



---

# Java – exemplarisch

---

Learning by doing

---

von  
Dr. Ägidius Plüss

---

Oldenbourg Verlag München Wien

---

---

Dr. Aegidius Plüss besitzt eine langjährige Erfahrung als Professor für Informatik und deren Didaktik an der Universität Bern (Schweiz) und Lehrer für Physik und Informatik am Mathematisch-naturwissenschaftlichen Gymnasium Bern-Neufeld. Während seiner Lehrtätigkeit hat er Assembler, Basic, Pascal, Logo, C/C++ und Java unterrichtet. Seine Forschungs- und Entwicklungsschwerpunkte liegen im Bereich der Simulationen und Echtzeitverarbeitung.

#### Bibliografische Information Der Deutschen Bibliothek

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

© 2004 Oldenbourg Wissenschaftsverlag GmbH  
Rosenheimer Straße 145, D-81671 München  
Telefon: (089) 45051-0  
[www.oldenbourg-verlag.de](http://www.oldenbourg-verlag.de)

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

Lektorat: Margit Roth  
Herstellung: Rainer Hartl  
Umschlag: Bernard Schlup, Bern  
Umschlagkonzeption: Kraxenberger Kommunikationshaus, München  
Gedruckt auf säure- und chlorfreiem Papier  
Druck: Grafik + Druck, München  
Bindung: R. Oldenbourg Graphische Betriebe Binderei GmbH

ISBN 3-486-20040-2

*Für Jaye*

# Inhalt

<b>Inhalt</b>	<b>VII</b>
<b>Vorwort</b>	<b>XVII</b>
<b>1 Einleitung</b>	<b>1</b>
<b>2 Daten und Programme</b>	<b>7</b>
2.1 Formulierung von Algorithmen .....	8
2.2 Die algorithmischen Grundstrukturen.....	9
2.3 Klassen und Objekte.....	9
2.4 Der Entwicklungsprozess eines Programms .....	11
<b>3 Das erste Java Programm</b>	<b>15</b>
3.1 Konventionen und Schlüsselwörter.....	15
3.2 Grundaufbau eines Java-Programms.....	17
3.3 Verwendung von Klassenbibliotheken .....	20
3.4 Turtlegrafik, das Schlüsselwort new .....	23
<b>4 Die grundlegenden Programmstrukturen</b>	<b>27</b>
4.1 Die Sequenz.....	27
4.2 Die Iteration (Wiederholung).....	28
4.2.1 Die while-Struktur.....	29
4.2.2 Die do-while-Struktur .....	33
4.2.3 Die for-Struktur .....	34
4.3 Die Selektion (Auswahl) .....	36
4.3.1 Die if-Struktur (einseitige Auswahl) .....	36
4.3.2 Die if-else-Struktur (zweiseitige Auswahl).....	38
4.3.3 Die switch-Struktur (Mehrfachauswahl).....	40

4.4	Varianten der for-Struktur.....	42
4.4.1	Mehrere Schleifenvariable .....	44
4.4.2	Weglassen einzelner Teile .....	44
4.4.3	Endlosschleife .....	45
4.4.4	for-Schleife ohne Körper .....	46
<b>5</b>	<b>Elementare Klassenkonstruktionen</b>	<b>47</b>
5.1	Objektorientiertes Konstruieren.....	47
5.2	Methoden in der Applikationsklasse.....	52
5.3	Instanzvariablen und lokale Variablen, Sichtbarkeit, Geltungsbereich.....	55
5.4	Überladen .....	61
<b>6</b>	<b>Datentypen, Operatoren</b>	<b>65</b>
6.1	Basistypen .....	65
6.2	Standard-Initialisierung.....	70
6.3	Zahlendarstellung und Rundungsfehler .....	70
6.4	Operatoren und Ausdrücke .....	72
6.4.1	Verknüpfungsoperatoren.....	75
6.4.2	Bedingungen und logische (boolesche) Operatoren .....	75
6.4.3	Binäre arithmetische Operatoren .....	80
6.4.4	Autoinkrement, Autodekrement, Verbundoperatoren .....	86
6.4.5	Der Bedingungsoperator .....	87
6.5	Unsigned Datentypen, big-endian, little-endian .....	88
6.6	Spezielle Werte für doubles .....	90
<b>7</b>	<b>Wichtige Standardklassen</b>	<b>93</b>
7.1	Zeichenketten (Strings).....	93
7.1.1	Erzeugung, Vergleich .....	93
7.1.2	Stringkonkatenation .....	95
7.1.3	Null-String und leerer String .....	97
7.1.4	Begleitende Klasse StringBuffer.....	97
7.2	Arrays (Felder).....	99
7.2.1	Eindimensionale Arrays.....	99
7.2.2	Kommandozeilen-Parameter.....	105
7.2.3	Mehrdimensionale Arrays.....	106
7.2.4	Character-Arrays.....	108
<b>8</b>	<b>Das Variablenkonzept</b>	<b>111</b>
8.1	Zuweisung bei Basis- und Referenztypen .....	111
8.2	Sprechweise bei Referenztypen .....	116

8.3	Vergleich von Referenztypen.....	117
8.4	Parameterübergabe bei Basis- und Referenztypen.....	118
8.4.1	Rückgabe über Instanzvariablen .....	119
8.4.2	Rückgabe über Referenzen.....	120
8.4.3	Rückgabe über den Returnwert.....	125
8.5	Bezeichner für Instanzvariablen und Parameter .....	127
8.6	Namenskonflikte zwischen Variablen und Parametern .....	128
<b>9</b>	<b>Hüllklassen (Wrapper-Klassen)</b>	<b>131</b>
<b>10</b>	<b>Rekursionen</b>	<b>135</b>
10.1	Direkte Rekursion, Verankerung .....	135
10.2	Prinzip von Teile-und-Herrsche .....	141
<b>11</b>	<b>Formatierung der Ausgabe</b>	<b>145</b>
<b>12</b>	<b>Klassendesign</b>	<b>153</b>
12.1	Vererbung.....	154
12.2	Delegation, Aggregation, Komposition .....	155
12.3	Klassendiagramme .....	155
12.4	Komposition versus Vererbung.....	156
12.5	Vererbung des Console-Fensters.....	158
12.6	Mustervorlage für den Klassenentwurf.....	160
12.7	Finale Klassen .....	168
12.8	Datenkapselung, Abschottung.....	169
<b>13</b>	<b>Zugriffsbezeichner</b>	<b>171</b>
13.1	Package-Konzept.....	171
13.2	Garbage Collection, finalize().....	177
<b>14</b>	<b>Polymorphie, dynamische Bindung</b>	<b>181</b>
<b>15</b>	<b>Statische Variablen und Methoden</b>	<b>185</b>
<b>16</b>	<b>Dynamische Datentypen, abstrakte Klassen</b>	<b>191</b>
16.1	Beispiel eines typischen Klassendesigns .....	191
16.2	Collections: Hashtable und Enumeration.....	196
16.3	Verwendung von Entwurfsmustern: Abstract-Factory .....	198

<b>17</b>	<b>Interfaces</b>	<b>203</b>
17.1	Interfaces zur losen Kopplung .....	204
17.2	Das Interface Comparable.....	207
17.3	Interfaces zur Definition von Konstanten .....	210
17.4	Interfaces & Komposition statt Vererbung .....	211
<b>18</b>	<b>Ereignissteuerung (Eventhandling)</b>	<b>217</b>
18.1	Ereignismodelle, innere und anonyme Klassen .....	217
18.2	Delegations-Eventmodell.....	219
18.3	Erstellen eigener Event-Klassen .....	224
18.4	Beispiele von GUI-Events .....	236
18.4.1	Maus-Events.....	236
18.4.2	Tastatur-Events .....	242
18.4.3	Tastatur- und Maus-Events mit Turtles .....	244
18.4.4	Button-Events.....	248
18.4.5	Menü-Events .....	255
18.5	Reentrance bei Callbackmethoden.....	262
<b>19</b>	<b>Grafik mit Java2D und GWindow</b>	<b>265</b>
19.1	Das Rendern von Grafiken unter Graphics2D .....	265
19.2	Grafik mit GWindow .....	267
19.3	Animierte Grafik bei Simulationen.....	278
<b>20</b>	<b>Exceptions</b>	<b>285</b>
20.1	Robuste Programme.....	285
20.2	Try-catch-Blöcke .....	286
20.3	Eigene Exceptions, Parsen von Eingaben.....	292
20.4	Assertions (Zusicherungen) .....	298
20.5	Exceptions nach oben werfen .....	299
<b>21</b>	<b>Klassenbibliotheken von Drittherstellern</b>	<b>301</b>
21.1	Plotlibraries .....	301
21.2	Komplexe Zahlen .....	306
21.3	Matrizen .....	310
<b>22</b>	<b>Drucken</b>	<b>315</b>
22.1	Drucken mit Bildschirmauflösung.....	315



22.2	Drucken mit Druckerauflösung.....	315
22.3	Drucken in Druckerauflösung mit den Klassen Turtle und GPanel .....	318
<b>23</b>	<b>Streams</b>	<b>323</b>
23.1	Das Java I/O-System .....	323
23.2	Dateiverwaltung .....	324
23.3	Textdatei lesen.....	327
23.4	Textdatei schreiben .....	329
23.5	Textdatei mit Zahlen schreiben .....	330
23.6	Zahlen aus einer Textdatei lesen .....	331
23.7	Binärdatei schreiben und lesen.....	333
23.8	Dateien mit wahlfreiem Zugriff (Random Access).....	335
23.9	Serialisierung.....	337
23.10	Programmressourcen .....	340
23.11	Verpacken in jar-Archiven .....	343
<b>24</b>	<b>Properties</b>	<b>347</b>
<b>25</b>	<b>Software engineering</b>	<b>353</b>
25.1	Die Phasen der Software-Entwicklung (design cycle).....	353
25.2	Die Qualität von Software.....	355
25.3	Software-Entwicklung im Kleinen.....	356
25.3.1	Formulieren .....	356
25.3.2	Entwerfen .....	356
25.3.3	Implementieren.....	357
<b>26</b>	<b>Konstruktoren</b>	<b>365</b>
26.1	Initialisierung mit Konstruktoren.....	365
26.2	Überladene Konstruktoren .....	367
26.3	Ablaufreihenfolge bei Konstruktoren .....	374
26.4	Statischer Initialisierungsblock .....	377
26.5	Verwendung von this in Konstruktoren .....	378
26.6	Fehlerabfang in Konstruktoren .....	379
<b>27</b>	<b>Überschreiben von Methoden und Variablen</b>	<b>383</b>
27.1	Überschreiben von Methoden .....	383

27.2	Überschreiben und Verdecken von Variablen .....	386
27.3	Zugriffsrechte bei überschriebenen Methoden .....	388
<b>28</b>	<b>Unstrukturierte Sprachelemente</b> .....	<b>391</b>
28.1	Vorzeitiges return.....	391
28.2	break und continue .....	393
28.3	Flags.....	394
28.4	Labeled break und labeled continue .....	396
<b>29</b>	<b>Grafische Benutzeroberflächen (GUI)</b> .....	<b>399</b>
29.1	Grundsätze .....	399
29.2	Grafik mit Swing.....	403
29.3	Wichtige Ein- und Ausgabekomponenten, Verwendung des Layout-Managers ....	404
29.3.1	Funktionsklassen.....	415
29.3.2	Fourier-Reihen .....	417
29.3.3	Modale Dialoge.....	420
29.3.4	Nicht-modale Dialoge .....	428
29.3.5	Look-and-Feel, Closing-Option, Fokus, Inputvalidierung .....	434
29.4	Einbetten eines GPanel.....	438
29.5	GUI-Mustervorlage.....	441
<b>30</b>	<b>Klonen, flache und tiefe Kopie</b> .....	<b>457</b>
<b>31</b>	<b>Nebenläufigkeit (Multithreading)</b> .....	<b>467</b>
31.1	Parallele Datenverarbeitung .....	467
31.2	Threads.....	468
31.3	Sichere Threads, Synchronisation kritischer Bereiche .....	473
31.4	Laufzeitverhalten, Terminieren von Programmen .....	486
31.5	Das Schlüsselwort volatile .....	488
<b>32</b>	<b>Debugging-Strategien</b> .....	<b>491</b>
32.1	Reproduzierbarkeit, Trial and Error.....	491
32.2	Versionsmanagement.....	492
32.3	Tracen, Debuggen .....	494
32.4	Debuggen mit print-Anweisungen.....	496
<b>33</b>	<b>Netzwerk-Programmierung (Sockets)</b> .....	<b>503</b>
33.1	Computernetzwerke, Schichtenmodell .....	503

33.2	Client-Server-Modell .....	505
33.3	Client- und Server-Sockets .....	507
33.4	Multithreaded Server.....	511
33.5	Telnet, Grundlagen von HTTP.....	514
33.6	Echo-Server .....	520
33.7	Chat-Client und -Server .....	524
<b>34</b>	<b>Serielle und parallele Schnittstellen</b>	<b>535</b>
34.1	Installation des Communication-API.....	535
34.2	Verwendung der seriellen Schnittstelle.....	536
34.3	Programmierung der seriellen Schnittstelle .....	538
34.4	Robotics.....	543
34.5	Datenakquisition mit Vielfachmessgerät .....	551
34.6	Auflisten der vorhandenen Ports .....	557
34.7	Verwendung der parallelen Schnittstelle .....	560
<b>35</b>	<b>Sound</b>	<b>565</b>
35.1	Digitalisierung von Sound.....	565
35.2	Das Java Sound-API.....	566
35.2.1	Wiedergabe eines Audioclips.....	568
35.2.2	Soundgenerator.....	571
35.2.3	Virtuelles Musikinstrument.....	576
35.2.4	Wiedergabe von Streaming Sound.....	579
35.2.5	Aufnahme von Sound.....	583
<b>36</b>	<b>AWT/Swing, Animation und Bildtransformation</b>	<b>591</b>
36.1	Schwergewichtige und leichtgewichtige Komponenten .....	591
36.2	Das Rendern von Grafikkomponenten.....	592
36.2.1	Callbackmechanismus für systemgetriggertes Rendern .....	592
36.2.2	Anwendungsgetriggertes Rendern, animierte Grafik.....	598
36.3	Optimierung bei inkrementellen Grafiken .....	607
36.3.1	Optimierung bei AWT .....	608
36.3.2	Optimierung bei Swing .....	612
36.4	Affine Bildtransformationen .....	617
<b>37</b>	<b>Applets</b>	<b>621</b>
37.1	Java's Sandkasten-Prinzip .....	621

37.2	Callbackmechanismus für Applets .....	624
37.3	Konzept zur Erstellung von klassischen Applets.....	625
37.4	Verwendung von Swing.....	632
37.5	Applets mit eigenständigem Fenster.....	637
37.5.1	Mustervorlage für Applets .....	637
37.5.2	Überführung einer GUI-Applikation in ein Applet .....	640
37.5.3	Verwendung von modalen Dialogen .....	644
37.5.4	Verwendung von GPanel .....	646
37.6	Parsen von XML-Dateien .....	648
<b>38</b>	<b>JDBC</b> .....	<b>655</b>
38.1	Relationale Datenbanken .....	655
38.2	Verwendung von MySQL.....	657
38.3	MS-Access als Datenbank-Client, ODBC .....	661
38.4	Client-Applikationen mit JDBC .....	662
<b>39</b>	<b>Reflection</b> .....	<b>679</b>
39.1	Laufzeit-Typeninformation.....	679
39.2	Dynamische Klassen- und Methodennamen.....	684
39.3	Reflection für Klassenbibliotheken.....	687
<b>40</b>	<b>Remote Method Invocation (RMI)</b> .....	<b>693</b>
<b>41</b>	<b>Dynamische Web-Seiten</b> .....	<b>699</b>
41.1	HyperText Transfer Protocol (HTTP) .....	699
41.2	Servlets.....	702
41.3	Java Server Pages (JSP).....	708
41.3.1	Scripting-Elemente .....	709
41.3.2	Nebenläufigkeit (Multithreading) bei JSP .....	716
41.3.3	Standard-Variablen (implizite Variablen) .....	718
41.3.4	Verwendung von Formular-Elementen .....	718
41.3.5	Sichtbarkeit und Lebensdauer, Sitzungsverfolgung .....	721
41.3.6	JSP-Direktiven, Datenbank-Zugriff.....	728
41.3.7	Debugging von JSP-Seiten .....	732
41.3.8	Sitzung mit offener Datenbankverbindung.....	733
41.3.9	Dynamische Links .....	738
41.3.10	Strukturierte JSP-Programmierung.....	740
41.3.11	Verpacken und Deployment von JSP-Applikationen .....	751
41.3.12	Java Beans, JSP-Aktionen .....	752
41.3.13	Web Content Management .....	759

---

<b>42</b>	<b>Anhang 1: Installation von externen Klassenbibliotheken</b>	<b>763</b>
42.1	Installation der Standard Extension .....	763
42.2	Klassen aus dem Package ch.aplu .....	763
<b>43</b>	<b>Anhang 2: Java Web Start</b>	<b>765</b>
43.1	Web-Start als Benutzer .....	765
43.2	Web-Start als Entwickler .....	766
<b>44</b>	<b>Anhang 3: Hinweise zur Installation von Tomcat (Version 5)</b>	<b>771</b>
	<b>Stichwortverzeichnis</b>	<b>775</b>

# Vorwort

Dieses Lehrbuch entstand aus dem Bedürfnis, ein Konzept auszuarbeiten, wie Java als Erstsprache unterrichtet und gelernt werden kann. Eine Analyse der großen Vielfalt vorliegender Publikationen und Kursangebote zur Programmierung in Java zeigt das ernüchternde Bild, dass oft dem behutsamen Vorgehen und sorgfältigen Aneinanderreihen von Lernschritten auf einer angepasst ansteigenden Lernkurve wenig Aufmerksamkeit gewidmet wird und mehr die Programmiersprache selbst als der Lernende im Zentrum der Bemühungen steht. Die Folge ist, dass sich beim Lernenden trotz gehöriger Anfangsbegeisterung nach kurzer Zeit ein Frust bemerkbar macht, weil er sich im Java-Klassen-Dschungel verirrt und wegen fehlender Grundlagen keine Erfolgserlebnisse mehr aufweisen kann. Dabei verliert er die Motivation und den Spaß am Lernen.

In diesem Buch werden die Konzepte der Programmiersprache Java und des objektorientierten Programmierens (OOP) ausschließlich an Hand von lauffähigen Programmen eingeführt. Diese können unter einer beliebigen Programmierumgebung (IDE) editiert, nach Belieben verändert, kompiliert und ausgeführt werden. In den Kapiteln 1–32 werden die Grundlagen von Java systematisch dargelegt, die übrigen Kapitel können als Ergänzungen für bestimmte Interessensschwerpunkte betrachtet werden. Die Installation und der Umgang mit der IDE sind nicht Teil dieses Buches. Es wird ausdrücklich empfohlen, eines der bekannten professionellen Produkte zu verwenden, die zu Lernzwecken meist kostenlos über das Internet zu beziehen sind. In Kurs- und Schulumgebungen bewährt haben sich JBuilder von Borland und Eclipse. Die Programmbeispiele setzen die Version 1.4 (oder höher) des Java Development Kits (JDK) voraus. Bei Problemen mit der Installation der IDE im Zusammenhang mit den Programmbeispielen wende man sich über die Website [www.aplu.ch](http://www.aplu.ch) an den Autor.

Da keine systematische Behandlung der umfangreichen Java Klassenbibliothek angestrebt wird, muss die Originaldokumentation der Java Foundation Class (JFC) herangezogen werden. Diese wird üblicherweise mit der IDE installiert, ist aber auch über Internet-Suchmaschinen zugänglich (für die Klasse xxx übliche Stichworte sind *java class xxx*). Dabei werden allerdings Grundkenntnisse der englischen Sprache vorausgesetzt.

Der Quellcode der Programme hält sich weitgehend an die üblichen Format-Konventionen (Java Coding Style Guide), aus methodischen Gründen wird aber in einigen Punkten davon abgewichen. Insbesondere werden die öffnenden und schließenden Klammern von Programmblöcken untereinander geschrieben, was die Lesbarkeit besonders für Anfänger wesentlich erhöht. Da viele Entwicklungsumgebungen eine automatische Codeformatierung enthalten, ist es mit wenig Aufwand möglich, die Formatierung den eigenen Gewohnheiten anzupassen. Der Leser wird aber dazu aufgefordert, in jedem Fall ästhetisch ansprechenden Quellcode zu schreiben, der sich an eine konsequente Formatierung hält. Der Einbau von Kommentaren in den Quellcode wird in diesem Buch auf ein Minimum beschränkt, vor al-

lem um den Umfang des publizierten Codes in Grenzen zu halten, aber auch deswegen, weil der Autor der Auffassung ist, dass gut geschriebener Code weitgehend selbsterklärend sein muss. Darum wurde in den Beispielen auch auf die Verwendung von JavaDoc verzichtet. Im Zweifelsfall gilt aber die Regel, eher zu viel als zu wenig zu dokumentieren. Eine Dokumentation im Quellcode ist immer dann zwingend, wenn Programmtechnik oder Ablauflogik unüblich sind.

Erklärtes Prinzip des Buches ist es, weitgehend ohne Vorwärts- oder Querreferenzen auszukommen. Es ist daher auch zum Einsteig in das Programmieren geeignet. Dabei wird Wert auf eine saubere, konsequente und logische Begriffsbildung gelegt. Wenn es sinnvoll erscheint, gewisse weiterführende Hinweise zu geben, sind diese in *Kleinschrift und kursiv* gesetzt. Sie sind aber nicht Teil des begrifflichen Gerüsts und können beim ersten Durchlesen übersprungen werden. Da der Autor der Meinung ist, dass die meisten Grundprinzipien der Programmierung einfach zu verstehen sind und sich die Grafik hervorragend eignet, diese anschaulich darzustellen, wird im Buch durchwegs eine eigene Grafikbibliothek eingesetzt, welche die Komplexität mit der grafischen Benutzeroberfläche (GUI) verdeckt. Da der Quellcode dieser Bibliothek aber als OpenSource frei verfügbar ist, kann ihn der fortgeschrittene Leser jederzeit heranziehen und zu seinen Zwecken verändern und weiter verwenden.

Beim genaueren Studium des Buches werden gewisse Vorlieben des Autors offensichtlich, die sich auf seine berufliche Herkunft und Tätigkeit zurückführen lassen. Es handelt sich um

- OOP-Design
- Grafik, insbesondere Richtungsgrafik und Swing
- Callbacks (Eventhandling)
- Rechnerkommunikation (Client-Server-Architektur)
- Datenbanken.

Obschon für ein Lehrbuch unüblich, wird auf ein Literaturverzeichnis verzichtet, und zwar aus zwei Gründen: Einerseits enthält das Buch praktisch keine Zitate oder Textteile aus anderen Werken, andererseits ist die Literatur auf dem Gebiet der Programmiersprache Java dermaßen umfangreich, dass es schwierig ist, gewissen Werken in einer Literaturzusammenstellung den Vorzug zu geben und sie dem Leser gegenüber anderen besonders zu empfehlen. Es ist aber selbstverständlich, dass viele der dargelegten Ideen und Programmbeispiele durch Zeitschriftenartikel, Webinformationen und Lehrbücher aus dem deutschen und englischen Sprachbereich inspiriert sind. Die wenigen Literaturreferenzen sind als Fußnoten angegeben.

Es wird nicht versucht, die deutsche Sprache frei von fachspezifischen Anglizismen zu halten, obschon der Computerslang zu sprachlich unerfreulichen Auswüchsen führen kann. Dies entspricht aber dem heutigen Trend der globalen Vereinheitlichung von Fachausdrücken und führt zu einem leichteren Verständnis innerhalb der Fachwelt, die vollständig auf die englische Sprache ausgerichtet ist. Die in Programmen frei wählbaren Bezeichner (Datei-, Variablen-, Methodennamen usw.) werden konsequent aus dem Englischen bezogen, was zur besseren Lesbarkeit in der professionellen Welt und zur leichteren Portierbarkeit (insbesondere im Internet) führt. Einzig Code-Kommentare und Dokumentationen sind ausnahmsweise auch in Deutsch verfasst, um den direkten Bezug zum Buchtext zu erleichtern. Gibt es für einen Begriff mehrere verbreitete Ausdrucksformen (beispielsweise Attribut, Instanzvariable, Feld), so musste etwas willkürlich für die eine Variante entschieden werden (Instanzvariable).

Links zu Websites sind an den wenigsten Stellen explizit aufgeführt, da diese sich erfahrungsgemäß rasch verändern. Vielmehr wird immer wieder darauf hingewiesen, mit welchen Stichworten man unter Verwendung einer Suchmaschine zu den gewünschten Informationen und Downloads gelangen kann. Weil davon ausgegangen werden kann, dass die Leser über einen vernünftig schnellen Zugang zum Internet verfügen, wird darauf verzichtet, dem Buch eine CD beizulegen. Vielmehr werden alle Programmbeispiele und sonstigen Ressourcen online zur Verfügung gestellt und sind damit immer auf dem neuesten Stand. Aktuelle Informationen zum Buch und Hinweise für den Download findet man unter **[www.oldenbourg.de/verlag](http://www.oldenbourg.de/verlag)** (unter Download) oder **[www.aplu.ch](http://www.aplu.ch)**. Unter dieser Adresse wird auch ein Diskussionsforum (mit einem FAQ) eröffnet. Die Leser sind dazu eingeladen, ihre Fragen oder Bemerkungen im Zusammenhang mit dem Buch in diesem Forum zu veröffentlichen.

Mehreren Personen bin ich zu Dank verpflichtet, denn sie haben in der einen oder anderen Art zum Gelingen dieses Buches beigetragen. Speziell erwähnen möchte ich Frau Kathrin Mönch für die sprachlichen Korrekturen, sowie Frau Margit Roth, Lektorin beim Oldenbourg-Verlag, mit deren freundlicher und umsichtiger Unterstützung die Herausgabe überhaupt erst möglich war.



# 1 Einleitung

Dem Buch liegt ein didaktisches Konzept zu Grunde, das sich beim Selbststudium und in der Kurs- und Unterrichtspraxis über Jahrzehnte bewährt hat. Wie in allen etablierten Fächern gibt es auch in der Informatik methodische und inhaltliche Grundsätze, die alle modischen Trends der Computerentwicklung überdauert haben. Das Buch baut auf folgenden Thesen auf:

## **These 1:**

### **Unterrichte eine moderne, weitverbreitete Programmiersprache**

Für das Programmieren im Informatikunterricht von allgemeinbildenden Schulen gibt es grundsätzlich drei Möglichkeiten:

1. Es wird darauf verzichtet
2. Es wird eine Programmiersprache derart ausgewählt, dass ihre Begrifflichkeit und ihr Umfang weitgehend didaktischen Prinzipien gehorcht. Gegebenenfalls wird eine neue erfunden
3. Es wird eine aktuelle, allgemein verbreitete Programmiersprache gewählt und versucht, die Vermittlung trotz gewisser didaktischer Defizite optimal zu gestalten.

Selbstverständlich gibt es nicht genügend Entscheidungskriterien, um der einen oder anderen Möglichkeit generell den Vorzug zu geben. Immer sind Kompromisse nötig, deren Gewichtung vom Schultyp und damit von den Zielsetzungen der Ausbildung abhängt. Die langjährige Erfahrung mit den mehr didaktisch ausgerichteten Programmiersprachen wie Basic, Pascal, Logo, Eiffel, Modula, Oberon, mit Skriptsprachen wie JavaScript, VisualBasic und mit Sprachen, die nicht für den Unterricht geschaffen wurden, wie Fortran, PL1, Ada, C und C++ u.a. zeigen aber, dass sich für höhere Mittelschulen (Gymnasien), sowie Programmier-einführungen an Hochschulen und Universitäten das pragmatische Vorgehen gemäß der dritten Möglichkeit bewährt hat. Voraussetzung dafür ist allerdings ein vermehrter methodischer Aufwand seitens der Lehrperson, da die Sprachentwickler sich weitgehend von der Problematik der Vermittlung der Sprache an Programmieranfänger abgekoppelt haben.

Dieses Lehrbuch soll zeigen, dass es möglich ist, auch eine nicht didaktisch ausgerichtete Programmiersprache wie Java für den Einstieg in das Programmieren einzusetzen. Durch den weitgehenden Verzicht auf Vorwärtsbezüge und den Willen, keine unverständenen Programmteile oder Begriffe kochbuchartig einzusetzen, sondern Programme in jedem Augenblick des Lernprozesses mit vollem Durchblick auf Grund des aktuellen Wissens und Könnens zu konstruieren, weicht es wesentlich von anderen Lehrmitteln ab.

## These 2:

### Teile den Lehrgang in kleine Lernschritte ohne Vorwärtsbezüge ein

Einem altbewährten Unterrichtsprinzip folgend sollte das Wissen in einer allgemeinbildenden Schule nicht kochbuchartig vermittelt werden. Neue Begriffe werden unter Berücksichtigung des gegenwärtigen Wissensstandes und der Persönlichkeit des Lernenden eingeführt und müssen von ihm logisch nachvollziehbar sein oder ihm zumindest plausibel erscheinen. Der Lernende sollte die beruhigende Sicherheit besitzen, Neues auch wirklich verstanden zu haben. Die Kunst des Unterrichtens besteht darin, den Unterricht in angepasst kleine Lernschritte zu strukturieren, dass der Lernende weder unter- noch überfordert wird.

Ein behutsames Vorgehen ohne Überforderung ist gerade in der Einführungsphase in ein neues Lerngebiet von großer Wichtigkeit, um den Lernenden nicht zu entmutigen. Regelmäßige Erfolge und „Aha“-Erlebnisse fördern seine Motivation und die Bereitschaft zur intellektuellen Anstrengung. Eine zu steile Lernkurve demotiviert, eine zu flache langweilt den Lernenden.

Zu einem systematisch aufgebauten Unterricht gehört der weitgehende Verzicht auf Unerklärbares und auf Hinweise, dass ein Verständnis erst später möglich sei. (Wir nennen solche Hinweise **Vorwärtsbezüge**.) Je nach Fachgebiet stellt dieser Verzicht hohe Anforderungen an die Stoffwahl und den Unterrichtsstil der Lehrperson, vor allem bei der Vermittlung komplexer Zusammenhänge. Im Unterricht mag systematisches Vorgehen nicht immer zwingend sein, bei einem Lehrbuch allerdings schon.

## These 3:

### Zeige Mut zur Lücke

Im naturwissenschaftlichen Unterricht stellt sich laufend die Frage, inwiefern eine Aussage richtig oder falsch sei. Da beispielsweise im Physikunterricht weitgehend die klassische Physik vermittelt wird, sind fast alle besprochenen Gesetzmäßigkeiten vom Standpunkt der Quanten- und Relativitätstheorie falsch oder zumindest nicht exakt. Trotzdem betrachtet man die Vermittlung der klassischen Physik als etwas sehr Wertvolles.

Die Physiklehrpersonen pflegen sich so zu rechtfertigen, dass sie von *Modellannahmen* oder *Voraussetzungen* sprechen, innerhalb derer die Gesetze gültig sind. Beispielsweise setzt man voraus, dass im betrachteten System die Geschwindigkeiten klein gegenüber der Lichtgeschwindigkeit sind, um klassische Mechanik ohne Einbezug der speziellen Relativitätstheorie zu betreiben oder man setzt voraus, dass die „Granularität des Phasenraums“ vernachlässigbar ist und damit die Quantentheorie keine Rolle spielt.

Naturwissenschaftler sind sich also gewöhnt, von unerwünschten Gegebenheiten und Einflüssen großzügig abzusehen, um eine Reduktion der Komplexität zu erhalten. Dabei wird die Möglichkeit offen gelassen, dieselben Fragen später mit einer vollständigeren Theorie nochmals anzupacken. Ähnliche Unterrichtsprinzipien werden auch im Mathematikunterricht angewendet. Es kann beispielsweise sinnvoll sein, die Exponentialschreibweise  $z = r \cdot e^{i\varphi}$  für komplexe Zahlen zu verwenden, bevor man sie mittels komplexer Reihen hergeleitet hat.

Genau dieses Vorgehen, vorerst großzügig von komplexen Zusammenhängen abzusehen, vielleicht sogar gewissen Programmiergrundsätzen zu widersprechen, eignet sich auch bei der Einführung von Programmiersprachen. Vom höheren oder professionellen Standpunkt aus gesehen wird man sich dabei zeitweise „verständigen“, um die Komplexität vorerst zu umgehen. Solche „Auslassungssünden“ sind statthaft und sind der Preis, der bezahlt werden muss, will man Java als Programmiersprache für Anfänger unterrichten. Zur Bewältigung der didaktischen Defizite von Java werden die folgenden Auslassungssünden in den ersten Kapiteln als legitim betrachtet. Das Weglassen von:

- Expliziten Zugriffsbezeichnern (private, protected, public)
- Exceptions
- Packages
- Interfaces
- Threads.

Es stellt sich die Frage, ob es überhaupt noch möglich ist, ohne die ausgeschlossenen Begriffe ein Java-Programm zu erstellen. Trotz dem Einsatz von programmtechnischen Hilfsmitteln gelingt dies leider nicht vollständig. Es kann also nur darum gehen, den Lehrgang derart aufzubauen, dass zu Beginn der Gebrauch der erwähnten Begriffe auf ein absolutes Minimum beschränkt wird. Am Anfang wird auch ein erheblicher Erklärungsbedarf entstehen, um die Vorwärtsbezüge zu vermeiden. Auch muss leider für den Einstieg auf besonders motivierende Themen (beispielsweise auf Applets) verzichtet werden, da diese eine weitgehende Beherrschung von Java voraussetzen. Mit umso mehr Genugtuung ist es aber in einer späteren Phase möglich, sich diesen Gebieten auf der Basis eines soliden Grundlagenwissens widmen zu können.

Bei der Behandlung der einzelnen Themen wird keine Vollständigkeit angestrebt. Dies würde dem exemplarischen Prinzip widersprechen, bei dem davon ausgegangen wird, dass einige gut ausgewählte Beispiele genügen, um das allgemeine Verständnis zu vermitteln. Es wäre zudem eine Verschwendung von Ressourcen, alle Methoden von Klassen aus der Standardbibliothek zu besprechen, da jederzeit auf eine sehr gute Originaldokumentation (allerdings nur in Englisch) zurück gegriffen werden kann. Im Gegensatz dazu sollen alle relevanten Aspekte der modernen professionellen Informatik, speziell im Zusammenhang mit der objektorientierten Programmierung und der Programmiersprache Java aufgegriffen werden, auch wenn dies den Umfang des Lehrbuch beängstigend anwachsen lässt. Es ist aber dem Leser überlassen, einzelne Spezialthemen (Netzwerkprogrammierung, Datenbanken usw.) beim ersten Durcharbeiten zu überspringen.

#### These 4:

**Führe jedes Konzept mittels lauffähiger Programmbeispielen ein**

Für den Lernenden ist es wichtig, neue Konzepte an Hand von Programmbeispielen einzuüben, die auch tatsächlich auf dem Computer ausgeführt werden können. Es handelt sich dabei um das Prinzip des **Learning by doing**, das sich in allen Lehrgebieten bewährt hat. Aus diesem Grund werden im Buch nicht nur Codefragmente (Code-Snippets), sondern fast ausschließlich vollständige Programme aufgenommen, obschon dadurch der Umfang etwas größer wird. Um ein möglichst großes Interessenspektrum abzudecken, stammen die Bei-

spiele aus der Berufspraxis, aber auch aus dem Informatik- und mathematisch-naturwissenschaftlichen Unterricht der Mittel- und Hochschulen.

Die Steilheit der Lernkurve ist für den Lernprozess von großer Bedeutung. Ist sie zu steil, so wird der Lernende nach kurzer Zeit durch die vielen Frustrationen demotiviert, ist sie zu flach, so langweilt er sich. In diesem Buch wird eine eher flache Lernkurve verfolgt, da Java, trotz gegenteiligen Reklameslogans, eine komplizierte Programmiersprache ist, die viel spezifisches Know-how erfordert. Allerdings braucht dies einen gewissen Mut zu Vereinfachungen, was aber nicht mit mangelnder Professionalität verwechselt werden darf. Vielmehr werden die modernen Auffassungen über die objektorientierte Programmierung von Anfang an mit großer Konsequenz vermittelt, beispielsweise durch die Gegenüberstellung von Komposition und Vererbung. In jedem Fall wird einer Problemlösung mittels eines sauberen Klassendesigns der Vorzug gegenüber einer klassischen prozeduralen Lösung gegeben, selbst wenn Letztere wegen der Einfachheit des gerade betrachteten Problems zu einer kürzeren Lösung führen würde.

Die Programme sind, vor allem aus Gründen der Motivation, fast durchwegs fensterbasierte Applikationen und nicht Kommandozeilen-Programme oder Applets. Um den sichtbaren Programmcode in Grenzen zu halten, werden für die grafische Benutzeroberfläche Hilfsklassen eingesetzt. Die Beispiele sollen Vorbilder für korrekte und ästhetische Programme sein und damit den Anfänger zu einem guten Programmierstil anleiten.

## These 5:

### Verwende didaktisch konzipierte Klassenbibliotheken

Der Programmieranfänger ist beim Lesen von längeren Programmen (mit mehr als 10 bis 20 Zeilen Code) sehr oft überfordert und nicht in der Lage, wichtige Teile von weniger wichtigen zu unterscheiden. Er verirrt sich besonders dann in einem „Wald“ von Code, wenn das Programm eine graphische Benutzeroberfläche (ein Graphics User Interface, **GUI**) aufweist. Verglichen mit dem algorithmischen und damit inhaltlich interessanten Teil ist nämlich das GUI mit seiner typischen Ereignissteuerung codeintensiv. Andererseits wäre ein Verzicht auf ein GUI ein Rückschritt in die Urzeiten der Kommandozeilen-Programme und für die Motivation des Lernenden, der sich an maus- und menügesteuerte Benutzeroberflächen gewöhnt hat, verheerend. Ein Ausweg aus diesem Dilemma ist möglich, wenn man geeignete didaktisch konzipierte Klassen einsetzt, die dem Anfänger ebenso natürlich erscheinen wie die vordefinierten Klassen der Programmiersprache. Besonders motivierend ist der Einsatz von Bildschirmgrafik, denn für viele Menschen *sagt ein Bild mehr als tausend Worte*.

Im Buch wird aus diesen Gründen zu Beginn eine Klasse mit Schildkröten (Turtles) eingesetzt, die sich auf dem Bildschirm bewegen. Da man davon ohne weiteres mehrere Exemplare erzeugen kann, eignet sich die Turtleklasse hervorragend für die Einführung in die objektorientierte Programmierung (OOP). Eine auf dem Bildschirm sichtbare Schildkröte wird von jedermann ganz natürlich als ein Objekt aufgefasst, das Eigenschaften (Farbe, Blickrichtung usw.) und Verhalten (gehe vorwärts, drehe nach links usw.) besitzt. Zudem eignet sich die Richtungsgrafik der Turtles hervorragend, um rekursive Muster (Fraktale usw.) zu erzeugen.

Obschon die Turtle mehr als ein Spielzeug ist, kann eine allzu einseitige Ausrichtung des Lehrgangs auf die Turtlegrafik auch kontraproduktiv sein, da der Lernende, gerade wenn er

besonders begabt ist, bereits nach kurzer Zeit aus dem methodisch wohlpräparierten Glashaus ausbrechen möchte, um sich mit den professionellen Programmierern zu messen. Daher wird die Unterstützung durch die didaktischen Klassenbibliotheken sukzessive reduziert und sie werden im zweiten Teil des Buches nach der Behandlung der Swing-Klassen nur noch wenig eingesetzt. Ein Spezialfall ist die Klasse `Console`, die sich immer wieder für alle Arten von Tests, schnelles Ausprobieren von Codeteilen (Prototyping) und Demonstrationen hervorragend eignet.

Bei der Behandlung der Konzepte der OOP wird eine gewisse Vollständigkeit angestrebt. Im Gegensatz dazu wird kein Versuch unternommen, eine Übersicht über die mehr als 2000 Klassen umfassende Java-Bibliothek (**JFC, Java Foundation Class**) zu vermitteln oder einzelne Klassen daraus möglichst vollständig zu beschreiben. Bekanntlich „erdrückt“ die riesige Klassenbibliothek der JFC den Anfänger mehr, als dass sie ihn anspornt. Allerdings werden wichtige Klassen der JFC da besprochen und verwendet, wo dies wegen der Praxisrelevanz notwendig erscheint. Es ist eine wichtige Zielsetzung des Buches, den Lernenden im Laufe des Studiums in die Lage zu versetzen, seine spezifischen Probleme mit den im Buch erworbenen Kenntnissen von Java und unter Beizug der Dokumentation der JFC ohne Verwendung der Hilfsklassen selbständig zu lösen.

## These 6:

### Vermittle die Denkweise der OOP von Anfang an

Seit mehreren Jahren wird in Ausbildungsinstitutionen die Frage diskutiert, ob es für den Programmieranfänger besser sei, zuerst den klassischen Programmierstil weitgehend unter Ausschluss der OOP und erst in einer zweiten Phase die Konzepte der OOP zu erlernen. In der Unterrichtspraxis zeigt sich, dass bei diesem Vorgehen zwar der Einstieg in das Programmieren leichter fällt, dass aber in der zweiten Phase die Akzeptanz der OOP schlecht ausfällt, da sich der Lernende bereits an eine Denkweise gewöhnt hat, die sich fundamental von der OOP unterscheidet. So ist es für den Anfänger nicht immer nachvollziehbar, warum ein Problem für ihn komplizierter mit einem Klassendesign gelöst werden soll, wenn es prozedural einfacher geht. Da die im Unterricht behandelten Probleme meist so kleinen Umfang haben, dass sich die OOP nicht bezahlt macht, ist es zudem nicht leicht, den Lernenden zum Umdenken zu bewegen.

Gewisse Lehrpersonen gehen sogar so weit zu behaupten, dass der klassische prozedurale Programmierstil den Anfänger derart verderbe, dass es für die Einführung in die OOP besser sei, überhaupt keine Programmierzubereitung zu haben. Dies erinnert stark an die polemische Aussage, welche auf den bekannten Informatiker Edsger Dijkstra zurückgeht: „Learning BASIC causes permanent brain damage“<sup>1</sup>. Es ist zwar richtig, dass das Einüben falscher Denkmuster zu vermeiden ist; da aber immer ein gewisser Teil der erlernten Programmier-technik gültig bleibt, sind solche kategorischen Aussagen fraglich. Dabei wird auch außer Acht gelassen, dass der intelligente Mensch durchaus in der Lage ist, in Kenntnis des Schlechten das Gute zu tun.

---

<sup>1</sup> Edsger W. Dijkstra, *Go To Statement Considered Harmful*, Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148

Dieses Buch ist in der Absicht geschrieben, den allgemein anerkannten Prinzipien der OOP von Anfang an und in der Folge überall konsequent treu zu bleiben. Es wird darum nicht versucht, durch programmtechnische Tricks, etwa durch weitgehende Verwendung von static und Verzicht auf Vererbung, die OOP zuerst auf ein Minimum zu reduzieren und in einen zweiten Teil zu verbannen. Vielmehr wird von Anfang an auf einen Programmierstil Wert gelegt, wie er von der professionellen Java-Gemeinschaft gefordert wird. Dadurch ist zwar der Einstieg etwas anspruchsvoller und nicht auf das schnelle Lösen einfacher Programmieraufgaben ausgerichtet. Wegen des weitgehenden Verzichts auf Vorwärtsreferenzen und des exemplarischen Vorgehens ist das Buch aber auch so vom Programmieranfänger, der harmonisch in die OOP hineinwächst, zu bewältigen.

## 2      Daten und Programme

Eine Programmiersprache ist ein Werkzeug, um Prozesse zu formulieren, die sich auf einem Computer ausführen lassen. Es gibt mehrere hundert Programmiersprachen, die sich im Wesentlichen einteilen lassen in

- Imperative Programmiersprachen (auch prozedurale Programmiersprachen genannt)
- Funktionale Programmiersprachen
- Objektorientierte Programmiersprachen.

Bei imperativen Programmiersprachen (typische Vertreter sind Pascal, Modula) steht der sequentielle Ablauf mit Programmanweisungen der Art: *führe aus, falls...dann, wiederhole* im Zentrum. Wichtig ist dabei das sinnvolle Gruppieren von Anweisungen in besonders bezeichneten Programmteilen (Modulen, Routinen, Prozeduren), wodurch man eine überblickbare Struktur des Programms erhält. Diese Strukturierung führt zur Programmierung **vom Großen zum Kleinen (Top-Down)**. Dabei überlegt man sich zuerst, welche Spezifikationen das Programm erfüllen muss, bevor man mit der Formulierung des Programms beginnt. Bei funktionalen Sprachen (typische Vertreter sind Lisp, Logo) interessiert man sich für die mathematische Formulierung des Zusammenhangs zwischen Eingabe- und Ausgabe-größen. Bei objektorientierten Sprachen schließlich (typische Vertreter sind Smalltalk, Eiffel, Java, C++) steht das **Objekt** im Zentrum. Es enthält einerseits **Daten** und bietet andererseits die **Dienste** an, die auf diesen Daten operieren. Die Dienste können zwar auch Aktionen auslösen, sie sind aber nie unabhängig vom Objekt, sondern eben **objekt**-orientiert. Das Programmieren beruht eher auf der Kommunikation zwischen den Objekten, als auf der Anwendung von Funktionen auf Daten.

Java wird zwar zu den objektorientierten Sprachen gezählt, da man kein Programm ohne Objekte schreiben kann. Es ist aber keineswegs so, dass mit der **objektorientierten Programmierung (OOP)** andere Programmiertechniken überflüssig werden. Vielmehr muss auf weiten Strecken auch in Java imperativ und funktional programmiert werden. Daher werden wir Wert darauf legen, auch imperative und funktionale Programmiertechniken zu pflegen.

## 2.1 Formulierung von Algorithmen

Bevor wir uns mit den Einzelheiten der Programmiersprache Java befassen, ist es sinnvoll, sich einen allgemeinen Überblick über die Funktionsweise eines Computers zu verschaffen. Dieses Vorgehen vom Allgemeinen zum Speziellen hat sich beim Einstieg in ein neues Fachgebiet gut bewährt, da es eine gedankliche Struktur schafft, an der wir uns jederzeit orientieren können.

Allgemein betrachtet dient ein Computer dazu, Informationen, die in Form von digitalen Daten vorliegen, wunschgemäß zu verarbeiten. Die Eingabedaten werden dabei nach einer Verarbeitungsvorschrift, die wir einen **Algorithmus** nennen, bearbeitet und als Ausgabedaten zur Verfügung gestellt. Als Eingabemedium dient im einfachsten Fall die Tastatur, als Ausgabemedium der Bildschirm. Der Algorithmus kann sehr unterschiedlich formuliert werden, beispielsweise umgangssprachlich. Wollen wir ihn aber auf einem Computer ausführen, so müssen wir ihn in einer Programmiersprache formulieren. Wir sagen, dass wir den Algorithmus in einer Programmiersprache **implementieren**. Damit der Prozessor des Computers das Programm ausführen kann, muss es in der für jeden Prozessor typischen Maschinensprache vorliegen. Diese enthält aus Effizienzgründen nur maschinennahe Befehle und ist daher für die Formulierung von Algorithmen ungeeignet. Wir schreiben daher das Programm in einer prozessorunabhängigen **Höheren Programmiersprache**. In einer nachfolgenden Phase muss das Programm in Maschinensprache umgewandelt (übersetzt, kompiliert) werden.

Nicht nur in der Mathematik, sondern auch in der Informatik ist der Begriff der **Funktion** von großer Wichtigkeit. Im allgemeinen Fall übernimmt eine Funktion gewisse Werte, die wir **Parameter** (oder **Argumente**) nennen, verarbeitet diese und liefert einen **Rückgabewert** an den Aufrufenden zurück. Im Spezialfall kann eine Funktion allerdings auch Aufgaben ausführen, ohne einen Rückgabewert zu liefern. (In funktionalen Sprachen heissen die Auswirkungen **Seiteneffekte**.) Funktionen, die keinen Rückgabewert liefern, nennt man in gewissen Programmiersprachen **Prozeduren**, in Java sagt man, dass die Funktion in diesem Fall `void` (nichts) zurückgibt. Statt von Funktionen werden wir durchwegs gemäß der in Java üblichen Terminologie von **Methoden** sprechen.

*In gewissen Programmiersprachen nennt man diese in sich geschlossenen Programmmodule auch **Unterprogramme** oder **Subroutinen**.*

Da das Programm immer auf Daten einwirkt, ist es offensichtlich, dass Daten und Programme eng miteinander verknüpft sind. Dieser direkte Zusammenhang wird in objektorientierten Programmiersprachen besonders betont, indem die Daten und die darauf operierenden Methoden zu einem **Objekt** zusammengefasst werden.



## 2.2 Die algorithmischen Grundstrukturen

Eine höhere Programmiersprache besitzt im Vergleich zu Umgangssprachen einen extrem kleinen Wortschatz. Die Wörter mit einer fest definierten Bedeutung nennen wir **Schlüsselwörter**. Die Verwendung der Schlüsselwörter zur Bildung von Ausdrücken (die **Syntax** oder **Grammatik**) ist exakt definiert, damit es nie zu Mehrdeutigkeiten kommt. Ein syntaktisch falsches Programm wird gar nicht erst zum Laufen kommen, ein syntaktisch richtiges Programm kann sich aber zur Laufzeit immer noch falsch verhalten. Wenn das Programm auch zur Laufzeit fehlerfrei ist, nennen wir es **semantisch** richtig.

Gemäß einem sehr allgemeinen Prinzip der Informatik, dem Theorem von Böhm und Jacopini<sup>1</sup>, genügen drei Grundstrukturen, um irgend einen auf einer Maschine ablaufenden Algorithmus zu beschreiben. Es handelt sich um

- Sequenz
- Selektion
- und Iteration.

Die Sequenz ist eine Aneinanderreihung von Aktionen, die streng zeitlich hintereinander ablaufen. Damit werden parallel ablaufende Prozesse ausgeschlossen, was für viele Aufgaben eine sinnvolle Beschränkung ist. Bei der Selektion wird der Ablauf auf Grund von gewissen Bedingungen verzweigt, und bei der Iteration handelt es sich um die Wiederholung gewisser Ablaufblöcke.

Die Beschreibung zeitlicher Abläufe ist auch im täglichen Leben von großer Wichtigkeit. Ein typisches Beispiel ist ein Kochrezept. Ein Rezept kann zwar in verschiedenen Sprachen abgefasst werden, sollte aber immer das gleiche Gericht ergeben. Analog dazu muss ein Algorithmus in verschiedenen Programmiersprachen abgefasst zum gleichen Resultat führen. Das Gericht hat möglichst unabhängig von der Köchin oder dem Koch zu sein, analog muss ein Algorithmus auf verschiedenen Computersystemen (**Plattformen**) gleiche Resultate liefern.

## 2.3 Klassen und Objekte

Da wir in Java schon von Beginn an objektorientiert programmieren müssen, wollen wir uns bereit jetzt näher mit dem Begriff des Objekts befassen. Glücklicherweise können wir uns dabei an Objekten aus unserem täglichen Leben orientieren. Hier betrachten wir beispielsweise Personen, Tiere, Alltagsgegenstände, Fahrzeuge, Musikinstrumente usw. als Objekte. Wir können mit Java solche Objekte softwaremäßig **modellieren**, was zu einer **Abstraktion** der realen Welt führt. Wie bei jedem Modell werden dabei gewisse Aspekte der Wirklichkeit richtig, andere nur lückenhaft oder sogar fehlerhaft abgebildet. Beim Übergang vom Modell

---

<sup>1</sup> Böhm C., Jacopini G., *Flow diagrams, turing machines and languages with only two formation rules*, Communications of the ACM 9(5), 366-371 (1966)

zur Wirklichkeit benötigen wir eine **Interpretation** der Modellgrößen in der realen Welt (Abb. 2.1).

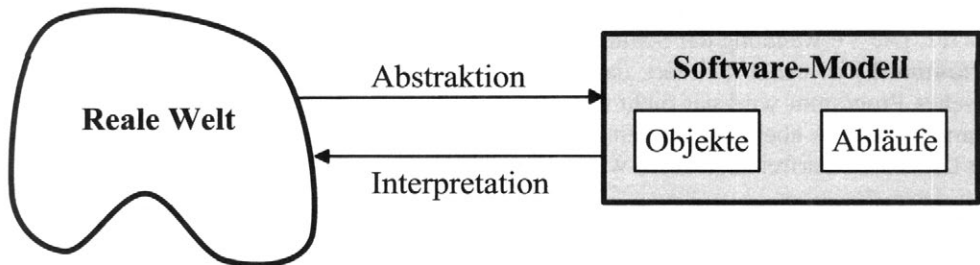


Abb. 2.1 Software-Modell

Bei der Abstraktion wird die Wirklichkeit vereinfacht und angenähert, indem nur diejenigen Aspekte betrachtet werden, die für die spezifische Aufgabenstellung wichtig sind. Es hat sich gezeigt, dass in vielen Fällen zwei Aspekte genügen, um ein reales Objekt softwaremäßig zu modellieren, nämlich seine **Eigenschaften** und sein **Verhalten**.

Die **Eigenschaften** (oder **Attribute**) beschreiben den aktuellen **Zustand** des Objekts. (Beispiel: Die Person ist weiblich und zwanzigjährig. Die Katze hat Hunger. Das Flugzeug steht auf der Rollbahn.) Ein Objekt kann auch aus anderen Objekten, aus **Komponenten**, aufgebaut sein. Man spricht in diesem Zusammenhang auch von der **Komposition** eines Objekts. Die Komponenten zählt man ebenfalls zu den Eigenschaften des Objekts und man sagt, dass es die Komponenten *enthalt*. (Beispiel: Die Person besteht aus Körperteilen, die Katze trägt ein Halsband, das Flugzeug besitzt Flügel.) In Java werden die Eigenschaften bzw. Attribute durch die **Instanzvariablen** beschrieben. Wir sprechen daher durchwegs von Instanzvariablen statt von Eigenschaften oder Attributen.

Das **Verhalten** beschreibt die Handlungsfähigkeiten des Objekts. (Beispiel: Die Person kann skifahren. Die Katze kann miauen. Das Flugzeug kann fliegen.) In Java wird das Verhalten durch die **Methoden** beschrieben (Abb. 2.2).

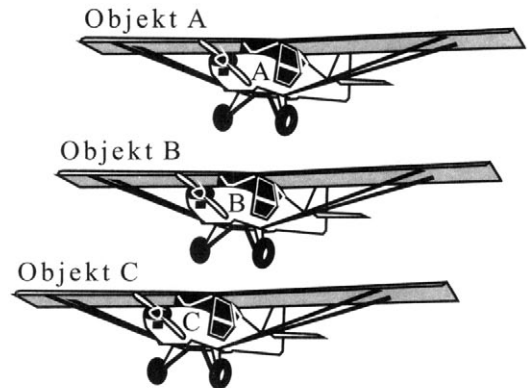
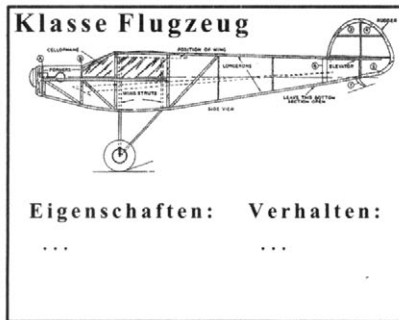


Abb. 2.2 Ein Objekt besitzt Eigenschaften und Verhalten

Zu jedem Objekt gehört ein **Bauplan**, welcher die Eigenschaften und das Verhalten beschreibt. Wir nennen diesen in Java eine **Klassendeklaration** und sagen, dass das Objekt zu dieser **Klasse** gehört. Das Objekt selbst ist eine Ausprägung oder eine Realisierung der Klasse, wir sprechen von einer **Klasseninstanz** oder kurz **Instanz**. Objektorientiertes Programmieren besteht also insbesondere darin, Klassen mit geeigneten Instanzvariablen und Methoden zu deklarieren und nachfolgend Instanzen zu **erzeugen** und zu verwenden.

In der Regel werden die Instanzen einer Klasse gewisse **Daten** und **Dienste (services)** anderen Klassen (**clients**) zur Verfügung stellen. Die Clients brauchen dabei die interne Struktur der Service-Klasse nicht zu kennen, sondern nur deren **Schnittstelle (interface)** nach außen. Man könnte daher eine Klasse auch als einen **Dienstanbieter (service provider)** auffassen.

Durch das Zusammenpacken von Instanzvariablen und Methoden erhält man eine übersichtliche **Datenstruktur** und es ist möglich, genau zu definieren, welche Teile eines Programms die Erlaubnis haben, Instanzvariablen zu lesen oder zu verändern. Die Eigenschaften eines Objekts werden dadurch vor unerlaubtem Zugriff von außen **geschützt** und die exakte Realisierung des Verhaltens nach außen **geheim** gehalten. Man spricht von **Datenkapselung (encapsulation, information hiding)** und auch vom **Geheimnisprinzip**.

## 2.4 Der Entwicklungsprozess eines Programms

Wir verfassen ein Java-Programm mit einem Texteditor, der normalerweise der Programmiersprache Java angepasste Eigenschaften besitzt, und speichern den Text als Datei auf der Festplatte. Man nennt diese Datei den **Quellcode (Sourcecode)** des Programms.

Als Nächstes wird der Quellcode mit Hilfe eines Systemprogramms, genannt Java-Compiler, **übersetzt (compiliert)**, und zwar in einen Zwischencode (**Java-Bytecode**). Damit dieser von der aktuellen Entwicklungsplattform unabhängig ist, enthält er keinen rechner-spezifischen Maschinencode, ist also nicht direkt ausführbar. Vielmehr erfolgt die Ausführung mit Hilfe einer Systemapplikation, der **Java Virtual Machine (JVM)**, die zusammen mit dem

**Java Runtime Environment (JRE)** installiert wird. Bei der **Ausführung** wird der Bytecode von der JVM Schritt um Schritt in Maschinencode übersetzt und erst dann vom Prozessor ausgeführt (der Bytecode wird **interpretiert**) (Abb. 2.3).

*Die JVM kann als eine Abstraktion einer CPU eines realen Computers aufgefasst werden, wobei der Bytecode dem Maschinenbefehlssatz entspricht. Tatsächlich entspricht der Bytecode weitgehend dem Konzept von Maschinensprachen. Der Name ist darauf zurückzuführen, dass jede Instruktion aus einem Opcode von einem Byte und nachfolgenden Operanden besteht.*

*Das Interpretieren des Bytecodes erfolgt, wie in einer von Neumannschen Rechnerarchitektur üblich, in einer Wiederholschleife von Holen-Interpretieren-Ausführen (fetch-decode-execute) und verlangsamt die Ausführungszeit eines Java-Programms um 10 bis 50 mal gegenüber einem entsprechenden Programm in Maschinencode. Für viele Anwendungen spielt dies eine untergeordnete Rolle. Grundsätzlich lässt sich der Bytecode auch in Maschinencode übersetzen, bevor er ausgeführt wird. Dies kann bereits zur Zeit der Entwicklung des Programms erfolgen oder erst beim Starten des Programms (Just-In-Time Compilation, JIT). Dadurch wird die Ausführungszeit wesentlich verbessert. Sie liegt dann in der Größenordnung von Programmen in bekannten compilierten Programmiersprachen wie Fortran und C/C++.*

*Die Verwendung von Bytecode besitzt einen großen Vorteil im Zusammenhang mit kleinen Geräten und Systemen, die durch einen internen Microcontroller gesteuert werden (Mobiltelefone, Haushaltgeräte, Personal Digital Assistants, Kleinroboter usw.). Das Java Programm wird dabei auf einem Hostsystem entwickelt und compiliert, anschließend in das Zielsystem geladen und dort von einer kleinen Java Virtual Machine (JVM) interpretiert. In solchen Geräten kommen sogar Java-Chips zu Einsatz, welche den Bytecode direkt ausführen.*

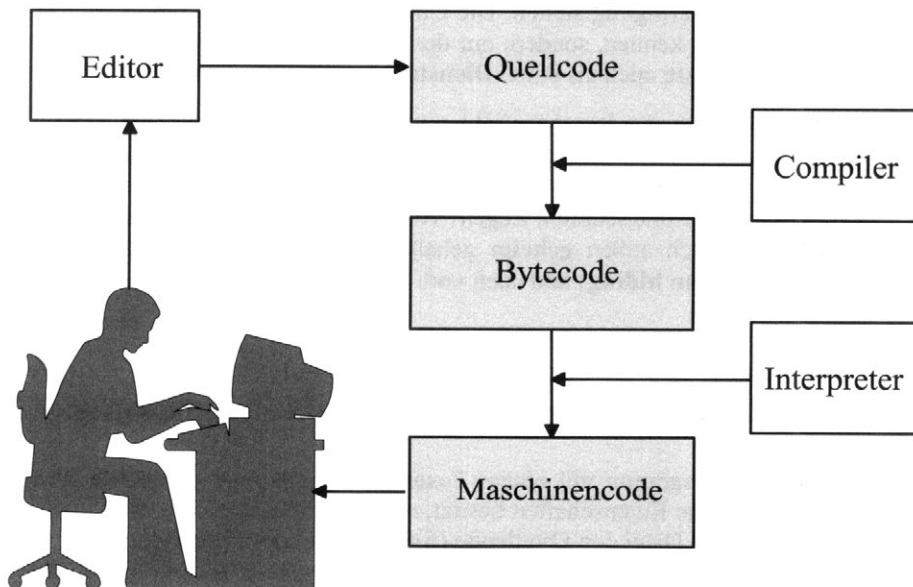


Abb. 2.3 Entwicklungsphasen eines Java-Programms

Um die Arbeit des Programmierers zu vereinfachen, erfolgt der ganze Entwicklungsprozess normalerweise innerhalb einer **Entwicklungsumgebung** (Integrated Development Environment, **IDE**).

*Für die Installation und die Verwendung einer IDE wird auf das Internet verwiesen, wo es leicht ist, sich einen Überblick über die aktuellen Java-Entwicklungsumgebungen für eine spezifische Plattform zu verschaffen. Für Lernzwecke gut bewährt haben sich die kostenfreien Versionen von Borland's JBuilder und Eclipse. Die Installation und Bedienung wird in zahlreichen Publikationen beschrieben und ist nicht Teil dieses Lehrbuchs<sup>1, 2</sup>.*

---


<sup>1</sup> Tabatt P., Wolf H., *Java programmieren mit JBuilder*, Software und Support (2004)

<sup>2</sup> Daum B., *Java-Entwicklung mit Eclipse 2*, dpunkt.verlag (2003)

# 3 Das erste Java Programm

## 3.1 Konventionen und Schlüsselwörter

Ein Java-Quellprogramm besteht aus einer oder mehreren zusammengehörenden Textdateien unter Einhaltung der **Java-Syntaxregeln**. Es lässt sich eine exakte, formale Sprachdefinition angeben, die festlegt, welche Texte auch tatsächlich syntaktisch korrekte Java-Programme sind. Für die Praxis ist diese exakte Sprachdefinition allerdings nicht sehr nützlich, da man sich bereits nach kurzer Zeit an die Sprachregeln gewöhnt hat. Im Wesentlichen besteht ein Quellprogramm aus eine Reihe von vordefinierten Wörtern, den **Schlüsselwörtern**, aus eigenen **Bezeichnern (Identifiers)** und **Operationszeichen**. Mit dem Zeilenumbruch und den Leerzeichen kann sehr freiheitlich umgegangen werden. Daher bleibt die Darstellung des Programms weitgehend dem Programmierer überlassen. Seiner Phantasie sind aber Grenzen gesetzt, denn unter professionellen Programmierern hält man sich an verbindliche Abmachungen (**programming style guide**), damit der Quellcode für alle gleichermaßen lesbar ist und leichter ausgetauscht werden kann. Auch wenn man über diese Konventionen unterschiedlicher Meinung sein kann, so formulieren wir eines der wichtigsten Prinzipien der erfolg- und genussreichen Programmierung:

 **Ein Programm muss nicht nur korrekt laufen, sondern auch schön geschrieben sein.**

Für die frei wählbaren Bezeichner halten sich Java-Programmierer strikt an folgende Regeln:

- Erlaubte Zeichen sind die Klein- und Großbuchstaben des Alphabets, die Zahlen und als einziges Spezialzeichen der Underline `_`. Insbesondere sind Umlaute, Akzente und Leerschläge nicht erlaubt
- Klassen werden wie Substantive großgeschrieben. Beispiel: `Turtle`
- Variablen und Methoden werden wie Verben kleingeschrieben. Beispiel: `forward`
- Lange zusammengesetzte Bezeichner werden bei Wortgrenzen wieder großgeschrieben. Beispiel `setColor, myTurtle`
- Selbsterklärende Wortbezeichner werden den mathematisch gebräuchlichen Buchstabenbezeichnern vorgezogen. Oft verwendet man als Bezeichner den kleingeschriebenen Klassennamen. Beispiel `turtle`



**Bezeichner sollten selbsterklärend sein, auch wenn dies zu einer größeren Schreibaufwand führt.**

In Java wird jede Anweisung mit einem Strichpunkt abgeschlossen. Wir gewöhnen uns daran, jede Anweisung auf eine neue Zeile zu schreiben. Dadurch gewinnen wir an Übersichtlichkeit und können uns für Diskussionen auf die Zeilennummer beziehen. Eine Zeile sollte aber nicht mehr als 80 Zeichen enthalten, damit sie ganz auf dem normal eingestellten Bildschirm sichtbar ist und problemlos ausgedruckt werden kann. (In diesem Buch müssen wir wegen des Layouts die Zeilenlänge sogar auf 62 Zeichen beschränken, was manchmal zu etwas künstlich wirkenden Zeilenumbrüchen führt.) Ein Block von Anweisungen muss in Java mit dem Klammerpaar `{ }` zusammengefasst werden. Für die Klammerung verwenden wir die Konvention, wonach zusammen gehörende Start- und Endklammern übereinander liegen, trotzdem viele Java-Programmierer die öffnende Klammer eine Zeile höher setzen, um eine Zeile zu „sparen“.

Eine Programmiersprache besitzt im Gegensatz zu einer Umgangssprache einen extrem kleinen Wortschatz. Da Java leider keine standardisierte Programmiersprache ist, kann sich die Anzahl der **Schlüsselwörter (keywords)** in verschiedenen Java-Versionen leicht ändern. In der vorliegenden Java-Version 1.4 gibt es 49 Schlüsselwörter und die 3 **reservierten Wörter** `true`, `false`, `null`. Die Schlüsselwörter sind aus Tab. 3.1 ersichtlich.

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

**Tab. 3.1** Java-Schlüsselwörter  
(`const` und `goto` werden in den vorliegenden Sprachversionen nicht verwendet.)

Schlüsselwörter dürfen nur in ihrer fest vordefinierten Bedeutung verwendet und nie für eigene Bezeichner eingesetzt werden. Wir werden uns auch daran gewöhnen müssen, dass die Groß-Kleinschreibung strikt eingehalten werden muss. Bezeichner, die sich bereits durch einen einzigen andersgeschriebenen Buchstaben wie beispielsweise `setX` und `setx` unterscheiden, werden als verschieden betrachtet.

Eines der wichtigsten Elemente eines Quellprogramms ist der einzeilige **Kommentar**. Es handelt sich um einen beliebigen Text, der in den meisten (aber nicht allen) Programmierumgebungen auch Umlaute, Akzente und Spezialzeichen umfassen darf. Dieser wird vom Compiler ignoriert und dient nur zur Dokumentation des Quellprogramms. Ein einzeiliger Kommentar stellt ein wichtiges Dokumentationsmittel dar. Er wird mit einem verdoppelten Bruchstrich `//` eingeleitet und nimmt den Rest der Zeile ein. Der Kommentar kann an beliebiger Stelle der Zeile beginnen.

## 3.2 Grundaufbau eines Java-Programms

Mit den erarbeiteten Vorkenntnissen sind wir in der Lage, ein erstes Java-Programm zu verstehen und auszuführen. Wir wollen dabei den Computer den Preis `p` berechnen lassen, der sich aus der gegebenen Stückzahl `z` und dem gegebenen Einheitspreis `e` eines Artikel ergibt.

Die ersten Entscheide, die wir beim Schreiben eines Programms treffen müssen, sind der Name und die Dateierweiterung der Quelldatei. Wir wählen als Namen `Preis` und als Dateierweiterung `.java`. Mit dem Quellcode-Editor editieren wir die Datei `Preis.java` und vermerken als Erstes in einem Kommentar den Namen der Quelldatei und auf weiteren Kommentarzeilen eventuell wichtige Informationen, wie der Name des Programmierers oder die Versionsnummer des Programms. Damit können wir das Quellprogramm leichter auffinden und laufen weniger in Gefahr der Verwechslung mit einer früheren, veralteten Version. Als Nächstes deklarieren wir mit dem Schlüsselwort `class` eine Klasse, wobei wir als Bezeichner den Namen der Quelldatei wählen, was in vielen Entwicklungsumgebungen vorge-schrieben ist.

```
// Preis.java

class Preis
{
}
```


Damit das Java-Laufzeitsystem weiß, welcher Code beim Programmstart als Erstes auszuführen ist (**entry point**), deklarieren wir in dieser Klasse eine Methode mit dem vorgegebenen Namen `main()`. Wir werden eine Klasse mit der Methode `main()` die **Applikationsklasse** (oder **Hauptklasse**) nennen, um sie von anderen Klassen zu unterscheiden. `main()` wird nie einen Wert für weitere Berechnungen zurückgeben, und wir sagen deshalb, dass ihr Rückgabetyt `void` (leer) ist.

Das Betriebssystem sollte die Möglichkeit erhalten, der Methode `main()` gewisse Werte beim Programmstart zu übergeben (**Kommandozeilen-Parameter**). Diese werden als eine



Aufzählung von Wörtern aufgefasst. Für ein Programm, das E-Mails verschickt, sind dies beispielsweise Absender- und Empfängeradresse. Der Datentyp eines einzelnen Wortes ist `String`, für eine Wortaufzählung `String[]`. Als Parameterbezeichner ist `args` üblich (aber nicht zwingend). Leider müssen wir diese Übergabe auch vorsehen, selbst wenn wir, wie in unserem Fall, gar keine Werte übergeben wollen.

In der Welt der objektorientierten Programmierung ist es üblich, dass von einer Klasse mehrere Instanzen existieren. Falls aber die Methode `main()` in mehreren Instanzen vorkommt, ist es für das Betriebssystem nicht klar, welche davon bei Programmstart aufzurufen ist. Um diese Mehrdeutigkeit zu verhindern, verlangen wir, dass die Methode `main()` nur **einmal in der Klasse** und nicht in jedem Objekt vorkommen soll und drücken dies mit dem Schlüsselwort `static` aus.

 Eine Methode, die `static` ist, kommt nur einmal in der Klasse und nicht in jedem Objekt vor.

Schließlich müssen wir dem Betriebssystem noch die Erlaubnis erteilen, überhaupt auf die Methode `main()` zuzugreifen. Wir drücken dies durch das Schlüsselwort `public` aus und erhalten endlich den sehr gewöhnungsbedürftigen Ausdruck

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

Das Programmgerüst erhält die folgende, für alle Java-Programme fundamentale Form:

```
// Preis.java

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

Dies ist bereits ein syntaktisch korrektes Java-Programm, das allerdings überhaupt nichts tut. Viele professionelle Entwicklungsumgebungen erzeugen solche Vorgabegerüste automatisch und ersparen uns die Schreibarbeit. Aber einige wenige **Anweisungen** in der Methode `main()` genügen, um die Berechnung des Preises auszuführen. Als Erstes benötigen wir aber **Platzhalter** für die drei numerischen Größen `z`, `e` und `p`. Wir nennen sie **Variablen**, da sie verschiedene Werte annehmen können. In Java müssen wir die Variablen zudem **deklieren**, das heißt angeben, aus welchem Wertebereich die Variablenwerte stammen dürfen. Durch Angabe des **Datentyps** der Variablen wird im Hauptspeicher des Computers ein Speicherplatz reserviert, der genau die angepasste Größe besitzt. Zudem kann das Übersetzungsprogramm, der **Compiler**, kontrollieren, ob Flüchtigkeitsfehler vorliegen, indem der Programmierer beispielsweise versucht, inkompatible Datentypen zu addieren.

Für Dezimalzahlen verwenden wir als Datentyp das Schlüsselwort **double** und legen für **z** und **e** auch gerade ihren Anfangswert, den **Initialisierungswert**, fest. Wir **deklarieren** und **initialisieren** **z** und **e** mit

```
double z = 5731;  
double e = 2.55;
```

und deklarieren **p** mit

```
double p;
```

Schließlich berechnen wir **p** aus dem Produkt von **z** und **e** und schreiben in üblicher algebraischer Darstellung  $p = z * e$ , wobei das Multiplikationszeichen **\*** im Gegensatz zur Algebra obligatorisch ist, also

```
p = z * e;
```

Diese Zeile nennen wir eine **Zuweisung** von **p**. Es handelt sich also nicht um eine algebraische Gleichung, sondern um eine Anweisung, die vom Computer verlangt, die Werte von **z** und **e** zu multiplizieren und das Resultat an die Speicherstelle von **p** zu setzen oder, wie man sagt, der Variablen **p** zuzuweisen. Im Gegensatz zu einer algebraischen Gleichung kann offensichtlich auf der linken Seite einer Zuweisung immer nur eine einzelne Variable stehen.

Für die Ausgabe des Wertes von **p** wollen wir ein einfaches Ausgabefenster heranziehen, das uns Java für Testzwecke zur Verfügung stellt. Dabei verwenden wir die Methode `print()`, die zu einer (statischen) Instanzvariablen `out` der Klasse `System` gehört. Wir drücken diesen Zusammenhang mit dem **Punktoperator** aus und schreiben

```
System.out.print(p);
```

Unser erstes Programm ist nun vollständig zusammengestellt und wir können es editieren und ausführen.

```
// Preis.java
```

```
class Preis  
{  
    public static void main(String[] args)  
    {  
        double z = 5731;  
        double e = 2.55;  
        double p;  
  
        p = z * e;  
        System.out.print(p);  
    }  
}
```

*Es ist instruktiv, den Bytecode (in Assembler-Schreibweise) anzusehen, der sich nach der Compilation von Preis.java in Preis.class befindet. Dazu ruft man den Kommandozeilen-Programm `javap -c Preis` auf. Man erkennt leicht die in Bytecode umgesetzte Multiplikation der beiden Größen.*

```
Compiled from Preis.java
class Preis extends java.lang.Object {
    Preis();
    public static void main(java.lang.String[]);
}

Method Preis()
    0 aload_0
    1 invokespecial #1 <Method java.lang.Object()>
    4 return

Method void main(java.lang.String[])
    0 ldc2_w #2 <Double 5731.0>
    3 dstore_1
    4 ldc2_w #4 <Double 2.55>
    7 dstore_3
    8 dload_1
    9 dload_3
   10 dmul
   11 dstore 5
   13 getstatic #6 <Field java.io.PrintStream out>
   16 dload 5
   18 invokevirtual #7 <Method void println(double)>
   21 return
```

### 3.3 Verwendung von Klassenbibliotheken

In Java gehört die Deklaration und Verwendung von Klassen zu den fundamentalen Tätigkeiten des Programmierers. Dabei ist es wesentlich leichter, vordefinierte Klassen zu verwenden als sie selbst zu erstellen. Es ist durchaus legitim, vordefinierte Klassen wie eine Blackbox zu verwenden, ohne dass man den Quellcode dieser Klassen kennt. Lediglich die **Schnittstelle** zu anderen Teilen des Programms muss exakt bekannt sein. Diese umfasst im Wesentlichen alle Instanzvariablen und Methoden, die von Außen zugänglich sind, sowie den Zusammenhang mit anderen Klassen. Dazu benötigen wir eine ausführliche Dokumentation, die gedruckt oder elektronisch auf der Festplatte oder im Internet vorliegen muss. Zweckmäßig sind Entwicklungsumgebungen, in denen ein kontextsensitives Hilfe-System integriert ist. Wir brauchen den Cursor im Quellcodefenster nur auf einen Begriff zu setzen und eine Spezialtaste zu drücken, damit ein Hilfetext über diesen Begriff erscheint.

Da Klassenbibliotheken in der Regel mehrere Klassen enthalten, fasst man diese in einem **Package** zusammen. Dieses besitzt einen Namen, der wie eine **URL (Uniform Resource**

**Locator**) einer Web-Seite aufgebaut ist. Gemäß einer Vereinbarung muss sich der Java-Bytecode der Klassen eines Packages auf der Festplatte in einem **Verzeichnisbaum** befinden, das dieser URL entspricht.

Um ein Programm zu compilieren, das Klassen verwendet, die nicht zur Java Foundation Class (JFC) gehören (diese werden automatisch mit Java installiert), muss der Bytecode dieser Klassen in einem Verzeichnisbaum (oder in einer jar-Datei) vorhanden sein. Zur Verwendung werden sie mit dem Schlüsselwort `import` **importiert**. Statt jede Klasse einzeln anzugeben, kann man für mehrere Klassen **im selben Verzeichnis** (aber nicht für Klassen in Unterverzeichnissen) den Joker `*` verwenden.

Für dieses Buch wurde zu didaktischen Zwecken eine Klassenbibliothek mit Hilfsklassen entwickelt, die den Einstieg in die Programmierung wesentlich vereinfachen. Um die Klassen aus dem Package `ch.aplu.util` bei der Entwicklung von Programmen verwenden zu können, ist zu Beginn des Quellprogramms die Compileranweisung

```
import ch.aplu.util.*;
```

nötig. Zudem muss das Package auf dem Rechner gemäß den Angaben im Anhang 1 installiert werden.

Im Package `ch.aplu.util` befinden sich mehrere für Lernzwecke entwickelte Klassen. Wir verwenden als Erstes die Klasse `Console`, welche es uns ermöglicht, in einem echten Bildschirmfenster, in Zukunft **Console-Fenster** genannt, zeilenweise auszuschreiben und einzulesen. Einem bereits geschriebenen Programm brauchen wir nur

```
Console.init();
```

vorzustellen, um alle Ausgaben von `System.out` in das Console-Fenster **umzuleiten** (man spricht in der Unix-Welt von einer **pipe**). Manchmal ist es auch sehr zweckmäßig, dass man alle Ausgaben durch die Angabe des Dateinames in eine Textdatei umleiten kann, also beispielsweise mit

```
Console.init("debug.txt");
```

in eine Datei `debug.txt`, die im selben Verzeichnis, in der sich das Programm befindet, neu erzeugt wird. Da die Klasse `Console` auch alle `print`-Methoden selbst implementiert, kann man diese an Stelle von `System.out` verwenden. Beim ersten Aufruf einer solchen Methode erscheint das Console-Fenster automatisch auf dem Bildschirm.

```
// PreisEx1.java

import ch.aplu.util.*;

class PreisEx1
{
    public static void main(String[] args)
    {
```

```
double z = 5731;
double e = 2.55;
double p;

p = z * e;
Console.print("Der Preis betraegt: ");
Console.print(p);
}
```

Bei der Verwendung der Klasse `Console` gilt es Folgendes zu beachten:

- Umlaute und Akzente werden nicht unterstützt
- Man kann die Ausgaben eines bestehenden Programms, welches die `print`-Methoden aus `System.out` benutzt, ohne Änderung des Codes in die `Console` umleiten, indem man zu Beginn `Console.init()` aufruft
- Die `print`-Methoden können aneinandergefügt werden, wie beispielsweise im vorhergehenden Beispiel

```
Console.print("Der Preis begtraegt: ").print(p);
```

Statt `import` zu verwenden, kann man auch den vollständigen Klassennamen einsetzen. Zudem wird das Ausschreiben einfacher, falls in der Parameterklammer von `print()` mehrere Strings mit **+ zusammenfügt (konkateniert)**. `PreisEx2` zeigt diese Varianten.

```
// PreisEx2.java

class PreisEx2
{
    public static void main(String[] args)
    {
        ch.aplu.util.Console.init();
        double z = 5731;
        double e = 2.55;
        double p;

        p = z * e;
        System.out.print("Der Preis betraegt: " + p);
    }
}
```

Weitergehende Möglichkeiten der Klasse `Console` entnimmt man der Klassen-Dokumentation.

## 3.4 Turtlegrafik, das Schlüsselwort new

Die Turtlegrafik eignet sich gut, um die Grundkonzepte einer objektorientierten Programmiersprache auf spielerische und anschauliche Art zu vermitteln. Schildkröten sind ganz offensichtlich Objekte mit Eigenschaften (Farbe, Position usw.) und Verhalten (können sich vorwärts und rückwärts bewegen, sich drehen usw.). Ein **Turtleobjekt** besitzt ein Bildschirmfenster (**Playground**), auf dem sie sich bewegt. Dabei zeichnet sie mit ihrem Schreibstift (**pen**) eine Spur, mit der man die Bewegung nachvollziehen kann. Der Kopf der Turtle zeigt immer in die Bewegungsrichtung, in der sich die Turtle mit dem Befehl (der Methode) `forward(distanz)` bewegt. Die Bewegungsrichtung kann mit dem Befehl `left(winkel)` bzw. `right(winkel)` geändert werden. Mehrere Turtles können sich als zweidimensionale Figuren im selben Playground bewegen, ohne sich gegenseitig zu beeinflussen. Die Sichtbarkeit ist so geregelt, dass sich eine aktuell bewegte Turtle immer oberhalb aller anderen befindet, wobei alle darunter liegenden Turtles und Spuren beim Wegziehen wieder sichtbar werden.

In Java erzeugt man eine **Instanz** (oder ein **Objekt**) einer Klasse mit dem Schlüsselwort `new`. Wir sagen auch, dass wir mit `new` ein Objekt **instancieren**. Dabei können wir uns vorstellen, dass für das Objekt im Hauptspeicher des Computers (auf dem so genannten **Heap**) Speicherplatz reserviert und die Eigenschaften des Objekts auf einen vorgegebenen **Standardwert (default value)** gesetzt werden. Gleichzeitig wird die Turtle als Figur mit diesen Eigenschaften auf dem Bildschirm sichtbar. Um das neu erzeugte Objekt anzusprechen, gibt `new` eine **Referenz** auf das erzeugte Objekt zurück. Anschaulich können wir uns diese wie eine „Fernsteuerung“ vorstellen, mit der wir mit dem Objekt kommunizieren. Diesen Rückgabewert weisen wir einer Variable von Typ `Turtle` zu, damit wir nachher mit dieser „Fernsteuerung“ arbeiten können. Statt beide Vorgänge mit

```
Turtle john;  
john = new Turtle();
```

in zwei Schritten auszuführen, können wir auch einfacher

```
Turtle john = new Turtle();
```

schreiben (Abb. 3.1).

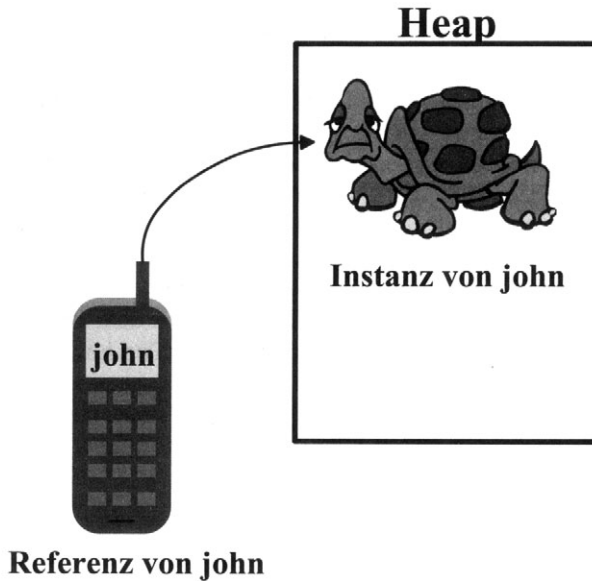


Abb. 3.1 Instanz und Referenz

Wir werden aber der Einfachheit halber öfters „vergessen“, dass es sich bei `john` nur um eine „Fernsteuerung“ für das Turtleobjekt handelt und uns gelegentlich erlauben, etwas salopp von der **Instanz** `john` oder vom **Objekt** `john` zu sprechen.

Die Turtle verhält sich wie ein kleiner Roboter, den man mit Befehlen **navigieren** kann. Die wichtigsten Methoden der Klasse `Turtle` sind aus Tab. 3.2 ersichtlich. Eine vollständige Beschreibung findet man in der Klassen-Dokumentation, welche mit der Distribution ausgeliefert wird.

Methode	Abkürzung	Aktion
<code>forward (double distanz)</code>	<code>fd (double distanz)</code>	bewegt Turtle vorwärts
<code>back (double distanz)</code>	<code>bk (double distanz)</code>	bewegt Turtle rückwärts
<code>left (double winkel)</code>	<code>lt (double winkel)</code>	dreht Turtle nach links
<code>right (double winkel)</code>	<code>rt (double winkel)</code>	dreht Turtle nach rechts
<code>hideTurtle()</code>	<code>ht()</code>	versteckt Turtle (zeichnet schneller)

showTurtle()	st()	zeigt Turtle
speed(double)		setzt Turtlegeschwindigkeit
penUp ()	pu ()	hebt den Zeichenstift (Spur unsichtbar)
penDown ()	pd ()	setzt Zeichenstift ab (Spur sichtbar)
setColor (Color color)		setzt Turtlefarbe
setPenColor (Color color)		setzt Stiftfarbe
penErase()	pe()	löscht (zeichnet mit Hinter- grundfarbe)
fill()		füllt eine vorhandene ge- schlossene Figur
setFillColor(Color color)		setzt Füllfarbe
setPos(double x, double y)		setzt Turtle an die Position (x,y)
setX (double)		setzt Turtle an x-Koordinate
setY (double)		setzt Turtle an y-Koordinate
getPos()		gibt die Turtleposition zurück
getX(double)		gibt die x-Koordinate zurück
getY(double)		gibt die y-Koordinate zurück
distance(double x, double y)		gibt Entfernung von (x,y) zurück
home()		bewegt Turtle in die Aus- gangsposition
reinit()		setzt Turtle in den Anfangszu- stand
clean()		löscht alle Zeichnungen
label(String)		schreibt Text an der aktuellen Position
setFont(Font font)		setzt die Schriftart
setFontSize(int size)		setzt die Schriftgröße

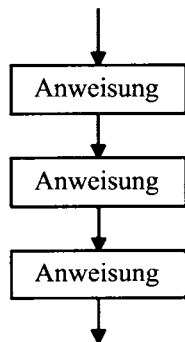
Tab. 3.2 Die wichtigsten Methoden der Klasse *Turtle*



# 4 Die grundlegenden Programmstrukturen

## 4.1 Die Sequenz

Ein Prozessor ist zu einem großen Teil damit beschäftigt, in zeitlicher Abfolge Befehl um Befehl eines Programms abzuarbeiten, wir sprechen von einem **sequentiellen** Prozess (im Gegensatz beispielsweise zu einem Lebewesen, in dem viele Prozesse miteinander (parallel) ablaufen). Darum ist in jeder Programmiersprache die Sequenz eine fundamentale Programmstruktur. Wir können diese Abfolge auch bildlich darstellen (Abb. 4.1).



*Abb. 4.1 Die Sequenz*

Bereits unser erstes Programm zur Berechnung des Preises bestand aus einer Sequenz von Anweisungen. Um nochmals zu erleben, was eine Sequenz ist, wollen wir die Turtle anweisen, eine fünfstufige Treppe zu zeichnen und programmieren dazu jeden einzelnen Schritt explizit aus.

```
// TuEx1.java  
  
import ch.aplu.turtle.*;
```

```
class TuEx1
{
    public static void main(String[] args)
    {
        Turtle john = new Turtle();    // Objekterzeugung

        john.forward(20);    // Gehe 20 Schritte vorwärts
        john.right(90);       // Drehe nach rechts
        john.forward(20);
        john.left(90);

        john.forward(20);
        john.right(90);
        john.forward(20);
        john.left(90);

        john.forward(20);
        john.right(90);
        john.forward(20);
        john.left(90);

        john.forward(20);
        john.right(90);
        john.forward(20);
        john.left(90);

        john.forward(20);
        john.right(90);
        john.forward(20);
        john.left(90);
    }
}
```

Das Programm ist zwar wenig elegant, aber wir erhalten das gewünschte Resultat (Abb. 4.2).

## 4.2 Die Iteration (Wiederholung)

Das Wiederholen von Anweisungsblöcken gehört zu den fundamentalen Aufgaben eines Computers. Man kann sogar davon sprechen, dass die Rechenmaschinen zu diesem Zweck erfunden wurden: Sie sollten dieselben Abläufe mit verschiedenen Werten immer und immer wieder abarbeiten und dazu den Menschen von mühsamer Routine entlasten. In Java werden Programmblöcke mit der linken geschweiften Klammer { eingeleitet und mit der rechten geschweiften Klammer } beendet. Die Klammern treten also immer paarweise auf. Im Innern eines Programmblocks können sich selbstverständlich weitere Programmblöcke befinden. Bei der Wiederholung wird im Allgemeinen ein gewisser Programmblock mehrmals durch-

laufen (im Spezialfall aber auch einmal oder keinmal). Damit der Durchlauf beendet wird, muss eine **Bedingung** formuliert werden. Unter einer Bedingung versteht man einen Ausdruck, der **wahr** oder **falsch** ist. In Java gibt es drei Varianten, diese Bedingung anzuordnen: Zu Beginn oder am Ende des Wiederholblocks oder als for-Schleife.

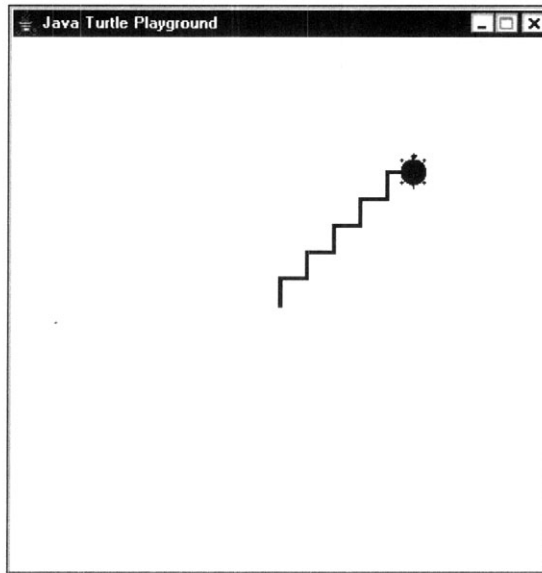


Abb. 4.2 Die Ausgabe von TuEx1

### 4.2.1 Die while-Struktur

Bei dieser Wiederholstruktur wird vor dem Eintritt in den Wiederholblock (manchmal auch **Körper** der Wiederholung genannt) eine Bedingung auf ihren Wahrheitsgehalt geprüft. Falls die Bedingung wahr ist, wird der Wiederholblock ausgeführt und nachher zur Bedingungsprüfung zurückgekehrt. Umgangssprachlich formuliert handelt es sich um eine Struktur der Art:

```
Solange ( bedingung erfüllt ) führe aus  
{  
    Anweisungsblock  
}
```

Ablaufdiagramme (Flussdiagramme) sind zwar in der Informatik verpönt, in diesem Zusammenhang liefern sie aber einen unübertrefflichen Einblick in den zeitlichen Ablauf (Abb. 4.3).

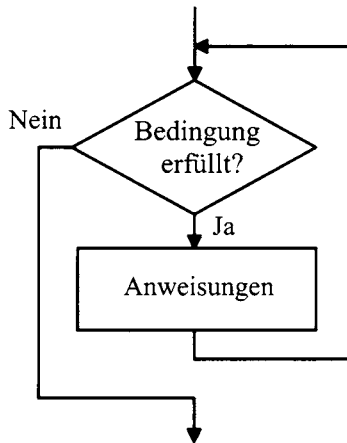


Abb. 4.3 while-Struktur

Da die Bedingung vor dem Eintritt in den Wiederblock geprüft wird, ist es auch möglich, dass der Block gar nie ausgeführt wird. Man spricht von einer **vorprüfenden** Wiederholung. Auf Grund der Rückführung des Programmablaufs sagt man statt while-Struktur auch oft **while-Schleife**.

Mit Hilfe der while-Struktur wird das Programm, welches die Turtle eine 5-stufige Treppe zeichnen lässt, wesentlich einfacher und eleganter.

```
// TuEx2.java
```

```
import ch.aplu.turtle.*;
```

```
class TuEx2
```

```
{
```

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

```
    {
```

```
        Turtle john = new Turtle();
```

```
        int i = 0;
```

```
        while (i < 5)
```

```
        {
```

```
            john.forward(20);
```

```
            john.right(90);
```

```
            john.forward(20);
```

```
            john.left(90);
```

```
            i = i + 1;
```

```
        }
```

```
    }
```

```
}
```

Wir benötigen dazu einen **Schleifenzähler** `i`, der bei jedem Durchgang durch die Schleife um 1 erhöht wird. Erreicht `i` den Wert 5, so ist die Laufbedingung `i < 5` nicht mehr wahr und die Wiederholung wird abgebrochen.

*Für die Anordnung der Blockklammern gibt es zwei übliche Konventionen, nämlich*

```
while ( bedingung )
{
    Anweisungsblock
}
oder
while ( bedingung ) {
    Anweisungsblock
}
```

*Beide haben Vor- und Nachteile: Die erste Version ergibt übersichtliche Blöcke, da beginnende und schließende Klammern immer übereinander stehen. Bei der zweiten Version gewinnt man eine Zeile (auf der nur eine Klammer steht). Obschon die zweite Version in der professionellen Java-Programmierungsgemeinde bevorzugt wird, wählen wir in diesem Buch die erste Version, welche die Ablauflogik wesentlich klarer zum Ausdruck bringt.*

Ein zweites Beispiel stammt aus der Zinseszinsrechnung und behandelt bereits ein Problem, das wir ohne Programmierung nicht ganz leicht lösen könnten. Wir wollen herausfinden, wie lange wir ein Kapital zu einem bestimmten Zinssatz anlegen müssen, bis es sich verdoppelt hat. Wir verwenden dabei die Möglichkeiten, über das Console-Fenster einen Wert einzulesen, auf einen Tastendruck zu warten und das Programm zu beenden. Da die Methoden der Klasse `Console` mehrmals verwendet werden, ist es angebracht, eine `Console`-Referenz `c` zu erstellen, die mehrmals verwendet wird.

*Statt der Zunahme eines Kapitals, könnten wir die Zunahme der Bevölkerung eines Landes bei gegebener jährlicher Zuwachsrate betrachten. Für das hier betrachtete Wachstumsverhalten gilt, dass die Zunahme pro Zeitschritt proportional zur aktuellen Größe ist, was das Verhalten von vielen Systemen richtig beschreibt und zu einem exponentiellen Wachstum führt.*

```
// Zins.java

import ch.aplu.util.*;

class Zins
{
    public static void main(String[] args)
    {
        Console c = Console.init();
        double anfangskapital = 200;
        c.print("Zinssatz? ");
        double zinssatz = c.readDouble();
        int n = 0; // Jahre
```

```
double kapital = anfangskapital;
while (kapital < 2 * anfangskapital)
{
    kapital = kapital * (1 + zinssatz/100);
    n++;
}
c.println("Laufzeit mind. " + n + " Jahre.");
c.print("Zum Beenden eine Taste druecken");
c.getKeyWait();
c.terminate();
}
```

#### Bemerkungen:

- Der Text, welcher mit `print` ausgeschrieben wird, kann auf einfache Art mit dem Pluszeichen zusammengesetzt werden. Wir nennen dies **Stringkonkatenation**. Der Text muss immer in Anführungszeichen gesetzt werden, die Werte von numerischen Variablen werden automatisch in die Stringform umgesetzt
- In der Mathematik werden algebraische Größen mit einem einzigen Buchstaben bezeichnet, eventuell mit hoch- oder tiefstehenden Zeichen oder Zahlenindizes. Dies ist deswegen nötig, weil zwei hintereinander geschriebene Buchstaben als Produkt der Größen aufgefasst werden. In Java ist das Multiplikationszeichen `*` obligatorisch, so dass Variablennamen mit mehreren Buchstaben nicht nur erlaubt sind, sondern wegen der besseren Übersicht sogar bevorzugt werden.
- Wir verwenden die Leerschläge in großzügiger Art, um den Quelltext übersichtlich zu gestalten. Insbesondere werden Leerschläge vor und nach dem Gleichheitszeichen, nach Kommas (bei Parameteraufzählungen), nach Strukturierungsschlüsselwörtern (`while` usw.) und bei mathematischen Ausdrücken gemacht. Weitere Regeln sind aus den Programmbeispielen ersichtlich.
- Blöcke werden konsequent mit 2 Leerschlägen eingerückt und Zeilen innerhalb eines Blocks immer exakt untereinander begonnen
- Wir verwenden die in Java übliche abkürzende Schreibweise `n++` für `n = n + 1`. Entsprechend kann man statt `n = n - 1` abkürzend `n--` schreiben
- Es ist ein schlimmer Anfängerfehler, grundsätzlich den Strichpunkt nach jeder Zeile zu setzen. Ein Strichpunkt am Ende der `while`-Zeile führt zwar zu einem syntaktisch korrekten Programm, da der Compiler dies als eine **leere Anweisung** betrachtet. Zur Laufzeit bleibt aber das Programm in der `while`-Schleife „hängen“, weil diese leere Anweisung als `while`-Block aufgefasst wird, in der auf die Laufbedingung kein Einfluß genommen wird.



**Über die Formatierung von Programmen kann man unterschiedlicher Meinung sein, innerhalb eines Quellprogramms hält man sich aber konsequent an die sich selbst auferlegten Konventionen.**

*Wir erkennen auch einen Nachteil der Computerlösung: Die Vermutung liegt nahe, dass die Verdopplungszeit unabhängig vom gewählten Anfangskapital ist, also nur vom Zinssatz abhängt. Wir können*

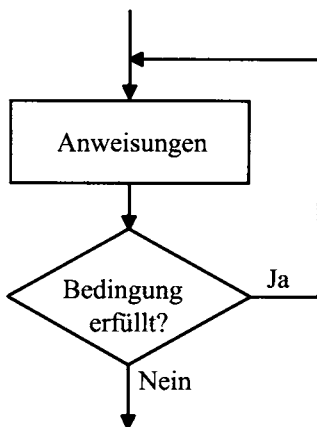
*die Vermutung zwar überprüfen, indem wir in einigen Testläufen das Anfangskapital ändern. Ganz offensichtlich ist dies aber kein Beweis für die Allgemeingültigkeit unserer Vermutung.*

### 4.2.2 Die do-while-Struktur

Im Gegensatz zur while-Struktur erfolgt hier der Test für die Laufbedingung erst am Ende des Wiederholblocks. Man spricht von einer **nachprüfenden** Wiederholung. Umgangssprachlich formuliert handelt es sich um eine Struktur der Art:

```
Führe aus  
{  
    Anweisungsblock  
} solange ( bedingung )
```

Im Flussdiagramm ist der Ablauf klar ersichtlich (Abb. 4.4).



**Abb. 4.4** do-while-Struktur

Da die Auswertung der Bedingung erst am Schluss erfolgt, wird der Wiederholblock mindestens einmal durchlaufen, was oft ein Nachteil ist. Aus diesem Grund wird die do-while-Struktur im Vergleich zur while-Struktur selten eingesetzt. Als Beispiel wählen wir wieder die 5-stufige Treppe.

```
// TuEx3.java  
  
import ch.aplu.turtle.*;  
  
class TuEx3  
{  
    public static void main(String[] args)  
    {
```

```

Turtle john = new Turtle();
int i = 0;
do
{
    john.forward(20);
    john.right(90);
    john.forward(20);
    john.left(90);
    i++;
} while (i < 5);
}

```

Man beachte, dass es sich bei der Bedingung um die Lauf- und nicht die Abbruchbedingung handelt. In vielen bekannten Programmiersprachen ist dies bei nachprüfenden Wiederholungen (repeat-until) gerade umgekehrt.

### 4.2.3 Die for-Struktur

Falls man in einer wiederholenden Struktur einen Schleifenzähler benötigt, kann die for-Struktur die while-Struktur vorteilhaft ersetzen. Dies ist bei numerisch orientierten Problemen oft der Fall, beispielsweise im Zusammenhang mit Vektoren. Das Zeichnen der Treppe mit 5 Stufen gehört auch zu Problemen, bei denen die for-Struktur angebracht ist. TuEx4 ist also die beste Version des Treppenprogramms.

```

// TuEx4.java

import ch.aplu.turtle.*;

class TuEx4
{
    public static void main(String[] args)
    {
        Turtle john = new Turtle();

        for (int i = 0; i < 5; i++)
        {
            john.forward(20);
            john.right(90);
            john.forward(20);
            john.left(90);
        }
    }
}

```



Für das anschauliche Verständnis der for-Struktur ist wieder ein Flussdiagramm geeignet (Abb. 4.5). Grau hinterlegt sind die Teile, welche in der for-Klammer angegeben werden.

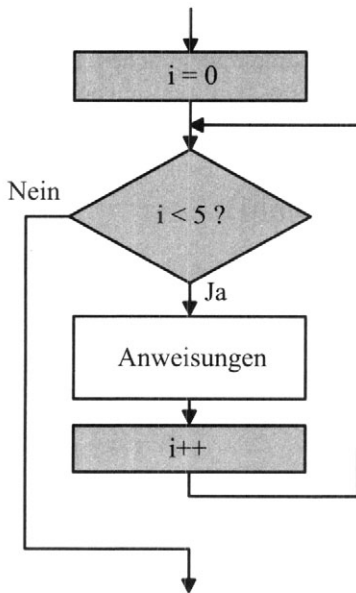


Abb. 4.5 for-Struktur

Man erkennt, dass es sich um eine spezielle, automatisierte Form einer while-Struktur handelt. Die Syntax ist sehr gewöhnungsbedürftig und kann leicht zu schwer auffindbaren Fehlern führen. Folgendes ist zu beachten:

- Der Schleifenzähler kann auch vor dem Eintritt in die for-Schleife deklariert werden und wird in der for-Klammer nur noch initialisiert. Er ist dann aber auch nach dem Ende der for-Schleife noch „sichtbar“. Es gehört aber zum guten Programmierstil, den Schleifenzähler in der for-Schleife zu „kapseln“, indem man ihn in und nicht außerhalb der for-Klammer deklariert
- Der Schleifenzähler wird im vorliegenden Fall bei jedem Durchlauf um 1 erhöht. Auch wenn es theoretisch möglich wäre, wird er nicht zusätzlich verändert, weil man die Auswirkungen schlecht durchblickt
- Man beachte, dass wie bei der while-Struktur ein unachtsam gesetzter Strichpunkt am Ende der for-Bedingung zu einer leeren Anweisung führt, die als Wiederholblock aufgefasst wird. Der eigentlich vorgesehene Wiederholblock befindet sich dann trotz der Einrückung außerhalb der for-Schleife, was syntaktisch richtig ist, aber zu einem bössartigen Laufzeitfehler führt.

*Wer häufig in anderen Programmiersprachen programmiert, muss sich daran gewöhnen, dass es sich im Gegensatz zu vielen bekannten Programmiersprachen um die Laufbedingung und nicht um die Endbedingung (for-to-Struktur) handelt.*

## 4.3 Die Selektion (Auswahl)

Programmverzweigungen auf Grund von bestimmten Bedingungen gehören zum grundlegenden Befehlssatz aller Prozessoren. Auf höhere Programmiersprachen übertragen heißt dies, dass ein logischer (boolescher) Ausdruck ausgewertet wird und je nachdem, ob dieser wahr oder falsch ist, ein Programmblock ausgeführt wird oder nicht.

### 4.3.1 Die if-Stuktur (einseitige Auswahl)

Am Flussdiagramm lässt sich der Ablauf wiederum am besten überblicken (Abb. 4.6).

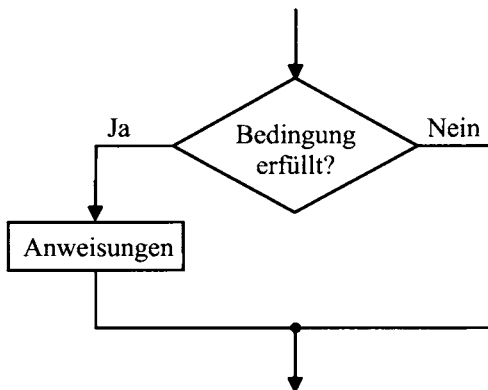


Abb. 4.6 if-Struktur

Falls die Bedingung wahr ist, verzweigt das Programm in den if-Block. Nachher wird es an derselben Stelle zusammengeführt, wie wenn die Bedingung nicht erfüllt ist. Mit anderen Worten, auf Grund der Bedingung wird ein zusätzlicher Programmblock ausgeführt oder nicht.

Umgangssprachlich handelt es sich um eine Struktur der Art

```

Falls ( bedingung ) dann
{
    Anweisungsblock
}
  
```

Als Beispiel lösen wir eine Aufgabe aus dem Gebiet der Spiele. Wir werfen 1000 mal zwei Würfel (Doppelwurf) und betrachten jeweils die Augensumme. Dabei zählen wir, wie oft die Augensumme 8 beträgt. Es ist zu erwarten, dass diese Zahl von Versuchreihe zu Versuchreihe etwas unterschiedlich ausfällt. Interessant ist es, eine Vermutung über den Mittelwert aufzustellen. Für „Fließaufgaben“ dieser Art ist der Computer hervorragend geeignet, denn er kann das Würfeln Millionen Mal schneller als der Mensch durchführen.

Im Programm lenken wir mit `Console.init()` zuerst alle Ausgaben in ein Console-Fenster um. Anschließend deklarieren wir einige Variablen, die wir im Programm benötigen. In der Simulation bedeutet das Werfen der beiden Würfel, dass der Computer zwei Zufallszahlen zwischen 1 und 6 generieren muss. Dazu verwenden wir die Java-Klasse `Random` und fordern mit der Methode `nextInt(6)` die nächste Zufallszahl im Bereich 0 bis 5 an. Wir verwenden auch neu den Operator `==`, der zwei Größen auf Gleichheit überprüft.

*In Java ist es zwingend nötig, die Zuweisung (mit dem Operator `=`) vom Vergleich (mit dem Operator `==`) zu unterscheiden. Es gehört zu den bekanntesten Programmierfehlern, die Verdoppelung des Gleichheitszeichens beim Vergleich zu vergessen. In anderen Programmiersprachen wird eine etwas vernünftigere Schreibweise verwendet.*

```
// Wuerfeln.java

import java.util.*;
import ch.aplu.util.*;

class Wuerfeln
{
    public static void main(String[] args)
    {
        Console.init();
        int summe = 8;
        int wuerfe = 1000;

        int m, n;
        int bingo = 0;

        Random rnd = new Random();

        for (int i = 0; i < wuerfe; i++)
        {
            m = rnd.nextInt(6); // Zufallszahl 0 <= m < 6
            n = rnd.nextInt(6);
            m++; // Augenzahl 1. Wuerfel
            n++; // Augenzahl 2. Wuerfel
            if (n + m == summe)
            {
                bingo++;
            }
        }
        System.out.print("Auf " + wuerfe + " Wuerfe hatte ich "
                        + bingo + "x die Summe " + summe);
    }
}
```

Der `if`-Block besteht aus einer einzigen Anweisung, die den Wert von `bingo` um eins erhöht. Falls, wie hier, der Block nur aus einer einzigen Anweisung besteht, können die Block-

klammern auch weggelassen werden. Man läuft aber dann in Gefahr, dass beim Hinzufügen einer weiteren Anweisung im if-Block die Klammern vergessen werden, was zu einem schwer auffindbaren Laufzeitfehler führt, da das Programm syntaktisch richtig bleibt.

Das Programm zeigt auch, dass es zum guten Programmierstil gehört, für numerische Werte, wie die Totalzahl der Würfe, eine Variable am Anfang des Programms zu deklarieren, statt sie irgendwo im Innern des Programms „hart zu verdrahten“. Dies gilt insbesondere dann, wenn man diese Zahl an mehreren Stellen verwendet. (Solche Zahlen werden ironisch auch **magic numbers** genannt.) Es ist nämlich unwahrscheinlich, dass man bei einer nachträglichen Modifikation des Werts alle Stellen auffindet, an denen dieser verändert werden muss. Meist wird man sich zufrieden geben, die erste gefundene Stelle zu modifizieren, was zu bösen Laufzeitfehlern führen kann.

### 4.3.2 Die if-else-Struktur (zweiseitige Auswahl)

Auch hier verschaffen wir uns einen Überblick mit einem Flussdiagramm (Abb. 4.7).

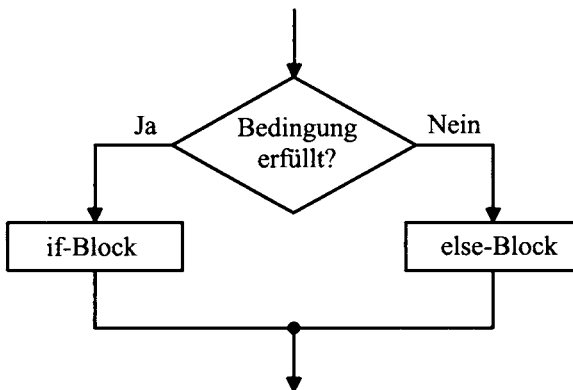


Abb. 4.7 if-else-Struktur

Umgangssprachlich können wir schreiben

```

Falls ( bedingung ) dann
{
    if-Anweisungsblock
}
sonst
{
    else-Anweisungsblock
}
  
```

Im folgenden Programm zeichnet die Turtle einen Halbkreis (eigentlich ein fünfzigseitiges Polygonsegment) im Gegenuhrzeigersinn und dann einen Halbkreis im Uhrzeigersinn.

```
// TuEx5.java

import ch.aplu.turtle.*;

class TuEx5
{
    public static void main(String[] args)
    {
        Turtle john = new Turtle();

        for (int i = 0; i < 100; i++)
        {
            john.forward(2);
            if (i < 50)
                john.left(3.6); // Drehe 3.6 Grad
            else
                john.right(3.6);
        }
    }
}
```

Bei der Verschachtelung von if-else-Strukturen geschehen leicht Überlegungsfehler. Aus diesem Grund sind tiefe Verschachtelungen, d.h. weitere if-else-Strukturen innerhalb eines if- oder else-Blocks zu vermeiden. Falls man, wie im obigen Beispiel, die Blockklammern weglässt, ist es besonders gefährlich, einen weiteren else-Block einzufügen. Will man beispielsweise die Turtle-Farbe lediglich beim ersten Halbkreis verändern, sobald die Turtle eine Höhe von 25 überschreitet, so setzt man aus Fahrlässigkeit

```
// TuEx5a.java

import ch.aplu.turtle.*;
import java.awt.Color;

class TuEx5a
{
    public static void main(String[] args)
    {
        Turtle john = new Turtle();

        for (int i = 0; i < 100; i++)
        {
            john.forward(2);
            if (i < 50)
                john.left(3.6); // Drehe 3.6 Grad
                if (john.getY() > 25)
                    john.setColor(Color.red);
            else
```

```

        john.right(3.6);
    }
}

```

Das Programm ergibt keinen Syntaxfehler. Aber das `else` gehört trotz des Einrückens zum zweiten `if`, was aber nicht die Absicht war (**tangling-else problem**). Abhilfe schafft das korrekte Setzen von Blockklammern.

// TuEx5b.java

```

import ch.aplu.turtle.*;
import java.awt.Color;

class TuEx5b
{
    public static void main(String[] args)
    {
        Turtle john = new Turtle();

        for (int i = 0; i < 100; i++)
        {
            john.forward(2);
            if (i < 50)
            {
                john.left(3.6); // Drehe 3.6 Grad
                if (john.getY() > 25)
                    john.setColor(Color.red);
            }
            else
                john.right(3.6);
        }
    }
}

```

Die Gefahr besteht nicht, wenn man immer, auch bei `if`- und `else`-Blöcken mit nur einer Anweisung, Blockklammern setzt.

### 4.3.3 Die switch-Struktur (Mehrfachauswahl)

Falls eine Variable mehrere verschiedene Werte annehmen kann und je nach Wert ein anderer Programmblock ausgeführt werden soll, bietet sich die `switch`-Struktur anstelle einer geschachtelten `if-else`-Struktur an. Die Logik der `switch`-Struktur stammt aus den Zeiten der Assembler-Programmierung und mutet deswegen etwas archaisch an.

Zuerst wird der Ausdruck der switch-Anweisung ausgewertet und dann der Reihe nach mit fest definierten (konstanten) Werten (auch Labels genannt) verglichen. Ergibt sich eine Übereinstimmung, so springt das Programm zum entsprechenden case-Block. Nachher läuft das Programm allerdings beim darunter stehenden case-Block weiter. Meist ist dies unerwünscht und man zwingt das Programm mit der break-Anweisung, die switch-Struktur zu verlassen.

Auch hier zeigt das Flussdiagramm (Abb. 4.8) das Verhalten anschaulich. Der Defaultblock wird ausgeführt, falls der Ausdruck a zu keiner Übereinstimmung mit den angegebenen Labels führt. Dieser Block kann auch weggelassen werden und das Programm springt bei fehlender Übereinstimmung zur nächsten Anweisung. Falls das break in einem case-Block fehlt, so spricht man vom **Durchfallen** in den nächsten Block. Erst beim nächsten break wird die switch-Struktur verlassen. In seltenen Fällen ist dies erwünscht und sollte besonders dokumentiert werden, oft wird aber das break vergessen, was zu einem schwerwiegenden Laufzeitfehler führt.

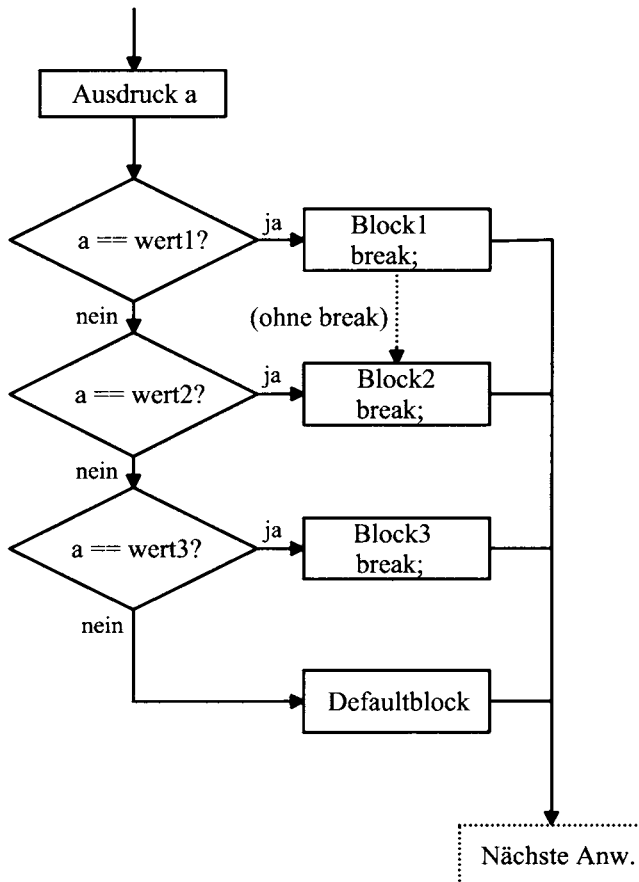


Abb. 4.8 switch-Struktur

Symbolisch für die Lernkurve beim Erlernen von Java soll die Turtle eine Treppe zeichnen, deren erster Tritt eine Stufenhöhe von 80 und deren zweiter Tritt eine Stufenhöhe von 40 aufweist. Nach der hohen Einstiegsschwelle soll die Stufenhöhe nur noch 20 betragen.

```
// TuEx6.java

import ch.aplu.turtle.*;

class TuEx6
{
    public static void main( String[] args )
    {
        Turtle john = new Turtle();
        for ( int i = 0; i < 5; i++ )
        {
            switch ( i )
            {
                case 0:
                    john.forward( 80 );
                    break;
                case 1:
                    john.forward( 40 );
                    break;
                default:
                    john.forward( 20 );
                    break;
            }
            john.right( 90 );
            john.forward( 20 );
            john.left( 90 );
        }
    }
}
```

Das `break` im Default-Block ist nicht nötig, wird aber zur Sicherheit hingesetzt, damit die Reihenfolge der `case/default` ohne Einfluss auf die Programmlogik verändert werden kann.

In `switch()` sind leider nur Variablen vom Typ `int`, `short`, `byte` oder `char` zulässig.

## 4.4 Varianten der for-Struktur

Die `for`-Struktur gehört zu den wichtigen Programmstrukturen von Java, insbesondere auch, weil sie sehr verschieden aufgebaut werden kann. Obschon es sich um eine lustige intellektuelle Herausforderung handelt, geistreiche `for`-Strukturen zu erfinden, zählt man exotisch aufgebaute `for`-Schleifen zum trickreichen Programmieren. War es in den Siebziger- und



Achtzigerjahren des letzten Jahrhunderts unter Studierenden „in“, die kürzeste Variante eines Algorithmus durch Anwendung von Programmiertricks zu präsentieren, sind vielfach aus den Tricks Stolpersteine geworden, die man besser meidet.

**Die Verwendung von Programmiertricks ist nicht Zeichen für fortgeschrittene professionelle Programmierung, sondern führt zu schwierig les- und wartbaren Programmen und macht deshalb nur in gut dokumentierten Ausnahmefällen Sinn.**

Da wir aber auch in der Lage sein müssen, Programme, die von anderen Programmierern geschrieben wurden, zu lesen und zu verstehen, besprechen wir hier die bekanntesten Varianten der for-Struktur. Sie gehen davon aus, dass sie die allgemeine Form

```
for (Startweisung; Laufbedingung; Schleifenanweisung)
{
    Schleifenkörper
}
```

besitzt, die als Ablaufschema in Abb. 4.9 dargestellt ist.

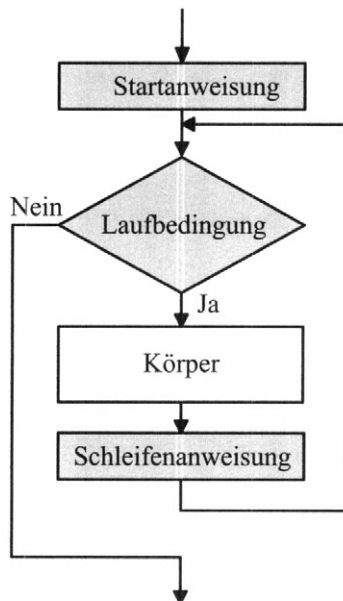


Abb. 4.9 Allgemeine for-Struktur

### 4.4.1 Mehrere Schleifenvariable

Schreiben wir mehrere Anweisungen, nur durch den **Kommaoperator** getrennt, hintereinander, so wird dies als eine einzige Anweisung betrachtet, die von links nach rechts abgearbeitet wird. In der for-Klammer muss der erste und letzte Term eine einzige Anweisung sein. Setzen wir hier den Kommaoperator ein, so können wir bereits in der for-Klammer kompliziertere Operationen ausführen lassen. Typisch ist die Verwendung von zwei Schleifenvariablen, beispielsweise ergibt Programm ForEx1 folgendes Resultat:

```
i: 0 k: 10  
i: 1 k: 9  
i: 2 k: 8  
i: 3 k: 7  
i: 4 k: 6
```

```
// ForEx1.java
```

```
import ch.aplu.util.*;
```

```
class ForEx1
```

```
{  
    public static void main(String[] args)  
    {  
        int i, k;  
        for (i = 0, k = 10; i < k; i++, k--)  
        {  
            Console.println("i: " + i + " k: " + k);  
        }  
    }  
}
```

### 4.4.2 Weglassen einzelner Teile

Es ist erlaubt, in der for-Klammer einzelne Teile wegzulassen oder durch eine Leeraanweisung zu ersetzen. Beispielsweise kann man die Initialisierung der Schleifenvariablen vor Eintritt in die Klammer erledigen:

```
int i = 0;  
for (; i < 5; i++)
```

oder die Schleifenanweisung in den Körper verlegen

```
for (i = 0; i < 5)  
{  
    ...  
    i++;  
}
```

### 4.4.3 Endlosschleife

Auf den ersten Blick mag eine Endlosschleife als unsinnig betrachtet werden, da man damit ein Programm erhält, das nie abbricht. Es gibt aber durchaus Möglichkeiten, eine Endlosschleife abubrechen, beispielsweise durch eine `break`-Anweisung oder in ereignisgesteuerten Programmen durch einen Event. Im folgenden Programm versucht die Turtle verzweifelt nach Hause zu kommen, wobei sie sich nach jedem Schritt in einer zufälligen Richtung weiterbewegt. Das folgende Programm bricht ab, sobald die Turtle die Ziellinie  $y = 100$  überschreitet. Sie gibt die Suche nach 20 Schritten auf. Es wird die Methode `Math.random()` verwendet, die bei jedem Aufruf eine Zufallszahl zwischen 0 und 1 liefert.

```
// ForEx2.java

import ch.aplu.turtle.*;

class ForEx2
{
    public static void main(String[] args)
    {
        Turtle john = new Turtle();

        int n = 0;
        for (;;)
        {
            john.forward(10);
            john.right(180 * (Math.random()-0.5));
            if (john.getY() > 100)
            {
                john.label("    Got home");
                break;
            }
            if (n++ == 20)
            {
                john.label("    Got lost");
                break;
            }
        }
    }
}
```

### 4.4.4 for-Schleife ohne Körper

Da die Schleifenanweisung am Ende des Wiederholblocks ausgeführt wird, kann man sie dazu benutzen, Anweisungen auszuführen, die sich normalerweise im Schleifenkörper befinden. Dies macht natürlich, wenn überhaupt, nur bei sehr einfachen Anweisungen einen Sinn. Will man beispielsweise die Summe der Quadratzahlen von 1 bis 4 berechnen, so schreibt man klassisch

```
int sum = 0;
for (int i = 1; i <= 4; i++)
{
    sum = sum + i*i;
}
```

Zieht man alle Trickregister, so könnte man das Programm wesentlich kürzer, aber damit wesentlich unleserlicher schreiben. Das folgende Beispiel ist deshalb lustig, aber nicht nachahmenswert.

```
// ForEx3.java

import ch.aplu.util.*;

class ForEx3
{
    public static void main(String[] args)
    {
        int sum = 0;
        for (int i = 1; i <= 4; sum += i*i++);
        Console.println("Summe der Quadratzahlen 1..4: "
                        + sum);
    }
}
```

# 5 Elementare Klassenkonstruktionen

## 5.1 Objektorientiertes Konstruieren

Obschon wir bereits eine Applikationsklasse deklariert und Objekte verwendet haben, waren unsere Beispiele vom Standpunkt der objektorientierten Programmierung noch mager. Java-Programme in diesem Programmierstil unterscheiden sich nur unwesentlich von Programmen in klassischen prozeduralen Programmiersprachen. Dabei verzichtet man bewusst auf einen wesentlichen Bestandteil der modernen Softwaretechnologie, was sich spätestens bei größeren Projekten rächen wird. Zudem ist uns der Zugang zu modernen Klassenbibliotheken verwehrt, da deren Verwendung gute Kenntnisse der OOP voraussetzen. Aus diesen Gründen versuchen wir, uns bereits jetzt in die objektorientierte Methodik einzuarbeiten, obschon wir uns dabei besonders anstrengen müssen.

Um den fundamentalen Unterschied zwischen klassischer Programmierung und OOP zu illustrieren, greifen wir das Problem der Zinsrechnung aus Kap. 4.2 noch einmal auf und lösen es objektorientiert. Dabei ist bereits der Einstieg völlig unterschiedlich. Statt uns nämlich zu Beginn auf den Algorithmus zu konzentrieren, **modellieren** oder **abstrahieren** wir zuerst die Wirklichkeit auf der Suche nach Objekten mit Eigenschaften und Verhalten. Zwar tönt dies sehr anspruchsvoll, es ist aber nahe liegend, dass wir ein Bankkonto als ein Objekt auffassen. Eigenschaften sind *Kapitalstand*, *Anlagejahre* und *Zinssatz* und die Verhalten geben uns die Möglichkeit, diese Eigenschaften „abzufragen“ sowie das Geld ein weiteres Jahr zinsbringend anzulegen.

Mit einer speziellen Methode, **Konstruktor** genannt, setzen wir die Eigenschaften, d.h. die Instanzvariablen, beim Erzeugen des Objekts auf gewünschte Anfangswerte. Um den Konstruktor von den anderen Methoden zu unterscheiden, müssen wir ihm den Klassennamen (unter Beachtung der Groß-Kleinschreibung) geben. Der Konstruktor unterscheidet sich von anderen Methoden auch noch dadurch, dass er keinen Rückgabetyt besitzt. Er wird nämlich nie wie eine gewöhnliche Methode aufgerufen, sondern beim Erzeugen des Objekts mit dem Operator `new` automatisch ausgeführt.

Wir packen das Problem am besten so an, dass wir zuerst in einem **Programmskelett** nur die **Instanzvariablen** und die **Methodenköpfe** mit leeren **Methodenkörpern** hinschreiben. Sowohl für die Instanzvariablen wie für die Methoden verwenden wir möglichst aussage-

kräftige Bezeichner. Wir gewöhnen uns bereits jetzt daran, Variablen, deren Anfangswert bei ihrer Deklaration bekannt ist, gerade dort zu initialisieren.

```
// Bankkonto.java
```

```
class Bankkonto
{
    double kapital;
    double zinssatz;
    int anlagejahre = 0;

    Bankkonto(double kap, double zs)
    // Konstruktor
    {
    }

    double kontostand()
    // Gibt aktuellen Kontostand zurück
    {
    }

    int laufzeit()
    // Gibt die Anlagezeit in Jahren zurück
    {
    }

    void jahresabrechnung()
    // Führt die Jahresabrechnung durch
    // Setzt neues Kapital, neue Laufzeit
    {
    }
}
```

Die Methoden `kontostand()` und `laufzeit()` geben einen Wert zurück. Der Datentyp des Rückgabewerts ist gleichzeitig der Datentyp der Methode und wird bei der Deklaration angegeben. Für die Rückgabe verwenden wir das Schlüsselwort `return`, welches bewirkt, dass der hinter `return` stehende Ausdruck ausgewertet, dieser Wert an den Aufrufenden zurückgeliefert und die weitere Verarbeitung der Methode abgebrochen wird. Grundsätzlich braucht `return` nicht die letzte Anweisung der Methode zu sein und eine Methode kann sogar mehrere `return` enthalten. Um ein übersichtlich strukturiertes Programm zu erhalten, werden wir aber nur in Ausnahmen von der Regel abweichen, ein einziges `return` als letzte Anweisung einer Methode zu verwenden.


Die Methode `jahresrechnung()` ermittelt das neue Kapital nach einem weiteren Jahr Laufzeit und verändert die Instanzvariablen dementsprechend. Da sie keinen Wert zurückgibt, setzt man als Rückgabotyp `void` (leer). (Dies ist eigentlich überflüssig und nur als Tribut an die Herkunft von Java zu verstehen.) Wir können darauf verzichten, am Ende der Methode ein `return` zu schreiben, da die Methode sowieso an dieser Stelle zurückkehrt.

Der Konstruktor `Bankkonto()` besitzt eine Parameterliste mit den zwei Parametern `kap` und `zs`. Man nennt diese auch **formale Parameter**, da sie bei der Deklaration lediglich als **Platzhalter** dienen und ihre Bezeichnung (aber nicht ihre Reihenfolge und ihr Datentyp) willkürlich ist.

Das Skelett legt die **Schnittstelle nach außen** fest oder anders gesagt, wie die Klasse *von außen gebraucht* werden kann. Wir nennen ein solches Skelett auch das **Klasseninterface** (in Java wird das Schlüsselwort `interface` noch in einem engeren Sinn verwendet, den wir erst später kennen lernen).

Im Weiteren können zwei Wege beschritten werden. Stellen wir uns für den Moment vor, dass jemand die Klasse `Bankkonto` bereits ausprogrammiert (**implementiert**) hat. Auf Grund der festgelegten Methodenköpfe (des Klasseninterfaces) sind wir in der Lage, die Klasse für eine Applikation einzusetzen. Man nennt dieses Vorgehen **Top-Down-Design**, da man sozusagen vom Allgemeinen (von oben) zum Speziellen (nach unten) vordringt. Diese Arbeitsweise wird auch **schrittweise Verfeinerung** (**stepwise refinement**) genannt. Bevor wir allerdings die Applikation verwenden können, müssen wir natürlich `Bankkonto` nun doch implementieren.

Beim umgekehrten Weg, genannt **Bottom-Up-Design**, wird zuerst die Klasse `Bankkonto` implementiert (und ausgetestet) und erst nachher eine Applikation geschrieben, in der man sie verwendet. Beide Vorgehensweisen haben Vor- und Nachteile und werden in der Praxis nebeneinander verwendet. Wir wollen hier aber bereits ein wichtiges Gütekriterium festhalten:

 **In einem guten Klassenentwurf genügt es, das Klasseninterface zu kennen, um die Klasse zu verwenden. Für die Verwendung der Klasse sollten die Einzelheiten der Implementierung keine Rolle spielen.**

Wir entscheiden uns hier für das Bottom-Up-Design und implementieren zuerst eine Klasse `Bankkonto` in der Quelldatei `Bankkonto.java`.

```
// Bankkonto.java
```

```
class Bankkonto
{
    double kapital;
    double zinssatz;
    int anlagejahre = 0;

    Bankkonto(double kap, double zs)
    // Konstruktor
    {
        kapital = kap;
        zinssatz = zs;
    }
}
```

```


double kontostand()
// Gibt aktuellen Kontostand zurück
{
    return kapital;
}

int laufzeit()
// Gibt die Anlagezeit in Jahren zurück
{
    return anlagejahre;
}

void jahresabrechnung()
// Führt die Jahresabrechnung durch
// Setzt neues Kapital, neue Laufzeit
{
    kapital = kapital * (1 + zinssatz / 100);
    anlagejahre++;
}
}

```

Diese Klasse Bankkonto ist bereits compilierbar, kann aber natürlich nicht ausgeführt werden, da die Methode `main()` fehlt. Wir könnten `main()` zwar in der Klasse Bankkonto hinzufügen, entscheiden uns aber dafür, eine Applikationsklasse in einer zweiten Datei `BankEx1.java` zu schreiben, welche eine Instanz der Klasse Bankkonto erzeugt und diese sinnvoll verwendet. Wir halten uns dabei auch bereits an folgende Regel professioneller Java-Projekte:

 **Jede Klassendeklaration gehört in eine eigene Quelldatei, wobei Dateiname (ohne Dateierweiterung) und Klassenname übereinstimmen müssen.**

In der Applikationsklasse wollen wir die interessante Aufgabe lösen, wie lange das Kapital von 200 bei einem Zinssatz von 2.5% anzulegen ist, bis es sich durch die Verzinsung verdoppelt. Es sind aber auch viele andere Applikationen der Klasse Bankkonto denkbar und deshalb ist die Trennung in zwei Klassen mit zwei Dateien überaus sinnvoll. In `BankEx1` erzeugen wir mit `new` eine Instanz des Bankkontos, wobei wir dem Konstruktor die **aktuellen Parameterwerte** des Kapitals und des Zinssatzes übergeben. Nachher lassen wir die Bank in einer Wiederholschleife Jahr um Jahr wirtschaften, solange der Kontostand kleiner als 400 ist.

```

// BankEx1.java

import ch.aplu.util.*;

class BankEx1
{

```



```
public static void main(String[] args)
{
    Console.init();
    Bankkonto konto = new Bankkonto(200, 2.5);

    while (konto.kontostand() < 400)
        konto.jahresabrechnung();

    System.out.println("Laufzeit mindestens "
        + konto.laufzeit() + " Jahre");
}
```

Wir können nun `BankEx1` compilieren und ausführen, wobei zur Laufzeit die vorher compilierte Klasse `Bankkonto` automatisch verwendet (geladen) wird.

#### Bemerkungen:

- Vergleichen wir unseren Lösungsansatz mit dem Programm `zins.java`, so erkennen wir fundamentale Unterschiede zwischen objektorientiertem und prozeduralem Programmieren
- Der Konstruktor versetzt das `Bankkonto` durch die Initialisierung der Instanzvariablen in einen definierten Anfangszustand. Die explizite Initialisierung der Instanzvariablen auf 0 könnte man weglassen, da Instanzvariablen automatisch auf 0 initialisiert werden. Wir wollen aber mit der expliziten Angabe des Initialisierungswerts darauf hinweisen, dass wir an die richtige Initialisierung gedacht haben
- Es gibt einen wesentlichen Unterschied zwischen Methoden, die auf das `Bankkonto` nur lesend zugreifen (**Akzessor**, **accessors**, **getter-Methoden**) und solchen, die den Zustand des Kontos verändern (**Mutator**, **mutator**, **setter-Methoden**). Die Verwendung von Akzessoren ist völlig unkritisch, die Verwendung von Modifikatoren aber prinzipiell gefährlich. Sie führen zu Veränderungen der Eigenschaften, auch **Seiteneffekte** genannt, und beeinflussen dadurch die Objekte grundlegend. In unserem Beispiel verzichten wird daher vollständig auf Mutatoren
- Man muss sich gut überlegen, ob das Programm die Laufzeit nicht um 1 Jahr falsch berechnet. Man spricht vom **Plus-Minus-Eins-Syndrom** (**off-by-one error**)
- Man sollte dokumentieren, für welche Anfangswerte das Programm richtig läuft. Beispielsweise verhält es sich für den Zinssatz 0 katastrophal. Man spricht von der Angabe der **Vorbedingungen** (**Preconditions**).

Der Vorteil des objektorientierten Vorgehens wird dann offensichtlich, wenn wir mehr als ein `Bankkonto` betrachten. Wollen wir mal eine Stadtbank, die einen Zinssatz von 2.5 % anbietet, mit einer Dorfbank mit dem höheren Zinssatz von 3.5% in Konkurrenz treten lassen und uns fragen, wie lange es dauert, bis wir bei der Dorfbank trotz einer kleineren Einlage einen höheren Kontostand erreicht haben, so können wir die Klasse `Bankkonto` unverändert lassen und in einer neuen Applikationsklasse `BankEx2.java` einfach zwei Instanzen

davon erzeugen. Wir werden dabei einen großen Teil des Codes **wiederverwenden**, wodurch wir das neue Problem wesentlich schneller und ohne neue Fehlerquellen lösen können.

```
// BankEx2.java

import ch.aplu.util.*;

class BankEx2
{
    public static void main(String[] args)
    {
        Console c = new Console();
        Bankkonto stadtbank = new Bankkonto(200, 2.5);
        Bankkonto dorfbank = new Bankkonto(180, 3.5);

        while (dorfbank.kontostand() < stadtbank.kontostand())
        {
            dorfbank.jahresabrechnung();
            stadtbank.jahresabrechnung();
        }

        c.println("Laufzeit: " + stadtbank.laufzeit()
                + " Jahre");
        c.println("Kontostand bei der Stadtbank: "
                + stadtbank.kontostand());
        c.println("Kontostand bei Dorfbank: "
                + dorfbank.kontostand());
    }
}
```

Wie wir sehen, wird das mühsame Wiederholen von `System.out` vermieden, indem wir mit `new Console()` eine Instanz `c` der `Console` erzeugen und diese für den Aufruf von `print()` bzw. `println()` verwenden.

## 5.2 Methoden in der Applikationsklasse

Bereits jetzt erkennen wir, dass es beim Programmieren sehr viele Freiheiten gibt, und es dem Einfallsreichtum des Programmierers überlassen bleibt, wie ein Problem gelöst wird. Meist wird erst eine langjährige Erfahrung zeigen, warum ein gewisses Konzept einem anderen überlegen ist, aber Programme werden glücklicherweise, ein wenig wie Kunstwerke, immer eine persönliche Note tragen, auf die wir hoffentlich stolz sein können. Arbeitet man allerdings in einem Team, so sind der Phantasie enge Grenzen gesetzt, denn die Konzepte müssen von allen Partnern getragen werden, damit Programmteile ausgetauscht und von mehreren Partnern gewartet werden können.

Haben wir die Wahl, so werden wir in jedem Fall immer einen objektorientierten Ansatz vorziehen. Wir zeigen dies an folgendem Beispiel, das die Fakultät  $n!$  einer natürlichen Zahl berechnet. Diese ist bekanntlich wie folgt definiert:

$$f(n) = n! = 1 * 2 * \dots * n$$

Die Zahl  $n$  soll der Benutzer mit einem Eingabedialog wählen können. Wir verwenden dazu die Klasse `InputDialog` aus dem Package `ch.aplu.util`. In unserem ersten Ansatz, der bereits zum Erfolg führt, wird der Algorithmus in der Methode `main()` ausgeführt.

```
// FacEx1.java

import ch.aplu.util.*;

class FacEx1
{
    public static void main(String[] args)
    {
        InputDialog id =
            new InputDialog("Fakultät",
                           "Gib eine natürliche Zahl ein");
        int n = id.readInt();
        int fak = 1;
        for (int i = 1; i <= n; i++)
            fak = fak * i;
        Console.println( n + "! = " + fak);
    }
}
```

Das Programm funktioniert zwar, verstößt aber gegen die Regel der strukturierten Programmierung, dass in sich abgeschlossene Teilaufgaben in eigenen Methode zu kapseln sind. Darum verbessern wir das Programm und deklarieren die Methode `f()`, welche die Zahl  $n$  als Parameterwert übernimmt, die Fakultät berechnet und den berechneten Wert zurückgibt.

```
// FacEx2.java

import ch.aplu.util.*;

class FacEx2
{
    int f(int n)
    {
        int fak = 1;
        for (int i = 1; i <= n; i++)
            fak = fak * i;
        return fak;
    }
}
```

```

public static void main(String[] args)
{
    InputDialog id =
        new InputDialog("Fakultät",
            "Gib eine natürliche Zahl ein");
    int n = id.readInt();
    Console.println(n + "! = " + f(n));
}
}

```

Das Programm sieht nun wesentlich besser aus. Aber leider kompiliert es mit einer Fehlermeldung der Art: *Nicht-statische Methode f(int) kann nicht aus einem statischen Kontext heraus referenziert werden.* Dieser Fehler ist zwar auf den ersten Blick unerwartet, denkt man aber ein bisschen darüber nach, so leuchtet ein, warum der Compiler „ausruft“. Die neu hinzugefügte Methode `f()` gehört zu jeder einzelnen Instanz der Klasse `FacEx2`. Da es im Programm gar keine Instanz dieser Klasse gibt, kann die Methode `f()` auch nicht verwendet werden.

Aus diesem Dilemma gibt es zwei Auswege: Man deklariert die Methode `f()` ebenfalls `static`. Damit ist `f()` nur einmal in der Klasse vorhanden und kann somit problemlos in `main()` aufgerufen werden. Die Verwendung von statischen Methoden wird aber in der OOP als schwerer Rückschritt hin zur prozeduralen Programmierung aufgefasst.

Da wir den Prinzipien der OOP treu bleiben wollen, wählen wir in diesem Buch einen anderen Weg, der auf den ersten Blick etwas komplizierter (oder sogar trickreich) erscheint. Dazu erzeugen wir in `main()` eine Instanz der Applikationsklasse selbst (also der Klasse, in der sich `main()` befindet) und verlagern den Code in den Konstruktor. Da der Konstruktor im Gegensatz zu `main()` keine statische Methode ist, können alle Methoden der Klasse (also auch nicht statische) problemlos verwendet werden.

```

// FacEx3.java

import ch.aplu.util.*;

class FacEx3
{
    FacEx3()    // Konstruktor
    {
        InputDialog id =
            new InputDialog("Fakultät",
                "Gib eine natürliche Zahl ein");
        int n = id.readInt();
        Console.println(n + "! = " + f(n));
    }

    int f(int n)
    {

```

```

    int fak = 1;
    for (int i = 1; i <= n; i++)
        fak = fak * i;
    return fak;
}

public static void main(String[] args)
{
    new FacEx3();
}
}

```

Es ist völlig unnötig, in `main()` mit

```
FacEx3 f = new FacEx3();
```

eine Variable `f` einzuführen, da diese nirgends gebraucht wird.

## 5.3 Instanzvariablen und lokale Variablen, Sichtbarkeit, Geltungsbereich

Wir wollen die bisher erarbeiteten Konzepte an einigen Beispielen vertiefen. Dazu verwenden wir die Klasse `GPanel` aus dem Package `ch.aplu.util`, welche ein Grafikfenster mit frei wählbaren `double`-Koordinaten (**Windowkoordinaten** genannt) zur Verfügung stellt. Die Fensterdimensionen können durch Ziehen mit der Maus verändert werden (zoomen), wobei die im Fenster enthaltenen Grafikelemente automatisch neu gezeichnet werden. Standardmäßig ist das Koordinatensystem  $x = 0 \dots 1$ ,  $y = 0 \dots 1$  mit dem Nullpunkt in der unteren linken Ecke des Fensters. Wie bei der `Turtle`-Klasse besitzt das `GPanel` ein „Erinnerungsvermögen“ an den letzten gezeichneten Punkt (**Zeichnungsposition**). Man nennt diesen auch den **Grafik-Cursor**, obschon er nicht sichtbar ist. Mit `draw(x, y)` wird beispielsweise eine Linie vom aktuellen Grafik-Cursor bis zum Punkt  $(x, y)$  gezeichnet und der Grafik-Cursor an die neue Stelle  $(x, y)$  gesetzt.

In der Tab. 5.1 sind einige Methoden der Klasse `GPanel` aufgeführt.

<code>move(double x, double y)</code>	setzt die aktuelle Zeichnungsposition
<code>draw(double x, double y)</code>	Linie von der aktuellen Position zu $(x, y)$
<code>line(double x1, double y1, double x2, double y2)</code>	Linie $(x1, y1) - (x2, y2)$

<code>point(double x, double y)</code>	Punkt (x, y)
<code>rectangle(double width, double height)</code>	Rechteck mit Zentrum an der aktuellen Position
<code>fillRectangle(double width, double height)</code>	gefülltes Rechteck, Zentrum an der aktuellen Position
<code>circle(double Radius)</code>	Kreis mit Zentrum an der aktuellen Position
<code>fillCircle(double Radius)</code>	Kreis gefüllt
<code>arc(double Radius, double <math>\alpha_1</math>, double <math>\alpha_2</math>)</code>	Kreisbogen, Startwinkel $\alpha_1$ , Öffnungswinkel $\alpha_2$
<code>fillArc(double Radius, double <math>\alpha_1</math>, double <math>\alpha_2</math>)</code>	gefüllter Kreisbogen
<code>triangle(double x1, double y1, double x2, double y2, double x3, double y3)</code>	Dreieck bestimmt durch Eckpunkte
<code>text(String)</code>	String an der aktuellen Position
<code>image(bild gif, double x, double y)</code>	Bild an der Stelle (x,y)
<code>color(Color color)</code>	Zeichenstiftfarbe auf color
<code>clear()</code>	löscht das Grafikfenster
<code>enableRepaint(boolean doRepaint)</code>	false: schaltet das automatische repaint aus
<code>getCharWait()</code>	wartet auf Tastendruck

**Tab. 5.1** Die wichtigsten Methoden der Klasse *GPanel*

Beim Konstruieren einer Instanz von *GPanel* wird das Grafik-Fenster automatisch auf dem Bildschirm sichtbar. Zur Einstimmung und zum Spaß erstellen wir ein Zufallsgemälde aus 100 farbigen Rechtecken. Dazu beschaffen wir uns mit `Math.random()` drei Zufallszahlen zwischen 0 und 1 und erzeugen damit eine Zufallsfarbe (vom Farbtyp `sRGB`). Das Zentrum und die Größe des Rechtecks werden ebenfalls zufällig ausgewählt.

```
// Painting.java
```

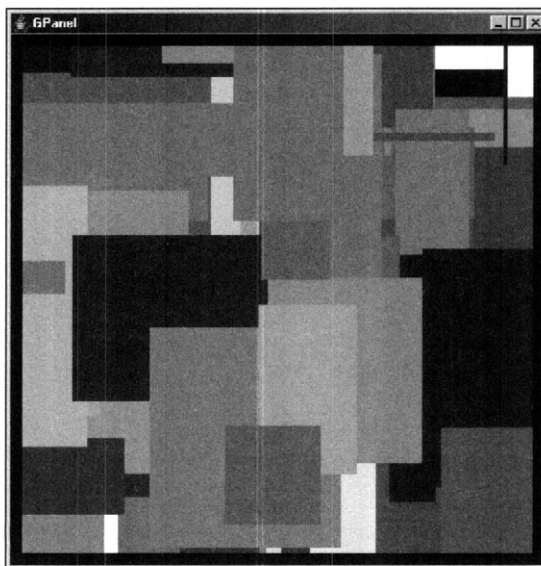
```
import ch.aplu.util.*;
import java.awt.*;
```

```
class Painting
{
    Painting()
```

```
{
    GPanel p = new GPanel();
    for (int i = 0; i < 100; i++)
    {
        float r = (float)Math.random();
        float g = (float)Math.random();
        float b = (float)Math.random();
        p.color(new Color(r, g, b));
        p.move(Math.random(), Math.random());
        p.fillRect(0.5 * Math.random(),
                    0.5 * Math.random());
    }
    p.color(Color.black);
    p.lineWidth(20);
    p.move(0.5, 0.5);
    p.rectangle(1, 1);
}

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

Das Resultat ist zwar kein Kunstwerk, gemessen am Aufwand aber durchaus sehenswert (Abb. 5.1).



**Abb. 5.1** Das Zufallsgemälde (als Graustufenbild)

Zur Erläuterung wichtiger neuer Begriffe betrachten wir wiederum eine Treppe mit 7 Stufen. Gemäß den Regeln der strukturierten Programmierung deklarieren wir eine Methode `step(double size)`, welche eine einzelne Stufe mit der Höhe und Breite `size` an der aktuellen Grafik-Cursorposition zeichnet. Im Konstruktor der Applikationsklasse rufen wir diese Methode mit einer `for`-Schleife 7 Mal auf. Da die Methode `step()` das `GPanel` verwendet, deklarieren wir die `GPanel`-Referenz als Instanzvariable. Diese sind in der ganzen Klasse sichtbar und haben damit die Eigenschaft von **klassenglobalen Variablen**. (Als Alternative könnte man `step()` eine `GPanel`-Referenz als Parameter übergeben.)

*In fast allen Programmiersprachen gibt es globale Variablen, die im ganzen Programm sichtbar sind. Diese sind zwar praktisch und beim Anfänger sehr beliebt, die missbräuchliche Verwendung solcher Variablen führt aber zu schwerwiegenden Programmierfehlern. In Java gibt es keine von der Syntax vorgesehenen globalen Variablen, sie können aber durch Instanzvariablen (insbesondere durch statische Variablen) simuliert werden und sind dann genau so gefährlich. Instanzvariablen werden auch Feldvariablen (field variables, field member, data member) genannt. Wir vermeiden diese Bezeichnung wegen der Verwechslungsgefahr mit dem Begriff Feld (Array).*

Da gemäß Vorgabe die Stufe an der aktuellen Cursor-Position gezeichnet werden muss, holen wir in `step()` die x-Koordinate mit `getPosX()` zurück.

```
// Treppe.java

import ch.aplu.util.*;

class Treppe
{
    GPanel p = new GPanel();

    Treppe()
    {
        double size = 0.1;
        int nbSteps = 7;
        for (int i = 0; i < nbSteps; i++)
            step(size);
    }


    void step(double size)
    {
        double x = p.getPosX();
        p.draw(x, x + size);
        p.draw(x + size, x + size);
    }

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



In diesem Programm wird der Vorteil, in `main()` lediglich eine Instanz der Applikationsklasse zu erzeugen, noch einmal offensichtlich. Der Konstruktor und die Methoden können problemlos auf die Instanzvariablen zugreifen. Im Gegensatz dazu kann `main()`, da es statisch ist, nur statische Instanzvariablen verwenden.

Die Variable `size` in `Treppe()` nennen wir eine **lokale Variable**, da sie nur in der Methode sichtbar ist, in der sie deklariert wird. Wir sprechen dabei auch vom **Geltungsbereich (Scope)** einer Variablen. Eine lokale Variable entsteht bei ihrer Deklaration und ihre **Lebensdauer** endet spätestens beim Verlassen der Methode. Im Gegensatz zu lokalen Variablen sind Instanzvariablen in der ganzen Klasse, falls keine besonderen Vorkehrungen durch **access specifiers** getroffen werden, sogar in allen Klassen desselben Packages sichtbar. Das Verständnis der Geltungsbereiche von Variablen ist in allen Programmiersprachen von großer Wichtigkeit, und es gilt ein wesentlicher Grundsatz:

 **Der Geltungsbereich (Scope) von Variablen sollte immer so klein wie möglich gemacht werden.**

Lokale Variablen können auch innerhalb von Anweisungsblöcken deklariert werden und sind dann nur im entsprechenden Block sichtbar. Es ist daher nicht immer gleichgültig, ob zusätzliche Blockklammern gesetzt werden, weil im Block deklarierte Variablen nur dort sichtbar sind. Wie wir bereits wissen, ist Schleifenvariable `i` ebenfalls eine blocklokale Variable, die nur im Schleifenkörper sichtbar ist. In Abb. 5.2 sind die Geltungsbereiche grafisch dargestellt.

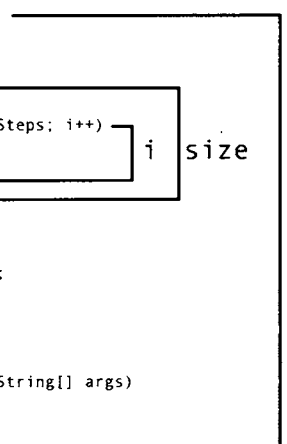
```
// Treppe.java
import ch.aplu.util.*;

class Treppe
{
    GPanel p = new GPanel();

    Treppe()
    {
        double size = 0.1;
        int nbSteps = 7;
        for (int i = 0; i < nbSteps; i++)
        {
            step(size);
        }
    }

    void step(double s)
    {
        double x = p.getPosX();
        p.draw(x, x + s);
        p.draw(x + s, x + s);
        s = 10;
    }

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



**Abb. 5.2** Geltungsbereich (Scope) von Variablen

In der Regel sollte man Variablen mit gleich lautendem Namen vermeiden. Deklariert man zwei Variablen mit demselben Namen in einem Block mit gleicher Sichtbarkeit, so ergibt

sich ein Compilationsfehler. In verschiedenen Sichtbarkeitsblöcken stören sich aber die Variablen mit demselben Namen grundsätzlich nicht. So wird oft `i` als Schleifenvariable an mehreren Stellen des Programms ohne gegenseitige Beeinflussung verwendet. Wird in einem inneren Block eine Variable mit demselben Namen deklariert, so **überdeckt** diese die Variable des äußeren Blocks. Darunter versteht man die Eigenschaft, dass die äußere Variable im inneren Block nicht ansprechbar ist. Ihr Wert und ihre Sichtbarkeit bleiben aber im Äußern des Block erhalten.

Beim Aufruf der Methode `step()` werden die aktuellen Parameterwerte in die formalen Parameter kopiert und dadurch der Methode übergeben. (Man nennt solche Parameter auch **Werteparameter** und diese Übergabeart **call-by-value**). Im Innern der Methode verhalten sich formale Parameter wie lokale Variable, d.h. sie können verändert werden, was aber selten nützlich ist. Es ist sehr wichtig, sich bewusst zu sein, dass beim Aufruf einer Methode immer nur die Werte und nicht etwa die Variablen übergeben werden. Eine Veränderung der Werte der formalen Parameter innerhalb der Methode hat also keine Auswirkungen auf die Übergabevariablen. Man kann dies leicht nachprüfen, indem man am Ende der Methode `step()`

```
s = 10;
```

setzt, was keinerlei Auswirkungen hat.

Da der Name des formalen Parameters unwesentlich ist, kann man den Parameter von `step()` statt `s` auch `size` nennen, was üblicherweise in solchen Fällen auch gemacht wird. Dabei sollte man sich aber bewusst sein, dass dieser formale Parameter mit der gleichlautenden lokalen Variablen in `Treppe()` nichts zu tun hat.

Man ist versucht, für die `for`-Schleife die Koordinate `x` zu verwenden und die Schleife abbrechen, sobald diese bei einer siebenstufigen Treppe mit der Stufengröße `0.1` den Wert `0.7` erreicht. Wir verwenden den Operator `!=`, der zwei Größen auf Ungleichheit prüft.

```
for (double x = 0; x != nbStep*size; x += size)
{
    step(size);
}
```

Obschon kein logischer Fehler vorliegt, benimmt sich das Programm katastrophal: Für eine `size` von `0.01` werden tatsächlich 7 Stufen gezeichnet, für eine `size` von `0.05` oder `0.1` hingegen viel mehr. Offenbar versagt der Computer bereits bei einfachen Rechenaufgaben. Dies hängt damit zusammen, dass er Dezimalzahlen nur mit begrenzter Genauigkeit darstellen kann. Auf Grund dieser schlechten Erfahrungen werden wir uns an folgende Regeln halten:



**Dezimalzahlen nie auf Gleichheit oder Ungleichheit testen**

und



Falls man eine exakte Anzahl von Schleifendurchgängen benötigt, sind ganzzahlige Schleifenzähler zu verwenden.

## 5.4 Überladen

In Java ist es möglich, mehrere Methoden mit demselben Namen zu deklarieren. Man sagt, dass diese Methoden **überladen** sind. Damit der Compiler beim Aufruf erkennt, welche der Methode nun tatsächlich gemeint ist, müssen sich die überladenen Methoden durch die Parameterliste unterscheiden. Dabei werden sowohl Anzahl wie Datentyp der Parameter berücksichtigt, hingegen nicht die Parameternamen und auch nicht der Rückgabotyp. Generell kann man davon ausgehen, dass der Compiler in dieser Hinsicht ebenso intelligent ist wie der Programmierer. Kann der Programmierer erkennen, welche Version der überladenen Methoden aufgerufen wird, so ist dies auch dem Compiler möglich.

Vom Überladen sollte man sparsam Gebrauch machen. Es ist nur sinnvoll, Methoden zu überladen, die einen sehr engen Zusammenhang haben. Überladen ist aber nie eine Notwendigkeit, sondern kann höchstens zur Eleganz eines Programms beitragen. Oft werden Konstruktoren überladen, damit es möglich ist, das Objekt auf verschiedene Arten zu initialisieren. Beispielsweise besitzt der Konstruktor von `GPanel` die Versionen

```
GPanel()
```

und

```
GPanel(double xmin, double xmax, double ymin, double ymax)
```

In der ersten Version entsteht ein Fenster mit Standard-Koordinaten  $(0, 1, 0, 1)$ , in der zweiten ist der Koordinatenbereich frei wählbar. Statt `GPanel(0, 1, 0, 1)` kann man also kurz `GPanel()` schreiben.

*Besitzt die Klasse mehrere Instanzvariablen, die initialisiert werden müssen, so ist man versucht, alle möglichen Initialisierungsarten durch überladene Konstruktoren anzubieten. Man spricht von der Konstruktor-Verrücktheit (**constructor madness**).*

Wie bei `GPanel` werden in der Praxis Konstruktoren oft überladen, damit man häufig vorkommende Initialisierungswerte weglassen kann (**Standardwerte, parameter defaulting**). Betrachten wir als Beispiel die Klasse `Bankkonto`, so ist es zweckmäßig, für die Konten, welche bei einem momentan festen Zinssatz von 2.5% erzeugt werden, einen eigenen Konstruktor einzuführen. Es wäre allerdings ein hässlicher Schnellschuss, den Code des allgemeinen Konstruktors mit

```
Bankkonto(double kap, double zs)
{
    kapital = kap;
    zinssatz = zs;
```

```

}

Bankkonto(double kap)
{
    kapital = kap;
    zinssatz = 2.5;
}

```

zu duplizieren. Vielmehr verwendet man in der spezialisierten Methode die allgemeine Methode mit den speziellen Parameterwerten. Allerdings kann man Konstruktoren nicht wie gewöhnliche Methoden aufrufen, sondern muss das Schlüsselwort `this` verwenden, das zudem an erster Stelle im überladenen Konstruktor stehen muss. Wir schreiben also für den zweiten Konstruktor

```

Bankkonto(double kap)
// Überladener Konstruktor mit festem Zinssatz
{
    this(kap, 2.5);
}

```

Auch in der Klasse `Turtle` gibt es überladene Konstruktoren. Der Konstruktor

```
Turtle(Turtle t)
```

erzeugt eine neue `Turtle` im gleichen Fenster (playground) wie die `Turtle t`. Der Konstruktor

```
Turtle(Color c)
```

erzeugt eine `Turtle` mit gegebener Farbe. Schließlich kombiniert der Konstruktor

```
Turtle(Turtle t, Color c)
```

beide Möglichkeiten. Im folgenden Beispiel wird davon Gebrauch gemacht.

```

// TuEx7.java

import ch.aplu.turtle.*;
import java.awt.Color;

class TuEx7
{
    TuEx7()
    {
        Turtle john = new Turtle();

        Turtle laura = new Turtle(john, Color.yellow);
    }
}

```

```
    laura.setPos(0, 100);
    laura.left(90);

    for (int i = 0; i < 5; i++)
    {
        step(john);
        step(laura);
    }
}

void step(Turtle t)
{
    t.forward(20);
    t.left(90);
    t.forward(20);
    t.right(90);
}

public static void main(String[] args)
{
    new TuEx7();
}
}
```

# 6 Datentypen, Operatoren

In diesem Kapitel wird eine Vollständigkeit der Themen angestrebt, weil diese zu den Grundlagen der numerischen Datenverarbeitung (in irgendeiner Programmiersprache) gehören. Dabei zeigt sich, dass auch beim Programmieren der Teufel im Detail steckt. Für den ersten Einstieg in das Programmieren können viele der Einzelheiten etwas großzügig durchgelesen werden, um den Leser nicht zu entmutigen.

## 6.1 Basistypen

Jede Variable besitzt in Java einen eindeutigen Datentyp. Dies ist nicht in jeder Programmiersprache der Fall. Wir sprechen bei Java von einer stark typisierten Programmiersprache oder von **starker Typenbindung**. Die starke Typenbindung ermöglicht es dem Compiler, gewisse schwerwiegende Programmierfehler bereits als Syntaxfehler zu erkennen, erhöht aber den Programmieraufwand. Darum werden nicht typisierte Sprachen vor allem für das *schnelle Prototyping* und als *Scriptsprachen* auf dem Web eingesetzt.

In einer typisierten Programmiersprache kann man sich unter einer Variablen einen Speicherplatz vorstellen, der einen Namen besitzt und von dem man weiß, welche Art von Daten er aufnehmen kann. Der Speicherplatz wird vor dem ersten Gebrauch mit einer **Variablen-deklaration** reserviert. Dabei werden Name und Datentyp festgelegt. Damit sind sowohl der Wertebereich, wie auch die erlaubten Operationen bestimmt.

In Java gehören die Variablen zwei grundsätzlich verschiedenen Datentypen an, den **Basistypen** und den **Referenztypen**. Wir werden in Kurzsprechweise sagen, dass eine Variable ein Basistyp oder ein Referenztyp *sei*. In der Variablendeklaration wird zuerst der Typ und dann der Name der Variablen angegeben, beispielsweise

```
double x;  
Turtle john;
```

In Kurzsprechweise sagen wir, *x ist ein double, john ist eine Turtle* (an Stelle von: *x hat den Datentyp double, john ist eine Referenz der Klasse Turtle*).

Grundsätzlich handelt es sich bei Referenztypen um Variablen, mit denen man Objekte ansprechen kann. Aus Effizienzgründen gibt es in Java neben den Objekten aber auch Datentypen, die nicht den Regeln der OOP unterworfen sind, die wir **Basistypen** nennen (üblich sind auch: **elementare, simple, primitive oder numerische Datentypen**). Die Tab. 6.1 zeigt die

Basistypen mit ihren wichtigsten Eigenschaften. Ein großer Vorteil ist, dass in Java Größe und Format der Basistypen unabhängig von der verwendeten Computerplattform sind.

Datentyp/ Schlüsselwort	Beschreibung	Größe/Format
<i>Ganzzahlen</i>		
Byte	Ganze Zahlen von -128 bis +127 ( $-2^7$ bis $2^7-1$ )	8-bit Zweierkomplement (1 byte)
Short	Ganze Zahlen von -32768 bis +32767 ( $-2^{15}$ bis $2^{15}-1$ )	16-bit Zweierkomplement (2 bytes)
Int	Ganze Zahlen von -2 147 483 648 bis 2 147 483 647 ( $-2^{31}$ bis $2^{31}-1$ )	32-bit Zweierkomplement (4 bytes)
Long	Ganze Zahlen von -9 223 372 036 854 775 808 bis 9 223 372 036 854 775 807 ( $-2^{63}$ bis $2^{63}-1$ )	64-bit Zweierkomplement (8 bytes)
<i>Fließkommazahlen</i>		
float	Dezimalzahl, ungefähr 7 Ziffern, Exponent $10^{-38}$ .. $10^{+38}$	32-bit IEEE 754 (24-bit Mantisse/8-bit Exponent)
double	Dezimalzahl, ungefähr 16 Ziffern, Exponent $10^{-308}$ .. $10^{+308}$	64-bit IEEE 754 (53-bit Mantisse/11-bit Exponent)
<i>Andere Typen</i>		
char	Zeichen	16-bit Unicode (kein Vorzeichen)
boolean	Wahrheitswert wahr oder falsch	true/false (1 byte)

**Tab. 6.1** Basistypen

Die Werte einiger Basistypen können auf verschiedene Arten mit Hilfe von **Literalen** (konstante symbolische Ausdrücke) wie folgt dargestellt werden:

`int`

Ziffernfolgen (ohne Dezimalpunkt), die nicht mit 0 beginnen, stellen standardmäßig ganze Zahlen zur Basis 10 dar. Wird eine 0 vorangestellt, so wird die Zahl **oktal** (Basis 8) aufgefasst (0123 steht für  $3*1 + 2*8 + 3*64 = 211$ (dezimal). Ein vorgestelltes 0x oder 0X bezeichnet eine **hexadezimale** Zahl (0x123 steht für  $3*1 + 2*16 + 1*256 = 291$ (dezimal)). Die Hexadezimaldarstellung erlaubt 16 verschiedene Ziffern und zwar 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F (Buchstaben auch kleingeschrieben). Die Buchstaben stehen dabei für die dezimalen Werte 10, 11, 12, 13, 14, 15.