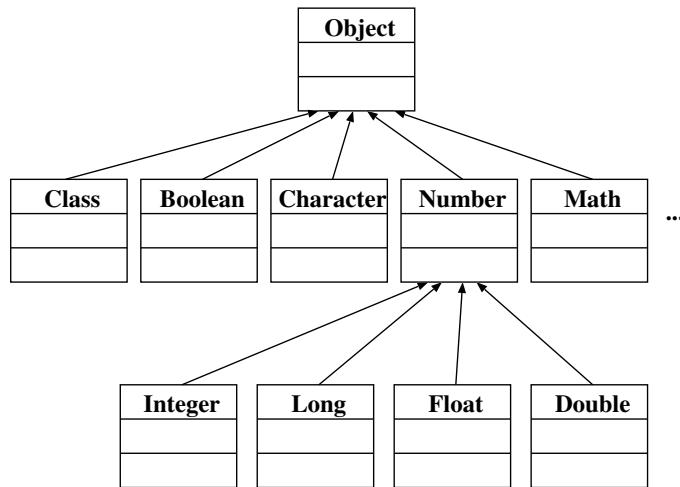


Figure 12.6: Kleiner Ausschnitt aus der Klassenhierarchie des Java SDK

## 12.2 Typanpassungen bei Referenzen

Bereits in Kapitel 7.6 wurde die Möglichkeit von Typanpassungen bei primitiven Typen besprochen, die implizit oder explizit durchgeführt wird. Dort wurden Typanpassungen eingesetzt, um einen Wert in einem Typ x unter Werterhalt bzw. genau definiertem (geringem) Wertverlust in einen ähnlichen Typ y umzuwandeln, um zum Beispiel im Typ y Operationen ausführen zu können. Dabei gab es eine unproblematische Richtung (erweiternde Umwandlung), die implizit oder explizit durchgeführt wird und bei der es zu keinem Wertverlust kommen kann (bis auf den Übergang zwischen ganzzahligen Typen auf Fließkommatypen). Weiterhin gab es eine problematische Richtung (einengende Typumwandlung), die nur explizit über einen Cast-Operator angegeben werden kann und bei der es zu evtl. erheblichen Wertänderungen kommen kann.

### Beispiel 12.6:

Das nachfolgende Programm zeigt noch einmal die prinzipiellen Möglichkeiten zur Typumwandlung bei primitiven Typen:

```

12-1 long l1 = 7;           // implizite erweiternde Typumwandlung
12-2 long l2 = (long)7;     // explizite erweiternde Typumwandlung
12-3
12-4 int i1 = (int)l1;      // explizite einengende Typumwandlung unter Werterhalt
12-5 int i2 = (int)(1L << 33); // explizite einengende Typumwandlung mit Wertaenderung
  
```

Im Zusammenhang mit Referenztypen sind ebenfalls Typanpassungen möglich. Auch hier muss man sich Gedanken machen, wozu solche Typanpassungen dienen sollen und wann sie überhaupt Sinn machen. Typanpassungen bei Referenztypen machen nur innerhalb einer Vererbungshierarchie Sinn. Denn dann kann man relativ unproblematisch von der Übertragbarkeit von Werten zwischen zwei Typen sprechen.

### Beispiel 12.7:

In der Klasse `String` gibt es die Instanzmethode `toUpperCase`, die zu einem Stringobjekt ein neues Stringobjekt liefert, in dem alle Kleinbuchstaben durch den entsprechenden Großbuchstaben ersetzt sind.

Was sollte es nun bedeuten, wenn man folgenden Code angeben würde:

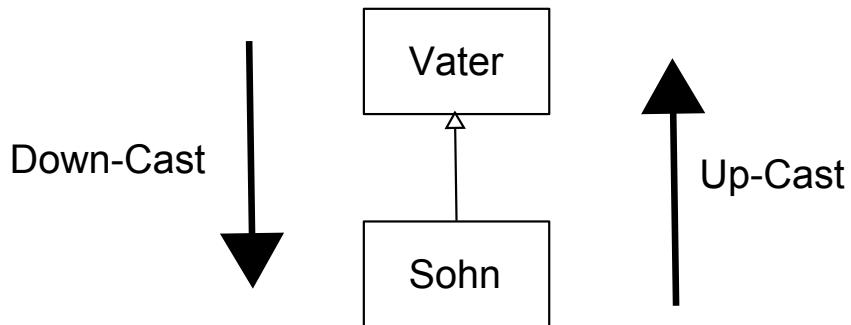


Figure 12.7: Up- und Down-Casts.

```

12-1 Tier t = new Tier();      // erzeuge Tier
12-2 ((String)t).toUpperCase() // caste Tier auf String und wende Methode an
12-3                                // Das ist Unsinn!

```

wo zwischen zwei Typen übertragen werden soll, die überhaupt nicht miteinander zu tun haben. ♦

Wenn eine Klasse X direkt oder indirekt von einer Klasse Y abgeleitet ist, so machen Umwandlungen von X nach Y und umgekehrt überhaupt nur Sinn.

Typanpassungen finden auch bei Referenztypen entweder implizit oder explizit über eine Cast-Operation statt. Prinzipiell sind zwei Richtungen möglich (**Vater** bezeichne eine Basisklasse, **Sohn** eine davon direkt oder indirekt abgeleitete Klasse; siehe Abbildung 12.7):

- **Up-Cast:** von einem spezielleren Typ **Sohn** (abgeleitete Klasse) zu einem allgemeineren Typ **Vater** (Basisklasse). Up-Casts sind problemlos möglich, wenn die beiden beteiligten Klassentypen in der angegebenen Vererbungsbeziehung zueinander stehen. Up-Casts können deshalb entweder implizit oder explizit über einen Cast durchgeführt werden.
- **Down-Cast:** von einem allgemeineren Typ **Vater** (Basisklasse) auf einen spezielleren Typ **Sohn** (abgeleitete Klasse). Down-Casts können *nur explizit* über einen Cast durchgeführt werden. Eine solche Operation ist nur zulässig, wenn die anzupassende Referenz auf ein Objekt der Klasse **Sohn** oder einer davon abgeleiteten Klasse zeigt. Ist dies nicht der Fall, so kommt es zu einem Laufzeitfehler (**ClassCastException**).

Abbildung 12.8 zeigt an einem Beispiel, was ein Up-/Down-Cast bewirkt. Durch eine Up-Cast Operation wird die Sicht auf ein Objekt eingeschränkt auf den Teil, den die Oberklasse definiert. Es ist sehr wichtig an dieser Stelle zu verstehen, dass das Objekt nach wie vor das gleiche Objekt bleibt, im Beispiel also ein Patentier-Objekt bleibt. Nur die Sicht auf das Objekt wird auf den ererbten Teil reduziert, im Beispiel also die Sicht auf einen Tierteil. Man kann nach dieser Cast-Operationen nur noch auf die Attribute und Methoden zugreifen, die die Tierklasse definiert hat. Die weitergehende Funktionalität der Patentierklasse ist nicht mehr sichtbar, aber im Gesamtobjekt nach wie vor vorhanden.

Bei einer Down-Cast Operation wird die Sicht erweitert von der reduzierten Sicht einer Basisklasse auf die Sicht der abgeleiteten Klasse. Dies macht nur Sinn, wenn das Ursprungsobjekt selbst auch ein Objekt dieser abgeleiteten Klasse (oder einer davon wiederum abgeleiteten Klasse) ist, auf die die Sicht erweitert werden soll. Im Beispiel muss das Ursprungsobjekt also ein Patentier sein, damit man von einer reduzierten Tiersicht wieder auf die ursprüngliche Patentiersicht gehen kann. Eine solche Überprüfung, ob ein vorliegendes Objekt tatsächlich ein Objekt der geforderten Klasse ist, kann aber erst zur Laufzeit geschehen. Hat dabei der

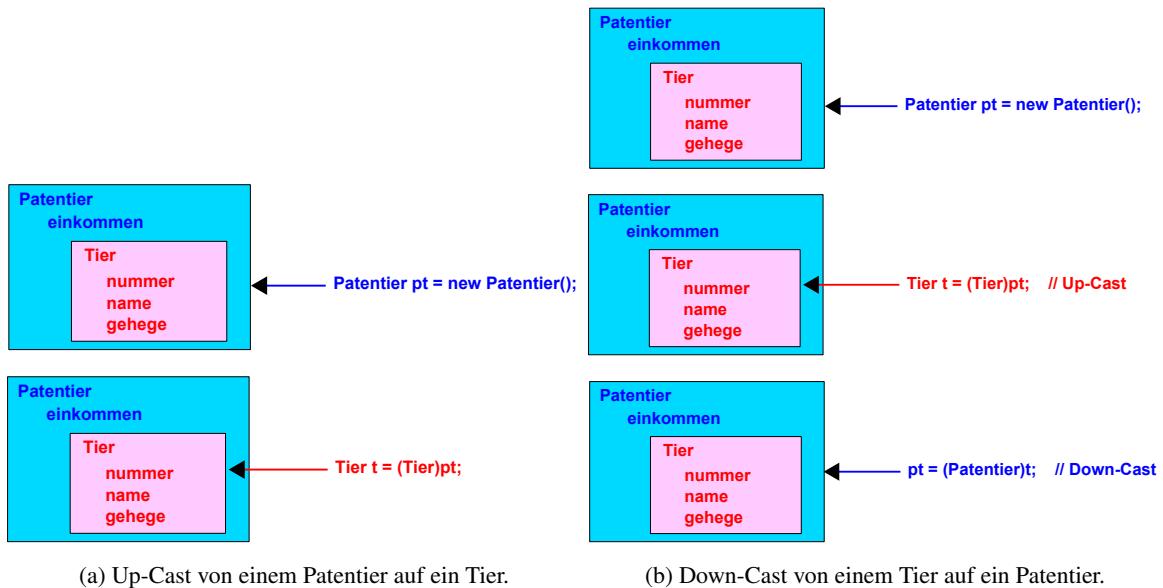


Figure 12.8: Up- und Down-Casts bei (Paten-)Tieren.

Listing 12.5: Beispiele zu Möglichkeiten der Typanpassung.

```

12-1 class Tier { ... }
12-2 class Patentier extends Tier { ... }
12-3 class Spezialtier extends Tier { ... }
12-4
12-5 Patentier pt = new Patentier();
12-6
12-7 Tier t = pt;           // impliziter Up-Cast
12-8     t = (Tier)pt;      // expliziter Up-Cast
12-9     pt = (Patentier)t; // expliziter Down-Cast moeglich
12-10    pt = t;            // Uebersetzungsfehler: impliziter Down-Cast nicht moeglich
12-11
12-12 Spezialtier st = pt; // Uebersetzungsfehler: nicht moeglich
12-13     st = (Spezialtier)pt; // Uebersetzungsfehler: nicht moeglich
12-14     st = (Spezialtier)t; // Laufzeitfehler:
12-15         // java.lang.TypeCastException: Patentier cannot be Cast to Spezialtier

```

Programmierer ein Fehler gemacht (das vorliegende Objekt ist zwar ein Tier, aber kein Patentier), so tritt ein Laufzeitfehler `java.lang.TypeCastException` auf (siehe später Exceptions in Kapitel 14). Das nachfolgende Beispiel verdeutlicht diese grundsätzliche Problematik.

Eine implizite Typumwandlung bei Referenztypen findet immer dann statt, wenn eine Referenz einer Basisklasse **vater** verlangt ist, aber eine Referenz einer abgeleiteten Klasse **sohn** vorliegt. Dies kann zum Beispiel bei einer Zuweisung sein, Parameterübergabe usw.

### Beispiel 12.8:

Listing 12.5 zeigt prinzipielle Möglichkeiten (und Fehler) zu Cast-Operationen bei Referenztypen. Insbesondere bei der letzten Zuweisung ist zu beachten, dass dieser Fehler erst zur Laufzeit erkannt werden kann, da zur Übersetzungszeit nicht (allgemein) bekannt ist, welche Variante eines Tieres in der Variablen **t** vorliegt. Wäre in **t** eine Referenz auf ein Objekt der Klasse **Spezialtier** (oder einer davon abgeleiteten Klasse)

Listing 12.6: Kovarianz bei Feldern.

```

12-1 class Tier { ... }
12-2 class Patentier extends Tier { ... }
12-3 class SpezialTier extends Tier { ... }
12-4
12-5 Tier[] t = new Patentier[10];      // Up-Cast
12-6 t[0] = new Patentier();            // Up-Ccast (kompatibel)
12-7 Patentier pt = (Patentier)t[0];   // Down-Cast
12-8
12-9 t[1] = new SpezialTier();         // Laufzeitfehler:
12-10                                // java.lang.ArrayStoreException: SpezialTier

```

gespeichert, so wäre dieser Cast korrekt und würde nicht zu einem Laufzeitfehler führen. ♦

In der Zeile `t = (Tier)pt` im Listing 12.5 zeigt `t` auf ein Patentierobjekt. Da `t` aber nur vom Typ `Tier` ist, kann man über diese Referenz auch nur auf Attribute und Methoden zugreifen, die in der Tierklasse vorhanden sind, also *nicht* die erweiterte Funktionalität nutzen, die ein Patentier gegenüber einem Tier hat. Es ist zu beachten, dass ein Patentierobjekt auch mit dieser Zuweisung an `t` ein Patentierobjekt bleibt, nur die Sicht wird durch den impliziten Cast eingeschränkt.

In Kapitel 12.8 wurde die Kovarianz im Zusammenhang mit Methoden besprochen. Die Kovarianz besteht auch im Zusammenhang mit der Typkompatibilität von Feldern. Wenn `T1` eine Oberklasse von `T2` ist, so ist auch `T1[]` ein Obertyp von `T2[]`.

### Beispiel 12.9:

Listing 12.6 zeigt Beispiele zur Nutzung der Kovarianz bei Feldern und Möglichkeiten (und Fehler) bei Cast-Operationen. ♦

## 12.3 Pakete und Module\*

Bis jetzt wurden verschiedene Ebenen der Strukturierung von Software vorgestellt:

- Blöcke
- Methoden
- Klassen (und Schnittstellen später)

Die nächste Stufe sind Pakete und Module.

### Pakete

In einem **Paket** werden Klassen, Schnittstellen und weitere Unterpakete zu einer Einheit zusammengefasst. Eine Paketbildung hat u.a. Auswirkungen auf die Sichtbarkeit / Zugriffsrechte von Klassen sowie deren Attribute und Methoden. Weiterhin bildet ein Paket einen eigenen **Namensraum**. Dies bedeutet u.a., dass der gleiche Name wie zum Beispiel für eine Klasse in verschiedenen Namensräumen (also Paketen) mit unterschiedlicher Bedeutung genutzt werden kann.

Listing 12.7: Deklaration von Paketen.

```

12-1 // Datei A.java
12-2 package paket1;      // alle Klassen in dieser Datei gehoeren Paket packet1 an
12-3 public class A {...}
12-4
12-5 // Datei B.java
12-6 package paket2;      // alle Klassen in dieser Datei gehoeren Paket packet2 an
12-7 public class B {
12-8     // Hier ist der qualifizierte Name fuer A notwendig.
12-9     // Der einfache Name A waere falsch, da anderer Namensraum.
12-10    paket1.A instanzvariable;
12-11 }

```

Jedes Paket hat einen freiählbaren Namen. Dazu muss in allen Dateien eines Pakets die Deklaration des Paketnamens vorgenommen werden, wodurch die Paketzugehörigkeit aller Klassen / Schnittstellen in dieser Datei ausgedrückt wird. Eine solche Deklaration muss bis auf mögliche vorausgehende Kommentarzeilen immer die erste Zeile in einer Datei sein. Pro Datei darf es auch nur eine Paketdeklaration geben. Vereinbarungsgemäß wird ein Paketname kleingeschrieben. Ordnet man eine Klasse keinem Paket zu (lässt also die Paketdeklaration zu Beginn einer Datei weg), so gehören die Klassen / Schnittstellen dieser Datei implizit dem **Default-Paket** ohne Namen an, das immer existiert.

### Beispiel 12.10:

Listing 12.7 zeigt ein Beispiel zur Deklaration von zwei Paketen. ♦

Zu einem Paket können **Unterpakete** gebildet werden, was zum Beispiel bei sehr umfangreichen Software-Projekten nützlich ist. Das Java SDK selbst ist ein Beispiel dazu. Namen von Unterpaketen werden gebildet, indem den Namen des Basispakets nimmt und mit einem Punkt trennt den Namen des Unterpakets anhängt. Zu einem Paket `paketName1` kann ein Unterpaket `paketName2` gebildet werden, so dass der gesamte Paketname des Unterpakets `paketName1.paketName2` ist. Es lassen sich somit auch beliebig tiefe Pakethierarchien angeben.

### Beispiel 12.11:

Das Java SDK umfasst nahezu 10.000 Klassen. Diese sind in zahlreichen Paketen und Unterpaketen organisiert. Die meisten der bis jetzt genutzten Klassen sind im Paket `java.lang` enthalten, das immer genutzt werden kann. Dieses Paket ist ein Unterpaket des Pakets `java`.

Ein weiteres Beispiel ist ein Scanner. Um ihn zu nutzen, musste man bisher immer zu Beginneines Pgogramms angeben:

```

12-1 import java.util.*; // bzw. java.util.Scanner

```

Ein weiteres Unterpaket von `java` ist demzufolge `util`, in dem einige nützliche Hilfsklassen (*utilities*) organisiert sind. ♦

Da Pakete eigene Namensräume bilden, muss von außerhalb eines Pakets zur Nutzung dieser Namen der Paketname vorangestellt werden. Man spricht in diesem Zusammenhang von einem **qualifizierten Namen**, im Gegensatz zu einem **einfachen Namen**.

Listing 12.8: Deklaration von Unterpaketen.

```

12-1 // Datei A.java
12-2 package paket1;      // alle Klassen in dieser Datei gehoeren Paket packet1 an
12-3 public class A { ... }
12-4
12-5 // Datei B.java
12-6 package paket2;      // alle Klassen in dieser Datei gehoeren Paket packet1 an
12-7 import paket1.A;      // Zugriff dadurch ueber einfachen Namen A moeglich
12-8 public class B {
12-9     A instanzvariable; // A alleine ist jetzt moeglich
12-10 }
```

Listing 12.9: Nutzung von Paketen.

```

12-1 // Datei A.java
12-2 package paket1;
12-3 public class A { ... }
12-4
12-5 // Datei C.java
12-6 package paket1;
12-7 public class C { ... }
12-8
12-9 // Datei B.java
12-10 package paket2;
12-11 import paket1.*;      // Zugriff dadurch ueber einfachen Namen A bzw. C moeglich
12-12 public class B {
12-13     A instanzvariable1; // A alleine ist jetzt moeglich
12-14     C instanzvariable2; // C alleine ist jetzt moeglich
12-15 }
```

Benutzt man intensiv Klassen oder Schnittstellen eines anderen Pakets in einer Java-Datei, so müsste man an allen Stellen des Gebrauchs den Paketpräfix angeben. Dies wäre sehr aufwändig. Aus diesem Grund lässt sich mit der **import**-Deklaration alle als **public** deklarierten Namen des anderen Pakets direkt ohne Paketpräfix zugreifen. Die **import**-Vereinbarung muss hinter einer **package**-Deklaration stehen (soweit vorhanden) und vor allen weiteren Deklarationen.

### Beispiel 12.12:

Listing 12.8 zeigt die Deklaration eigener Unterpakete und die Nutzung der import-Deklaration. ♦

Ein Paket wird bereits immer *implizit* importiert: **java.lang**. Dadurch ist zum Beispiel die Klasse **String**, die zu diesem Paket gehört, überall über ihren einfachen Namen erreichbar.

Möchte man *alle* Klassen / Schnittstellen eines (Unter-)Pakets importieren, so ist das mit einem **\*** als Angabe der Klasse möglich. Diese Erweiterung gilt nicht für Unterpakethierarchien. Es wäre also falsch anzugeben:  
**import meinPaket.\*.\*;**

### Beispiel 12.13:

Listing 12.9 zeigt ein Beispiel zur Deklaration von Paketen und deren Import. ♦

Java-Klassen, die mit `public` markiert sind, müssen wie bekannt in einer Datei entsprechenden Namens abgelegt sein. Nutzt man Java-Pakete, so werden diese Pakete so abgebildet, dass zu jedem Paket ein eigenes Verzeichnis im Projektordner angelegt wird, in dem dann die Klassen / Schnittstellen dieses Pakets abgelegt sein *müssen*. Unterpakete bilden dann folgerichtig Unterverzeichnisse zu einem Paket. Dies ist zwingend so vorgeschrieben und dient dem Laufzeitsystem u.a. zum Auffinden von Klassendateien während der Laufzeit eines Programms.

## Module

Eine weitere Form der Modularisierung noch oberhalb der Pakete sind Module. Über Moduldeklarationen lassen sich Abhängigkeiten zwischen Paketen spezifizieren. Auf Module wird an dieser Stelle nicht weiter eingegangen.

## 12.4 Zugriffsrechte

In diesem Kapitel wird der Begriffe Schnittstelle vorab verwandt, der erst in einem späteren Kapitel beschrieben wird (Kapitel 13.3). eingeführt; Bis dahin können Sie sich eine Schnittstelle als eine Art Klasse vorstellen.

Java erlaubt durch die Verwendung von bestimmten Schlüsselwörtern, den sogenannten **Modifikatoren** in Klassen, Attribut- und Methodendeklarationen Einfluss zu nehmen darauf, wer Zugriff hat auf eine Klasse oder ein Attribut und wer eine Methode aufrufen darf. Zugriff heißt, dass man die Sichtbarkeit einer Klasse, eines Attributs oder einer Methode gezielt beschränken kann. Ein wichtiges Ziel der Einschränkung von Sichtbarkeit ist die Entwicklung sicherer Software. Nicht jeder Software-Entwickler in einem großen Team soll etwa jede Klasse instanzieren oder ein beliebiges Attribut manipulieren sollen. In sehr großen Software-Projekten kann man über Zugriffsregeln sehr feingranular definieren, welche Klassen von welchen anderen Klassen gesehen werden können oder auch nicht. Generell versucht man (in großen Software-Projekten), die Zugriffsrechte möglich restriktiv zu setzen, um so ungewollte Zugriffe / unbeabsichtigte Änderungen zu verhindern.

Diese Zugriffskontrolle geschieht über die Einschränkung der Sichtbarkeit des qualifizierten und einfachen Namens. Durch Verwendung geeigneter Modifikatoren kann man also die Sichtbarkeit von Namen einschränken und damit bei geeigneter Verwendung eine Software sicherer machen, so dass nur ein Teil aller Namen in bestimmten Programmteilen sichtbar ist.

Als Modifikatoren bei Attributen und Methoden sind möglich: `public`, `protected`, `private` und das Weglassen eines Modifikators, was eine eigene Zugriffsspezifikation bedeutet. Bei Klassen und Schnittstellen ist nur `public` oder das Weglassen möglich.

### 12.4.1 Zugriffsrechte bei Klassen und Schnittstellen

Mit der Diskussion zu Klassen soll begonnen werden. Ist eine Klasse (oder Schnittstelle; siehe 13.3) als `public` markiert, so ist diese Klasse/Schnittstelle überall verwendbar / sichtbar. Fehlt dieser Modifikator, so ist diese Klasse / Schnittstelle nur innerhalb des eigenen Pakets verwendbar. Selbst in Unterpaketen des eigenen Pakets hat man keinen Zugriff auf Klassen, die ohne Modifikator angegeben sind.

#### Beispiel 12.14:

Listing 12.10 zeigt ein Beispiel zur Nutzung von Klassen anderer Pakete. Die Klasse `b` ist außerhalb des eigenen Pakets nicht sichtbar und damit nicht nutzbar. ♦

Als weitere Besonderheit ist zu erwähnen, dass eine Klasse `abc`, die mit `public` markiert ist, in einer Datei

Listing 12.10: Zugriff auf Klassen anderer Pakete.

```

12-1 // Datei A.java
12-2 package paket1;
12-3 public class A { ... }
12-4 class B { ... }
12-5
12-6 // Datei C.java
12-7 package paket1.paket2; // Unterpaket zu paket1
12-8 public class C {
12-9     paket1.A instanzVariable1; // ist erlaubt
12-10    paket1.B instanzVariable1; // Uebersetzungsfehler wegen Zugriffsrechten
12-11 }
```

Listing 12.11: Beispiel zu Zugriffsrechten bei Klassen.

```

12-1 // Datei Test1.java
12-2 package p1;
12-3 public class Test1 {
12-4     private int x;      // nicht sichtbar in anderen Klassen
12-5     public int y;       // sichtbar in anderen Klassen
12-6 }
12-7
12-8 // Datei Test2.java
12-9 package p2;
12-10 class Test2 {        // ohne public
12-11     private int x;    // nicht sichtbar in anderen Klassen
12-12     public int y;     // nicht sichtbar in anderen Klassen anderer Pakete
12-13                         // weil die Klasse dort schon nicht sichtbar ist
12-14 }
```

abgelegt sein muss, die den Namen `Abc.java` haben muss (vergleiche erste Anmerkungen dazu in Kapitel 11.2). Als Konsequenz ist umgekehrt damit auch klar, dass pro Datei nur eine `public` Klasse existieren darf. Es dürfen aber auch beliebig viele weitere Klassen *ohne public* in einer Datei enthalten sein. Ein Sonderfall ist, dass in einer Java-Datei nur Klassen ohne `public` stehen; dann kann der Dateiname beliebig sein.

### 12.4.2 Zugriffsrechte bei Attributen und Methoden

Als nächstes werden Zugriffsrechte im Zusammenhang mit Attribute und Methoden diskutiert. Dazu muss vorweg gesagt werden, dass der Modifikator der Klasse schon einen Schutzrahmen vorgibt (hat höhere Priorität). Hat eine Klasse etwa keinen Modifikator (und ist somit nicht `public`), so sind alle Attribute und Methoden per se für Klassen anderer Pakete gesperrt, also nicht ansprechbar. Innerhalb der Klasse angegebene Modifikatoren zu Attributen / Methoden können zwar beliebig angegeben werden, sind aber niemals gegenüber der Klasse weiter öffnend, allenfalls restriktiver.

#### Beispiel 12.15:

Listing 12.11 zeigt ein Beispiel, in dem in der Klassedefinition der Klasse `Test2` kein `public` angegeben wurde. Dadurch ist das Attribut `y` außerhalb des Pakets nicht sichtbar, obwohl das Attribut selber als `public` markiert ist. ♦

Schlüsselwort	<b>public</b>	<b>protected</b>	keine Angabe	<b>private</b>
eigene Klasse	ja	ja	ja	ja
Klassen in gleichem Paket	ja	ja	ja	nein
zusätzlich Unterklasse in anderem Paket	ja	ja	nein	nein
überall	ja	nein	nein	nein

Table 12.1: Übersicht über Modifikatoren bei Attributen und Methoden.

Allgemein gilt weiterhin auch, dass Methoden und Datenfelder einer Schnittstelle implizit **public** sind. Würde man dort explizit einen anderen Modifikator als **public** angeben, so würde daraus ein Übersetzungsfehler resultieren.

Bezüglich der Attribute und Methoden in Klassen gilt nun folgende Regelung, wobei unter der Sichtbarkeit wie oben schon angemerkt der qualifizierte Namen gemeint ist. In der eigenen Klasse und abgeleiteten Klassen könnte ggfs. darüberhinaus auch der einfache Name möglich sein, was an der nachfolgenden Diskussion aber nichts ändert.

1. Ist ein Attribut / eine Methode **private** deklariert, so ist dieses Attribut / diese Methode nur innerhalb der eigenen Klasse sichtbar.
2. Fehlt ein Modifikator in der Attribut- / Methodendefinition, so kann aus allen Klassen *des gleichen Pakets* auf die Attribute / Methoden zugegriffen werden.
3. Ist als Modifikator **protected** angegeben, so sind diese Attribute/Methoden einerseits sichtbar in allen Klassen des eigenen Pakets (siehe letzter Punkt). Darüberhinaus sind diese Attribute / Methoden aber auch in abgeleiteten Klassen in anderen Paketen sichtbar.
4. Wird als Modifikator **public** angegeben, so ist das Attribut / die Methode überall sichtbar (in allen Klassen in allen Paketen).

Tabelle 12.1 gibt eine Übersicht dazu. Ein Eintrag *ja* in der Tabelle in der Spalte **public** und Zeile *Klassen im gleichen Paket* bedeutet, dass in allen Klassen im gleichen Paket prinzipiell auf eine so markierte Methode/Attribut zugegriffen werden kann, also über den qualifizierten Namen sichtbar ist.

### Beispiel 12.16:

Die Listings 12.12 und 12.13 zeigen ein zusammenhängendes Beispiel, in dem alle Fälle zu Zugriffsrechten auftreten. ♦

Mit gezielter Verwendung dieser Modifikatoren lassen sich auch Getter/Setter-Methoden sehr einfach lösen, die ja den direkten Zugriff auf Attribute verhindern sollen, ihre Nutzung aber über die Getter-/Setter-Methoden erlauben sollen. Dazu definiert man jeweils ein Attribut sehr einschränkend (je nach Nutzung zum Beispiel in abgeleiteten Klassen), also zum Beispiel als **private**. Dadurch ist der direkte Zugriff auf dieses Attribut von außerhalb der Klasse nicht möglich. Also bei einer Instanzvariablen wäre **objektReferenz.attributName** nicht möglich und bei einem Klassenattribut auch nicht **klassename.attributName**.

Weiterhin definiert man eine Getter- und/oder Setter-Methode, die dann aber **public** sein muss (beziehungsweise etwas weiter eingeschränkt, falls notwendig). Dadurch ist zum Beispiel das Abfragen des Wertes des Attributes generell möglich.

Listing 12.12: Auswirkung von Zugriffsrechten (Teil 1).

```

12-1 /**
12-2 * Basisklasse fuer Zugriffsrechte und weitere Klassen im gleichen Paket
12-3 */
12-4 package paket1;
12-5
12-6 public class Zugriffsrechte1 {
12-7
12-8     private int a;      // nur innerhalb der eigenen Klasse gueltig
12-9     int b;            // in allen Klassen des gleichen Pakets
12-10    protected int c; // auch in abgeleiteten Klassen anderer Pakete
12-11    public int d;    // ueberall gueltig
12-12
12-13    void test() {
12-14        a = 4711;   // moeglich nur in eigener Klasse
12-15        b = 4712;   // moeglich
12-16        c = 4713;   // moeglich
12-17        d = 4714;   // moeglich
12-18    }
12-19 }
12-20
12-21 // andere Klasse gleiches Paket
12-22 class AndereKlasseGleichesPaket1 {
12-23     void test(Zugriffsrechte1 obj) {
12-24         //obj.a = 4711; // nicht moeglich
12-25         obj.b = 4712; // moeglich, weil gleiches Paket
12-26         obj.c = 4713; // moeglich, weil gleiches Paket
12-27         obj.d = 4714; // moeglich, weil public
12-28     }
12-29 }
12-30
12-31 // andere Klasse gleiches Paket, abgeleitet
12-32 class AndereKlasseGleichesPaket2 extends Zugriffsrechte1 {
12-33     void test() {
12-34         // a = 4711; // nicht moeglich wegen private
12-35         b = 4712; // moeglich, weil gleiches Paket
12-36         c = 4713; // moeglich, weil gleiches Paket
12-37         d = 4714; // moeglich, weil public
12-38     }
12-39 }
```

Listing 12.13: Auswirkung von Zugriffsrechten (Teil 2).

```

12-1 /**
12-2 * andere Klassen in anderem Paket
12-3 */
12-4 package paket2;
12-5
12-6 // andere Klasse gleiches Paket
12-7 class AndereKlasseAnderesPaket1{
12-8     void test(paket1.Zugriffsrechte1 obj) {
12-9         //obj.a = 4711;    // nicht moeglich
12-10        //obj.b = 4712;    // nicht moeglich, weil anderes Paket
12-11        //obj.c = 4713;    // nicht moeglich, weil anderes Paket und nicht abgeleitet
12-12        obj.d = 4714;    // moeglich, weil public
12-13    }
12-14 }
12-15
12-16 // andere Klasse gleiches Paket, abgeleitet
12-17 class AndereKlasseAnderesPaket2 extends paket1.Zugriffsrechte1 {
12-18     void test() {
12-19         //a = 4711;    // nicht moeglich wegen private
12-20         //b = 4712;    // nicht moeglich, weil anderes Paket
12-21         c = 4713;    // moeglich, weil abgeleitete Klasse anderes Paket
12-22         d = 4714;    // moeglich, weil public
12-23     }
12-24 }
```

Listing 12.14: Getter-/Setter Methoden unter Nutzung von Modifikatoren.

```

12-1 public class Test {
12-2     private static int var1;    // Klassenvariable privat
12-3     private int var2;          // Instanzvariable privat
12-4
12-5     // hier in komprimierter Form notiert
12-6     public static int getVar1() { return var1; }
12-7     public static void setVar1(int wert) { var1 = wert; }
12-8     public int getVar2() { return var2; }
12-9     public void setVar2(int wert) { var2 = wert; }
12-10 }
```

**Beispiel 12.17:**

Listing 12.14 zeigt ein Beispiel zur gezielten Nutzung der Zugriffsrechte zur Realisierung von Getter-/Setter-Methoden. ♦

## 12.5 Das Prinzip des Polymorphismus

In der Beschreibung der primitiven Datentypen in Java (`boolean`, `int`, `float` usw.) gab es zu jedem dieser Typen eine binäre Operation  $==: T \times T \rightarrow \text{boolean}$ , wobei  $T$  für den jeweiligen Typen (zum Beispiel `int`) steht. Diese Operation  $==$  macht für alle Grundtypen sinngemäß das gleiche, der Test zweier Elemente der Grundmenge auf Gleichheit. Es ist zwar jeweils eine andere Operation – der Vergleich zweier char-Werte ist anders durchzuführen als der Vergleich zweier double-Werte –, die Bedeutung der Operation ist aber doch immer ähnlich: der Vergleich zweier Werte des zugrundeliegenden Typs. Man spricht in solch einem Fall von **Polymorphismus** (Vielgestaltigkeit). Welche Vergleichsmethode bei Verwendung des Symbols  $==$  genau

gemeint ist, ist durch die Typen der Operanden bestimmt. Jedes Objekt kennt seine konkrete Ausrichtung der Implementierung des Vergleichstests.

### Beispiel 12.18:

In `3 == 4` wird der Infixoperator `==: int × int → boolean` verwendet, in `3.0 == 4.0` der Operator `==: double × double → boolean`. ♦

Im objektorientierten Programmieren bedeutet Polymorphismus, dass die gleiche Nachricht (Methodenaufruf) an Objekte verschiedener Klassen gesendet werden kann. Objekte all dieser Klassen reagieren auf diese Nachricht, jedoch kann die Reaktion im Detail / in der Realisierung durchaus unterschiedlich sein (char-Werte vergleichen bzw. double-Werte vergleichen). Im Zoobeispiel ist dies etwa die Methode `toString()`, die in allen Klassen (`Tier`, `Patentier`, `Gehege`) vorhanden ist, aber alle Klassen reagieren auf ihre Weise auf die Anfrage, indem sie den Inhalt der Attributwerte des Objekts (Zustand des Objekts) in lesbarer Form ausgeben. Der Vorteil des Polymorphismus ist der, dass der Sender der Nachricht (Aufrufer der Methode) die gleiche Nachricht an *unterschiedliche* Objekte schicken kann, ohne sich darum kümmern zu müssen, wie die entsprechende Methode bei diesem Objekt genau heißen mag (etwa `printGehege()`, `ausgabeTier()`, `druckePatentier()`). Der Sender der Nachricht muss also nur wissen, dass das Objekt eine solche Nachricht versteht, zu welcher exakten Klasse dieses Objekt gehört, interessiert an dieser Stelle nicht.

Der Grundgedanke beim Einsatz des Polymorphismus in der objektorientierten Programmierung ist natürlich der, dass durch Polymorphismus hinter Nachrichten mit dem gleichen Namen auch sinnverwandte Aktionen stattfinden (zum Beispiel der Ausdruck des internen Zustands des Objektes, der Vergleich zweier Werte usw.)

### Beispiel 12.19:

Schaut man sich das Beispiel der `toString()`-Methode in den bekannten Klassen an, so würde ohne Polymorphismus (also ohne die Möglichkeit `toString()` zu überschreiben und so einheitlich wiederzuverwenden) ein Programmentwickler direkt oder indirekt ein Konstrukt der folgenden Form einsetzen müssen (in Pseudocode, keine korrekte Java-Syntax):

```

12-1  public void meinPolyPrint(Object o) {
12-2      switch(welcheKlasseIstObjekt(o)) {
12-3          case Gehege:   o.printGehege();      break;
12-4          case Tier:    o.ausgabeTier();     break;
12-5          case Patentier: o.druckePatentier(); break;
12-6      }
12-7  }
```

Würde der Entwickler nun zum Beispiel eine Klasse `Luxusgehege` von der Klasse `Gehege` ableiten, so müsste er an *allen* Stellen, an denen solch eine eigene Polymorphismusemulation eingesetzt wird, zusätzlichen Code einfügen, der diese neue Klasse behandeln würde. Solch ein Vorgehen würden den Konzepten moderner Software-Entwicklung widersprechen. ♦

## 12.6 Überladen und Überschreiben von Methoden

Man kann in einer abgeleiteten Klasse Methoden der Basisklasse *überladen*, wie dies im Kapitel 8.3 beschrieben wurde (dort in Bezug auf eine Klasse, also ohne Vererbung).

Listing 12.15: Beispiel zum Überladen einer Methode über Klassengrenzen hinweg.

```

12-1 class Test1 {
12-2     int x = 1;
12-3     int addiere(int y) {
12-4         return x+y;
12-5     }
12-6 }
12-7
12-8 class Test2 {
12-9     int addiere(int y, int z) {
12-10         return x+y+z;
12-11     }
12-12 }
```

Listing 12.16: Beispiel zum Überschreiben einer Methode.

```

12-1 class Tier {
12-2     ...
12-3     public String toString() {
12-4         return "Tier " + getName()
12-5             + ", Gehege=" + ((gehege == null) ? "ohne Gehege" : nummer);
12-6     }
12-7 }
12-8
12-9 class Patentier {
12-10     ...
12-11     public String toString() {
12-12         return super.toString() + ", Einkommen=" + einkommen;
12-13     }
12-14 }
```

**Beispiel 12.20:**

Die Klassen in Listing 12.15 zeigen die Möglichkeit des Überladens über Klassengrenzen hinweg. In der Klasse `Test2` sind zwei "Versionen" von `addiere` verfügbar: mit einem Parameter und mit zwei Parametern.

Neu ist bei der Vererbung nun das Prinzip des *Überschreibens* einer Methode, indem man in der abgeleiteten Klasse eine Methode *gleicher Signatur* (gleicher Name, gleiche Anzahl, Reihenfolge und Typen der Parameter) und Ergebnistyp definiert (und die Methode der Basisklasse darf nicht als `private` oder `final` markiert sein). Dies bewirkt, dass bei Aufruf der Methode (mit einfachem Namen) innerhalb der abgeleiteten Klasse auch die Methode der abgeleiteten Klasse aufgerufen wird (vergleiche das Konzept der Gültigkeit und Sichtbarkeit bei Variablen, Kapitel 6.2). Die Signatur-gleiche Methode der Basisklasse könnte nach wie vor explizit über `super` aufgerufen werden.

Im Zusammenhang mit dem Überschreiben von Methoden ergeben sich eine Reihe von interessanten (und komplexen) Fragestellungen, die in den nachfolgenden Abschnitten diskutiert werden.

**Beispiel 12.21:**

Ein Beispiel zur sinnvollen Nutzung ist das Überschreiben der Methode `toString` im Tier-/Patentierbeispiel (siehe Listing 12.16).

Hinweis: Die Methode `toString` der Tierklasse überschreibt dabei bereits selber eine Methode gleicher

Listing 12.17: Nutzung des Polymorphismus.

```

12-1  /*
12-2   * Polymorphismus
12-3   */
12-4  public class Polymorphismus {
12-5
12-6      // zeige Zustand eines Tieres auf dem Bildschirm
12-7      public static void zeigeTier(String ueberschrift, Tier t) {
12-8          System.out.println(ueberschrift + " : " + t.toString());
12-9      }
12-10
12-11     public static void main(String[] args) {
12-12         // normales Tier erzeugen
12-13         Tier t = new Tier();
12-14         // Patentier erzeugen
12-15         Patentier pt = new Patentier(100);
12-16
12-17         // Methode kann mit beiden Tieren aufgerufen werden
12-18         zeigeTier("normales Tier", t);
12-19         zeigeTier("Patentier", pt);
12-20     }
12-21 }
```

Signatur, die in der Klasse `Object` definiert ist. ♦

Mit dem Überladen und Überschreiben hat man damit zwei Techniken zur Realisierung von Polymorphismus.

## 12.7 Substitutionsprinzip

Auch auf Klassen-/Objektebene lässt sich Polymorphismus einsetzen. Überall dort, wo ein Objekt einer Basisklasse  $K_1$  genutzt werden kann, lässt sich auch ein Objekt einer Klasse  $K_2$  angeben, wobei  $K_2$  direkt oder indirekt von  $K_1$  abgeleitet sein muss. Das nennt man das **Substitutionsprinzip**. An dieser Stelle sei nochmals darauf hingewiesen, dass man mit `super` auf den ererbten Teil eines Objekts Bezug nehmen kann. Und wie vorher bereits darauf hingewiesen wurde, ist dieser Teil, also `super`, ein Objekt.

Das Substitutionsprinzip wird umfassend in Klassenbibliotheken und Frameworks genutzt. Man benötigt beispielsweise durch dieses Prinzip nur *eine* Methode, um eine ganze Hierarchie von Klassen hinsichtlich eines gemeinsamen Aspekts einheitlich behandeln zu können. Diese Bibliotheken / Frameworks lassen sich damit aber auch nur von Programmierern nutzen, die dieses Prinzip und dessen sinnvollen Einsatz verstanden haben!

### Beispiel 12.22:

In Listing 12.17 ist ein Beispiel zur Nutzung des Substitutionsprinzips (und Polymorphismus) angeben. Man beachte, dass in der Typspezifikation des Parameters der Methode `zeigeTier` der Typ `Tier` angegeben ist. Ruft man die Methode `zeigeTier` mit einem normalen Tier auf, so passt der Typ des Arguments mit dem Typ des Parameters überein, was nicht weiter erkläруngsbedürftig ist.

Im zweiten Aufruf dieser Methode wird aber ein `Patentier` übergeben, das von `Tier` abgeleitet ist. An dieser Stelle tritt das Substitutionsprinzip ein, da ein Patentier auch einen Tieranteil hat. Auf den Aspekt des Aufrufs der Methode `toString` in `zeigeTier` wird im folgenden Abschnitt eingegangen. ♦

Listing 12.18: Beispiel zur Kovarianz im Ergebnistyp.

```

12-1 class Tier {
12-2     Tier gebeTier() {
12-3         return this;
12-4     }
12-5 }
12-6
12-7 class Patentier extends Tier {
12-8     // hier wird die Methode ueberschrieben, obwohl der Rueckgabetyp sich unterscheidet
12-9     Patentier gebeTier() {
12-10         return this;
12-11     }
12-12 }
```

## 12.8 Kovarianz, Kontravarianz und Invarianz \*

In diesem Kapitel geht es im Wesentlichen um die Kompatibilität von Typen bei Vererbung und damit auch um die Frage, wann eine Methode überschrieben wird oder nicht. Wird in einer abgeleiteten Klasse eine Methode der Basisklasse überschrieben, so müssen in der überschreibenden Methode der abgeleiteten Klasse neben dem Methodennamen auch die Parametertypen und der Rückgabetyp jeweils den Typen der überschriebenen Methode der Basisklasse entsprechen (siehe Kapitel 12.6).

Beim Überschreiben von Methoden, die einen Referenztyp als Rückgabetyp haben, gibt es eine Besonderheit, die man als **kovarianten Rückgabetyp** bezeichnet. Ist als Rückgabetyp einer Methode der Basisklasse `t1` spezifiziert, so kann in der überschreibenden Methode der abgeleiteten Klasse auch ein Typ `t2` angegeben werden, wenn `t2` von `t1` abgeleitet ist. Trotz des unterschiedlichen Ergebnistyps wird die Methode der Basisklasse damit dennoch überschrieben. Man spricht von Kovarianz, weil *mit* der Vererbungsrichtung sich auch der Typ ändert, vom Basistyp hin zum abgeleiteten Typ.

### Beispiel 12.23:

Gegeben ist das Programm in Listing 12.18. Die Methode `gebeTier` existiert in beiden Klassen, allerdings mit *unterschiedlichem* Ergebnistyp (ein Referenztyp), die Typen hängen bei Kovarianz in der angegeben Richtung der Vererbung voneinander ab. Ein Aufruf von `new Tier().gebeTier()` (also das Erzeugen eines neuen Tieres und mit diesem Tier als Bezugsobjekt die Instanzmethode `gebeTier` aufrufen) liefert als Ergebnis ein Tierobjekt, ein Aufruf von `new Patentier().gebeTier()` liefert als Ergebnis ein Patentierobjekt. ♦

Im Zusammenhang mit Feldern gibt es eine weitere Art der Kovarianz. Gegeben sei eine Basisklasse `t1` und eine davon abgeleitete Klasse `t2`. Man kann nun einen Feldtyp zu `t1` und `t2` angeben: `t1[]` beziehungsweise `t2[]`. In `t1[]` (aufbauend auf der Basisklasse) lassen sich nun auch Objekte der abgeleiteten Klasse ablegen.

### Beispiel 12.24:

Die Klasse `Object` ist die "Mutter aller Klassen" in Java. Demzufolge kann man ein Feld zu diesem Typ anlegen und *alle* Objekte irgendeines Referenztyps dort speichern (Is-A-Beziehung). Listing 12.19 zeigt ein Beispiel zur Nutzung. ♦

Listing 12.19: Beispiel zur Nutzung von `Object` als Basisklasse.

```

12-1 // lege Feld zur Basisklasse Object an
12-2 Object[] container = new Object[100];
12-3
12-4 // lege ein Element des Basistyps dort ab
12-5 container[0] = new Object();
12-6 // lege Objekte ab, deren Klasse von Object abgeleitet ist
12-7 container[1] = new Tier();
12-8 container[2] = new Patentier();
12-9 ...

```

Listing 12.20: Kovarianz bei Feldern.

```

12-1 /**
12-2 * Polymorphismus (Version 2)
12-3 */
12-4 public class Polymorphismus2 {
12-5
12-6     // zeige Zustand eines Tieres auf dem Bildschirm
12-7     public static void zeigeTier(String ueberschrift, Tier t) {
12-8         System.out.println(ueberschrift + " : " + t.toString());
12-9     }
12-10
12-11     public static void main(String[] args) {
12-12         // Feld von Tieren
12-13         Tier[] tiere = new Tier[2];
12-14
12-15         // normales Tier erzeugen
12-16         tiere[0] = new Tier();
12-17         // Patentier erzeugen. Auch durch die Zuweisung an eine Variable vom Typ Tier
12-18         // bleibt das Objekt ein Patentier
12-19         tiere[1] = new Patentier(100);
12-20
12-21         // Methode kann mit einem beliebigen Tier aufgerufen werden
12-22         for(int i=0; i<tiere.length; i++) {
12-23             zeigeTier("was ist es?", tiere[i]);
12-24         }
12-25     }
12-26 }

```

**Beispiel 12.25:**

In Listing 12.20 ist ein Beispiel zur sinnvollen Nutzung des Substitutionsprinzips im Zusammenhang mit Kovarianz angegeben. Hier wird eine Feldvariable `tiere` zur Basisklasse angelegt. In diesem Feld lassen sich dann sowohl Tiere als auch Patentiere speichern (und alle weiteren Objekte von Klassen, die von `Tier` direkt oder indirekt abgeleitet wären). Wichtig zum Verständnis ist dabei, dass durch diese Speicherung ein Patentier nach wie vor ein Patentier *bleibt*. Die Konsequenzen dazu werden ebenfalls im nächsten Abschnitt noch eingehender behandelt.



Unter **Kovarianz** versteht man allgemein in der Objektorientierten Programmierung, dass *mit der Vererbungsrichtung* auch eine gewisse Eigenschaft übertragen wird, zum Beispiel ein Typ. Demgegenüber spricht man von **Kontravarianz**, wenn eine Eigenschaft *entgegen der Vererbungsrichtung* übertragen wird. Und es handelt sich um **Invarianz**, wenn keine Änderung vorliegt.

Die **Kontravarianz** spielt bei Übergabeparametern eine wichtige Rolle. Wird eine Methode einer Basisklasse `T1` in einer abgeleiteten Klasse `T2` überschrieben, so können die Typen der Übergabeparameter in der überschreibenden Methode auch einen Typ haben, der genereller ist (höher in der Vererbungshierarchie in Richtung Basisklasse `T1`).

## 12.9 Statische und dynamische Bindung

Schickt ein Objekt `o1` an ein anderes Objekt `o2` eine Nachricht, d.h. ruft eine Methode des Objektes `o2` auf, wie wird (und wer) ermittelt, welche Methode welcher Klasse genau ausgeführt werden muss? Dies ist im Zusammenhang mit überschriebenen Methoden, Vererbung und dem Substitutionsprinzip eine interessante (und keine einfach zu beantwortende) Frage!

**Beispiel 12.26:**

Im Programm aus Listing 12.21 existieren neben der Klasse `Aufrufstest` vier Klassen, die einen Baum von abgeleiteten Klassen generieren, d.h. `K4` ist von `K2` abgeleitet und `K2` und `K3` von `K1`. `K1`, `K2` und `K3` besitzen eine Methode `toString()`, `K4` nicht. Erzeugt man ein Objekt des Typs `K4` und ruft von diesem Objekt die Methode `toString()` auf, so ist eine Frage, ob und wenn ja welches `toString()` konkret aufgerufen wird (Abbildung 12.9).



Aufgrund der Wirkungsweise der Prinzipien der Vererbung und des Überschreibens kennt jede Klasse in einer Klassenhierarchie ihre eigenen Methoden und die ererbten Methoden aller direkten und indirekten Oberklassen. Bei jedem Methodenaufruf zu einer Instanzmethode wird nun *immer vom Gesamtbezugssubjekt ausgehend* die Vererbungshierarchie nach oben schauend die erste Methode gewählt, die von der Signatur her zum Aufruf passt. Diese wird dann ausgeführt.

Von dieser allgemeinen Regel gibt es Ausnahmen, nämlich dann, wenn ein Überschreiben einer Methode in einer abgeleiteten Klasse garnicht möglich ist. Dies ist der Fall, wenn eine Methode als `private`, `static` oder `final` deklariert ist. Muss ein Methodenaufruf aufgeführt werden und ist an dieser Aufrufstelle im Programm eine Methode mit einer passenden Signatur sichtbar, die mit `private`, `static` oder `final` deklariert ist, so wird diese Methode aufgerufen!

Handelt es sich also bei dem ursprünglichen Aufruf einer Methode um eine Instanzmethode, so muss zu Beginn entschieden werden, was das relevante Objekt ist, also die Instanz zu diesem Aufruf. Aufgrund des Substitutionsprinzips und möglicher Typanpassungen kann zum Beispiel eine Referenz vom Typ einer Basisklasse nur auf ein Teillobject eines umfassenderen Objektes verweisen (zum Beispiel der Tier-Teil in

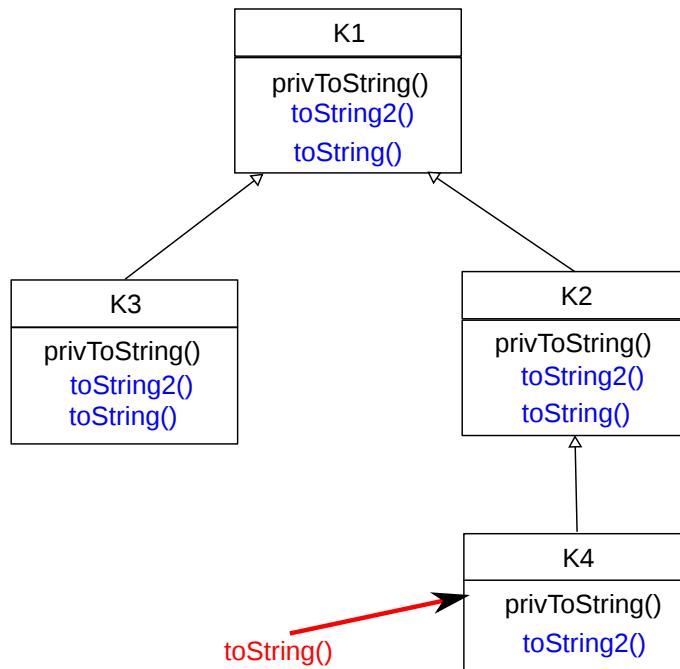


Figure 12.9: Baum der abgeleiteten Klassen

Listing 12.21: Methodenaufrufe mit Objekten.

```

12-1 /**
12-2 * Polymorphie und dynamisches/statisches Binden
12-3 */
12-4 public class AufrufTest {
12-5     public static void main(String [] args) {
12-6         K4 k4 = new K4();
12-7         System.out.println("k4: " + k4.toString());
12-8     }
12-9 }
12-10
12-11 class K1 {
12-12     private String privToString() { return "priv in K1"; }
12-13     public String toString2() { return "toString2 in K1"; }
12-14     public String toString() { return "in K1" + privToString(); }
12-15 }
12-16
12-17 class K2 extends K1 {
12-18     private String privToString() { return "priv in K2"; }
12-19     public String toString2() { return "toString2 in K2"; }
12-20     public String toString() { return "in K2" + privToString() + toString2(); }
12-21 }
12-22
12-23 class K3 extends K1 {
12-24     private String privToString() { return "priv in K3"; }
12-25     public String toString2() { return "toString2 in K3"; }
12-26     public String toString() { return "in K3" + privToString(); }
12-27 }
12-28
12-29 class K4 extends K2 {
12-30     public String toString2() { return "toString2 in K4"; }
12-31     private String privToString() { return "priv in K4"; }
12-32 }
  
```

Listing 12.22: Beispiel zur Polymorphie.

```

12-1 public class Polymorphismus2 {
12-2     public static void zeigeTier(String ueberschrift, Tier t) {
12-3         System.out.println(ueberschrift + " : " + t.toString());
12-4     }
12-5
12-6     public static void main(String[] args) {
12-7         zeigeTier("Tier", new Tier());
12-8         zeigeTier("Patentier", new PatenTier());
12-9     }
12-10 }
```

einem Patentierobjekt). Im Zusammenhang mit einem Methodenaufruf einer Instanzmethode, die nicht mit **private** oder **final** markiert ist, wird aber immer das Objekt genommen, auf das die Referenz tatsächlich verweist (also das umfassendere Ursprungsobjekt, zum Beispiel das Patentier).

### Beispiel 12.27:

Im ersten Aufruf der Methode **zeigeTier** in Listing 12.22 wird ein Tier übergeben, womit der Typ des übergebenen Objekts identisch ist mit dem Typ des Methodenparameters **t**. Im zweiten Aufruf wird ein Patentier übergeben. Innerhalb der Methode **zeigeTier** wird deshalb das Patentierobjekt als Instanz genommen (und nicht nur der Tier-Teil des Patentierobjekts) und die überschriebene Methode **toString** in der Patentierklasse aufgerufen. ♦

In dem besprochenen Fall (Instanzmethode ohne **private** und **final**) wird vom Java-Laufzeitsystem *zur Laufzeit des Programms* immer der Baum der abgeleiteten Klassen von dem Blatt, das den Typ des Objekts darstellt, hin zur Wurzel (Basisklasse) durchlaufen und die erste gefundene Methode mit passender Signatur auf diesem Weg für diesen Aufruf genommen. Im Listing 12.21 wäre dies also die Methode **toString()** der Klasse **K2**. Diese Entscheidung erst zur Laufzeit und immer vom Ursprungsobjekt ausgehend nennt man auch **dynamische Bindung**.

Im Gegensatz dazu gibt es auch die **statische Bindung** bei Methoden, die mit **private**, **final** oder **static** markiert sind, und bei denen *zur Übersetzungszeit des Programms* festgelegt wird, welche Methode konkret damit aufgerufen wird.

Die statische Bindung hat praktisch gesehen und auf den Punkt gebracht höhere Priorität als die dynamische Bindung. Zuerst schaut der Compiler, ob in der Klasse, in der der Methodenaufruf stattfindet, oder in einer Klasse in der Ableitungshierarchie aufwärts eine passende Methode mit statischer Bindung existiert. Diese würde, so existent, aufgerufen. Ansonsten, *und nur dann*, wird mittels dynamischer Bindung von der Klasse des *Ursprungsobjekts* ausgehend in der Klassenhierarchie aufwärts nach dem passenden Methodenaufruf gesucht. Die Ausgabe zu dem Programm in Listing 12.21 ist also:

```
12-1 in K2 priv in K2 toString2 in K4
```

das heißt:

- Die Methode **toString** aus der Klasse **K2** wurde aufgerufen aufgrund der dynamischen Bindung: gehe vom Ursprungsobjekt **K** aus und suche in der Ableitungshierarchie das erste Vorkommen der passenden Methode (in **K2** gefunden).
- Der dort erfolgte Aufruf der Methode **privToString** ist *nicht* vom Ursprungsobjekt ausgegangen

(sonst wäre der Teil der Ausgabe `priv in K4`), sondern aufgrund der existierenden *statischen* Methode `printToString` in `K2` wurde diese ausgeführt (statische Bindung für diesen Methodenaufruf).

- Der weitere Aufruf in `K2` der Methode `toString2` erfolgte aufgrund der dynamischen Bindung (es gab dazu keine Methode mit passender Signatur und statischer Bindung) wiederum vom Ursprungsobjekt aus, weshalb in der Ausgabe steht `toString2 in K4`. Gäbe es eine passende Methode `toString2` mit `static / private / final` in `K2` oder in der Klassenhierarchie aufwärts (also hier konkret nur `K1`), so wäre diese aufgrund der stärker bindenden statischen Bindung aufgerufen worden. Also beispielsweise, wenn die Methode `toString2` in `K1` und `K2` als `private` deklariert worden wäre.

### Beispiel 12.28:

In Listing 12.17 wurde das Substitutionsprinzip angewandt, um eine einheitliche Methode zur Ausgabe von Tieren und Patentieren zu schreiben. Ruft man das Programm auf, so erscheint als Ausgabe:

```
12-1 normales Tier : Tier unbekannt, Gehege=ohne Gehege
12-2 Patentier : Tier unbekannt, Gehege=ohne Gehege, Einkommen=100
```

Die `toString` Methode wurde also *relativ zum Ursprungsobjekt* ausgeführt, und nicht immer die Methode der Klasse `Tier`! Also dynamische Bindung ist hier der Fall.

Analog geschieht dies im zweiten Beispiel in Listing 12.20. Hier ist die Ausgabe des Programms

```
12-1 was ist es? : Tier unbekannt, Gehege=ohne Gehege
12-2 was ist es? : Tier unbekannt, Gehege=ohne Gehege, Einkommen=100
```

also auch hier wurde die Methode `toString` des Ursprungstyps aufgerufen. ♦

## 12.10 Operator instanceof

Hat man eine abgeleitete Klasse, so kann der Fall auftreten, dass man formal eine Referenz eines Basistyps hat, die aber sowohl auf ein Objekt des Basistyps als auch auf ein Objekt eines abgeleiteten Typs verweisen kann. Dies war zum Beispiel der Fall im Beispiel des letztern Abschnitts (Listing 12.20), wo in der Methode `zeigeTier` sowohl ein `Tier`-Objekt als auch ein `Patentier`-Objekt übergeben werden konnte.

Das Beispiel zu 2D- und 3D-Punkten sowie Geraden soll als Basis der nachfolgenden Diskussion genommen werden (siehe Listings 11.16, 12.2 und 11.17). Bis jetzt wurde die Klasse `Gerade` nur mit 2D-Punkten genutzt. Was wäre zu tun, wenn man eine Gerade auch mit 3D-Punkten nutzen wollte, also damit auch eine 3D-Gerade damit bauen könnte?

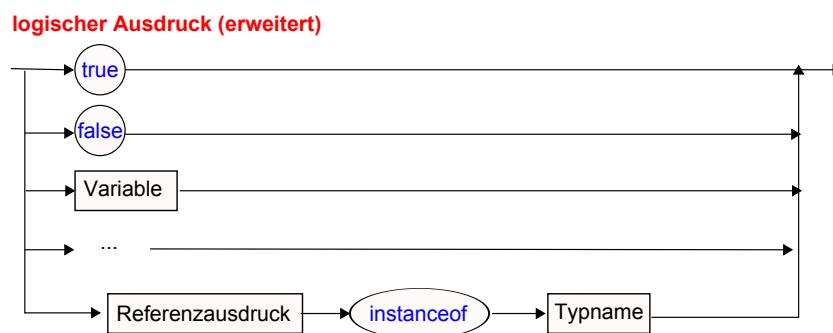
Eine Anpassung der Klasse `Gerade` auf diese Erweiterung wäre keine gut Idee, das man ja eigentlich in Punktklassen die Ausprägungen von Punkten kapseln will (also an einer Stelle), und nicht dort, wo man die Punkte verwendet / nutzt (das können ganz viele Stellen sein). Da die Klasse `Punkt3D` von der Klasse `Punkt` abgeleitet ist, ist die Klasse `Punkt` aufgrund des Substitutionsprinzips der in `Gerade` nach wie vor zu verwendende Referenztyp, die Klasse muss also nicht angepasst werden. Der bessere Ansatz ist es deshalb, dass man in der Klasse `Punkt3D` entsprechende Anpassungen vornimmt, um herauszufinden, ob man aktuell einen 2D-Punkt oder 3D-Punkt vorliegen hat, um so eine Verschiebung in 2D oder 3D durchzuführen. Mit den bisher bekannten Möglichkeiten ist das aber nicht möglich!

Listing 12.23 zeigt die angepasst Klasse `Punkt3D`. In ihr wird die Methode `move` der Klasse `Punkt` überschrieben. Man beachte, dass der Parametertyp der Methode deswegen `Punkt` sein muss und nicht `Punkt3D` sein darf! Nur so ist garantiert, dass der Methodenaufruf `p1.move(p)` und `p2.move(p)` auch wirklich in der Klasse `Punkt3D` behandelt wird. Ansonsten würde die Methode `move` nicht überschrieben und immer die 2D-Methode in der Klasse `Gerade` aufgerufen. Trotzdem kann aber die Methode `move` natürlich aufgrund des

Listing 12.23: Punkt3D unter Verwendung von instanceof.

```

12-1 /**
12-2 * Punkt in 3D
12-3 */
12-4 public class Punkt3D extends Punkt {
12-5     // zusätzliche Instanzvariable: dritte Koordinate
12-6     private double z;
12-7
12-8     // Konstruktor
12-9     public Punkt3D(double x, double y, double z) {
12-10         // (richtigen) Konstruktor der Oberklasse aufrufen
12-11         super(x,y);
12-12         this.z = z;
12-13     }
12-14
12-15     public double getZ() {
12-16         return z;
12-17     }
12-18
12-19     // verschiebe Punkt
12-20     public void move(Punkt p) {
12-21         // verschiebe 2D Anteil
12-22         super.move(p);
12-23
12-24         // teste, ob das Bezugsobjekt ein 2D oder 3D Punkt ist
12-25         if(p instanceof Punkt3D) {
12-26             // verschiebe zusätzlichen 3D Anteil
12-27             z += ((Punkt3D)p).z;
12-28         }
12-29     }
12-30
12-31     // erzeuge String-Darstellung
12-32     public String toString() {
12-33         return "(" + super.getX() + "," + super.getY() + "," + z + ")";
12-34     }
12-35 }
```

Figure 12.10: Erweitertes Syntaxdiagramm für logische Ausdrücke um den `instanceof` Operator.

Listing 12.24: Nutzung von 2D- und 3D-Geraden.

```

12-1  /** Geometrietest (Version 3)
12-2  */
12-3  public class GeometrieTest {
12-4
12-5      public static void main(String [] args) {
12-6          // mit 2D arbeiten
12-7          Punkt p1 = new Punkt(0,0);
12-8          Punkt p2 = new Punkt(1,1);
12-9          Punkt p3 = new Punkt(1,1);
12-10
12-11         Gerade g1 = new Gerade(p1, p2);
12-12         System.out.println("g1 vor verschieben: " + g1);
12-13         g1.move(p3);
12-14         System.out.println("g1 nach verschieben: " + g1);
12-15
12-16         // mit 3D arbeiten
12-17         Punkt3D p31 = new Punkt3D(0,0,0);
12-18         Punkt3D p32 = new Punkt3D(1,1,1);
12-19         Punkt3D p33 = new Punkt3D(1,1,1);
12-20
12-21         Gerade g2 = new Gerade(p31, p32);
12-22         System.out.println("g31 vor verschieben: " + g2);
12-23         g2.move(p33);
12-24         System.out.println("g31 nach verschieben: " + g2);
12-25     }
12-26 }
```

Substitutionsprinzips mit einem Objekt der Klasse `Punkt3D` aufgerufen worden sein. Um die richtige Methode (2D oder 3D) in der Klasse `Punkt3D` anzuwenden, muss man *zur Laufzeit* entscheiden können, von welchem Typ dieses Objekt wirklich ist: eine Instanz der Klasse `Punkt` oder `Punkt3D`. Oder allgemein gesprochen eine Instanz der Basisklasse beziehungsweise einer Instanz einer abgeleiteten Klasse. Diese Funktionalität gewährleistet der `instanceof`-Operator im Zusammenhang mit einem Wahrheitsausdruck (siehe Abbildung 12.10).

Listing 12.24 zeigt die Nutzung von Geraden mit 2D-Punkten und 3D-Punkten. Man beachte nochmals, dass die Klasse `Gerade` nicht verändert werden musste!

## 12.11 Implizite Aktionen

An mehreren Stellen wurden bisher implizite Aktionen erwähnt. An dieser Stelle sollen diese Aktionen nochmals zusammengefasst werden.

1. Wenn man eine Klasse nicht explizit von einer anderen ableitet, so wird die Klasse implizit von `Object` angeleitet

### Beispiel 12.29:

Für die Klasse

```

12-1  public class Testklasse {
12-2 }
```

findet implizit eine Ableitung von der Klasse `Object` statt, als ob dort stehen würde:

```
12-1 public class Testklasse extends Object {  
12-2 }
```

2. Ein Default-Konstruktor (parameterloser Konstruktor mit leerem Rumpf) wird in einer Klasse implizit definiert, wenn man selbst keinen Konstruktor angegeben hat.

#### **Beispiel 12.30:**

Für die Klasse

```
12-1 public class Testklasse {  
12-2 }
```

findet implizit die Deklaration des Default-Konstruktors statt, als ob dort stehen würde:

```
12-1 public class Testklasse {  
12-2     Testklasse() {}  
12-3 }
```

3. In einer abgeleiteten Klasse wird der parameterlose Konstruktor der Oberklasse implizit aufgerufen, wenn man als erste Aktion im Konstruktor nicht explizit einen Oberklassekonstruktor aufruft. Das gilt auch, wenn man selbst keinen Konstruktor angibt. In dem Fall ruft der dann implizit definierte Default-Konstruktor (siehe vorheriger Punkt) den parameterlosen Konstruktor der Oberklasse auf.

#### **Beispiel 12.31:**

Für die Klasse

```
12-1 public class Testklasse1 {  
12-2 }  
12-3 public class Testklasse2 extends Testklasse1 {  
12-4     Testklasse2() {}  
12-5 }
```

findet implizit der Aufruf der parameterlosen Oberklassekonstruktor statt, als wenn dort stehen würde:

```
12-1 public class Testklasse1 {  
12-2 }  
12-3 public class Testklasse2 extends Testklasse1 {  
12-4     Testklasse2() {  
12-5         super();  
12-6     }  
12-7 }
```

4. Die `toString` Methode wird implizit aufgerufen, wenn aufgrund des Kontextes der Verwendung ein String eines Objektes benötigt wird.

**Beispiel 12.32:**

Für die Klasse

```

12-1 public class Testklasse {
12-2
12-3     public static void main(String[] args) {
12-4         Testklasse k = new Testklasse();
12-5         // hier wird ein String zu dem Objekt k benoetigt
12-6         System.out.println(k);
12-7     }
12-8 }
```

findet implizit der Aufruf der `toString`-Methode statt, als wenn dort stehen würde:

```

12-1 public class Testklasse {
12-2
12-3     public static void main(String[] args) {
12-4         Testklasse k = new Testklasse();
12-5         // hier wird ein String zu dem Objekt k benoetigt
12-6         System.out.println(k.toString());
12-7     }
12-8 }
```

Da die `toString`-Methode in der Klasse `Object` definiert ist, kennt jedes Objekt in Java diese Methode.♦

- Bei Operationen mit unterschiedlichen Typen findet – soweit dies prinzipiell möglich ist – implizit eine Typumwandlung auf den größeren Typ statt (inkl. Upcasts bei Referenztypen)

**Beispiel 12.33:**

Für die Klasse

```

12-1 public class Testklasse1 {
12-2 }
12-3 public class Testklasse2 extends Testklasse1 {
12-4
12-5     public static void main(String[] args) {
12-6         // impliziter Upcast durch die Zuweisung bedingt
12-7         Testklasse1 k = new Testklasse2();
12-8     }
12-9 }
```

findet implizit ein Upcast statt.

```

12-1 public class Testklasse1 {
12-2 }
12-3 public class Testklasse2 extends Testklasse1 {
12-4
12-5     public static void main(String[] args) {
12-6         // expliziter Upcast
12-7         Testklasse1 k = (Testklasse1) new Testklasse2();
12-8     }
12-9 }
```

## 12.12 Zusammenfassung und Hinweise

### Literaturhinweise

Auch hierzu gibt es weitere Erläuterungen und Beispiele in den Büchern [HMHG11], [Ull12] und [Sch10].

### Verstehen

Das Prinzip der Vererbung wird eingesetzt, um für ähnliche, aber nicht gleiche Klassen eine einheitliche Oberklasse angeben zu können. Damit kann man Funktionalität einmalig in der Basisklasse angeben und in den abgeleiteten Klassen nur die spezifischen Änderungen / Ergänzungen angeben. Dies wird eingesetzt, um Programmcode nicht mehrfach angeben zu müssen und damit zusammenhängend Software wartbarer zu machen.

Mächtige Konzepte wie das Überschreiben von Methoden, das Substitutionsprinzip und die Bindung von Methodenaufrufen werden im Zusammenhang mit der Vererbung eingesetzt.

### Kurz und knapp merken

- In einer abgeleiteten Klasse werden alle Attribute und Methoden der Basisklasse geerbt.
- Mit `this` und `super` bezeichnet man das Gesamtobjekt beziehungsweise den ererbten Teil (auch ein Objekt).
- In einem Konstruktor einer abgeleiteten Klasse wird zuerst implizit oder explizit ein Konstruktor der Oberklasse aufgerufen.
- Java besitzt eine sehr umfangreiche Klassenhierarchie mit der Klasse `object` an der Spitze, von der alle Klassen direkt oder indirekt abgeleitet sind.
- Leitet man eine eigene Klasse nicht ab, so ist diese implizit von der Klasse `object` abgeleitet.
- Pakete dienen der Programmstrukturierung auf oberster Ebene.
- Durch die Wahl geeigneter Schlüsselwörter bei der Definition von Klassen, Methoden und Attributen kann die Sichtbarkeit dieser Namen beschränkt werden.
- Polymorphismus ist ein Mittel zur einheitlichen Handhabung von Objekten. Dies kann über verschiedene Mechanismen realisiert werden: Überladen und Überschreiben von Methoden und das Substitutionsprinzip.
- Nach dem Substitutionsprinzip kann an allen Stellen, an denen ein Objekt einer Basisklasse genutzt wird, auch ein Objekt einer davon abgeleiteten Klasse genutzt werden.
- Bei einem Methodenaufruf sucht der Compiler ausgehend von der Aufrufstelle in der Klassenhierarchie aufsteigend nach einer Methode, deren Signatur passend ist zum Methodenaufruf und die mit einem der Schlüsselwörter `static`, `private` oder `final` deklariert ist (statische Bindung). Existiert keine solche Methode, wird zur Laufzeit vom *Gesamtobjekt ausgehend* (vergleiche Substitutionsprinzip) nach einer passenden Methode gesucht (dynamische Bindung).
- Up-Casts bei Referenztypen sind immer möglich (implizit oder explizit), Down-Casts dagegen nur explizit mit einem Cast-Operator. Bei Down-Casts kann es ggfs. zu einem Laufzeitfehler kommen.
- An einigen Stellen in Java passieren Dinge implizit, was man wissen muss:

- An den Stellen, wo ein String-Objekt in einem Ausdruck erwartet wird, aber eine Objektreferenz eines anderen Typs vorliegt, wird implizit die `toString`-Methode dieses Objekts aufgerufen, die einen String erzeugt.
- Ist eine Klasse nicht explizit von einer anderen Klasse abgeleitet, so ist sie implizit von der Klasse `Object` abgeleitet.
- Ist in einer Klassen überhaupt kein Konstruktor explizit angegeben, so besitzt diese Klassen einen impliziten parameterlosen Konstruktor.
- Wird in einem Konstruktor einer abgeleiteten Klasse als erste Anweisung nicht explizit ein Konstruktor der Oberklasse aufgerufen, so wird implizit der parameterlose Konstruktor der Oberklasse aufgerufen. Dieser muss dann auch existieren, explizit oder implizit.

## Häufige Fehler

Im Zusammenhang mit dem Substitutionsprinzip und statischer / dynamischer Bindung und aus Unkenntnis der Wirkung dieser Mechanismen werden häufig schwerwiegende Fehler gemacht, die zu unerwünschten Ergebnissen führen. Siehe dazu zum Beispiel Listing 12.21.

## Übungsfragen

- Wo setzt man das Konzept der Vererbung ein?
- Was bewirkt, wenn ich eine K2 von einer Klasse K1 ableite?
- Ist das möglich: `class K1 extends K2 {} class K2 extends K1 {}`
- Kann ich in Java von mehreren Klassen erben?
- Wie kann ich in einer abgeleiteten Klasse auf Instanzmethoden der Oberklasse zugreifen?
- Wie kann ich in einer abgeleiteten Klasse auf Instanzvariablen der Oberklasse zugreifen?
- Wie kann ich in einer abgeleiteten Klasse auf Klassenmethoden der Oberklasse zugreifen?
- Wie kann ich in einer abgeleiteten Klasse auf Klassenvariablen der Oberklasse zugreifen?
- Welche Bedeutung hat die Klasse `Object`?
- Welche Methoden gibt es u.a. in der Klasse `Object`?

## Reflektion des Stoffs

- Welche Vor- und Nachteile hat eine “Mutter aller Klassen” `Object`?
- Geben Sie ein Szenario an, wie man den Wert einer als private markierten Instanzvariablen eines Referenztyps auch außerhalb der eigenen Klasse verändern kann, ohne selbst auf die Instanzvariable direkt zuzugreifen.
- Gibt es Gründe, dass man nicht grundsätzlich alles (Klassen, Methoden, Attribute) als `public` markiert? Das wäre doch viel einfacher.
- Schauen Sie sich die bisherigen Beispiele zu Typanpassungen bei Referenztypen an. Überlegen Sie sich allgemeine Szenarien, wo Up- bzw. Down-Casts sinnvoll eingesetzt werden können.
- Diskutieren Sie, welche Vor- und Nachteile mit statischer / dynamischer Methodenbindung existieren. In welchen Programmszenarios würde man sich statische Bindung wünschen, in welchen dynamische?
- Modellieren Sie folgende Problemstellung in Java. Eine Person wird beschrieben über ihren Vor- und Nachnamen sowie Geburtsdatum. Studierende sind Personen, die für einen Studiengang (zum Beispiel

Informatik) eingeschrieben sind. Professoren sind Personen, die in einem Studiengang lehren. Erzeugen Sie eine “normale“ Person, zwei Studierende und einen Professor.



# Chapter 13

## Weiterführende Konzepte der Objektorientierten Programmierung

In diesem Kapitel werden einige weiterführende Konzepte zur Objektorientierung (in Java) vorgestellt.

### 13.1 Finalisieren

In Kapitel 7.5 wurde bereits das Schlüsselwort `final` im Zusammenhang mit Variablendefinitionen besprochen. Java kennt das Schlüsselwort `final`, dass sich allgemeiner im Zusammenhang mit der Deklaration von Variablen, Methoden und Klassen verwenden lässt. Die Bedeutung ist bei diesen Möglichkeiten im Detail unterschiedlich, im Wesentlichen wird dadurch aber immer bewirkt, dass keine Veränderungen an diesem Konstrukt mehr möglich sind. Damit kann man erreichen, dass Programmcode klarer / leichter lesbar ist, effizienter durch den Compiler übersetzbare ist oder aber sicherer wird.

#### 13.1.1 Finalisieren von Variablen

Bei der Deklaration einer Variablen (blocklokale, Instanz- oder Klassenvariablen) kann man zusätzlich das Schlüsselwort `final` einsetzen, um zu spezifizieren, dass der Wert der Variablen nach der ersten Zuweisung nicht mehr verändert werden darf. Dies ist zum Beispiel hilfreich, um einem unveränderbaren Wert (eine Konstante) einen leicht verständlichen und sinngebenden Namen zu geben und/oder dem Compiler zusätzliche Informationen zur Optimierung zu geben. Weiterhin überprüft der Compiler, dass tatsächlich im Gültigkeitsbereich dieser finalen Variablen keine Veränderung vorgenommen wird. Würde man dies im Code angeben, so würde dies zu einem Fehler bei der Übersetzung führen. Bei Referenzvariablen ist zu beachten, dass die Referenz zwar dadurch unveränderbar ist, aber nicht die Daten hinter dieser Referenz. Nach den Java Code Conventions werden finale Klassenvariablen mit Großbuchstaben und ggfs. Unterstrichen angegeben. Eine Wertzuweisung an eine finale Variable ist erlaubt bei der Deklaration in Form eines Initialisierungsausdrucks. Geschieht dies dort nicht, so kann eine Zuweisung bei einer blocklokalen Variablen genau einmal im nachfolgenden Programmcode geschehen. Bei Instanz- und Klassenvariablen ist eine Initialisierung außerhalb eines Initialisierungsausdrucks nur einmalig in Konstruktoren erlaubt.

#### Beispiel 13.1:

In dem Beispieldiagramm in Listing 13.1 lassen sich alle drei als final markierten Variablen nicht mehr im Programm ändern. Die Referenz in der Variablen `feld` ist konstant, nicht aber der Inhalt des Feldes, auf den diese Variable zeigt.

Listing 13.1: Beispiel zu finalen Variablen.

```

13-1 class Motor {
13-2     // finale Klassenvariable
13-3     static final int MOTOREL_KLASSE = 1;
13-4     // finale Instanzvariable
13-5     final int maximalDrehzahl = 7600;
13-6
13-7     void starten() {
13-8         maximalDrehzahl = 19000;      // Compilerfehler
13-9
13-10        // blocklokale Variable
13-11        final int[] feld = {1,2,3}; // Variable feld ist final, aber nicht der Feldinhalt
13-12        feld[0] = 4711;           // ist moeglich
13-13        feld = null;          // ist nicht moeglich (Compilerfehler)
13-14    }
13-15 }
```

Listing 13.2: Beispiel zu finalen Methoden.

```

13-1 class Authorization {
13-2     final boolean check(String passwort) {
13-3         ...
13-4     }
13-5 }
13-6
13-7 class Phisher extends Authorization {
13-8     boolean check(String passwort) { // Compilerfehler hier
13-9         // stehle Passwort und rufe Methode der Oberklasse auf
13-10         return super.check(passwort);
13-11     }
13-12 }
```

### 13.1.2 Finalisieren von Methoden

Durch die Angabe von **final** in einer Methodendefinition wird die Methode markiert, dass Sie in einer abgeleiteten Klasse nicht mehr überschrieben werden kann. Dies setzt man zum Beispiel aus Sicherheitsgründen ein, wie nachfolgendes Beispiel zeigt. Da aufgrund des Substitutionsprinzips auch ein Objekt einer abgeleiteten Klasse an Stellen verwendet werden kann, an denen ein Objekt der Basisklasse möglich ist, kann bei Aufrufen mit dynamischer Bindung ein unerwünschter Effekt vermieden werden. Man gewährleistet also, dass genau diese Methode angewandt wird.

#### Beispiel 13.2:

Im Programm in Listing 13.2 könnte ohne die **final**-Markierung die Methode in einer abgeleiteten Klasse überschrieben werden und aufgrund des Substitutionsprinzips auch ein Objekt der abgeleiteten Klasse **Phisher** eingesetzt werden. Durch den Aufruf von **super.check(passwort)** müsste dies zudem nicht unbedingt auffallen.

Listing 13.3: Beispiel zu finalen Klassen.

```

13-1 final class Authorization {
13-2     boolean check(String passwort) {
13-3         ...
13-4     }
13-5 }
13-6
13-7 class Phisher extends Authorization { // Compilerfehler hier
13-8     boolean check(String passwort) {
13-9         // stehle Passwort und rufe Methode der Oberklasse auf
13-10        return super.check(passwort);
13-11    }
13-12 }
```

### 13.1.3 Finalisieren von Klassen

Auch bei der Deklaration von Klassen lässt sich **final** verwenden. Dies bewirkt, dass sich von einer solchen Klasse keine abgeleiteten Klassen bilden lassen. Der Einsatz ist analog zu den eben besprochenen finalen Methoden darin zu sehen, dass unerwünschte Vererbung ausgeschlossen wird.

#### Beispiel 13.3:

Listing 13.3 zeigt ein Beispiel zur Finalisierung einer Klasse. Damit wird erreicht, dass diese Klasse nicht als Basisklasse weitere Klassen genutzt werden kann und somit Methoden überschrieben werden könnten. ♦

Einige Klassen der Java Bibliothek sind final deklariert, zum Beispiel **String**, **Math** (enthält nur finale Klassenvariablen und Klassenmethoden) oder **System**.

## 13.2 Abstrakte Klassen und Methoden

Die in diesem Abschnitt behandelten abstrakten Klassen und Schnittstellen dienen dazu, eine Teilimplementierung oder einen funktionalen Rahmen / Spezifikation vorzugeben, ohne jedoch in der Definition eine komplette Realisierung liefern zu müssen.

### 13.2.1 Abstrakte Klassen

Die Tierklasse enthält eine allgemeine Beschreibung von Tieren. Bis jetzt konnte man damit einzelne Tiere instanziieren, die dann aufgrund der gleichen Beschreibung aller Tiere über diese Klasse sich nur in den Attributwerten unterschieden. Die Aufgabenstellung soll nun dahingehend erweitert werden, dass man darauf aufbauend zusätzlich folgende Tierarten unterscheiden möchte:

- alle Ameisen gehören einer Kaste an: Königin, Männchen, Arbeiterin.
- alle Eulen können fliegen.

Und man möchte auch, dass es nur noch Objekte dieser zwei (konkreten) Tierarten gibt, Tiere als solche also nicht mehr instanzierbar sind. Die Tierklasse soll also nur noch als gemeinsamer Oberbegriff / Basisklasse dienen.

Listing 13.4: Abstrakte Tierklasse.

```

13-1 /** Tier (abstract und abgespeckt)
13-2  */
13-3 public abstract class Tier {
13-4     private String name;
13-5
13-6     public Tier() { this ("unbekannt"); }
13-7     public Tier(String name) { this.name = name; }
13-8     public String getName() { return name; }
13-9     public String toString() { return "Tier " + getName(); }
13-10 }
```

Listing 13.5: Konkrete Ameisenklasse.

```

13-1 /**
13-2  * Ameise
13-3  */
13-4 public class Ameise extends Tier {
13-5     // moegliche Kastenwerte
13-6     public static final String KOENIGIN = "Koenigin";
13-7     public static final String ARBEITERIN = "Arbeiterin";
13-8     public static final String MEANNCHEN = "Maennchen";
13-9     // die Kaste einer Ameise
13-10    private String kaste;
13-11
13-12    // Konstruktor
13-13    Ameise(String kaste) { this.kaste = kaste; }
13-14
13-15    // erzeuge textuelle Darstellung
13-16    public String toString() { return super.toString() + ", Kaste=" + kaste; }
13-17 }
```

Durch Vererbung könnte man jetzt eine Ameisenklasse und eine Eulenklasse von einer Tierklasse ableiten. Die Tierklasse würde alle gemeinsamen Eigenschaften und Fähigkeiten aller Tierarten enthalten, wodurch man sich u.a. Programmiermehrfacharbeit spart (das wurde mit Vererbung ja so genutzt). Die abgeleiteten Klassen würden nur noch die speziellen Eigenschaften der Tierarten zusätzlich definieren (das wurde bei Patentier ja so genutzt). Diese Lösung hätte aber den Nachteil, dass sich auch die Tierklasse instanziieren lässt, man also direkt mit `new Tier()` Tierobjekte erzeugen kann, was man aber gerade verhindern wollte.

Die Lösung für dieses und ähnliche Probleme ist es nun, eine (gemeinsame) Oberklasse mit dem Schlüsselwort **abstract** als abstrakt zu kennzeichnen, mit der Wirkung, dass sich aus dieser Klasse keine Instanzen mehr erzeugen lassen. Ein Java-Compiler würde einen solchen Versuch als Fehler abweisen. Die Klasse dient also nur noch als reine Oberklasse zur Bereitstellung gemeinsamer Funktionalität für andere Klassen.

### Beispiel 13.4:

Listing 13.4 zeigt die abstrakte Tierklasse und Listing 13.5 die konkrete Ameisenklasse. In der Tierklasse sind also weiterhin viele Sachen (einheitlich) realisiert, die alle abgeleiteten Klassen erben. Weiterhin wird durch die Markierung dieser Klasse als abstrakt aber auch erreicht, dass zu der Klasse `Tier` keine Instanzen mehr gebildet werden können. ♦

Eine mit **abstract** gekennzeichnete Klasse nennt man eine **abstrakte Klasse**, eine instanzierbare Klasse nennt man **konkrete Klasse**. Abstrakte Klasse können von anderen abstrakten wie konkreten Klassen erben.

### Beispiel 13.5:

```

13-1      class A { }
13-2 abstract class B extends A {}
13-3 abstract class C extends B {}
13-4      class D extends C {}

```

Abstrakte Klassen werden in Vererbungshierarchien eingesetzt, um eine gemeinsame Funktionalität (für mehrere abgeleitete Klassen) in einer Oberklasse zu kapseln, zu dieser Klasse sollen aber keine Objekte erzeugt werden können.

Von einer abstrakten Klasse kann man weitere abstrakte oder nicht abstrakte Klassen ableiten. Umgekehrt kann eine abstrakte Klasse von einer abstrakten oder auch nicht abstrakten Klasse abgeleitet sein.

Auch wenn sich von einer abstrakten Klasse keine Instanz erzeugen lässt, so ist es doch meist ratsam, einen oder mehrere Konstruktoren zu definieren, wenn diese Klasse in einer Klassenhierarchie verwendet wird. Dadurch kann etwa der geeignete Konstruktoraufruf in der Klassenhierarchie aufwärts aufgerufen werden.

Zu abstrakten Klassen lassen sich Referenzen / Referenzvariablen angeben, auch wenn die Klasse selbst nicht instanzierbar sind.

Hinweis: Man kann die Fähigkeit einer Klasse, dass keine Objekte von ihr erzeugbar sein sollen, auch dadurch erreichen, dass man alle Konstruktoren als **private** deklariert. Dies ist im Zusammenhang mit einem **Singleton**-Entwurfsmuster interessant, auf das hier aber nicht weiter eingegangen wird.

### 13.2.2 Abstrakte Methoden

Neben einer ganzen Klasse kann man auch eine Methode (oder mehrere) in einer Klasse mittels **abstract** kennzeichnen. Ist auch nur eine Methode in einer Klasse als **abstract** deklariert, so muss auch die gesamte Klasse (zusätzlich) als abstrakt deklariert werden. Eine erste Auswirkung davon ist natürlich, dass sich dadurch wiederum keine Instanz einer solchen Klasse erzeugen lässt. Weiterhin erzwingt man damit aber auch, dass in der weiteren Ableitungshierarchie diese Methode konkret realisiert werden muss, und zwar *genau nach der vorgegebenen Signatur*. Für eine konkrete Klasse (die also instanzierbar sein soll) müssen alle in der Vererbungshierarchie aufwärts als **abstract** gekennzeichneten Methoden realisiert werden, entweder in dieser Klasse oder in einer geerbten Klasse.

Der praktische Einsatz dieses Konstrukt ist somit ein anderer als in der Markierung einer ganzen Klasse als abstrakt. Gibt man eine Methode als abstrakt vor, so zwingt man instanzierbare abgeleitete Klassen dazu, diese Methode zu realisieren. Man gibt also in der Oberklasse vor, welche Methoden mit welcher Signatur existieren müssen, ohne aber die konkrete Realisierung dieser Methode angeben zu müssen (vergleiche im Gegensatz zu die Motivation für abstrakte Klassen im letzten Abschnitt). Dies ist zum Beispiel dann notwendig, wenn man in der abstrakten Klasse keine sinnvolle Realisierung der Methode angeben kann, jedes Objekt einer abgeleiteten Klasse aber eine bestimmte Funktionalität über eine einheitliche Methoden bereitstellen soll.

Die Syntax einer abstrakten Methode ist ein Methodenkopf inklusive **abstract**, gefolgt von einem Semikolon statt eines Methodenrumpfs.

Listing 13.6: Abstrakte Tierklasse mit abstrakter Methode `bewegen`.

```

13-1 /**
13-2 */
13-3 public abstract class Tier {
13-4     private String name;
13-5
13-6     public Tier() { this("unbekannt"); }
13-7     public Tier(String name) { this.name = name; }
13-8     public String getName() { return name; }
13-9     public String toString() { return "Tier " + getName(); }
13-10    // abstrakte Methode wie sich ein Tier bewegt
13-11    // An dieser Stelle macht eine Realisierung einer solchen Methode keinen Sinn
13-12    abstract void bewegen();
13-13 }
```

Listing 13.7: Konkrete Ameisenklasse mit realisierter Methode `bewegen` (Version 2).

```

13-1 /**
13-2 * Ameise
13-3 */
13-4 public class Ameise extends Tier {
13-5     // moegliche Kastenwerte
13-6     public static final String KOENIGIN = "Koenigin";
13-7     public static final String ARBEITERIN = "Arbeiterin";
13-8     public static final String MEANNCHEN = "Maennchen";
13-9     // die Kaste einer Ameise
13-10    private String kaste;
13-11
13-12    // Konstruktor
13-13    Ameise(String kaste) { this.kaste = kaste; }
13-14
13-15    // realisiere abstrakte Methode der Oberklasse
13-16    void bewegen() { System.out.println("bewege 6 Beine"); }
13-17
13-18    // erzeuge textuelle Darstellung
13-19    public String toString() { return super.toString() + ", Kaste=" + kaste; }
13-20 }
```

Listing 13.8: Konkrete Eulenklasse mit realisierter Methode `bewegen`.

```

13-1 /**
13-2 * Eule
13-3 */
13-4 public class Eule extends Tier {
13-5     // realisiere abstrakte Methode der Oberklasse
13-6     void bewegen() { System.out.println("bewege Fluegel"); }
13-7     // erzeuge textuelle Darstellung
13-8     public String toString() { return super.toString(); }
13-9 }
```

Listing 13.9: Zooklasse, die die Methode `bewegen` für alle Tiere nutzt.

```

13-1  /**
13-2   * Zoo
13-3   */
13-4 public class Zoo {
13-5     public static void main(String [] args) {
13-6       // Tier t1 = new Tier(); // fuehrt zu Compiler-Fehler
13-7       Ameise a1 = new Ameise(Ameise.ARBEITERIN);
13-8       Eule e1 = new Eule();
13-9       System.out.println(a1);
13-10      System.out.println(e1);
13-11      a1.bewegen();
13-12      e1.bewegen();
13-13
13-14      // als Referenztyp ist Tier aber verwendbar
13-15      Tier [] meineTiere = new Tier[2];
13-16      meineTiere[0] = a1;
13-17      meineTiere[1] = e1;
13-18      meineTiere[0].bewegen();
13-19      meineTiere[1].bewegen();
13-20    }
13-21 }

```

**Beispiel 13.6:**

Listing 13.6 zeigt die abstrakte Tierklasse mit der abstrakten Methode `bewegen`. In der Tierklasse macht eine Realisierung dieser Methode keinen Sinn, weil sich konkrete Tiere unterschiedlich bewegen. Man möchte aber vorgeben, dass alle instanzierbaren Tiere gleich welcher Art eine solche Methode *nach dieser Signatur realisieren müssen*. Die beiden Klassen `Ameise` (Listing 13.7) und `Eule` (Listing 13.8) sind konkrete Klassen und realisieren nach dieser Vorgabe der Oberklasse diese Methode. In der gezeigten Zooklasse (Listing 13.9) sieht man den Nutzen, denn jedes Tier muss jetzt eine Methode `bewegen` realisieren. ♦

## 13.3 Schnittstellen

Möchte man eine bestimmte Funktionalität vorgeben, so wurde im letzten Abschnitt die Möglichkeit von abstrakten Methoden in einer abstrakten Klasse vorgestellt. In Java gibt es aber nur die Möglichkeit der Einfachvererbung, also jede Klasse kann von maximal einer anderen Klasse direkt erben. Möchte man unterschiedliche Funktionalitäten vorgeben, so müsste man diese deshalb alle in *einer* abstrakten Oberklasse angeben, obwohl sie vielleicht garnicht inhaltlich zusammen passen würden.

Ein Beispiel zu dieser Problematik wäre folgendes. Ein Tier muss geimpft werden (Ameisen werden anders geimpft als Eulen), was über eine einheitliche Methode `impfen` realisiert werden soll. Über eine abstrakte Methode in einer abstrakten Tierklasse wäre dies mit den bisher bekannten Konstrukten möglich. Jetzt sollen aber auch Menschen (die keine Tiere sind) auch geimpft werden können, und zwar über eine identisch definierte Methode `impfen`. Man müsste also jetzt vorgeben können, dass sowohl Menschen als auch Tiere geimpft werden können, und zwar auf eine einheitliche Art über eine vorgegebene Methode `impfen`. Dies kann also nicht mehr in einer abstrakten Tierklasse vorgegeben werden, da die Menschklasse davon nicht abgeleitet sein kann / soll. Man könnte als Ausweg für diesen Fall noch eine abstrakte Basisklasse `Impfen` mit genau einer abstrakten Methode `impfen` definieren, davon die Tier- und Menschenklasse ableiten und von der Tierklasse dann wiederum die Ameisenklasse etc. Das wäre keine gute Idee, da Menschen und Tiere bis auf das Impfen keine Gemeinsamkeiten haben.

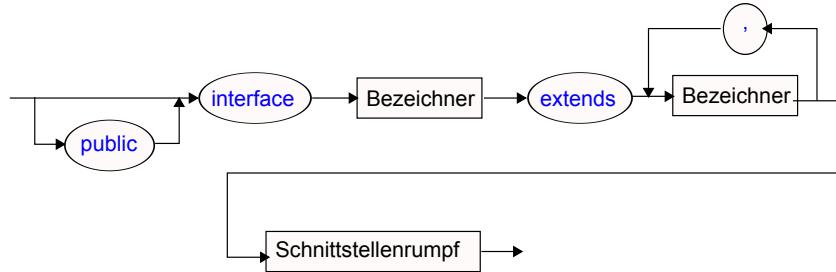
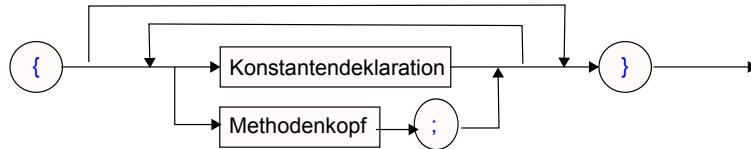
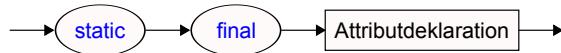
**Schnittstellendefinition****Schnittstellenrumpf****Konstantendeklaration**

Figure 13.1: Schnittstellendefinition.

Endgültig nicht mehr realisierbar wäre dann aber die Problematik, dass neben dem Impfen auch noch der Zoodirektor Tiere und Gehege verkaufen möchte, wozu eine Preisermittlung möglich sein muss über eine Methode `preisErmitteln`. Würde man dies in die oben besprochene abstrakte Impfklasse (was bereits so eine schlechte Lösung war!) mit packen, müsste eine Menschklasse dann auch eine `preisErmitteln` Methode realisiert haben, was aber Unsinn wäre. Eine Aufteilung in zwei abstrakte Klassen, eine für das Impfen und die zweite für die Ermittlung eines Preises, macht aber auch keinen Sinn, weil man eine Klasse in Java nicht von zwei Klassen direkt ableiten kann. Eine saubere Lösung wäre mit den derzeit bekannten Konstrukten nicht möglich.

Java kennt das Konstrukt der **Schnittstelle**. Abbildung 13.1 beschreibt die Syntax einer Schnittstellendefinition. In einer Schnittstelle wird eine Funktionalität beschrieben / vorgegeben, ohne sie zu realisieren. Während in einer abstrakten Klasse Realisierungen von einigen Methoden vorgenommen werden können, ist eine Schnittstelle eine rein beschreibende Spezifikation. Man kann damit aus funktioneller Sicht eine Schnittstelle als eine Art abstrakte Klasse ansehen, in der aber nur abstrakte Methoden definiert sind.

In einer Schnittstelle werden Methodensignaturen vorgegeben, die durch ein Semikolon abgeschlossen werden. Im Gegensatz zu einer abstrakten Klasse ist es also in Schnittstellen *grundsätzlich nicht möglich*, zumindest einige Methoden auch zu implementieren. Zusätzlich zu Methoden kann man auch Attribute deklarieren (zum Beispiel für globale Konstanten, die inhaltlich zu dieser Schnittstelle gehören), die automatisch immer implizit die Attribute `public static final` besitzen, also wie öffentlich sichtbare finale Klassenvariablen behandelt werden. Eine in einer Schnittstelle angegebene Methode muss bei einer Realisierung in einer Klasse immer als `public` angegeben werden. Vereinbarungsgemäß haben Namen von Schnittstelle die Endung `able`, was verdeutlicht soll, dass diese Schnittstelle beschreibt, wozu jemand/etwas in der Lage ist. (Die Namensgebung in den Beispielen wird dadurch denglisch).

In einer Klasse kann man dann explizit angeben, dass diese Klasse die durch die Schnittstelle beschriebene Funktionalität realisiert und man muss dann auch diese Funktionalität spätestens in einer konkreten Klasse realisieren. Ein wesentlicher Unterschied zu abstrakten Klassen ist es aber nun, dass eine Klasse *mehrere*

Listing 13.10: Schnittstelle **Impfeable**.

```

13-1 /**
13-2 * Schnittstelle Impfeable
13-3 */
13-4 interface Impfeable {
13-5     public void impfen();
13-6 }
```

Listing 13.11: Schnittstelle **PreisErmitteable**.

```

13-1 /**
13-2 * Schnittstelle zur Ermittlung eines Preises
13-3 */
13-4 interface PreisErmitteable {
13-5     double preisErmitteln();
13-6 }
```

Schnittstellen gleichzeitig realisieren kann. Man kann also mehrere Funktionalitäten (`impfen` und `Preis ermitteln`), die nichts miteinander zu tun haben, in unterschiedlichen Schnittstellen beschreiben, und Klassen können dann angeben, welche Funktionalitäten sie realisieren.

### Beispiel 13.7:

Konkret kann man also jetzt eine Schnittstelle `Impfeable` definieren und eine Schnittstelle `PreisErmitteable`. Die abstrakte Tierklasse oder aber eine konkrete Tierklasse wie `Ameise` würden so markiert, dass diese Schnittstelle dort realisiert wird und dann auch eine Implementierung für die Funktionen liefern. Listing 13.10 - 13.13 zeigen die Umsetzung dieser Ideen.

Gleichzeitig könnte die Menschklasse ebenfalls die Schnittstelle `Impfeable` realisieren, müsste aber *nicht* auch die Schnittstelle `PreisErmitteable` realisieren.

Ähnlich wie man Klassen von anderen Klassen ableiten kann, können Schnittstellen auch von anderen Schnittstellen abgeleitet werden (aber nicht von Klassen). Dies geschieht analog zur Klassenvererbung

Listing 13.12: Abstrakte Tierklasse, die zwei Schnittstellen realisiert.

```

13-1 /**
13-2 */
13-3 public abstract class Tier implements Impfeable, PreisErmitteable {
13-4     private String name;
13-5
13-6     public Tier() { this("unbekannt"); }
13-7     public Tier(String name) { this.name = name; }
13-8     public String getName() { return name; }
13-9     public String toString() { return "Tier " + getName(); }
13-10    // abstrakte Methode wie sich ein Tier bewegt
13-11    // An dieser Stelle macht eine Realisierung einer solchen Methode keinen Sinn
13-12    abstract void bewegen();
13-13 }
```

Listing 13.13: Konkrete Ameisenklasse mit realisierten Schnittstellenmethoden (Version 3).

```

13-1  /**
13-2   * Ameise
13-3   */
13-4  public class Ameise extends Tier implements Impfeable, PreisErmitteable {
13-5      // moegliche Kastenwerte
13-6      public static final String KOENIGIN = "Koenigin";
13-7      public static final String ARBEITERIN = "Arbeiterin";
13-8      public static final String MEANNCHEN = "Maennchen";
13-9      // die Kaste einer Ameise
13-10     private String kaste;
13-11
13-12     // Konstruktor
13-13     Ameise(String kaste) { this.kaste = kaste; }
13-14
13-15     // erzeuge textuelle Darstellung
13-16     public String toString() { return super.toString() + ", Kaste=" + kaste; }
13-17
13-18     // realisiere abstrakte Methode der Oberklasse
13-19     void bewegen() { System.out.println("bewege 6 Beine"); }
13-20
13-21     // Schnittstellenmethode realisieren
13-22     public void impfen() { System.out.println("ich werde geimpft"); }
13-23
13-24     // Schnittstellenmethode realisieren
13-25     public double preisErmitteln() { return 0.01; } // Ameisen sind billig
13-26 }
```

auch hier mit `extends` im Kopf der Schnittstellendeklaration. Die Bedeutung ist ebenfalls analog zur Klassenvererbung zu sehen: Die abgeleitete Schnittstelle erbt alle Attribute und Methodenspezifikationen der Basisschnittstelle. Im Gegensatz zu Klassen ist jetzt aber auch eine Mehrfachvererbung bei Schnittstellen möglich, die durch Kommas getrennt aufgezählt werden. Auf die dadurch potentiellen Probleme wird hier nicht eingegangen (Mehrdeutigkeiten bei Namen).

Eine Schnittstelle `x` mit der Markierung `public` muss ebenso wie bei einer Klasse in einer Datei mit Namen `x.java` stehen und wird auch mit übersetzt. Eine Schnittstelle stellt einen eigenen (Referenz-)Typ dar. Ein solcher Typ kann also zur Deklaration von Variablen oder Parametern verwendet werden. Da keine Instanzen einer Schnittstelle erzeugt werden können, spielen diese Schnittstellentypen nur eine Rolle im Zusammenhang mit Typanpassungen (vergleiche Kapitel 12.2).

Zur Nutzung von Schnittstellen kann man in einer Klassendefinitionen nun zusätzlich angeben, dass diese Klasse eine oder auch mehrere Schnittstellen realisiert / implementiert (siehe Abbildung 13.2 zur Syntax). Gibt eine Klasse an, dass Sie eine Schnittstelle X realisiert, so müssen alle in der Schnittstelle angegebenen Methoden in der Klasse realisiert werden (dann ist es eine konkrete Klasse), oder aber die Klasse muss selber `abstract` sein um zu markieren, dass in abgeleiteten Klassen noch Methoden zu realisieren sind.

### Beispiel 13.8:

Listing 13.14 zeigt die von der Schnittstelle `Impfen` abgeleitete Schnittstelle `Piekseable`, die eine weitere Methode vorschreibt. Wer also die Schnittstelle `Piekseable` realisiert, muss demzufolge zwei Methoden realisieren: `impfen` und `desinfizieren`.

Die abstrakte Klasse `Tier` wird diesmal nicht verändert, denn nicht jedes Tier bekommt eine Piekimpfung, sondern in unserem Fall nur Eulen. Deshalb hat die Eulenklasse in Listing 13.15 den Zusatz `implements`

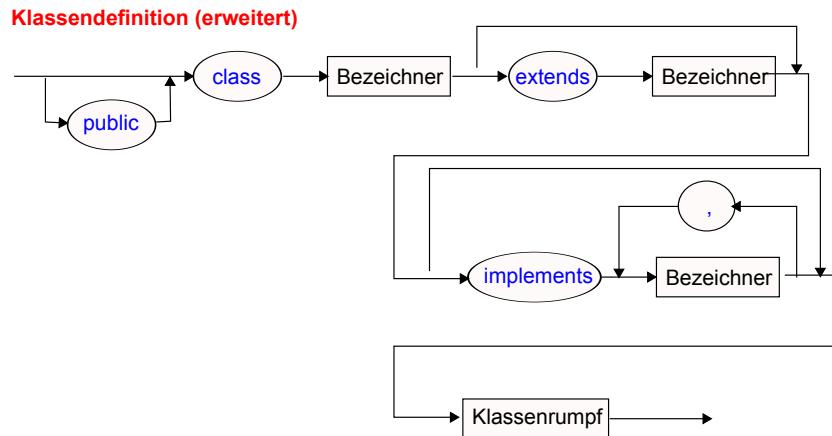


Figure 13.2: Erweiterung der Klassendefinition zur Realisierung einer oder mehrerer Schnittstellen.

Listing 13.14: Schnittstelle **Piekseable**, von der Schnittstelle **Impfeable** abgeleitet.

```

13-1 /**
13-2 * Impfung mit pieken (Nadel). Vorher muss die Stelle desinfiziert werden.
13-3 */
13-4 public interface Piekseable extends Impfeable {
13-5     // vorher muss desinfiziert werden
13-6     public void desinfizieren();
13-7 }
  
```

Listing 13.15: Konkrete Eulenklasse mit drei realisierten Schnittstellenmethoden.

```

13-1 /**
13-2 * Eule
13-3 */
13-4 public class Eule extends Tier implements Piekseable, PreisErmitteable {
13-5     // erzeuge textuelle Darstellung
13-6     public String toString() { return super.toString(); }
13-7     // realisiere abstrakte Methode der Oberklasse
13-8     void bewegen() { System.out.println("bewege Fluegel"); }
13-9     public void impfen() { System.out.println("hier mit Nadel"); }
13-10    public void desinfizieren() { System.out.println("wir desinfizieren die Stelle"); }
13-11    public double preisErmitteln() { return 5000.0; } // Eulen sind teuer
13-12 }
  
```

Listing 13.16: Cast auf einen Schnittstellentyp.

```

13-1 // Schnittstelle
13-2 public interface Piekseable extends Impfeable {
13-3     public void desinfizieren();
13-4 }
13-5
13-6 // Klasse, die diese Schnittstelle realisiert
13-7 public class Eule extends Tier implements Piekseable {
13-8     ...
13-9     void bewegen() { System.out.println("bewege Fluegel"); }
13-10    public void desinfizieren() { System.out.println("wir desinfizieren die Stelle"); }
13-11 }
13-12
13-13 public class Zoo {
13-14
13-15     public static void main(String[] args) {
13-16         Eule e = new Eule();           // Eulenobjekt erzeugen
13-17         Piekseable p = (Piekseable)e; // cast auf Schnittstelle
13-18
13-19         e.desinfizieren();        // moeglich
13-20         p.desinfizieren();        // ueber Schnittstellenreferenz auch moeglich
13-21
13-22         e.bewegen();            // moeglich
13-23         p.bewegen();            // Fehler: nicht moeglich
13-24     }
13-25 }
```

Piekseable, und nicht die Tierklasse. ♦

Während von Schnittstellen selbst keine Instanzen erzeugt werden können, können sehr wohl Typanpassungen auf einen Schnittstellentyp vorgenommen werden, wenn zum Beispiel eine Klasse eine Schnittstelle realisiert. Damit ist dann die Sichtweise auf die Funktionalität der Schnittstelle eingeschränkt.

### Beispiel 13.9:

Listing 13.16 zeigt Beispiele zur Nutzung der Kovarianz bei Feldern und Möglichkeiten (und Fehler) bei Cast-Operationen. ♦

Schnittstellen spielen im Software-Design eine wichtige Rolle. Darüber lässt sich sauber die funktionale Spezifikation von der Implementierung trennen. Ist eine Schnittstelle definiert, so kann der Nutzer dieser Schnittstelle unabhängig vom Implementierer der Schnittstelle arbeiten.

Weiterhin garantiert eine Schnittstellenspezifikation die *Austauschbarkeit* der Implementierung. Programmiert man als Nutzer einer Schnittstelle ausschließlich gegen diese Schnittstelle (ohne die internen Details der Realisierung zu nutzen), so lässt sich die Realisierung der Schnittstelle beliebig austauschen. Das setzt aber natürlich voraus, dass man keine Eigenschaften der realisierten Klasse nutzt, die über die Schnittstellenspezifikation hinausgehen, also an der Spezifikation vorbei arbeitet.

Hinweis: Ab Java 8 können in Schnittstellen sogenannte Default-Methoden nicht nur spezifiziert, sondern sogar *realisiert* werden.

Listing 13.17: Eigene Klasse zur Kapselung von primitiven Werten.

```

13-1 class MeinInteger {
13-2     int wert;
13-3     MeinInteger(int wert) { this.wert = wert; }
13-4     int getWert() { return wert; }
13-5     void setWert(int wert) { this.wert = wert; }
13-6 }
13-7
13-8 class Nutzung {
13-9     public static void main(String[] args) {
13-10         MeinInteger i = new MeinInteger(5);
13-11         meineMethode(i);
13-12     }
13-13
13-14     void meineMethode(MeinInteger i) {
13-15         i.setWert(6); // veraendere Wert
13-16     }
13-17 }
```

## 13.4 Besondere Arten von Klassen

Es gibt von der Bedeutung beziehungsweise der Angabemöglichkeit einige spezielle Arten von Klassen, die in diesem Abschnitt behandelt werden.

### 13.4.1 Wrapper-Klassen \*

Will man Werte eines primitiven Typs in einem Methodenaufruf übergeben, so wird aufgrund der call-by-value Strategie eine Kopie des Wertes übergeben. Deshalb kann innerhalb der Methode der Originalwert auch grundsätzlich nicht verändert werden. Bei Objekten liegt diese Restiktion prinzipiell nicht vor: über die Referenz ließe sich das Originalobjekt verändern (siehe dazu aber Kapitel 12.4).

Möchte man aber ein solches Verhalten (Veränderbarkeit über Methodenaufrufe) auch für Werte eines primitiven Typs erreichen, so könnte man eine eigene Klasse schreiben, die Werte eines primitiven Typs kapselt. Listing 13.17 zeigt eine Möglichkeit der Umsetzung.

Da diese Problematik nur für primitive Typen auftritt und sie häufiger in der Nutzung auftritt, gibt es in der Java Klassenbibliothek zu jedem primitiven Typ bereits eine vordefinierte sogenannte **Wrapper-Klasse**. Die selbsterklärenden Namen sind `Boolean`, `Character`, `Byte`, `Short`, `Integer`, `Long`, `Float` und `Double`. Diese Klassen besitzen u.a. einen Konstruktor mit einem Wert des zugehörigen primitiven Typs sowie eine Reihe von Methoden und Konstanten. Zu den Details sei auf die entsprechende Dokumentation [Java] verwiesen.

Java kennt im Zusammenhang mit der Nutzung von Wrapper-Klassen das Prinzip des **Auto-Boxing**. Wird etwa als Parametertyp in einer Methode ein Wrapper-Typ angegeben, als Argument aber ein Wert des entsprechenden primitiven Typs übergeben, so wandelt der Java-Compiler diesen Wert automatisch in ein Wrapper-Objekt um. Ebenso führt etwa die Zuweisung eines Wrapper-Objekts an eine Variable des entsprechenden primitiven Typs zu einem Entpacken des Wrapper-Objekts.

#### Beispiel 13.10:

In Listing 13.18 wird der Wert `4711` vom Compiler automatisch in ein Objekt vom Typ `Integer` umgewandelt, das den Wert `4711` kapselt. Umgekehrt wird bei der Zuweisung von `iwrap` an `n` automatisch ein Unboxing durchgeführt. ♦

Listing 13.18: Beispiel zum Auto-Boxing.

```

13-1 // automatische Boxing
13-2 void f(Integer iwrap) { ... }
13-3 f(4711);
13-4
13-5 // automatisches Unboxing
13-6 Integer iwrap = new Integer(4711);
13-7 int n = iwrap;

```

### 13.4.2 Geschachtelte Klassen

Manchmal benötigt man nur lokal an einer Stelle im Programm (und meistens dann auch nur für eine kleine Aufgabe) eine eigene Klasse, also eine Art Hilfsklasse. Java kennt drei verschiedene Möglichkeiten eine solche anzugeben, je nachdem, in welcher Strukturtiefe man eine solche Klasse angeben will: innerhalb einer Klasse, innerhalb eines Blocks (zum Beispiel einem Methodenrumpf) und innerhalb eines Ausdrucks.

#### Innere Klasse

In Java ist das ineinanderschachteln von Klassen erlaubt. Innerhalb einer Klasse können an den Stellen, wo Attribut- und Methodendefinitionen erlaubt sind, auch **innere Klassen** definiert werden. Dabei hat die innere Klasse direkten Zugriff auf Attribute und Methoden der äußeren Klasse über den einfachen Namen und umgekehrt auch. Allerdings sind keine Definitionen von Klassenvariablen und -methoden erlaubt. Analog zu Methoden sind bei inneren Klassen alle Modifikatoren erlaubt mit analoger Bedeutung wie bei Methoden. Listing 13.19 zeigt ein Beispiel zur Nutzung.

Auf statische innere Klassen wird hier nicht eingegangen, hier sei bei Interesse auf die Literatur verwiesen.

#### Lokale Klasse

Analog zur Definition einer Klasse in einer umfassenden Klasse lassen sich auch Klassen innerhalb von Blöcken definieren, also eine weitere Stufe tiefer in der Strukturhierarchie. Dies sind die sogenannten **lokalen Klassen**. Listing 13.20 zeigt eine Beispiel zur Definition und Nutzung. Der Gültigkeitsbereich ist auf den unschließenden Block beschränkt.

Beim Zugriff auf Variablen der Instanz, der Klasse und des umschließenden Blocks gibt es gewisse Einschränkungen. Ebenso wie bei inneren Klassen sind keine statischen Variablen oder Methoden erlaubt. Wird eine lokale Klasse innerhalb einer Instanzmethode definiert, hat sie Zugriff auf die Instanz- und Klassenvariablen der umschließenden Klasse. Für lokale Klassen innerhalb von Klassenmethoden ist natürlich nur der Zugriff auf Klassenvariablen möglich. Der Zugriff auf blocklokale Variablen ist einschränkender. Ein solcher Zugriff ist nur möglich, wenn die blocklokale Variable als `final` deklariert ist.

#### Anonyme Klasse

Die nächste Stufe ist die Möglichkeit der Definition einer Klasse als Teil einer Deklarations- oder Ausdrucksanweisung. Damit entsteht eine **anonyme Klasse**. Anonyme Klasse – Nomen est Omen – haben keinen Namen und können demzufolge auch keinen Konstruktor haben. Während der Programmausführung wird an dieser Stelle genau ein Objekt dieser Klasse erzeugt. Da die Klasse keinen Namen hat, wäre eine Instanziierung an anderen Stellen auch garnicht möglich.

Anonyme Klassen werden oft im Zusammenhang mit dem `new`-Operator verwendet. In diesem Zusammenhang werden sie üblicherweise eingesetzt, um eine existierende Klasse in der Funktionalität zu erweitern oder zu

Listing 13.19: Beispiel für eine innere Klasse.

```

13-1  /**
13-2   * Beispiel fuer eine innere Klasse
13-3   */
13-4  public class BeispielInnereKlasse {
13-5
13-6      private MeinInteger wert;
13-7      private double wert2 = 2;
13-8
13-9      // innere Klasse
13-10     class MeinInteger {
13-11         private int meinWert;
13-12         public MeinInteger(int wert) {
13-13             this.meinWert = wert;
13-14             wert2 = 2*wert;    // hier Zugriff auf ein Attribut der aeusseren Klasse
13-15         }
13-16         public int getMeinWert() { return meinWert; }
13-17         public void setMeinWert(int wert) { this.meinWert = wert; }
13-18     }
13-19
13-20     // Konstruktor aeussere Klasse
13-21     BeispielInnereKlasse(int wert) {
13-22         // Nutzung innere Klasse
13-23         this.wert = new MeinInteger(wert);
13-24         this.wert.meinWert = 4711;    // nicht schoen, aber prinzipiell moeglich
13-25     }
13-26 }
```

Listing 13.20: Beispiel für eine lokale Klasse.

```

13-1  /**
13-2   * Beispiel zur Nutzung einer lokalen Klasse
13-3   */
13-4  public class BeispielLokaleKlasse {
13-5
13-6      // Methode der aeusseren Klasse
13-7      public void untersuche() {
13-8          // final ist noetig, um die Variable in der lokalen Klasse nutzen zu koennen
13-9          final int wert2 = 0;
13-10
13-11         // lokale Klasse (Teil des Methodenrumpfs)
13-12         class MeinInteger {
13-13             private int meinWert;
13-14             public MeinInteger(int wert) {
13-15                 this.meinWert = wert * wert2;    // hier Zugriff auf blocklokale Variable
13-16             }
13-17             public int getMeinWert() { return meinWert; }
13-18             public void setMeinWert(int wert) { this.meinWert = wert; }
13-19         }
13-20
13-21         // weiter im Methodenrumpf
13-22         MeinInteger wert = new MeinInteger(5);
13-23         wert.setMeinWert(4711);
13-24     }
13-25 }
```

Listing 13.21: Beispiel für eine anonyme Klasse.

```

13-1 /**
13-2 * Beispiel zur Nutzung einer anonymen Klasse
13-3 */
13-4 public class BeispielAnonymeKlasse {
13-5
13-6     public static void main(String [] args) {
13-7         // Ausdruck
13-8         new MeinInteger(5) {
13-9             // hier ist schon die anonyme Klasse
13-10            // ueberschreibe Methode der Ursprungsklasse, z.B. fuer einen besondere Zweck
13-11            public String toString() { return "anonym: " + this.getMeinWert(); }
13-12        };
13-13    }
13-14 }
13-15
13-16 // weitere Klasse
13-17 class MeinInteger {
13-18     private int meinWert;
13-19     public MeinInteger(int wert) {
13-20         meinWert = wert;
13-21         System.out.println(this);
13-22     }
13-23     public int getMeinWert() { return meinWert; }
13-24     public void setMeinWert(int wert) { this.meinWert = wert; }
13-25     public String toString() { return "Wert ist " + meinWert; }
13-26 }
```

Listing 13.22: Beispiel für eine anonyme Klasse mit der Nutzung von Schnittstellen.

```

13-1 // fuge einen ActionListener zu einem Menu-Eintrag hinzu
13-2 menu.addActionListener(new ActionListener() {
13-3     // Durch Druecken der Menu-Anzeige wird actionPerformed aufgerufen
13-4     public void actionPerformed(ActionEvent e) {
13-5         ...
13-6     }
13-7 });
```

verändern (siehe Beispiel). Ein weiteres Einsatzgebiet ist die Implementierung einer Schnittstelle, ohne diese mittels **extends** erweitern zu müssen.

### Beispiel 13.11:

Listing 13.21 gibt ein Beispiel einer anonymen Klasse an, in dem die Funktionalität der **toString**-Methode der Ursprungsklasse verändert werden soll. ♦

### Beispiel 13.12:

Ein Beispiel im Zusammenhang mit der Nutzung von Schnittstellen zeigt Listing 13.22, das aus dem Bereich der Programmierung von Benutzeroberflächen stammt. Die Schnittstelle **ActionListener** aus der Java Klassenbibliothek fordert eine Methode **void actionPerformed(ActionEvent e)**. Eine Schnittstelle kann nicht instanziert werden. Bei der Erzeugung des Objekts von diesem Schnittstellentyp wird aber

gleichzeitig durch die anonyme Klasse diese geforderte Methode realisiert, wodurch eine Instanzierung zu dieser anonymen Klasse möglich ist. ♦

## 13.5 UML Klassendiagramme \*

Die **Unified Modeling Language** (UML) ist eine weitgehend schematisch und grafikorientierte BeschreibungsSprache zur Spezifikation, aber auch Dokumentation von Software-Systemen. In ihr sind verschiedene Diagrammarten definiert, die jeweils spezifische Sichtweisen auf ein Software-System erlauben. Zu diesen Diagrammarten zählt das in Kapitel 2.4.3 kurz eingeführte Aktivitätsdiagramm sowie das nachfolgende vorgestellte Klassendiagramm. Daneben gibt es aber noch eine Vielzahl weiterer Diagrammarten, auf die hier nicht eingegangen wird.

Ein **Klassendiagramm** der UML dient zur Darstellung von Klassen und deren Beziehungen zueinander. Im Folgenden werden nicht alle Möglichkeiten der UML Klassendiagramme angegeben, sondern lediglich die wesentlichen Elemente aufgezeigt, die für die bisher vorgestellten Java-Konstrukte relevant sind. Der allgemeine Aufbau zur Beschreibung einer Klasse ist in Abbildung 13.3 gegeben. Ein solche Angabe besteht also aus drei Teilen: dem Klassennamen, der Angabe von Attribut en und der Angabe von Methoden.

Attributangaben folgen der Syntax:

```
<Attribut> ::= [ <Sichtbarkeit> ] <Name> [ : <Typ> ] [ = <Initialisierungswert> ]
<Sichtbarkeit> ::= ε | + | - | # | ~
```

<Name> entspricht dem Attributnamen und <Typ> gibt einen Java-Typ an. Bezuglich der Sichtbarkeit haben die Symbole folgende Bedeutung:

- Eine fehlende Angabe lässt die Sichtbarkeit unspezifiziert.
- + entspricht einer Angabe **public** in Java.
- - entspricht einer Angabe **private** in Java.
- # entspricht einer Angabe **protected** in Java.
- ~ entspricht der Sichtbarkeit nur im gleichen Paket.

Methodenangaben folgen der Syntax:

```
<Methode> ::= [ <Sichtbarkeit> ] <Name> ( [ <Parameterliste> ] ) [ : <Typ> ]
```

<Sichtbarkeit>, <Name> und <Typ> sind analog zu Attributen. Die Parameterliste entspricht in einer vereinfachten Form durch Kommas getrennte Angaben <Name> : <Typ>. Als Besonderheit gegenüber einer Java-Angabe ist also insbesondere zu sehen, dass der (optionale) Ergebni styp angehängt wird.

Klassenattribute und Klassenmethoden werden immer unterstrichen, um diese gegenüber Instanzvariablen / Instanzmethoden kenntlich zu machen.

Um Vererbungsbeziehungen kenntlich zu machen, wird ein Linie von einer abgeleiteten Klasse zur Oberklasse gezogen. Die Pfeilspitze an der Oberklasse ist ein nicht ausgefülltes Dreieck. Es sei hier nur erwähnt, dass es eine Reihe unterschiedlicher Assoziationen zwischen Klassen geben kann (zum Beispiel ist-Teil-von), die durch unterschiedliche Linienarten und Pfeilsymbolen kenntlich gemacht werden.

Abbildung 13.4 zeigt das komplette Zoo-Beispiel als UML Klassendiagramm (ohne weitere Assoziationen).

<b>Klassenname (fett, zentriert)</b>
Attribute (linksbündig)
Methoden (linksbündig)

Figure 13.3: Schematischer Aufbau eines Klassendiagramms zu einer Klasse.

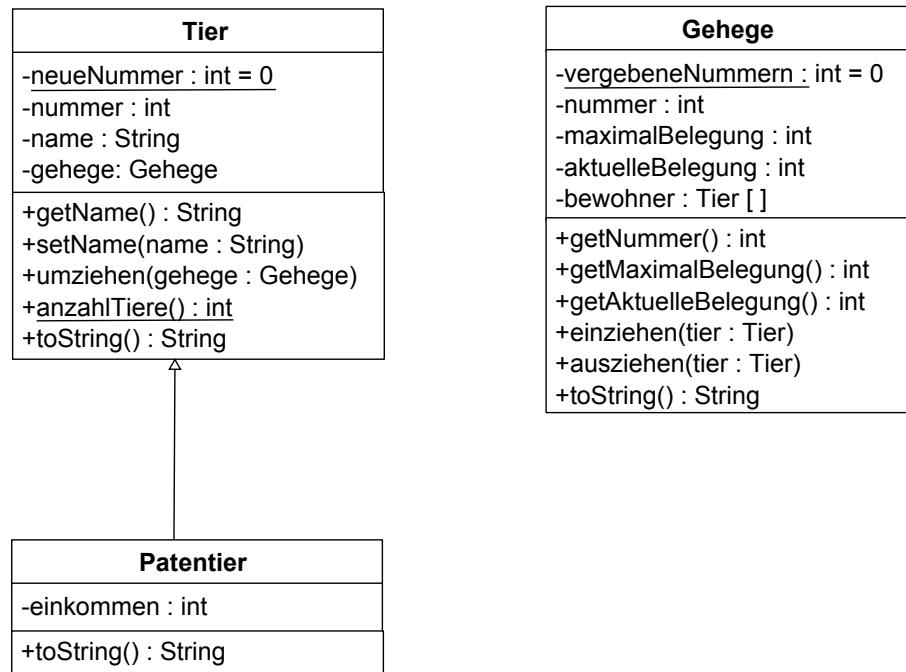


Figure 13.4: Klassendiagramm zur Zoo-Anwendung.

## 13.6 Zusammenfassung und Hinweise

### Literaturhinweise

Auch hierzu gibt es weitere Erläuterungen und Beispiele in den Büchern [HMHG11], [Ull12] und [Sch10]. Es sei auf versiegelte Klassen (*sealed classes*) hingewiesen. Diese erlauben – ergänzend zu finalen Klassen – Zugriffs-/Vererbungsmöglichkeiten anderer Klassen zu steuern.

### Verstehen

Abstrakte Klassen / Methoden spielen bei Nutzung von Vererbung eine wichtige Rolle, Schnittstellen insbesondere in der Entwicklung größerer Softwareprodukte (im Team). Über das Finalisieren und über die gezielte Vergabe von Zugriffsrechten und Paketbildung kann man Software sicherer machen.

### Kurz und knapp merken

- Mit dem Finalisieren von Variablen, Methoden oder Klassen verhindert man eine Veränderung / Überschreiben.
- Abstrakte Klassen ohne explizite abstrakte Methoden dienen dazu, gemeinsame Funktionalität für abgeleitete Klassen in einer Oberklasse bereitzustellen. Durch die Eigenschaft abstrakt lässt sich eine solche Klasse aber nicht instanziiieren. Nutzt man in einer abstrakten Klasse zusätzlich auch abstrakte Methoden, so will man damit in abgeleiteten Klassen eine Realisierung einer Methode nach einer definierten Signatur erzwingen.
- Schnittstellen dienen der funktionalen Beschreibung und spielen eine wichtige Rolle im Software-Design, da über sie eine Trennung zwischen Spezifikation und Implementierung möglich ist. Schnittstellen in Java kennen Mehrfachvererbung.

- Mit Wrapper-Klassen lassen sich Werte eines primitiven Typs kapseln. Objekte einer solchen Wrapper-Klasse können somit als Referenz an Methoden übergeben und dort verändert werden.
- Über geschachtelte, lokale und anonyme Klassen können in der Strukturierungshierarchie auf Klassen-, Methoden- und Blockebene Klassen definiert oder in der Funktionalität verändert werden. Dieses wird meist im Zusammenhang mit kleinen Klassen genutzt, die nur lokal genutzt werden.
- Pakete sind in der Strukturierungshierarchie an oberster Stelle und stellen jeweils einen eigenen Namensraum dar.
- Über Zugriffsrechte lassen sich Klassen, Schnittstellen, Instanz- und Klassenvariablen sowie Methoden in der Sichtbarkeit einschränken / steuern.
- Ebenso wie bei primitiven Typen gibt es auch bei Referenztypen die Möglichkeit der Typanpassung in Form von Downcasts und Upcasts. Dabei sind je nach Gebrauch Typanpassungen möglich, nicht möglich (Übersetzungsfehler) oder werden erst zur Laufzeit erkannt.

## Übungsfragen

- Welche cast-Operationen sind bei Referenztypen erlaubt?
- Was ist ein Upcast bei Referenztypen? Beispiel!
- Was ist ein Downcast bei Referenztypen? Was ist dabei wichtig? Beispiel!
- Welche Zugriffsrechte kann ich für Klassen vergeben? Bedeutung?
- Welche Zugriffsrechte kann ich für Instanzvariablen vergeben? Bedeutung?
- Was ist der Unterschied von Klassen mit und ohne public?
- Worauf muss ich bei public Klassen achten?
- Was gilt, wenn die Klasse mit/ohne public ist bzgl. der Zugriffsrechte der Instanzvariablen?
- Was versteht man unter Datenkapselung?
- Was sind Getter- und Setter-Methoden? Nutzung?
- Wieso sollte man Attribute eher mit eingeschränkten Zugriffsrechten versehen?
- Was versteht man unter dem Begriff überladen? Sinnvoller Einsatz? Beispiel!
- Was versteht man unter dem Begriff überschreiben? Sinnvoller Einsatz? Beispiel!
- Was versteht man unter Polymorphismus? Umsetzung?
- Was besagt das Substitutionsprinzip? Anwendung?
- Was bedeutet dynamische / statische Bindung? Beispiel!
- Wie setzt man dynamische Bindung sinnvoll ein? Beispiel!
- Wie setzt man statische Bindung sinnvoll ein? Beispiel!
- Wozu setzt man den Operator instanceof sinnvoll ein?
- Was lässt sich alles finalisieren?
- Was ist der Sinn hinter dem Finalisieren einer Variablen?
- Was ist der Sinn hinter dem Finalisieren einer Methode?
- Was ist der Sinn hinter dem Finalisieren einer Klasse?
- Welche Methoden kann man finalisieren? Instanz- / Klassenmethoden? Beide?
- Wieso gibt es abstrakte Klassen?
- Was steckt man in der Modellierung in eine abstrakte Klasse?
- Was passiert, wenn man zu einer abstrakten Klasse K ein new K() macht?
- Was versteht man unter einer konkreten Klasse? Gegenteil?
- Kann eine abstrakte Klasse von einer konkreten Klasse erben? Umgekehrt?
- Kann man in einer (nicht-instanziierbaren) abstrakten Klasse einen Konstruktor angeben? Wieso?

- Abstrakte Klasse K1, konkrete davon Klasse K2. Ist K1 k1 = new K2() möglich? Wenn nein, wieso?  
Wenn ja, Bedeutung?
- Wieso gibt es abstrakte Methoden?
- K2 von abstrakte Klasse K1 mit abstrakter Methode f() abgeleitet. Was bedeutet es, wenn in K2 f() realisiert / nicht realisiert wird?
- Wann ist es sinnvoll, dass eine konkrete Klasse K1 eine abgeleitete abstrakte Klasse K2 hat?
- Wozu dienen Schnittstellen?
- Was würde man in der Modellierung in eine Schnittstelle stecken?
- Wieso reichen abstrakte Klassen nicht, sondern es gibt zusätzlich Schnittstellen?
- Wie wird eine Schnittstelle in Java angegeben?
- Welche Zugriffsregeln gibt es zu Schnittstellen und wie wirken sie?
- Wie nutzt man Schnittstellen in Java?
- Wozu nutzt man Vererbung bei Schnittstellen?
- Ist das möglich? Wenn ja, macht das Sinn? `interface S1 { int f(); } interface S2 extends S1 { int f(); }`
- Ist das möglich? Wenn ja, macht das Sinn? `interface S1 { int f(); } interface S2 extends S1 { int g(); }`
- Darf man ein Objekt einer Klasse K auf den Schnittstellentyp S casten, wenn K S realisiert?
- Wozu nutzt man innere/lokale/anonyme Klassen?
- Wie gibt man innere/lokale/anonyme Klassen an?
- Welche Einschränkungen gibt es bei Inneren Klassen?
- Welche Einschränkungen gibt es bei Lokalen Klassen?

## Reflektion des Stoffs

- Sollte man grundsätzlich nur Instanz- oder nur Klassenmethoden finalisieren oder beides?
- Macht das Finalisieren einer Methode Sinn, wenn es nur eine statische Methodenbindung zu der Methode gibt? Nur dynamische?
- Kann man eine Klasse gleichzeitig als `abstract` und `final` deklarieren? Wenn ja, was ist dann die Bedeutung?
- Kann man eine Schnittstelle durch eine abstrakte Klasse mit ausschließlich abstrakten Methoden ersetzen? Immer? Nie? Wenn nur manchmal, wann?
- Was sind die Unterschiede / Vorteile von Schnittstellen gegenüber abstrakten Klassen?
- Erarbeiten Sie sich drei eigene Beispiele, wo jeweils einer der drei Typen von geschachtelten Klassen Sinn jeweils macht.
- Nehmen Sie das Beispiel der Obsthandlung aus Kapitel 11 als Grundlage. Hier nochmals wiederholt:  
  
 Modellieren Sie eine Obsthandlung, die Bananen und Äpfel verkauft. Bananen haben einen Krümmungsradius, ein Gewicht und einen Preis, der vom Gewicht abhängig ist. Der Kilopreis für Bananen ist 2 Euro. Äpfel haben einen Durchmesser, eine Farbe, ein Gewicht und einen gewichtsabhängigen Preis. Der Kilopreis für Äpfel ist 3 Euro. In der Obsthandlung liegen 3 Bananen mit unterschiedlicher Krümmung und Gewicht (suchen Sie sich Werte dafür aus) und 4 Äpfel aus, auch mit unterschiedlichem Durchmesser und Gewicht. Ermitteln Sie den Verkaufspreis aller Bananen und Äpfel in der Obsthandlung.

Modellieren Sie darauf aufbauend, dass es verschiedene Sorten von Äpfeln geben kann. Die Sorte Granny Smith ist immer grün und kostet pro Kilo 1,99 Euro. Die Sorte Elstar ist rötlich und kostet 1,79 Euro das Kilo. Nur Grany Smith Objekte und Elstar Objekt sollen noich erlaubt sein, nicht mehr allgemeine Apfelobjekte.

- In Kapitel 13.3 wurde das Beispiel zur Schnittstelle `Impfen` gezeigt. Man hätte die Alternativen, dass 1) die abstrakte Tierklasse angibt die Schnittstelle zu realisieren oder aber 2) die konkreten Tierklassen `Ameise` etc. selbst. Diskutieren Sie die beiden Fälle! Welcher davon ist konzeptionell besser in diesem Beispiel und wieso?



# Chapter 14

## Fehlerbehandlung

Während der Programmausführung kann es zu (Ausnahme-)Situationen kommen, sogenannten *Exceptions*. Zum Beispiel aufgrund der Seltenheit des Auftretens oder weil ihre Behandlung an dieser Programmstelle die Programmlogik zerteilen würde, will man die Behandlung solcher Ausnahmen oft außerhalb des eigentlichen Programmablaufs behandeln, selbst garnicht an dieser Programmstelle behandeln oder es gar dem System selbst überlassen. Allgemeine Beispiele für solche Ausnahmen sind etwa der Überlauf des Wertebereichs bei arithmetischen Operationen, der Zugriff über einen Referenzausdruck, dessen Wert `null` ist, fehlender Heap-Speicherplatz bei Aufruf von `new`, die Verletzung von Feldgrenzen, Probleme beim Lesen von einer Datei oder der Abbruch einer Netzverbindung. Diese Fehler werden in jedem Programm (hoffentlich) nur selten auftreten, also eine Ausnahme sein, und sollten demzufolge auch außerhalb des normalen Programmablaufs als Spezialfall gesondert behandelt werden können, um den eigentlichen Programmcode nicht unnötig aufzublähen oder kompliziert zu machen. Fehler, die ein Compiler schon zur Übersetzungszeit zuverlässig erkennen kann (etwa die Nutzung von nicht initialisierten blocklokalen Variablen), sind in dieser Betrachtung außen vor. Es geht hier also ausschließlich um Fehler, die erst zur Laufzeit erkannt werden können.

Dabei können solche Ausnahmefehler auftreten, weil das System (Hardware, Betriebssystem, Java Laufzeitsystem) einen Fehler erkannt hat oder aber das Programm selbst hat eine solche unnormale Fehlersituation erkannt, deren Behandlung an dieser Programmstelle aber beispielsweise weniger Sinn machen würde. Man muss also das Auftreten bzw. Melden eines Fehlers unterscheiden von der Behandlung eines solchen Fehlers, die durchaus woanders im Programm stattfinden kann. Und auch die Fehlerart kann von Interesse sein. Man spricht in diesem Zusammenhang auch davon, eine **Exception werfen** und eine **Exception fangen**.

### Beispiel 14.1:

Listing 14.1 zeigt ein Beispielprogramm, in dem zwei ganzzahlige Werte aus der Kommandozeile gelesen werden und das Resultat der Division dieser Werte auf dem Bildschirm ausgegeben wird.

1. Was passiert, wenn statt der spezifizierten zwei Argumenten dem Programm nur ein oder kein Argument übergeben wird?
2. Was passiert, wenn eines der beiden Argumente nicht die zulässige Darstellung eines ganzzahligen Wertes im erlaubten Wertebereich ist?
3. Was ist, wenn das zweite Argument 0 ist (Division durch 0)?

In diesem kleinen Programm mit drei Anweisungen müsste man erheblichen Zusatzaufwand treiben, um alle mögliche Spezialfälle zu behandeln. Dieser Zusatzaufwand würde dann den überwiegenden Teil des

Listing 14.1: Beispielprogramm mit möglichen (Fehler-)Ausnahmen.

```

14-1  /**
14-2   * Beispiel fuer Ausnahmen im Programmablauf
14-3   */
14-4 public class AusnahmeFall {
14-5     public static void main(String [] args) {
14-6         // was passiert, wenn weniger als 2 Argumente uebergeben werden?
14-7         // was passiert, wenn args[0]/args[1] keine Darstellung eines int enthaelt?
14-8         int i = Integer.parseInt(args[0]);
14-9         int j = Integer.parseInt(args[1]);
14-10
14-11         // Was ist, wenn j==0 ist?
14-12         System.out.println(i / j);
14-13     }
14-14 }
```

Gesamtprogramms ausmachen. Und die eigentlich Logik dessen, was man tun will, würde in den zahlreichen Selektionen untergehen. Hier sind die Ausgaben, wenn man dieses Programm mit problematischen in der Kommandozeile startet:

1. Ohne Argumente: `java AusnahmeFall`

```

14-1 Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
14-2     at AusnahmeFall.main(AusnahmeFall.java:7)
```

2. Unzulässiger ganzzahliger Wert: `java AusnahmeFall true 1`

```

14-1 Exception in thread "main" java.lang.NumberFormatException: For input string: "true"
14-2     at java.lang.NumberFormatException.forInputString(NumberFormatException.java
14-3         :65)
14-4     at java.lang.Integer.parseInt(Integer.java:492)
14-5     at java.lang.Integer.parseInt(Integer.java:527)
14-6     at AusnahmeFall.main(AusnahmeFall.java:7)
```

3. Zweites Argument ist 0: `java AusnahmeFall 1 0`

```

14-1 Exception in thread "main" java.lang.ArithmaticException: / by zero
14-2     at AusnahmeFall.main(AusnahmeFall.java:11)
```

In allen Fällen wird also anscheinend auch ohne eine explizite Behandlung im eigenen Programm der Fehler erkannt und auf eine bestimmte generische Weise behandelt (Ausgabe einer Fehlermeldung und Programmabbruch). ♦

### Beispiel 14.2:

Listing 14.2 zeigt zwei Methoden, die beide gleiches machen sollen. In `methode1` ist der Code ohne Fehlerbehandlung zu sehen (so, wie man es eigentlich i.W. haben will). `methode2` zeigt, welche Fehler abgefangen werden müssten und wie sehr der Programmcode dadurch aufgeblättert würde und vom eigentlichen Code entfernt ist. ♦

Insgesamt sind also mehrere Aspekte im Zusammenhang mit Ausnahmesituationen zu berücksichtigen:

Listing 14.2: Beispielprogramm, wenn man selbst alle Fehler abfangen müsste.

```

14-1 /**
14-2 * (schlechtes) Programm mit Fehlerüberprüfungen
14-3 */
14-4 public class OhneExceptions {
14-5
14-6     double[] methode1(int i, double value) {
14-7         double[] a = new double[10];
14-8         a[i] = 1.0 / value;
14-9         return a;
14-10    }
14-11
14-12    double[] methode2(int i, double value) {
14-13        if (System.verfügbarerSpeicher() > 10*8) {
14-14            double[] a = new double[10];
14-15            if (a != null) {
14-16                if ((i >= 0) && (i < 10)) {
14-17                    if (value != 0.0) {
14-18                        a[i] = 1.0 / value;
14-19                        return a;
14-20                    } else {
14-21                        System.out.println("Division durch 0");
14-22                        return null;
14-23                    }
14-24                } else {
14-25                    System.out.println("illegaler Index");
14-26                    return null;
14-27                }
14-28            } else {
14-29                System.out.println("Referenz ist null");
14-30                return null;
14-31            }
14-32        } else {
14-33            System.out.println("kein Speicherplatz mehr vorhanden");
14-34            return null;
14-35        }
14-36    }
14-37 }

```

Listing 14.3: Beispiel für eine Fehlerbehandlung als Teil der Programmlogik.

```

14-1 public static void main( String [] args ) {
14-2     ...
14-3     int monat;
14-4     boolean fehlerFall;
14-5
14-6     do {
14-7         System.out.println("Geben Sie eine gueltige Monatsangabe ein.");
14-8         monat = sc.nextInt();
14-9
14-10        // gueltiger Wert?
14-11        fehlerFall = (monat < 1) || (monat > 12);
14-12        if (fehlerFall){
14-13            System.out.println("Fehler in Monatsangabe");
14-14        }
14-15    } while(fehlerfall);
14-16    ...
14-17 }
```

- Das Auftreten einer Ausnahme muss zur Laufzeit kenntlich gemacht werden können (implizit durch das System wie im Beispiel oben oder explizit durch das Programm).
- Ausnahmen müssen behandelt werden können, und dies muss auch an anderen Stellen im Programm möglich sein als an der Stelle, an der der Fehler aufgetreten ist.
- Der Code für den normalen Programmablauf sollte möglichst wenig gestört werden durch die Behandlung von solchen Ausnahmen.

In Listing 14.2 sind Fehler angegeben, die wenn sie auftreten entweder auf beschränkte Ressourcen (kein Speicher mehr vorhanden) oder schlicht auf eine fehlerhafte Programmierung zurückzuführen sind ( $i$  wäre ein Index außerhalb des erlaubten Bereichs). Beim, letzteren Fehlerfall müsste eigentlich ein Programm *von der Programmlogik her* selber gewährleisten, dass dieser Fehler nicht auftreten kann. Ebenso bei der Division durch Null sollte die Programmlogik gewährleisten, dass der Dividend nicht Null ist. Insofern sollte das Auftreten dieser konkreten Fehler in dem Beispiel die absolute Ausnahme sein, und teilweise zudem auch bei Auftreten als Programmierfehler angesehen werden.

### Beispiel 14.3:

Eine Fehlerüberprüfung von logischen Annahmen sollte Teil der Programmlogik sein und nicht eine Ausnahmesituation. Listing 14.3 zeigt ein Beispiel dazu, wo eine Eingabe von der Tastatur vorgenommen wird. Ist der eingegebene Wert fehlerhaft (es ist eine Monatsangabe zwischen 1 und 12 gefordert), so wird eine Fehlermeldung ausgegeben und eine erneute Eingabe eingefordert. ♦

In den nachfolgenden Unterkapiteln werden die Möglichkeiten gezeigt zur Definition von Ausnahmeklassen, das Auslösen von Ausnahmen und die Behandlung von / Reaktion auf aufgetretene Fehler im Programm.

## 14.1 Klassen von Ausnahmefällen definieren

In Java werden Ausnahmen allgemein beschrieben in Fehlerklassen, die in einem Klassenbaum organisiert sind mit der Klasse `Throwable` aus dem Paket `java.lang` als Basisklasse dieser Hierarchie. Jede solche

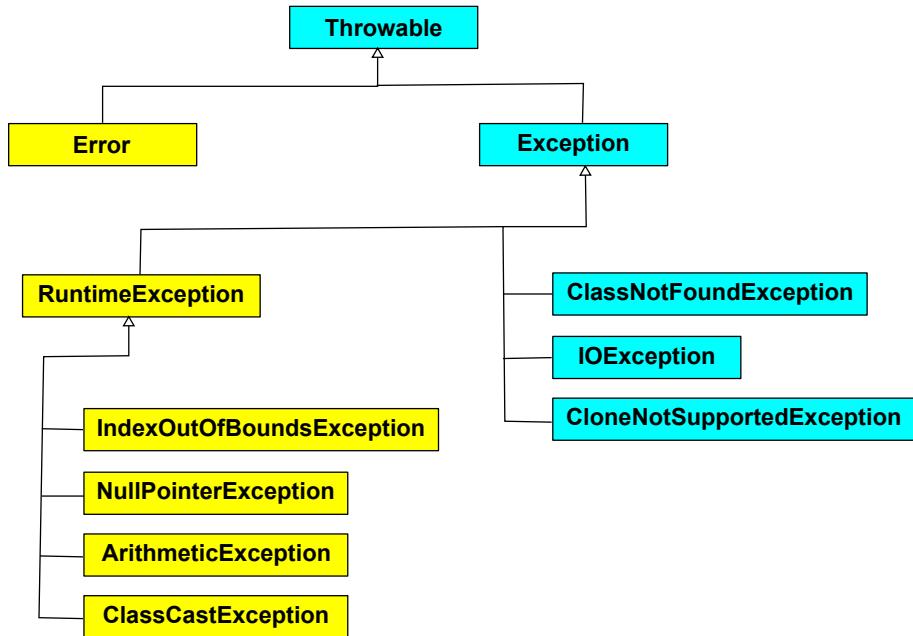


Figure 14.1: Ausschnitt der Klassenhierarchie zu Java Exceptions (gelb *unchecked*, blau *checked*).

Klasse in dieser Hierarchie steht für eine bestimmte Art von Fehlern. Entsprechend der Logik der Vererbung stehen Klassen der oberen Schichten dieser Hierarchie jeweils für eine mehr allgemeinere Fehlerbeschreibung (zum Beispiel `RuntimeException`), Klassen weiter unten in der Hierarchie für speziellere Fehler (zum Beispiel `NullPointerException` als eine spezielle Ausprägung von `RuntimeException`). Abbildung 14.1 zeigt einen Ausschnitt aus diesem Klassenbaum, wie er in Java bereits definiert ist. An der Wurzel des Baums ist also die allgemeinste Fehlerart, je tiefer man in der Ableitungshierarchie kommt, umso spezieller ist der Fehler.

Man unterscheidet dabei grob zwischen zwei Ausnahmearten, die in Java unterschiedlich behandelt werden können/müssen. Bei *checked exceptions* wird vom Compiler (also bereits zur Übersetzungszeit des Programms) überprüft, ob an den Stellen, an denen diese auftreten können, sie auch im Programmcode behandelt werden, ob es also auf jeden Fall eine Stelle im Programm gibt, die eine an dieser Programmstelle erzeugten Fehler dieser Art behandelt (siehe nachfolgendes Kapitel 14.3 dazu). Kann ein Java-Compiler nicht erkennen, dass zu einem möglichen Ausnahmefall auch eine Behandlung geschieht, ist dies ein Übersetzungsfehler. Bei *unchecked exceptions* findet keine solche Überprüfung durch den Compiler statt. Die Idee hinter dieser Zweiteilung ist, dass man mit *checked exceptions* einerseits möglichst viel Programmsicherheit garantieren will. Andererseits können Fehler der Art *unchecked exceptions* an sehr vielen Stellen in Programmen auftreten, wo dann *überall* entsprechender Programmcode vorhanden sein *müsste*, der diese dann auch behandelt (zum Beispiel bei jeder Division, Dereferenzierung einer Referenz, Feldzugriff). Da letztere Art von Fehlern aber meist auf Programmierfehler zurückzuführen sind (und demzufolge keine wirklichen Ausnahmen in der Programmausführung wären), ist hier ein Kompromiss aus Sicherheit und Programmierbarkeit getroffen worden. Behandelt ein Programm nicht selbst *unchecked exceptions*, so tut dies das Laufzeitsystem (siehe Ausgabeprotokoll zu Listing 14.1).

Es besteht auch die Möglichkeit, auf dieser vordefinierten Klassenhierarchie aufbauend eigene Fehlerarten/klassen zu definieren. Dazu muss man eine eigene Klasse von einer bereits existierenden Fehlerklasse ableiten. Je nach Basisklasse kann man damit also auch *checked* oder *unchecked exceptions* programmieren. Da eine Fehlerinstanz (also eine Instanz einer solchen Klasse) bereits über die Klasse selbst den Fehler spezifiziert, sind keine weiteren Methoden oder Attribute in einer solchen Klasse vorgesehen. Der parameterlose Konstruktor dieser eigenen Klasse sollte den Konstruktur der Oberklasse mit einem String aufrufen, der aus

Listing 14.4: Definition einer eigenen Fehlerklasse.

```

14-1  /**
14-2   * Definition einer eigenen Fehlerklasse (checked exception)
14-3   */
14-4  class MeineException extends Exception {
14-5
14-6      MeineException() {
14-7          // Aufruf des Konstruktors von Exception
14-8          super("MeineException wurde geworfen");
14-9      }
14-10
14-11     // falls diese Klasse selber wieder abgeleitet werden soll
14-12     MeineException(String message) {
14-13         // Aufruf des Konstruktors von Exception
14-14         super(message);
14-15     }
14-16 }
```

einer aussagekräftigen Fehlernachricht bestehen sollte. In der Oberklasse `Throwable` gibt es eine Methode `getMessage`, mit der man zu einem Fehlerobjekt diesen Text ermitteln kann. Will man die eigene Klasse als Basisklasse für weitere Fehlerklassen weiterverwenden, so muss man ebenfalls einen Konstruktor mit einem String als Argument angeben, der diesen String an die eigene Oberklasse weiterreicht.

#### Beispiel 14.4:

Listing 14.4 zeigt ein Beispiel für eine eigene Fehlerklasse, die von `Exception` abgeleitet ist. Der parameterlose Konstruktor ruft den Konstruktor der Oberklasse mit einem Fehlertext auf. Der zweite Konstruktor, der nur für den Fall der Ableitung dieser Klasse sinnvoll ist, gibt eine übergebene Fehlernachricht in der Klassenhierarchie weiter nach oben. ♦

## 14.2 Ausnahmen auslösen

Während diese Ausnahmeklassen jeweils Fehler gleicher Art beschreiben (Klassenbildung), wird *das Auftreten einer Ausnahme zur Laufzeit eines Programms* durch eine Instanz einer solchen Klasse repräsentiert. Jedes Auftreten eines Ausnahmeeignisses zur Laufzeit eines Programms wird dann durch ein entsprechendes Java-Objekt repräsentiert. Tritt ein Fehler mehrfach auf, so werden auch unterschiedliche Java-Objekte zu dem jeweiligen Auftreten des Fehlers generiert.

Ausnahmen können einerseits von der Hardware, vom Betriebssystem oder vom Java-Laufzeitsystem erkannt und eine Exception geworfen werden. Zu solchen Exceptions gehört zum Beispiel die Division durch Null (erkennst die Hardware) oder wenn kein Speicher mehr frei bei der Anwendung des `new`-Operators (Java-Laufzeitsystem). In einem solchen Fall würde letztendlich durch das Java-Laufzeitsystem eine zu diesem Fehlerfall passende Exception geworfen.

Weiterhin kann aber auch ein Programm selbst das Auftreten einer Ausnahmesituation (zum Beispiel zu einer eigenen Fehlerklasse) erkennen und dann auch signalisieren. Die Bekanntgabe eines solchen Fehlers geschieht, indem ein Objekt einer entsprechenden Fehlerklasse instanziert wird und weiterhin mit diesem Fehlerobjekt dieser Fehler dem Java-Laufzeitsystem über eine entsprechenden `throw`-Anweisung bekannt gemacht wird. Man spricht davon, dass eine Exception geworfen wird.

Das Java Laufzeitsystem wird dann (wie in den anderen impliziten Fehlerfällen auch) nach einem

Listing 14.5: Werfen und Fangen einer eigenen Exception.

```

14-1  /**
14-2   * eine eigene Exception Klasse werfen und fangen
14-3   */
14-4  class MeineExceptionFangen {
14-5
14-6      public static void main(String [] args) {
14-7
14-8          try {
14-9              // Exception -Objekt erzeugen
14-10             MeineException e = new MeineException();
14-11
14-12             // Exception werfen
14-13             throw e;
14-14             // hier hin kommen wir nicht in der Ausfuehrung
14-15             System.out.println("hier kommen wir nie hin");
14-16         } catch(MeineException e) {
14-17             // Fehlermeldung ausgeben
14-18             System.out.println("Exception geworfen: " + e.getMessage());
14-19             // Aufrufhierarchie von Methoden als Hinweis ausgeben
14-20             e.printStackTrace();
14-21         }
14-22     }
14-23 }
```

entsprechenden *Exception Handler* suchen (siehe Kapitel 14.3) und falls gefunden diesen aktivieren. Die Programmausführung wird damit an der Stelle des Auftretens des Fehlers abgebrochen und an einer anderen Stelle im Programm weitergeführt, im *Exception Handler*. Nach der Ausführung des *Exception Handlers* wird die Ausführung des Programms mit dem Code nach dem *Exception Handler* weiter geführt, und *nicht* an der Fehlerstelle. Findet das Java Laufzeitsystem keinen *Exception Handler* zur Behandlung dieser Fehlerklasse, bricht das Programm ab.

### Beispiel 14.5:

Unter Bezug auf Listing 14.4 zeigt Listing 14.5, wie man eine Exception instanziieren und werfen kann. Weiterhin ist auch die Behandlung dieses Fehlers dort gezeigt. Als Ausgabe zu dem gezeigten Programm würde auf dem Bildschirm erscheinen:

```

14-1 MeineException: MeineException wurde geworfen
14-2      at MeineExceptionFangen.main(MeineExceptionFangen.java:10)
```

## 14.3 Aufgetretene Ausnahmen behandeln

Kann während der Programmausführung in Teilen eines Programms eine Ausnahme auftreten (eine Exception wird geworfen), so kann (bei *unchecked exceptions*) bzw. muss (bei *checked exceptions*) man Programmcode angeben, der ein solches Auftreten behandelt (Exception fangen). Solche Programmstellen *zur Behandlung einer Ausnahme* heißen *Exception Handler*. Diese werden in einer *try-catch*-Anweisung im Programm angegeben.

Listing 14.6: Prinzipieller Aufbau einer `try`-Anweisung.

```

14-1   try {
14-2       ... // normaler Code
14-3   } catch (ExceptionKlasse1 name) {
14-4       ... // Behandlung eines Fehlers zur ExceptionKlasse1
14-5   } catch (ExceptionKlasse2 name) {
14-6       ... // weitere catch-Blocke
14-7   }
14-8   } finally {
14-9       ... // Code, der auf jeden Fall ausgefuehrt wird
14-10  }
```

Der prinzipielle Aufbau einer `try-catch`-Anweisung folgt dem Schema, wie es in Listing 14.6 zu sehen ist. Dabei steht `ExceptionKlasseX` für eine beliebige Klasse aus der Hierarchie der Fehlerklassen (inklusive eigener Fehlerklassen) und `name` ist ein beliebiger Variablenname zur Aufnahme des Ausnahmektos (zusammen ist dies ähnlich einer Parameterspezifikation in einer Methode).

Eine `try-catch`-Anweisung beginnt mit einem `try`-Block. Dieser Teil besteht aus dem Schlüsselwort `try` gefolgt von einem Block mit (normalem) Code, in dem eventuell eine Ausnahme auftreten könnte, die gefangen werden soll. Nach diesem `try`-Block können eine oder mehrere `catch`-Blöcke folgen und abschließend ein `finally`-Block. Ist ein `finally`-Block vorhanden, können `catch`-Blöcke entfallen. Ist mindestens ein `catch`-Block vorhanden, so kann der `finally`-Block entfallen. Jeder `catch`-Block stellt dabei einen *Exception Handler* zu einer oder mehreren Ausnahmeklassen dar.

Die Semantik einer solchen `try-catch`-Anweisung ist nun wie folgt. Zuerst wird der Code im `try`-Block ausgeführt. Tritt dabei keine Ausnahme auf, so wird keiner der `catch`-Fälle überprüft und nach der `try-catch`-Anweisung mit dem Programmcode fortgefahren (aber siehe dazu auch die Diskussion zu `finally` gleich).

Tritt jedoch eine Ausnahme im `try`-Block auf, so wird die Programmausführung abgebrochen und es werden *nacheinander in der angegebenen Reihenfolge* die `catch`-Angaben überprüft. Dabei wird jeweils überprüft, ob das im `try`-Block geworfene Ausnahmektos eine Instanz der in der jeweiligen `catch`-Überprüfung angegebenen Exception-Klasse ist (inklusive Instanz einer davon abgeleiteten Klasse). Ist dies bei einer `catch`-Überprüfung der Fall, so wird der Code in dem dazu gehörigen Block ausgeführt und weitere `catch`-Überprüfungen finden nicht mehr statt.

Ist ein `finally`-Block vorhanden, so wird dieser *auf jeden Fall* zuletzt ausgeführt, egal, ob eine Ausnahme auftrat oder nicht, und egal, ob ein `catch`-Block vorher ausgeführt wurde oder nicht! Ein Beispiel für eine sinnvolle Nutzung wäre also ein Aktion, die in jedem Fall getan werden muss (siehe nachfolgendes Beispiel). Auch wenn die `try-catch`-Anweisung ähnlich einer `switch`-Anweisung aussieht, so ist die Bedeutung des `finally` in dieser Anweisung nicht mit der Bedeutung des `default` in einer `switch`-Anweisung zu vergleichen (`default` wird *nur* betrachtet, wenn kein Fall vorher genommen wurde).

Trifft keiner der Fälle eines `catch`-Blocks zu und ist kein `finally` vorhanden, so wird die Ausnahme (an dieser Stelle) nicht behandelt.

Wir eine Exception irgendwo von einem Exception-Handler behandelt, so spicht man auch davon, dass diese Exception dort gefangen wird (als Gegenstück zum Werfen einer Exception).

### Beispiel 14.6:

Listing 14.7 zeigt die sinnvolle Verwendung von `finally`. Egal, ob im `try`-Block ein Fehler aufgetreten ist oder nicht, muss der Scanner geschlossen werden. Zu beachten ist auch, dass je nachdem wo ein Fehler auftrat, die Variable `sc` eine Referenz auf ein Scanner-Objekt erhält oder nicht. ♦

Listing 14.7: Beispiel für den Einsatz von `finally`.

```
14-1  /**
14-2   * Beispiel zum Einsatz von finally
14-3   */
14-4  import java.util.Scanner;
14-5  import java.io.*;
14-6
14-7  public class FinallyBeispiel {
14-8
14-9    public static void main(String[] args) {
14-10      Scanner sc = null;
14-11      try {
14-12          sc = new Scanner(new File(args[0]));
14-13          int summe = 0;
14-14          for(int i=0; i<10; i++) {
14-15              summe += sc.nextInt();
14-16          }
14-17          System.out.println("summe=" + summe);
14-18      } catch(FileNotFoundException e) {
14-19          System.out.println("Datei nicht gefunden");
14-20      } catch(NumberFormatException e) {
14-21          System.out.println("Zahl in Datei falsch angegeben");
14-22      } catch(Exception e) {
14-23          System.out.println("anderer Fehler");
14-24      } finally {
14-25          // der Scanner muss auf jeden Fall geschlossen werden
14-26          if(sc != null) {
14-27              sc.close();
14-28          }
14-29      }
14-30  }
```

Listing 14.8: Beispiel zu einer Multi-catch-Anweisung.

```

14-1   try {
14-2     ... // normaler Code
14-3   } catch (ExceptionKlasse1 | ExceptionKlasse3 | ExceptionKlasse3 name) {
14-4     ... // Behandlung eines Fehlers zur ExceptionKlasse1, 2 oder 3
14-5   } finally {
14-6     ... // Code, der auf jeden Fall ausgefuehrt wird
14-7 }
```

Wie besprochen, werden die `catch`-Fälle der Reihe nach getestet. Der Diskriminatator ist dabei die Fehlerart, konkret die Exception-Klasse. Da alle Exception-Klassen in einem Ableitungsbaum organisiert sind, ist es sehr wichtig, in welcher Reihenfolge man diese `catch`-Blöcke angibt, oder anders ausgedrückt, wie umfänglich dieses Tests jeweils angegeben werden. Exception-Klasse, die in der Ableitungshierarchie höher angesiedelt sind, werden also mehr Fehler anfangen können als Exception-Klassen, die sehr speziell sind (tief unten in dieser Hierarchie). Es macht demzufolge keinen Sinn, eine allgemeinere Exception in einem frühen `catch`-Block anzugeben und gleichzeitig in einem späteren `catch`-Block eine davon abgeleitete Exception (dies wäre sogar ein semantischer Fehler, der vom Compiler erkannt wird).

Aufgrund der eben besprochenen Semantik macht es also Sinn, in den `catch`-Anweisungen, die zu Beginn stehen, eher speziellere Ausnahmen abfragen (in der Klassenhierarchie tiefer unten angesiedelt) und in späteren `catch`-Anweisungen allgemeiner werden, womit mehr Fehlerklassen behandelt werden können (siehe dazu Beispiel 14.10).

Beginnend mit Java 7 gibt es sogenannte *multi-catch*-Anweisungen. Dies sind `catch`-Anweisungen, in denen mehr als eine Exception-Klasse angegeben werden kann. Dies muss man dann als eine Art Oder-Verknüpfung lesen: wenn das Exception-Objekt auf eine der Exception-Klassen passt, so wird der Block zu diesem `catch` ausgeführt in der obigen Semantik ausgeführt. Ein prinzipielles Beispiel dazu ist in Listing 14.8 angegeben.

Weiterhin wurde mit Java 7 das try-with-resources Konzept eingeführt, worauf hier nicht weiter eingegangen wird.

#### **Beispiel 14.7:**

Listing 14.9 zeigt das Beispielprogramm aus Listing 14.1 mit zusätzlicher Fehlerbehandlung. Zu beachten ist, dass die eigentliche Programmlogik nach wie vor als ein zusammenhängender Programmteil vorhanden ist, also nicht mit einer Vielzahl von möglichen Fehlerabfragen verändert wurde. Alle möglichen Ausnahmen in diesem Programmteil werden in einem `catch`-Block im Abschluss an den eigentlichen Code generisch behandelt, indem die Klasse `Exception` als Oberklasse aller Ausnahmen gewählt wurde und so jedes Ausnahmenobjekt darunter fallen muss.

Listing 14.10 zeigt das gleiche Programm, wo aber eine Fehlerdifferenzierung stattfindet, um anhand der Ausnahmeobjekts die Fehlerursache zu ermitteln und entsprechend handeln zu können. Man beachte den letzten `catch`-Block, wo mit der Klasse `Exception` wiederum die Basisklasse aller Ausnahmen gewählt wurde und so alle bis jetzt nicht zutreffenden Fehlerarten behandelt werden. ♦

## 14.4 Propagieren von Ausnahmen

Soll in einer Methode die Fehlerbehandlung selbst nicht stattfinden, so gibt es die Möglichkeit, dass eine Ausnahme in der Aufrufhierarchie weiter nach oben propagierte wird. Bei *unchecked exceptions* ist dazu nichts weiter im Programm anzugeben. Kann jedoch eine *checked exception* auftreten, die nicht lokal behandelt

Listing 14.9: Beispiel für eine generische Ausnahmebehandlung.

```

14-1 /**
14-2 * Beispiel fuer Ausnahmen im Programmablauf mit generischer Behandlung
14-3 */
14-4 public class AusnahmeFall2 {
14-5     public static void main(String[] args) {
14-6         try {
14-7
14-8             // Was passiert, wenn weniger als 2 Argumente uebergeben werden?
14-9             // Was passiert, wenn args[0]/args[1] keine Darstellung eines int enthaelt?
14-10            int i = Integer.parseInt(args[0]);
14-11            int j = Integer.parseInt(args[1]);
14-12
14-13            // Was ist, wenn j==0 ist?
14-14            System.out.println(i / j);
14-15
14-16        } catch(Exception e) {
14-17            System.out.println("Da ist was falsch gelaufen");
14-18        }
14-19    }
14-20}

```

Listing 14.10: Beispiel für eine individuelle Ausnahmebehandlung.

```

14-1 /**
14-2 * Beispiel fuer Ausnahmen im Programmablauf mit individueller Behandlung
14-3 */
14-4 public class AusnahmeFall3 {
14-5     public static void main(String[] args) {
14-6         try {
14-7
14-8             // Was passiert, wenn weniger als 2 Argumente uebergeben werden?
14-9             // Was passiert, wenn args[0]/args[1] keine Darstellung eines int enthaelt?
14-10            int i = Integer.parseInt(args[0]);
14-11            int j = Integer.parseInt(args[1]);
14-12
14-13            // Was ist, wenn j==0 ist?
14-14            System.out.println(i / j);
14-15
14-16        } catch(ArrayIndexOutOfBoundsException e) {
14-17            System.out.println("Nicht genug Werte uebergeben");
14-18        } catch(NumberFormatException e) {
14-19            System.out.println("Keine ganzzahligen Werte");
14-20        } catch(ArithmetricException e) {
14-21            System.out.println("Fehler in der Arithmetik");
14-22        } catch(Exception e) { // Oberklasse faengt alle Fehler
14-23            System.out.println("irgendein sonstiger Fehler");
14-24        }
14-25    }
14-26}

```

Listing 14.11: Beispiel zur Propagation von Exceptions.

```

14-1 /**
14-2 * Beispiel fuer Ausnahmen im Programmablauf mit Propagierung
14-3 */
14-4 import java.util.*;
14-5 import java.io.*;

14-6
14-7 public class AusnahmeFall14 {
14-8     static Scanner sc = null;
14-9
14-10    public static void main(String[] args) {
14-11        // Beispiel 1: Exception wird in Methode selbst gefangen
14-12        initializeScanner1(args[0]);
14-13
14-14        // Beispiel 2: propagierte Exception fangen
14-15        try {
14-16            initializeScanner2(args[0]);
14-17
14-18            } catch(FileNotFoundException e) {
14-19                System.out.println("Datei nicht gefunden");
14-20            }
14-21        }
14-22
14-23        static void initializeScanner1(String filename) {
14-24            // Scanner von der Datei "test.txt" anlegen
14-25            try {
14-26                sc = new Scanner(new File(filename));
14-27            } catch(FileNotFoundException e) {
14-28                System.out.println("Datei nicht vorhanden");
14-29                return;
14-30            }
14-31        }
14-32
14-33        static void initializeScanner2(String filename) throws FileNotFoundException {
14-34            // Scanner von der Datei "test.txt" anlegen
14-35            sc = new Scanner(new File(filename));
14-36        }
14-37    }

```

wird, so *muss* im Methodenkopf der Methode dann angegeben werden, welche Ausnahmen diese Methode alle werfen kann (die sie selbst nicht behandelt). Dies geschieht, indem an den Methodenkopf das Schlüsselwort **throws** angehängt wird, gefolgt von einer durch Kommas getrennten Liste der möglichen Ausnahmen.

Wird dann eine Ausnahme in der Ausführung dieser Methode geworfen (inklusive in Methoden, die in der Methode aufgerufen wurden), so findet die Behandlung nicht in der Methode statt und in der Aufrufhierarchie der Methodenaufrufe wird nach einer passenden **try**-Anweisung gesucht. Bei *checked exceptions* *muss* also ein **throws** im Methodenkopf angegeben werden, bei *unchecked exceptions* *kann* **throws** angegeben werden.

### Beispiel 14.8:

Listing 14.11 zeigt zwei Beispiele für Methodenaufrufe in **main**. In beiden Methoden kann es während der Ausführung zu einer *checked exception* kommen (**FileNotFoundException**) falls eine Datei nicht geöffnet werden kann). In der Methode **initializeScanner1** wird die Exception mit einer **try-catch**-Anweisung lokal behandelt, indem der problematische Methodenaufruf **new File(filename)** in einem **try**-Block steht und ein

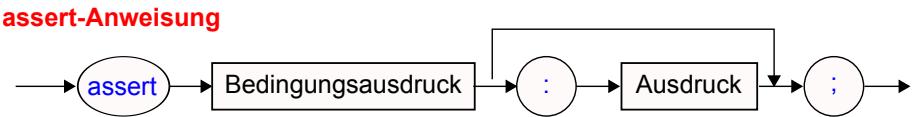


Figure 14.2: Syntax der assert-Anweisung.

zugehöriger `catch`-Block die potentielle Exception behandeln kann.

Im der Methode `initializeScanner2` wird ebenfalls eine Datei geöffnet. Allerdings wird die potentielle checked exception `FileNotFoundException` hier nicht lokal behandelt. Deshalb muss diese Methode im Methodekopf eine `throws`-Erweiterung haben. Die Behandlung dieser Exception wird stattdessen an der Stelle vorgenommen, an der der Methodenaufruf sattfindet. ♦

## 14.5 Assertions

Exceptions dienen der Erkennung und Behandlung von Laufzeitfehlern und sind damit Bestandteil jedes halbwegs komplexen Programms. Einen gänzlich anderen Zweck erfüllen sogenannte **Assertions**, die über eine `assert`-Anweisung in Java formuliert werden. Diese Assertions dienen während der Programmierung (und nur dann) dazu, möglichst fehlerfreien Code zu entwickeln. Eine Assertion (Zusage, Behauptung) ist eine logische Bedingung, die ein Programmierer formuliert und die an einer dafür vorgesehenen Stelle im Programm *immer* gelten muss. Diese Bedingung wird während Tests bei der Programmierung jeweils an der entsprechenden Programmstelle zur Laufzeit des Programms überprüft. Ist diese dort formulierte Annahme nicht wahr, die Annahme stimmt also nicht, so tritt ein Laufzeitfehler auf mit einer Ausgabe, die auf den Fehler und die Programmstelle hinweist. Mit dieser Information kann ein Programmierer dann eine Fehleranalyse betreiben und den Fehler beheben.

Solche Assertions sollten strikt von einer normalen Fehlerbehandlung getrennt werden, die jedes Programm haben sollte. Assertions dienen dazu, Fehler, die eigentlich aufgrund der Programmlogik niemals auftreten sollten, aber doch (während der Entwicklung) auftreten, dann zu erkennen. Dies ist zum Beispiel dann möglich, wenn die Programmlogik anders als gedacht fehlerhaft ist, etwa einen Spezialfall nicht berücksichtigt hat. Wird eine Assertion als falsch erkannt, so macht eine wie auch immer gartete Fehlerbehandlung im Programm selbst auch keinen Sinn, das Programm ist dann einfach falsch und muss umprogrammiert werden.

Die Syntax für eine assert-Anweisung ist in Abbildung 14.2 angegeben. Die Bedeutung dieses Konstrukts ist, dass zuerst der Bedingungsausdruck ausgewertet wird. Ist der Wert dieses Ausdrucks `true`, so wird die Ausführung mit der nachfolgenden Anweisung fortgesetzt. Ansonsten wird in der ersten Variante (ohne den optionalen Zweig) eine Exception `AssertionError` geworfen. In der zweiten Variante (mit optionalem Zweig) wird der Ausdruck hinter dem Doppelpunkt ausgewertet und mit dessen Wert der Konstruktor von `AssertionError` aufgerufen; das daraus resultierende Objekt wird dann geworfen.

Die Klasse `AssertionError` ist von `Error` abgeleitet, und nicht von `Exception`. Solche Fehler sollte man selbst nicht fangen und die Behandlung dem System überlassen. Wird ein `AssertionError` vom System gefangen, so wird das Programm mit einer Meldung abgebrochen.

Man könnte eine Assertion eigentlich auch selber mit einer entsprechenden if-Anweisungen programmieren. Der Vorteil von Assertions ist es aber nun, dass man den hinter einer Assertion stehenden Test selektiv ausführen lassen kann oder nicht, ohne das Programm ändern oder neu übersetzen zu müssen. Assertions werden zuerst einmal wie entsprechenden if-Anweisungen übersetzt. Beim Starten eines Programms über die Java Virtual Machine kann man aber nun angeben, ob Assertions berücksichtigt werden sollen (in der Entwicklungsphase) oder einfach ignoriert werden sollen (im Produktionsbetrieb). Dies geschieht über die

Listing 14.12: Beispiel für den Einsatz von Assertions.

```

14-1  /**
14-2   * Test zu Assertions
14-3   * @author Rudolf Berrendorf
14-4   * @version 1.0
14-5   */
14-6 public class AssertionTest {
14-7
14-8     public static void main(String[] args) {
14-9         float degreeCelsius = Float.parseFloat(args[0]);
14-10
14-11         // hier sollte keine Assertion sein,
14-12         // weil dies das normale Auffangen falscher Benutzereingaben ist
14-13         if(degreeCelsius < -273.15f) {
14-14             //
14-15             System.out.println("falsche Eingabe fuer Temperatur in Grad Celsius");
14-16             // Aus die Maus...
14-17             System.exit(1);
14-18         }
14-19
14-20         // rechne in Kelvin um
14-21         float degreeKelvin = celsiusToKelvin(degreeCelsius);
14-22         // Der Wert muss groesser gleich 0 sein, sonst wuerde etwas bei uns nicht stimmen.
14-23         assert(degreeKelvin >= 0);
14-24         System.out.println("Die Temperatur ist " + degreeKelvin + " Grad Kelvin");
14-25     }
14-26
14-27     // Umrechnen von Celsius nach Kelvin
14-28     static float celsiusToKelvin(float c) {
14-29         // hier ist ein Fehler in der Logik: -1 am Ende ist falsch
14-30         return c + 273.15f - 1;
14-31     }
14-32 }
```

Option **-ea** des Java Laufzeitsystems. Auf weitere Unteroptionen zur Granularität der eingeschalteten Tests wird hier nicht eingegangen. Der Default ist, dass alle Assertions ausgeschaltet sind.

### Beispiel 14.9:

Listing 14.12 zeigt ein Beispiel, wo man Assertions nicht verwenden sollte und wo man es sinnvoll einsetzen kann. In dem Beispiel soll von der Kommandozeile eine Zahl eingelesen werden, die eine Temperaturangabe in Grad Celsius sein soll. Sollte beim Programmstart eine falsche Temperaturangabe angegeben worden sein (die Temperatur wäre unter dem absoluten Nullpunkt), so sollte dies mit der normalen Programmlogik erkannt und entsprechend behandelt werden (hier Programmabbruch), da ein Programm jederzeit mit einer fehlerhaften Eingabe rechnen muss und darauf reagieren muss. An dieser Stelle wäre eine Assertion deshalb nicht angebracht!

Winige Zeilen darunter findet der Aufruf der Methode `celsiusToKelvin` statt, bei der der Temperaturwert von Grad Celsius nach Grad Kelvin umgerechnet wird. Nach dem Methodenaufruf muss (bei korrekter Programmlogik) also ein Wert größer oder gleich 0 vorliegen. In einer Assertion kann man diese Bedingung formulieren, die beider Programmentwicklung, aber nicht mehr im Produktionslauf getestet würde.

Start man das Programm nun in der Entwicklungsphase mit eingeschalteten Assertions, so erscheint folgende Ausgabe:

```

14-1 prompt> java -ea AssertionTest -273
14-2 Exception in thread "main" java.lang.AssertionError
14-3      at AssertionTest.main(AssertionTest.java:22)

```

Hätte man die Entwicklungsphase abgeschlossen und setzt das Programm im Produktivbetrieb ein, so würde ohne die Option `-eq` (also der Default) erscheinen (mit einem Wert, der natürlich nicht so nicht sein darf):

```

14-1 prompt> java AssertionTest -273
14-2 Die Temperatur ist -0.8500061 Grad Kelvin

```

## 14.6 Zusammenfassung und Hinweise

### Literaturhinweise

Auch hierzu gibt es weitere Erläuterungen und Beispiele in den Büchern [HMHG11], [Ull12] und [Sch10].

### Verstehen

Ausnahmen / *Exceptions* dienen dazu, den normalen Programmfluss von der Behandlung von eventuell sogar nur selten auftretenden Ausnahmen zu trennen. Mit *checked* und *unchecked Exceptions* gibt es zwei Arten von Ausnahmen, die unterschiedlich behandelt werden können / müssen. *Exceptions* können über Methodengrenzen hinweg propagiert werden. Im Gegensatz dazu sollen Assertions in der Programmentwicklung dabei helfen, fehlerfreien/-freieren Code zu entwickeln.

### Kurz und knapp merken

- Exceptions werden über eigene Fehlerklassen angegeben. Dazu gibt es bereits eine vordefinierte Hierarchie von Klassen. Man kann aber davon auch eigene Fehlerklassen ableiten.
- *Checked Exceptions* müssen gefangen werden, *unchecked Exceptions* können selbst gefangen werden. Die Klasse `RuntimeException` ist die Basisklasse aller *unchecked Exceptions* (neben der `Error`-Klasse, die aber keine Exception darstellt).
- Über `try-catch`-Anweisungen können Exceptions gefangen werden.
- Über die Reihenfolge von `catch`-Blöcke muss man sich jeweils Gedanken machen. Allgemein gesprochen wird man bei mehreren `catch`-Blöcken von sehr speziellen hin zu allgemeinen Exceptions vorgehen.
- Fängt eine Methode keine in ihr auftretenden Exceptions, so kann (*unchecked Exceptions*) bzw. muss (*checked Exceptions*) sie im Methodenkopf `throws` mit den nicht gefangenen Exception-Klassen spezifizieren. Checked Exceptions müssen irgendwo in der Aufrufhierarchie gefangen werden.
- Eine Exception wird explizit geworfen mit der `throw`-Anweisung, in der ein Exception-Objekt verwendet wird.
- Eine Assertion ist eine Zusicherung des Programmierers, die an dieser Stelle im Programm immer gelten muss. Während Programmläufen in der Entwicklungsphase werden die Assertions überprüft, im Produktionscode sollten diese Überprüfungen über eine Compiler-Option herausgenommen werden.

## Häufige Fehler

Es wird oft nicht berücksichtigt, dass eine `finally`-Anweisung *auf jeden Fall* ausgeführt wird, egal ob eine `catch`-Bedingung vorher erfüllt war oder nicht, und egal, ob ein Fehler aufgetreten ist oder nicht.

Das Propagieren von (*unchecked*) Exceptions über viele Aufrufhierarchien hinweg kann zu Situationen führen, wo ein Fehler an einer ganz anderen Stelle erkannt/behandelt wird, als er eigentlich aufgetreten ist.

## Übungsfragen

- Wozu werden Exceptions eingesetzt?
- Wie kann ich eine eigene Exception programmieren? Was muss ich dabei beachten?
- Wie ist die in Java existierende Hierarchie der Exception-Klassen aufgebaut?
- Wieso gibt es nicht `Exception` / `RuntimeException`, sondern eine ganze Hierarchie mit einer Vielzahl an Fehlerklassen?
- Was ist eine checked / unchecked Exception?
- Wie wird eine Exception ausgelöst?
- Was passiert, wenn eine Exception geworfen wird?
- Ist die Behandlung unterschiedlich von Exceptions, die selbst mit `throw` bzw. vom System geworfen wurden?
- Kann man nach Ausführung eines Exceptions-Handlers wieder mit `return` an die Stelle zurückkehren, wo die Exception geworfen wurde?
- Wie können Exceptions behandelt werden?
- Was ist die Semantik eines try-catch-Blocks?
- Wann wird ein `finally`-Block in einem Exception-Handler ausgeführt?
- Muss ich alle Exception durch try-catch behandeln?
- Was passiert, wenn ich eine checked/unchecked Exception nicht behandle?
- Wie würde man mehrere catch-Blöcke anordnen, wenn die entsprechenden Exceptions aus einer Exception-Hierarchie stammen?
- Was versteht man unter dem Propagieren einer Exception?
- Wann müssen Exceptions in einer Methode propagiert werden?
- Kann ich bei einer check/unchecked Exception diese propagieren, wenn ich selbst nicht lokal behandle? Muss ich checked / unchecked propagieren?
- Was ist der Unterschied zwischen Exception und Assertion?
- Wie gebe ich eine Assertion in Java an?
- Geben Sie ein Beispiel für eine Assertion an!

## Reflektion des Stoffs

- Geben Sie eigene Beispiele dazu an, wo man in der Programmlogik selbst Fehlersituationen vermeiden sollte (ohne Exceptions) beziehungsweise wo die Behandlung über Exceptions Sinn macht.
- Wann machen eigene Fehlerklassen Sinn? Beispiel?
- Diskutieren Sie folgenden Code:

```

14-1 try { throw /* ueberlegen Sie sich einige Faelle hier */ ;
14-2 } catch (Exception e) { System.out.println("hallo 1");
14-3 } catch(MeineException e) { System.out.println("hallo 2");
14-4 } finally { System.out.println("Hallo 3");
14-5 }
```

# Chapter 15

## Komplexitätsbegriff und einfache Such- und Sortierverfahren \*

Mit der Ausführung eines Algorithmus oder auch Programms ist ein gewisser Aufwand verbunden, er werden dabei unterschiedliche Ressourcen genutzt oder verbraucht. Zu solchen Ressourcen gehört zum Beispiel die Nutzung des Prozessors über eine bestimmte Zeit, der genutzte Hauptspeicher, das Übertragungsvolumen über ein Netzwerk und vieles mehr. In diesem Kapitel wird der Komplexitätsbegriff eingeführt, über den in diesem Zusammenhang qualitative Aussagen über die Leistungsfähigkeit und Güte eines Algorithmus möglich ist. Anhand einfacher Such- und Sortierverfahren soll dies dann anschließend praktisch angewandt werden.

### 15.1 Komplexitätsbegriff

Neben der Korrektheit eines Algorithmus, die später in Kapitel 17 noch behandelt wird, ist eine weitere Eigenschaft von Algorithmen in der Praxis oft sehr bedeutend: die Komplexität eines Algorithmus. Unter der **Komplexität** eines Algorithmus ist der Aufwand an Ressourcen zu verstehen, der zur Bearbeitung des Algorithmus notwendig ist. Ressourcen sind insbesondere die Laufzeit des Algorithmus (Rechenzeit), das heißt die abgelaufene Zeit vom Start des Algorithmus bis zum Ende des Algorithmus, sowie der erforderliche Speicherplatz, meist bezogen auf den Hauptspeicher. Von Fall zu Fall können auch weitere Ressourcen von Interesse sein, wie etwa die benötigte Anzahl von Prozessoren, entstehende Kosten, Stromverbrauch, Datenbankplatz, Übertragungsvolumen über ein Netzwerk und vieles mehr. Dies ergibt sich üblicherweise aus dem Zusammenhang oder wird explizit vorgegeben.

Algorithmen nehmen gewöhnlich Eingabedaten entgegen und führen mit diesen Verarbeitungsschritte durch. Deshalb wird der Bedarf an Ressourcen oft mit der Menge an Eingabedaten oder in Abhängigkeit von ihrem Inhalt schwanken. Man kennt dieses Verhalten auch aus dem Alltagsleben: will man zum Beispiel Spielkarten bei einem Kartenspiel in der Hand sortieren, so wird dies umso länger dauern, je mehr Karten man in der Hand hält.

Den konkreten Ressourcenbedarf eines Algorithmus könnte man durch Implementierung des Algorithmus in einer konkreten Programmiersprache auf einem bestimmten Rechner mit einer Menge repräsentativ ausgewählter Eingabedaten konkret messen, indem man die Rechenzeit ermittelt, den Speicherbedarf konkret bestimmt und so weiter. Die dadurch gewonnenen Resultate bezüglich der erforderlichen Ressourcen lassen sich aber nur schwer verallgemeinern. Zu einem anderen Rechner oder für andere Eingabedaten lassen sich mit den gewonnenen Erkenntnissen eventuell keine Aussagen machen. Aus diesem Grunde ermittelt man für einen Algorithmus zuerst einmal **charakterisierende Parameter**, anhand derer man *allgemeingültige Komplexitätsaussagen* über den Ressourcenbedarf des Algorithmus angeben kann. Es geht also nicht mehr

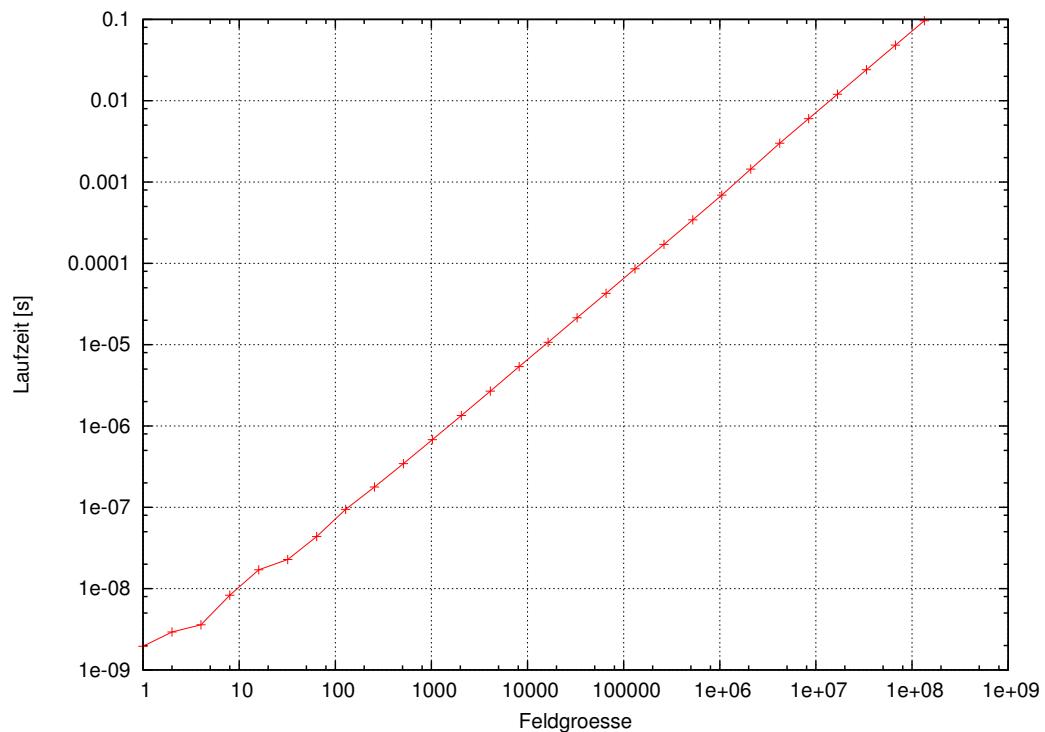


Figure 15.1: Laufzeit des Java-Programms zum Aufsummieren aller Feldelemente in Abhängigkeit der Feldgröße auf einem konkreten Rechner.

darum, auf die Millisekunde genau eine Vorhersage für einen bestimmten Algorithmus auf einem bestimmten Rechner machen zu können, sondern vielmehr darum, größenzahlmäßig anzugeben, wie sich der Algorithmus an sich verhält, im Wesentlichen unabhängig von einem konkreten Rechner oder einer konkreten Programmiersprache.

Was man gerne hätte, wäre also eine Komplexitätsfunktion, über die der Ressourcenverbrauch eines Algorithmus in Abhängigkeit von charakterisierenden Parametern spezifiziert ist:  $f(x,y,\dots) = \dots$ . Kennt man die Komplexitätsfunktion eines Algorithmus, so lassen sich daraus auch Vorhersagen bezüglich des Ressourcenverbrauchs angeben.

Listing 15.1: Aufsummieren von Feldinhalten.

```

15-1  /**
15-2   * Berechne Zeitaufwand fuer die Summation von Feldelementen
15-3   * in Abhaengigkeit der Feldlaenge
15-4   */
15-5  public class SummeBerechnen {
15-6
15-7      // Summe ueber alle Feldelemente berechnen
15-8      public static int summeBerechnen(int[] feld) {
15-9          int summe = 0;
15-10         for(int i=0; i<feld.length; i=i+1) {
15-11             summe = summe + feld[i];
15-12         }
15-13         return summe;
15-14     }
15-15
15-16     // Feld anlegen und mit Beispielwerten initialisieren
15-17     public static int[] erzeugen(int n) {
15-18         int[] feld = new int[n];
15-19         for(int i=0; i<feld.length; i++)
15-20             feld[i] = i;
15-21         return feld;
15-22     }
15-23
15-24     public static void main(String[] args) {
15-25         // maximale Groesse Feld
15-26         final int N_MAX = 1024*1024*128;
15-27         /* Die Zeitaufloesung ist zu gering, so dass wir den Code
15-28            mehrfach ausfuehren muessen.
15-29         */
15-30         final long ANZAHL_OPERATIONEN = N_MAX * 64L;
15-31
15-32         System.out.println("n    Zeit_in_sekunden");
15-33         for(int n=1; n<=N_MAX; n=n*2) {
15-34             long iterationen = ANZAHL_OPERATIONEN / n;
15-35             int[] feld = erzeugen((int)n);
15-36             // Startzeit nehmen
15-37             long t0 = System.currentTimeMillis();
15-38             // Operation mehrfach wiederholen wegen Zeitaufloesung
15-39             for(long iter=0; iter<iterationen; iter++) {
15-40                 int summe = summeBerechnen(feld);
15-41             }
15-42             // Endzeit nehmen und Differenz bilden
15-43             long t1 = System.currentTimeMillis();
15-44             System.out.println(n + " " + (((t1-t0)/(double)iterationen)/1000.0));
15-45         }
15-46     }
15-47 }
```

**Beispiel 15.1:**

Gegeben sei ein Feld mit  $n$  ganzen Zahlen. Bestimme die Summe über alle Feldelemente. Das Programm in Listing 15.1 zeigt eine Umsetzung.

Abbildung 15.1 zeigt die Laufzeit dieses Java-Programms in Abhängigkeit der Feldlänge auf einem konkreten Rechner. Wie man leicht erkennen kann, braucht dieses Programm umso mehr Rechenzeit, je größer das Feld ist, da alle Zahlen nacheinander auf die Gesamtsumme aufaddiert werden. Der für die Laufzeit charakterisierende Parameter dieses Algorithmus ist in diesem Falle also *die Anzahl der aufzusummierenden Zahlen* in dem Feld, also die Länge des Feldes. Verdoppelt man für eine große Anzahl Zahlen diese Anzahl, wird sich in etwa auch die Laufzeit des Algorithmus verdoppeln, vervierfacht man die Anzahl wird sich in etwa auch die Laufzeit vervierfachen. Es gibt also in diesem Beispiel bei einer großen Anzahl an Zahlen einen proportionalen Zusammenhang zwischen der Anzahl der Zahlen und der Laufzeit des Algorithmus, es liegt also eine lineare Wachstumsfunktion  $f(n) = a \cdot n$  in Abhängigkeit der Anzahl  $n$  von Feldelementen vor. Egal, auf welchem Rechner man diesen Algorithmus ausführt, welche Programmiersprache man verwendet, welche konkreten Zahlen in dem Feld vorliegen, wird dieser Zusammenhang erkennbar sein.

Wie man ebenfalls leicht in Abbildung 15.1 sehen kann, gilt diese Verdopplungseigenschaft der Laufzeit nicht unbedingt bei einer sehr kleinen Anzahl von Feldelementen. Dort zeigt die Kurve kein so eindeutiges Verhalten wie bei einer großen Feldgröße. Dies liegt daran, dass im vorliegenden Programm zum Beispiel die Zuweisung `int summe = 0;` oder `return summe;` genau einmal ausgeführt wird, unabhängig davon, wie viele Zahlen aufaddiert werden sollen. Der Einfluss dieser Anweisungen und auch weitere, hier nicht erläuterte Einflüsse sind also bei einer sehr kleinen Anzahl an Zahlen nicht vernachlässigbar, bei einer großen Anzahl von Zahlen aber sehr wohl. ♦

Die Laufzeit des Algorithmus im letzten Beispiel hing eindeutig von der Anzahl der Feldelemente ab. Die Werte der Feldelemente spielten dabei keine Rolle, bei anderen Werten würde man den gleichen Kurvenverlauf sehen. Nun kann es aber auch sein, dass die Laufzeit eines Algorithmus nicht nur mit der Anzahl der Eingabedaten schwanken kann, sondern auch mit ihrem Inhalt. Ein Beispiel ist das Suchen einer Zahl in einem Feld von Zahlen.

**Beispiel 15.2:**

Listing 15.2 zeigt eine Methode, die feststellt, ob ein bestimmter Wert in einem Feld von Werten vorkommt.

Dieser Algorithmus zum Auffinden eines Wertes in einem Feld geht sequentiell von vorne nach hinten das Feld durch und vergleicht jedes Feldelement mit dem gesuchten Wert. In diesem Beispiel soll die Anzahl der Vergleichsoperationen als Aufwand/Ressource betrachtet werden. Die Anzahl an Vergleichsoperationen wird sehr gering sein, wenn der gesuchte Wert zu Beginn des Feldes gefunden wird und die Anzahl der Vergleichsoperationen wird sehr groß sein, wenn der gesuchte Wert erst am Ende des Feldes gefunden wird oder garnicht im Feld vorhanden ist (Ergebniswert -1). ♦

Listing 15.2: Suchen eines Wertes in einem Feld.

```

15-1 public static int sucheZahl(int[] feld, int wert) {
15-2     for(int i=0; i<feld.length; i++) {
15-3         if(feld[i] == wert)
15-4             return i;
15-5     }
15-6     return -1;
15-7 }
```

Um auch in solchen Fällen entsprechende Komplexitätsabschätzungen machen zu können, kann man unterscheiden zwischen dem besten möglichen Fall **best-case**, dem durchschnittlich zu erwartenden Fall **average-case** und dem schlimmsten zu erwartenden Fall **worst case**.

Im Suchbeispiel oben ist der günstigste Fall, dass der erste Eintrag bereits der gesuchte Eintrag ist, das heißt, der *best case* Aufwand ist genau eine Vergleichsoperation. Im schlechtesten Fall wird der gesuchte Eintrag der letzte Eintrag des Feldes sein beziehungsweise garnicht vorkommen. Das heißt der *worst case* Aufwand sind  $n$  Vergleichsoperationen, wenn das Feld  $n$  Elemente enthält. Geht man von einer Gleichverteilung der Werte in einem unsortierten Feld aus, und wiederholt vielfach die Suche mit verschiedenen, ebenfalls zufälligen Werten aus dem Feld, so wird der *average case* Aufwand  $n/2$  sein.

Zu beachten ist in diesem Beispiel, dass die *best case* Komplexitt unabhangig von der Anzahl der Elemente ist, namlich genau eine Vergleichsoperation, egal, ob das Feld ein Element oder eine Million Elemente enthalt. In den meisten Fallen ist die Bestimmung einer *worst case* oder eventuell *average case* Komplexitt von Interesse. Wahrend die Bestimmung eines *best case* oder *worst case* Aufwands in der Praxis oft relativ einfach bestimmbar ist, ist eine *average case* Analyse oft vergleichsweise schwierig.

Um aber *prinzipiell* den Aufwand für ein Programm abschätzen zu können, abstrahiert man oft vom konkreten Aufwand für unterschiedliche Operationen und wendet die folgenden Regeln an, die sich an den Konstrukten einer Programmiersprache orientieren:

- Einfache Operationen wie Zuweisung, Addition, Subtraktion, Feldindizierung und so weiter verursachen 1 Kosteneinheit.
  - Seien  $A_1, \dots, A_n$  Anweisungen mit Kosten  $k_1, \dots, k_n$ . Dann hat die Sequenz der Anweisungen  $A_1; \dots; A_n$  die Kosten  $k_1 + \dots + k_n$ .
  - Seien  $A_1$  und  $A_2$  Anweisungen mit Kosten  $k_1$  und  $k_2$  sowie  $B$  eine Bedingung, deren Auswertung Kosten  $k_3$  verursacht. Dann hat die Selektion  
**if** ( Bedingung )  $A_1$  **else**  $A_2$  minimal die Kosten  $k_3 + \min(k_1, k_2)$  und maximal die Kosten  $k_3 + \max(k_1, k_2)$ .
  - Sei  $A$  eine Anweisung mit Kosten  $k_1$  und  $B$  eine Bedingung, deren Auswertung Kosten  $k_2$  verursacht. Dann hat die Iteration  
**while** (  $B$  ) **do**  $A$  die Kosten  $n \cdot (k_1 + k_2) + k_2$ , falls die Bedingung  $n$  mal wahr ist und beim  $n+1$  mal die Bedingung falsch ist. Andere Formen der Iteration analog.

### Beispiel 15.3:

Ausgangspunkt ist das Beispiel von eben. Aus Gründen der direkten Anwendung obiger Regeln wird der Algorithmus äquivalent mit einer while-Schleife angegeben:

Angenommen, die Schleifenbedingung `i < feld.length` sei für gegebene Daten  $n$ -mal richtig und der gesuchte Wert ist das  $(n+1)$ -te Feldelement. Dann ist der konkrete Aufwand für diesen Algorithmus für diese gegebenen Daten  $1 + (n \cdot (1 + 2 + 2) + 2 + 1 = 4 + 5n)$  Basisoperationen / Kosteneinheiten. Denn  $S_1$  wird zu Beginn einmal ausgeführt. Dann werden insgesamt  $n$ -mal die Schleifenbedingung in  $S_2$  ausgewertet sowie die Anweisung  $S_3$  und  $S_5$  ausgeführt. Im  $(n+1)$ -ten Durchlauf wird die Schleifenbedingung nochmals zu `true` ausgewertet, die Anweisung  $S_3$  ausgewertet und dann als Ergebnis in  $S_4$  `true` geliefert. Daraus ergibt sich insgesamt der Aufwand  $f(n) = 4 + 5n$  Operationen. ♦

### Beispiel 15.4:

Im Beispiel zur Summation aller Feldelemente ist der Aufwand an Operationen wie folgt:

```

15-1  public static int summeBerechnen(int[] feld) {
15-2      int summe = 0;           // S1: 1 Operation
15-3      int i = 0;             // S2: 1 Operation
15-4      while(i < feld.length) { // S3: 2 Operationen
15-5          summe = summe + feld[i]; // S4: 3 Operationen
15-6          i = i + 1;           // S5: 2 Operationen
15-7      }
15-8      return summe;         // S6: 1 Operation
15-9  }
```

Hat das Feld insgesamt  $n$  Elemente, so ergibt sich als Aufwand  $1 + 1 + n \cdot (2 + 3 + 2) + 2 + 1$  und damit als Aufwandsfunktion  $f(n) = 7 \cdot n + 5$ . ♦

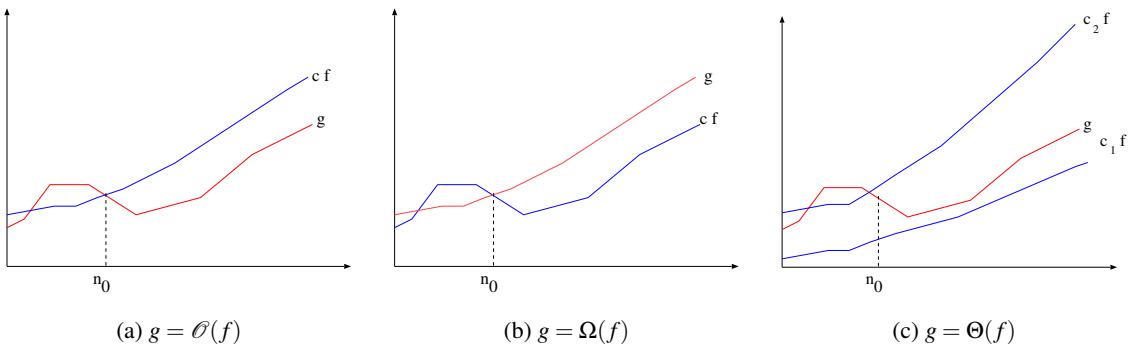
Nachdem man zu den einzelnen Konstrukten eines gegebenen Programms den Aufwand bestimmt hat, versucht man anschließend, den funktionalen Zusammenhang in Abhängigkeit von charakterisierenden Parametern zu bestimmen. Im letzten Beispiel wäre der charakterisierende Parameter die Länge des Feldes (diese Länge soll  $n$  bezeichnet sein) und der Berechnungsaufwand in Anzahl Operationen wäre dann gegeben durch die Funktion  $f(n) = 7 \cdot n + 5$ .

Meist ist man nur am *prinzipiellen* Verlauf der Aufwandsfunktion in Abhängigkeit der charakterisierenden Größen interessiert. Für eine Aufwandsfunktion, die zum Beispiel quadratisch wie  $f(x) = a + b \cdot x^2$  mit Konstanten  $a, b \in \mathbb{R}$  wächst, sind die linearen Anteile wie im Beispiel  $a$  und selbst die konstanten Faktoren wie  $b$  ab einer bestimmten Größe von  $x > 1$  in der Komplexitätsdiskussion weniger interessant, ja oft geradzu vernachlässigbar. Denn ein konstanter Faktor  $x$  könnte dadurch ausgeglichen werden, wenn man zum Beispiel einen  $x$ -fach schnelleren Rechner nutzt. Und solche Einflüsse will man ja gerade in der jetzigen Diskussion abstrahieren!

Während konstante Faktoren also weniger interessant sind, ist es aber ganz wesentlich in solch einer Diskussion, ob die Aufwandsfunktion linear oder quadratisch oder gar exponentiell wächst. Genau dies ist nämlich die gewollte Aussage. In unserem Beispiel ist es für eine qualitative Aussage für großes  $n$  unerheblich, ob die Laufzeitfunktion  $f(n) = 4 + 5 \cdot n$  ist oder  $f(n) = 100 + 5 \cdot n$ , wie man leicht einsieht, wenn man zum Beispiel für  $n$  den Wert  $10^{10}$  einsetzt.

Zur Angabe eines solchen prinzipiellen Wachstums, also der *Größenordnung des Wachstums*, gibt es drei Schreibweisen, um Funktionen asymptotisch nach oben, nach unten beziehungsweise nach oben und unten abzuschätzen und lineare Terme und kostante Faktoren zu vernachlässigen.

Zur Angabe der asymptotischen oberen Grenze benutzt man die **O-Notation**, zur Angabe einer asymptotisch unteren Schranke nutzt man die  $\Omega$ -Notation und will man sowohl eine asymptotisch obere als auch eine untere Schranke angeben, so ist dies mit der  $\Theta$ -Notation möglich.

Figure 15.2: Grafische Beispiele zur Veranschaulichung von  $\mathcal{O}(f)$ ,  $\Omega(f)$  und  $\Theta(f)$ .**Definition 15.1 (O-Notation):**

Sei  $\mathfrak{F}$  die Menge aller Funktion  $f : \mathbb{N} \rightarrow \mathbb{R}_+$ , für die ein  $n_0 \in \mathbb{N}$  existiert, so dass  $f(n) > 0$  für alle  $n \geq n_0$ . Für eine gegebene Funktion  $f \in \mathfrak{F}$  sei:

$$\mathcal{O}(f) := \{g \in \mathfrak{F} \mid \exists c > 0, \exists n_0 \geq 0, \forall n \geq n_0 : c \cdot f(n) \geq g(n)\}$$

$$\Omega(f) := \{g \in \mathfrak{F} \mid \exists c > 0, \exists n_0 \geq 0, \forall n \geq n_0 : c \cdot f(n) \leq g(n)\}$$

$$\Theta(f) := \{g \in \mathfrak{F} \mid \exists c_1, c_2 > 0, \exists n_0 \geq 0, \forall n \geq n_0 : c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}$$

◆

Mit  $\mathcal{O}(f)$ ,  $\Omega(f)$  und  $\Theta(f)$  werden also für eine vorgegebene Funktion  $f$  eine (undendliche) Menge von Funktionen definiert, die ein bestimmtes Kriterium hinsichtlich  $f$  erfüllen. Man spricht auch von einer Funktionenklasse. Abbildung 15.2 verdeutlicht in grafischer Form die Bedeutung der Parameter in den drei Schreibweisen. Üblicherweise, was aber streng genommen eine inkorrekte Schreibweise ist, benutzt man abkürzenderweise eine Schreibweise wie  $\mathcal{O}(n^2)$  (mit  $n$  bezeichnet man dabei den charakterisierenden Parameter) statt korrekterweise  $\mathcal{O}(f)$  mit  $f(n) = n^2$  zu notieren. Genauso hat es sich eingebürgert  $g = \mathcal{O}(f)$  zu schreiben, statt der eigentlich korrekten Schreibweise  $g \in \mathcal{O}(f)$ .

Für einige wichtige Funktionsklassen haben sich folgende Notationen eingebürgert:

O-Notation	Name	Beispiel
$\mathcal{O}(1)$	konstant	
$\mathcal{O}(\log n)$	logarithmisch	binäre Suche
$\mathcal{O}(n)$	linear	sequentielle Suche
$\mathcal{O}(n \log n)$	$n \log n$	gute Sortierverfahren
$\mathcal{O}(n^2)$	quadratisch	schlechte Sortierverfahren
$\mathcal{O}(n^3)$	kubisch	
$\mathcal{O}(2^n)$	exponentiell	Matrixmultiplikation

Was man mit diesen Notationen also beschreiben will ist die *Größenordnung* einer Aufwandsfunktion eines Algorithmus. Also, zum Beispiel ob die Laufzeit eines Algorithmus sich konstant verhält, ob sie höchstens linear wächst, ob sie höchstens quadratisch wächst und so weiter. Irgendwelche Spezialfälle zu Beginn des Funktionsverlaufs (wie in Abbildung 15.1), wo sich noch kein klares Bild ergibt, filtert man über die Angabe des  $n_0$  heraus.

Als Anmerkung sei erwähnt, dass man üblicherweise die kleinste umschließende Klasse angibt, also die schärfste Schranke. Denn liegt zum Beispiel eine Aufwandsfunktion  $g$  in  $\mathcal{O}(n^2)$ , so liegt sie automatisch auch in  $\mathcal{O}(n^3), \mathcal{O}(n^4)$  und jedem  $\mathcal{O}(n^k)$  für  $k > 2$ , wie man leicht beweisen kann (Übungsaufgabe).

Will man nun beweisen, dass eine bestimmte Aufwandsfunktion  $g$  in der Komplexitätsklasse  $\mathcal{O}(f)$  für ein bestimmtes  $f$  liegt, so muss man überprüfen, ob die Definition für  $f$  hinsichtlich des  $g$  erfüllt ist. Das heißt also, man muss *konkrete* Werte für  $c$  und  $n_0$  angeben und anschließend zeigen, dass für alle  $n \geq n_0$  die Behauptung gilt, dass  $c \cdot f(n) \geq g(n)$  für die gegebenen  $f$  und  $g$ .

### Beispiel 15.5:

Behauptung: Die Funktion  $g(n) = 3n^2 + 6n + 7$  ist in  $\mathcal{O}(n^2)$ .

Zum Beweis dieser Behauptung müssen also geeignete  $n_0$  und  $c$  angegeben werden und anschließend gezeigt werden, dass für alle  $n \geq n_0$ :  $g(n) \leq c \cdot n^2$ .

Zuerst wird durch geeignete Abschätzungen  $n_0$  und  $c$  angegeben. Für alle  $n > 0$  gilt:

$$3n^2 + 6n + 7 \leq 3n^2 + 6n^2 + 7n^2 = 16n^2$$

Damit sind aber bereits geeignete Konstanten gefunden. Denn mit  $c = 16$  und  $n_0 = 1$  gilt dann für alle  $n \in \mathbb{N}, n \geq n_0$ :  $g(n) \leq c \cdot n^2$  oder explizit ausgeschrieben:  $3n^2 + 6n + 7 \leq 16 \cdot n^2$ . Und damit ist also  $g(n) = \mathcal{O}(n^2)$ . ♦

### Beispiel 15.6:

Behauptung: Die Funktion  $h(n) = n^3 + 12 \cdot \log(n) + 7$  ist in  $\mathcal{O}(n^3)$ .

Zum Beweis dieser Behauptung muss man also wieder geeignete  $n_0$  und  $c$  finden und damit zeigen, dass für alle  $n \geq n_0$ :  $h(n) \leq c \cdot n^3$ .

Auch hier kann abgeschätzt werden. Es gilt für alle  $n > 0$ :

$$n^3 + 12 \cdot \log(n) + 7 \leq n^3 + 12n^3 + 7n^3 \leq 20n^3$$

Mit  $c = 20$  und  $n_0 = 1$  gilt dann für alle  $n \in \mathbb{N}, n \geq n_0$ :  $h(n) \leq c \cdot n^3$  oder explizit ausgeschrieben:  $n^3 + 12 \cdot \log(n) + 7 \leq 20 \cdot n^3$ . Also gilt  $h(n) = \mathcal{O}(n^3)$ . ♦

Das Vorgehen in den beiden Beispielen war ähnlich: Durch teilweise sehr großzügiges Abschätzen kann man direkt Werte für  $n_0$  und  $c$  ablesen und damit die Behauptung zeigen. Man orientiert sich in diesen einfachen Beispielen dabei am dominierenden Term, bei Polynomen ist dies die höchste vorkommende Potenz. In den beiden Beispielen waren dies  $n^2$  und  $n^3$ .

Mit  $\Theta$  wird eine Äquivalenzrelation auf  $\mathfrak{F}$  angegeben. Das heißt für  $f, g, h \in \mathfrak{F}$  gilt:

- |  |   |
|--|---|
| 1. $f(n) = \Theta(f(n))$<br>2. $f(n) = \Theta(g(n)) \Rightarrow g(n) == \Theta(f(n))$<br>3. $f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$ | $\Theta$ ist reflexiv<br>$\Theta$ ist symmetrisch<br>$\Theta$ ist transitiv |
|--|---|

Man beachte, dass  $O$  und  $\Omega$  *keine* Äquivalenzrelationen sind, da dafür die Symmetrieeigenschaft im Allgemeinen nicht gilt.

Es gelten folgende Eigenschaften für  $f, g, h \in \mathfrak{F}$ :

- 1)  $f(n) = \mathcal{O}(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$
- 2)  $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \mathcal{O}(g(n)) \wedge f(n) = \Omega(g(n))$
- 3)  $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$

- 4)  $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \mathcal{O}(g(n)) \wedge f(n) = \Omega(g(n))$
- 5)  $\mathcal{O}(f(n)) = \mathcal{O}(g(n)) \Leftrightarrow \Omega(f(n)) = \Omega(g(n)) \Leftrightarrow \Theta(f(n)) = \Theta(g(n))$
- 6) Für eine beliebige positive Konstante  $c$  gilt:  $\Omega(f(n)) = \Omega(c \cdot f(n))$ ,  $\Theta(f(n))\Theta(c \cdot f(n))$ ,  $\mathcal{O}(f(n))\mathcal{O}(c \cdot f(n))$ .
- 7) Falls für zwei Funktionen  $g$  und  $h$  gilt:  $\forall n \in \mathbb{N} : g(n) \leq h(n), h(n) = \mathcal{O}(f(n))$ , so gilt  $g(n) = \mathcal{O}(f(n))$ .
- 8)  $\mathcal{O}(\mathcal{O}(f(n))) = \mathcal{O}(f(n))$

Und weiterhin gelten folgende Rechenregeln für  $f, g, h \in \mathfrak{F}$ :

- 1)  $c \cdot f(n) = \mathcal{O}(f(n))$  für  $c \in \mathbb{R}$ .  
Konstante Faktoren sind vernachlässigbar.
- 2) Für Polynome der Form  $g(n) = a_0n^0 + a_1n^1 + \dots + a_{k-1}n^{k-1} + a_kn^k$  gilt:  $\mathcal{O}(g(n)) = \mathcal{O}(n^k)$ .  
Die höchste Potenz eines Polynoms dominiert.
- 3)  $\mathcal{O}(\log_b(n)) = \mathcal{O}(\log(n))$ .  
Die Basis für einen Logarithmus ist vernachlässigbar.
- 4)  $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max(f(n), g(n)))$
- 5)  $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$

Der Beweis dieser Regeln geschieht jeweils über den Nachweis der Gültigkeit der definierenden Kleiner-Eigenschaft mit Angabe geeigneter  $n_0$  und  $c$  und ist dem Leser zur Übung überlassen. Insbesondere die letzten beiden Rechenregeln haben direkten Bezug zur Abschätzung von Komplexitäten größerer Programmteile. Die Regel  $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max(f(n), g(n)))$  besagt, dass in einer Sequenz von zwei Programmteilen der Programmteil mit der höheren Komplexität dominiert. Die Regel  $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$  ist anwendbar auf geschachtelte Schleifen, wo man die Komplexität des Schleifenrumpfs und die Anzahl der Schleifendurchläufe kennt und durch Produktbildung die Gesamtkomplexität bestimmen kann.

Für einen praktischen Einsatz selbst auf schnellsten Rechnern sind solche Algorithmen geeignet, deren Komplexität maximal  $\mathcal{O}(n^k)$  für eine Konstante  $k$  ist, also maximal polynomial wachsen. Algorithmen mit einer exponentiellen Komplexität sind auch auf den schnellsten Rechnern selbst für relativ kleine Eingabegrößen nicht mehr durchführbar, weil das Wachstum einer exponentiellen Funktion einfach zu gross ist.

### Beispiel 15.7:

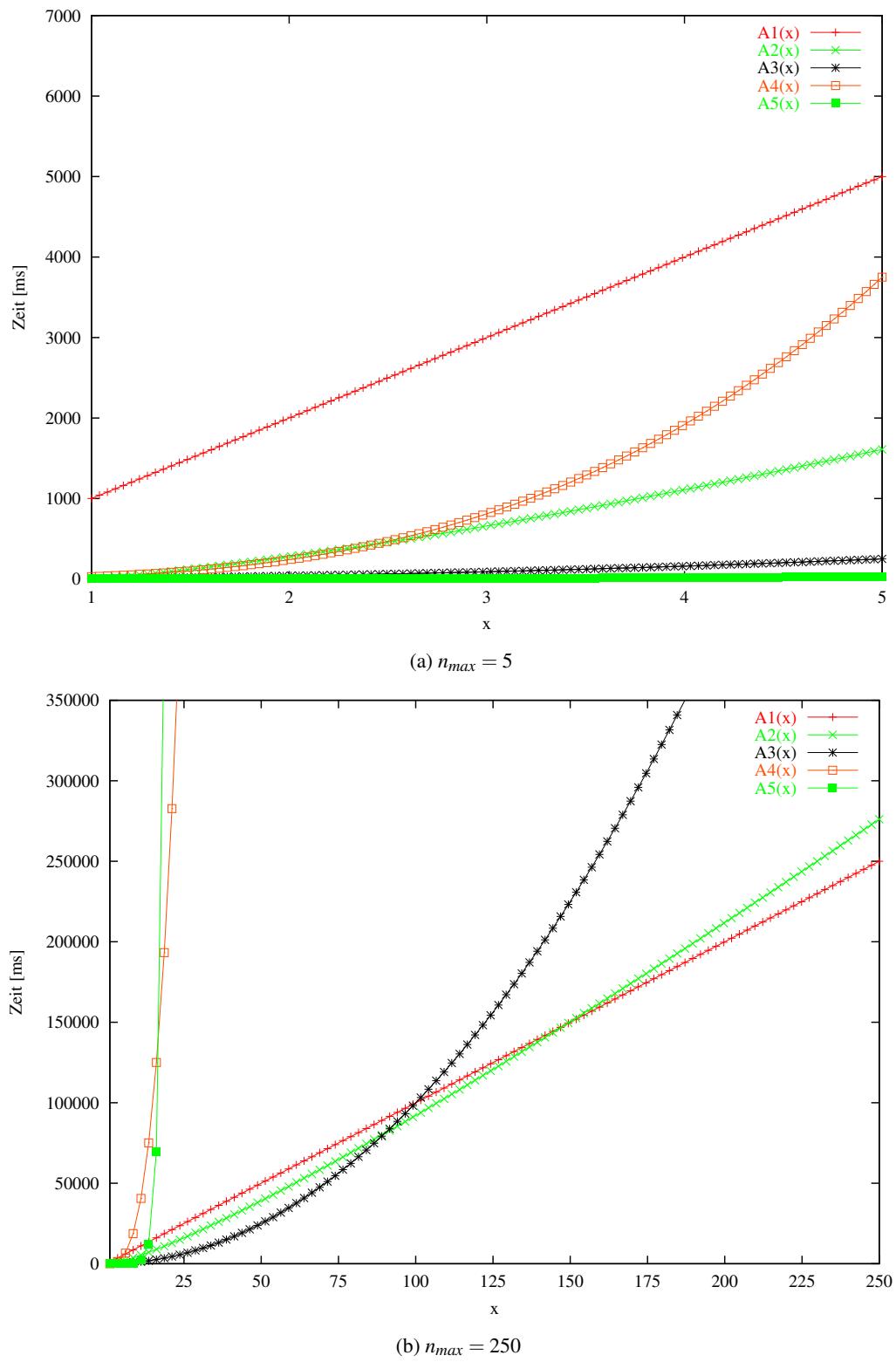
Um dies zu verdeutlichen seien 5 Algorithmen gegeben:

Algorithmus	Laufzeit in ms in Abh. von $n$	am schnellsten für	max. rechenbare Problemgrösse in 3.600 s	in 36.000 s
A1	$1.000n$	$n \geq 149$	3.600	36.000
A2	$200n \log n$	$90 \leq n \leq 148$	1.500	13.500
A3	$10n^2$	$10 \leq n \leq 89$	600	1.900
A4	$30n^3$	nie	50	106
A5	$2^n$	$1 \leq n \leq 9$	22	25

Diese Tabelle zeigt, dass *keine* Algorithmusvariante für alle  $n$  am besten ist, das heißt immer die geringste Laufzeit benötigt. Vielmehr gibt es in diesem Beispiel Intervalle, in denen jeweils eine Algorithmusvariante die beste ist.

Der Übergang von 3.600 zu 36.000 Sekunden in der letzten Spalte der Tabelle ist qualitativ gleichbedeutend mit der Aussage, dass man 3.600 s auf einem 10 mal schnelleren Rechner rechnen kann. Abbildungen 15.3a und 15.3b zeigen in grafischer Weise den Funktionsverlauf für kleine  $n$  und große  $n$ .

An dieser Stelle soll eine Aufwandsabschätzung für die Methoden `fac` und `fib` vorgenommen werden, die schon vorher eingeführt wurde. Die rekursive Berechnung der Fakultätsfunktion geschieht mit der Methode,

Figure 15.3: Verlauf der Funktionen für eine maximale Problemgrösse von  $n_{max} = 5$  und  $n_{max} = 250$ .

Listing 15.3: Fakultätsberechnung.

```

15-1  /**
15-2   * Fakultaet rekursiv
15-3   */
15-4  public class Fakultaet {
15-5
15-6      /**
15-7       * Fakultaetsfunktion
15-8       * @param n Argument
15-9       * @return n!
15-10      */
15-11     static int fakultaet(int n) {
15-12         int wert = 0;
15-13         if (n == 0) {
15-14             return 1;
15-15         } else {
15-16             wert = fakultaet(n-1);
15-17             return n * wert;
15-18         }
15-19     }
15-20
15-21     public static void main(String [] args) {
15-22         System.out.println("fakultaet(2)=" + fakultaet(2));
15-23     }
15-24
15-25 }
```

wie sie in Listing 15.3 angegeben ist. Für die Methode `fakultaet` gilt: Ist der Wert von `n` gleich 0, so ist der Aufwand gleich einer Konstanten  $c_1$ , weil eine konstante Anzahl von Operationen ausgeführt werden muss. Ist der Wert von `n` echt größer 0, so ist der Aufwand gleich einer Konstanten  $c_2$  plus dem Aufwand für den Aufruf von `fakultaet(n-1)`. Durch Induktion gelangt man zum Schluss, dass der Gesamtaufwand also  $c_1 + c_2 + \dots + c_n$  ist. Für  $c = \max(c_1, \dots, c_n)$  erhält man damit eine Abschätzung von  $n \cdot c$  und damit eine Komplexität von  $\mathcal{O}(n)$ .

Die rekursive Umsetzung der Fibonacci-Funktion ist im Programm in Listing 15.4 zu sehen. Für die Methode `fib` gilt: Ist das Argument `n` gleich 0 oder 1 so ist der Aufwand durch eine Konstante  $c_1$  beschränkt. Ist das Argument echt größer 1, so ist der Aufwand gleich einer Konstanten  $c_2$  plus dem Aufwand für `fib(n-2)` plus dem Aufwand für `fib(n-1)`. Der Aufruf von `fib(n-1)` bedeutet wiederum einen Aufwand  $c_3$  oder  $c_4$  plus dem Aufwand für `fib(n-3)` plus dem Aufwand für `fib(n-2)` und so weiter. Insgesamt ergibt sich damit ein Aufwand von  $c \cdot 2^n$  für  $c = \max(c_1, \dots, c_m)$  oder  $\mathcal{O}(2^n)$ .

Listing 15.4: Fibonacci-Funktion.

```

15-1  /**
15-2   * Fibonacci-Funktion rekursiv
15-3   */
15-4  public class Fibonacci {
15-5
15-6      /**
15-7       * Fibonacci-Funktion
15-8       * @param n Argument
15-9       * @return Wert der Fibonacci-Folge an der Stelle n
15-10      */
15-11     static int fibonacci(int n) {
15-12         if (n == 0) {
15-13             return 0;
15-14         } else if (n == 1) {
15-15             return 1;
15-16         } else {
15-17             return fibonacci(n-2) + fibonacci(n-1);
15-18         }
15-19     }
15-20 }
```

## 15.2 Einfache Suchverfahren

Eine oft zu lösende Aufgabe ist es zu bestimmen, ob ein bestimmter Wert in einer vorhandenen Wertemenge vorkommt oder nicht. Beispielsweise ist zu entscheiden, ob eine gerade gezogene Lottozahl in meinen getippten Zahlen vorkommt oder nicht. Oder, ob ein Teilnehmer einer Veranstaltung schon bezahlt hat oder nicht.

Im Folgenden soll das eben sehr allgemein formulierte Problem des Suchens etwas präzisieren beschrieben und dabei auch eingeschränkt werden. Man geht im Folgenden davon aus, dass alle vorkommenden Werte ganze Zahlen sind und der Wertebestand in einem Feld von  $n$  Zahlen vorliegt. Ausgehend von dieser Problemformulierung unterscheiden wir dann zwei Fälle. Im ersten Fall macht man keine Annahmen über den Inhalt des Feldes. Im zweiten Fall geht man davon aus, dass die Daten im Feld sortiert vorliegen und nutzen dies bei unserer Suche aus.

### 15.2.1 Sequentielle Suche

Im ersten betrachteten Fall geht man davon aus, dass ein Feld mit  $n$  Zahlen vorliegt, in dem man den Wert suchen sollen, und dass man keinerlei Annahmen über diese Zahlen machen kann. Im Zweifelsfall muss man damit bei jedem Feldelement nachschauen, ob dieses Feldelement der gesuchte Wert ist oder nicht (Überlegen Sie sich, wieso das so ist). Man will aber auch nicht im Laufe der Verfahrens bei einem Feldelement mehrfach überprüfen, ob dieses Element der gesuchte Wert ist oder nicht. Insofern muss man sich also eine *Besuchsstrategie* überlegen, die bei jedem Feldelement genau einmal vorbeischaut (*Besuchseigenschaft*). Die einfachste Möglichkeit dazu im Zusammenhang mit einem Feld ist das Durchlaufen des Feldes von vorne nach hinten, denn damit schaue ich mir ein Feldelement nach dem anderen an und überprüfe kein Element mehrfach. Alternativ kann man von hinten nach vorne das Feld durchlaufen oder auch andere Strategien entwickeln, die die Besuchseigenschaft haben.

In Kapitel 9.1.1 wurde im Zusammenhang mit Feldern bereits ein solches Verfahren vorgestellt, mit dem man in einem Feld sequentiell durchlaufend einen vorgegebenen Wert suchen kann. Zur Wiederholung ist das Programm in Listing 15.5 nochmals angegeben.

Listing 15.5: Sequentielles Suchen in einem Feld.

```

15-1  /**
15-2   * Sequentielles Suchen in einem Feld als Methode
15-3   */
15-4 public class SequentiellesSuchen2 {
15-5
15-6     /**
15-7      * @param a Feld mit Werten
15-8      * @param x gesuchter Wert
15-9      * @result Position (Index), an der Wert gefunden wurde. Ansonsten -1
15-10     */
15-11    public static int suchen(int[] a, int x) {
15-12        int position = -1;
15-13        // durchlaufe alle Feldelemente
15-14        for(int i=0; i<a.length; i++) {
15-15            if(a[i] == x) {
15-16                // Wert gefunden, wir koennen aufhoeren
15-17                position = i;
15-18                break;
15-19            }
15-20        }
15-21        return position;
15-22    }
15-23
15-24    public static void main(String[] args) {
15-25        int x = 5;                      // zu suchender Wert
15-26        int[] a = {1,2,3,4,5,6,7,8,9}; // Feld mit Daten
15-27
15-28        // suchen
15-29        int position = suchen(a, x);
15-30
15-31        if(position >= 0) {
15-32            System.out.println("gefunden an Position " + position);
15-33        } else {
15-34            System.out.println("nicht gefunden");
15-35        }
15-36    }
15-37 }
```

Dieses Verfahren macht keine Annahmen darüber, welche Werte wie im Feld vorliegen. Es werden einfach alle Elemente des Feldes nach und nach mit dem gesuchten Wert verglichen. Man könnte genauso von hinten nach vorne die Feldelemente mit dem Suchwert vergleichen oder jede andere Besuchsreihenfolge nehmen, die garantiert, dass jedes Feldelement genau einmal besucht wird (Übungsaufgabe: überlegen Sie sich alternative Strategien).

Für eine Aufwandsabschätzung der sequentiellen Suche muss man zuerst die charakteristische Eingabegröße bestimmen. Im vorliegenden Fall des sequentiellen Suchens wird die Laufzeit des Algorithmus durch die Feldgröße, das heißt die Anzahl der Feldelemente maßgeblich beeinflusst. Nimmt die Anzahl der Elemente im Feld zu, wird auch die Laufzeit zunehmen. Für das sequentielle Suchen kommt man zu folgenden Komplexitätsangaben, wobei  $n$  die Anzahl der Feldelemente angeben soll:

1. Die *best case* Komplexität ist  $\mathcal{O}(1)$ , weil im besten Fall das gesuchte Element schon im ersten Feldelement gefunden wird, was mit einer konstanten Anzahl von Operationen möglich ist.
2. Die *worst case* Komplexität ist  $\mathcal{O}(n)$ , weil im schlimmsten Fall alle Feldelemente untersucht werden müssen und erst im letzten Feldelement der gesuchte Wert gefunden wird beziehungsweise erst dort erkannt wird, dass das Element nicht vorkommt.
3. Zur Bestimmung der *average case* Komplexität geht man davon aus, dass immer ein Wert gesucht wird, der auch im Feld vorkommt und dass der Suchwert bei einer Vielzahl von Suchoperationen gleichverteilt über die möglichen Werte ist. Für die möglichen  $n$  Fällen dividiert durch die Anzahl dieser Fälle ergibt sich damit als durchschnittliche Anzahl von Vergleichsoperationen  $\frac{1+2+\dots+n}{n} = \frac{n+1}{2}$  und damit als *average case* Komplexität  $\mathcal{O}(n)$ . Im Durchschnitt wird unter den obigen Annahmen der gesuchte Wert nach  $(n+1)/2$  Vergleichsoperationen gefunden.

### 15.2.2 Binäre Suche

Würde man ein gebundenes Telefonbuch mit dem oben beschriebenen sequentiellen Suchverfahren nach einem bestimmten Namen durchsuchen, müsste man erhebliche Zeit investieren. Üblicherweise kommt man aber bei der Telefonbuchsue wesentlich schneller zum Ziel. Woran liegt das? Man wendet dabei nämlich nicht eine sequentielle Suche an, das heißt nacheinander von vorne nach hinten Eintrag für Eintrag mit dem Suchnamen vergleichen, sondern man wendet stattdessen eine andere Methode an. Man schlägt (meist willkürlich) eine Seite im Telefonbuch auf. Falls der Name lexikografisch (nach Buchstaben sortiert) vor dem aktuellen Eintrag auf der Seite liegt, schlägt man nach vorne im Buch weiter, ansonsten nach hinten. Die Positionen, wo man rechts und links nicht weitersuchen muss, markiert man zum Beispiel durch Finger, die man an den Positionen in das Telefonbuch steckt. An jeder neuen aufgeschlagenen Seite wendet man den gleichen Algorithmus an und so weiter. Mit jedem Schritt werden die Seiten, die noch in Frage kommen weniger. Eine Voraussetzung für diesen Algorithmus ist allerdings, dass die Einträge im Telefonbuch sortiert sind.

Die umgangssprachliche Beschreibung der Telefonbuchsue soll jetzt als Suche eines Eintrags in einem Feld konkreter gefaßt werden. Die beschriebene Methode ist unter dem Namen **Binäres Suchen** bekannt. Angenommen, es soll in einem Feld  $a$  der Länge  $n$  nach einem Wert  $x$  gesucht werden. Bei diesem Verfahren wird vorausgesetzt, dass die Werte im Feld sortiert vorliegen, also:

$$a[i] \leq a[i+1] \quad \forall 0 \leq i < n-1$$

In diesem Feld  $a$  soll nach dem Wert  $x$  gesucht werden. Zu Beginn des Suchens betrachtet man eine linke Indexgrenze  $min = 0$  und eine rechte Indexgrenze  $max = n - 1$ , das heißt das gesamte Feld ist unser Suchraum. Man wählt einen Index  $m$  zwischen unterer und oberer Grenze, zum Beispiel die Mitte  $m = (min + max)/2$ . Falls jetzt an dieser Stelle  $a[m]$  das gesuchte Element  $x$  vorliegt, so sind wir sofort fertig. Falls jedoch  $x$  kleiner als  $a[m]$  ist, müssen wir links des mittleren Elements weitersuchen, denn rechts vom mittleren Element kann

(weil das Feld sortiert ist) der gesuchte Wert auf keinen Fall mehr vorkommen. Dazu werden die Grenzen des Suchraums entsprechend angepasst, indem man die rechte Grenze des Suchraums auf  $\max = m - 1$  setzen und damit die komplette rechte Hälfte des Suchraums im Folgenden ignorieren. Ansonsten, der Fall  $x$  ist größer als  $a[m]$ , macht man das Gleiche mit der unteren Grenze und setzt  $\min = m + 1$  und ignoriert bei der weiteren Suche damit die gesamte linke Hälfte des Suchraums. Mit den neuen Grenzen führt man das Verfahren wie eben beschrieben weiter.

Durch die sukzessive Verkleinerung des Suchintervalls können zwei Fälle auftreten (hier ohne Beweis):

1. Das gesuchte Element  $x$  ist im Feld enthalten. Dann gewährleistet das Verfahren, dass man irgendwann zu dem Punkt kommt, dass  $a[m]$  gleich dem gesuchten Element  $x$  ist und damit der Algorithmus abbricht.
2. Das gesuchte Element  $x$  ist nicht im Feld enthalten. Dann gilt zu einem Zeitpunkt:  $\min > \max$ . In diesem Fall muss man also ebenfalls abbrechen.

Das Programm in Listing 15.6 zeigt die Umsetzung dieser Idee.

Listing 15.6: Binäres Suchen in einem Feld.

```

15-1  /**
15-2   * Binaeres Suchen in einem Feld als Methode
15-3   */
15-4  public class BinaeresSuchen {
15-5
15-6      /**
15-7       * @param a sortiertes Feld mit Werten
15-8       * @param x gesuchter Wert
15-9       * @result Position(Index), an der Wert gefunden wurde. Ansonsten -1
15-10      */
15-11     public static int binaeresSuchen(int[] a, int x) {
15-12         int links = 0;           // linke Suchgrenze
15-13         int rechts = a.length-1; // rechte Suchgrenze
15-14         int mitte = (links + rechts) / 2; // mittleres Element
15-15
15-16         while((a[mitte] != x) && (links <= rechts)) {
15-17             //System.out.println("links=" + links + ", mitte=" + mitte + ", rechts=" +
15-18             //rechts);
15-19             if(x < a[mitte]) {
15-20                 rechts = mitte - 1;
15-21             } else {
15-22                 links = mitte + 1;
15-23             }
15-24             // bestimme neue Mitte
15-25             mitte = (links + rechts) / 2;
15-26         }
15-27
15-28         return (a[mitte] == x) ? mitte : -1;
15-29     }
15-30
15-31     public static void main(String[] args) {
15-32         int n = 1000000;          // Groesse des Feldes
15-33         int x;                  // zu suchender Wert
15-34         int[] a = new int[n]; // Feld mit Daten
15-35
15-36         // Feld mit sortierten Werten belegen
15-37         for(int i=0; i<a.length; i++) {
15-38             a[i] = i;
15-39         }
15-40
15-41         // alle Elemente einmal suchen
15-42         for(int i=0; i<a.length+1; i++) {
15-43             x = i;
15-44             int position = binaeresSuchen(a, x);
15-45             if(position >= 0) {
15-46                 System.out.println("gefunden an Position " + position);
15-47             } else {
15-48                 System.out.println("nicht gefunden");
15-49             }
15-50         }
15-51     }

```

**Beispiel 15.8:**

In Abbildung 15.4 ist ein Beispiel zum binären Suchen angegeben. Es soll der Wert 6 im Feld gesucht werden. Das Feld hat insgesamt 9 Feldelemente, die aufsteigend sortiert vorliegen. In jedem Schritt wird der Index  $m$  des mittleren Elements bestimmt und das mittlere Element  $a[m]$  mit dem Suchwert verglichen. Zu beachten ist, dass die Division zur Bestimmung von  $m$  eine ganzzahlige Division ohne Rest ist. Der Algorithmus bricht an der zuletzt angegeben Stelle ab, weil  $a[2]$  das gesuchte Element ist. Würde man  $suchwert = 8$  suchen, wäre ein weiterer Schritt (Intervallhalbierung) erforderlich.

Bezüglich der Aufwandsabschätzung der binären Suche kommt man zu folgenden Komplexitätsangaben, wobei  $n$  wiederum die Anzahl der Feldelemente angeben soll:

1. Die *best case* Komplexität ist  $\mathcal{O}(1)$ , weil im besten Fall das gesuchte Element schon im ersten untersuchten Feldelement  $a[(n-1)/2]$  gefunden wird.
2. Die *worst case* Komplexität ist  $\mathcal{O}(\log_2(n))$ . In jedem Schritt des Algorithmus, in dem das mittlere Element  $a[\text{mitte}]$  ungleich dem gesuchten Wert ist, wird das Suchintervall durch die Anpassung der Grenzen halbiert. Eine Halbierung kann man maximal  $\log_2(n)$  mal durchführen, bis man auf ein Intervall der Länge 1 stößt, was sich nicht weiter zerlegen lässt. In diesem Fall ist entweder das Element gefunden ( $a[\text{mitte}] == \mathbf{x}$ ) oder durch die dadurch bewirkte Veränderung von **rechts** beziehungsweise **links** würde in der nachfolgenden Iteration **links > rechts** sein und damit das Verfahren ebenfalls abbrechen.
3. Zur Bestimmung der *average case* Komplexität geht man wieder von der Annahme aus, dass eine Gleichverteilung der Daten vorliegt und dass das gesuchte Element im Feld vorhanden ist. Nach  $\lceil \log_2(n) \rceil$  Schritten weiß man definitiv, ob und wo der Suchwert im Feld vorliegt. Im Durchschnitt weiß man das bei einer Gleichverteilung der Daten aber schon einen Schritt vorher. Für die binäre Suche findet man also im Schnitt nach  $\lceil \log_2(n) \rceil - 1$  Schritten das gesuchte Element. Insgesamt ist also die *average case* Komplexität ebenfalls  $\mathcal{O}(\log_2 n)$ .

Die sequentielle Suche wurde eben konkret für Felder gezeigt, in denen ein bestimmter Wert gefunden werden sollte. Der wesentlich Aspekt des Verfahrens war dabei, dass alle Werte nach einer bestimmten Besuchsstrategie (nämlich in dem Fall sequentiell) auf Gleichheit mit dem Suchwert untersucht wurden. Man könnte – ohne den Ablauf des Verfahrens an sich ändern zu müssen – statt des Tests auf Gleichheit auch ein beliebiges Prädikat einsetzen, das angewandt auf das i-te Feldelement (und gegebenenfalls weiteren Werten) entweder **true** (Prädikat trifft auf den Wert zu) oder **false** (Prädikat trifft auf den Wert nicht zu) liefern müsste. Beispielsweise könnte man also diesen Verfahrensablauf mit einem geeigneten Prädikat auch dazu nutzen, einen Wert im Feld zu finden, der in einer Epsilon-Umgebung zu dem Suchwert ist, also einen ähnlichen (statt dem exakten) Wert finden.

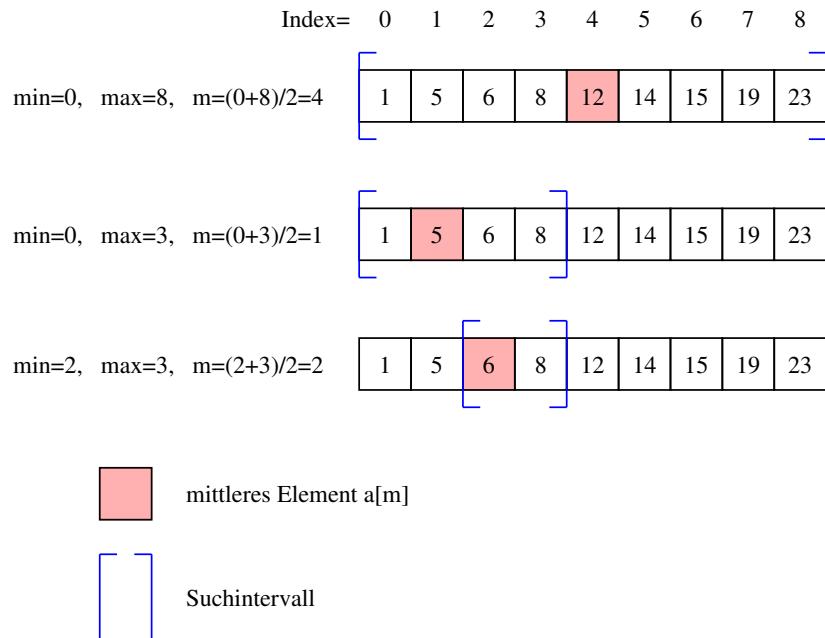
Wie vorher auch schon diskutiert wurde, ist die Besuchsreihenfolge aber eigentlich auch nicht wirklich wichtig für den Ablauf des Suchverfahrens, solange jedes Feldelement einmal besucht wird und kein Element doppelt besucht wird. Man könnte also statt eines Feldes einen beliebigen anderen *Container* für Werte nehmen, in dem die Werte organisiert sind, wenn über diesen Container eine Besuchsstrategie umsetzbar wäre. Beispiele für andere Container als Feld wären etwa Stack, Schlange, Binärbaum und Folge als uns bereits bekannte Strukturen aus Kapitel 10). In Kapitel 16.2 wird auf dieses Thema nochmals eingegangen.

Ein allgemeiner Algorithmus zum Suchen in einem Container zu einem beliebigen Prädikat unter Nutzung des Aufzählens von Elementen ist dann:

```

15-1   while (es sind noch unbesuchte Elemente im Container vorhanden) {
15-2       element = naechstes unbesuchtes Element im Container;
15-3       if (Praedikat(element)) {
15-4           return element;
15-5       }

```

Figure 15.4: Beispiel zum Binären Suchen für  $x = 6$ 

```

15-6   return Element nicht gefunden;
15-7 }
```

## 15.3 Einfache Sortierverfahren

Will man sehr oft im gleichen (großen) unveränderten Feld suchen, so bietet es sich aufgrund oben gemachter Komplexitätsaussagen an, das Feld vor den Suchoperationen zu sortieren. Es stellt sich dann natürlich die Frage, wie man ein Feld sortieren kann. Dazu gibt es eine Vielzahl auch sehr unterschiedlicher Verfahren, deren Entwicklung ebenso wie Untersuchung eine lange Tradition hat [Knu98].

An dieser Stelle sollen nur sehr einfache Verfahren vorgestellt werden, die eine *worst case* Komplexität von  $\mathcal{O}(n^2)$  haben und damit in der Praxis schon sehr zeitaufwändig werden können. Bessere aber auch komplexere Verfahren, die hier nicht vorgestellt werden, besitzen eine *worst case* Komplexität von  $\mathcal{O}(n \cdot \log(n))$ .

Die Verfahren sind alle *in-place* Verfahren, die keinen zusätzlichen (nennenswerten) Speicherbedarf haben. Zudem werden alle Daten im Hauptspeicher gehalten. Für sehr große Datenbestände wie etwa der Inhalt kompletter Datenbanken von vielen Tera- oder Petabyte Größe wären diese Verfahren damit nicht anwendbar.

### 15.3.1 Sortieren durch Auswählen

Ein einfaches Verfahren zum Sortieren eines Feldes ist das **Sortieren durch Auswählen**. Die Idee ist dabei, dass man das Feld in zwei Teile teilt, einem bereits sortierten ersten vorderen Teil und einem zweiten noch unsortierten hinteren Teil. Zu Beginn ist der sortierte Teil leer und das gesamte Feld stellt den unsortierten Teil dar. Man geht jetzt sukzessive für  $i = 0, \dots, n - 2$  vor, und versucht, in dem unsortierten Teil das verbliebene kleinste Element zu finden, etwa an der Stelle  $j$  ( $i \leq j < n - 1$ ). Hat man dieses kleinste Element bestimmt, so tauscht man es mit dem  $i$ -ten Element, wodurch der sortierte Teil um ein Element größer geworden ist und der unsortierte Teil ein Element weniger hat.

Listing 15.7: Sortieren durch Auswählen.

```

15-1  /*
15-2   * Sortieren durch Auswaehlen
15-3   */
15-4  public class SortierenAuswaehlen {
15-5
15-6      // Sortieren durch Auswaehlen
15-7      static void sortierenDurchAuswaehlen(int [] a) {
15-8          int tmp, min;
15-9
15-10         // besorge den Wert fuer die i-te Position
15-11         for( int i = 0; i < a.length -1; i++) {
15-12
15-13             // die i-te Position hat den bis jetzt kleinsten Wert
15-14             min = i;
15-15
15-16             // suche einen kleineren Wert im Restfeld
15-17             for(int j = i+1; j < a.length; j++) {
15-18                 if( a[j] < a[min] )
15-19                     // kleineren Wert gefunden, Position merken
15-20                     min = j;
15-21             }
15-22
15-23             // tausche a[i] und a[min]
15-24             tmp = a[i];
15-25             a[i] = a[min];
15-26             a[min] = tmp;
15-27         }
15-28     }
15-29
15-30     public static void main(String [] args) {
15-31         // Beispielaufruf
15-32         int [] feld = {1,3,5,7,9,2,4,6,8,10};
15-33         sortierenDurchAuswaehlen(feld);
15-34     }
15-35 }
```

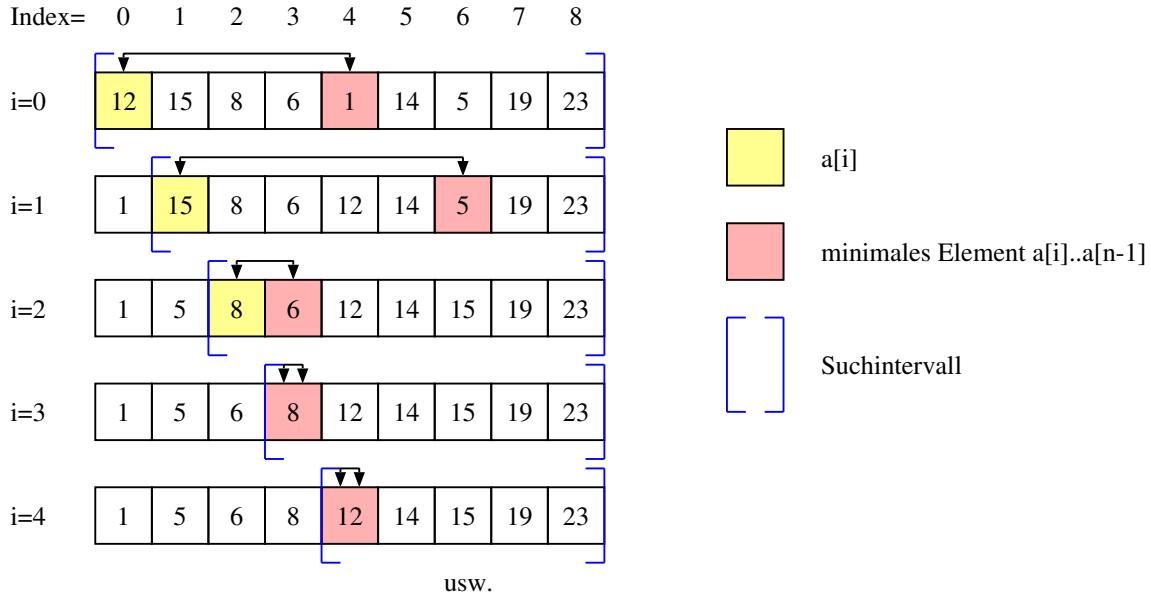


Figure 15.5: Beispiel zum Sortieren durch Auswählen

Das Java-Programm dazu ist in Listing 15.7 gegeben. Der Ablauf des Programms für ein Beispieldfeld ist in Abbildung 15.5 zu sehen. Die äußere Schleife ( $i$ -Schleife) legt das Suchintervall fest, ab wo das Minimum bis zum Ende des Feldes in jedem Schritt zu suchen ist. Die innere Schleife ( $j$ -Schleife) sucht dann innerhalb des Suchintervalls das kleinste Element, indem das  $j$ -te Element mit dem bis jetzt gefunden minimalen Element verglichen wird. Nach dem Ende der  $j$ -Schleife enthält die Variable `min` die Position des minimalen Elements im unsortierten Teil. Über den Puffer `tmp` werden dann die beiden Werte `a[i]` und `a[min]` vertauscht. Zu beachten ist, dass auch für  $i = min$  der Tausch korrekt ist.

Zu diesem Sortierverfahren lassen sich Angaben zur Komplexität machen. Die Größe des sei  $n$  Elemente. Der Schleifenrumpf der äußeren  $i$ -Schleife wird  $n - 1$  mal ausgeführt. In jede dieser Iterationen finden insgesamt  $c_1 = 4$  Zuweisungen statt und die innere  $j$ -Schleife. In dieser inneren  $j$ -Schleife werden  $m = n - i - 1$  Iterationen in Abhängigkeit von  $i$  ausgeführt. Innerhalb der  $j$ -Schleife finden höchstens  $m \cdot c_2$  Operationen statt. Für die Gesamtzahl der ausgeführten Operationen  $o$  ergibt sich damit als Obergrenze:

$$\begin{aligned}
 o &\leq (n-0-1) \cdot c_2 + (n-1-1) \cdot c_2 + \dots + (n-(n-3)-1) \cdot c_2 + (n-(n-2)-1) \cdot c_2 + (n-1) \cdot c_1 \\
 &= (n-1) \cdot c_2 + (n-2) \cdot c_2 + \dots + 2 \cdot c_2 + 1 \cdot c_2 + (n-1) \cdot c_1 \\
 &= (n-1) \cdot c_1 + \sum_{k=1}^{n-1} k \cdot c_2 \\
 &= (n-1) \cdot c_1 + c_2 \cdot \sum_{k=1}^{n-1} k \\
 &= (n-1) \cdot c_1 + c_2 \cdot \frac{n \cdot (n-1)}{2} \\
 &= (n-1) \cdot c_1 + c_2 \cdot \frac{n^2 - n}{2} \\
 &= \mathcal{O}(n^2)
 \end{aligned}$$

Das heißt, der Aufwand dieses Verfahrens wächst (höchstens) quadratisch mit der Feldgröße. Schaut man sich

das Verfahren an, so ist der *best case* Aufwand aber gleichfalls  $\mathcal{O}(n^2)$ , also im Wesentlichen unabhängig vom Inhalt des Feldes.

Das Sortieren durch Auswählen ist ein sogenanntes *stabiles Verfahren*. Ein Sortierverfahren heißt stabil, wenn bei gleichen Werten im zu sortierenden Feld die Reihenfolge dieser Werte untereinander erhalten bleibt. Dies ist zum Beispiel dann interessant (und erwünscht), wenn ein Datensatz nacheinander nach mehreren Kriterien sortiert werden soll, wie dies in Datenbanken häufig auftritt.

### 15.3.2 Sortieren durch Einfügen

Das zweite Verfahren, das betrachtet werden soll, heißt **Sortieren durch Einfügen**. Die Idee dabei ist wiederum, dass man das Feld in zwei Teile teilt, einen bereits sortierten ersten vorderen Teil und einen zweiten noch unsortierten hinteren Teil. Zu Beginn besteht der vordere sortierte Teil aus dem ersten Feldelement, dieses Teilstück der Länge 1 ist per Definition sortiert. Man geht jetzt sukzessive vor und nimmt sich das nächste Element des unsortierten Teils und fügt dies an die korrekte Stelle im sortierten Teil ein.

Der Java-Code ist in Listing 15.8 angegeben, ein Beispiel zum Ablauf ist in Abbildung 15.6 zu sehen.

Die *worst case* Komplexität des Verfahrens kann analog zum Sortieren durch Auswählen angegeben werden. Die äußere i-Schleife wird bei einer Feldlänge von  $n$  insgesamt  $n - 1$  mal durchlaufen. Die innere j-Schleife wird maximal  $m = n - 1 - i$  mal in Abhängigkeit von  $i$  ausgeführt. Dies ist dann der Fall, wenn das einzufügende Element kleiner als alle bereits eingesortierten Elemente ist, also an die erste Feldposition gehört. Analog zum ersten Sortierverfahren ergibt sich auch hier ein *worst case* Aufwand von  $\mathcal{O}(n^2)$ . Dieser schlimmste Fall tritt dann ein, wenn das einzusortierende Element jeweils ein kleineres Element als die die Elemente des sortierten Teils ist. Dies ist dann der Fall, wenn das Feld zu Beginn umgekehrt, also absteigend sortiert vorliegt.

Bei diesem Verfahren ist aber interessant, was passiert, wenn das Feld bereits sortiert oder fast sortiert vorliegt. Wenn nämlich in der inneren j-Schleife schon beim ersten Test `a[j] > wert` nicht wahr ist, bricht die innere Schleife sofort ab. Tritt dieser Fall immer ein, was der Fall ist, wenn das Feld schon zu Beginn sortiert vorliegt, ergibt sich insgesamt für das Verfahren eine *best case* Komplexität von  $\mathcal{O}(n)$ . Der Aufwand für das Sortierverfahren durch Einfügen ist also abhängig von den Daten, die vorliegen.

Listing 15.8: Sortieren durch Einfügen.

```

15-1  /**
15-2   * Sortieren durch Einfuegen
15-3   */
15-4  public class SortierenEinfuegen {
15-5
15-6      static void sortierenDurchEinfuegen(int[] a) {
15-7          // betrachte alle einzufuegenden Elemente ab der 2-ten Position
15-8          for (int i=1; i<a.length; i++) {
15-9
15-10             // Wert merken
15-11             int wert = a[i];
15-12
15-13             // verschiebe alle vorhergehenden Elemente, die groesser sind als a[i]
15-14             int j = i-1;
15-15             while((j >= 0) && (a[j] > wert)) {
15-16                 a[j+1] = a[j];
15-17                 j--;
15-18             }
15-19
15-20             // Einfuegeposition gefunden
15-21             a[j+1] = wert;
15-22         }
15-23     }
15-24
15-25     public static void main(String[] args) {
15-26         // Beispielaufruf
15-27         int[] feld = {1,3,5,7,9,2,4,6,8,10};
15-28         sortierenDurchEinfuegen(feld);
15-29     }
15-30 }
```



Figure 15.6: Beispiel zum Sortieren durch Einfügen

### 15.3.3 Bubble Sort

Das dritte und letzte hier vorgestellte Verfahren – Bubble Sort oder weniger geläufig der eingedeutschte Name **Sortieren durch Aufsteigen** – basiert auf der Idee, dass man falsch positionierte Werte durch Verschieben mit dem direkten Nachbarn näher an die Zielposition bringt, wo dieser Wert hingehört. Man beginnt an der ersten Feldposition und vergleicht das erste Feldelement mit dem zweiten. Sind diese beide Werte in der falschen Ordnung zueinander, so werden sie vertauscht, ansonsten belassen. Danach betrachtet man das zweite und dritte Element und vertauscht diese bei Bedarf, dann das dritte und vierte und so weiter. Nach diesem ersten Durchlauf steht das größte Element am rechten Rand des Feldes und damit an der richtigen Position (Überlegen Sie sich, wieso das so ist). Dann fängt man wieder von vorne an und vergleicht das erste mit dem zweiten, dann das zweite mit dem dritten und so weiter. Diesmal aber nur bis zur vorletzten Position, weil ja an der letzten Position das richtige Element angelegt wurde. Nach diesem zweiten Durchlaufen steht zusätzlich das zweitgrößte Element an der zweitletzten, richtigen Position. Diese Iterationen von vorne nach hinten wiederholt man solange, bis man alle bis auf die erste Position mit dem richtigen Wert belegt hat. Da nur ein Wert übrig bleibt, muss dies der kleinste sein und steht zudem an der richtigen ersten Position. Die Implementierung in Java wird dem Leser als Übungsaufgabe überlassen. Ebenso die Überlegung, ob man das Verfahren nicht optimieren kann, indem man Fälle erkennt, dass man schon früher aufhören kann.

### 15.3.4 Sortieren mit großen Datenelementen

Genauso, wie man ein Feld ganzzahliger Werte anlegen kann, ist es auch möglich, ein Feld von Fließkommawerten, ein Feld von Bankkonten oder ein Feld von Videodaten anzulegen. An dieser Stelle soll uns nicht interessieren, wie zum Beispiel die Definition eines Feldes von Videodaten genau aussieht. Aber sind die einzelnen Videodaten sehr umfangreich und möchte man das Feld nach einem Schlüssel sortieren (zum Beispiel Erstellungsdatum), so würden beispielsweise mit Sortierverfahren durch Auswählen (aber auch andere Verfahren) sehr große Datenmengen bewegt, weil der Austausch beispielsweise von `feld[i]` mit `feld[min]` jeweils das Kopieren zweier kompletter Videodatensätze über die Variable `tmp` – die dann einen anderen Typ als `int` haben müßte – bedeuten würde:

```

15-1   tmp = feld[ i ];
15-2   feld[ i ] = feld[ min ];
15-3   feld[ min ] = tmp;

```

Wie man leicht einsieht, würden dadurch sehr großen Datenmengen bewegt werden müssen. Einen Ausweg aus dieser Problematik bietet die Nutzung eines **Indexfeldes** an, über das die Feldelemente **indirekt** angesprochen werden können. Ein Indexfeld ist ein Feld von Indizes. Die Größe des Indexfeldes entspricht der Größe des eigentlichen Feldes, der Basistyp ist ein ganzzahliger Typ, zum Beispiel `int`. Um den Sinn eines Indexfeldes zu verstehen, schaut man sich zunächst die Definition einer Permutation an.

#### Definition 15.2 (Permutation):

Unter einer **Permutation** von  $n$  Elementen versteht man eine bijektive Abbildung einer Menge  $\{x_1, \dots, x_n\}$  auf sich. Man notiert die Permutation als

$$\begin{pmatrix} x_1 & \dots & x_n \\ x_{i_1} & \dots & x_{i_n} \end{pmatrix}$$



**Beispiel 15.9:**

Die möglichen Permutationen der Menge  $M = \{1, 2, 3\}$  sind:

$$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix} \quad \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix} \quad \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix} \quad \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

Für eine Menge mit  $n$  Elementen gibt es  $n!$  verschiedene Permutationen.

Die Permutation  $\begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix}$  bezeichnet man aus naheliegenden Gründen als Identitätspermutation. ♦

Beginnend mit einer Identitätspermutation in einem Indexfeld `ix`, benutzen wird das Indexfeld nun *überall* dort, wo ein Feldelement indiziert werden muss. Statt `feld[i]` notiert man stattdessen `feld[ix[i]]`. Das Indexfeld `ix` stellt zu jeder Zeit eine Permutation der Indizes. Startend mit der Identitätspermutation für die Zahlen  $0, \dots, n - 1$ , also `ix[i]=i`, bedeutet ein Feldzugriff der Form `feld[ix[i]]` mit dieser Identitätspermutation in `ix` nichts anderes als `feld[i]`, da ja `ix[i]=i` gilt.

Die Problematik beim Sortieren großer Einzeldaten ist ja das Bewegen dieser Daten, zum Beispiel beim Tausch der Elemente `feld[i]` und `feld[min]`. Im Sortieralgorithmus kann man nun bei einem tausch stattdessen das Indexfeld nutzen, indem man statt des Tauschs der Feldelemente `feld[i]` mit `feld[min]` stattdessen Indizes `ix[i]` und `ix[min]` austauscht, also permutiert, was erheblich weniger Kopieraufwand bedeuten kann. Durch das Indexfeld werden bei einem solchen Austausch insgesamt drei int-Werte umkopiert. Wenn stattdessen 3 Videodatenbestände verschoben werden müssten, wäre dies ganz erheblicher Mehraufwand.

Das modifizierte Programm ist in Listing 15.9 angegeben. Zu beachten ist die *durchgehende Verwendung* des Indexfeldes bei der Indizierung des Datenfeldes. Eine indirekte Indizierung kann an vielen anderen Stellen sinnvoll eingesetzt werden, wie zum Beispiel zur Abspeicherung dünnbesetzter Datenstrukturen, in denen nur wenige (Feld-)Elemente Daten enthalten, die restlichen (Feld-)Elemente sind nicht mit sinnvollen Werten belegt und ungenutzt.

Listing 15.9: Indirektes Sortieren.

```

15-1  /**
15-2   * Sortieren durch Auswaehlen mit Indirektemfeld
15-3   */
15-4  public class SortierenIndirekt {
15-5
15-6      // Sortieren durch Auswaehlen mit Indexfeld
15-7      static void sortierenDurchAuswaehlenIndirekt(int[] a, int[] ix) {
15-8          int tmp, min;
15-9
15-10         // besorge den Wert fuer die i-te Position
15-11         for( int i = 0; i < a.length; i++) {
15-12
15-13             // die i-te Position hat den bis jetzt kleinsten Wert
15-14             min = i;
15-15
15-16             // suche einen kleineren Wert im Restfeld
15-17             for(int j = i+1; j < a.length; j++) {
15-18                 if( a[ix[j]] < a[ix[min]] )
15-19                     // kleineren Wert gefunden, Position merken
15-20                     min = j;
15-21             }
15-22
15-23             // frueher: tausche a[i] und a[min]
15-24             // jetzt: tausche die Indizes und nicht die Daten
15-25             tmp = ix[i];
15-26             ix[i] = ix[min];
15-27             ix[min] = tmp;
15-28         }
15-29     }
15-30
15-31     public static void main(String[] args) {
15-32         // Beispieldaufruf
15-33         int[] feld = {1,3,5,7,9,2,4,6,8,10};
15-34         int[] ix = {0,1,2,3,4,5,6,7,8,9};
15-35         sortierenDurchAuswaehlenIndirekt(feld, ix);
15-36
15-37         // Ausgabe des sortierten Feldes
15-38         for(int i=0; i < feld.length; i++) {
15-39             System.out.println(feld[ix[i]]);
15-40         }
15-41     }
15-42 }
```

## 15.4 Zusammenfassung und Hinweise

### Literaturhinweise

Ein Standardwerk zur Literaturverfahren mit tiefergehenden Laufzeitanalysen ist [Knu98].

### Verstehen

Den Unterschied und den Zweck einer Komplexitätsbetrachtung im Gegensatz zu einer konkreten Zeitmessung sollte verstanden sein.

### Kurz und knapp merken

- Über Komplexitätsangaben möchte man zu einem Algorithmus allgemein gültige Aussagen zum Ressourcenverbrauch machen.
- Dazu findet man Parameter, mit denen sich der Ressourcenverbrauch als Funktion angeben / abschätzen lässt.
- Um allgemeine Aussagen zur Komplexität machen zu können, verwendet man die  $O$ ,  $\Omega$  und  $\Theta$  Notation.
- Mit Hilfe einer solchen Notation beschreibt man eine ganze Klasse sich ähnlich verhaltender Funktionen.
- Es existieren eine Reihe von Rechenregeln, auch für den praktischen Gebrauch.
- Zum Suchen eines Wertes in einem unsortierten Feld ist das Lineare Suchen eine Strategie, die alle Elemente des Feldes im schlimmsten Fall ein mal überprüft. Die Komplexität ist  $\mathcal{O}(n)$ .
- Für ein sortiertes Feld kann man durch eine binäre Suchstrategie die Suche mit  $\mathcal{O}(\log_2(n))$  wesentlich beschleunigen.
- Einfache Sortierverfahren sind das Sortieren durch Auswählen, Sortieren durch Einfügen und Bubblesort, die alle einen Berechnungsaufwand von  $\mathcal{O}(n^2)$  haben.

### Reflektion des Stoffs

- Was ist der charakteristische Parameter, wenn man das Alter der ältesten Person im Raum ermitteln will?  
Was wäre der oder die Parameter, wenn man die Person bestimmen will, die dem Durchschnittsalter am nächsten kommt?
- Welche Rechenregel(n) hilft/helfen bei folgendem Beispielauszug aus einem Programm:

```
15-1 s = 0;
15-2 for (int i=0; i<a.length; i++) for (int j=0; j<a.length; j++) s+=a[i]*a[j];
15-3 for (int k=0; k<a.length; k++) a[i] /= (k+1);
```

- Kann man die Lineare Suche auch auf ein sortiertes Feld anwenden? Die Binäre Suche auf ein unsortiertes Feld? Wenn Sie eine Frage mit nein beantworten: an welchen Stellen im Algorithmus ist der entscheidende Punkt dafür?
- Vergleichen Sie das Best Case Verhalten der vorgestellten Sortieralgorithmen.
- Angenommen, Sie dürfen sich den Feldinhalt vorher immer anschauen. Welches Sortierverfahren würden Sie dann für welche Art von Eingabedaten (Inhalt) verwenden?

- Das Suchen in einem sortierten Feld geht ja wesentlich schneller, als in einem unsortierten Feld. Wäre es dann immer eine gute Strategie, zuerst ein Feld zu sortieren und dann (schnell) zu suchen? Wieso?



## Chapter 16

# Allgemeine Strategien in Algorithmen

Viele Programme in der Praxis dienen oft nur der mehr oder weniger intelligenten Speicherung und Aufbereitung von Daten. Dazu zählen etwa die bekannten Funktionen von sozialen Netzwerken wie **Facebook** oder **Twitter**, in die einige Nutzer relativ simple Daten (Texte, Bilder, Videos) hineinstellen und andere Nutzer, die sich dafür registriert haben, diese Informationen etwas hübsch aufbereitet aber im Wesentlichen unverändert angezeigt bekommen. Die Programme für diese Netzwerke brauchen also keine "Intelligenz", da sie mehr oder weniger bei auslösenden Aktionen des Einstellers nur Bytes hin- und herbewegen (die Entwickler dieser Dienste mögen mir dafür verzeihen) und dem Leser als Webseite zusammenfasst darstellen. (Außen vorgelassen ist hierbei die interne Verarbeitung bei diesen Diensten, die sehr anspruchsvoll sind und zum Beispiel ermittelt, welche Werbung man einer Person am besten zeigt.)

Bei anderen Problemstellungen führt eine reine Verschiebung von Daten aber nicht mehr zum Ziel. Die notwendigen Lösungsalgorithmen müssen "intelligent" vorgehen, indem sie etwa aus einer unübersehbaren Anzahl an potentiellen Lösungen eine gute oder sogar die optimale Lösung auswählen. Ein Beispiel ist etwa ein Routenplaner, der zu einem bekannten Wegenetz, zum Beispiel dem Westeuropas, und einem gegebenen Start- und Zielort die kürzeste oder schnellste Verbindung herausfinden soll.

Schaut man sich eine Vielzahl von Algorithmen an, die eine (möglichst) optimale Lösung zu einer bestimmten Problemstellung finden sollen, so wird man oft gewisse Gemeinsamkeiten bei den Lösungsansätzen feststellen, die sich auf die Herangehensweise zur Lösung beziehen. Im Folgenden sollen einige einfache solcher Herangehensweisen vorgestellt werden, wie sie insbesondere in Problemklassen angewandt können, in denen eine optimale Lösung aus einer Vielzahl möglicher Lösungen gefragt ist (Optimierungsproblem). Das Ziel ist dabei weniger, einen umfassenden Überblick über Optimierungsverfahren zu geben (was definitiv hier nicht geschieht), sondern vielmehr die Denkweise zu unterschiedlichen allgemeinen Lösungsstrategien zu vermitteln.

### Beispiel 16.1:

Gegeben sind 4 voneinander unabhängige Arbeitsschritte, die 4,5,6 und 9 Minuten dauern. Wie sollen diese Arbeitsschritte auf zwei Arbeiter aufgeteilt werden, so dass die Gesamtbearbeitungszeit minimal wird.

Der **Lösungsraum** ist die Menge aller möglichen Lösungen. Im Beispiel sind das alle möglichen Aufteilungen der 4 Arbeitsschritte auf die 2 Arbeiter. Eine mögliche Strategie könnte es jetzt sein, alle möglichen Lösungen im Lösungsraum aufzuzählen, jeweils zu bewerten und daraus die Lösung auszuwählen, die die beste Bewertung hat.

Beispieldateiung:

Arbeiter	zugewiesene Arbeitsschritte	Bearbeitungszeit
1	4,5	$4+5=9$
2	6,9	$6+9=15$

Bei dieser Aufteilung würde also die Gesamtbearbeitungszeit 15 Zeiteinheiten betragen, weil nach diesen 15 Zeiteinheiten der letzte Bearbeitungsschritt beendet wäre. Geht es besser? ♦

## 16.1 Einführendes Beispiel

In diesem einführenden Beispiel zur Gesamtthematik des Algorithmenentwurfs soll die oft enge Verflechtung von Operationen mit den Daten beziehungsweise Datenstrukturen motiviert werden. In Kapitel 5.6 wurde bereits im Zusammenhang mit dem vorzeitigen Schleifenabbruch ein Beispiel gebracht, das alle Primzahlen bis zu einer vorgegebenen Obergrenze  $n$  berechnet. Zur Erinnerung: Eine Primzahl ist eine natürliche Zahl  $i \geq 2$ , die nur durch die Zahl 1 und sich selber teilbar ist.

Eine Variante des früher gezeigten Ansatzes, die aber im Wesentlichen auf der gleichen Idee beruht, ist das Verfahren **Sieb des Eratosthenes**, benannt nach dem griechischen Gelehrten Eratosthenes von Kyrene (ca. 270–194 v.Chr). Dabei besitzt das Sieb, ein Behälter für Zahlen, zu Beginn alle Zahlen von 2 bis zur vorgegebenen Obergrenze  $n$ , die alle unmarkiert sind. Das Verfahren geht nun wiederholt so vor, dass die kleinste im Sieb verbliebene Zahl, die noch nicht markiert ist, eine Primzahl ist. Findet man keine solche Zahl mehr, ist das Verfahren abgeschlossen und das Sieb enthält alle Primzahlen zwischen 2 und  $n$ . Findet man aber noch eine solche Zahl, die dann eine Primzahl ist, so markiert man diese Zahl und entfernt alle Vielfachen dieser Zahl aus dem Sieb. Als Hinweis sei angemerkt, dass bei der Suche nach der kleinsten verbliebenen unmarkierten Zahl im Sieb nur eine größere Zahl in Frage kommt, als beim letzten Suchen gefunden wurde.

### Beispiel 16.2:

Zu berechnen sind alle Primzahlen bis 10. Dazu legt man ein Sieb an, hier als Menge angegeben:  $\{2, 3, 4, 5, 6, 7, 8, 9, 10\}$ . Die kleinste Zahl im Sieb ist 2, also ist 2 eine Primzahl, wird markiert und alle Vielfachen von 2 aus dem Sieb gestrichen. Das Ergebnis ist das Sieb  $\{2, 3, 5, 7, 9\}$ . Die kleinste nicht markierte Zahl ist 3, also ist 3 eine Primzahl, wird markiert und alle Vielfachen von 3 werden aus dem Sieb entfernt:  $\{2, 3, 5, 7\}$ . Dies wiederholt sich für 5 und 7, sodass das Endergebnis  $\{2, 3, 5, 7\}$  ist, weil keine unmarkierte Zahl im Sieb verblieben ist. ♦

Um den umgangssprachlich beschriebenen Algorithmus in einem Java-Programm zu formulieren, muss man sich Gedanken machen über:

- Daten: wie werden die vorkommenden Daten und Datenstrukturen geeignet abgebildet?
- Kontrollfluss: wie soll die Abfolge der Arbeitsritte im Programm formuliert werden?

Begonnen werden soll mit Überlegungen zu den Daten. Primzahlen sind natürliche Zahlen, also sollte zur Darstellung der möglichen vorkommenden Zahlen zweckmäßigerweise ein ganzzahliger Typ genommen werden. Je nachdem, wie groß die Obergrenze  $n$  gewählt wird / werden kann, kommen `int` oder `long` in Frage. Die kleineren ganzzahligen Typen `byte` und `short` eher weniger (wieso?). `int` hat als Obergrenze  $2^{31} - 1$ , kann also den Bereich bis ca. 2 Milliarden abdecken, `long` auch darüber hinaus. Bevor man sich bequem den größeren Bereich mit `long` auswählt, sollte man aber auch darauf achten, wie das Sieb dargestellt werden soll; diese Diskussion `int`/`long` wird später wieder aufgenommen.

Die nächste Frage ist die nach der Umsetzung der Sieb-Idee als Container für die vorkommenden Zahlen. Dazu muss man sich überlegen, welche Operationen mit dem Sieb vollzogen werden müssen. Die Operationen sind:

- Finde die kleinste unmarkierte Zahl im Sieb. Hier nochmals der Hinweis, dass bei der Suche nach der

N=10  
betrachtete Zahl: 3  
restliches Sieb: [3,5,7,9]

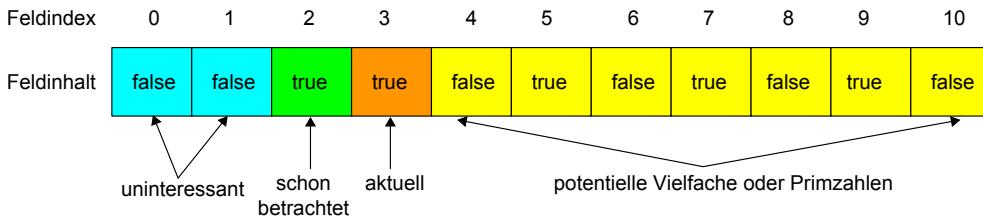


Figure 16.1: Darstellung des Siebes durch ein Feld von Wahrheitswerten.

kleinsten verbliebenen unmarkierten Zahl im Sieb nur eine größere Zahl in Frage kommt, als beim letzten Suchen gefunden wurde.

- Markiere die Zahl  $i$ .
- Entferne die Zahl  $i$  aus dem Sieb.

Mit den bis jetzt bekannten Möglichkeiten in Java würde sich aufgrund der benötigten Operationen ein Feld anbieten, das für jede Zahl entsprechende Information vorhält (Zahl im Sieb vorhanden / nicht vorhanden, Zahl markiert / nicht markiert), womit man über die Zahl  $i$  auf den  $i$ -ten Feldeintrag direkt zugreifen kann und wo man ausgehend von einer Position die nächste vorhandene nicht markierte Zahl ermitteln kann. Bei diesem Ansatz und mit den vorher gemachten Überlegungen zum Suchen der kleinsten unmarkierten Zahl kommt man zur Erkenntnis, dass eine Markieroperationen nicht mehr nötig ist, da nur Zahlen größer als die letzte gefundene Primzahl in Betracht kommen. Bleibt als Information zu einer Zahl eigentlich nur noch, ob diese Zahl im Sieb noch vorhanden ist oder nicht, wofür als Basistyp des Feldes `boolean` hinreichend ist (siehe Abbildung 16.1). Die Größe des Feldes müsste so bemessen sein, dass Informationen zu den Zahlen  $2, \dots, n$  aufgenommen werden könnte. Feldindizes beginnen in Java bei 0. Man würde auch gerne der Übersichtlichkeit halber auf die Information zur  $i$ -ten Zahl mit dem Index  $i$  wie `sieb[i]` zugreifen wollen. Aus diesen Gründen bietet es sich an, dass man für das Feld/Sieb ein klein bisschen mehr Speicher speniert, dafür aber dann einfacher damit arbeiten kann. Für eine Maximalzahl  $n$  definiert man deshalb das Sieb als ein Feld von Wahrheitswerten der Länge  $n + 1$ . Konkret: `boolean[] sieb = new boolean[n+1];` Die oben angesprochenen Operationen könnten dann wie folgt umgesetzt werden:

- Finde die kleinste unmarkierte Zahl im Sieb: suche ab dem letzten gefundenen Primzahlindex den nächsten Eintrag  $j$ , für den `sieb[j] == true` gilt.
- Markiere eine Zahl: diese Operation ist nicht mehr nötig.
- Entferne eine Zahl aus dem Sieb: `sieb[zahl] = false`.

Eben wurde die Diskussion `int` beziehungsweise `long` für Zahlen und Primzahlen geführt. Dies betrifft konkret im unten abgebildeten Programm zu diesem Verfahren die Schleifenvariablen. Würde man sich gegen `int` und für `long` entscheiden, muss man sich darüber im Klaren sein, dass ein Feld von `boolean` mit 2 Milliarden Elementen, was man mit `int` noch erreichen könnte, schon ca. 2 Gigabyte Hauptspeicher belegen würde. Wenn man also `long` in Betracht zieht, um damit sehr große Primzahlmengen zu erzeugen, so wird das wesentliche Problem der Hauptspeicherbedarf dieses Verfahrens sein.

Der Ablauf des Programms ergibt sich nach den eben gemachten Vorüberlegungen zur Umsetzung der Daten dann eigentlich direkt aus dem vorgestellten Gedanken zum Siebverfahren. Das Sieb muss angelegt und so

Listing 16.1: Sieb des Eratosthenes als Java-Programm.

```

16-1  /**
16-2   * Verfahren des Eratosthenes zur Primzahlerzeugung
16-3   */
16-4  public class Eratosthenes
16-5  {
16-6      public static void main(String [] args) {
16-7          // bis zu welcher Zahl wollen wir die Primzahlen ausrechnen?
16-8          int maximalzahl = 100;
16-9
16-10         // Das Sieb (Achtung: Indizes laufen von 0..n-1 in Java)
16-11         boolean sieb [] = new boolean[maximalzahl+1];
16-12
16-13         // Setze alle relevanten Werte im Sieb auf true (Zahl im Sieb vorhanden)
16-14         for( int i = 2; i < maximalzahl+1; i++)
16-15             sieb[i] = true;
16-16
16-17         // Fuehre den folgenden Test fuer alle Zahlen ab 2 durch
16-18         for(int startzahl = 2; startzahl <= maximalzahl; startzahl++) {
16-19             if(sieb[startzahl]) {
16-20                 // Primzahl gefunden: Gebe zum Beispiel Information aus
16-21                 System.out.println(startzahl + " ist eine Primzahl");
16-22
16-23                 // Streiche alle Vielfachen im Sieb
16-24                 for(int vielfaches = 2*startzahl; vielfaches <=maximalzahl; vielfaches+=
16-25                     startzahl)
16-26                     sieb[vielfaches] = false;
16-27                 }
16-28             }
16-29         }

```

initialisiert werden, das alle Zahlen von 2 bis  $n$  im Sieb vorhanden sind. Danach muss in einer zweiten Phase jeweils das nächste Element im Sieb gesucht werden und alle Vielfachen davon aus dem Sieb entfernt werden. Das gesamte Programm sieht dann nach diesen Überlegungen wie in Listing 16.1 gezeigt aus.

## 16.2 Besuchsstrategien

Einfache Algorithmen bauen teilweise auf dem Grundsatz auf, dass mit jedem Element aus einer Datensammlung eine bestimmte Aktion ausgeführt werden soll oder dass basierend auf den Einzelinformationen in den Elementen eine Gesamtinformation produziert werden soll, zu der jedes Einzelement einen Beitrag leistet.

### Beispiel 16.3:

Ein praktisches Beispiel ist die Bestimmung der Maximaltemperatur eines Tages, wenn 24 Einzelmesswerte zur vollen Stunde vorliegen. Die Diskussion, ob zwischen den 24 gemessenen diskreten Werten zur vollen Stunde eventuell höhere Werte hätten dazwischen auftreten können, lässt man außen vor. Abstrakt gesehen muss man also aus einer Menge von  $n$  Werten ( $x_1, \dots, x_n$ ) das Maximum dieser Werte berechnen.

Eine weitere praktische Aufgabenstellung ist es, aus den 24 Temperaturmesswerten die Durchschnittstemperatur des Tages zu ermitteln. Auch hier muss abstrakt aus einer Menge von  $n$

Werten  $(x_1, \dots, x_n)$  das arithmetische Mittel  $\frac{1}{n} \sum_{i=1}^n x_i$  dieser Werte berechnen werden, also die Summe aller Einzelwerte bilden und das resultierende Ergebnis durch die Anzahl  $n$  der Zahlen dividieren.

Es sei angemerkt, dass in beiden Beispielfällen der gesamte Datenbestand, hier 24 Zahlen, durchlaufen werden muss. Weiterhin sei auch angemerkt, dass es für diese beiden Anwendungsbeispiele sogar unerheblich ist, in welcher Besuchsreihenfolge dies geschieht. Genauso wie streng sequentiell von  $0, \dots, n-1$  hätte man in den beiden Beispielen auch von  $n-1, \dots, 0$  rückwärts vorgehen können oder eine beliebige andere Strategie, die jedes Feldelement genau einmal besucht.

Eine beliebige Besuchsreihenfolge aller Datenelemente sich aussuchen zu können ist aber nicht immer möglich. Vielmehr würde es bei unterschiedlichen Besuchsreihenfolgen der Datenelemente auch zu unterschiedlichen Ergebnissen kommen. Ein Beispiel dazu ist die Operation, die zu einem Feldelement die Summe aller vorhergehenden Elemente addiert. ♦

Abstrakt gesehen wird also eine Strategie benötigt, die genau einmal zu jedem Datenelement hinführt, um eine geeignete Operation mit diesem Datenelement durchzuführen (Beispiel: Vergleich des Temperaturwertes des aktuellen Elements mit dem bis jetzt gefundenen Maximalwert oder addiere den aktuellen Wert zu einem Summenwert auf). In Kapitel 15.2.2 wurde bereits auf eine solche Verallgemeinerung im Zusammenhang mit Durchlaufstrategien in Containern hingewiesen. Hier etwas abgeändert das Schema. `aktion` sei eine Funktion, die auf alle Elemente nacheinander angewandt werden soll. Dies ist in Java so wie angegeben nicht möglich.

```
16-1  public static void durchlaufen(Container container, Aktion aktion) {
16-2      while (es sind noch unbesuchte Elemente im container vorhanden) {
16-3          element = naechstes unbesuchtes Element(container);
16-4          aktion(element);    // fuehre Aktion auf Element aus
16-5      }
16-6  }
```

Weniger abstrakt und mehr im Detail ist jetzt allerdings zu unterscheiden, in welcher Form eine solche Durchlaufoperation organisiert wird. In der folgenden Betrachtung soll als Basisdatentyp eines Containers konkret `int` genommen werden, andere auch komplexe Datentypen sind analog zu behandeln.

Für ein Feld als Container wäre diese allgemeine Strategie wie folgt umzusetzen:

```
16-1  public static void durchlaufenFeld(int[] a, Aktion aktion) {
16-2      for(int element : a) {
16-3          aktion(element);    // fuehre Aktion auf Element aus
16-4      }
16-5  }
```

Das Zählschleifenkonstrukt erhält für genau diese Fälle die angegebene syntaktische Variante, die auch bereits im Zusammenhang mit Felder eingeführt wurde.

Auch für andere Strukturen lassen sich Besuchsalgorithmen angeben. Beispielhaft soll dies an einer weiteren Datenstruktur gezeigt werden, dem in Kapitel 10.3 eingeführten Binärbaum. Dort wurden bereits drei verschiedene Besuchsstrategien eingeführt: `preorder`, `inorder` und `postorder`. Mit den in diesem Kapitel eingeführten abstrakten Operationen auf Binärbäumen als Klassenmethoden implementiert sähe eine Methode, die alle Knoten eines Binärbaums durchläuft und die Inhalte in einem String sammelt, in Java folgendermaßen aus:

```
16-1  // Durchlauf in preorder Reihenfolge
16-2  public static String preorder(Binaerbaum t) {
16-3      String s = "";
16-4      if (!isempty(t)) {
16-5          // Inhalt anhaengen
16-6          s += root(t) + " " + preorder(lefttree(t)) + preorder(righttree(t));
```

Listing 16.2: Berechnung der Quersumme einer zweistelligen Zahl.

```

16-1  /**
16-2   * Bestimme, welche zweistelligen Dezimalzahlen eine Quersumme von 2 haben
16-3   */
16-4  public class Quersumme2 {
16-5
16-6      // bestimme zu einer Zahl in [10, ..., 99] die Quersumme
16-7      static int quersumme2(int zahl) {
16-8          // addiere die beiden Ziffern
16-9          return (zahl / 10) + (zahl % 10);
16-10     }
16-11
16-12     public static void main(String [] args) {
16-13         final int wunschzahl = 2;
16-14         // teste alle Möglichkeiten durch
16-15         for(int zahl=10; zahl<=99; zahl++) {
16-16             if(quersumme2(zahl) == wunschzahl)
16-17                 System.out.println("gefundene Zahl ist " + zahl);
16-18         }
16-19     }
16-20 }
```

```

16-7     }
16-8     return s;
16-9 }
```

### 16.3 Erschöpfende Suche

Ähnlich einer Durchlaufstrategie durch eine Containerstruktur (Feld,...), wo jedes Datenelement genau einmal besucht und verarbeitet wird, lässt sich dieses allgemeine Prinzip auch in Form einer **vollständigen Suche** in einem beliebigen *diskreten* Lösungsraum als Strategie anwenden. Dazu müssen alle möglichen Lösungen systematisch aufgezählt werden und hinsichtlich Eignung / Optimalität bewertet werden. Ein anderer Begriff dafür ist der Ansatz der **Rohen Gewalt** (englisch: *Brute Force*).

Wichtig ist dabei, dass ein endlicher Lösungsraum vorliegen muss, da man alle potentiellen Kandidaten aufzählen muss. Möchte man zudem eine *optimale* Lösung ermitteln, müssen die einzelnen Lösungen auch bewertbar und vergleichbar sein. Man erzeugt bei diesem Ansatz einfach *alle* möglichen Lösungsinstanzen, bewertet jede einzelne Lösung und trifft dann basierend auf den Bewertungen aller Lösungen die geeignete Wahl für die optimale Lösung. Dieser Ansatz ist prinzipiell nur möglich, wenn der Lösungsraum endlich ist, und praktisch nur möglich, wenn der Lösungsraum nicht sehr groß ist beziehungsweise die Erzeugung der Elemente des Lösungsraums nicht zu aufwändig ist.

#### Beispiel 16.4:

Die Aufgabenstellung ist es, alle zweistelligen Dezimalzahlen ohne führende Null zu finden, deren Quersumme 2 ergibt. Dies lässt sich einfach feststellen (siehe Listing 16.2), indem man nacheinander für alle zweistelligen Dezimalzahlen  $\{10, 11, \dots, 99\}$  die Quersumme berechnet und diese Quersumme mit der Zielzahl vergleicht. Das Ergebnis ist hier also  $\{11, 20\}$ . ♦

Solche *Brute Force* Algorithmen werden dann eingesetzt, wenn entweder der Lösungsraum sehr klein ist oder wenn keine besseren Verfahren bekannt sind (zum Beispiel Passwortsuche).

**Beispiel 16.5:**

Dass *Brute Force* Ansätze oft aufgrund der Größe des möglichen Lösungsraums schnell an ihre Grenzen stoßen, zeigt die folgende Problemstellung. Es soll untersucht werden, ob es natürliche Zahlen  $a, b, c, n \in \mathbb{N}$  gibt, die die Bedingung  $a^n + b^n = c^n$  erfüllen (Fermat'sche Vermutung). Für  $n = 2$  ist eine Lösung bekannt:  $3^2 + 4^2 = 5^2$ . Für größeres  $n$  ist bis heute keine Lösung bekannt, ebensowenig ein *intelligentes* Verfahren, eine solche Zahlenkombination zu ermitteln, außer alle Möglichkeiten durchzuprobieren. Die Korrektheit der Vermutung wurde mathematisch in den 90er Jahren bewiesen.

Der hier verfolgte brutale Ansatz zur Überprüfung der Vermutung für eine Teilmenge der natürlichen Zahlen bis zu einer Obergrenze  $n$  ist nun, dass man einfach alle Möglichkeiten bis dieser Obergrenze für  $a, b, c, n$  durchprobiert. Dies ist als Programm sehr einfach zu formulieren, jedoch ist der Berechnungsaufwand dieses Programms sehr hoch. Im vorliegenden Fall wächst der Aufwand, wenn man bis zu einem Wert  $n$  alle Parameter testen will, wie  $\mathcal{O}(n^4)$ , da man bei vier Parametern  $a, b, c, n$  alle Kombinationen von Zahlen von 1 bis  $n$  durchtesten muss. Diese Aussage ist nicht ganz richtig, da man einen Teil der Möglichkeiten sich sparen kann, was dem Leser als Übung überlassen wird.

Bei diesem Beispiel sei als Besonderheit erwähnt, dass die Werte für  $a^n, b^n$  und  $c^n$  durch die Potenzbildung sehr schnell sehr groß werden können und damit eventuell nicht mehr mit dem ganzzahligen Datentyp `int` darstellbar sind. Aus diesem Grunde wurde der Datentyp `long` als Resultatwert der Potenzbildung gewählt. Doch selbst bei dieser Wahl kommt man schnell an Grenzen der Darstellbarkeit, da auch dieser Datentyp "nur" positive Werte bis maximal  $2^{63} - 1 \approx 9,2 \cdot 10^{18}$  darstellen kann.

Frage: Wäre die Verwendung des Datentyps `double` eine Lösung, um für diese Problemstellung mit noch größeren Werten zu rechnen? Überlegen Sie sich zuerst die Lösung zu dieser Frage und testen ihre Vermutung anschließend in Form eines modifizierten Programms aus. ♦

Listing 16.3: Teilüberprüfung der Fermatschen Vermutung.

```

16-1  /** Fermat pruefen
16-2   * @author Rudolf Berrendorf
16-3   * @version 1.0
16-4   */
16-5  public class Fermat {
16-6
16-7      static long hoch(long basis, long exponent) {
16-8          long resultat = 1;
16-9          for(long i=1; i<=exponent; i++)
16-10              resultat = resultat * basis;
16-11          return resultat;
16-12      }
16-13
16-14      static void pruefeFermat(long max) {
16-15          boolean gefunden = false;
16-16
16-17          // wir brauchen die Werte nach der Schleife, deshalb Variablen-deklaration
16-18          // außerhalb der Schleifen (Gültigkeitsbereich!)
16-19          long a=1, b=1, c=1, n=3;
16-20          for(a=1; a<=max; a=a+1) {
16-21              for(b=1; b<=max; b=b+1) {
16-22                  for(c=1; c<=max; c=c+1) {
16-23                      for(n=3; n<=max; n=n+1) {
16-24                          if(hoch(a,n) + hoch(b,n) == hoch(c,n)) {
16-25                              System.out.println("Loesung gefunden: " + a + ", " + b + ", " + c + ", " + n);
16-26                              System.out.println(hoch(a,n) + " + " + hoch(b,n) + " = " + hoch(c,n));
16-27                          gefunden = true;
16-28                          break;    // Frage: was bewirkt das hier und was nicht?
16-29                      }
16-30                  }
16-31              }
16-32          }
16-33      }
16-34
16-35      if(!gefunden) {
16-36          System.out.println("keine Loesung gefunden");
16-37      }
16-38  }
16-39
16-40  public static void main(String[] args) {
16-41      // lese Wert aus Kommandozeile ein, bis zu dem getestet werden soll
16-42      long max = Long.parseLong(args[0]);
16-43      pruefeFermat(max);
16-44  }
16-45 }
```

## 16.4 Gieriger Ansatz (Greedy)

Beim gierigen Ansatz (englisch: Greedy) wird in einem nächsten Schritt immer der Teilschritt genommen, der am meisten Profit, Fortschritt und so weiter verspricht.

**Arbeitsverteilung** Bei einer Arbeitsverteilung (englisch: Scheduling) geht es darum, eine Reihe von Aufträgen mit einer bestimmten Bearbeitungsdauer so auf Maschinen mit einer bestimmten Verarbeitungsleistung zu verteilen, dass die gesamte Arbeit möglichst schnell erledigt ist. Wir beschränken uns hier auf den Spezialfall, dass alle Maschinen die gleiche Verarbeitungsleistung haben.

Als Beispiel sollen drei Maschinen und die folgenden Arbeiten genommen werden, angegeben jeweils durch den Arbeitsaufwand in Zeiteinheiten:

Aufgabe	1	2	3	4	5	6
Zeitaufwand	5	8	2	3	5	1

Eine gierige Strategie würde die nächste noch nicht bearbeitete Aufgabe an die Maschine vergeben, die nach der bisherigen Verteilung am Wenigsten zu tun hat. Demnach würde die Aufteilung erfolgen:

Aufgabe	1	2	3
Zeitaufwand	5	8	2

Eine bessere Lösung wäre allerdings durch folgende Aufteilung gegeben:

Dieses Beispiel zeigt, dass eine solche gierige Strategie nicht unbedingt immer die beste sein muss.

## 16.5 Problemlösung durch Rekursion

Rekursion ist ein sehr mächtiges Hilfsmittel um komplexe Probleme zu lösen, indem man komplexe Fälle auf einfache Fälle zurückführt. In den meisten Anwendungen der rekursiven Definition bis jetzt haben wir eine im Prinzip beliebig große Struktur (zum Beispiel Folge, Binärbaum) auf einfache Strukturen zurückgeführt und dies in der Definition von Funktionen oder Operationen ausgenutzt. Eine Folge hat ein erstes Element und eine Restfolge, ein Binärbaum besteht aus einem Wurzelknoten sowie linkem und rechtem Teilbaum, die jeweils wieder Binäräume sind.

Anhand von zwei Beispielen sollen weitere Einsatzmöglichkeiten der Rekursion betrachtet werden: Türme von Hanoi und das Sortieren einer Folge von Zahlen.

### 16.5.1 Türme von Hanoi

**Türme von Hanoi:** Der Legende nach befasste sich ein buddhistischer Mönchsorden in Hanoi (Vietnam) mit der folgenden Aufgabe. Auf einem Brett sind 3 senkrechte Stäbe, mit  $a$ ,  $b$  und  $c$  bezeichnet, und  $n$  runde Scheiben verschiedener Größe, die ein Loch in der Mitte haben, so dass man sie auf einen der senkrechten Stäbe stecken kann (Abbildung 16.2). Zu Beginn des Spieles sind alle Scheiben auf Stab  $a$  aufgesteckt, so dass die größte Scheibe unten liegt und die kleinste Scheibe oben. Die Aufgabe ist es, den ursprünglichen Turm von Scheiben von Stab  $a$  auf Stab  $b$  zu bringen, so dass wiederum die größte Scheibe unten und die kleinste Scheibe oben liegt. Als Einschränkungen gelten jedoch:

- 1) Nur die oberste Scheibe eines Stapels darf auf einem der drei Stäbe bewegt werden.
- 2) Es darf nie eine größere Scheibe auf einer kleineren liegen.

Aufgrund der obigen Einschränkungen ist es klar, dass die einzige Möglichkeit, die größte zuunterst liegende Scheibe vom Stab  $a$  als unterste Scheibe auf Stab  $b$  zu bekommen darin liegt, alle restlichen Scheiben auf dem Stapel  $a$  wegzuschaffen, was wiederum nur über den verbleibenden dritten Stab geschehen kann. Zuerst einmal soll vernachlässigt werden, ob und wie man dieses Wegschaffen eine Teilstapels fertig bringt. Eine algorithmische Formulierung für die Bewegung der untersten Scheibe von  $a$  nach  $b$  ist also:

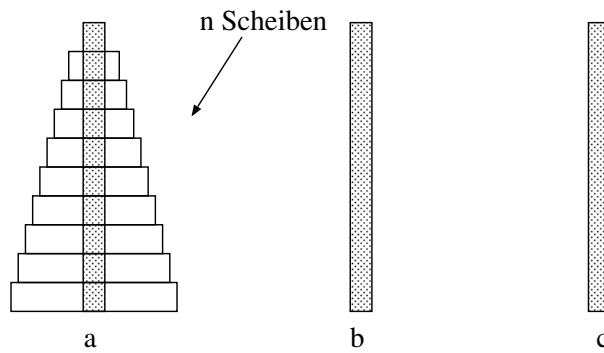
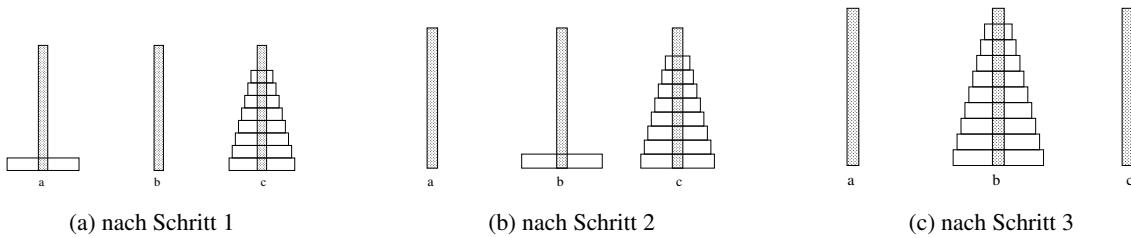
Figure 16.2: Spielbrett für *Türme von Hanoi*

Figure 16.3: Zwischenschritte in der Bewegung der untersten Scheibe

Übertrage die obersten  $n-1$  Scheiben von Stab  $a$  nach Stab  $c$

Bewege die unterste Scheibe von Stab  $a$  nach Stab  $b$

Übertrage die obersten  $n-1$  Scheiben von Stab  $c$  nach Stab  $a$

Unterschieden wird auch sprachlich zwischen dem Verschieben eines Teilstapels (*Übertragen*) und der Elementaroperation des Verschiebens genau einer Scheibe (*Bewegen*). Mit *Quelle* wird der Stab aus dem ursprünglichen Stapel bezeichnet, mit *Senke* der Stab, wohin übertragen werden soll und als *Arbeitsbereich* der Stab, der temporär die Scheiben des Teilstapels aufnehmen soll. Dann kann man die obigen Schritte in der bekannten Modulschreibweise auch allgemeiner formulieren:

**Modul** Schritt (Quelle, Senke, Arbeitsbereich,  $n$ )

    übertragen(Quelle, Arbeitsbereich,  $n-1$ )

    bewege(Quelle, Senke)

    übertragen(Arbeitsbereich, Senke,  $n-1$ )

**Modulende**

Unser konkretes Problem, die unterste Scheibe von  $a$  nach  $b$  zu bekommen, würde also dann lauten: *Schritt*( $a, b, c, n$ ). Abbildung 16.3 zeigt die Zwischenschritte im Ablauf.

Nach der Übertragung der größten Scheibe nach obigem Algorithmus hat man das nächste Problem, dass man die zweitgrößte Scheibe von  $c$ , wo ja jetzt der restliche Stapel liegt, nach  $b$  bekommen muss. Auch dies kann man aufgrund der Einschränkungen der Spielregeln nur erreichen, indem die größte verbleibende, auf Stapel  $c$  zuunterst liegende Scheibe nach  $b$  bewegt werden kann, indem vorher alle restlichen Scheiben auf dem Stapel  $c$  weggeschafft werden, was wiederum nur über den verbleibenden leeren Stab, diesmal  $a$ , geschehen kann. Unter Ausnutzung unseres oben definierten Moduls könnte dies also formuliert werden als *Schritt*( $c, b, a, n-1$ ). Nach der zweitgrößten Scheibe würde die drittgrößte Scheibe genauso behandelt werden und so weiter. Allgemein ergibt sich für die  $i$ -te Scheibe das Muster *Schritt*(*Quelle, SenkeArbeitsbereich,  $i-1$* ).

Insgesamt ergibt sich als Gesamtlösung folgender rekursiver Algorithmus:

**Modul Hanoi** (Quelle, Senke, Arbeitsbereich, n)

```

Falls n = 1
    bewege(Quelle, Senke)
sonst
    Hanoi(Quelle, Arbeitsbereich, Senke, n-1)
    bewege(Quelle, Senke)
    Hanoi(Arbeitsbereich, Senke, Quelle, n-1)

```

### Modulende

Die Lösung des Gesamtproblems wird also auf die Lösung eines verkleinerten aber gleichartigen Problems zurückgeführt.

Der Ablauf soll an einem Beispiel mit 3 Scheiben nachvollzogen werden. Abbildung 16.4 zeigt links die Aufrufhierarchie, rechts das Resultat der Schritte, wenn eine Scheibe bewegt wurde. Modulaufrufe sind jeweils durch Rahmen gekennzeichnet.

Der Algorithmus als Java-Programm sieht dann aus wie in Listing 16.4. Die Ausgabe des Programms ist:

```

16-1 Bewege Scheibe von 0 nach 2
16-2 Bewege Scheibe von 0 nach 1
16-3 Bewege Scheibe von 2 nach 1
16-4 Bewege Scheibe von 0 nach 2
16-5 Bewege Scheibe von 1 nach 0
16-6 Bewege Scheibe von 1 nach 2
16-7 Bewege Scheibe von 0 nach 2

```

Das ursprüngliche Problem war für  $n = 64$  formuliert. Jeder Aufruf von Hanoi, der in den else-Fall läuft, bewirkt 2 rekursive Aufrufe von Hanoi, das heißt 2 Berechnungen für  $n = 63$ , jeder dieser Aufrufe wiederum 2 Aufrufe für  $n = 62$  und so weiter. Damit ergeben sich insgesamt  $2^n - 1$  Aufrufe von Hanoi. Da bei jedem Aufruf genau eine Scheibe bewegt wird (in beiden Fällen der Selektion findet genau eine Bewegung statt), erhält man  $2^n - 1 = \mathcal{O}(2^n)$  Scheibenbewegungen. Wenn die Mönche jede Sekunde eine Scheibe bewegen und dabei nie einen Fehler machen würden, würde die Ausführung des Algorithmus durch die Mönche ungefähr  $2^{64} - 1 \text{ s} \approx 600.000.000.000 \text{ Jahre}$  dauern.

## 16.5.2 Rekursives Sortieren

Als zweites Beispiel für einen rekursiv definierten Lösungsprozess soll das Sortieren einer Folge von Zahlen dienen. Gegeben ist eine Folge von  $n$  Zahlen:  $f_1 = < x_1, \dots, x_n >$ , aus denen eine neue Folge  $f_2 = < x_{i_1}, \dots, x_{i_n} >$  aus aufsteigend sortierten Zahlen der ersten Folge erzeugt werden soll, das heißt  $x_{i_k} \leq x_{i_{k+1}} \forall 1 \leq k < n$ . Aus der Folge  $< 8, 7, 5, 6, 4, 2, 3, 1 >$  soll also die Folge  $< 1, 2, 3, 4, 5, 6, 7, 8 >$  erzeugt werden.

Das Sortieren von Zahlenfolgen ist ein in der Informatik intensiv behandeltes Thema, das später auch noch einmal aufgegriffen wird. An dieser Stelle soll jedoch eine einfache und elegante Lösung gefunden werden. Man geht vereinfachend davon aus, dass die Anzahl der zu sortierenden Zahlenwerte eine Zweierpotenz ist, das heißt  $n = 2^k$  für ein  $k \in \mathbb{N}$ .

1. Der Fall  $n = 1$  ist trivial, eine Zahlenfolge bestehend aus einer Zahl ist sortiert.
2. Für den Fall  $n = 2$  kann man folgende direkte Lösung angeben: Vergleiche die beiden Zahlen und vertausche sie, falls die erste Zahl größer ist als die zweite.
3. Für  $n = 4$  wäre ein direkter Vergleich aller 4 Zahlen schon sehr mühselig. Man kann sich aber zunutze machen, dass man bereits wissen, wie Folgen mit zwei Zahlen sortiert werden können. Wir sortieren zuerst die beiden ersten Zahlen nach diesem Verfahren, anschließend sortiert man die beiden letzten Zahlen nach diesem Verfahren. Für diese beiden sortierten Teilstücke kann man anschließend eine Art

Hanoi(a,b,c,3)

Quelle=a, Senke=b, Arbeit=c, n=3

Falls 3=1

dann ...

sonst Hanoi(a,c,b,2)

Quelle=a, Senke=c, Arbeit=b, n=2

Falls 2=1

dann

sonst Hanoi(a, b, c, 1)

Quelle=a, Senke=b, Arbeit=c, n=1

Falls 1=1

dann bewege(a,b)

bewege(a,c)

Hanoi(b, c, a, 1)

Quelle=b, Senke=c, Arbeit=a, n=1

Falls 1=1

dann bewege(b, c)

bewege(a,b)

Hanoi(c,b,a,2)

Quelle=c, Senke=b, Arbeit=a, n=2

Falls 2=1

dann

sonst Hanoi(c, a, b, 1)

Quelle=c, Senke=a, Arbeit=b, n=1

Falls 1=1

dann bewege(c, a)

bewege(c,b)

Hanoi(a, b, c, 1)

Quelle=a, Senke=b, Arbeit=c, n=1

Falls 1=1

dann bewege(a, b)

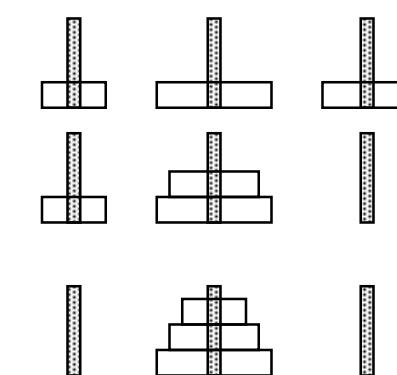
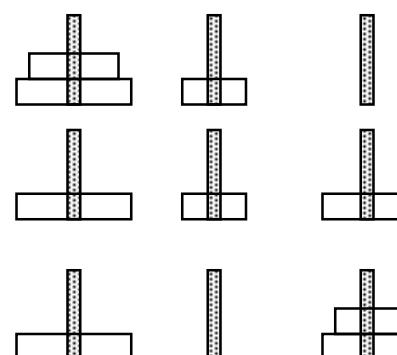
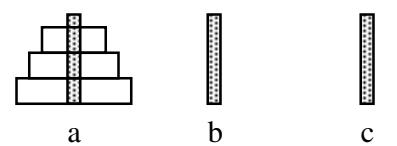


Figure 16.4: Ablauf des Beispiels mit 3 Scheiben

Listing 16.4: Türme von Hanoi als Java-Programm.

```

16-1  /**
16-2   * Tuerme vpon Hanoi
16-3   * @author Rudolf Berrendorf
16-4   * @version 1.0
16-5   */
16-6 public class Hanoi {
16-7
16-8     public static void main(String[] args) {
16-9         int n=3; // Anzahl zu verwendeter Scheiben
16-10
16-11         // bewege Stapel mit n=3 Scheiben von Stab 0 nach Stab 1 Stab 2
16-12         Hanoi(0,1,2,n);
16-13     }
16-14
16-15     /** Rekursive Loesung fuer Tuerme von Hanoi
16-16      * @param quelle Stab, auf dem die zu bewegenden Scheiben liegen
16-17      * @param senke Stab, auf dem die zu bewegenden Scheiben landen sollen
16-18      * @param arbeitsbereich Stab, der als Zwischenlager genutzt werden kann
16-19      * @param n Anzahl an Scheiben
16-20      */
16-21     static void Hanoi(int quelle, int senke, int arbeitsbereich, int n) {
16-22         if(n == 1)
16-23             // bewege eine Scheibe direkt
16-24             bewege(quelle, senke);
16-25         else {
16-26             // uebertrage n-1 obersten Scheiben und lege unterste der n Scheiben frei
16-27             Hanoi(quelle, arbeitsbereich, senke, n-1);
16-28             // bewege diese eine Scheibe
16-29             bewege(quelle, senke);
16-30             // uebertrage die Scheiben vom Zwischenspeicher zum Zielstab
16-31             Hanoi(arbeitsbereich, senke, quelle, n-1);
16-32         }
16-33     }
16-34
16-35     /** Bewege eine Scheibe.
16-36      * In dieser Methode koennten Sie auch einen Roboterarm scnschliessen, der dies
16-37      * ausfuehrt.
16-38      * @param quelle Stab, von dem die oberste Scheibe weg bewegt werden soll
16-39      * @param senke Stab, auf dem die Scheibe landen soll
16-40      */
16-41     static void bewege(int quelle, int senke) {
16-42         System.out.println("Bewege Scheibe von " + quelle + " nach " + senke);
16-43     }
16-44 }
```

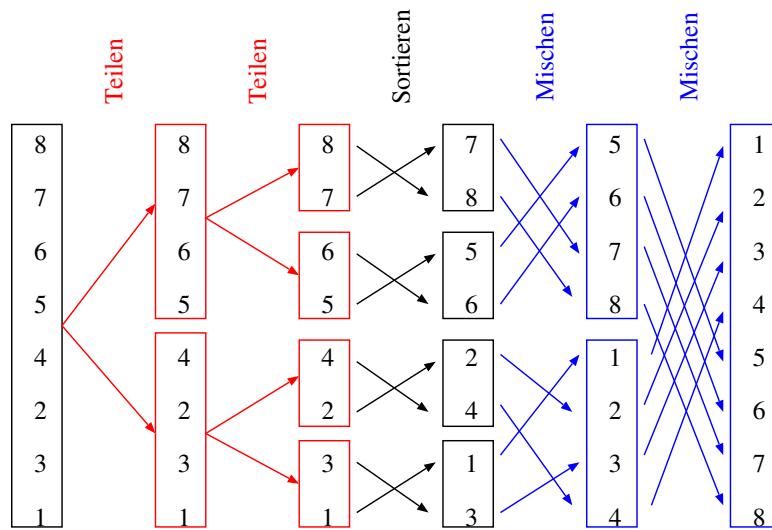


Figure 16.5: Rekursives Sortieren einer Folge

Reißverschlußverfahren anwenden, indem man jeweils das erste Element der ersten Teilfolge mit dem ersten Element der zweiten Folge vergleicht. Das kleinere von beiden ist dann das erste Element der Ergebnisfolge. Anschließend wird das erste noch nicht übernommene Element der ersten Folge mit dem ersten noch nicht übernommenen Element der zweiten Folge verglichen und wiederum das kleinere in die Resultatfolge übernommen und so weiter. Nach spätestens vier solcher Schritte sind alle Zahlen der beiden Ursprungsteilfolgen in die Resultatfolge übernommen, die sortiert vorliegt. Dieses zuletzt vorgestellte Reißverschlußverfahren nennt man Mischen.

4. Für  $n = 8$  würde man das gleiche Verfahren wie bei  $n = 4$  anwenden, nur dass diesmal Teilfolgen der Länge 4 vorsortiert werden müssten.

Als allgemeine Lösung des Problems ergibt sich nun aus der vorangegangenen Diskussion folgender Algorithmus:

#### Modul Sortieren(f,n)

```

Falls n = 2
    Falls first(f) > first(rest(f)) vertausche( first(f), first(rest(f)))
Sonst
    Teile die Folge f in zwei Hälften  $f_{11}$  und  $f_{12}$  auf
    Sortiere (rekursiv)  $f_{11}$  zu  $f_{21}$ 
    Sortiere (rekursiv)  $f_{12}$  zu  $f_{22}$ 
    Mische (füge zusammen) die Resultatsfolgen  $f_{21}$  und  $f_{22}$ 
```

#### Modulende

Die Rekursion bricht also ab, wenn Folgen der Länge 2 erreicht sind. In diesem Fall wird das erste mit dem zweiten Element direkt verglichen und vertauscht, falls die beiden Elemente in der falschen Reihenfolge vorliegen. Das Mischen zweier sortierter Folgen  $f_i$  und  $f_j$  geschieht so, dass aus den beiden Folgen  $f_i$  und  $f_j$  jeweils das kleinere Element in die Ergebnisfolge übernommen wird, bis alle Elemente der Ausgangsfolgen übertragen sind.

Abbildung 16.5 zeigt ein Beispiel zum Ablauf des Algorithmus für die obige Beispieldfolge.

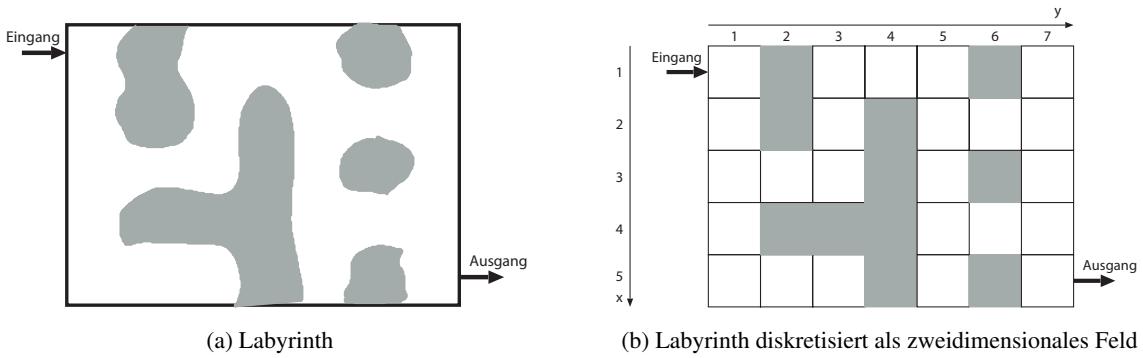


Figure 16.6: Reales und diskretisiertes Labyrinth.

## 16.6 Backtracking

Rekursion ist ein mächtiges Werkzeug, das implizit auch genutzt wird in Algorithmen, die nach dem Prinzip des **Backtracking** (Zurückverfolgung) arbeiten. Backtracking könnte man auch Systematisches Ausprobieren mit Erinnern nennen.

Ein typisches Beispiel für den Einsatz eines solchen Algorithmus ist das Auffinden eines Weges durch ein unbekanntes Labyrinth von einem gegebenen Startpunkt zu einem Endpunkt (Abbildung 16.6a). Dieses Problem scheint zuerst einmal wenig geeignet, es direkt lösen zu können, auch wenn eine Lösung für das Beispillabyrinth aufgrund der geringen Dimensionen für den menschlichen Betrachter offensichtlich ist. Um einen Algorithmus für die gegebene Problemstellung zu entwickeln, der bei beliebig großen Labyrinthen eine Lösung (oder alle Lösungen) findet, ist es in diesem Fall hilfreich, das unstrukturierte Problem in ein geeignetes Modell abzubilden. Es soll nun versucht werden, für das Modell eine Lösung zu entwickeln und diese Lösung auf das ursprüngliche Problem zu übertragen.

Man stellt sich nun vor, dass das gesamte Labyrinth in rechteckige Felder aufgeteilt ist (Abbildung 16.6b), wobei jedes Feld entweder belegt ist (dort steht eine Hecke, Mauer etc.) oder frei ist (dort kann man sich aufhalten beziehungsweise durchgehen). Dieser Übergang vom Labyrinth, wie man es in der Natur sieht, zum rechteckigen Feld war bereits ein wichtiger Schritt: es wurde ein geeignetes (Informatik-) Modell gebildet, auf das sich – wie man noch sehen kann – das (Informatik-) Prinzip des Backtracking anwenden lässt um zu einem für (im Prinzip) beliebig große Labyrinthe gültigen Lösungsalgorithmus zu kommen. Abbildung 16.7 zeigt dieses Vorgehen.

Wie könnte man nun einen Weg durch das Labyrinth finden? Beginnt man mit der Startposition (1,1), so sieht man, dass man in diesem Beispillabyrinth für den ersten Schritt keine Alternative hat. Man kann nur zur Position (2,1) und von dort zur Position (3,1) gehen, wenn man von (2,1) nicht sofort wieder zurück nach (1,1) gehen will. An Position (3,1) müssen man sich dann aber definitiv entscheiden: nach unten weitergehen oder rechts abbiegen? Wie soll weiter verfahren werden? Wie könnte man die Suche nach einem Weg (oder in einer modifizierten Aufgabenstellung nach allen möglichen Wegen) durch das Labyrinth systematisch durchführen?

Ein einfacher Lösungsalgorithmus ist es, einfach alle möglichen vom Startpunkt ausgehenden Wege auszuprobieren. Dabei tauchen aber (mindestens) zwei Fragen auf:

1. Kann man (und wenn ja, wie) alle möglichen Wege erzeugen?

Die Antwort lautet: ja, man kann alle möglichen Wege erzeugen. Ein Verfahren dazu wird gleich vorgestellt.

2. Ist die Menge der möglichen Wege überhaupt endlich? Die Antwort auf diese Frage lautet: nein! Dies

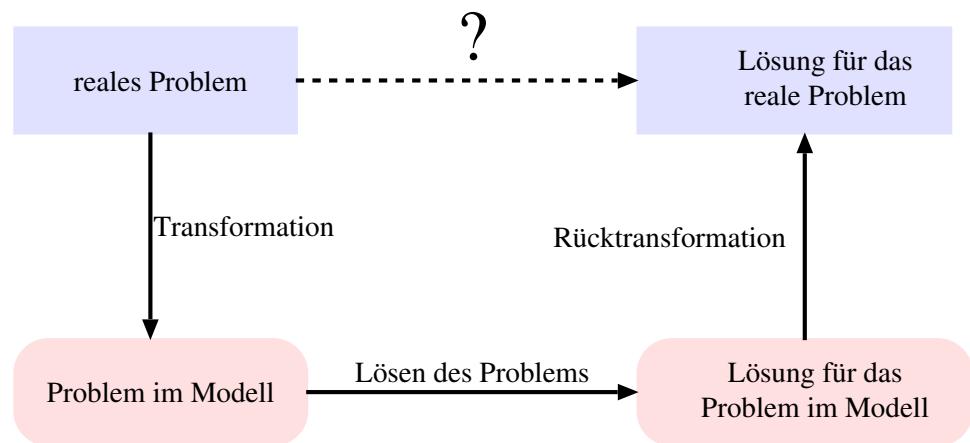


Figure 16.7: Modellbildung

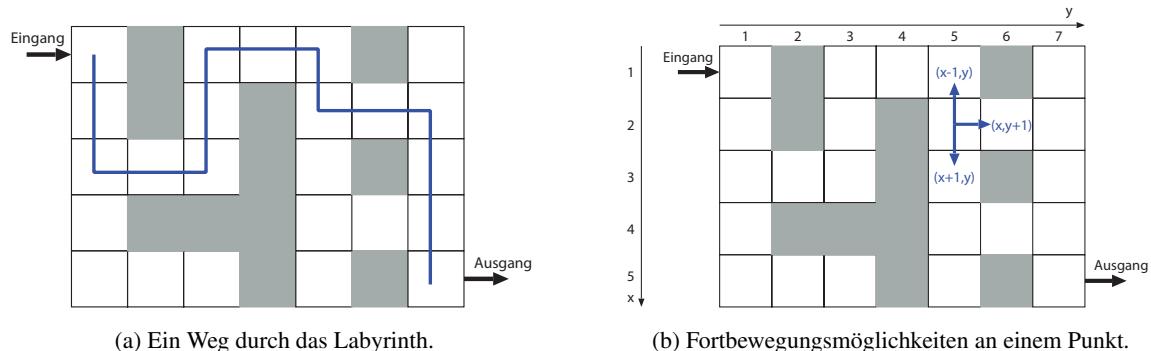


Figure 16.8: Wege im Labyrinth

wird sofort klar, wenn man sich ein Labyrinth anschaut, in dem ein Rundweg (Kreis) möglich. Wenn ich einmal im Kreis laufe, so kann ich auch zweimal im Kreis laufen, dreimal und so weiter. Genauso ist es natürlich möglich (wenn auch unsinnig), dass man vorwärts und gleich wieder zurück geht. Wir werden aber auch dieses Problem lösen, so dass man letztendlich auf eine endliche Zahl von zu testenden Wegen kommt.

Wie aber kann man nun garantieren, dass alle möglichen Wege auch wirklich getestet werden? Dazu kann man als Fortbewegungsmethode folgende Strategie anwenden: Auf einer Position  $(x,y)$  wird immer (wenn möglich) zuerst nach oben gegangen, dann nach unten, dann nach rechts und zuletzt nach links. Welche Strategie hier konkret angewandt wird (zuerst nach links oder rechts oder oben oder unten), ist nebensächlich, wichtig ist nur, dass eine konkrete Strategie existiert alle von  $(x,y)$  ausgehenden Wege zu besuchen. Wendet man die Strategie auch beim Besuch der Nachbarfelder und deren Nachbarfelder an, so werden alle möglichen von  $(x,y)$  ausgehenden Wege durchlaufen.

An dieser Stelle muss man allerdings aufpassen, dass man nicht unendlich im Kreise läuft! Man muss sich merken, ob man schon einmal an einer Position  $(x,y)$  war. Ist dies der Fall, so darf man von dort nicht weitergehen, sondern muss die Suche an dieser Stelle abbrechen. Ist man nämlich an einer Position  $(x,y)$  angelangt, an der man bereits zu einem früheren Zeitpunkt erfolglos alle möglichen von  $(x,y)$  ausgehenden Wege ausprobiert hat, so weiß man beim zweiten und jedem weiteren Besuch, dass es nutzlos ist hier weiterzumachen. Man muss sich also zu jedem Punkt  $(x,y)$  die Information merken, ob dieser Punkt bereits erfolglos getestet wurde oder nicht.

Zusammenfassend kann man jetzt folgende Strategie anwenden. Alle freien Felder sind zu Beginn als unbesucht markiert. Beginne bei der Startposition. Kommt man auf der Wanderung zu einer Position  $(x,y)$ , die man bereits besucht hat, so bricht man dort die Suche sofort ab, denn man hat alle möglichen Wege von dort bereits betrachtet. War man noch nicht an dieser Position, markiert man diese als besucht. Ist die Position  $(x,y)$  die Endposition, so war man in der Suche erfolgreich und ist fertig. Ansonstenwendet man die Besuchsstrategie an und versucht zuerst einen Weg nach unten zu verfolgen. Ist dieser Weg erfolgreich, hat man eine Gesamtlösung gefunden. Ansonsten versucht man es nach oben, dann nach links und dann nach rechts. Waren alle möglichen Wege erfolglos, so ist vom Punkt  $(x,y)$  keine Lösungsweg zu finden und gibt dies als Ergebnis der Suche vom Punkt  $(x,y)$  an.

Die oben beschriebene Lösungsstrategie an einem Punkt  $(x,y)$  ist in der Methode `wegFinden` angegeben. Die Parameter `aktuell_x` und `aktuell_y` entsprechen den Koordinaten  $(x,y)$  und die Parameter `ende_y` und `ende_y` geben den Zielpunkt an. Wird die Methode für einen Punkt  $(x,y)$  aufgerufen, so wird zuerst dieser Punkt als besucht markiert, indem `brett[aktuell_x][aktuell_y]` auf `B` (besucht) gesetzt wird. Als nächstes testen wir, ob der vorliegende Punkt der gesuchte Zielpunkt ist. Falls dies der Fall ist, so bricht man sofort ab und liefert als Resultat der Methode den Wahrheitswert `true`. Ansonsten befindet man sich auf einem noch nicht besuchten Feld, das nicht dem Endpunkt entspricht und muss von dort alle möglichen Wege testen. Dies geschieht – wie oben bereits beschrieben –, indem man zuerst alle Wege testet, die mit dem unteren Nachbarfeld beginnen. Nun, das liefert natürlich ein rekursiver Aufruf unserer Methode `wegFinden` mit den Koordinaten `(aktuell_x + 1, aktuell_y)`. Dieser rekursive Aufruf darf aber nur erfolgen, falls überhaupt ein unteres Nachbarfeld existiert und dieses zudem noch unbesucht ist, was im Java-Programm die Codezeile `if((aktuell_x+1 < brett.length) && (brett[aktuell_x+1][aktuell_y] == F))` bewirkt. Liefert der rekursive Aufruf das Ergebnis `true`, so wurde ein Weg gefunden, der mit dem unteren Nachbarfeld beginnt und man kann abbrechen. Ist das Resultat des Aufrufs dagegen `false`, so existiert kein Weg zum Ziel, das vom unteren Nachbarfeld ausgeht und man muss mit den drei verbliebenen Nachbarfelder (oben, rechts, links) der Reihe nach genauso verfahren.

Listing 16.5: Durchlaufen eines Labyrinths (Teil 1).

```

16-1 /**
16-2 * Finde den Weg durch ein Labyrinth ueber Backtracking .
16-3 * @author Rudolf Berrendorf
16-4 */
16-5 public class Labyrinth {
16-6     // Die naechsten drei Konstanten dienen der Markierung eines Feldes im Labyrinth
16-7     static final int X = -1;           // Feld blockiert
16-8     static final int F = 0;           // Feld ist frei
16-9     static final int B = 1;           // Feld schon besucht
16-10    // Dies ist unser anfaengliches Spielbrett
16-11    static int[][] brett =
16-12    {
16-13        {F, X, F, F, F, X, F},
16-14        {F, X, F, X, F, F, F},
16-15        {F, F, F, X, F, X, F},
16-16        {F, X, X, X, F, F, F},
16-17        {F, F, F, X, F, X, F},
16-18    };
16-19
16-20    public static void main(String[] args) {
16-21        // Suchen einen Weg durch das Labyrinth beginnend vom Startpunkt
16-22        wegFinden(0, 0, brett.length-1, brett[0].length-1);
16-23    }
16-24
16-25 /**
16-26     * Gebe den aktuellen Inhalt des Labyrinths (inkl. Zusatzinformationen) aus.
16-27     * @param aktuell_x aktuelle Position in x-Richtung
16-28     * @param aktuell_y aktuelle Position in y-Richtung
16-29 */
16-30    static void drucke_labyrinth(int aktuell_x, int aktuell_y) {
16-31        System.out.println("O=aktuelle Position , X=blockiert , .=frei , *=schon besucht");
16-32        for(int i = 0; i < brett.length; i++) {
16-33            for(int j = 0; j < brett[i].length; j++) {
16-34                if((i == aktuell_x) && (j == aktuell_y)) {
16-35                    // aktuelle Position
16-36                    System.out.print("O");
16-37                } else if(brett[i][j] == X) {
16-38                    // Feld blockiert
16-39                    System.out.print("X");
16-40                } else if(brett[i][j] == F) {
16-41                    // Feld frei
16-42                    System.out.print(".");
16-43                } else if(brett[i][j] == B) {
16-44                    // Feld schon ausprobiert
16-45                    System.out.print("*");
16-46                } else {
16-47                    // sollte nie vorkommen :-
16-48                    System.out.println("hier stimmt was nicht");
16-49                }
16-50            }
16-51
16-52            // Ausgabezeile beenden
16-53            System.out.println();
16-54        }
16-55
16-56        // Eine Leerzeile hinterherschieben
16-57        System.out.println();
16-58    }

```

Listing 16.6: Durchlaufen eines Labyrinths (Teil 2).

```

16-1  /*
16-2   * Suche systematisch einen Weg durch ein Labyrinth
16-3   * @param aktuell_x aktuelle Position in x-Richtung
16-4   * @param aktuell_y aktuelle Position in y-Richtung
16-5   * @param ende_x      Zielposition in x-Richtung
16-6   * @param ende_y      Zielposition in y-Richtung
16-7   * @return Liefert true, wenn ein Weg gefunden wurde, ansonsten false
16-8 */
16-9  static boolean wegFinden(int aktuell_x, int aktuell_y, int ende_x, int ende_y) {
16-10    boolean resultat;
16-11
16-12    // Diese Stelle ist ab sofort ausprobiert
16-13    brett[aktuell_x][aktuell_y] = B;
16-14    // Brett zur Kontrolle ausgeben
16-15    drucke_labyrinth(aktuell_x, aktuell_y);
16-16
16-17    // Test, ob wir den Endpunkt erreicht haben
16-18    if((aktuell_x == ende_x) && (aktuell_y == ende_y)) {
16-19      // wir haben einen Weg gefunden und sind fertig
16-20      System.out.println("Weg gefunden (umgekehrte Reihenfolge):");
16-21      return true;
16-22    }
16-23
16-24    else {
16-25      // Wir sind noch nicht am Endpunkt angelangt
16-26      // Wir probieren es einen Schritt nach unten (x+1)
16-27      if((aktuell_x+1 < brett.length) && (brett[aktuell_x+1][aktuell_y] == F)) {
16-28        resultat = wegFinden(aktuell_x + 1, aktuell_y, ende_x, ende_y);
16-29        if(resultat) {
16-30          System.out.println("(" + aktuell_x + "," + aktuell_y + ")");
16-31          return true;
16-32        }
16-33      }
16-34      // Wir probieren es einen Schritt nach oben (x-1)
16-35      if((aktuell_x-1 >= 0) && (brett[aktuell_x-1][aktuell_y] == F)) {
16-36        resultat = wegFinden(aktuell_x - 1, aktuell_y, ende_x, ende_y);
16-37        if(resultat) {
16-38          System.out.println("(" + aktuell_x + "," + aktuell_y + ")");
16-39          return true;
16-40        }
16-41      }
16-42      // Wir probieren es einen Schritt nach rechts (y+1)
16-43      if((aktuell_y+1 < brett[aktuell_x].length) && (brett[aktuell_x][aktuell_y+1] ==
16-44        F)) {
16-45        resultat = wegFinden(aktuell_x, aktuell_y+1, ende_x, ende_y);
16-46        if(resultat) {
16-47          System.out.println("(" + aktuell_x + "," + aktuell_y + ")");
16-48          return true;
16-49        }
16-50      }
16-51      // Wir probieren es einen Schritt nach links (y-1)
16-52      if((aktuell_y-1 >= 0) && (brett[aktuell_x][aktuell_y-1] == F)) {
16-53        resultat = wegFinden(aktuell_x, aktuell_y - 1, ende_x, ende_y);
16-54        if(resultat) {
16-55          System.out.println("(" + aktuell_x + "," + aktuell_y + ")");
16-56          return true;
16-57        }
16-58      }
16-59      // Ueber diesen Punkt sind wir nicht zum Ziel gekommen
16-60      return false;
16-61    }
16-62  }

```

## 16.7 Zusammenfassung und Hinweise

### Literaturhinweise

Zur historischen Entwicklung einiger Algorithmen gibt [Zie96] und [Cha99] eine Übersicht. Dort sind auch didaktisch aufbereitete Herleitungen einiger einfacher Algorithmen zu finden.

Eine Übersicht über allgemeine Lösungsansätze gibt [MF04].

## Chapter 17

# Einführung in das Testen und Verifizieren

Jeder, der schon einmal programmiert hat, wird Fehler während der Entwicklung der Programmlogik oder während der Programmierung selber gemacht haben. Manche von diesen Fehlern sind dem Compiler bereits aufgefallen, manche vielleicht bis heute überhaupt nicht. Die *Korrektheit* eines Programms, das heißt die Erfüllung einer vorgegebenen Spezifikation, ist aber eine notwendige Voraussetzung für seinen Einsatz. Erfüllt das Programm die Spezifikation nicht, so macht ein Einsatz keinen Sinn; denn dann könnten Sie auch irgendein anderes Programm verwenden, das diese Spezifikation dann ebenfalls nicht erfüllt.

Arbeitet ein Programm nicht korrekt, so liegt ein Fehler vor. Eine erste Fehlerquelle ist deshalb auch schon bei der Spezifikation zu sehen, zum Beispiel ungenaue oder falsche Angaben der Problembeschreibung. Man teilt mögliche Fehler im Rahmen der eigentlichen Programmierung in drei unterschiedliche Klassen ein:

1. Ein *Syntaxfehler* liegt dann vor, wenn der formulierte Programmtext die syntaktischen Regeln der Programmiersprache verletzt (vergleiche Kapitel 3). Oder anders ausgedrückt, wenn vom Startsymbol der Sprachgrammatik keine Ableitung zum Programmtext möglich ist. Dies bedeutet dann auch, dass der formulierte Programmtext kein ableitbarer Satz der Sprache ist und damit der Text kein gültiges Programm ist.

Beispiel: `(3+4`

Alle Syntaxfehler werden vom Compiler erkannt und entsprechend markiert. Solange ein Programm mindestens einen Syntaxfehler enthält, wird der Compiler kein lauffähiges Programm erzeugen.

2. Ein *semantischer Fehler* liegt vor, wenn das Programm syntaktisch korrekt ist, aber die Bedeutung des Programms nicht korrekt im Sinne der Sprachsemantik ist.

Beispiel: `int[] a = {1, 2, 3}; int x = a[3];`

Teilweise können einige einfache semantischen Fehler vom Compiler oder Programmierwerkzeugen erkannt werden, viele aber nicht.

3. Ein *logischer Fehler* liegt vor, wenn sowohl die Syntax als auch die Semantik der Sprache eingehalten wurde, aber die Programmlogik nicht korrekt ist, also nicht das ausdrückt, was eigentlich formuliert werden sollte.

Beispiel: `double kreisumfang = Math.PI * radius; //berechne Kreisumfang`

Der Programmausschnitt ist korrekter Java-Code, aber es wird etwas anderes berechnet, als eigentlich gewollt war. Bei diesem Fehlertyp ist eine (automatische) Unterstützung in der Fehlervermeidung oder Fehlerfindung sehr schwer möglich. Die später behandelte Verifikation und ähnlich motivierte Ansätze gehen aber in diese Richtung.

Was man für ein gegebenes spezifiziertes Problem gerne hätte, wäre ein korrekt formulierter Algorithmus, der für alle zulässigen Eingabewerte korrekte Ausgabewerte produziert. In diesem Zusammenhang unterscheidet man im Rahmen der Qualitätssicherung im Software-Entwicklungsprozess verschiedene Begriffe und damit einhergehend auch verschiedene Ansätze und Aussagemöglichkeiten. Man spricht vom **Testen**, wenn ein gegebenes Programm  $p$  mit Hilfe von Testdatensätzen dahingehend überprüft werden soll, ob das Programm der Spezifikation genügt. Man spricht von **Verifikation**, wenn bewiesen werden soll, dass das Programm  $p$  immer (für alle erlaubten Eingabedaten und alle möglichen Fälle) der Spezifikation genügt.

Es gibt zwar Verfahren, um Programme systematisch mit Eingabedaten zu testen (vergleiche nachfolgenden Abschnitt), aber weder ist ein Testen aller möglichen Eingabedaten im Normalfall praktikabel noch lässt sich durch Testen die Korrektheit für alle erlaubten Eingabedaten formal nachweisen. Denn durch Testen erkennt man die dabei aufgetretenen Fehler, aber beim Testen nicht aufgetretene Fehler können natürlich nach wie vor existieren und beispielsweise mit anderen Datensätzen gegebenenfalls dann auftreten. E. Dijkstra hat es in einer Turing Award Lecture so formuliert [Dij72], dass man durch Testen zwar das Vorkommen, nicht aber das Fehlen von Fehlern (und damit die Korrektheit) nachweisen kann.

### Beispiel 17.1:

Die Behauptung ist, dass der nachfolgende Programmausschnitt für einen Eingabewert  $n \in \mathbb{N}, n < 1000$  als Ergebnis in der Variablen `resultat` den Wert  $n^2$  liefert.

```
17-1 int n = ...;
17-2 int resultat;
17-3 if (n < 3) {
17-4     resultat = n * n;
17-5 } else {
17-6     resultat = 4711;
17-7 }
```

Testet man den Programmausschnitt mit dem Eingabewert 1 und 2 und vergleicht den Resultatwert mit dem korrekten Wert für diese Operation, so kommt in beiden Fällen das richtige Ergebnis heraus. Man weiss damit aber nicht, ob das Programm auch für  $n = 3, 4, 5, \dots$  korrekt oder nicht korrekt arbeitet. Testet man anschließend auch mit dem Wert 3, so liefert dieser Test das falsche Ergebnis, man hat also den Nachweis der Inkorrektheit für diesen Wert geführt. Eine Aussage über  $n = 4, 5, \dots$  hat man aber immer noch nicht. ◆

## 17.1 Testen von Programmen

An dieser Stelle soll nur ein kurzer Einstieg in das Testen von Programmen gegeben werden, da der Schwerpunkt auf der Verifikation im nachfolgenden Kapitel liegt. Das Testen eines Programms dient dazu, für bestimmte Eingabedaten die korrekte Ausführung des Programms zu überprüfen. Oft ist die Hoffnung mit einem (oder mehreren) Test verbunden, dass damit das Programm auch für weitere Eingabedaten das korrekte Ergebnis liefert.

Man unterscheidet:

- *Black Box Tests*: hierbei wird nur das äußere Verhalten eines Algorithmus getestet, ohne die Implementierung / die internen Details zu kennen. Es wird hierbei also gegen eine spezifizierte Schnittstelle getestet.
- *White Box Tests*: hierbei wird der Programmcode mit einbezogen, also Kenntnisse über die interne Arbeitsweise der zu testenden Implementierung.

Listing 17.1: Beispielprogramm 1.

```

17-1  /**
17-2   * Testprogramm 1
17-3   */
17-4 public class Test1 {
17-5     public static void main(String [] args) {
17-6         int a = Integer.parseInt(args[0]);
17-7         long b = 0;
17-8         for(int i=0; i<a; i++) {
17-9             b += 2;
17-10        }
17-11        System.out.println(b);
17-12    }
17-13 }
```

Listing 17.2: Beispielprogramm 2.

```

17-1  /**
17-2   * Testprogramm 2
17-3   */
17-4 public class Test2 {
17-5     public static void main(String [] args) {
17-6         int a = Integer.parseInt(args[0]);
17-7         int b = Integer.parseInt(args[1]);
17-8         long c = 0;
17-9         for(int i=0; i<a; i++) {
17-10            for(int j=0; j<b; j++) {
17-11                c++;
17-12            }
17-13        }
17-14        System.out.println(c);
17-15    }
17-16 }
```

Um Programme zu testen sind zwei Sachen zusätzlich zum Programm notwendig: (zulässige oder auch unzulässige) Testdaten sowie die dazu erwarteten Ergebnisse. Das Programm wird dann mit einem Testdatensatz ausgeführt und die Resultatdaten mit den erwarteten Ergebnisdaten verglichen. Stimmen diese Daten überein, so geht man von einem korrekten Programm zumindest für diesen Testdatensatz aus. Stimmen die Daten nicht überein, so wird das Programm als fehlerhaft angesehen. Ein einzelner Testdatensatz ist aber meist nicht sehr aussagekräftig.

### **Beispiel 17.2:**

In Listing 17.1 ist ein Beispiel gegeben, an dem die Problematik des Testens zum Ausschluss von Fehlern verdeutlicht werden soll. Gegeben ist folgendes Programm: Die Behauptung ist, dass dieses Programm für einen Eingabewert  $a$  den Resultatwert  $b = 2 \cdot a$  für alle natürlichen Zahlen im Intervall  $[0, 1.000.000.000)$  berechnet. Um die Korrektheit des Programms für *alle* zulässigen Eingabewerte zu testen, müssen also 1 Milliarde Tests durchgeführt werden, denn so viele verschiedene Eingabewerte sind möglich.

In Listing 17.2 ist eine etwas veränderte Variante gezeigt. Um hier die Aussage zu überprüfen, dass für zwei beliebige Eingabezahlen im Bereich  $[0, 1.000.000.000)$  *immer* das korrekte Ergebnis berechnet wird (welches ist es?), müssten hier bei einem vollständig den Eingabedatenraum überdeckenden Testdatensatz

$1.000.000.000 \cdot 1.000.000.000 = 10^{18}$  Testfälle betrachtet werden. Wenn jeder Testfall in 1 Mikrosekunde durchgeführt werden könnte, so würde der komplette Test damit  $10^{18} * 10^{-6} = 10^{12}$  Sekunden oder annähernd 32.000 Jahre dauern. ♦

Oft verwendet man Testdatensätze, die als repräsentativ oder als besonders kritisch angesehen werden. Es existieren Testverfahren, die durch systematisches Testen mit solchen ausgesuchten Testdatensätzen möglichst viele Fehler zu finden hoffen. Bekannte Testmethoden mit teilweise unterschiedlichen Ausrichtungen sind etwa verschiedene Formen von Überdeckungsverfahren (Pfadüberdeckung, Anweisungsüberdeckung, Bedingungsüberdeckung). Bei diesen Verfahren werden nicht alle möglichen Fälle getestet, sondern lediglich eine sinnvolle Auswahl der möglichen Fälle. Für Details hierzu sei auf die entsprechende weiterführende Literatur verwiesen.

Ein extremer Ansatz aus dem Bereich des Software Engineerings ist das *Test-Driven Development*, bei dem das Testen von Software integraler Bestandteil der Software Entwicklung ist, und hier auch bereits im sehr frühen Entwicklungsstadium eingesetzt wird. *Bevor* mit der Implementierung des eigentlichen Programms begonnen wird, werden erst Tests (Datensätze und erwartete Ergebnisse) für die zu entwickelnden Teilkomponenten spezifiziert und programmiert. Während der eigentlichen Programmierung werden dann kontinuierlich die Tests zur Überprüfung der inkrementell neu entwickelten Software eingesetzt. Hierzu dienen dann sogenannte **Unit Tests**, im Java-Bereich gibt es das verbreitete Werkzeug *Junit[jun]* für diesen Zweck, mit dem viele der mit diesem Ansatz anfallenden Aufgaben (teil-)automatisiert werden können.

## 17.2 Verifikation von Programmen

Die Verifikation von Programmen ist ein wichtiges Thema, das in Zukunft auch sicherlich noch größere Bedeutung bekommen wird. Der im Nachfolgenden vorgestellte Ansatz von C.A.R. Hoare ist dabei *ein Ansatz*, die Korrektheit eines Programms, das heißt die Einhaltung einer Spezifikation zu *beweisen*. Weitere populäre Ansätze in dieser Richtung mit anderen Vorgehensweisen und/oder Zielsetzungen sind etwa das Model Checking[Mod] zum vollautomatischen Nachweis von Programmeigenschaften sowie Beweisverfahren basierend auf Petri Netzen zum Nachweis von Eigenschaften in nebenläufigen Systemen (zum Beispiel der Nachweis der Verklemmungsfreiheit). *Design by Contract* (DbC)[Mey00] ist ein Ansatz ursprünglich eingeführt mit der objektorientierten Programmiersprache Eiffel, der die Idee des Beweisens über Schnittstellenspezifikationen als Teil des Software-Entwicklungsprozesses auffasst.

C.A.R. Hoare hat 1969 ein Axiomen- und Regelsystem entworfen [Hoa69], mit dem man das Ein-/Ausgabeverhalten von Programmen beschreiben und über das sich gegebenenfalls die Korrektheit bestimmter Programme beweisen lassen kann. Der Grundgedanke dabei ist, das Ein-/Ausgabeverhalten eines Programms durch logische Formeln zu Beginn und am Ende des Programms zu beschreiben. Man benötigt also verlässliche Angaben zum Beispiel zu den Eingabewerten (Beispiel:  $x \geq 0 \wedge y \geq 0$ ) und man gibt dann an (und beweist dies), worauf sich ein Benutzer des Programms, der sich an die Vorbedingung hält, nach Ausführung des Programms verlassen kann (Beispiel:  $z = x + y$ ). Den Beweis zu einem solchen Ein-/Ausgabeverhalten des Gesamtprogramms führt man dabei zurück auf das Ein-/Ausgabeverhalten von Programmteilen bis hinunter zu den Basisbausteinen einer Programmiersprache, zu denen Grundregeln in dem Beweissystem existieren müssen. Man spricht in einem solchen Fall auch von einer strukturellen Induktion, weil der Beweisgang zu einem konkreten Programm immer über den strukturellen Aufbau des Programms geführt wird.

Logische Formeln, die bestimmte Bedingungen ausdrücken sollen, sind dabei:

- In einer **Vorbedingung** (englisch *precondition*) wird festgelegt, unter welchen Bedingungen der Programmcode genutzt werden kann.

- Eine **Nachbedingung** oder Zusicherung (englisch *postcondition*) legt fest, was unter Einhaltung der Vorbedingung nach Ausführung des Programms gelten wird.
- Eine **Invariante** ist eine Bedingung, die immer an einer bestimmten Programmstelle gelten muss.

### Beispiel 17.3:

Sei *addieren* eine Methode zur Berechnung der Addition zweier Werte  $x, y \in \mathbb{N}, x \geq 0, y \geq 0$ . Die Wirkung (Korrektheit) dieses Programms kann durch zwei logische Formeln  $p$  als Vorbedingung und  $q$  als Nachbedingung wie folgt beschrieben werden:

$$\{p\} z = \text{addieren}(x, y) \{q\}$$

mit

$$p \equiv (x, y \in \mathbb{N} \wedge x \geq 0 \wedge y \geq 0)$$

und

$$q \equiv z = x + y$$

Die Formulierung  $\{x, y \in \mathbb{N} \wedge x \geq 0 \wedge y \geq 0\} z = \text{addieren}(x, y) \{z = x + y\}$  bedeutet demnach, wenn vor der Ausführung von *addieren* die logische Formel  $p \equiv (x, y \in \mathbb{N} \wedge x \geq 0 \wedge y \geq 0)$  als Vorbedingung wahr ist, dann ist nach der Ausführung des Programms *addieren* die logische Formel  $q \equiv z = x + y$  als Nachbedingung auch wahr, also in  $z$  die Summe der Eingabewerte  $x$  und  $y$  enthält. ♦

Allgemein wird die Korrektheit eines Programms  $S$  ausgedrückt durch eine solche **Korrektheitsformel**  $\{p\} S \{q\}$ , die das Ein-/Ausgabeverhalten des Programms  $S$  spezifiziert. Eine Korrektheitsformel  $\{p\} S \{q\}$  heißt **partiell korrekt**, wenn jede terminierende Berechnung von  $S$ , die in einem  $p$ -Zustand beginnt, in einem  $q$ -Zustand endet.  $\{p\} S \{q\}$  heißt **total korrekt**, wenn jede Berechnung von  $S$ , die in einem  $p$ -Zustand beginnt, terminiert und ihr Endzustand  $q$  erfüllt. Im Falle der partiellen Korrektheit wird also über nicht-terminierende Berechnungen von  $S$  keine Aussage gemacht. In einem Programm ohne Schleifen stimmt die partielle Korrektheit mit der totalen Korrektheit überein, das ein solches Programm auf jeden Fall terminiert.

#### 17.2.1 Zusicherungen in Struktogrammen

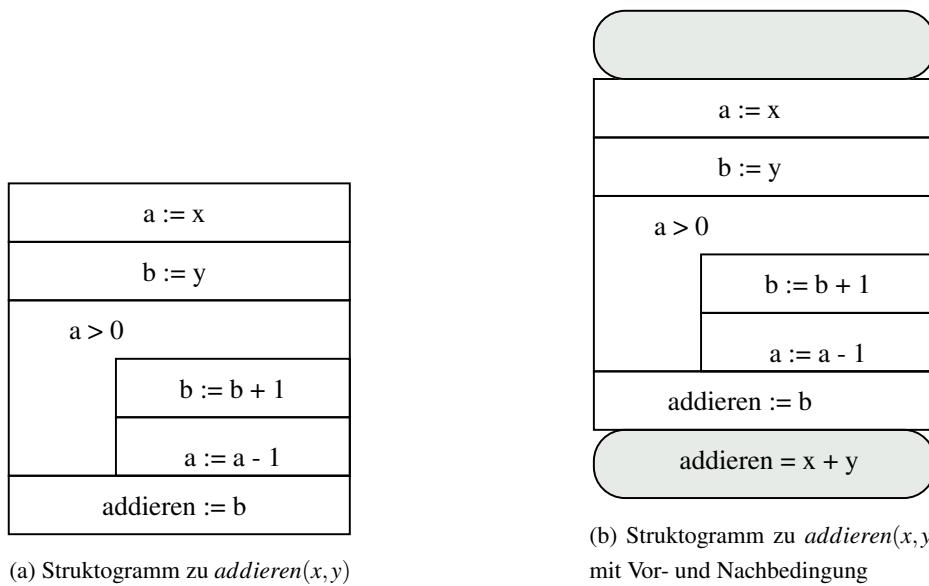
Bevor die formalen Regeln des Hoarschen Kalküls und deren konkrete Anwendung angegeben werden, sollen zuvor etwas informaler die Denkweise in solchen Beweisgängen anhand einfacher Problemstellungen erläutert werden. Die Idee ist dabei, dass man Struktogramme dahingehend erweitert, dass vor und nach jedem Anweisungsblock jeweils eine logische Aussage steht. Ausgehend von der Aussage vor dem Block muss man aufgrund des ausgeführten Inhalts in dem Block zu der nachfolgenden logischen Aussage kommen.

In einem ersten einfachen Beispiel sollen zwei natürliche Zahlen  $x$  und  $y$  addiert werden, wobei die Addition beliebiger Zahlen auf die Addition und Subtraktion einer Zahl mit 1 zurückgeführt werden soll. Man will also zeigen, dass das Programm für erlaubte Eingabewerte  $x, y \in \mathbb{N}$  korrekt  $x + y$  berechnet. Die entsprechende Java-Methode ist wie folgt. Hierbei soll ein möglicher Bereichsüberlauf der Zahlendarstellung ignoriert werden.

```

17-1 public static int addieren(int x, int y) {
17-2     int a = x;
17-3     int b = y;
17-4     while(a > 0) {
17-5         b = b + 1;
17-6         a = a - 1;
17-7     }

```

Figure 17.1: Struktogramme zu  $\text{addieren}(x,y)$ 

```

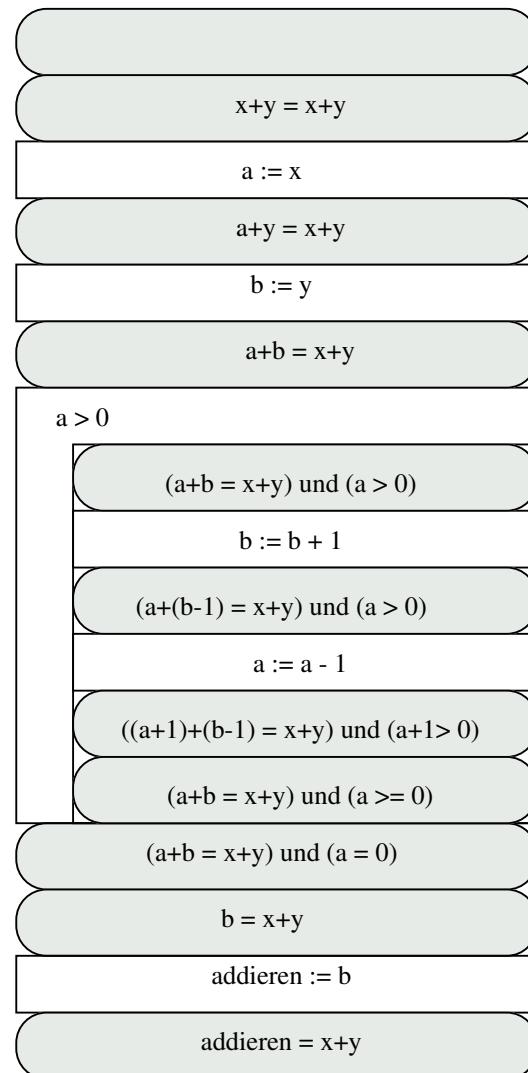
17-8   int z = b;
17-9   return z;
17-10 }

```

Abbildung 17.1a zeigt das dazugehörige Struktogramm. Die Methode besteht aus einer Sequenz von drei Anweisungen, zwei Zuweisungen zu Beginn und einer Schleife als letzte Anweisung. Der Schleifenrumpf besteht wiederum aus einer Sequenz von zwei Zuweisungen. Unsere Behauptung ist nun, dass für alle  $x, y \in \mathbb{N}$  die Methode  $x + y$  berechnet, das heißt, nach Ausführung der letzten Zuweisungszeile gilt:  $z = x + y$ . Das um die Vor- und Nachbedingung erweiterte Struktogramm ist in Abbildung 17.1b zu sehen. Zur visuellen Unterscheidung werden die Zusicherungen in runden Kästchen notiert und hinterlegt.

Die Vor- und Nachbedingung ist zwar aufgestellt, damit aber noch nicht *bewiesen*, dass für alle  $x, y \in \mathbb{N}$ , durch die Ausführung der Methode am Ende (falls die Berechnung terminiert) die Nachbedingung gilt. Dies gilt es jetzt zu zeigen. Und dazu muss man die Argumentation über die Struktur des vorliegenden Programms führen. Zur Erinnerung: Die Methode besteht aus einer Sequenz von drei Anweisungen, zwei Zuweisungen zu Beginn und einer Schleife als letzte Anweisung. Der Schleifenrumpf besteht wiederum aus einer Sequenz von zwei Zuweisungen. Um zu einer Gesamtaussage zu diesem Programm zu kommen, muss die Argumentation bei unserem Vorgehen also darauf aufbauen, dass gewisse Aussagen zu den einzelnen Zuweisungen (Elementaranweisungen) gemacht werden können, darauf aufbauend kann man eine Aussage zu einer kompletten Sequenz machen und darauf aufbauend dann eine Aussage zum gesamten Methodenrumpf.

Ausgehend von der leeren Vorbedingung kann man daraus die (triviale) wahre Aussage  $x + y = x + y$  schließen, die uns im Weiteren als Ausgangspunkt nützlich sein wird. Aufgrund der ersten Zuweisung ( $a = x$ ) kann man auch die linke Seite der Formel entsprechend modifizieren ( $a + y = x + y$ ), da man aufgrund der Semantik der Zuweisung weiß, dass nach deren Ausführung die Werte von  $a$  und  $x$  identisch sind. Analog kann man bei der zweiten Zuweisung ( $b = y$ ) vorgehen, so dass vor der Schleife gilt:  $a + b = x + y$ . Dies wird die **Schleifeninvariante** sein, eine Aussage, die vor und nach jeder Ausführung des Schleifenrumpfes gültig ist. Wenn der Schleifentest  $a > 0$  wahr ist, so tritt man in den Schleifenrumpf ein, man weiß aufgrund des Schleifentests zusätzlich aber auch, dass  $a > 0$  gelten muss. Aufgrund der Zuweisung  $b = b + 1$  können wir die Bedingung  $a + b = x + y$  modifizieren zu  $a + (b - 1) = x + y$ , da der neue Wert von  $b$  dem um eins erhöhten

Figure 17.2: Struktogramme zu `addieren(x,y)` mit Zusicherungen

alten Wert entspricht. Analog geht man für die Zuweisung  $a = a - 1$  vor, so dass nach dieser Zuweisung die Aussage ist:  $(a + 1) + (b - 1) = x + y$ , was vereinfacht werden kann zu  $a + b = x + y$ , was gerade der gewählten Schleifeninvarianten entspricht, die beim Schleifentest gelten muss (damit diese Formel nach Definition eine Schleifeninvariante ist).

Der Schleifenrumpf kann beliebig oft ausgeführt werden, jeweils zu Beginn und zum Ende des Schleifenrumpfes gilt die Schleifeninvariante, unabhängig von den ursprünglichen Werten von  $x$  und  $y$ . Verlässt man die Schleife, so weiß man einerseits, dass die Schleifeninvariante nach wie vor gilt, zusätzlich aber auch, dass die Bedingung des Schleifentests ( $a > 0$ ) falsch sein muss, weil die Schleifenausführung ja abgebrochen wurde. Das heißt, an dieser Stelle muss insgesamt gelten:  $a + b = x + y \wedge \neg(a > 0)$ , oder anders ausgedrückt  $a + b = x + y \wedge a = 0$ , oder wiederum vereinfacht  $b = x + y$ . Die nachfolgende Zuweisung überträgt diese Aussage auf  $z$ , so dass man insgesamt bewiesen hat, dass für zulässige Eingaben  $x, y \in \mathbb{N}, x, y \geq 0$  in der Vorbedingung nach Ausführung des Programms die Nachbedingung  $z = x + y$  gelten muss, falls die Berechnung terminiert.

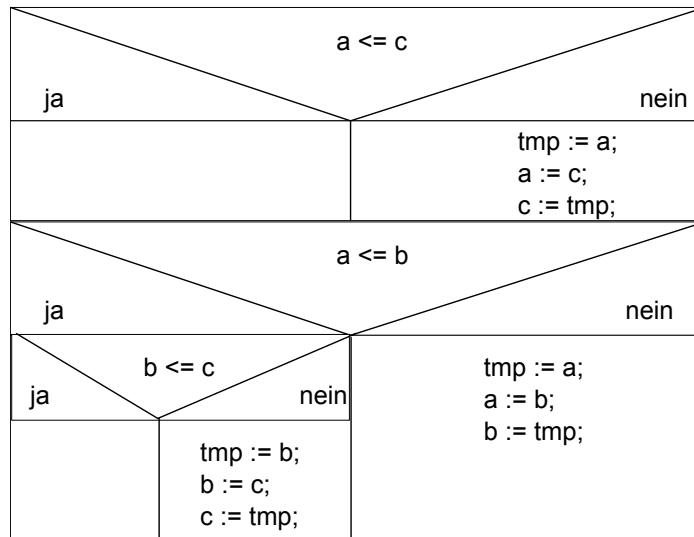
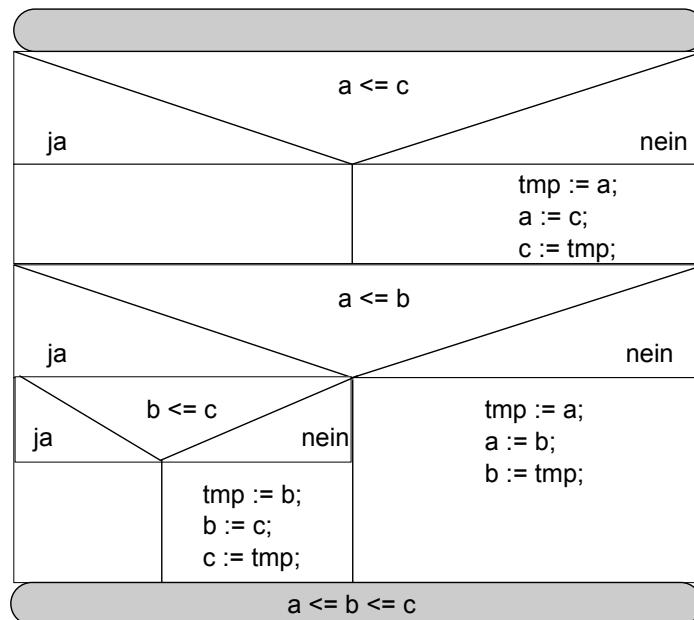
Als zweites Beispiel soll die Korrektheit eines kleinen Programms bewiesen werden, dass den Inhalt dreier ganzzahliger Variablen  $a, b, c$  so vertauscht, dass anschließend gilt:  $a \leq b \leq c$ .

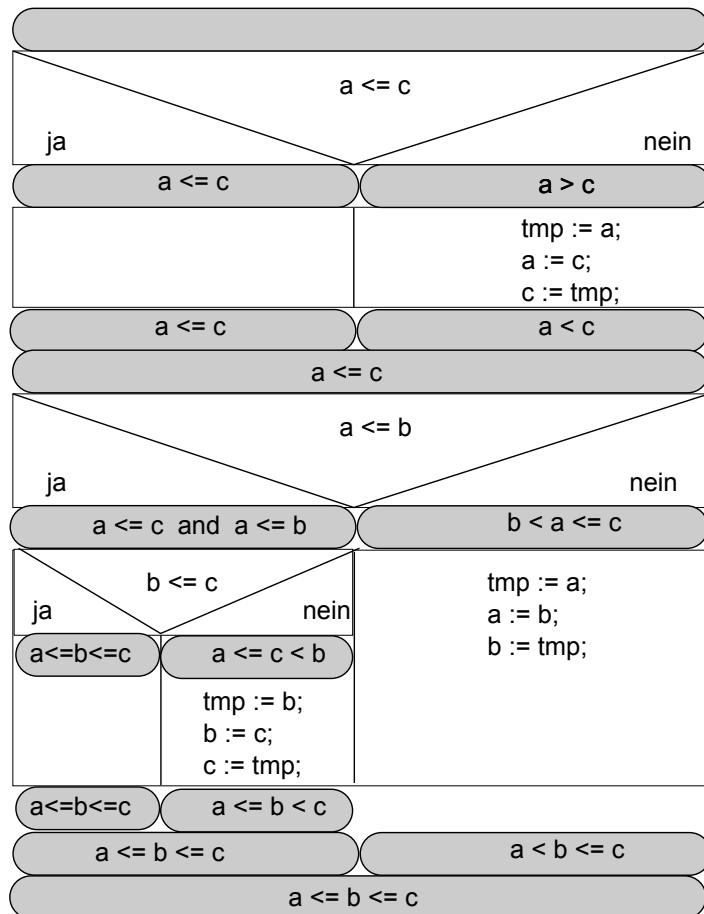
```

17-1 public static void tauschen(int a, int b, int c) {
17-2     if(a <= c) {
17-3         // nichts zu tun
17-4     } else {
17-5         // vertausche a<->c
17-6         int tmp = a;
17-7         a = c;
17-8         c = tmp;
17-9     }
17-10
17-11    if(a <= b) {
17-12        if(b <= c) {
17-13            // nichts zu tun
17-14        } else {
17-15            // vertausche b<->c
17-16            int tmp = b;
17-17            b = c;
17-18            c = tmp;
17-19        }
17-20    } else {
17-21        // vertausche a<->b
17-22        int tmp = a;
17-23        a = b;
17-24        b = tmp;
17-25    }
17-26    // hier gilt: a <= b <= c
17-27 }
```

Das Struktogramm dazu ist in Abbildung 17.3a, die Annotation mit Vor- und Nachbedingung in Abbildung 17.3b. Wie dort zu sehen ist, ist die Nachbedingung  $a \leq b \leq c$ , die Vorbedingung ist leer, diese Aussage gilt also für alle Werte  $a, b, c$ .

Die logischen Aussagen ergeben sich analog zu dem Vorgehen im ersten Beispiel direkt. Anmerkungswürdig sind die Aussagen nach einer if-Anweisung, beispielsweise für den ersten Block mit der Bedingung  $a \leq c$ . Aufgrund der if-Bedingung kommt man bei Durchlaufen des leeren then-Zweiges zu den Aussage  $a \leq c$ . Durchläuft man den else-Zweig, so kommt man aufgrund der negierten if-Bedingung  $\neg(a \leq c) \equiv a > c$  und dem Vertauschen der beiden Variableninhalte zu der Nachbedingung für diesen Zweig  $a < c$ . An dieser Stelle ist nun ein wichtiger Punkt, auch im Hinblick auf das nachfolgende Kapitel des Hoareschen Beweissystems. Denn aus diesen beiden verschiedenen Resultataussagen für den then- und für den else-Zweig muss man zu

(a) Struktogramm zu  $\text{tauschen}(a, b, c)$ (b) Struktogramm zu  $\text{tauschen}(a, b, c)$  mit Vor- und NachbedingungFigure 17.3: Struktogramme zu  $\text{tauschen}(a, b, c)$

Figure 17.4: Struktogramme zu `tauschen(a,b,c)` mit Zusicherungen

einer *gemeinsamen Aussage* für die gesamte if-Anweisung kommen, mit der man den Beweisgang fortsetzen kann. Für diesen Fall ist eine solche Aussage  $a \leq c$ , die sowohl aus der Nachbedingung des then-Zweigs als auch aus der Nachbedingung des else-Zweigs herleitbar ist. Der Leser mache sich klar, dass die Aussage  $a < c$  dies nicht leisten würde.

### 17.2.2 Hoare-Kalkül

Das **Hoare-Kalkül**[Hoa69] ist ein System zum Nachweis der partiellen Korrektheit deterministischer und sequentieller Programme. Eine zu beweisende Formel in diesem System hat die Form  $\{p\} s \{q\}$ , wobei  $s$  das Programm ist und  $p$  und  $q$  Formeln der Prädikatenlogik erster Stufe[Sch00] sind. Vereinfacht ausgedrückt sind diese Formeln syntaktisch aufgebaut wie logische Ausdrücke (Konstanten, Variablen, logischen Operatoren,...) und liefern bei einer Berechnung wahr oder falsch als Resultat der Berechnung. Man möchte also mit so einer Formel ausdrücken, dass aus der wahren Aussage  $p$  nach Ausführung des Programms folgt, dass die Aussage  $q$  ebenfalls (und auch immer) wahr ist. Man kann sich einen solchen Nachweis weniger abstrakt vorstellen als einen Beweis einer Aussage der Form  $p \xrightarrow{S} q$ , also einer logischen Implikation. Der Beweis kann vorwärts gerichtet erfolgen, dass aus der Gültigkeit der Vorbedingung  $p$  die Gültigkeit der Nachbedingung  $q$  bewiesen wird. Nach den Regeln der Logik kann einer solcher Beweis aber auch dadurch geführt werden, dass man aus der Gültigkeit der Nachbedingung keine Widersprüche in der Vorbedingung findet ( $p \Rightarrow q$  ist äquivalent zu  $\neg q \Rightarrow \neg p$ ).

Der Vorteil des Hoareschen Kalküls ist die Tatsache, dass man die partielle Programmkorrektheit induktiv über den syntaktischen Aufbau eines Programms beweisen kann. Das System besteht aus einer Menge von Regeln, die spezifisch zur zugrundeliegenden Programmiersprache sind. Alle Regeln in dem Kalkül haben die Form

$$\frac{\text{Prämissen}}{\text{Konklusion}}$$

**Prämissen** sind Korrektheitsformeln oder Zusicherungen. Eine Konklusion ist eine Korrektheitsformel. Aus der Gültigkeit der Prämissen kann man auf die Gültigkeit der Konklusion schließen, das heißt wenn man die Gültigkeit der Prämissen nachgewiesen hat, so gilt auch die entsprechende Konklusion. Regeln ohne Prämissen nennt man **Axiome**.

Es muss allerdings auch auf eine in der Praxis wichtige und prinzipielle Einschränkung des Hoareschen axiomatischen Ansatzes hingewiesen werden. In [Jr.79] wurde nämlich bewiesen, dass dieser axiomatische Ansatz nicht das Gewünschte leistet, wenn gewisse Eigenschaften in der zugrundeliegenden Programmiersprache erlaubt sind. Wenn eine Programmiersprache etwa bestimmte Formen der Rekursion erlaubt, so garantiert das Axiomensystem nicht mehr die Korrektheit der Aussagen.

Da die Hoareschen Regeln im Detail immer spezifisch zum Aufbau einer Programmiersprache sind, soll hier eine sehr kleine und überschaubare Miniprogrammiersprache festgelegt werden, die aber die wesentlichen üblichen Konstrukte realer Programmiersprachen enthält. Die Syntax lehnt sich an Java an, ist aber nicht identisch. Für eine andere Programmiersprache würden die Regeln im Detail eventuell anders lauten, da das Kalkül ja an den syntaktischen Aufbau einer Sprache gebunden ist. Die hier zugrundegelegte einfache Beispielprogrammiersprache enthält folgende Konstrukte:

- Variablen, Konstanten und Ausdrücke, zum Beispiel  $x, y, \dots$  und  $0, 4711, \dots$  und  $x+1$ .
- leere Anweisung: **skip**
- Zuweisung:  $u = t$  für eine Variable  $u$  und einen Ausdruck  $t$
- Sequenz: Anweisung ; Anweisung
- Selektion: **if** Bedingung **then** Anweisung **else** Anweisung **endif**

- Iteration: **while** Bedingung **do** Anweisung **enddo**

Ausdrücke, die bei der Zuweisung, Selektion und Iteration vorkommen, sind mit der üblichen Notation möglich, darauf wird hier nicht weiter speziell eingegangen. Nicht erlaubt sind in Ausdrücken jedoch Operationen mit Seiteneffekten, wie etwa

```
if  $a++ < 0$  then  $x = 1$  else  $x = 2$  endif
```

#### Beispiel 17.4:

Das folgende Programm in dieser Programmiersprache berechnet  $z = x + y$  für zwei Eingabewert  $x, y \in \mathbb{N}$ .

```

17-1   a = x;
17-2   b = y;
17-3   while a > 0 do
17-4       b = b + 1;
17-5       a = a - 1
17-6   enddo;
17-7   z = b

```

Die Axiome und Regeln für die syntaktischen Konstrukte dieser einfachen Programmiersprache sind jetzt zusammengefasst in Abbildung 17.5 aufgeführt (vergleiche die Definitionsregeln der Sprache). Nachfolgend sind diese Regeln im Einzelnen dann beschrieben.

#### 1. Regel zur leeren Anweisung

$$\frac{\{\}}{\{p\} \text{ skip } \{p\}}$$

Die Regel zur leeren Anweisung hat eine leere Prämisse (deshalb spricht man dann auch von einem Axiom) und durch die Ausführung der leeren Anweisung bleibt die Vorbedingung in der Nachbedingung erhalten. Oder anders ausgedrückt: wenn eine bestimmte Bedingung gilt und die leere Anweisung wird ausgeführt, dann gilt diese Bedingung anschließend noch immer, da ja nichts in der Anweisung verändert wurde.

#### Beispiel 17.5:

Man will beweisen, dass, wenn vor der skip-Anweisung die Bedingung  $x > 0$  gilt, so wird durch die Ausführung der leeren Anweisung diese Aussage nicht verändert und als Nachbedingung gilt ebenfalls  $x > 0$ .

Behauptung:

$$\{x > 0\} \text{ skip } \{x > 0\}$$

Beweis: Die direkte Anwendung der Regel zur leeren Anweisung mit  $p \equiv x > 0$  liefert direkt die Behauptung. ♦

#### 2. Zuweisungsregel

$$\frac{\{\}}{\{p[u/t]\} u = t \{p\}}$$

wobei  $p[u/t]$  bedeutet: Alle Vorkommen von  $u$  werden in der Bedingung  $p$  durch  $t$  ersetzt. Man kann sich diese Ersetzung so vorstellen, dass man die Formel  $p$  hinschreibt, alle Vorkommen von  $u$

1. Regel zur leeren Anweisung

$$\frac{\{\}}{\{p\} \text{ skip } \{p\}}$$

2. Zuweisungsregel

$$\frac{\{\}}{\{p[u/t]\} u = t \{p\}}$$

3. Sequenzregel

$$\frac{\begin{array}{c} \{p\} S_1 \{q\} \\ \{q\} S_2 \{r\} \end{array}}{\{p\} S_1; S_2 \{r\}}$$

4. Selektionsregel

$$\frac{\begin{array}{c} \{p \wedge B\} S_1 \{q\} \\ \{p \wedge \neg B\} S_2 \{q\} \end{array}}{\{p\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ endif } \{q\}}$$

5. Iterationsregel

$$\frac{\{p \wedge B\} S \{p\}}{\{p\} \text{ while } B \text{ do } S \text{ enddo } \{p \wedge \neg B\}}$$

6. Konsequenzregel

$$\frac{\begin{array}{c} p \Rightarrow p_1 \\ \{p_1\} S \{q_1\} \\ q_1 \Rightarrow q \end{array}}{\{p\} S \{q\}}$$

Figure 17.5: Übersicht über die Hoareschen Regeln.

durchstreicht und stattdessen an diese Stelle  $t$  hinschreibt.  $u$  kann hierbei eine beliebige Variable der Programmiersprache sein,  $t$  ein beliebiger Ausdruck der Sprache. Auch dieses Axiom hat eine leere Prämissen. Über dieses Axiom lassen sich also Aussagen machen über die Auswirkung einer Zuweisung eines Wertes (der Wert des Ausdrucks  $t$ ) an eine Variable ( $u$ ).

Diese Regel wendet man üblicherweise *rückwärts* an, was einen Anfänger oft irritiert. Es wird also von einer gegebenen Nachbedingung aufgrund der Regel ermittelt, welche Vorbedingung sich durch die Anwendung der Regel ergibt. Bei einer Regelanwendung wird demnach die Formel der Nachbedingung als Vorbedingung ""kopiert"" und in dieser Formel dann alle Vorkommen der Variablen durch den Ausdruck ersetzt. Die Ersetzung ist rein syntaktisch, also mit Blatt und Papier streicht man alle Vorkommen der Variablen durch und schreibt stattdessen den Ausdruck dahin.

### Beispiel 17.6:

Behauptung:

$$\{y < 32\} y = y + 23 \{y < 55\}$$

Beweis: Das Programm besteht aus genau einer Zuweisung. Dehalb kann die Zuweisungsregel direkt (rückwärts) angewandt werden.

$$\begin{aligned} & \{y < 55[y/y+23]\} \quad y = y + 23 \{y < 55\} \\ \Leftrightarrow & \{y + 23 < 55\} \quad y = y + 23 \{y < 55\} \\ \Leftrightarrow & \{y < 32\} \quad y = y + 23 \{y < 55\} \end{aligned}$$

Damit ist die Behauptung  $\{y < 32\} y = y + 23 \{y < 55\}$  bewiesen.

Zur Erläuterung: Im ersten Schritt wird die Regel angewandt (Nachbedingung kopieren zur Vorbedingung, Ersetzung der Variablen  $y$  durch den Ausdruck  $y + 23$ ). Im nachfolgenden Schritt wird die Ersetzung angewandt, also alle Vorkommen von  $y$  durch  $y + 23$  ersetzt. Und im letzten Schritt wird der resultierende Ausdruck in der Vorbedingung vereinfacht. ♦

### 3. Sequenzregel

$$\frac{\begin{array}{c} \{p\} S_1 \{q\} \\ \{q\} S_2 \{r\} \end{array}}{\{p\} S_1; S_2 \{r\}}$$

Die Sequenzregel ist ähnlich einem Transitivgesetz. Wenn zwei beliebige Anweisungen  $S_1$  und  $S_2$  vorliegen und die Nachbedingung von  $S_1$  gleich der Vorbedingung von  $S_2$  ist, so kann man Aussagen machen über das Hintereinanderausführen beider Anweisungen.

### Beispiel 17.7:

Behauptung:

$$\{x < 1\} x = x + 1; x = x + 2 \{x < 4\}$$

Beweis: Zur Durchführung des Beweises muss zuerst die Struktur des Programms analysiert werden, da über diese Struktur der Beweis geführt wird. Das Programm besteht aus einer Sequenz von zwei Zuweisungen. Daraus ergibt sich zwangsläufig, dass man im Beweisgang einmal die Sequenzregeln anwenden muss und zweimal die Zuweisungsregel.

Um die Sequenzregel für das Gesamtprogramm anwenden zu können (also die Aussage der Konklusion dieser Regel machen zu können), müssen wir erst die Aussagen in der Prämisse der Sequenzregel beweisen. Laut Prämisse muss man im vorliegenden Fall also die beiden Teilaussagen beweisen:

- 1)  $\{x < 1\} x = x + 1 \{q\}$
- 2)  $\{q\} x = x + 2 \{x < 4\}$

für ein geeignetes  $q$ , das man sich aussuchen kann. Wenn man diese beiden Teilaussagen in der Prämisse der Sequenzregel bewiesen hat, kann man die Konklusion der Regel ziehen. Zum Beweis der beiden Teilaussagen geht man aufgrund der vorkommenden Zuweisungen rückwärts vor. Man nimmt die Nachbedingung  $\{x < 4\}$  der zweiten Teilaussage, schaut aufgrund der Zuweisungsregel, welche Vorbedingung  $q$  sich daraus ergibt, geht mit diesem  $q$  als Nachbedingung in die erste Teilaussage und schaut aufgrund der Zuweisungsregel wiederum, welche Vorbedingung sich daraus ergibt.

Teilbeweis 1 (Anwendung der Zuweisungsregel auf die zweite Zuweisung):

$$\begin{aligned} & \{x < 4[x/x+2]\} \quad x = x + 2 \{x < 4\} \\ \Leftrightarrow & \{x+2 < 4\} \quad x = x + 2 \{x < 4\} \\ \Leftrightarrow & \{x < 2\} \quad x = x + 2 \{x < 4\} \end{aligned}$$

Die Vorbedingung  $q$ , die man hiermit gefunden hat, lautet also  $q \equiv x < 2$ .

Teilbeweis 2 (Anwendung der Zuweisungsregel auf die erste Zuweisung mit der gefundenen Nachbedingung  $q$ ):

$$\begin{aligned} & \{x < 2[x/x+1]\} \quad x = x + 1 \{x < 2\} \\ \Leftrightarrow & \{x+1 < 2\} \quad x = x + 1 \{x < 2\} \\ \Leftrightarrow & \{x < 1\} \quad x = x + 1 \{x < 2\} \end{aligned}$$

Damit hat man bis jetzt folgende Teilaussagen bewiesen, die die Prämissen der Sequenzregel im konkreten Fall sind:

- 1)  $\{x < 1\} x = x + 1 \{x < 2\}$
- 2)  $\{x < 2\} x = x + 2 \{x < 4\}$

Damit kann man die Konklusion der Sequenzregel anwenden und erhält die Aussage über das Gesamtprogramm:

$$\{x < 1\} x = x + 1; x = x + 2 \{x < 4\}$$

womit die Behauptung bewiesen ist. ♦

#### 4. Selektionsregel

$$\frac{\begin{array}{c} \{p \wedge B\} S_1 \{q\} \\ \{p \wedge \neg B\} S_2 \{q\} \end{array}}{\{p\} \text{ if } B \text{ then } S_1 \text{ else } S_2 \text{ endif } \{q\}}$$

Man möchte eine Gesamtaussage zur Selektion machen, unabhängig davon, ob man nun den then- oder else-Zweig durchläuft. Um zu so einer Gesamtaussage in der Konklusion zu kommen, muss man vorher in der Prämisse zeigen, dass mit einer gemeinsamen Vorbedingung  $p$  sowohl der then-Zweig mit der Anweisung  $S_1$  zu der Nachbedingung  $q$  führt, als auch der else-Zweig mit der Anweisung  $S_2$ . Als Zusatzinformation bekommt man im then-Zweig, dass zusätzlich zur Vorbedingung  $p$  auch

die Bedingung  $B$  wahr sein muss (sonst würde man nicht in den then-Zweig zu  $S_1$  kommen) und im else-Zweig, dass zusätzlich zur Vorbedingung  $p$  die Bedingung  $B$  falsch ist (sonst wäre man nicht in den else-Zweig zu  $S_2$  gekommen).

### Beispiel 17.8:

Behauptung:

$$\{a \neq b\} \text{ if } a \geq b \text{ then } x = a - b \text{ else } x = b - a \text{ endif } \{x > 0\}$$

Beweis: Zur Durchführung des Beweises muss wiederum zuerst die Struktur des Programms analysiert werden. Das Programm besteht aus einer Selektion, deshalb muss letztendlich für die Gesamtaussage auch die Selektionsregeln angewandt werden. Um die Konklusion dieser Regel anwenden zu können, müssen aber vorher die Aussagen der Prämisse bewiesen werden. Im vorliegenden Fall liegen zwei Zuweisungen vor, also muss zwei mal die Zuweisungsregel angewandt werden.

Laut Prämisse muss man im vorliegenden Fall also die beiden Teilaussagen beweisen:

- 1)  $\{(a \neq b) \wedge (a \geq b)\} x = a - b \{x > 0\}$
- 2)  $\{(a \neq b) \wedge \neg(a \geq b)\} x = b - a \{x > 0\}$

Wenn man diese beiden Teilaussagen in der Prämisse der Selektionsregel bewiesen hat, kann man die Konklusion der Regel ziehen. Zum Beweis der beiden Teilaussagen geht man aufgrund der vorkommenden Zuweisungen wiederum jeweils rückwärts vor.

Teilbeweis 1 (Anwendung der Zuweisungsregel auf die erste Zuweisung):

$$\begin{aligned} & \{x > 0[x/a - b]\} \quad x = a - b \{x > 0\} \\ \Leftrightarrow & \{a - b > 0\} \quad x = a - b \{x > 0\} \\ \Leftrightarrow & \{a > b\} \quad x = a - b \{x > 0\} \end{aligned}$$

Teilbeweis 2 (Anwendung der Zuweisungsregel auf die zweite Zuweisung):

$$\begin{aligned} & \{x > 0[x/b - a]\} \quad x = b - a \{x > 0\} \\ \Leftrightarrow & \{b - a > 0\} \quad x = b - a \{x > 0\} \\ \Leftrightarrow & \{b > a\} \quad x = b - a \{x > 0\} \end{aligned}$$

Bis jetzt hat man also bewiesen:

- 1)  $\{a > b\} x = a - b \{x > 0\}$
- 2)  $\{b > a\} x = b - a \{x > 0\}$

Laut Prämisse der Regel wäre aber eigentlich zu zeigen:

- 1)  $\{(a \neq b) \wedge (a \geq b)\} x = a - b \{x > 0\}$
- 2)  $\{(a \neq b) \wedge \neg(a \geq b)\} x = b - a \{x > 0\}$

An dieser Stelle kommt man im Beweisgang nicht weiter, weil die bewiesenen Vorbedingungen anders lauten als die laut Regel einzusetzenden. Später wird mit der Konsequenzregel das Hilfsmittel zur Verfügung gestellt, in dem vorliegenden Fall weiter zu kommen und die hier vorliegenden Prämissen zu beweisen.

Unter der Annahme, dass man diesen Schluss machen kann (Details dazu gleich bei der Konsequenzregel), kann man die Konklusion der Selektionsregel ziehen und erhält damit die Aussage

über das Gesamtprogramm:

$$\{a \neq b\} \text{ if } a \geq b \text{ then } x = a - b \text{ else } x = b - a \text{ endif } \{x > 0\}$$

womit die Behauptung bewiesen ist. ♦

### 5. Iterationsregel

$$\frac{\{p \wedge B\} S \{p\}}{\{p\} \text{ while } B \text{ do } S \text{ enddo } \{p \wedge \neg B\}}$$

Die Iterationsregel erfordert etwas mehr Erläuterung zu ihrer Herleitung. Die Herausforderung bei dieser Regel ist es, dass man anders als bei den bisherigen Regeln eine Aussage zu der Gesamtschleife machen muss, und damit unabhängig von einer festen Anzahl an Iterationen des Schleifenrumpfes, der aus einer beliebigen Anweisung bestehen kann. Der Schleifenrumpf könnte 0 mal, 1 mal oder 100 mal durchlaufen werden, je nachdem, wann die Bedingung  $B$  zum Abbruch der Schleife führt. Die Lösung dieser Problematik ist, dass man in der Prämisse ausgehend von der Vorbedingung  $p$  zeigen muss, dass nach *einmaliger* Ausführung des Schleifenrumpfes diese Bedingung  $p$  immer noch gilt und damit die Vorbedingung für ein nochmaliges Ausführen wieder vorliegt. Gleichzeitig kann man aber in der Vorbedingung zum Schleifenrumpf auch davon ausgehen, dass die Schleifenbedingung  $B$  ebenfalls gilt, denn ansonsten würde der Schleifenrumpf ja nicht ausgeführt werden. In der Prämisse ist also zu zeigen, dass die Bedingung  $p$  durch die Ausführung des Schleifenrumpfes invariant bleibt, also erhalten bleibt. Eine solche Formel  $p$  nennt man auch Schleifeninvariante.

Hat man diese Prämisse gezeigt, so lässt sich damit die Konklusion ziehen, dass bei wahrer Vorbedingung  $p$  die Schleife ausgeführt wird und die Schleifenbedingung  $B$  irgendwann zum Abbruch der Schleife führt. Aufgrund obiger Argumentation muss nach einem Abbruch nachievor die Schleifeninvariante  $p$  wahr sein, aber die Bedingung  $B$  kann nicht mehr wahr sein (weil sie ja für den Abbruch der Schleife gesorgt hat).

In der Praxis kann das Finden einer geeigneten Schleifeninvariante durchaus eine Herausforderung sein, denn mit ihr sind zwei Sachen verbunden. Einerseits muss diese Formel invariant sein, also vor wie nach der Ausführung des Schleifenrumpfes gelten, was oft vermehrte Denkarbeit erfordert. Weiterhin muss die Schleifeninvariante aber auch dergestalt sein, dass man Sie natürlich als Vorbedingung in einem größeren Programm auch herleiten können muss und weiterhin mit der Aussage der Nachbedingung auch den weiteren Beweisgang ausführen können muss.

### Beispiel 17.9:

Behauptung:

$$\{x < y\} \text{ while } x < 10 \text{ do } x = x + 1; y = y + 1 \text{ enddo } \{x < y\}$$

Beweis: Das Programm besteht aus einer Schleife, der Schleifenrumpf aus einer Sequenz von zwei Zuweisungen. Demzufolge muss man zwei Zuweisungsregeln anwenden, eine Sequenzregel und eine Iterationsregel.

Um die Konklusion der Iterationsregel anwenden zu können und damit die Gesamtaussage zu beweisen, muss vorher die Aussage der Prämisse bewiesen werden. Als Schleifeninvariante wählen wir:  $x < y$ .

In der Prämisse ist also zu zeigen:

$$\{(x < y) \wedge (x < 10)\} x = x + 1; y = y + 1 \{x < y\}$$

Dies beweist man mit Hilfe der Sequenzregel, diese wiederum über zweimaliges (rückwärts gerichtetes) Anwenden der Zuweisungsregeln.

Teilbeweis 1 (Anwendung der Zuweisungsregel auf die zweite Zuweisung):

$$\begin{aligned} & \{x < y[y/y+1]\} \quad y = y + 1 \quad \{x < y\} \\ \Leftrightarrow & \quad \{x < y + 1\} \quad y = y + 1 \quad \{x < y\} \end{aligned}$$

Teilbeweis 2 (Anwendung der Zuweisungsregel auf die erste Zuweisung mit der ermittelten Vorbedingung von Teilbeweis 1 als Nachbedingung):

$$\begin{aligned} & \{x < y + 1[x/x+1]\} \quad x = x + 1 \quad \{x < y + 1\} \\ \Leftrightarrow & \quad \{x + 1 < y + 1\} \quad x = x + 1 \quad \{x < y + 1\} \\ \Leftrightarrow & \quad \{x < y\} \quad x = x + 1 \quad \{x < y + 1\} \end{aligned}$$

Bis jetzt hat man also bewiesen:

- 1)  $\{x < y + 1\} \quad y = y + 1 \quad \{x < y\}$
- 2)  $\{x < y\} \quad x = x + 1 \quad \{x < y + 1\}$

Mit Hilfe der Konklusion der Sequenzregel kann man damit nun beweisen:

$$\{x < y\} \quad x = x + 1; y = y + 1 \quad \{x < y\}$$

Eigentlich wäre als Prämisse der Iterationsregel zu zeigen gewesen:

$$\{(x < y) \wedge (x < 10)\} \quad x = x + 1; y = y + 1 \quad \{x < y\}$$

Auch an dieser Stelle benötigt man die Hilfe der nachfolgenden Konsequenzregel (Details gleich) und kann damit dann ebenfalls die Prämisse zeigen.

Unter der Annahme, dass man diesen Schluss machen kann, kann man die Konklusion der Iterationsregel ziehen und erhält damit folgende Aussage über das Gesamtprogramm:

$$\{x < y\} \quad \textbf{while } x < 10 \textbf{ do } x = x + 1; y = y + 1 \textbf{ enddo } \quad \{(x < y) \wedge \neg(x < 10)\}$$

Auch hier benötigt man die nachfolgende Konsequenzregel, da die bis jetzt vorliegend Nachbedingung  $(x < y) \wedge \neg(x < 10)$  nicht exakt der Aussage entspricht, die man eigentlich zeigen wollte ( $x < y$ ). Mit Hilfe der Konsequenzregel ergibt sich aber diese Gesamtaussage. ♦

## 6. Konsequenzregel

$$\frac{\begin{array}{c} p \Rightarrow p_1 \\ \{p_1\} S \{q_1\} \\ q_1 \Rightarrow q \end{array}}{\{p\} S \{q\}}$$

Diese Regel ist nicht an die Syntax der Sprache gebunden, sondern basiert auf einer logischen Umformung. Die Bedeutung dieser Regel ist, dass Vorbedingungen verstärkt, Nachbedingungen abgeschwächt werden können. Eine Verstärkung der Vorbedingung geschieht z.B. durch das Weglassen von Und-Termen oder das Hinzufügen von Oder-Termen. Eine Abschwächung der Nachbedingung

geschieht z.B. durch das Hinzufügen von Und-Termen oder das Weglassen von Oder-Termen. Insbesondere dürfen aber  $p$  und  $q$  auch durch gleiche oder äquivalente Ausdrücke ersetzt werden. Wenn man also beispielsweise eine Nachbedingung  $q_1$  für  $S$  bewiesen hat, müsste aber eigentlich  $q$  beweisen, so hilft an der Stelle die Konsequenzregel weiter. Wenn nämlich gilt  $q_1 \Rightarrow q$ , so darf man auch in der Gesamtaussage der Konklusion  $q$  anstatt  $q_1$  verwenden. Analoges gilt für die Vorbedingung.

Diese Regel wird – wie in den Beispielen oben gesehen – oft dann benötigt, wenn die Vor- oder Nachbedingung nicht mit der Aussage übereinstimmt, die man im Gesamtbeweisgang an der entsprechenden Stelle hergeleitet hat. Durch die Implikation  $p \Rightarrow p_1$  beziehungsweise  $q_1 \Rightarrow q$  darf man auch in der Konklusion  $p$  und  $q$  verwenden.

### Beispiel 17.10:

Behauptung:

$$\{x = 3\} x = 5 \{x = 5\}$$

Beweis: Das Programm besteht aus einer Zuweisung, also kann die Zuweisungsregel direkt (rückwärts) angewandt werden, was liefert:

$$\begin{aligned} & \{x = 5[x/5]\} \quad x = 5 \{x = 5\} \\ \Leftrightarrow & \{5 = 5\} \quad x = 5 \{x = 5\} \end{aligned}$$

An dieser Stelle hat man ein eher technisches Problem im Beweisgang. Denn man hat gezeigt, dass unter der Annahme  $5 = 5$  (diese Annahme ist *immer* wahr) sich die Aussage ergibt. Eigentlich wollte man aber zeigen:

$$\{x = 3\} x = 5 \{x = 5\}$$

Es gilt aber:  $(x = 3) \Rightarrow (5 = 5)$  und somit kann man mit Hilfe der Konsequenzregel (im Vorbedingungsteil) die Gesamtaussage ziehen:

$$\{x = 3\} x = 5 \{x = 5\}$$

was zu zeigen war. ♦

### Beispiel 17.11:

Im Beweisgang der Selektionsregel war man an die Stelle gekommen, wo bewiesen war:

- 1)  $\{a > b\} x = a - b \{x > 0\}$
- 2)  $\{b > a\} x = b - a \{x > 0\}$

Laut Prämisse der Regel wäre aber eigentlich zu zeigen:

- 1)  $\{(a \neq b) \wedge (a \geq b)\} x = a - b \{x > 0\}$
- 2)  $\{(a \neq b) \wedge \neg(a \geq b)\} x = b - a \{x > 0\}$

Mit Hilfe der Konsequenzregel und

- 1)  $((a \neq b) \wedge (a \geq b)) \Rightarrow (a > b)$
- 2)  $((a \neq b) \wedge \neg(a \geq b)) \Leftrightarrow ((a \neq b) \wedge (a < b)) \Rightarrow (a < b)$

gelten die beiden Aussagen der Prämisse ebenfalls. ♦

**Beispiel 17.12:**

Im Beweisgang der Selektionsregel war man an die Stelle gekommen, wo bewiesen war:

$$\{x < y\} \quad x = x + 1; y = y + 1 \quad \{x < y\}$$

Eigentlich wäre als Prämisse der Iterationsregel zu zeigen gewesen:

$$\{(x < y) \wedge (x < 10)\} \quad x = x + 1; y = y + 1 \quad \{x < y\}$$

Auch hier hilft die Konsequenzregel im Beweisgang weiter, denn es gilt:

$$((x < y) \wedge (x < 10)) \Rightarrow (x < y)$$

und damit gilt die zu zeigende Aussage. ♦

**Beweisführung** Die Beweisrichtung ist letztendlich aus Programmsicht von oben nach unten zu sehen. Insbesondere aufgrund der Anwendung der Zuweisungsregel ist es aber meist sinnvoll, dass man für die Beweisfindung, zumindest für Teile des Programms, rückwärts, also von unten nach oben vorgeht.

**Beispiel: Ganzzahlige Division mit Rest** Die Anwendung der Hoareschen Beweisregeln soll nachfolgend an einem umfangreicherem und etwas komplexerem Beispiel durchgeführt werden. Dem Leser wird empfohlen, zuerst den groben Beweisgang nachzuvollziehen und dann anschließend die Details des Beweises durchzuarbeiten.

Als Problem ist gegeben, den ganzzahligen Anteil der Division (in der Variablen  $q$ ) zweier ganzer Zahlen  $x, y \in \text{NatuerlicheZahlen}$  und den Rest der Division (in der Variablen  $r$ ) zu bestimmen. Folgendes Programmstück *div* berechnet die korrekten Werte in *q* und *rest*. Dies ist zumindest bis jetzt die Behauptung, die erst noch bewiesen werden muss.

```

17-1 q = 0;
17-2 r = x;
17-3 while r > y do
17-4   r = r - y;
17-5   q = q + 1
17-6 enddo

```

Dieses kleine Programm soll mit *div* bezeichnet werden. Zuerst muss man sich überlegen, wie man eine Aussage bezüglich der korrekten Ausführung formulieren kann, also die Vor- und Nachbedingung für das Programm angeben. Als Vorbedingung wählt man  $x \geq 0 \wedge y \geq 0$ . Es sei an dieser Stelle schon darauf hingewiesen, dass der Teil der Vorbedingung  $y \geq 0$  bewusst so gewählt wurde und am Ende des Beweisgangs noch einmal diskutiert wird.

Die Nachbedingung erfordert etwas mehr Aufwand. Nimmt man beispielweise die Zahlen  $x = 7$  und  $y = 3$ , so wäre auch das Resultat  $q = 1$  und  $r = 4$  korrekt Teiler und Rest in unserem Sinne, denn umgekehrt gilt ja  $1 \cdot 3 + 4 = 7$ . Was man aber will ist die Bestimmung des *maximalen* Quotienten, also im Beispiel müsste  $q = 2$  und  $r = 1$  sein. Der Quotient ist maximal, wenn sich der verbliebene Rest nicht noch weiter durch  $y$  teilen lässt, oder anders ausgedrückt, wenn  $r < y$ .

Nach diesen Vorüberlegungen ergibt sich damit als Nachbedingung

$$\{(q \cdot y + r = x) \wedge (0 \leq r < y)\}$$

und insgesamt die

**Behauptung:**

$$\{(x \geq 0) \wedge (y \geq 0)\} \text{ div } \{(q \cdot y + r = x) \wedge (0 \leq r < y)\}$$

**Beweis:** Um den Beweis zu führen ist die erste Überlegung wiederum, welche Struktur das Programm hat, da darüber der Beweis zu führen ist. Das Programm besteht aus einer Sequenz von drei Anweisungen, zwei Zuweisungen zu Beginn und einer Iteration als dritte Anweisung. Die Iteration hat als Schleifenrumpf wiederum eine Sequenz von zwei Zuweisungen. Man wird also (mindestens) vier mal die Zuweisungsregel anwenden, ein mal die Iterationsregel und zwei mal die Sequenzregel.

Um die Korrektheit nachzuweisen, muss für die Iterationsregel eine geeignete Schleifeninvariante  $p$  gewählt werden, die wie folgt gewählt wird:

$$p \equiv ((q \cdot y + r = x) \wedge (r \geq 0))$$

Diese Überlegung ist von zentraler Bedeutung für den Beweisgang. Denn wie in der Diskussion der Iterationsregel bereits erläutert, muss bei der Wahl der Schleifeninvarianten drei Sachen berücksichtigt werden:

- 1) Diese Schleifeninvariante muss sich aus dem Beweisgang vor der Schleife herleiten lassen
- 2) Diese Schleifeninvariante muss – der Name sagt es – invariant zum Schleifenrumpf sein
- 3) Ausgehend von der Schleifeninvarianten und der negierten Schleifenbedingung in der Konklusion der Iterationsregel muss sich der weitere Beweisgang nach der Schleife daraus herleiten lassen

Mit der gewählten Schleifeninvarianten lässt sich dann folgendes zeigen, was dann auch unser Plan für den Beweis ist:

1. Die Schleifeninvariante  $p$  gilt vor erstmaliger Ausführung der **while**-Schleife, d.h.

$$\{(x \geq 0) \wedge (y \geq 0)\} q = 0; r = x \{p\}$$

2.  $p$  bleibt bei einmaliger Ausführung des Schleifenrumpfes wahr:

$$\{p \wedge (r \geq y)\} r = r - y; q = q + 1 \{p\}$$

3.  $p$  und  $\neg B$  wird die gewünschte Nachbedingung implizieren.

Nun also zu den Beweisen dieser drei Teilaussagen.

Teilbehauptung 1: Die Schleifeninvariante  $p$  gilt vor erstmaliger Ausführung der **while**-Schleife, d.h.

$$\{(x \geq 0) \wedge (y \geq 0)\} q = 0; r = x \{p\}$$

Beweis: Um diese Behauptung nun zu beweisen, muss – der Struktur dieses Teilprogramms folgend – die Sequenzregel angewendet werden und in der Prämisse dieser Regel zwei mal die Zuweisungsregel, wie gehabt rückwärts angewandt.

Es gilt (Zuweisungsregel auf die zweite Anweisung rückwärts angewandt):

$$\begin{aligned} & \{((q \cdot y + r = x) \wedge (r \geq 0))[r/x]\} \quad r = x \{((q \cdot y + r = x) \wedge (r \geq 0))\} \\ \Leftrightarrow & \{((q \cdot y + x = x) \wedge (x \geq 0))\} \quad r = x \{((q \cdot y + r = x) \wedge (r \geq 0))\} \end{aligned}$$

Weiterhin gilt (ermittelte Vorbedingung der zweiten Anweisung als Nachbedingung der ersten Zuweisung und Zuweisungsregel rückwärts angewandt):

$$\begin{aligned} & \{((q \cdot y + x = x) \wedge (x \geq 0))[q/0]\} \quad q = 0 \{((q \cdot y + x = x) \wedge (x \geq 0))\} \\ \Leftrightarrow & \{((0 \cdot y + x = x) \wedge (x \geq 0))\} \quad q = 0 \{((q \cdot y + x = x) \wedge (x \geq 0))\} \\ \Leftrightarrow & \{(x = x) \wedge (x \geq 0)\} \quad q = 0 \{((q \cdot y + x = x) \wedge (x \geq 0))\} \end{aligned}$$

Mit Hilfe der Konsequenzregel und  $((x \geq 0) \wedge (y \geq 0)) \Rightarrow (x \geq 0)$  ergibt sich die Aussage der Teilbehauptung 1.

Teilbehauptung 2: die Schleifeninvariante  $p$  bleibt bei einmaliger Ausführung des Schleifenrumpfes wahr:

$$\{p \wedge (r \geq y)\} r = r - y; q = q + 1 \{p\}$$

Beweis: Es liegt eine Sequenz mit zwei Zuweisungen vor. Dementsprechend ist unser Vorgehen analog zu vorher.

Es gilt (Zuweisungsregel auf die zweite Anweisung rückwärts angewandt):

$$\begin{aligned} & \{((q \cdot y + r = x) \wedge (r \geq 0)) [q/q + 1]\} \quad q = q + 1 \{((q \cdot y + r = x) \wedge (r \geq 0))\} \\ \Leftrightarrow & \{(((q + 1) \cdot y + r = x) \wedge (r \geq 0))\} \quad q = q + 1 \{((q \cdot y + r = x) \wedge (r \geq 0))\} \end{aligned}$$

Es gilt weiterhin (Zuweisungsregel auf die erste Anweisung rückwärts angewandt mit eben ermittelte Vorbedingung als Nachbedingung):

$$\begin{aligned} & \{(((q + 1) \cdot y + r = x) \wedge (r \geq 0)) [r/r - y]\} \quad r = r - y \{(((q + 1) \cdot y + r = x) \wedge (r \geq 0))\} \\ \Leftrightarrow & \{(((q + 1) \cdot y + (r - y) = x) \wedge (r - y \geq 0))\} \quad r = r - y \{(((q + 1) \cdot y + r = x) \wedge (r \geq 0))\} \\ \Leftrightarrow & \{((q \cdot y + y + r - y = x) \wedge (r \geq y))\} \quad r = r - y \{(((q + 1) \cdot y + r = x) \wedge (r \geq 0))\} \\ \Leftrightarrow & \{((q \cdot y + r = x) \wedge (r \geq y))\} \quad r = r - y \{(((q + 1) \cdot y + r = x) \wedge (r \geq 0))\} \end{aligned}$$

Mit der Konklusion der Sequenzregel erhalten wir:

$$\{((q \cdot y + r = x) \wedge (r \geq y))\} r = r - y; q = q + 1 \{((q \cdot y + r = x) \wedge (r \geq 0))\}$$

Zu zeigen war aber:

$$\{((q \cdot y + r = x) \wedge (r \geq 0) \wedge (r \geq y))\} r = r - y; q = q + 1 \{((q \cdot y + r = x) \wedge (r \geq 0))\}$$

Auch hier kommt man mit Hilfe der Konsequenzregel angewandt auf die Vorbedingung zum Ziel, denn es gilt:

$$((q \cdot y + r = x) \wedge (r \geq 0) \wedge (r \geq y)) \Rightarrow ((q \cdot y + r = x) \wedge (r \geq y))$$

womit diese Teilbehauptung auch bewiesen ist.

Teilbehauptung 3:  $p$  und  $\neg B$  impliziert die gewünschte Nachbedingung dann auch die Gesamtaussage.

Beweis: Als Nachbedingung der Iteration bekommt man ( $p \wedge \neg B$ ):

$$\begin{aligned} & (q \cdot y + r = x) \wedge (r \geq 0) \wedge \neg(r > y) \\ \Leftrightarrow & (q \cdot y + r = x) \wedge (r \geq 0) \wedge (r \leq y) \end{aligned}$$

Bemerkungen:

1. Der obige Beweis zeigt die partielle, aber nicht totale Korrektheit. Das Programm terminiert nämlich nicht für  $y = 0$ .
2. Die totale Korrektheit von  $div$ , das heißt dass dieses Programm auch zusätzlich noch terminiert, muss getrennt nachgewiesen werden. Gilt  $y > 0$ , so kann man zeigen, dass  $r$  bei jedem Schleifendurchlauf um einen Wert echt größer 0 (den Wert von  $y$ ) abnimmt. Über diesen Ansatz lässt sich dann auch ein Beweis zur Terminierung und damit insgesamt zur totalen Korrektheit führen.

## 17.3 Zusammenfassung und Hinweise

### Literaturhinweise

In [Gel01] wird die Thematik sehr schön erläutert.

### Verstehen

Testen dient dem Auffinden von Fehlern. Über den Beweis der Korrektheit eines Programms garantiert man, dass *jede* Programmausführung mit zulässigen Argumenten korrekt (gegenüber der Spezifikation) ist.

### Kurz und knapp merken

- Man unterscheidet die Fehlerarten syntaktischer, semantischer und logischer Fehler.
- Alle syntaktischen Fehler können von einem Compiler erkannt werden, manche semantischen und keine logischen Fehler.
- Das erschöpfende Testen aller Eingabemöglichkeiten ist meistens unrealistisch.
- Einige Beweisverfahren zur Korrektheit von Programmen basieren auf Zusicherungen. Ausgehend von einer Vorbedingung beweist man, dass eine bestimmte Nachbedingung immer gelten muss.
- Eine Invariante ist eine Bedingung, die immer gelten muss. Ein Beispiel dazu ist eine Schleifeninvariante, die zu Beginn jeder Iteration gelten muss.
- Das Hoareschen Kalkül basiert auf Regeln, die Aussagen zu den Grundelementen einer Sprache machen.
- Der Beweis zu einem Programm erfolgt in Form eines strukturellen Beweisgangs, indem man die Struktur des Programms analysiert und darüber einen Beweisgang mit Hilfe der Regeln durchführt.
- Bei zusammengesetzten Konstrukten wie etwa einer Sequenz, die auf anderen Sprachkonstrukten wie etwa einer Anweisung aufsetzen, muss man zuerst eine Aussage zu den Basiskonstrukten zeigen (Prämissen der Regel), um dann anschließend eine Gesamtaussage tätigen zu können (Konklusion der Regel).

### Häufige Fehler

Das A und O bei der Anwendung des Hoareschen Kalküls ist das Erkennen der Struktur eines vorliegenden Programms. Denn darüber wird der Beweis geführt. Wenn man dies nicht berücksichtigt oder nicht erkennt, wird man keinen Beweis führen können.

### Übungsfragen

- Was ist der Unterschied zwischen Testen und Beweisen von Programmeigenschaften?
- Was ist der Unterschied zwischer partieller und totaler Korrektheit?
- Wie ist das allgemeine Vorgehen in einem Beweisgang nach dem Hoareschen Kalkül?
- Wie würde in Hoareschen Kalkül eine Anweisung der Form `++x` behandelt werden?
- Wie ist der Hoare-Beweisgang zu:  
 $\{x = a\}$   
 $y = x;$   
 $\{y = a\}$

- Wie ist der Hoare-Beweisgang zu:

$$\{x = a\}$$

$$y = x; z = y;$$

$$\{z = a\}$$

- Wie ist der Hoare-Beweisgang zu:

$$\{x = a\}$$

$$y = -x; z = y;$$

$$\{z = -a\}$$

- Wie ist der Hoare-Beweisgang zu:

$$\{x = a\}$$

$$y = x; \text{if } (y > 0) \text{ then } z = y; \text{ else } y = -y; \text{ end}$$

$$\{z = -a\}$$

- Wie ist der Hoare-Beweisgang zu:

$$\{x = a\}$$

$$y = x; \text{if } (y > 0) \text{ then } z = -y; \text{ else } z = y; \text{ end}$$

$$\{z = a\}$$

- Wie ist der Hoare-Beweisgang zu:

$$\{x = a\}$$

$$\text{if } (x > 0) \text{ then } y = x; \text{ else } y = -x; \text{ end}$$

$$\{y > 0\}$$

- Wie ist der Hoare-Beweisgang zu:

$$\{x = a\}$$

$$\text{if } (x > 0) \text{ then } y = -x; \text{ else } y = x; \text{ end}$$

$$\{y < 0\}$$

- Wie ist der Hoare-Beweisgang zu:

$$\{x = a\}$$

$$y = 0; \text{while } (x > 0) x = x - 1; y = y + 1; \text{enddo}$$

$$\{y = a\}$$

- Wie ist der Hoare-Beweisgang zu:

$$\{x = a\}$$

$$y = x; \text{while } (x > 0) x = x - 1; y = y + 1; \text{enddo}$$

$$\{y = 2 * a\}$$

- Wie ist der Hoare-Beweisgang zu:

$$\{x = a\}$$

$$y = 0; \text{while } (x > 0) x = x - 1; y = y + 2; \text{enddo}$$

$$\{y = 2 * a\}$$

- Wie ist der Hoare-Beweisgang zu:

$$\{x \leq a\}$$

$$\text{while } (x < a) x = x + 1; \text{enddo}$$

$$\{x = a\}$$

## Reflektion des Stoffs

- Würde es Sinn machen, für wenige Eingabewerte die Korrektheit eines Programms durch Testen zu ermitteln (alle diese Eingabewerte liefern ein korrektes Resultat) und für den Rest einen Beweisgang über das Hoaresche Kalkül durchführen?

- Gibt es Programme, zu denen man eine "Programminvariante" angeben könnte, analog zu einer Schleifeninvarianten bei Schleifen? Wenn es so wäre, was für eine Rolle würde diese spielen (im Zusammenhang mit Nachbedingungen)?



# Chapter 18

## Programmierparadigmen

Es wurden bereits verschiedene Ansätze vorgestellt, wie man eine Problemlösung in einem Programm darstellen kann. Beispiele waren Programme der URM, der eher klassische Absatz ohne Objekte in Java, der objektorientierte Ansatz zum Programmieren in Java usw. An dieser Szelle sollen einige grundlegende Ansätze und gegenüberstellt werden und so auch ihre Vor- und Nachteile dargestellt werden. Einen generellen Ansatz/Vorgabe zur Programmierung nennt man auch **Programmierparadigma**.

Zur Verdeutlichung der Unterschiede wird in der Beschreibung jedes Programmierparadigmas jeweils eine Beispiel angegeben. Es gibt kein bestes Paradigma, sondern jeder Ansatz hat Vor- und Nachteile, die je nach Problemstellung das eine oder andere Modell günstiger erscheinen lassen. Insofern werden die Beispiele auch jeweils so gewählt, dass Vor- und/oder Nachteile verdeutlicht werden können.

### 18.1 Imperativer Ansatz

Die URM, das von-Neumann-Modell eines Rechners, immer nahmen die Register bzw. der Speicher eine zentrale Rolle ein. Der Zustand eines Rechners oder Rechnermodells war über den Inhalt der Register bzw. des Speichers beschreibbar. Und die Wirkung eines Befehls war letztendlich eine Zustandstransformation, d.h. eine Änderung des Speichers. Variablen in Programmiersprachen bezeichnen dabei nichts anderes als Speicherstellen, Variablen sind also benannte Speicherstellen!

Tritt in einem Programmiermodell der Speicher und die Zustandstransformation des Speichers durch eine Folge von Befehlen in den Vordergrund, so spricht man von imperativer Programmierung (*imperare*: lat. befehlen, herrschen). Ein wesentlicher Grundbaustein imperativer Programmiersprachen ist insofern die Zuweisung **Variablen = Ausdruck**. Variablen bezeichnen passive Objekte, die durch Befehle beeinflusst werden.

Die meisten der heutigen Programmiersprachen sind imperativer Natur oder haben zumindest imperative Elemente. Beispiele: C, C++, Pascal, Basic, Fortran, Java. Den imperativen Teil von Java wurde bereits intensiv betrachtet, so dass hier nicht weiter darauf eingegangen wird.

#### Beispiel 18.1:

Ein Beispiel für den imperativen Ansatz ist das Verfahren **Sieb des Eratosthenes**, das alle Primzahlen bis zu einer vorgegebenen Maximalzahl berechnet (siehe Kapitel 16.1). Das Verfahren ist nach dem Griechen Eratosthenes von Kyrene (ca. 276-194 v.Chr.) benannt. Zur Erinnerung: Eine Primzahl ist nur durch 1 und sich selber teilbar. Die Methode ist einfach, es sei jedoch angemerkt, dass es weitaus effizientere Verfahren zur Primzahlerzeugung gibt. Beginnend mit einer Menge, die alle Zahlen bis zum vorgegebenen Maximalwert enthält, werden sukzessive alle Nichtprimzahlen aus dieser Menge entfernt. Dazu wird sukzessive die kleinste noch nicht betrachtete Zahl der Menge genommen (was eine Primzahl ist) und alle Vielfachen dieser Zahl aus

der Menge entfernt. Dieser Vorgang wird solange wiederholt, wie noch nicht betrachtete Zahlen in der Menge enthalten sind.

Das Sieb, d.h. die Menge mit Zahlen, lässt sich als ein Feld von boolschen Werten anlegen. Ein Eintrag `sieb[wert]` ist `true`, wenn die Zahl `wert` in der Menge enthalten ist, ansonsten ist der Eintrag `false`. Der Einfachheit halber lässt man eine Schleife von 2 bis N (der vorgegebenen Maximalzahl) laufen und schaut für jeden Wert `i` nach, ob `sieb[i] true` ist (noch nicht gestrichen). In Listing 16.1 wurde das vollständige Java-Programm bereits angegeben.

Der Ablauf der Programmes für `maximalzahl=10` ist folgendermassen:

Startzahl	Sieb	Vielfache	Sieb - Vielfache
2	{2,3,4,5,6,7,8,9,10}	{4,6,8,10}	{2,3,5,7,9}
3	{2,3,5,7,9}	{6,9}	{2,3,5,7}
4	{2,3,5,7}	{8}	{2,3,5,7}
5	{2,3,5,7}	{10}	{2,3,5,7}
6	{2,3,5,7}	{}	{2,3,5,7}
7	{2,3,5,7}	{}	{2,3,5,7}
8	{2,3,5,7}	{}	{2,3,5,7}
9	{2,3,5,7}	{}	{2,3,5,7}
10	{2,3,5,7}	{}	{2,3,5,7}



## 18.2 Funktionaler Ansatz

In der Beschreibung des funktionalen Berechnungsmodells in Kapitel 2.5.2 wurde hervorgehoben, *was* berechnet wird, und nicht *wie* eine Berechnung stattfindet. Im funktionalen Programmierparadigma wird dieser Ansatz übernommen. In einem reinen funktionalen Programmierparadigma gibt es keinen Speicher, keine Variablen und somit auch keine Zustandstransformationen mehr. Die Definitionen neuer Funktionen über Basisfunktionen oder bereits definierte Funktionen und die Anwendung von Funktionen (deshalb manchmal auch als **applikative Programmierung** bezeichnet) steht im Vordergrund. Eine Funktionsanwendung liefert einen Wert, der nicht in eine Speicherstelle geschrieben wird (Variablenzuweisung), sondern als Eingabe weiterer Funktionsanwendungen dienen kann (Einsetzungsschema). Selbst das Hauptprogramm ist dann nichts anderes als eine Funktion. Iterationen, im imperativen Programmierparadigma meist durch das Herunter- oder Heraufzählen einer Variablen gekennzeichnet, werden im funktionalen Ansatz durch Rekursion ersetzt.

Ein Nachteil des reinen funktionalen Ansatzes kann man in der Fibonacci-Funktion sehen. Die mathematische Definition der Fibonacci-Funktion ist wie folgt:

$$fib(n) = \begin{cases} 1 & \text{falls } n = 0 \vee n = 1 \\ fib(n-2) + fib(n-1) & \text{sonst} \end{cases}$$

In einer funktionalen Programmiersprache wie ML sieht dies folgendermassen aus:

```

18-1 > fun fib n =
18-2   if n = 0 then 1
18-3   else if n = 1 then 1
18-4   else fib(n-1) + fib(n-2);
18-5 val fib = fn: int -> int
18-6 >

```

Das ML-System erkennt automatisch den Typ der neu definierten Funktion `fib` als `fib : int → int`. Dies geschieht über ein **Unifikationsverfahren**, in dem der Typ hergeleitet wird. Die Auswertung der Fibonacci-Funktion für das Argument 5 bedeutet im rein funktionalen Ansatz, dass zum Beispiel `fib(2)` mehrfach

ausgewertet werden muss. Dies ist jedoch unnötig, wenn man diesen Wert einmal berechnet und ihn zwischenspeichern würde.

ML hat eine für den Programmierer komfortable Mustererkennung. So kann man etwa mit

```
18-1 > fun head (h::t) = h
18-2 > fun tail (h::t) = t
```

zwei Funktionen `head` und `tail` definieren, die angewandt auf eine Folge (in ML Liste genannt) das erste Element bzw. den Rest der Liste liefern. Der doppelte Doppelpunkt bezeichnet in ML die Konkatenation zweier Folgen bzw. eines Elementes mit einer Folge. Das ML-System erkennt anhand des Musters, dass in der Funktionsdefinition mit `h::t` eine Folge gemeint ist, die sich aus einem Kopf und dem Rest der Liste zusammensetzt.

Beispiele für funktionale Programmiersprachen sind Haskell, ML, Miranda und Lisp.

## 18.3 Logikorientierter Ansatz

Im logikorientierten Ansatz wird eine Datenbasis in Form von Fakten geschaffen, eine Menge von (allgemeinen) logischen Regeln (Implikationen) angegeben und darauf aufbauend Anfragen gestartet, die das System mit Hilfe der Fakten und Regeln ableiten oder verifizieren soll.

Eine verbreitete Programmiersprache für diesen logikorientierten Ansatz ist Prolog, in der Fakten als Relationen zwischen Objekten notiert werden. Eine Relation wird in der Form

$$\text{relation}(x_1, \dots, x_n)$$

angegeben, was bedeutet, dass  $x_1, x_2, \dots, x_n$  in dieser Relation stehen. Es können zur Schaffung einer Faktenbasis beliebige Relationen zwischen Objekten angegeben werden.

### Beispiel 18.2:

Ein Beispiel zur Verdeutlichung des logikorientierten Ansatzes in Prolog ist die Verwandtschaftsbeziehung zwischen zwei Personen. Zum Aufbau der Datenbasis (Fakten) werden zwei Relationenpaare definiert:

1. `maennlich(x)` bzw. `weiblich(x)` gibt an, ob das "Objekt"  $x$  männlich bzw. weiblich ist.
2. `sohn(x,y)` bzw. `tochter(x,y)` gibt an, dass  $x$  Sohn bzw. Tochter von  $y$  ist.

Die Fakten für das Beispiel sind wie folgt gegeben (Objekte und Relationen werden in Prolog in Kleinbuchstaben, Variablen in Großbuchstaben):

```
18-1 maennlich (karl).
18-2 maennlich (peter).
18-3 weiblich (karin).
18-4 weiblich (susanne).
18-5 sohn (peter , karl).
18-6 sohn (peter , karin).
18-7 tochter (susanne , karl).
18-8 tochter (susanne , karin).
```

Die Objekte karl und peter sind also in der Relation maennlich enthalten, die Objekte karin und susanne in der Relation weiblich. Weiterhin gelten die beiden sohn- bzw. tochter-Beziehungen.

Neben den Fakten werden allgemein gültige Regeln in Form von Wenn-Dann-Regeln (Klauseln) angegeben, wie man logische Schlußfolgerungen ziehen kann. An dieser Stelle sollen die Zusammenhänge im Beispiel vereinfacht werden, indem nicht alle möglichen Fälle des deutschen Familienrechts betrachtet werden. Ist  $x$

Vater von **y**, so kann man dies ausdrücken durch die Regel, dass **x** männlich ist und entweder **y** ein Sohn oder eine Tochter von **x** ist. Übertragen gilt dies gleichfalls für eine Mutter. Für nichtkinderlose Paare gilt weiterhin die Beziehung, dass **x** Ehefrau von **y** ist, wenn **x** weiblich, **y** männlich ist und beide ein gemeinsames Kind haben.

```

18-1  vater(X,Y) :- maennlich(X), sohn(Y,X).
18-2  vater(X,Y) :- maennlich(X), tochter(Y,X).
18-3  mutter(X,Y) :- weiblich(X), sohn(Y,X).
18-4  mutter(X,Y) :- weiblich(X), tochter(Y,X).
18-5  kind(X,Y) :- sohn(X,Y).
18-6  kind(X,Y) :- tochter(X,Y).
18-7  ehefrau(X,Y) :- weiblich(X), maennlich(Y), (kind(Z,X), kind(Z,Y)).
18-8  ehemann(X,Y) :- weiblich(X), maennlich(Y), (kind(Z,X), kind(Z,Y)).
```

Ein Komma auf der rechten Seite einer Klausel bedeutet eine Und-Verknüpfung der beiden Klauseln rechts und links des Kommas. Oder-Verknüpfungen schreibt man in Form von mehreren Regeln untereinander. Die Reihenfolge der Klauseln ist wichtig, da sie genau in dieser Reihenfolge bei späteren Anfragen durchsucht werden.

Nach Eingabe der Fakten und Regeln kann man nun Anfragen an das System stellen. Anfragen beginnen in Prolog mit **?-**. Beispiele für Anfragen wären jetzt:

```

18-1 ?- sohn(peter, karl).
18-2 ja
18-3 ?- vater(karl, peter).
18-4 ja
18-5 ?- ehemann(karl, karin).
18-6 ja
18-7 ?- ehemann(peter, karin).
18-8 nein
18-9 ?- ehemann(karl, X).
18-10 karin
```

Die erste Anfrage fragt an, ob Peter Sohn von Karl ist. In der internen Datenbank des Prolog-Systems ist das Paar (peter, karl) als Relation gespeichert, also ist die Antwort ja.

In der zweiten Anfrage wird gefragt, ob für die beiden Objekte Karl und Peter die Vater-Relation gilt. Dazu muss überprüft werden, ob Karl in der maennlich-Relation enthalten ist und ob Peter und Karl in der Sohn-Relation enthalten ist. Beides ist der Fall, also ist die Antwort ja.

In der dritten und vierten Anfrage muss überprüft werden, ob eine Person der Ehemann einer zweiten Person ist. Dazu muss im ersten Fall überprüft werden, ob für einen beliebigen Wert von Z die Klausel

`ehemann(X,Y) :- weiblich(Y), maennlich(X), (kind(Z,X), kind(Z,Y)).`

für X=peter und Y=karin erfüllt ist.

Die letzte Anfrage schließlich fragt an, für welche Werte von **x** die Relation **ehemann** erfüllt ist. Eine Variable auf der linken Seite einer Klausel bedeutet, dass diese Variable jeden Wert annehmen kann, für den diese Klausel gilt. Eine freie Variable auf der rechten Seite einer Klausel bedeutet, dass sie für mindestens einen Wert gelten muss. Eine Klausel der Form

`ehemann(X,Y) :- weiblich(Y), maennlich(X), (kind(Z,X), kind(Z,Y)).`

bedeutet also (hat als Wert) alle Werte für **x** und **y**, für die ein **z** existiert, so dass **y** weiblich, **x** männlich und **z** ist Kind von **x** und Kind von **y**.



In Prolog werden sogenannte **Horn-Klauseln** verwendet, die die Form haben:

$$x_1 \wedge x_2 \wedge \dots \wedge x_n \Rightarrow y$$

was in Prolog in der bereits verwendeten Form notiert wird:

```
18-1 y :- x1 , x2 , ... , xn
```

## 18.4 Zusammenfassung und Hinweise

### Verstehen

Es gibt grundlegend verschiedene Programmieransätze. Der objektorientierte Ansatz ist ein vielfach genutzter Ansatz.

### Kurz und knapp merken

- Beim imperativen Ansatz steht die sukzessive Veränderung des Zustands (die Werte aller Variablen) im Vordergrund. Die Reihenfolge wird durch Kontrollflussanweisungen gesteuert.
- Beim objektorientierten Ansatz steht die Modellierung der Realität durch Objekte im Vordergrund. Die Objekte kommunizieren über Methodeaufrufe miteinander.
- Im funktionalen Ansatz ist die Definition von Funktionen und deren Anwendung das Ziel. Im rein funktionalen Ansatz gibt es keine Variablen zur Speicherung von Werten. lediglich Funktionsparameter zur Benennung von Übergabewerten.
- Im logikorientierte Ansatz werden deklarativ Regeln und Fakten angegeben. Eine Anfrage versucht ein solches System dann anhand der Fakten und Regeln zu beantworten.



# Appendix A

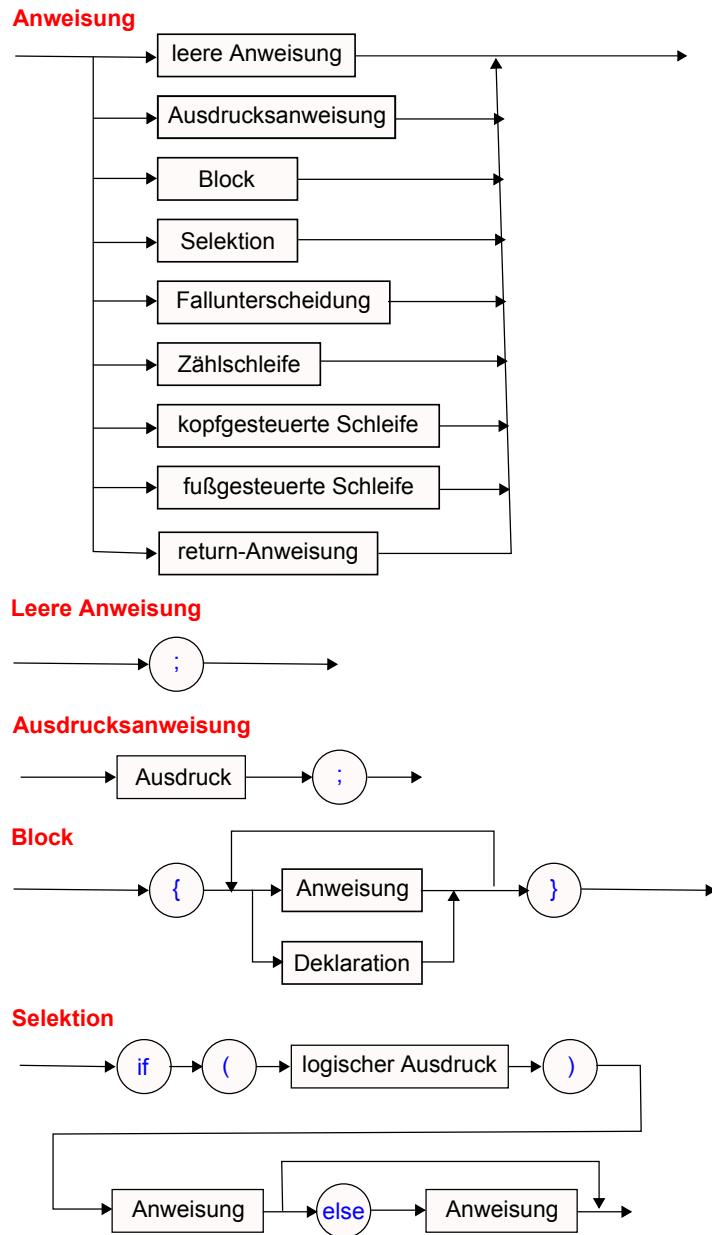
## Notation

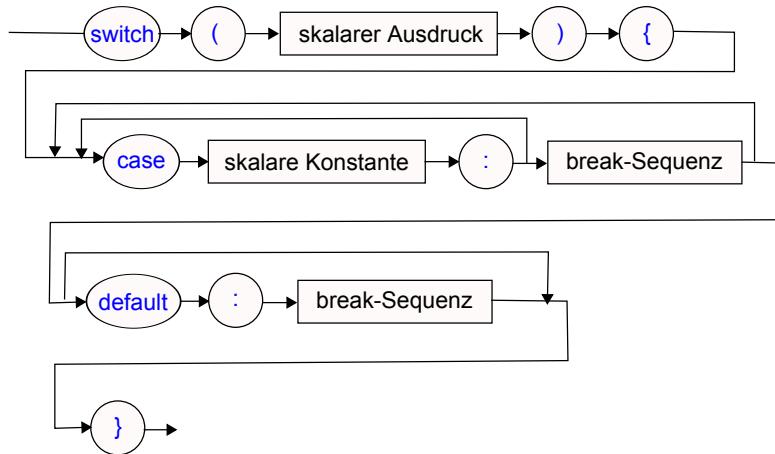
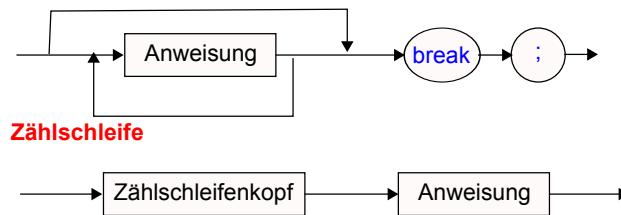
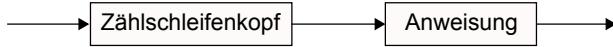
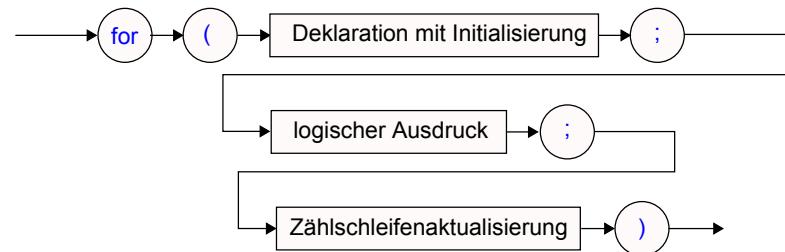
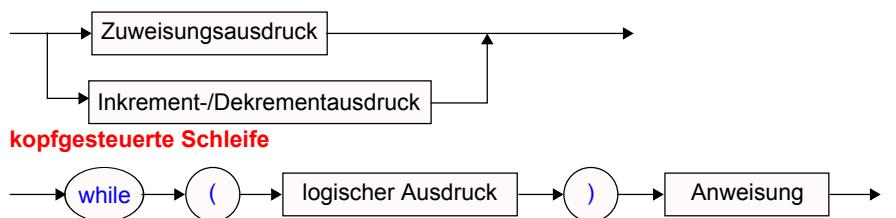
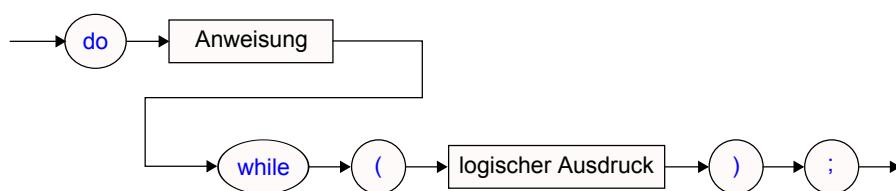
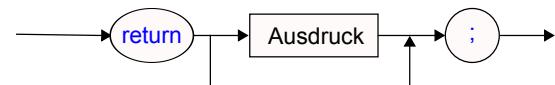
Notation	Bedeutung
$x := y$	$x$ wird durch $y$ definiert
$x : \Leftrightarrow y$	$x$ wird durch $y$ definiert
$\forall x$	für alle $x$
$\exists x$	ex existiert ein $x$
$M_1 \cup M_2$	Mengenvereinigung von $M_1$ und $M_2$
$M_1 \cap M_2$	Mengenschnitt von $M_1$ und $M_2$
$M_1 \setminus M_2$	Mengendifferenz (alle Elemente aus $M_1$ abzüglich den Elementen aus $M_2$ )
$M_1 \subseteq M_2$	$M_1$ ist Teilmenge von $M_2$ (Gleichheit eingeschlossen)
$M_1 \subset M_2$	$M_1$ ist echte Teilmenge von $M_2$
$\Sigma^*$	Menge aller Worte über $\Sigma$
$\Sigma^+$	Menge aller nichtleeren Worte über $\Sigma$ ( $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ )



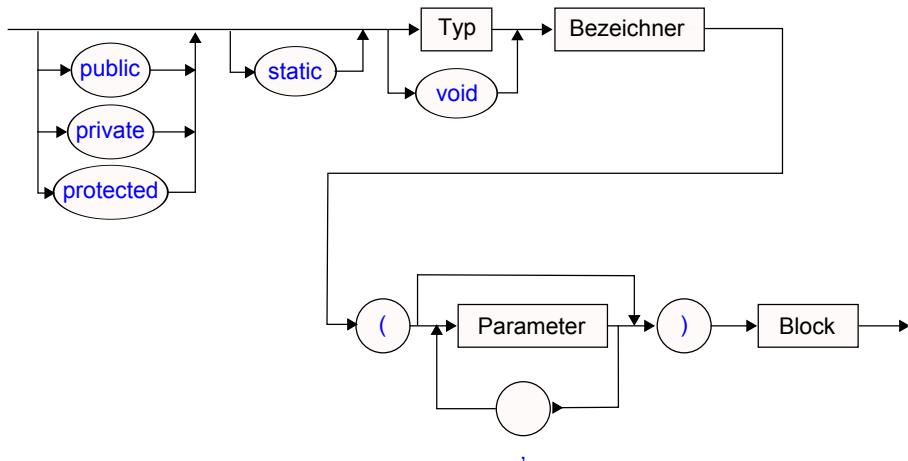
## Appendix B

# Syntaxdiagramme



**Fallunterscheidung****break-Sequenz****Zählschleife****Zählschleifenkopf****Zählschleifenaktualisierung****fußgesteuerte Schleife****return-Anweisung**

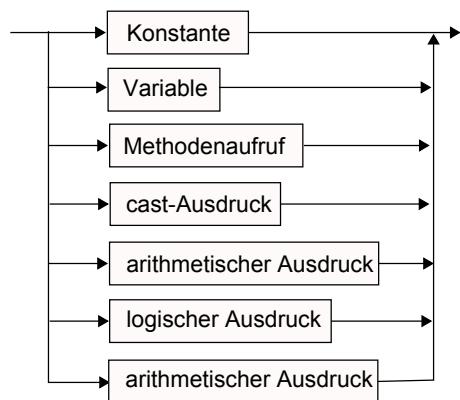
### Methodendefinition

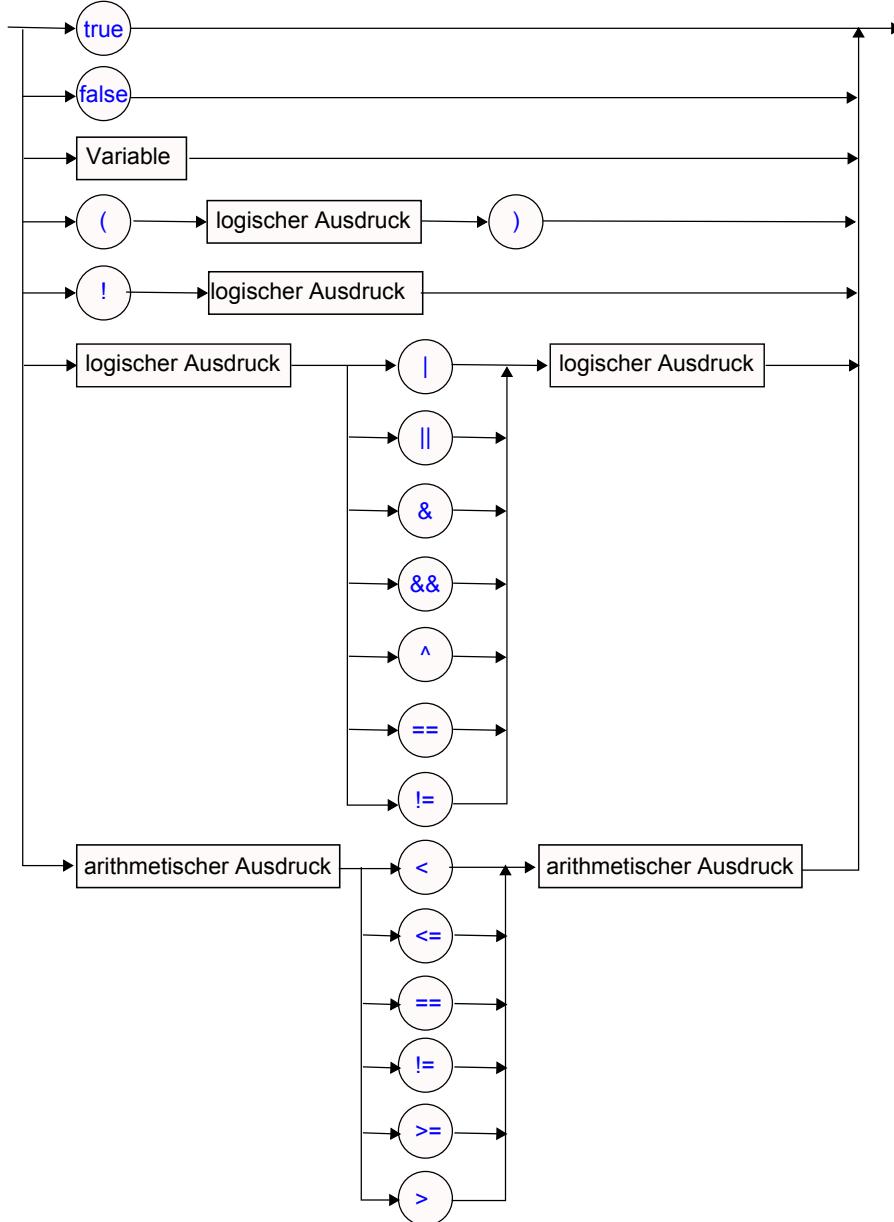
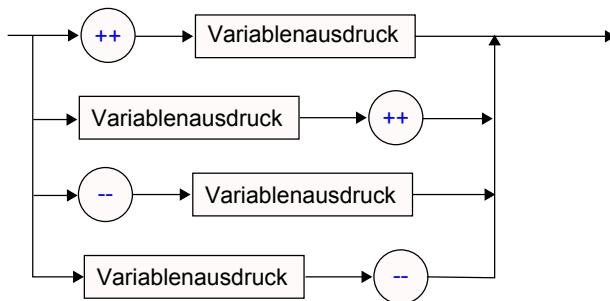


### Parameter



### Ausdruck

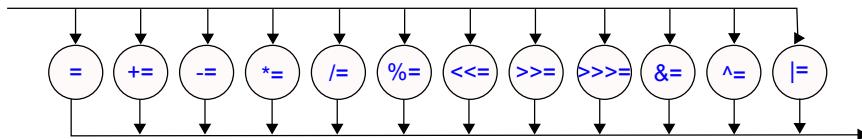


**logischer Ausdruck****Bedingungsausdruck****Inkrement-/Dekrementausdruck**

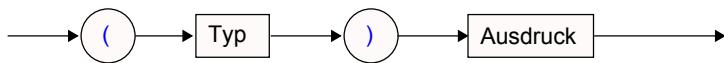
### Zuweisungsausdruck



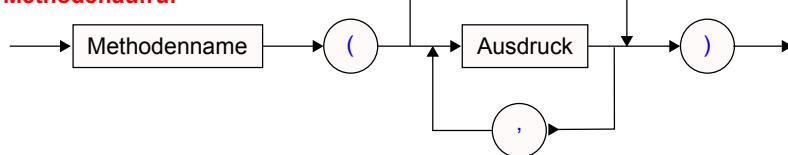
### Zuweisungsoperator



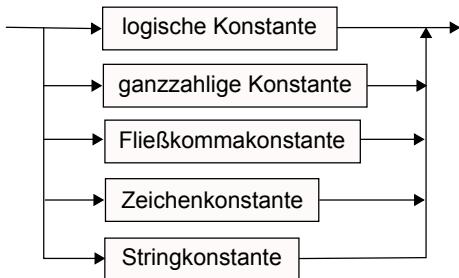
### cast-Ausdruck

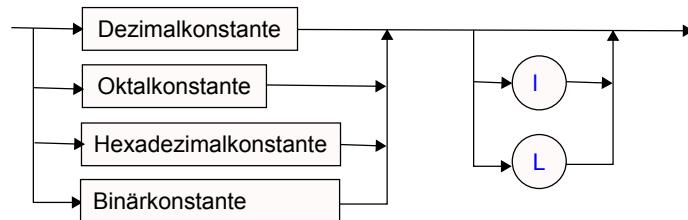
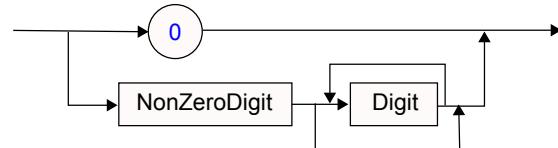
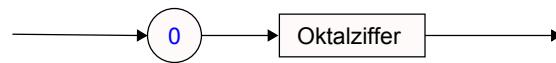
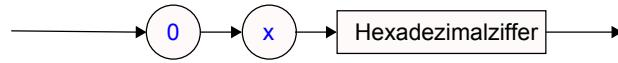
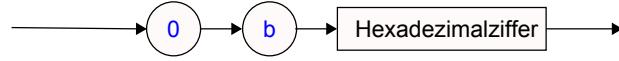
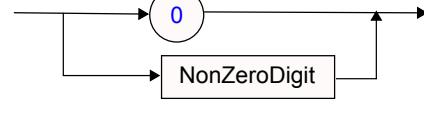
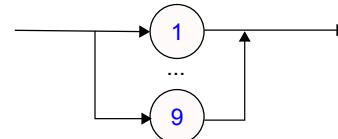
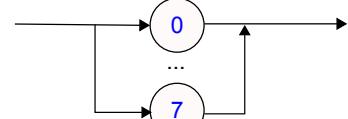
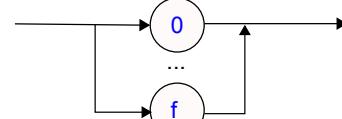
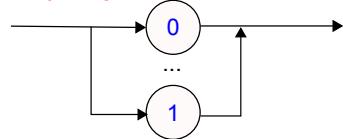


### Methodenaufruf

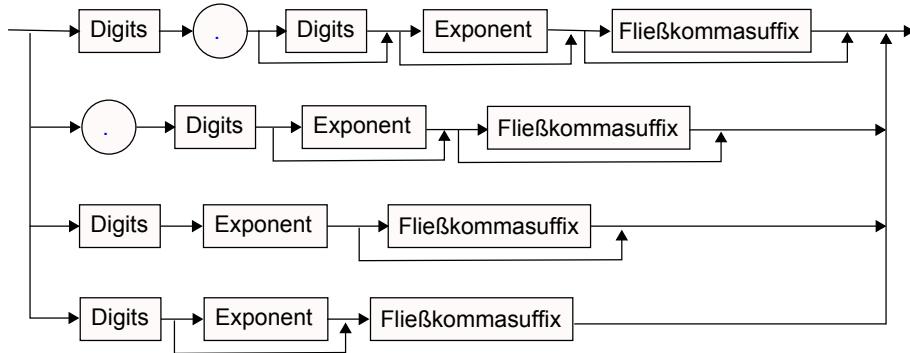


### Konstante

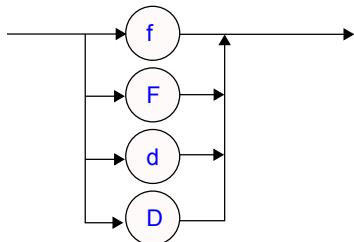


**Ganzzahlkonstante****Dezimalkonstante****Oktalkonstante****Hexadezimalkonstante****Binärkonstante****Digit****NonZeroDigit****Oktalziffer****Hexadezimalziffer****Binärziffer**

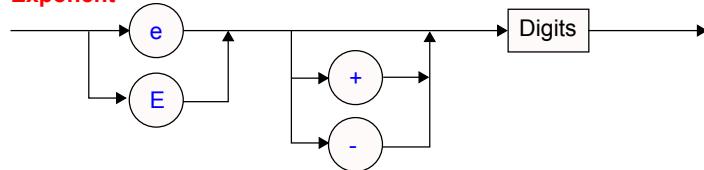
### Fließkommakonstante



### Fließkommasuffix



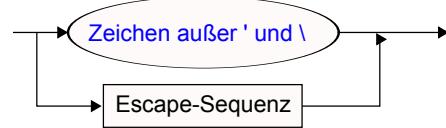
### Exponent



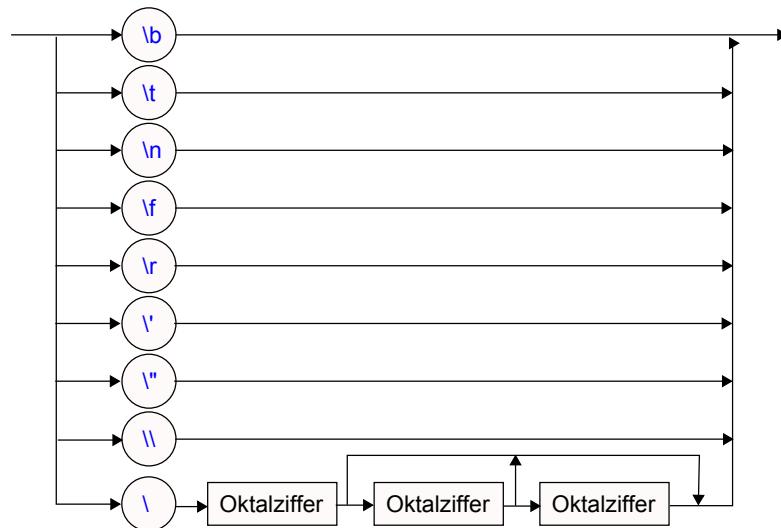
### Zeichenkonstante

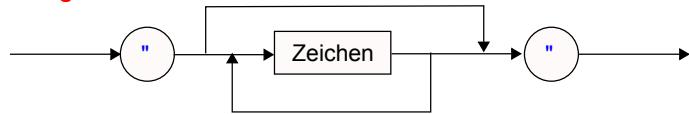


### Zeichen



### Escape-Sequenz



**String-Konstante**

## **Appendix C**

# **Priorität und Assoziativität von Operatoren**

Operator	Beschreibung	Assoziativität
()	Klammerung	links
[]	Feldzugriff	links
()	Methodenaufruf	links
.	Zugriff auf Komponente	links
()	Klammerung	links
++	Inkrement	rechts
--	Dekrement	rechts
+	Vorzeichenplus	rechts
-	Vorzeichenminus	rechts
~	Bitkomplement	rechts
!	logische Negation	rechts
(Typ)	Cast-Operation	rechts
new	Objekterzeugung	rechts
*	Multiplikation	links
/	Division	links
%	ganzzahlige Modulobildung	links
+	Addition	links
-	Subtraktion	links
+	Stringverkettung	links
<<	Shift-Operator	links
>>	Shift-Operator	links
>>>	Shift-Operator	links
<	kleiner als	links
<=	kleiner gleich	links
>	größer als	links
>=	größer gleich	links
instanceof	Instanztest	links
==	gleich	links
!=	ungleich	links
&	bitweises Und	links
&	bitweises Und	links
^	logisches Exklusives Oder	links
^	bitweises Exklusives Oder	links
	logisches Oder	links
	bitweises Oder	links
&&	logisches Und	links
	logisches Oder	links
?:	Bedingung	rechts
=	Zuweisung	rechts
*=	kombinierte Zuweisung	rechts
/=	kombinierte Zuweisung	rechts
+=	kombinierte Zuweisung	rechts
--=	kombinierte Zuweisung	rechts
%=	kombinierte Zuweisung	rechts
<<=	kombinierte Zuweisung	rechts
>>=	kombinierte Zuweisung	rechts
>>>=	kombinierte Zuweisung	rechts
&=	kombinierte Zuweisung	rechts
^=	kombinierte Zuweisung	rechts
=	kombinierte Zuweisung	rechts

Table C.1: Priorität und Assoziativität der Java-Operatoren

## Appendix D

# Regeln zum Gültigkeitsbereich und der Lebensdauer von Java-Variablen

Variablenart	Gültigkeitkeit	Lebensdauer
blocklokal	Deklarationspunkt bis Blockende	Ausführungszeit des Blocks
schleifenlokal	Schleife	Ausführungszeit der Schleife
Methodenparameter	Methode	Ausführungszeit der Methode
Instanzvariable	Klasse	Lebensdauer der Objektinstanz
Klassenvariable	Klasse	Laufzeit des Programms



# Bibliography

- [AB02] ASTEROTH, Alexander ; BAIER, Christel: *Theoretische Informatik*. München : Pearson Studium, 2002
- [AK02] ALLEN, Randy ; KENNEDY, Ken: *Optimizing Compilers for Modern Architectures*. San Francisco : Morgan Kaufmann, 2002
- [ALSU08] AHO, Alfred V. ; LAM, Monica S. ; SETHI, Ravi ; ULLMAN, Jeffrey D.: *Compiler - Prinzipien, Techniken und Werkzeuge*. 2. aktualisierte Auflage. München : Pearson Studium, 2008
- [Bal05] BALZERT, Helmut: *Lehrbuch Grundlagen der Informatik*. 2. Auflage. Elsevier Verlag, 2005
- [Bal08] BALZERT, Helmut: *Lehrbuch der Softwaretechnik. Software-Management*. 2. Auflage. Heidelberg : Spektrum Akademischer Verlag, 2008
- [Cha99] CHABERT, Jean-Luc (Hrsg.): *A History of Algorithms*. Berlin : Springer, 1999
- [Cho55] CHOMSKY, Noam: *Transformational Analysis*, University of Pennsylvania, Diss., 1955
- [Dij72] DIJKSTRA, Edsger W.: The Humble Programmer. In: *Comm. ACM* 15 (1972), Oktober, Nr. 10, S. 859–866
- [ecl] *eclipse.* : *eclipse.* <http://www.eclipse.org/>
- [Gel01] GELLNER, Michael ; UNIVERSIT"AT ROSTOCK (Hrsg.): *Der Umgang mit dem Hoare-Kalkül zur Programmverifikation*. <http://www.informatik.uni-rostock.de/mmis/courses/ss07/23002/>; Universit"at Rostock, 2001
- [GL96] GOLUB, Gene H. ; LOAN, Charles F.: *Matrix Computations*. third edition. Baltimore : The John Hopkins University Press, 1996
- [Gol91] GOLDBERG, David: What Every Computer Scientist Should Know About Floating-Point Arithmetic. In: *ACM Computing Surveys* 23 (1991), März, Nr. 1
- [Gro] GROUP, Object M.: *UML*. <http://www.omg.org/spec/UML/>
- [HMHG11] HEINISCH, Cornelia ; MÜLLER-HOFMANN, Frank ; GOLL, Joachim: *Java als erste Programmiersprache*. 6. überarbeitet Auflage. Wiesbaden : Vieweg und Teubner, 2011
- [HMU02] HOPCROFT, John E. ; MOTWANI, Rajeev ; ULLMAN, Jeffrey D.: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. 2. überarbeitete Auflage. München : Pearson Studium, 2002
- [Hoa69] HOARE, C. A. R.: An Axiomatic Basis for Computer Programming. In: *Comm. ACM* 12 (1969), Oktober, Nr. 10, S. 567–783

- [HP05] HENNESSY, John L. ; PATTERSON, David A.: *Rechnerorganisation und -entwurf*. 3. Auflage. Spektrum Akademischer Verlag, 2005
- [ISO96] ISO/IEC 14977 : 1996(E) (Hrsg.): *EBNF*. Genf, Schweiz: ISO/IEC 14977 : 1996(E), 1996
- [Java] *Java API Specification*. : *Java API Specification*. <http://docs.oracle.com/javase/7/docs/api/>
- [Javb] *Java Code Conventions*. : *Java Code Conventions*. <http://java.sun.com/docs/codeconv/>
- [Javc] *Java Language Specification*. : *Java Language Specification*. <https://docs.oracle.com/javase/specs/>
- [Javd] *Java Tutorial*. : *Java Tutorial*. [java.sun.com/docs/books/tutorial/](http://java.sun.com/docs/books/tutorial/)
- [Jave] *JavaDoc*. : *JavaDoc*. <http://java.sun.com/j2se/javadoc/writingdoccomments/>
- [Jr.79] JR., Edmung Melson C.: Programming Language Constructs for Which Its Is Impossible To Obtain Good Hoare Axiom Systems. In: *Journal of the ACM* 26 (1979), Januar, Nr. 1, S. 129–147
- [jun] *Junit*. : *Junit*. <http://www.junit.org/>
- [Knu98] KNUTH, Donald E.: *The Art of Computer Programming*. Bd. 2 : Seminumerical Algorithms. 3. Reading, MA : Addison Wesley, 1998
- [Mey00] MEYER, Bertrand: *Object-Oriented Software Construction*. second edition. Prentice Hall, 2000
- [MF04] MICHALEWICZ, Zbigniew ; FOGEL, David B.: *How to Solve It: Modern Heuristics*. Berlin : Springer, 2004
- [Mod] *Model Checking*. : *Model Checking*. [http://de.wikipedia.org/wiki/Model\\_Checking](http://de.wikipedia.org/wiki/Model_Checking)
- [Muc97] MUCHNICK, Steven S.: *Advanced Compiler Design and Implementation*. San Francisco : Morgan Kaufmann, 1997
- [NBB<sup>+63</sup>] NEUER, Peter ; BACKUS, J.W. ; BAUER, F.L. ; GREEN, J. ; KATZ, C. ; McCARTHY, J. ; PERLIS, A.J. ; H.RUTISHAUSER ; SAMELSON, K. ; VAUQUOIS, B. ; WEGSTEIN, J.H. ; WIHNGAARDEN, A. van ; WOODGER, W.: Revised Report on the Algorithmic Language ALGOL 60. In: *Comm. ACM* 6 (1963), Januar, Nr. 1, S. 1–17
- [PH09] PATTERSON, David A. ; HENNESSY, John L.: *Computer Organization and Design – The Hardware Software Interface*. 4. Auflage. Morgan Kaufmann, 2009
- [RQZ07] RUPP, Chris ; QUEINS, Stefan ; ZENGLER, Barbara: *UML 2 glasklar*. 3. Auflage. München : Hanser Verlag, 2007
- [Rum88] RUMP, Siegfried M.: Algorithm for Verified Inclusions - Theory and Practice. In: MOORE, R.E. (Hrsg.): *Reliability in Computing* Bd. 19, Academic Press, 1988 (Perspectives in Computing), S. 109–126
- [Sch00] SCHÖNING, Uwe: *Logik für Informatiker*. 5. Auflage. Heidelberg : Spektrum Akademischer Verlag, 2000
- [Sch10] SCHIEDERMEIER, Reinhard: *Programmieren mit Java*. 2. aktualisierte Auflage. München : Pearson Studium, 2010
- [SLO] [http://en.wikipedia.org/wiki/Source\\_lines\\_of\\_code](http://en.wikipedia.org/wiki/Source_lines_of_code). : [http://en.wikipedia.org/wiki/Source\\_lines\\_of\\_code](http://en.wikipedia.org/wiki/Source_lines_of_code)

- [SW11] SEDGEWICK, Robert ; WAYNE, Kevin: *Einführung in die Programmierung mit Java*. München : Pearson Studium, 2011
- [Ull12] ULLENBOOM, Christian: *Java ist auch eine Insel*. 10., aktualisierte und überarbeitete Auflage. Bonn : Galileo Press, 2012
- [VW06] VOSSEN, Gottfried ; WITT, Kurt-Ulrich: *Grundkurs Theoretische Informatik*. 4. Auflage. Wiesbaden : Vieweg Verlag, 2006
- [Wir73] WIRTH, Niklaus: The Programming Language Pascal (Revised Report) / ETH Zürich. Zürich, Juli 1973 (5). – Forschungsbericht. – Berichte der Fachgruppe Computer-Wissenschaften
- [Zie96] ZIEGENBALG, Jochen: *Algorithmen*. Heidelberg : Spektrum Akademischer Verlag, 1996



# Listings

1.1	Hello World in Java. . . . .	7
2.1	Verfahren zur Berechnung des größten gemeinsamen Teilers. . . . .	19
2.2	ggT-Programm in Java. . . . .	26
3.1	Umrechnung von Dollar nach Euro. . . . .	42
3.2	Umrechnung von Dollar nach Euro. . . . .	45
4.1	Beispiel zum Datentyp boolean. . . . .	89
4.2	Beispiel zum Datentyp int. . . . .	106
4.3	Berechnung der Quersumme einer dreistelligen Zahl. . . . .	107
4.4	Beispiel für Bit-Operationen. . . . .	108
4.5	Bestimmung des Maschinenepsilon. . . . .	115
4.6	Beispiel zur Fließkommaproblematik. . . . .	116
4.7	Beispiel zur Fließkommazahlen. . . . .	118
4.8	Berechnung zum Schiefen Wurf. . . . .	120
4.9	Beispiel zur Fließkommaproblematik (nach [Rum88]). . . . .	121
4.10	Beispiel zur Nutzung von Zeichen. . . . .	127
4.11	Weiteres Beispiel zur Nutzung von Zeichen. . . . .	128
4.12	Beispiel zur Manipulation von Zeichencodes. . . . .	129
4.13	Beispiel zur String-Verarbeitung. . . . .	130
4.14	Beispiel zu String-Operationen. . . . .	131
4.15	Einlesen von Werten aus der Kommandozeile. . . . .	133
4.16	Einlesen von Werten über die Tastatur. . . . .	134
4.17	Einlesen von Werten aus einer Datei. . . . .	135
4.18	Beispiele zur formatierten Ausgabe. . . . .	137
5.1	Umwandlung von Fahrenheit nach Celsius. . . . .	144
5.2	Berechnung des Maximums zweier Werte. . . . .	145
5.3	Berechnung des Maximums und Minimums zweier Werte. . . . .	146
5.4	Berechnung des Maximums und Minimums dreier Werte. . . . .	148
5.5	Berechnung zu Schaltjahren über geschachtelte Selektionen. . . . .	150
5.6	Berechnung zu Schaltjahren über einen logischen Ausdruck. . . . .	150
5.7	Umwandlung von Ziffern in Zahlen. . . . .	152
5.8	Berechnung der Anzahl an Tagen in einem Monat. . . . .	154
5.9	Potenzberechnung mit einer Zählschleife. . . . .	158
5.10	Summenberechnung. . . . .	159
5.11	Fakultätsberechnung mit einer Zählschleife. . . . .	159
5.12	Berechnung einer Doppelsumme. . . . .	160
5.13	Erzeugung aller dreistelligen Binärzahlen. . . . .	161
5.14	Erzeugung von Quadrat- und Kubikzahlen. . . . .	161

5.15 Ausgabe zur Programmausführung des Programms in Listing 5.14. . . . .	162
5.16 Erzeugung von Quadrat- und Kubikzahlen. . . . .	162
5.17 Berechnung von Annuitätendarlehen. . . . .	164
5.18 Ausgabe zur Annuitätenberechnung. . . . .	165
5.19 Fehlerhaftes Beispiel. . . . .	165
5.20 Fehlerhaftes Beispiel, richtig formatiert. . . . .	165
5.21 ggT-Berechnung. . . . .	167
5.22 Iterative Berechnung einer Quadratwurzel. . . . .	169
5.23 Ausgabe zur Berechnung der Quadratwurzel. . . . .	169
5.24 Iterative Berechnung einer Quadratwurzel (Version 2). . . . .	171
5.25 Zentrale Schleife eines Herzschrittmachers. . . . .	171
5.26 Iterative Berechnung einer Quadratwurzel (do-while). . . . .	173
5.27 Iterative Berechnung einer Quadratwurzel (while). . . . .	174
5.28 Berechnung von Primzahlen. . . . .	176
5.29 Berechnung von Primzahlen (mit break). . . . .	177
5.30 Einlesen und aufsummieren von Zahlen. . . . .	178
5.31 Einlesen und verarbeiten von 10 Zahlen. . . . .	179
5.32 Beispiel zur leeren Anweisung. . . . .	181
6.1 Umrechnung von Fahrenheit nach Celsius. . . . .	184
6.2 Umrechnung von Fahrenheit nach Celsius. . . . .	184
6.3 Blocklokale Variablen. . . . .	187
6.4 Schleifenlokale Variable. . . . .	187
6.5 Variablendeclaration mir var. . . . .	189
6.6 Beispiel zum Gültigkeitsbereich blocklokaler Variablen. . . . .	189
6.7 Beispiel 2 zum Gültigkeitsbereich blocklokaler Variablen. . . . .	190
6.8 Beispiel 3 zum Gültigkeitsbereich blocklokaler Variablen. . . . .	191
6.9 Beispiel zum Gültigkeitsbereich schleifenlokaler Variablen. . . . .	191
6.10 Beispiel zur Sichtbarkeit von Variablen. . . . .	193
6.11 Beispiel zur Lebensdauer von Variablen. . . . .	194
7.1 Umwandlung von Fahrenheit nach Celsius. . . . .	199
7.2 Beispiel zur Nutzung des Bedingungsoperators. . . . .	201
7.3 Beispiele zur Nutzung von Inkrement-/Dekrementoperatoren. . . . .	202
7.4 Beispiele zur Nutzung von Zuweisungsoperatoren. . . . .	203
7.5 Reihenfolge von Operationen. . . . .	205
7.6 Reihenfolge von Operationen. . . . .	205
7.7 Beispiel zu einer final-Deklaration einer Variablen/Konstanten. . . . .	207
7.8 Einengende Typumwandlung. . . . .	210
7.9 Einige prinzipielle Möglichkeiten für Typumwandlungen. . . . .	210
8.1 Programm zur Flächenberechnung. . . . .	216
8.2 Beispiel zur korrekten Definition von Methoden in einer Klasse. . . . .	220
8.3 Beispiel zu möglichen Fehlern bei Methodendefinitionen. . . . .	221
8.4 Beispiel zu Auswertung eines Methodenaufrufs. . . . .	223
8.5 Quadratische Gleichung lösen. . . . .	224
8.6 Abstandsberechnung. . . . .	226
8.7 Beliebige Namen von formalen Parametern. . . . .	227
8.8 Überladen von Methodennamen. . . . .	229
8.9 Berechnung der Quadratwurzel. . . . .	231

8.10 Berechnung der Quadratwurzel überladen. . . . .	232
8.11 Direkte Umsetzung rekursiv definierter Funktionen in Java. . . . .	235
8.12 Ackermann-Funktion. . . . .	236
8.13 Abstandsberechnung. . . . .	237
8.14 Beispiel zu Übergabestrategien. . . . .	241
8.15 Beispiel 2 zu Übergabestrategien. . . . .	241
8.16 Fakultätsberechnung. . . . .	244
8.17 Was passiert? . . . . .	246
8.18 Annäherungsweise Berechnung von e (rekursiv). . . . .	249
8.19 Annäherungsweise Berechnung von e (iterativ). . . . .	251
8.20 Beispiel zur Umwandlung von Iteration nach Rekursion. . . . .	252
9.1 Skalarprodukt. . . . .	259
9.2 Vektordurchschnitt. . . . .	259
9.3 Einfaches Beispiel zu Felddeklarationen. . . . .	261
9.4 Einfaches Beispiel zu Felddeklarationen über Initialisierung. . . . .	262
9.5 Sequentielles Suchen in einem Feld. . . . .	262
9.6 Sequentielles Suchen in einem Feld als Methode. . . . .	263
9.7 Zugriff auf Feldindizes außerhalb des erlaubten Bereichs. . . . .	264
9.8 Beispiel für die erweiterte Schleifensyntax. . . . .	265
9.9 Bildschirmausgabe einer Matrix. . . . .	267
9.10 Addition zweier Matrizen. . . . .	268
9.11 Anlegen eines sehr großen Feldes. . . . .	270
9.12 Zuweisung von Referenzen. . . . .	272
9.13 Kopieren / Clonen eines Feldes. . . . .	273
9.14 Vergleich von Referenzen. . . . .	274
9.15 Zuweisung von Referenzen (Beispiel 2). . . . .	275
9.16 Zuweisung von Referenzen (Beispiel 3). . . . .	275
9.17 Wiederbetrachtung von Übergabestrategien. . . . .	277
11.1 Klasse Tier (Version 1, unvollständig). . . . .	313
11.2 Klasse Gehege (Version 1, unvollständig). . . . .	313
11.3 Klasse Tier mit Instanzvariablen und Instanzmethoden (Version 2, unvollständig). . . . .	315
11.4 Klasse Gehege mit Instanzvariablen und Instanzmethoden (Version 2, unvollständig). . . . .	316
11.5 Klasse Tier (Version 3 mit Konstruktoren, unvollständig). . . . .	319
11.6 Klasse Gehege (Version 3 mit Konstruktoren, unvollständig). . . . .	320
11.7 Klasse Zoo (Version 1, unvollständig). . . . .	321
11.8 Beispiel zur Nutzung von Instanzmethoden. . . . .	322
11.9 Beispiel zur Lebensdauer von Objekten. . . . .	324
11.10 Beispiel zu Klassenvariablen. . . . .	325
11.11 Beispiel zur Nutzung von Klassenvariablen und -methoden. . . . .	326
11.12 Beispiel zu Zugriffsmöglichkeiten von Instanz-/Klassenvariablen/-methoden. . . . .	328
11.13 Klasse Zoo. . . . .	330
11.14 Klasse Gehege. . . . .	330
11.15 Klasse Tier. . . . .	331
11.16 Klasse Punkt. . . . .	332
11.17 Klasse Gerade. . . . .	333
11.18 Klasse GeometrieTest (Version 1). . . . .	334
12.1 Erste Version des Patentiers basierend auf der Tierklasse. . . . .	340

12.2 Klasse Punkt3D. . . . .	341
12.3 Beispiel zum Gültigkeitsbereich bei Vererbung. . . . .	342
12.4 Klasse Patentier. . . . .	344
12.5 Beispiele zu Möglichkeiten der Typanpassung. . . . .	349
12.6 Kovarianz bei Feldern. . . . .	350
12.7 Deklaration von Paketen. . . . .	351
12.8 Deklaration von Unterpaketen. . . . .	352
12.9 Nutzung von Paketen. . . . .	352
12.10 Zugriff auf Klassen anderer Pakete. . . . .	354
12.11 Beispiel zu Zugriffsrechten bei Klassen. . . . .	354
12.12 Auswirkung von Zugriffsrechten (Teil 1). . . . .	356
12.13 Auswirkung von Zugriffsrechten (Teil 2). . . . .	357
12.14 Getter-/Setter Methoden unter Nutzung von Modifikatoren. . . . .	357
12.15 Beispiel zum Überladen einer Methode über Klassengrenzen hinweg. . . . .	359
12.16 Beispiel zum Überschreiben einer Methode. . . . .	359
12.17 Nutzung des Polymorphismus. . . . .	360
12.18 Beispiel zur Kovarianz im Ergebnistyp. . . . .	361
12.19 Beispiel zur Nutzung von <code>object</code> als Basisklasse. . . . .	362
12.20 Kovarianz bei Feldern. . . . .	362
12.21 Methodenaufrufe mit Objekten. . . . .	364
12.22 Beispiel zur Polymorphie. . . . .	365
12.23 Punkt3D unter Verwendung von <code>instanceof</code> . . . . .	367
12.24 Nutzung von 2D- und 3D-Geraden. . . . .	368
13.1 Beispiel zu finalen Variablen. . . . .	376
13.2 Beispiel zu finalen Methoden. . . . .	376
13.3 Beispiel zu finalen Klassen. . . . .	377
13.4 Abstrakte Tierklasse. . . . .	378
13.5 Konkrete Ameisenklasse. . . . .	378
13.6 Abstrakte Tierklasse mit abstrakter Methode <code>bewegen</code> . . . . .	380
13.7 Konkrete Ameisenklasse mit realisierter Methode <code>bewegen</code> (Version 2). . . . .	380
13.8 Konkrete Eulenklasse mit realisierter Methode <code>bewegen</code> . . . . .	380
13.9 Zooklasse, die die Methode <code>bewegen</code> für alle Tiere nutzt. . . . .	381
13.10 Schnittstelle <code>Impfeable</code> . . . . .	383
13.11 Schnittstelle <code>PreisErmitteable</code> . . . . .	383
13.12 Abstrakte Tierklasse, die zwei Schnittstellen realisiert. . . . .	383
13.13 Konkrete Ameisenklasse mit realisierten Schnittstellenmethoden (Version 3). . . . .	384
13.14 Schnittstelle <code>Piekseable</code> , von der Schnittstelle <code>Impfeable</code> abgeleitet. . . . .	385
13.15 Konkrete Eulenklasse mit drei realisierten Schnittstellenmethoden. . . . .	385
13.16 Cast auf einen Schnittstellentyp. . . . .	386
13.17 Eigene Klasse zur Kapselung von primitiven Werten. . . . .	387
13.18 Beispiel zum Auto-Boxing. . . . .	388
13.19 Beispiel für eine innere Klasse. . . . .	389
13.20 Beispiel für eine lokale Klasse. . . . .	389
13.21 Beispiel für eine anonyme Klasse. . . . .	390
13.22 Beispiel für eine anonyme Klasse mit der Nutzung von Schnittstellen. . . . .	390
14.1 Beispielprogramm mit möglichen (Fehler-)Ausnahmen. . . . .	398
14.2 Beispielprogramm, wenn man selbst alle Fehler abfangen müsste. . . . .	399

14.3 Beispiel für eine Fehlerbehandlung als Teil der Programmlogik. . . . .	400
14.4 Definition einer eigenen Fehlerklasse. . . . .	402
14.5 Werfen und Fangen einer eigenen Exception. . . . .	403
14.6 Prinzipieller Aufbau einer <code>try</code> -Anweisung. . . . .	404
14.7 Beispiel für den Einsatz von <code>finally</code> . . . . .	405
14.8 Beispiel zu einer Multi-catch-Anweisung. . . . .	406
14.9 Beispiel für eine generische Ausnahmebehandlung. . . . .	407
14.10 Beispiel für eine individuelle Ausnahmebehandlung. . . . .	407
14.11 Beispiel zur Propagation von Exceptions. . . . .	408
14.12 Beispiel für den Einsatz von Assertions. . . . .	410
15.1 Aufsummieren von Feldinhalten. . . . .	415
15.2 Suchen eines Wertes in einem Feld. . . . .	416
15.3 Fakultätsberechnung. . . . .	423
15.4 Fibonacci-Funktion. . . . .	424
15.5 Sequentielles Suchen in einem Feld. . . . .	425
15.6 Binäres Suchen in einem Feld. . . . .	428
15.7 Sortieren durch Auswählen. . . . .	431
15.8 Sortieren durch Einfügen. . . . .	434
15.9 Indirektes Sortieren. . . . .	437
16.1 Sieb des Eratosthenes als Java-Programm. . . . .	444
16.2 Berechnung der Quersumme einer zweistelligen Zahl. . . . .	446
16.3 Teilüberprüfung der Fermatschen Vermutung. . . . .	448
16.4 Türme von Hanoi als Java-Programm. . . . .	453
16.5 Durchlaufen eines Labyrinths (Teil 1). . . . .	458
16.6 Durchlaufen eines Labyrinths (Teil 2). . . . .	459
17.1 Beispielprogramm 1. . . . .	463
17.2 Beispielprogramm 2. . . . .	463

# Index

*G<sub>bin</sub>*, 54, 56, 60, 61                          <=, 202  
*G<sub>dez</sub>*, 55, 57, 59–62                          <<, 105, 504  
*G<sub>expr2</sub>*, 63    <=, 87, 105, 119, 498, 504  
*G<sub>expr</sub>*, 59, 60, 63                                  <, 87, 105, 119, 498, 504  
*L(G)*, 56    ==, 87, 105, 119, 498, 504  
 $\Sigma^*$ , 52    =, 48, 142, 202, 504  
 $\Sigma^+$ , 53    >=, 87, 105, 119, 498, 504  
 $\vdash$ , 55    >>=, 202  
 $\vdash^*$ , 55    >>>=, 202  
 $\vdash^n$ , 55    >>, 105, 504  
 $\varepsilon$ , 52, 286    >, 105, 504  
 $|w|$ , 52    ?:, 200  
|, 90    D, 117, 501  
||, 90    E, 117, 501  
(), 210    F, 117, 501  
+, 278    L, 103, 500  
.NET, 77    ", 129, 502  
.a, 77    %=, 202  
.dll, 77    %, 105, 504  
.o, 76    &=, 202  
.obj, 76    &&, 86, 87, 498, 504  
.so, 77    &, 86, 87, 105, 498, 504  
=, 48    \', 126, 501  
==, 273, 278    \\, 126, 501  
&, 90    \b, 126, 501  
&&, 90    \f, 126, 501  
( ), 87, 498, 504    \n, 126, 501  
\*=, 202    \r, 126, 501  
\*, 105, 119, 504    \t, 126, 501  
++, 200, 201    ^=, 202  
+=, 202    ^, 105, 504  
+, 90, 105, 119, 131, 504                                  {}, 143  
--, 200, 201    ~, 504  
-=, 202    ^, 87, 498  
-, 105, 119, 504    **abstract**, 378  
/=, 202    **assert**, 409  
/, 105, 119, 504    **break**, 151, 153  
:, 153    **byte**, 151  
;, 180

**case**, 151, 153  
**catch**, 141  
**char**, 151  
**default**, 153  
**d**, 117, 501  
**else**, 145  
**e**, 117, 501  
**false**, 87, 498  
**f**, 117, 501  
**if**, 145  
**int**, 151  
**log**, 151  
1, 103, 500  
**public**, 218  
**sealed**, 392  
**short**, 151  
**static**, 218  
**switch**, 151, 153  
**true**, 87, 498  
**try**, 141  
**var**, 188  
**void**, 218, 219  
, 131, 215  
  
Abbruchkriterium, 156  
Ableitung, 55, 65, 66, 69, 75  
abs(), 119  
Abstraktion, 312  
Ackermann-Funktion, 234  
activation record, 239  
Adresse, 185, 243, 269  
    Hauptspeicher, 8  
Aktivitätsdiagramm, 22, 391  
al-Khowarizmi, 11  
Algorithmus, 11  
    Komplexität, 413  
Alphabet, 51  
Analyse, 69  
    lexikale, 72  
    syntaktische, 74  
anonyme Klasse, 388  
Anweisung, 15, 38, 142, 143, 145, 147, 153, 471, 495  
    Ausdrucks-, 141  
    leere, 180, 495  
Anweisungsüberdeckung, 464  
Argument, 222  
ASCII, 122  
Assembler, 42  
Assemblersprache, 39, 42  
Assertion, 409  
Assoziativität, 204  
Attribut, 311  
Aufrufbereich, 239  
    dynamischer, 244  
    statischer, 244  
aufrufen, 221  
Auftraggeber, 1  
Auftragnehmer, 1  
Ausdruck, 15, 38, 141, 142, 197, 200, 202, 210, 269, 471, 497  
    arithmetischer, 48, 66, 197, 296, 304  
Priorität von Operatoren, 203  
Typ, 198  
Wert, 198  
Ausdrucksanweisung, 141, 142, 495  
ausführbares Programm, 72  
Ausgabe, 20  
Ausnahme, 397  
Auswahl, 145  
auswerten, 221  
Auto-Boxing, 387  
average case, 417  
Axiom, 471  
Backtracking, 455  
Backus-Naur-Form, 60  
Basic, 487  
Basisfunktion, 31  
Baum, 75, 344, 345  
    Binärbaum, 292  
    Blatt, 292  
    Knoten, 292  
    Teilbaum, 292  
    Wurzel, 292  
Bean, 329  
Bedingungsausdruck, 200, 498  
Bedingungsüberdeckung, 464  
Befehl, 8  
Befehlszähler, 27  
Berechnungsmodell  
    funktional, 31  
Bereichsüberschreitung, 109  
best case, 417  
Besuchsstrategie, 424

- Betriebssystem, 39
- Bezeichner, 45, 72, 185, 186, 218
- Bezugsobjekt, 321
- Bibliothek, 77
- Binden, 77
- Binder, 72
- Bindung
  - dynamische, 365
  - statische, 365
- Binomialkoeffizient, 234
- binär, 200
- Binärbaum, 292, 344, 429
  - Durchlaufstrategie, 299
- Binäres Suchen, 426
- Bit, 9, 81
- Bitmap Font, 126
- Black Box Test, 462
- Blatt, 292
- Block, 142, 143, 146, 147, 189, 193, 350, 495
  - schachteln, 190
- blocklokal, 186
- BNF, 60
- Boolean, 387
- boolean, 86
- break-Anweisung, 175
- break-Sequenz, 153
- Brute Force, 446
- Bubble Sort, 435
- Byte, 83, 387
- byte, 101
- Byte Order Mark, 125
- C/C++, 40, 216, 271, 276, 487
- call-by-reference, 243
- call-by-value, 240, 276, 387
- Cast, 348
  - Down-, 348
  - Up-, 348
- cast
  - Ausdruck, 499
  - Operator, 211
- CFG, 60
- char, 127
- Character, 387
- charakterisierender Parameter, 413
- Chomsky, 57
- Chomsky-Grammatik, 54
- Churchsche These, 27
- CIL, 77
- CLR, 77
- Cobol, 40
- Codeerzeugung, 76
- Codierung, 82
- Compiler, 71, 146, 185, 238, 397
  - Phasen, 72
- Container, 429
- continue, 175
- continue-Anweisung, 175
- cos(), 119
- CPU, 8
- Datenkapselung, 329
- Datentyp, 81, 84, 260
  - abstrakter, 283, 284
  - konkreter, 284
  - primitiv, 38
  - primitiver, 85, 218
- Default-Konstruktor, 318
- Definition
  - rekursive, 287
- Deklaration, 143, 144, 186, 193
  - Variable, 47
- Dekrementoperator, 200
- Design by Contract, 464
- Dezimalkonstante, 103, 500
- Digit, 103, 500
- Digitalrechner, 9, 81
- Digits, 117, 501
- Dijkstra
  - E., 462
- DIN 66001 (Flussdiagramm), 21
- DIN 66261 (Struktogramm), 23
- Diskriminator, 340, 344
- do-while, 175
- Dokumentationskommentar, 220
- Double, 387
- Down-Cast, 348
- Durchlaufstrategie, 264, 299
- dynamische Bindung, 365
- eclipse, 5
  - Klasse anlegen, 5
  - Programm starten, 8
  - Projekt anlegen, 5
- Eiffel, 464

- Eingabe, 20, 28  
Einsetzungsschema, 32, 488  
Einskomplement, 99  
Einzelanweisung, 15  
Einzelschrittfunktion, 29  
else-Fall, 145  
Embedded System, 39  
Endlicher Automat, 58, 70  
Endlosschleife, 17, 170, 206  
endrekursiv, 252  
Entwurfsphase, 2  
Eratosthenes, 442  
Ergebnis, 239  
Error, 401  
Escape-Sequenz, 126, 127, 501  
Exbi, 83  
Exception, 105, 397, 401  
    checked, 401  
    fangen, 397  
    unchecked, 401  
    werfen, 397, 402  
Excess-Code, 112  
 $\exp()$ , 119  
Exponent, 111, 117, 501  
extends, 340
- Fakultätsfunktion, 27, 158, 244, 246, 250  
Fallunterscheidung, 142, 151, 496  
false, 86  
Fehler, 146, 461  
    logischer, 461  
    semantisch, 461  
    Syntax, 461  
Fehlerklassen, 461  
Feld, 258  
Festkommazahl, 110  
Fibonacci-Funktion, 234, 253, 488  
final, 206  
Fließkommakonstante, 117, 501  
Fließkommasuffix, 117, 501  
Fließkommazahl, 91, 111  
Float, 387  
floating point numbers, 91  
Flussdiagramm, 21, 149  
Folge, 286, 429, 451  
    leere, 286  
Font, 126
- Bitmap, 126  
True Type, 126  
for, 175  
Fortran, 40, 216, 242, 487  
Framework, 1  
Funktion, 216  
    partielle, 29  
funktionales Programmierparadigma, 488  
funktionales Berechnungsmodell, 31  
Funktionseinheit, 8  
fussgesteuerte Schleife, 496  
fußgesteuerte Schleife, 170  
fußgesteuerte Schleife, 142
- Ganzzahl, 91  
Ganzzahlkonstante, 103, 500  
Garbage Collector, 323  
garbage collector, 276  
Generalisierung, 340  
ggT, 22, 166  
Gibi, 83  
Gleitkommazahl, 111  
Grammatik, 51, 54, 70  
    Chomsky, 54  
    kontextfrei, 58, 60, 74  
    kontextsensitiv, 58  
    regulär, 58, 72  
    Typ, 57  
Graph, 292  
Graphentheorie, 292  
Grundsymbol, 70, 72  
größter gemeinsamer Teiler, 166  
Gödel  
    Kurt, 24  
Gültigkeitsbereich, 189, 194, 218, 220, 221, 314, 326, 342
- halblogarithmisches Format, 110  
Halteproblem, 24  
Hanoi  
    Türme von, 449  
Haskell, 489  
Hauptspeicher, 8, 185  
Hauptspeicheradresse, 8, 269  
Heap, 247, 269, 278, 318  
Heron, 230  
Hexadezimalkonstante, 103, 500  
Hexadezimalzahl, 96

- Hexadezimalziffer, 103, 500
- Hoare
  - C.A.R., 464
  - Kalkül, 471
- Horn-Klausel, 490
- Hornerschema, 94
- HTML, 69
- höchstwertig, 83
- IEC, 83
- IEEE 754-2008, 112
- imperatives Programmierparadigma, 185, 487
- Indexfeld, 435
- Induktion
  - strukturelle, 464
- induktiv, 471
- Infix, 53
- Infixoperator, 200
- Informatik
  - Theoretische, 24
- Information Hiding, 329
- Inkonsistenz, 199
- Inkrement-/Dekrementausdruck, 201, 498
- Inkrementoperator, 200
- innere Klasse, 388
- Inorder, 299
- Instanz, 312, 317
- Instanziierung, 317
- Instanzmethode, 314
- Instanzvariable, 221, 314, 324
- int, 101
- Integer, 387
- Integer numbers, 91
- Interpreter, 69, 238
- Interpretierer, 70
- Invariante, 465
- Invarianz, 363
- Is-A-Beziehung, 345
- ISO, 62, 123
- ISO 8859-15, 123
- ISO 5807 (Flussdiagramm), 21
- ISO 8859-1, 123
- Iteration, 17, 155, 472
- Iterationsregel, 473, 477
- Java, 40, 487
- Java Code Conventions, 329
- Java Beans, 329
- Java Code Conventions, 312
- Java Code Conventions, 43, 144, 146, 151, 157, 185, 206, 218
- Java Virtual Machine, 77
- java.Math, 119
- javadoc, 220
- Junit, 464
- JVM, 76, 77
- Kalender
  - gregorianischer, 149
- Kante, 21, 292
- Kellerautomat, 59, 74
- Kellerspeicher, 303
- Kibi, 83
- Klasse, 5, 38, 45, 312, 350
  - abgeleitete, 340
  - anonym, 388
  - Basis, 340
  - innere, 388
  - lokal, 388
  - Ober-, 340
  - Unter-, 340
  - versiegelt, 392
  - Wrapper-, 387
- Klassendiagramm, 391
- Klassenvariable, 221, 324
- Klassifikationsprinzip, 345
- Klausel, 489, 490
  - Horn-, 490
- Knoten, 21, 292
- Kodierung, 2, 92
- Kommentar, 44
  - Dokumentations-, 220
- Komplement
  - Einskomplement, 99
  - Zweierkomplemen, 99
- Komplexität, 413
  - average case, 417
  - average-case, 426, 429
  - best case, 417
  - best-case, 426, 429
  - worst case, 417
  - worst-case, 426, 429
- Komponente, 285, 329
- Konkatenation, 52
- Konklusion, 471

Konsequenzregel, 473, 478  
Konstante, 72, 198, 499  
Konstruktor, 278, 317  
    Default-, 318  
kontextfrei, 58, 60  
kontextfreie Grammatik, 74  
kontextsensitiv, 58  
Kontravarianz, 363  
Kontrolleinheit, 8  
Kontrollfluss, 21, 219  
Kontrollstruktur, 141  
kopfgesteuerte Schleife, 142, 166, 496  
korrekt  
    partiell, 465  
    total, 465  
Korrektheit, 461  
Korrektheitsformel, 465  
Kovarianz, 361, 363  
  
L-Wert, 143  
LaTeX, 40  
Latin-1, 123  
Laufzeitstack, 244, 246  
Lebendauer, 194  
Lebensdauer, 194, 218, 314, 323, 326  
leere Anweisung, 142, 180, 471, 495  
    Regel, 472, 473  
leeres Wort, 52  
Level, 295  
lex, 65  
lexikale Analyse, 72  
Linker, 72, 77  
Lisp, 489  
Literal, 72, 198  
log(), 119  
log10(), 119  
logischer Fehler, 461  
logischer Ausdruck, 87, 145, 200, 498  
lokale Klasse, 388  
Long, 387  
long, 101  
Länge  
    Wort, 52  
Lösungsraum, 441  
  
main, 215, 220  
Mainboard, 8  
Mantisse, 111  
    normalisiert, 111  
Maschinencode, 76, 238, 246  
Matlab, 40  
Maximum, 149  
Mebi, 83  
Mehrachelselektion, 16, 151  
Metasymbol, 50, 60  
Methode, 45, 200, 216, 311, 350, 497  
    aufrufen, 143, 221  
    auswerten, 221  
    definieren, 217, 326  
    Ergebnis, 239  
    Instanz-, 314, 326  
    Klassen-, 326  
    Kopf, 217  
    Parameterübergabe  
        call-by-reference, 243  
        call-by-value, 240  
    Rumpf, 217, 218  
Methodenaufruf, 499  
Minimum, 149  
Miranda, 489  
mischen, 454  
ML, 488, 489  
Model Checking, 464  
Modell, 455  
Modifikator, 353  
Modul, 18, 19  
Muster, 32, 141  
  
Nachbedingung, 465  
Nachricht, 309  
Name, 185  
    einfach, 351  
    qualifiziert, 351  
Namensraum, 350  
Nassi-Shneiderman-Diagramm, 23  
Nebeneffekt, 142  
nebenläufig, 20  
new, 318  
Nichtterminalsymbol, 54  
niederwertig, 83  
NonZeroDigit, 103, 500  
normalisiert, 111  
Notation  
    Infix, 300  
    Postfix, 301

- Präfix, 301
- null, 274
- numerisches Verfahren, 168
- Näherungsverfahren, 168
- O-Notation, 418
- Oberklasse, 340
- Objekt, 309
- Objektdatei, 76
- Objektorientierte Programmierung, 216
- objektorientierter Ansatz, 309
- Objektprogramm, 72
- Octave, 40
- Oktalkonstante, 103, 500
- Oktalzahl, 96
- Oktalziffer, 103, 500
- Omega-Notation, 418
- Operand, 48
- Operationssymbol, 283
- Operator, 48, 200
  - Assoziativität, 204
  - binär, 200
  - cast, 211
  - Dekrement, 200
  - Infix, 200
  - Inkrement, 200
  - Postfix, 200
  - Prefix, 200
  - Priorität, 203
  - ternär, 200
  - unär, 200
  - überladen, 203
- Optimierungsphase, 76
- Paket, 350
  - Default-, 351
  - Unter-, 351
- Palindrom, 289
- PAP, 21
- parallel, 20
- Parameter, 497
  - aktueller, 19, 222
  - charakterisierender, 413
  - formaler, 19, 217, 222
- partial korrekt, 465
- Pascal, 40, 216, 487
- PC, 8
- PDF, 40
- Pebi, 83
- Permutation, 82, 160, 435
  - Identitäts-, 436
- Petri Netz, 464
- Pfadüberdeckung, 464
- Phase
  - Entwurf, 2
  - Kodierung, 2
  - Spezifikation, 2
  - Test, 2
- Phasenmodell, 2
- Platine, 8
- Polymorphismus, 357
- portabel, 40
- Portabilität, 39
- postcondition, 465
- Postfix, 53
- Postfix-Notation, 304
- Postfixnotation, 301
- Postfixoperator, 200
- Postorder, 299
- pow(), 119
- precondition, 464
- Prefixoperator, 200
- Preorder, 299
- Primitive Rekursion, 292
- Primitive Rekursion, 32
- primitiver Datentyp, 85
- Primzahl, 175, 442, 487
- Priorität, 203
- private, 353, 379
- Problemdefinition, 2
- Problemlösungsprozess, 1
- Problemspezifikation, 2
- Produktion, 54
- Programmablaufplan, 21
- Programmiermuster, 141
- Programmierparadigma, 487
  - applikatives, 488
  - funktionales, 488
  - imperatives, 185, 487
  - logikorientiertes, 489
  - objektorientiertes, 185, 309
- Programmiersprache, 37
  - Historie, 38
  - maschinenorientiert, 39
  - problemorientiert, 39

- universell, 40
- Programmierung, 2
- Programmspeicher, 27
- Programmsprache
  - Ausbildung, 38
  - Projekt, 5
  - Prolog, 489
  - Property, 329
  - protected, 353
  - Prozessmodell, 2
  - Prozessor, 8
  - Prädikatenlogik, 471
  - Präfix, 53
  - Präfixnotation, 301
  - Prämisse, 471
  - public, 353
  - Python, 71
- Qualitätssicherung, 462
- Quellprogramm, 72
- Queue, 306
- R-Wert, 143
- Rechnerarchitektur, 81
- record, 334
- Referenz, 243, 244, 269
- Referenztyp, 218, 269
- Regel
  - Iteration, 473, 477
  - Konsequenz, 473, 478
  - leere Anweisung, 473
  - Selektion, 473, 475
  - Sequenz, 473, 474
  - Zuweisung, 472
- Regel zur leeren Anweisung, 472
- Register, 8, 27
- regulär, 58
- Reihenfolge, 20
- Rekursion, 234
  - direkte, 234
  - geschachtelte, 236
  - indirekte, 234
- rekursiv, 233, 287
- Relation, 489
- terminieren, 170
- return-Anweisung, 142, 496
- Rohe Gewalt, 446
- Rumpf, 19
- Rundungsfehler, 111
- RunTimeException, 401
- Rücksprungadresse, 239
- Schaltjahr, 149
- Schema, 32
  - Einsetzungsschema, 32
  - Primitive Rekursion, 32
- Schlange, 306, 429
- Schleife, 175
  - fußgesteuerte, 170
  - kopfgesteuerte, 166
  - Zähl-, 156
- Schleifeninvariante, 477
- Schleifenkontrollausdruck, 175
- Schleifenkopf, 156, 166
- schleifenlokale Variable, 186
- Schleifenrumpf, 19, 156, 166, 175
- Schlüsselwort, 46, 72, 185
- Schnittstelle, 217, 350, 382, 462
- Schnittstellenspezifikation, 18
- SDK, 347
- Seiteneffekt, 206, 472
- Selektion, 16, 142, 145, 471, 495
- Selektionsregel, 473, 475
- Semantik, 51, 67, 94
- Semantikfehler, 461
- semantische Analyse, 76
- sequentielle Ausführung, 15
- sequentielles Suchen, 261
- Sequenz, 15, 143, 471
- Sequenzregel, 473, 474
- Short, 387
- short, 101
- Sichtbarkeit, 193, 194, 342
- Sieb des Eratosthenes, 487
- Sieb des Eratosthenes, 442
- Signatur, 318, 359
- sin(), 119
- skalare Konstante, 153
- skalarer Ausdruck, 153
- Skalarprodukt, 40, 258
- skip, 471
- Software Engineering, 1, 4, 464
- Sonderzeichen, 72
- Sorte, 283
- Sortieren, 451

- Bubble Sort, 435
- durch Auswählen, 430
- durch Einfügen, 433
- durch Aufsteigen, 435
- stabil, 433
- Speicher
  - Adresse, 185
  - Spezialisierung, 340
  - Spezifikation, 461, 464
  - Sprache, 53
    - erzeugte, 56
  - `sqrt()`, 119
  - stabil, 433
  - Stack, 71, 244, 303, 429
  - Stapel, 303
  - Startsymbol, 54
  - static, 324, 326
  - statische Bindung, 365
  - Stellenwertsystem, 93
  - strikte Auswertung, 206
  - String, 129, 278
    - Konstante, 129
  - String Pool, 278
  - String-Konstante, 502
  - Struktogramm, 23, 149, 465
  - strukturelle Induktion, 464
  - Strukturierte Programmierung, 216
  - Substitutionsprinzip, 360
  - Substitutionsregel, 345
  - Suchbaum, 301
  - Suchen
    - binäres, 426
    - sequentielles, 261
  - super, 342
  - swap, 276
  - switch, 175
  - syntaktische Analyse, 74
  - Syntax, 51, 94, 146, 471
    - Fehler, 71
  - Syntaxbaum
    - abstrakter, 75
    - attributierter abstrakter, 76
  - Syntaxdiagramm, 65
  - Syntaxfehler, 147, 461
  - Tag, 221
  - `tan()`, 119
- Tebi, 83
- terminieren, 206
- ternär, 200
- Test
  - Black Box, 462
  - White Box, 462
- Test-Driven Development, 464
- Testdaten, 463
- Testen, 462
- Testphase, 2
- then-Fall, 145
- Theta-Notation, 418
- throw, 402
- Throwable, 401
- Tiefe, 295
- Token, 70, 72
- total korrekt, 465
- Traveling Saleman Problem, 24
- Treiber, 39
- Trennzeichen, 74
- true, 86
- True Type Font, 126
- Trägermenge, 283
- Tupel, 285
- Turing
  - Alan, 24
- Typ, 47, 185, 198, 260
  - einfacher, 269
  - primitiv, 269
  - Referenz-, 218, 269
- Typangabe, 210
- Typumwandlung
  - einengend, 347
  - erweiternd, 208, 347
  - verengend, 208
- Übersetzer, 71
- UCS, 124
  - Transformation Format, 124
- UML, 22, 391
- undefiniert, 186
- unentscheidbar, 24
- Unicode, 123
- Unified Modeling Language, 22, 391
- Unifikation, 488
- Unit Test, 464
- Universal Character Set, 124

- Universelle Registermaschine, 27
- Unix, 246
- Unterklasse, 340
- Unterprogramm, 216
- Unterprogrammtechnik, 216
- unär, 200
- Up-Cast, 348
- UPN, 301
- URM, 27, 239, 487
  - berechenbar, 30
  - Programm, 29
  - Zustand, 28
  - Zustandsraum, 29
  - Zustandstransformation, 29
- UTF-8, 124
- Variable, 14, 46, 54, 87, 142, 198, 200, 294, 471, 487, 498
  - Adresse, 185
  - blocklokal, 186, 189, 193, 314, 326
  - Deklaration, 186, 189
  - Gültigkeitsbereich, 189
  - Initialisierung, 326
  - Instanz-, 314, 324, 326
  - Klassen-, 324, 326
  - Lebensdauer, 194
  - Name, 185
  - schleifenlokal, 186, 189, 193
  - Sichtbarkeit, 193
  - Typ, 185
  - Wert, 185
- Variablen, 45
- Variablenausdruck, 201, 202
- Vererbung, 339
- Vergleich, 273, 278
- Verifikation, 462, 464
- Vollständige Induktion, 32
- vollständige Suche, 446
- von Neumann
  - Architektur, 185, 238, 487
- Vorbedingung, 464
- Vorgehensmodell, 2
- Vorzeichen, 111
- Vorzeichenbit, 99
- Vorzeichendarstellung, 98
- Wert, 185, 198
- Wertübergabe, 240
- while, 175
- White Box Test, 462
- Wiederholung, 155
- worst case, 417
- Wort, 52
- Worterkennungsproblem, 57, 58
- Wrapper-Klasse, 387
- Wurzel, 292
- yacc, 65
- Yobi, 83
- Zebi, 83
- Zeichen, 126, 129, 501, 502
- Zeichenkette, 129
- Zeichenkonstante, 126, 501
- Zeichensatz, 121
  - ASCII, 122
  - ISO 8859-1, 123
  - ISO 8859-15, 123
  - Latin-1, 123
  - Unicode, 123
  - UTF-8, 124
- Zeiger, 243, 244, 247, 269
- Zentraleinheit, 8
- Zusicherung, 465
- Zustand, 185, 487
- Zustandstransformation, 185, 487
- Zuweisung, 142, 471
  - Regel, 473
- Zuweisungsausdruck, 202, 499
- Zuweisungsoperator, 202
- Zuweisungsregel, 472, 473
- Zweierkomplement, 99, 160
- Zwischencode, 76
- Zählschleife, 142, 156, 264, 496
- Zählschleifenaktualisierung, 496
- Zählschleifenkopf, 496
- Überdeckung
  - Anweisungs-, 464
  - Bedingungs-, 464
  - Pfad-, 464
- Übersetzer, 69
- überladen, 203, 318
- überschreiben, 359