Bernhard Steppan

Einstieg in Java

- Für Programmiereinsteiger und Java-Neulinge
- Mit vielen Beispielen und kommentierten Lösungen
- Verstehen, anwenden und nachschlagen

Galileo Computing

Liebe Leserin, lieber Leser,

vor etwa drei Jahrzehnten begann der Siegeszug objektorientierter Sprachen – doch nur wenige setzten sich durch. Heute ist Java neben C++ die wichtigste Programmiersprache bzw. Plattform; Java nimmt eine Spitzenposition ein, wenn es um den Anteil an zurzeit realisierten Softwareentwicklungsprojekten geht.

Sie sind also auf dem richtigen Weg und haben sich gut entschieden. Und jetzt können Sie es kaum erwarten, die ersten Programme zu schreiben? Nur zu, keiner hält Sie auf.

Neben einem ausgezeichneten Lehrwerk haben Sie auch eine Buch-CD erworben, die keine Wünsche offen lässt. Das aktuelle SDK finden Sie ebenso wie alle gängigen Java-Tools. Keine Zeit wird verschwendet für lange Downloads, die kurz vor Beendigung eh immer abbrechen. Und noch etwas haben wir uns einfallen lassen: 1300 Seiten, also das vollständige Java-Kultbuch »Java ist auch eine Insel« findet sich außerdem auf der Buch-CD. Zum Weiterlernen und leichtem Navigieren eignet sich diese HTML-Version hervorragend. Es sollten also keine Fragen offen bleiben – zumindest die zu Java beantworten wir vollständig.

Sollten Sie kritische und freundliche Anmerkungen haben, so zögern Sie nicht, sich mit Herrn Steppan oder mir in Verbindung zu setzen. Ihre Verbesserungsvorschläge und Ihr Zuspruch sind unentbehrlich für weitere gute Auflagen.

Ich bin gespannt auf Ihre Rückmeldung und wünsche ich Ihnen viel Spaß beim Lesen und Programmieren!

Judith Stevens-Lemoine
Lektorat Galileo Computing

judith.stevens@galileo-press.de

Galileo Press • Gartenstraße 24 • 53229 Bonn

Auf einen Blick

Inhalt

15 Datenbankanwendungen 407

16 Dynamische Websites 427

Teil V Anhang

21 Werkzeuge 499

Vorwort

»Es gibt drei goldene Regeln, um ein Fachbuch zu schreiben – leider sind sie unbekannt.« (frei nach William Somerset Maugham)

Liebe Leserin, lieber Leser!

Unter den dicken Java-Büchern ist dies sicher eines der dünnsten. Das liegt daran, dass sich dieses Buch auf das konzentriert, was für Java-Einsteiger am wichtigsten ist: möglichst schnell und ohne Ballast ansprechende Java-Programme zu entwickeln.

Aufbau des Buchs

Dieses Buch gliedert sich in fünf Teile. Es führt Sie von den Grundlagen der Softwareentwicklung (Teil I) über eine Java-Einführung (Teil II) zu der Entwicklung stabiler, professioneller Java-Programme (Teil III). Diese Java-Programme werden Schritt für Schritt in Tutorien entwickelt. Jedes dieser Tutorien schließt mit Übungsaufgaben ab, die Ihnen helfen, Ihr Wissen zu vertiefen. Die Musterlösungen finden Sie im vorletzten Teil des Buchs (Teil IV). Der Anhang (Teil V) rundet das Buch mit je einem Kapitel über Java-Werkzeuge, die Hardware-Grundlagen, einem Glossar und einem Literaturverzeichnis ab.

Beispielprogramme

Dieses Buch enthält neben rund hundert kleineren Beispielprogrammen acht größere, sorgfältig dokumentierte Projekte aus den wichtigsten Bereichen der Java-Programmierung. Diese Projekte sind als Vorlage für Ihre eigenen Arbeiten gedacht. Auf der beiliegenden CD finden Sie sowohl diese Projekte als auch alle anderen Beispielprogramme und Lösungen komplett mit Projektdateien für die Entwicklungswerkzeuge *JBuilder* und *Together*.

Ebenfalls enthalten ist eine Kurzanleitung für den Import der Beispiele in die Entwicklungsumgebungen *Eclipse* und *NetBeans*. Die Installation der Beispielprogramme ist einfach und wird zudem durch ein Installationsprogramm unterstützt. Die Beispiele liegen als ausführbare Dateien für Windows, Linux, Solaris und Mac OS X vor. Um ein Beispiel zu starten, genügt also in den meisten Fällen ein Doppelklick auf das fertige Programm oder ein Start von der Kommandozeile aus.

Werkzeuge

Das Buch ist kein Ratgeber bei der Auswahl von Java-Werkzeugen. Trotzdem enthält es im Anhang ein Kapitel zu Java-Werkzeugen. Da ständig neue Werkzeuge erscheinen, finden Sie eine aktuelle Fassung auf meiner Website. Dort können Sie zudem eine Marktübersicht über die momentan aktuellen Tools herunterladen. Die Beispielprogramme dieses Buchs lassen sich übrigens mit den meisten dort aufgeführten Werkzeugen wie zum Beispiel Eclipse, JBuilder, Java Development Kit, NetBeans und Together problemlos verwenden.

Vorkenntnisse

Ob Sie das Buch zum Selbststudium verwenden, zur Prüfungsvorbereitung oder weil Programmieren Ihr Hobby ist: Sie benötigen in keinem dieser Fälle Vorkenntnisse über Computerprogrammierung. Für einige Kapitel setze ich allerdings ein minimales Mathematikverständnis voraus.

Schriftdarstellung

Um verschiedene Textteile deutlicher hervorzuheben, verwendet dieses Buch eine einheitliche Schriftdarstellung (→ Tabelle 1).

Textteil	Darstellung
Programmquelltext (Listings)	`Schrift mit fester Zeichenbreite`
Optionale Parameter	[]
Menübefehle, deren Menüs bzw. Untermenüs	Menü | Befehl | Befehl
Java-Bezeichner wie Variablen, Methoden und Klassen	Kursivschrift
Programmbeispiel auf der CD: hier Kapitel 04, Beispiel 2	//CD/examples/ch04/ex02
Datenamen, Pfadangaben und Programmausgaben	`Schrift mit fester Zeichenbreite`
Querverweise auf andere Buchteile (Abbildungen, Listings, Kapitel etc.)	*Æ*

Tabelle 1 Verwendete Schriftkonventionen

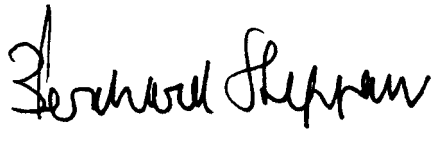
Manche Quelltexte sind aus Platzgründen nicht komplett im Buch abgedruckt, sondern nur die zum Verständnis wichtigen Teile. Diese Quelltexte habe ich mit einer Ellipse `(...)` gekennzeichnet. Sie finden die Quelltexte natürlich vollständig auf der beiliegenden CD.

Danksagung

Ich möchte mich herzlich bedanken bei meiner Frau Christiane, die mich wie immer sehr unterstützt hat, meinem Sohn für die unzähligen selbstlosen Tests des Beispielprogramms »Memory« und meiner Lektorin Judith Stevens-Lemoine für die unproblematische Zusammenarbeit. Die Firmen Borland, IBM und Sun haben die Erlaubnis erteilt, ihre Werkzeuge auf die CD zum Buch zu brennen – vielen Dank auch ihnen für die Unterstützung!

Schreiben Sie mir, ...

wenn Sie Fragen, Anregungen oder Verbesserungswünsche haben. Richten Sie bitte alle Post an bernhard@steppan.de oder den Verlag Galileo Press – nun wünsche ich Ihnen aber viel Spaß bei der Lektüre und viel Erfolg bei der Entwicklung Ihrer Java-Programme!



Bernhard Steppan

Bad Homburg, im Juni 2003

Teil I
Basiswissen

Das Erlernen jeder Programmiersprache beginnt mit den Grundlagen der Softwareentwicklung. Darum fängt auch dieses Buch mit den grundlegenden Konzepten der Datenverarbeitung und Programmierung an. Dieser Teil stellt die Konzepte in drei Kapiteln vor: Digitale Informationsverarbeitung (Kapitel 1), Programmiersprachen (Kapitel 2) und Objektorientierte Programmierung (Kapitel 3).

1 Digitale Informationsverarbeitung

»Dies hier ist ein erstes Kapitel, welches verhindern soll, dass vorliegendes Werkchen mit einem zweiten Kapitel beginne.« (Franz Werfel)

1.1 Einleitung

Alle Informationen, ganz gleich, ob es sich um ein Java-Programm oder einen Brief handelt, sind für den Computer nichts anderes als Informationen. Dieses Kapitel gibt Ihnen einen Überblick darüber, wie der Computer diese Informationen speichert und verarbeitet.

1.2 Zahlensysteme

Zahlensysteme dienen dazu, Zahlen nach einem bestimmten Verfahren darzustellen. Dazu besitzt jedes Zahlensystem einen spezifischen Satz an Ziffern. Es existieren sehr viele verschiedene Zahlensysteme. Für die Java-Programmierung ist die Kenntnis von Dezimal-, Binär- und Hexadezimalsystem in den meisten Fällen ausreichend.

1.2.1 Dezimalsystem

Das Dezimalsystem (lat. decem: zehn) verwendet bis zu zehn Ziffern zur Zahlendarstellung. Da es für den Menschen besonders einfach ist, mit diesem System zu rechnen, ist es das heute in aller Welt bevorzugte Zahlensystem.

Zehnerpotenzen

Die → Abbildung 1.1 zeigt, dass sich eine Zahl wie beispielsweise 214 in Dezimaldarstellung aus lauter Zehnerpotenzen (10^X) zusammensetzt. Man sagt, alle Zahlen beziehen sich auf die Basis 10. Teilt man die Zahl 214 in eine Summe von Zehnerpotenzen auf, ergibt sich folgendes Bild: $214 = 2 * 10^2 + 1 * 10^2 + 4 * 10^1$.

Verwendet man unterschiedliche Zahlensysteme parallel in einer Darstellung, so schreibt man zur besseren Unterscheidung entweder eine tiefgestellte Zehn (214_{10}) an die Dezimalzahl oder man verwendet ein Doppelkreuz als Präfix (#214) beziehungsweise eine Abkürzung als Postfix (214d). Ich bevorzuge die erste Schreibweise.

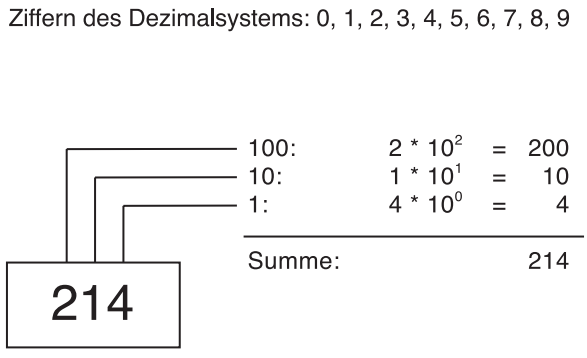


Abbildung 1.1 Darstellung der Dezimalzahl 214 mit Hilfe des Dezimalsystems

1.2.2 Binärsystem

Das Binärsystem (lat. bini: je zwei) benötigt im Gegensatz zum Dezimalsystem nur maximal zwei Ziffern zur Zahlendarstellung. Das Zahlensystem nennt sich auch Digital- (lat. digitus: fingerbreit, Zoll) oder Dualsystem (lat. duo: zwei, beide). Zahlen dieses Systems bezeichnet man als Binärzahlen oder Digitalzahlen.

Digitalcomputer

Das Binärsystem passt sehr gut zu der Informationsverarbeitung heutiger Computer. Deren Speicher bestehen aus sehr vielen kleinen primitiven Bauelementen (Flip-Flops), die nur zwei elektrische Zustände einnehmen können: *hohe Spannung* oder *niedrige Spannung*.

Jedes Flip-Flop mit niedriger Spannung in einem Computer entspricht informationstechnisch einer Null, jedes mit hoher Spannung einer Eins. Praktisch alle heutigen Computer basieren auf dieser Bauweise mit primitiven Bauelementen. Sie verarbeiten ausschließlich Digitalzahlen und werden daher auch Digitalcomputer genannt.

Binärprogramme

Computerprogramme, die für den Computer auch nichts anderes als Informationen sind, bestehen aus einer Abfolge von Stromimpulsen in einer bestimmten Zeiteinheit. Jeder Stromimpuls entspricht einer digitalen Eins. Fehlt ein Impuls, entspricht dies einer Null. Das Format, in dem ein Computer Software direkt ausführen kann, bezeichnet man nach dem Zahlensystem als Binärformat. Die Programme nennen sich Binärprogramme oder Maschinenprogramme.

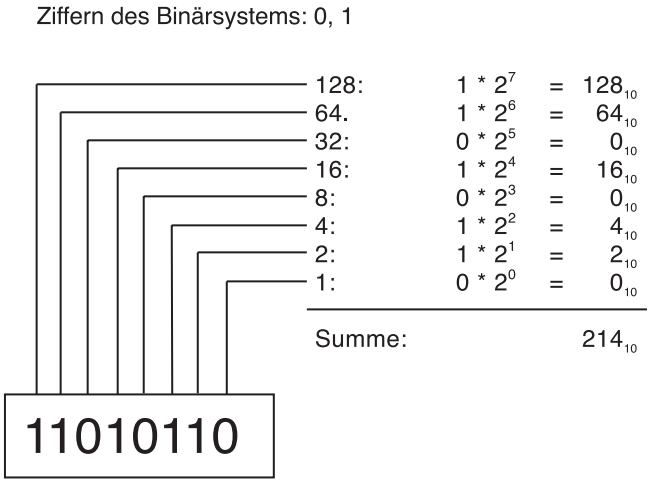


Abbildung 1.2 Darstellung der Dezimalzahl 214 mit Hilfe des Binärsystems

Wenn Sie einen Blick auf → Abbildung 1.2 werfen, sehen Sie, wie die Dezimalzahl 214 in einer *binären* Form dargestellt wird. Es ist wichtig zu betonen, dass die Darstellung hier binär interpretiert wird. Später, im Abschnitt über → Kodierung von Zeichen, werden Sie sehen, dass digitale Zahlenkolonnen auch ganz anders interpretiert werden können.

Zweierpotenzen

Bei der Binärdarstellung besteht die Dezimalzahl 214 aus lauter Zweierpotenzen, deren Basen sich auf die Zahl 2 beziehen. Die Summe ergibt sich durch Addition folgender Summanden: $214_{10} = 1 * 2^7 + 1 * 2^6 + 0 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$. Jeder dieser Summanden entspricht der kleinsten Informationseinheit, dem legendären *Bit*, auf das ich im nächsten Abschnitt noch ausführlich eingehen werde.

Wertebereich

In → Tabelle 1.1 sehen Sie, wie viele Informationen sich mit drei Bit darstellen lassen. Die kleinste und die größte darstellbare Zahl ergeben den *Wertebereich*. Die maximale Anzahl der Informationen können Sie mit folgender Formel berechnen: $Anzahl = 2^{(Anzahl\ Bit)}$. In diesem Fall ergibt sich die Anzahl aus $2^3 = 2 * 2 * 2 = 8_{10}$.

Stellen Sie sich vor, Sie wollten die Dezimalzahl 214 im Binärsystem statt durch Flip-Flops mit Hilfe von Glühlampen darstellen oder eine solche Zahl speichern. Dazu bräuchten Sie für jeden Summanden (von $1 * 2^7$ bis $0 * 2^0$) eine Glühlampe. Das »Schaltbrett« besäße also acht Glühlampen (→ Abbildung 1.3).

Dezimalzahl	Binärzahl
0	0 0 0
1	0 0 1
2	0 1 0
3	0 1 1
4	1 0 0
5	1 0 1
6	1 1 0
7	1 1 1

Tabelle 1.1 Der Wertebereich einer Informationseinheit von drei Bit

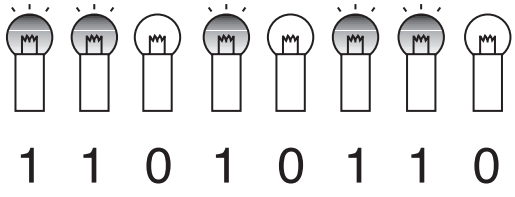


Abbildung 1.3 Darstellung der Zahl 214_{10} mit Hilfe von Glühlampen

Wenn der Informatiker in einem Dokument verschiedene Zahlensysteme nebeneinander verwendet, muss er sie kennzeichnen, damit der Leser sie unterscheiden kann. Zahlen des Binärsystems kennzeichnet man entweder durch eine tiefgestellte Zwei (11010110_2) oder man verwendet ein Prozent- oder Dollarzeichen als Präfix (%11010110) beziehungsweise eine Abkürzung als Postfix (11010110b). In diesem Buch verwende ich die erste Schreibweise.

1.2.3 Hexadezimalsystem

Das Hexadezimalsystem, auch Sedezimalsystem genannt (lat. sex, hexa-: sechs), basiert auf sechzehn Ziffern zur Zahlendarstellung (→ Abbildung 1.4). Es ist das bevorzugte Zahlensystem der Computerfachleute, um binäre, vom Computer gespeicherte Informationen darzustellen.

Binärzahlen werden schnell sehr lang und unübersichtlich. Aus diesem Grund haben Informatiker nach einer besseren Darstellungsform für Binärzahlen gesucht und festgestellt, dass sich das Hexadezimalsystem dafür sehr gut eignet. Warum dies der Fall ist, zeigt folgendes Beispiel:

Ziffern des Hexadezimalsystems: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

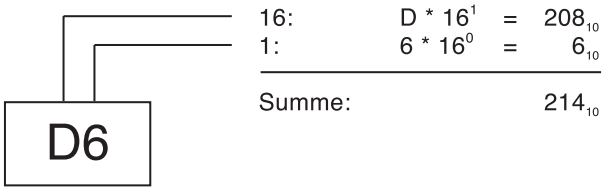


Abbildung 1.4 Darstellung der Dezimalzahl 214 mit Hilfe des Hexadezimalsystems

Sechzehnerpotenzen

Bei der hexadezimalen Darstellung einer Zahl besteht diese aus lauter Potenzen zur Basis 16. Die Summe ergibt sich durch Addition folgender Summanden: $214_{10} = D * 16^1 + 6 * 16^0$. Vollständig auf das Dezimalsystem übertragen lautet die Gleichung $214_{10} = 13 * 16 + 6 * 1$.

Leichte Umwandlung in Binärzahlen

Vergleichen Sie nun die Hexadezimaldarstellung der Dezimalzahl 216 mit der Binärdarstellung (→ Abbildung 1.5). Wenn Sie die acht Stellen der Binärzahl in zwei vierstellige Abschnitte zerlegen, erkennen Sie, wie leicht sich die Binärdarstellung einer Zahl in eine Hexadezimaldarstellung umwandelt lässt. Jeder geübte Programmierer ist mit Hilfe des Hexadezimalsystems in der Lage, die native Zahlendarstellung des Computers, das Binärsystem, besser zu lesen.

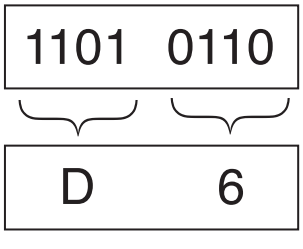


Abbildung 1.5 Vergleich der Darstellung von Hexadezimal- und Binärsystem

Zur leichteren Unterscheidung markiert man Zahlen des Hexadezimalsystems bei der Darstellung von Zahlen unterschiedlicher Zahlensysteme entweder durch eine tiefgestellte Zahl ($D6_{16}$) oder man verwendet ein Prozent- oder Dollarzeichen als Präfix ($D6) beziehungsweise eine Abkürzung als Postfix (D6h). Im Folgenden verwende ich wieder die erste Schreibweise.

1.3 Informationseinheiten

Wie für physikalische Größen, zum Beispiel Entfernungen oder Gewichte, gibt es auch Maßeinheiten, die den Informationsgehalt angeben. Eine Übersicht der wichtigsten Maßeinheiten und deren Wert finden Sie in → Tabelle 1.2.

Informationseinheit	Wert [Bit]	Wert [Byte]
Bit	1	0,125
Halbbyte (Nibble)	4	0,25
Byte	8 * 1	1
Wort	8 * 2	2
Doppelwort	8 * 4	4
KByte (Kilobyte)	8 * 1024	1024
MByte (Megabyte)	$8 * 1024^2$	1024^2
GByte (Gigabyte)	$8 * 1024^3$	1024^3
TByte (Terabyte)	$8 * 1024^4$	1024^4

Tabelle 1.2 Die wichtigsten Maßeinheiten der Information

1.3.1 Bit

Die kleinste Informationseinheit ist das so genannte *Bit* (engl. binary digit). Mit Hilfe eines Bits lassen sich – wie mit einer Glühlampe – lediglich zwei Zustände speichern: ein- oder ausgeschaltet, leitend oder nicht leitend. Sie haben gesehen, wie viele Bits notwendig sind, um die Dezimalzahl 214 darzustellen. Eine einzelne dieser Speicherzellen konnte diese Information nicht festhalten. Um größere Datenmengen speichern zu können, fasst man deshalb Gruppen von Bits zu Einheiten zusammen.

1.3.2 Byte

Die wichtigste Informationseinheit neben dem Bit ist das Byte. Ein Byte entspricht 8 Bit. Große Datenmengen gibt man in Vielfachen von Byte an, wie zum Beispiel 1 Kilobyte (Abkürzung KByte). 1 KByte entspricht übrigens 1024 Byte und nicht 1000 Byte. Um einer Verwechslung mit dem physikalischen Faktor Kilo ($k = 10^3 = 1000$) vorzubeugen, schreiben die meisten Informatiker das K vor dem Byte mit einem Großbuchstaben, also entweder KB oder KByte.

1.3.3 Wort

Das Wort (2 Byte = 16 Bit) spielt ebenfalls eine große Rolle bei der Darstellung von Informationen. Es wird in → Kapitel 2 bei den Bestandteilen eines Programms wieder auftauchen.

1.4 Kodierung von Zeichen

Wie eingangs schon erwähnt, sind für den Computer alle Informationen, die er verarbeitet, binäre Zahlenströme. Da er nur mit Zahlen operiert, bezeichnet man den Computer (lat. Computator: Rechner) im Deutschen auch sehr richtig als Rechner. Nun möchte man den Computer aber nicht nur dazu verwenden, mathematische Berechnungen durchzuführen, sondern auch, um Zeichen auszugeben.

Da der Computer nur mit dem Binärformat von Zahlen arbeiten kann, bedarf es zur Zeichendarstellung eines Tricks: Die Zeichen des Alphabets müssen in Binärzahlen übersetzt werden. Dieser Vorgang nennt sich Kodierung. Die Kodierung sieht so aus, dass jedes Zeichen, das dargestellt werden soll, eine Binärzahl eindeutig zugewiesen bekommt. Auf diese Weise entstehen Übersetzungstabellen, von denen es drei sehr bedeutende gibt: die ASCII-, ANSI- und Unicode-Tabellen.

1.4.1 ASCII-Code

ASCII ist eine Abkürzung für American Standard Code for Information Interchange, also den amerikanischen Standardcode für Informationsaustausch. Er basierte anfangs darauf, Zeichen mit 7 Bit zu kodieren. Das heißt, der Wertebereich beschränkte sich auf lediglich 128 Zeichen. Nationale Sonderzeichen konnten noch nicht kodiert werden.

Keine nationalen Sonderzeichen

Warum war man damals so sparsam? Zu dem Zeitpunkt, als der Code entwickelt wurde, war der Speicherplatz noch sehr kostbar und es wurde versucht, möglichst wenig davon zu verbrauchen. Außerdem war die Internationalisierung noch nicht so bedeutend und die Darstellung nationaler Sonderzeichen nicht so wichtig.

Hexadezimal	Binär	ASCII
3A	00111010	:
3B	00111011	;
3C	00111100	<

Tabelle 1.3 Ausschnitt aus dem ersten Teil der ASCII-Tabelle

Hexadezimal	Binär	ASCII
3D	00111101	=
3E	00111110	>
3F	00111111	?
40	01000000	@
41	01000001	A
42	01000010	B
43	01000011	C
44	01000100	D
45	01000101	E

Tabelle 1.3 Ausschnitt aus dem ersten Teil der ASCII-Tabelle (Forts.)

Da der Wertebereich von 128 Zeichen viel zu klein für die Menge an Zeichen war, die weltweit verwendet wurden, erweiterte man den ASCII-Code später auf 8 Bit (ein Byte). Damit beträgt der Wertebereich 256 Zeichen ($2^8 = 256$).

Erweiterung um nationale Sonderzeichen

Der erste Teil des normalen und erweiterten ASCII-Codes besteht aus Druckersteuerzeichen, die nicht ausgegeben werden können. Sie dienten beispielsweise dazu, einen Zeilenvorschub auszulösen. Nach diesen Druckersteuerzeichen folgt ein Abschnitt mit Zeichen für die Interpunktion sowie den »normalen« Zeichen des Alphabets. Der erweiterte Teil des ASCII-Codes ist für nationale sowie andere Sonderzeichen reserviert.

Erweiterung nicht standardisiert

Leider war der erweiterte Teil des ASCII-Codes nicht standardisiert, so dass die Sonderzeichen eines ASCII-Textes auf einem IBM-Computer anders dargestellt wurden als auf einem Apple-Computer. Ein deutschsprachiger Brief, der auf einem Apple Macintosh in einem Standardtextformat geschrieben worden war, war auf einem IBM PC schlecht lesbar – alle nationalen Sonderzeichen wurden falsch dargestellt. Um diese Beschränkungen des ACSII-Codes zu überwinden und eine Internationalisierung zu fördern, entwickelte man den ANSI-Code.

1.4.2 ANSI-Code

Der ANSI-Code wurde vom American National Standards Institute (ANSI) festgelegt. Er basiert auf den ersten 127 Zeichen des ASCII-Codes, verwendet aber 16 Bit

zur Darstellung von Zeichen und besitzt daher einen Wertebereich von 65536 Zeichen (2^{16} Zeichen = 65536).

Hexadezimal	Binär	ANSI
C0	11000000	À
C1	11000001	Á
C2	11000010	Â
C3	11000011	Ã
C4	11000100	Ä
C5	11000101	Å
C6	11000110	Æ
C7	11000111	Ç
C8	11001000	È
C9	11001001	É
CA	11001010	Ê
CB	11001011	Ë

Tabelle 1.4 Ausschnitt aus einem Teil der ANSI-Tabelle

Der ANSI-Code war allerdings auch nicht der Weisheit letzter Schluss. Sein Wertebereich war zwar ausreichend, doch nicht international normiert. Daher entschlossen sich Fachleute verschiedener Länder, das Nonplusultra der Zeichencodes zu entwickeln: den international standardisierten Unicode, den auch die Programmiersprache Java verwendet.

1.4.3 Unicode

Der Unicode ist vom Unicode-Konsortium entwickelt worden, einer Vereinigung, die aus Linguisten und anderen Fachleuten besteht. Unicode ist seit der Version 2.0 auch mit der internationalen Norm ISO/IEC 10646 abgestimmt und verwendet wie der ANSI-Code 16 Bit zur Zeichendarstellung. Im Gegensatz zum ANSI-Code ist Unicode jedoch unabhängig vom Betriebssystem, unabhängig vom Programm und unabhängig von der Landessprache.

Zeichen aller Länder

Der Unicode enthält Zeichen aller bekannten Schriftkulturen und Zeichensysteme, darunter das lateinische, tibetanische, kyrillische, hebräische, japanische

und chinesische Alphabet. Damit können Programme und deren Oberflächen problemlos in andere Sprachen übersetzt werden.

Java und Unicode

Für die Java-Programmierung hat der Unicode die größte Bedeutung, weil Java-Programme unter allen Betriebssystemen und in allen Ländern funktionieren müssen. Man kann ohne Übertreibung sagen, dass die Entwicklung des Unicodes eine der Voraussetzungen für einige Merkmale von Java war.

1.5 Kodierung von logischen Informationen

Neben der Kodierung von Zahlen und Zeichen ist die Kodierung logischer Informationen für die Programmierung von großer Bedeutung. Unter logischen Informationen fallen Zustandsinformationen wie *Wahr* oder *Falsch* sowie logische Verknüpfungen wie *Oder* beziehungsweise *Und*. Diese Informationen steuern den Programmfluss, wie das folgende Beispiel zeigt.

1.5.1 Und-Funktion

Viele Programme überprüfen vor dem Programmende, ob ein Dokument (zum Beispiel ein Textdokument wie ein Brief) noch gespeichert werden muss, damit keine Informationen verloren gehen. Das geschieht zum Beispiel folgendermaßen:

- Wenn das Dokument seit dem letzten Mal, an dem es gespeichert wurde, geändert wurde (Bedingung A)
- und (Verknüpfung)
- das Programm beendet werden soll (Bedingung B),
- dann frage den Anwender, ob er das Dokument speichern möchte (Aktion).

Entscheidungstabelle

Das Ganze lässt sich in Form einer Entscheidungstabelle darstellen, wobei Folgendes zu beachten ist: Der Zustand *Wahr* lässt sich im Computer als eine 1 darstellen, der Zustand *Falsch* als eine 0. Ein logisches *Und* zwischen den Bedingungen A und B wird wie folgt geschrieben: $A \wedge B$.

Und-Funktion

Die Tabelle (→ Abbildung 1.6) zeigt Folgendes: Sind beide Bedingungen falsch (Fall 1) und werden sie mit *Und* verknüpft, ist das Ergebnis ebenfalls falsch. Ist dagegen nur eine der Bedingungen falsch und werden sie mit *Und* verknüpft (Fall

2 und 3), ist das Ergebnis ebenfalls falsch. In diesen beiden Fällen ist keine Aktion notwendig.

	A	B	A ^ B	
Fall 1	0	0	0	Keine Aktion
Fall 2	0	1	0	Programmende
Fall 3	1	0	0	Keine Aktion
Fall 4	1	1	1	Dialog zeigen

A: Dokument nicht gespeichert
B: Programm soll beendet werden

Abbildung 1.6 Und-Funktion

Nur im Fall 4 – dann, wenn beide Bedingungen erfüllt sind (Zustand *Wahr*) – wird das Programm einen Dialog einblenden, bevor es sich verabschiedet. In diesem Fall muss das Programm die Antwort des Anwenders auswerten und das Dokument eventuell speichern.

	A	B	A ∨ B	
Fall 1	0	0	0	Keine Aktion
Fall 2	0	1	1	Dialog zeigen
Fall 3	1	0	1	Dialog zeigen
Fall 4	1	1	1	Dialog zeigen

A: Programm soll beendet werden
B: Dokument wird geschlossen

Abbildung 1.7 Oder-Funktion

1.5.2 Oder-Funktion

Das ganze Beispiel lässt sich um eine Bedingung erweitern:

- Wenn das Dokument seit dem letzten Mal, an dem es gespeichert wurde, geändert wurde (Bedingung A)
- und (Verknüpfung)
- das Programm beendet werden soll (Bedingung B)
- oder (Verknüpfung)
- das Dokument geschlossen wird (Bedingung C),
- dann frage den Anwender, ob er das Dokument speichern möchte (Aktion).

Wenn Bedingung A mit B durch ein logisches *Oder* verknüpft wird, schreibt man dies wie folgt: $A \vee B$. Auf den neuen Anwendungsfall übertragen sieht die Gleichung folgendermaßen aus: $A \wedge B \vee C$. Damit eindeutig ist, wie der Ausdruck auszuwerten ist, setzt man ihn besser in Klammern: $A \wedge (B \vee C)$.

1.5.3 Nicht-Funktion

Die Nicht-Funktion findet überall dort Verwendung, wo es notwendig ist, einfache logische Aussagen zu überprüfen. Dabei kehrt sie einfach den Wert einer Information in ihr Gegenteil um. Wenn A = 0 ist, dann ist Nicht-A eben 1. Nicht-A schreibt sich $\neg A = 1$.

	Gespeichert	¬Aktion	
Fall 1	1	0	Keine Aktion
Fall 2	0	1	Dialog zeigen

Abbildung 1.8 Nicht-Funktion

Angenommen, Sie möchten zu einem bestimmten Zeitpunkt überprüfen, ob ein Dokument innerhalb eines Programms gespeichert wurde. Sie benutzen dazu eine Variable namens *Gespeichert*.

Ist der Wert dieser Variablen 1, so ist wahr, dass das Dokument gespeichert wurde. Ist die Variable hingegen 0, so hat der Anwender das Dokument nicht gespeichert. In diesem Fall soll ein Dialog mit der Frage »Wollen Sie jetzt speichern?« eingeblendet werden.

Die Bedingung für das Auslösen dieser Aktion lautet: Falls das Dokument *Nicht-Gespeichert* ist, zeige den Dialog »Dokument sichern«. Nicht-Gespeichert muss wahr sein, damit diese Bedingung erfüllt ist (→ Abbildung 1.8).

1.6 Zusammenfassung

Der Computer speichert alle Informationen mit Hilfe primitiver Bauelemente, die nur zwei Zustände einnehmen können. Diese Bauelemente werden als Träger von binären Zahlen eingesetzt.

Die Binärdarstellung von Informationen nennt sich Binärformat. Im Binärformat gespeicherte Programme heißen Binär- oder Maschinenprogramme. Binär dargestellte Informationen sind für den Menschen nur schlecht verständlich. Aus diesem Grund verwendet man lieber andere Zahlensysteme wie zum Beispiel das Hexadezimal- und das Dezimalsystem.

Die vom Computer gespeicherten Informationen in Form binärer Zahlen lassen sich auf einfache Weise in Dezimal- oder Hexadezimaldarstellung umwandeln. Der Informationsgehalt dieser Daten wird in Bits und Bytes gemessen.

Im Gegensatz zur Zahlendarstellung basiert die Zeichendarstellung auf Codetabellen wie dem ASCII-Code. Für die Java-Programmierung ist von allen Zeichentabellen die Unicode-Tabelle am wichtigsten. Der Unicode erleichtert die Internationalisierung von Programmen, da er Zeichen aller Länder darstellen kann.

1.7 Aufgaben

Versuchen Sie folgende Aufgaben zu lösen:

1.7.1 Zahlensysteme

1. Woher kommt der Name Digitalcomputer?
2. Warum arbeiten heutige Digitalcomputer mit Binärzahlen?
3. Welchen Vorteil bietet das Hexadezimalzahlen bei der Darstellung von Binärzahlen?
4. Wandeln Sie die Jahreszahl $7D3_{16}$ manuell in eine Dezimalzahl um. Beschreiben Sie den Lösungsweg.

1.7.2 Informationseinheiten

1. Was ist die kleinste Informationseinheit eines Computers?
2. Wie viele Bytes haben Sie zur Darstellung der Zahl $7D3_{16}$ benötigt?

3. Wie viele Bytes sind ein KByte?
4. Wie kommt es zu der ungewöhnlichen Schreibweise von KByte?

1.7.3 Zeichenkodierung

1. Warum benötigt man Codetabellen?
2. Was sind die großen Vorteile des Unicodes?

1.7.4 Kodierung logischer Informationen

1. Welche verschiedenen Arten gibt es, logische Informationen zu kodieren?
2. Was ist das Ergebnis von folgendem Ausdruck: $1 \wedge (0 \vee 1)$?

Die Lösungen zu den Aufgaben finden Sie in → Teil IV, Kapitel 17, ab Seite 441.

2 Programmiersprachen

»In keiner Sprache kann man sich so schwer verständigen wie in der Sprache.« (Karl Kraus)

2.1 Einleitung

Dieses Kapitel gibt Ihnen einen Überblick über die babylonische Vielfalt der Programmiersprachen. Es hilft Ihnen, die Programmiersprache Java in den nachfolgenden Kapiteln besser einzuordnen, die Entwicklung der Sprache besser nachzuvollziehen und ihre Konzepte besser zu verstehen.

2.1.1 Verständigungsschwierigkeiten

In → Kapitel 1 haben Sie erfahren, dass ein Digitalcomputer Informationen auf sehr primitive Art darstellt. Vielleicht haben Sie sich gefragt, wie eine so dumme Maschine in der Lage ist, vom Menschen entwickelte intelligente Programme auszuführen. Das ist in der Tat nicht einfach.

Zwischen dem Menschen und dem Computer gibt es enorme Verständigungsschwierigkeiten, da sich die menschliche Sprache und die Maschinensprache des Computers stark unterscheiden. Es hat einige Jahrzehnte gedauert, die Verständigungsschwierigkeiten halbwegs aus dem Weg zu räumen. Der Schlüssel dazu liegt in der Entwicklung geeigneter Programmiersprachen.

2.1.2 Definition

Programmiersprachen sind Sprachen, mit deren Hilfe ein Softwareentwickler Befehle (Rechenvorschriften) für den Computer formuliert. Eine bestimmte Ansammlung von Befehlen ergibt ein Computerprogramm. Die Befehle dieser Programmiersprachen sind nicht so leicht verständlich, wie es die natürliche Sprache für uns ist. Diese Sprachen können aber vom Menschen viel besser verstanden werden als der Binärcode des Computers. Programmiersprachen vermitteln also zwischen beiden Welten im Kreislauf zwischen Mensch und Maschine (→ Abbildung 2.1).

Damit Sie eine Programmiersprache wie Java verstehen, müssen Sie diese Sprache wie jede Fremdsprache erlernen. Der Computer hat es besser: Er muss die Fremdsprache Java nicht erlernen. Für ihn haben findige Softwareentwickler eine Art Dolmetscher (Interpreter, Compiler) erfunden. Dieser *Babelfish*[1] übersetzt die Java-Sprache in die Muttersprache des Computers (→ Kapitel 5 und 6).

1 Aus Douglas Adams, »Per Anhalter durch die Galaxis«: Ein Babelfish ist ein Fisch, den man sich ins Ohr steckt und der per Gedankenübertragung alle Sprachen übersetzt.

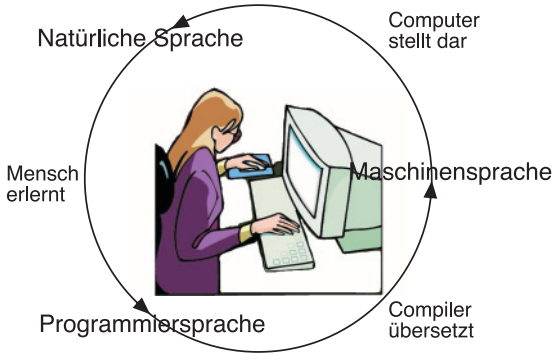


Abbildung 2.1 Kreislauf zwischen Mensch und Maschine

2.1.3 Klassifizierung

Es gibt verschiedene Möglichkeiten, Programmiersprachen einzuordnen: entweder nach Sprachmerkmalen (→ Abbildung 2.2) oder nach ihrer Abstammung (→ Abbildung 2.3) oder chronologisch (→ 2.1.4 Geschichte).

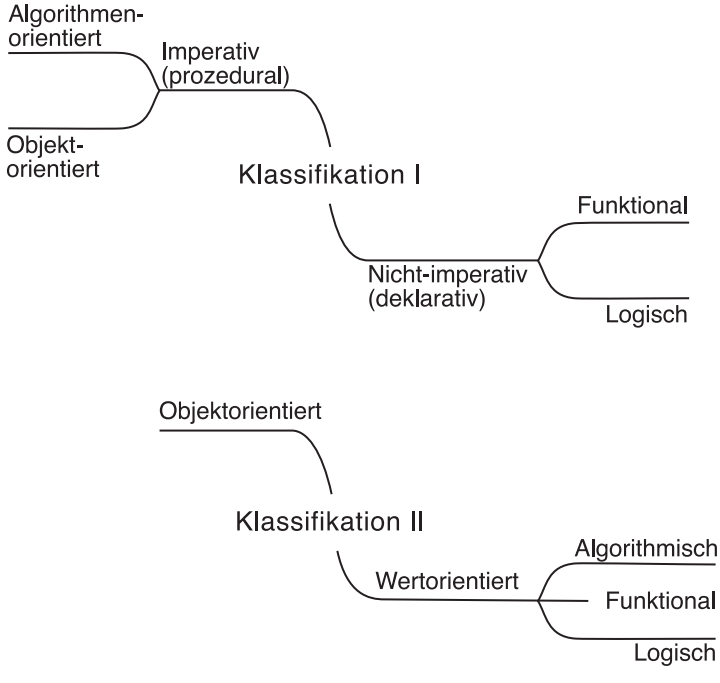


Abbildung 2.2 Klassifikation nach Rechenberg

Peter Rechenberg sagt, ein einziges Klassifikationsschema sei niemals ausreichend, und schlägt stattdessen gleich zwei verschiedene Schemata vor (→ Abbil-

dung 2.2). Dieses Kapitel gruppiert die Programmiersprachen chronologisch und beginnt daher mit ihrer Geschichte.

2.1.4 Geschichte

Programmiersprachen unterliegen einem steten Wandel. Ständig kommen neue Sprachen hinzu, alte verschwinden wieder. Der Grund für diese hektische Betriebsamkeit ist die Suche der Softwareentwickler nach *der* optimalen Programmiersprache.

Die Idealsprache ist extrem leicht zu erlernen, für jeden Einsatzbereich geeignet und beflügelt die Entwicklung hochwertiger, extrem schneller Software, die auf jedem Computersystem ausgeführt werden kann –, kurz: diese Sprache gibt es (noch) nicht.

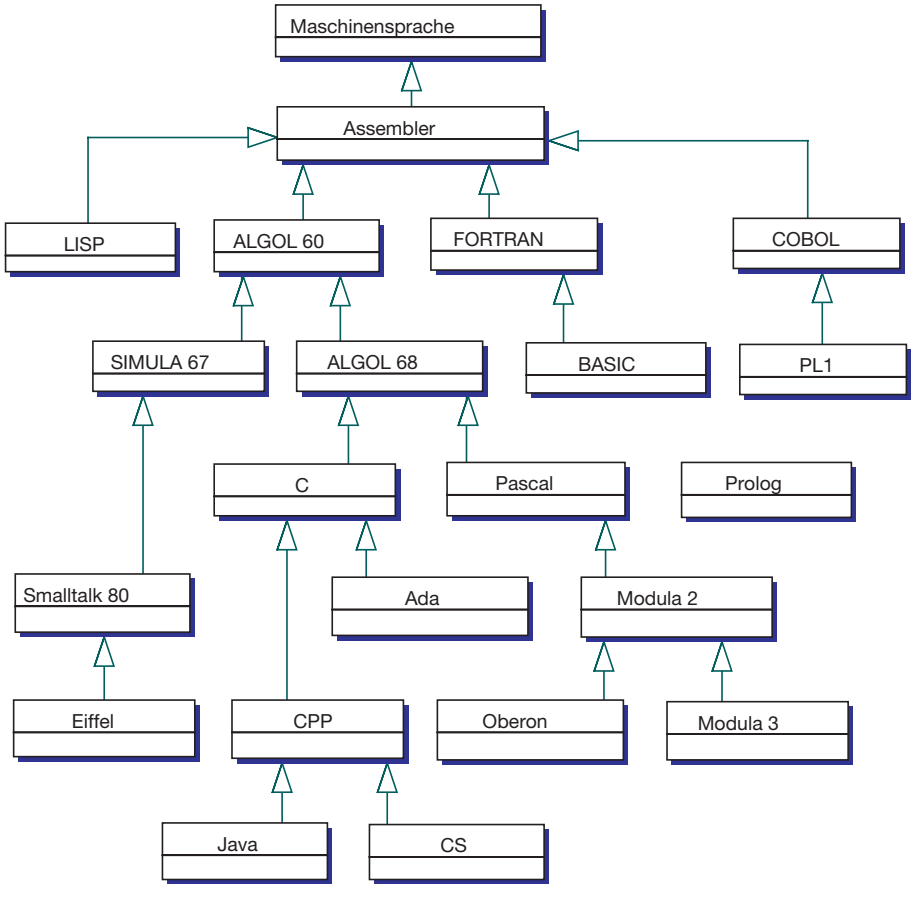


Abbildung 2.3 Stammbaum der wichtigsten Programmiersprachen

Auch wenn die optimale Programmiersprache noch nicht existiert, ist der bisher erzielte Fortschritt bei der Entwicklung neuer Programmiersprachen beachtlich. Ausgangspunkt dieser Entwicklung war die »Muttersprache« der Computer, die so genannte Maschinensprache (→ Abbildung 2.3).

Von der maschinennahen Programmierung hat man sich jedoch im Lauf der Zeit immer weiter entfernt. Ordnet man die Programmiersprachen chronologisch, so kommt man heute je nach Zählweise auf bis zu sechs Generationen von Programmiersprachen, die ich Ihnen vorstellen möchte.

2.2 Programmiersprachen der ersten Generation

Als die ersten Computer entwickelt wurden, programmierte man sie direkt in Maschinensprache. Die »Muttersprache« des Computers nennt sich Maschinensprache, weil sie vom Computer (der Maschine) direkt und ohne Übersetzung ausgeführt werden kann.

Die Maschinensprache ist von den natürlichen Sprachen (Humansprachen), mit denen sich Menschen verständigen, sehr verschieden. Sie besteht aus kleinen Codeeinheiten im *Binärformat*, den eingangs erwähnten Befehlen.

2.2.1 Programmaufbau

Wenn ein Maschinenprogramm abläuft, liest der Computer diese binären Zahlen und interpretiert sie als Befehle. Das Programm befindet sich während seiner Ausführung in einem bestimmten Bereich des Hauptspeichers (→ Kapitel 22). Somit kann jedem Befehl eine eindeutige Adresse zugeordnet werden. Das → Listing 2.1 zeigt auf der linken Seite die Adresse im Hauptspeicher in hexadezimaler Notation und auf der rechten Seite das eigentliche Maschinenprogramm in hexadezimaler Notation.

```
;CD/examples/ch01/ex01
(...)
XXXX:0100   B9  01  00
XXXX:0103   B8  00  00
XXXX:0106   01  C8
XXXX:0108   41
XXXX:0109   83  F9  05
XXXX:010C   76  F8
(...)
```

Listing 2.1 Ausschnitt aus einem Maschinenprogramm (Intel-80x86-CPU)

Verständlichkeit

Durch die hexadezimale Notation des Maschinenprogramms ist es einem Experten schon viel besser möglich, das Programm zu verstehen – besonders gut lesbar ist die Ansammlung mehr oder weniger verständlichen Anweisungen jedoch nicht.

Mikrobefehle

Das Maschinenprogramm besteht aus einer Reihe von Mikrobefehlen, auf die ich an dieser Stelle bewusst nicht weiter eingehen möchte. Die Erläuterung der Befehle folgt im nächsten Abschnitt, wo das gleiche Programm in der Assemblersprache vorgestellt wird.

Binärcode

Nur so viel an dieser Stelle: Was Sie im → Listing 2.1 sehen, ist der so genannte *Binärcode* eines Computerprogramms. Ein binäres Computerprogramm besteht aus Befehlen in der nativen (eigentlichen) Sprache des Computers. Wenn man ein Computerprogramm direkt in dieser Maschinensprache schreibt, muss es nicht mehr in die Sprache des Computers übersetzt werden.

2.2.2 Portabilität

Es ist wichtig zu wissen, dass sich Computer unterschiedlicher Bauart mehr oder weniger stark in ihrem Binärcode unterscheiden. Ein Maschinenprogramm für einen Apple Macintosh ist verschieden von einem Maschinenprogramm für einen Pentium-PC oder einem IBM-Großrechner. Durch diese Tatsache kann ein Maschinenprogramm, das für einen Pentium-PC entwickelt wurde, nicht direkt auf einem anderen Computersystem wie einem IBM-Großrechner ausgeführt werden.

Um das zu erreichen, also beispielsweise ein Maschinenprogramm, das für einen Pentium-PC entwickelt wurde, auf einem IBM-Großrechner ausführen zu können, muss es in die Maschinensprache des IBM-Großrechners übersetzt werden. Wenn dieser Vorgang Schritt für Schritt während der Ausführung des Programms abläuft, spricht man davon, dass das Programm interpretiert wird. Wird das Programm hingegen in einem einzigen Durchlauf übertragen (kompiliert) und erst danach ausgeführt, spricht man von Portierung (engl. portable: übertragbar).

Wenn Sie nochmals einen Blick auf → Listing 2.1 werfen, können Sie sich vorstellen, was es bedeutet, Tausende von derart simplen Instruktionen zu übertragen. Ein Entwickler, der diese Arbeit manuell durchführt, muss – neben unendlicher Geduld – über sehr gute Kenntnisse der Hardware *beider* Computersysteme verfügen.

In Maschinensprache geschriebene Computerprogramme lassen sich ab einer bestimmten Komplexität praktisch nicht mehr auf andere Computersysteme übertragen. Dies ist neben der schlechten Verständlichkeit einer der Hauptnachteile der Maschinensprache.

2.2.3 Ausführungsgeschwindigkeit

Direkt in Maschinensprache entwickelte Programme sind meistens sehr effizient programmiert. Sie laufen im Vergleich zu Programmen, die in Hochsprachen (Pascal, Java) programmiert sind, oftmals viel schneller, benötigen nur wenig Hauptspeicher und Festplattenkapazität. Entwickler von Maschinenprogrammen besitzen in der Regel sehr gute Hardwarekenntnisse und können daher den Programmcode und Speicherplatzbedarf der Programme stark optimieren.

2.2.4 Einsatzbereich

Direkt in Maschinensprache wird trotz ihres Geschwindigkeitsvorteils aufgrund ihrer Hardwareabhängigkeit und ihrer extrem schlechten Verständlichkeit heute kaum mehr programmiert. Wenn man überhaupt maschinennah programmiert, dann in Assembler, der Programmiersprache der zweiten Generation.

2.3 Programmiersprachen der zweiten Generation

Um den Computer nicht in der für Menschen schlecht verständlichen Maschinensprache programmieren zu müssen, hat man die Assembler-Sprache erfunden. In Assembler geschriebene Programme bestehen aus einzelnen symbolischen Anweisungen, die sich der Programmierer besser merken kann.

2.3.1 Programmaufbau

Beachten Sie bitte nur den rechten Teil des Programmbeispiels (→ Listing 2.2), der mit `MOV` beginnt. Der gesamte linke Bereich ist das entsprechende Maschinenprogramm aus → Listing 2.1, das ich zum besseren Vergleich nochmals eingefügt habe.

Verständlichkeit

Das Assembler-Programm besteht im Gegensatz zu den kryptischen Zahlencodes des Maschinenprogramms aus symbolischen Befehlen. Ein Computer verfügt über mindestens einen Hauptprozessor, die so genannte Central Processing Unit (CPU). Dieser Prozessor besitzt einen typspezifischen Befehlssatz und mehrere Register (→ Kapitel 22).

Mikrobefehle

Die Register dienen beim Ausführen des Programms als kurzfristiger Zwischenspeicher für Zahlenwerte, mit denen der Hauptprozessor beschäftigt ist: Sie besitzen daher die extrem wichtige Funktion eines Kurzzeitgedächtnisses für den Prozessor.

```
;CD/examples/ch01/ex02
(...)
XXXX:0100   B9  01  00   MOV   CX,  1
XXXX:0103   B8  00  00   MOV   AX,  0
XXXX:0106   01  C8       ADD   AX,  CX
XXXX:0108   41           INC   CX
XXXX:0109   83  F9  05   CMP   CX,  05
XXXX:010C   76  F8       JBE   106
(...)
```

Listing 2.2 Ein Assembler-Programm (Intel-80x86-CPU)

Der Anfang des Programms (→ Listing 2.2) wird von einem Kommentar markiert, der den Namen des Programms enthält. Danach lädt der Prozessor den Wert 1 in das Register `CX`. Der entsprechende Assemblerbefehl lautet `MOV` und ist eine Abkürzung von »to move« (bewegen). Jeder dieser Mikrobefehle ist ein solches Kürzel, das man sich leicht merken kann und deshalb auch Mnemonik (Stütze fürs Gedächtnis) nennt.

Die zweite Anweisung `MOV AX, 0` initialisiert das Akkumulatorregister mit dem Wert 0, um die nachfolgende Berechnung bei 0 zu beginnen. Im Anschluss daran erhöht der Befehl `INC CX` den Anfangswert um 1. Das Mnemonik lautet `INC` und bedeutet »increment« (Zunahme).

Nachfolgend vergleicht der Prozessor den Wert des Registers `CX` mit dem Wert 5 und springt zur Adresse `106`, wenn der Wert kleiner oder gleich 5 ist. Die beiden Befehle `CMP` und `JBE` bilden demnach eine Einheit. `CMP` bedeutet »to compare« (vergleichen) und `JBE` »jump below or equal« (springe, wenn kleiner oder gleich).

Binärcode

Was Sie in → Listing 2.2 sehen, ist der so genannte *Assemblercode* eines Computerprogramms. Damit der Computer diesen Programmtext (ASCII-Code) verstehen kann, muss er in ein binäres Computerprogramm (*Binärcode*) übersetzt werden. Dazu verwendet der Softwareentwickler ein spezielles Entwicklungswerkzeug, den so genannten Assembler. Der Assembler *fügt* das Programm *zusammen* (engl. to assemble: zusammenfügen, montieren). Daher bekam die Programmiersprache ihren Namen.

2.3.2 Portabilität

Ein Assembler-Programm von einem Computersystem auf ein anderes zu übertragen, ist ähnlich schwer wie die Portierung eines Maschinenprogramms. Meist ist es sinnvoller, sich die Dokumentation durchzulesen und das gesamte Programm neu zu schreiben.

Bedenken Sie, was die Hardwareabhängigkeit von Software bedeutet: Nicht nur, um ein Computerprogramm von einem Computertyp auf einen anderen zu übertragen, muss die Software verändert werden. Sie müsste eigentlich auch dann verändert werden, wenn ein neueres Modell des gleichen Computertyps erscheint, wenn dessen Maschinensprache umfangreicher geworden ist. Wenn Sie ein in Assembler geschriebenes Programm ausliefern, müssten Sie unterschiedliche Versionen für unterschiedliche Computertypen und -modelle produzieren.

Aus diesem Grund ist der Anteil der Assembler-Programmierung bei komplexen Projekten inzwischen unbedeutend. Es ist einfach unwirtschaftlich, in Assembler zu programmieren.

2.3.3 Ausführungsgeschwindigkeit

Aber egal, wie man zur hardwarenahen Programmierung steht: Da die Assemblersprache mit der Maschinensprache sehr verwandt ist, kann ein Assembler-Programm extrem leicht in effizienten Maschinencode umgesetzt werden. Es ist kompakt, benötigt also sehr wenig Festplattenspeicherplatz, beansprucht normalerweise wenig Hauptspeicher und kann bei geschickter Programmierung deutlich schneller ausgeführt werden als vergleichbare Hochsprachenprogramme.

2.3.4 Einsatzbereich

Sinnvolle Anwendungsbereiche der Assembler-Sprachen sind dort, wo extreme Anforderungen an die Ausführungsgeschwindigkeit und Kompaktheit des Codes auftreten, zum Beispiel bei Computerspielen, bei Gerätetreibern oder bei geschwindigkeitskritischen Betriebssystemteilen.

2.4 Programmiersprachen der dritten Generation

Da heute aufgrund der genannten Nachteile niemand mehr seinen Computer ausschließlich in Maschinen- oder Assembler-Sprache programmieren möchte, hat man eine Reihe von so genannten höheren Programmiersprachen entwickelt. Deren wichtigste Vertreter sind FORTRAN, COBOL, Algol, Pascal, BASIC, SIMULA, C, C++, Java und C#.

Die Programmiersprachen der dritten Generation stehen zwischen der unverständlichen, aber extrem effizienten Maschinensprache und der für den Menschen optimal verständlichen, aber aus Maschinensicht ineffizienten und unpräzisen natürlichen Sprache.

Der Übergang von der Assemblersprache zu den Programmiersprachen der dritten Generation kommt einem Quantensprung gleich. Die neue Generation unterstützt die Umsetzung von Algorithmen (→ Kapitel 9) viel besser als die Assemblersprachen und besitzt nicht deren extreme Hardwareabhängigkeit.

Obwohl es heute Sprachen der fünften Generation gibt, dominieren die Programmiersprachen der dritten Generation die Welt der Softwareentwicklung. Sie bieten einen guten Kompromiss zwischen der Flexibilität der Assemblersprache und der Mächtigkeit der Sprachen der fünften Generation.

2.4.1 Programmaufbau

Programme, die in einer höheren Programmiersprache geschrieben wurden, gleichen sich prinzipiell im Aufbau. Sie verfügen über eine Deklaration von Datenstrukturen, über Funktionen und Kontrollstrukturen. Ein Beispiel zeigt das Pascal-Programm in → Listing 2.3.

```
{  CD/examples/ch01/ex03  }
Program  Addition;
begin
    i  :=  0;
```

```
    while  (i  <=  5)
        i  =  i  +  1;
end.
```

Listing 2.3 Ein Pascal-Programm

Verständlichkeit

Das kleine Programm leistet das Gleiche wie das Assembler-Programm zuvor, ist aber sicher selbst von jedem Informatik-Laien weit besser zu verstehen. Das liegt zum Teil daran, dass sich die Pascal-Programmiersprache sehr an die Bezeichnungen der Mathematik anlehnt und natürliche Begriffe als Schlüsselwörter (`begin`, `end`, `while`) verwendet.

Makrobefehle

Wenn Sie die Assembler-Programme mit Pascal-Programmen vergleichen, stellen Sie fest, dass ein Pascal-Befehl in der Regel weit mächtiger ist als ein Assembler-Befehl. Mit anderen Worten: Sie müssen nicht so viel schreiben und kommen bei der Problemlösung schneller zum Ziel.

Binärcode

Damit der Computer den *Pascalcode* (ASCII-Text) ausführen kann, muss dieser in ein binäres Computerprogramm (*Binärcode*) übersetzt werden. In der Regel verwendet man dazu einen Compiler. Der Compiler *stellt* aus dem Pascal-Programm ein Maschinenprogramm *zusammen* (lat. compilare: zusammenraffen, plündern).

Die meisten Entwicklungsumgebungen für Sprachen der dritten Generation verwenden Compiler. Statt eines Compilers lässt sich aber auch ein Interpreter einsetzen, um das Programm Schritt für Schritt in Maschinensprache zu übertragen. Bei BASIC-Programmen war dies früher die Regel.

2.4.2 Portabilität

Neben der besseren Verständlichkeit und höheren Produktivität besitzt das vorliegende Programm gegenüber dem vorangehenden Assembler-Beispiel einen weiteren entscheidenden Vorteil: Es ist nicht hardwarespezifisch, sondern sehr leicht von einem Computersystem auf ein anderes zu übertragen.

Programmiersprachen der dritten Generation sind in der Regel unabhängig von der Hardware des Computers. Es gibt zwischen Sprachen wie C++ und Java jedoch einige Unterschiede: Bei den einen Sprachen ist nur der Quelltext portabel, bei anderen hingegen auch der Binärcode.

Java beispielsweise gehört zu den Programmiersprachen, deren Programme am leichtesten portierbar sind. Der vom Java-Compiler erzeugte Binärcode (Bytecode: → Kapitel 6, 6.2 Bytecode) kann bei Einhaltung von bestimmten Programmierregeln unverändert sowohl auf einem Macintosh als auch auf einem IBM-Großrechner ausgeführt werden.

2.4.3 Ausführungsgeschwindigkeit

Es gibt heute sehr leistungsfähige Compiler, die aus einem Hochsprachenprogramm ein schnelles Maschinenprogramm erzeugen. Trotzdem ist es im Regelfall so, dass ein optimales, in Assembler geschriebenes Programm schneller ausgeführt wird als ein optimales Hochsprachenprogramm. Dieser Unterschied rechtfertigt jedoch in den meisten Fällen nicht den Einsatz der Assembler-Sprache.

2.4.4 Einsatzbereich

Programmiersprachen der dritten Generation sind Allzweckprogrammiersprachen, die auf allen Gebieten der Softwareentwicklung verwendet werden. Mittlerweile verdrängen sie die Assembler-Sprache sogar auf dem Gebiet der Treiberprogrammierung.

2.5 Programmiersprachen der vierten Generation

Mit den Programmiersprachen der vierten Generation versuchten die Entwickler Probleme wie den Datenbankzugriff in einer abstrakteren Art und Weise zu lösen als die Sprachen der dritten Generation. Ein Beispiel für eine Programmiersprache dieser Generation ist Natural.

2.5.1 Programmaufbau

Natural-Programme bestehen wie Programme, die mit Sprachen der dritten Generation geschrieben wurden, aus Datendeklarationen, Kontrollstrukturen und Funktionen. Ein Beispiel sehen Sie in Listing 2.4.

```
*  CD/examples/ch01/ex04
PGM-ID:  Addition
DEFINE  DATA
  LOCAL
      01  #i
END-DEFINE
```

```
*  ----------------
FOR  #i  1  TO  5  STEP  1
END-FOR
```

Listing 2.4 Ein Natural-Programmbeispiel

Verständlichkeit

Wenn Sie dieses Beispiel mit dem eingangs gezeigten Assembler-Listing vergleichen, erkennen Sie ebenfalls, dass es erheblich besser zu verstehen ist.

Makrobefehle

Natural-Programme verfügen über Befehle, die im Vergleich zu Sprachen der dritten Generation in der Regel weit mächtiger sind. Die Sprache ist – im Vergleich zu C++ oder Java – weit weniger flexibel, aber in bestimmten Einsatzbereichen sicher annähernd so produktiv.

Binärcode

Um *Naturalcode* in ein binäres Computerprogramm (*Binärcode*) zu übersetzen, kommt entweder ein Compiler oder ein Interpreter zum Einsatz.

2.5.2 Portabilität

Wie das Pascal-Beispiel ist auch das Natural-Programm im Vergleich zu Assembler-Programmen leicht portierbar, da es keine direkten Abhängigkeiten zu der zugrunde liegenden Hardware besitzt.

2.5.3 Ausführungsgeschwindigkeit

Mir sind hier keine vergleichenden Studien bekannt, die auf Unterschiede in der Ausführungsgeschwindigkeit zwischen Natural- und Assembler-Programmen hinweisen. Da Programmiersprachen wie Natural vor allem in Zusammenhang mit Datenbankprogrammierung eingesetzt werden, kann man davon ausgehen, dass die Ausführungsgeschwindigkeit zufrieden stellend ist.

2.5.4 Einsatzbereich

Programmiersprachen wie Natural haben ihren Haupteinsatzbereich in der Datenbankprogrammierung.

2.6 Programmiersprachen der fünften Generation

Noch weiter entfernt von der Maschinensprache als Natural sind Programmiersprachen der fünften Generation. Sie wurden konzipiert, um Expertensysteme zu entwickeln. Programmiersprachen dieser Generation nennt man auch logische Programmiersprachen. Prominentester Vertreter dieser Gattung ist neben Datalog die Sprache Prolog (PROgramming in LOGic).

2.6.1 Programmaufbau

Ein Prolog-Programm besteht aus einer Reihe von Funktionen, deren Reihenfolge egal ist. Die Funktionen bestehen aus Sätzen (*clauses*). Bei diesen ist die Reihenfolge sehr wichtig. Es gibt zwei Typen von clauses: Tatsachen (*facts*) und Regeln (*rules*). Ein Beispiel sehen Sie in Listing 2.5.

```
% CD/examples/ch01/ex05
man (peter). % peter = man
% peter = parent, paul = child (parent <-> child):
parent (peter, paul).
% Every man who is a child's parent is also its
father:
father (FA, CH):-man (FA),parent (FA, CH).
% Who is paul's father?
?-father(X, paul).
```

Listing 2.5 Ein Prolog-Programmbeispiel

Verständlichkeit

Prolog ist für Programmierer von mathematisch geprägten Sprachen wie Pascal sehr ungewohnt. Man kann jedoch leicht erkennen, dass die Sprache ideal ist, um Software zu entwickeln, die logische Probleme lösen soll (Expertensysteme).

Makrobefehle

Die Befehle von Prolog-Programmen sind sehr mächtig. Wie Natural ist die Sprache nicht sehr flexibel, aber in einem bestimmten Nischenbereich ist der Entwickler damit sehr produktiv.

Binärcode

Auch *Prologcode* muss wieder in ein binäres Computerprogramm (*Binärcode*) übersetzt werden. Dazu verwendet man entweder einen Compiler oder einen Interpreter.

2.6.2 Portabilität

Prolog ist wegen ihres starken Abstraktionsgrades von der Hardware wie andere höhere Programmiersprachen prinzipiell leicht zu portieren. Es gibt eine Vielzahl von Compilern für die unterschiedlichsten Computertypen.

2.6.3 Ausführungsgeschwindigkeit

Mir sind keine vergleichenden Studien zwischen in Prolog geschriebenen Programmen und Assembler-Programmen bekannt. Es soll jedoch optimierende Compiler geben, die hocheffizienten Maschinencode erzeugen können.

2.6.4 Einsatzbereich

Logische Programmiersprachen wie Prolog werden vorzugsweise zur Programmierung von Expertensystemen eingesetzt. Expertensysteme sind Programme, die auf künstlicher Intelligenz (KI) aufbauen und logische Schlussfolgerungen ziehen können.

2.7 Programmiersprachen der sechsten Generation

Ähnlich wie der Übergang von der hardwarenahen Assembler-Sprache zu den Hochsprachen wird auch der Übergang zu den Sprachen der sechsten Generation eine Zäsur darstellen: Diese Sprachen beschreiben ein Computerprogramm nicht wie gewohnt durch Text, sondern in Form von Grafiken (zum Beispiel Ablaufdiagrammen). Mit der Version 2.0 der *Unified Modeling Language* (UML) in Verbindung mit der *Model Driven Architecture (*MDA) entwickelt sich zurzeit eine Programmiersprache der sechsten Generation.

2.7.1 Programmaufbau

Von der grafischen Programmiersprache UML kann ich Ihnen kein Listing präsentieren, da die Programme nur grafisch beschrieben werden. Ein Beispiel für einen Ausschnitt eines Programms zeigt → Abbildung 2.4.

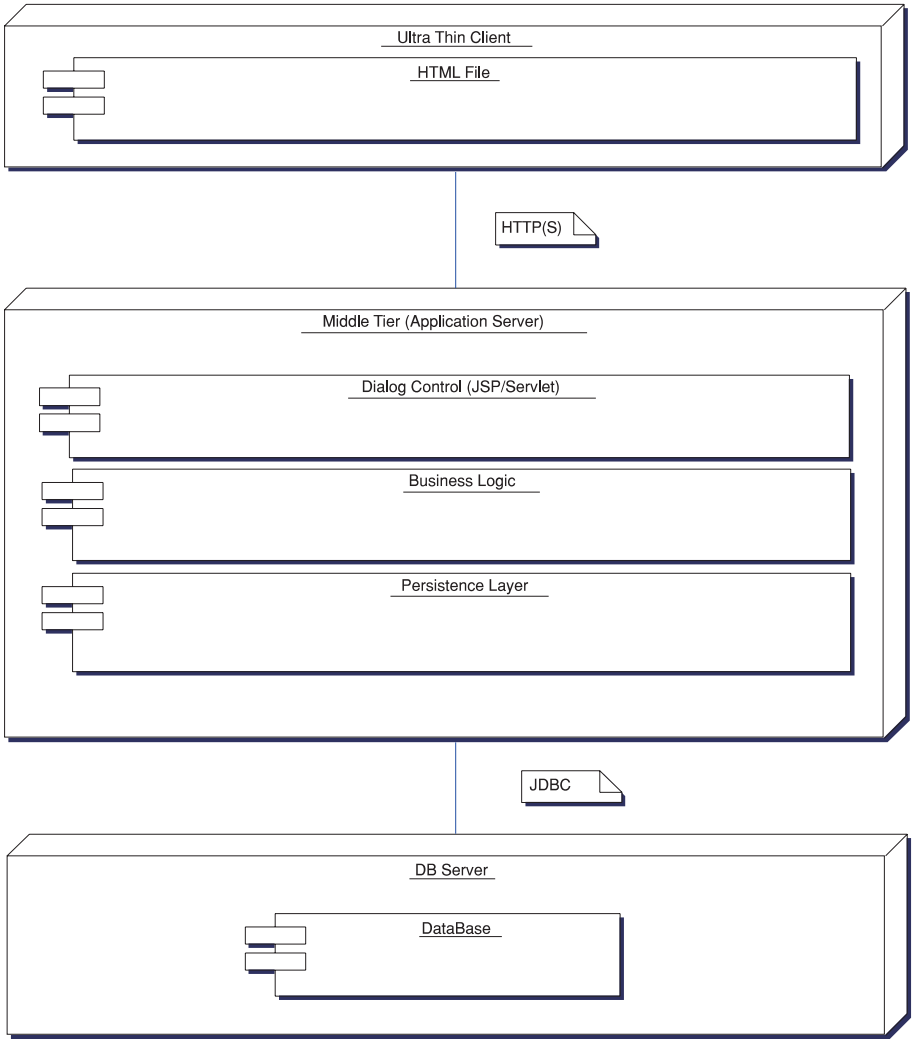


Abbildung 2.4 Eine UML-Beispielgrafik (Verteilungsdiagramm)

Verständlichkeit

Wie Sie aus der Abbildung entnehmen können, ist hier der prinzipielle Aufbau einer Internet-Anwendung skizziert. Die Abbildung zeigt allerdings nicht das vollständige Programm, sondern nur einen Teil davon. Um ein Programm mit der UML vollständig zu beschreiben, ist eine Vielzahl von solchen Graphiken notwendig. Diese sind jedoch sehr gut verständlich.

Graphiken

In Sprachen der sechsten Generation wird es keine Befehle mehr geben, sondern nur eine Vielzahl von Graphiken, die das Modell der Software bilden. Aus diesen Modellen muss ein Generator Programmcode erzeugen. Dieser Programmcode ist in einer Sprache der dritten Generation verfasst.

Binärcode

Der erzeugte *Programmcode* muss letztendlich wieder – direkt oder indirekt – in ein Binärprogramm (*Binärcode*) übersetzt werden. Dazu wird man entweder einen Compiler oder einen Interpreter verwenden.

2.7.2 Portabilität

Ein Programm, das nach der Model Driven Architecture entwickelt wurde, soll unabhängig von der Hardware sein und daher leicht von einem Computertyp auf einen anderen übertragen werden können.

2.7.3 Ausführungsgeschwindigkeit

Da aus den Modellen beispielsweise C#- oder Java-Code erzeugt wird, verhält es sich mit der Ausführungsgeschwindigkeit wie bei Programmiersprachen der dritten Generation. Sie ist abhängig vom Geschick des Entwicklers und von der Qualität des Compilers oder Interpreters.

2.7.4 Einsatzbereich

Programmiersprachen der sechsten Generation sind gerade in der Entstehung. Ihr Einsatz ist bis auf weiteres Zukunftsmusik.

2.8 Zusammenfassung

Programmiersprachen haben sich aus der nativen Sprache der Computer (Maschinensprache) entwickelt. Sie entfernen sich von der hardwarenahen Programmierung (erste und zweite Generation) und entwickeln sich in Richtung der natürlichen Sprache (Humansprache).

Programmiersprachen der ersten und zweiten Generation eignen sich für hardwarenahe Programme, während Programmiersprachen der vierten und fünften Generation für spezielle Anwendungsfälle, wie Datenbankprogrammierung und Expertensysteme, geeignet sind. Programmiersprachen der dritten Generation beherrschen heute die Programmentwicklung. Zu ihnen gehört auch die Programmiersprache Java.

2.9 Aufgaben

Versuchen Sie folgende Aufgaben zu lösen:

2.9.1 Programmiersprachen der ersten Generation

1. Wie nennen sich Programmiersprachen der ersten Generation?
2. Woher stammt ihr Name?
3. Weshalb programmiert man heute nicht mehr mit Sprachen der ersten Generation?

2.9.2 Programmiersprachen der zweiten Generation

1. Was sind die drei wichtigsten Vorteile der Assembler-Sprache gegenüber Hochsprachen?
2. Für welche Software setzt man heute noch die Assembler-Sprache ein?
3. Was sind die drei wesentlichen Vorteile von Hochsprachen gegenüber der Assembler-Sprache?

2.9.3 Programmiersprachen der dritten Generation

1. Was versteht man unter portablen Computerprogrammen?
2. Nennen Sie drei Programmiersprachen der dritten Generation.

Die Lösungen zu den Aufgaben finden Sie in → Teil IV, Kapitel 17, ab Seite 441.

3 Objektorientierte Programmierung

»Ich sehe ein Pferd, dann sehe ich noch ein Pferd – dann noch eins. Die Pferde sind nicht ganz gleich, aber es gibt etwas, das allen Pferden gemeinsam ist, und das, was allen Pferden gemeinsam ist, ist die 'Form' des Pferds. Was unterschiedlich oder individuell ist, gehört zum 'Stoff' des Pferds.« (Jostein Gaarder)

3.1 Einleitung

Mitte der 60er Jahre des letzten Jahrhunderts kam es zu einer Softwarekrise. Die Anforderungen an Programme stiegen, die Software wurde dadurch komplexer und fehlerhafter. Auf Kongressen diskutierten Experten die Ursachen der Krise und die Gründe für die gestiegene Fehlerrate. Ein Teil der Softwareexperten kam zu dem Schluss, dass die Softwarekrise nicht mit den herkömmlichen Programmiersprachen zu bewältigen sei. Sie begannen deshalb, eine Generation von neuen Programmiersprachen zu entwickeln.

Die Entwickler dieser Sprachen kritisierten an den herkömmlichen Programmiersprachen vor allem, dass sich die natürliche Welt bisher nur unzureichend abbilden lasse. Um dem zu entgehen, gingen sie von natürlichen Begriffen aus, wie sie die Formenlehre der klassischen griechischen Philosophie geprägt hat, und wandelten sie für die Programmierung ab. Da sich alles um den Begriff des Objekts dreht, nannten sie die neue Generation von Sprachen »objektorientiert«.

3.1.1 Grundbegriffe

Alan Kay, einer der Erfinder der Sprache Smalltalk, hat die Grundbegriffe der objektorientierten Programmierung folgendermaßen zusammengefasst:

- Alles ist ein Objekt.
- Objekte kommunizieren durch Nachrichtenaustausch.
- Objekte haben ihren eigenen Speicher.
- Jedes Objekt ist ein Exemplar einer Klasse.
- Die Klasse modelliert das gemeinsame Verhalten ihrer Objekte.
- Ein Programm wird ausgeführt, indem dem ersten Objekt die Kontrolle übergeben und der Rest als dessen Nachricht behandelt wird.

3.1.2 Prinzipien

Neben diesen Grundbegriffen sind folgende Prinzipien wichtig:

- Abstraktion
- Vererbung
- Kapselung
- Beziehungen
- Persistenz
- Polymorphie

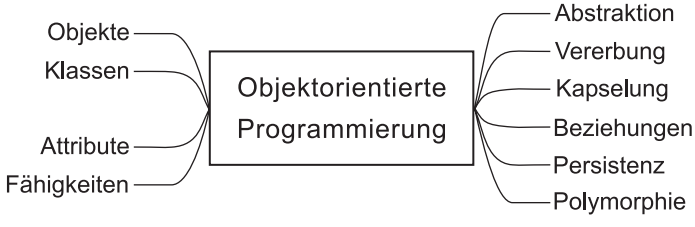


Abbildung 3.1 Hauptbegriffe der objektorientierten Programmierung

3.2 Objekte

Objekte sind für ein Java-Programm, was Zellen für einen Organismus bedeuten: Aus diesen Elementarteilchen setzt sich eine Anwendung zusammen. Objekte haben eine bestimmte Gestalt (Aussehen, Attribute, Kennzeichen) und bestimmte Fähigkeiten (Methoden, Funktionen). Gestalt und Fähigkeiten eines Objekts werden durch seine Erbinformationen bestimmt. Diese Erbinformationen sind der Bauplan, nach dem das Objekt erzeugt wird. In der objektorientierten Programmierung ist der Bauplan eines Objekts seine → Klasse.

Wenn Sie eine Reihe von Pferden betrachten, fällt Ihnen auf, dass ihnen die prinzipielle *Gestalt* gemeinsam ist. Ihre Unterschiede sind die *Attribute*, die das einzelne Pferd kennzeichnen. Ein Objekt der Klasse *Pferd* ist beispielsweise ein Pferd mit dem Namen *Xanthos*, ein anderes Pferd heißt *Balios*. Beide *Exemplare*[1] haben einen ähnlichen Körperbau (Gestalt) und ähnliche Fähigkeiten. Sie können beispielsweise beide laufen und wiehern.

1 Exemplar und Objekt sind gleichbedeutend. Im Gegensatz dazu ist der Begriff Instanz eine Fehlübersetzung (engl. Instance: Exemplar) und taucht in diesem Buch deshalb nicht auf.

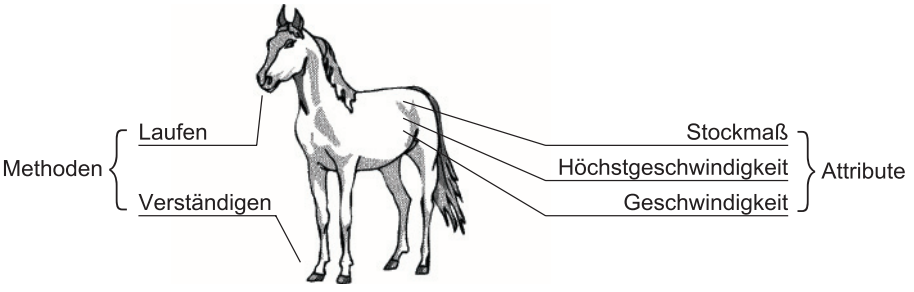


Abbildung 3.2 Jedes Objekt besitzt sein bestimmtes Verhalten und Aussehen.

Die beiden Objekte weisen aber auch einige deutliche Unterschiede auf: *Xanthos* ist weiß und *Balios* braun und *Xanthos* kann schneller laufen als *Balios*. Obwohl *Xanthos* und *Balios* zur gleichen Klasse gehören, sind nur ihre *prinzipiellen* Fähigkeiten identisch, nicht aber ihre *individuellen* Attribute. Was das bedeutet, wird im nächsten Abschnitt deutlich.

3.3 Klassen

Xanthos und *Balios* gehören zu der Klasse *Pferd*. Die Klasse ist es, welche die prinzipielle Gestalt und die Fähigkeiten der beiden Pferde-Objekte festlegt. Man bezeichnet Klassen daher als

- *Bauplan* für Objekte *oder*
- *Oberbegriff* für verschiedene Objekte (Klassifizierung) *oder*
- *Schablone* für verschiedene Objekte.

3.3.1 Attribute

Die eingangs erwähnte Klasse *Pferd* soll die Attribute *Stockmaß (Größe)*, *Höchstgeschwindigkeit* und *Geschwindigkeit* besitzen. Wenn aus dieser Klasse neue Objekte (Exemplare) entstehen, besitzen *alle* ein bestimmtes *Stockmaß*, eine bestimmte *Höchstgeschwindigkeit* und eine bestimmte *Geschwindigkeit* – aber welche Werte haben diese Attribute? Sie werden erst beim Entstehen (Erzeugen) des Objekts mit den individuellen Werten belegt.

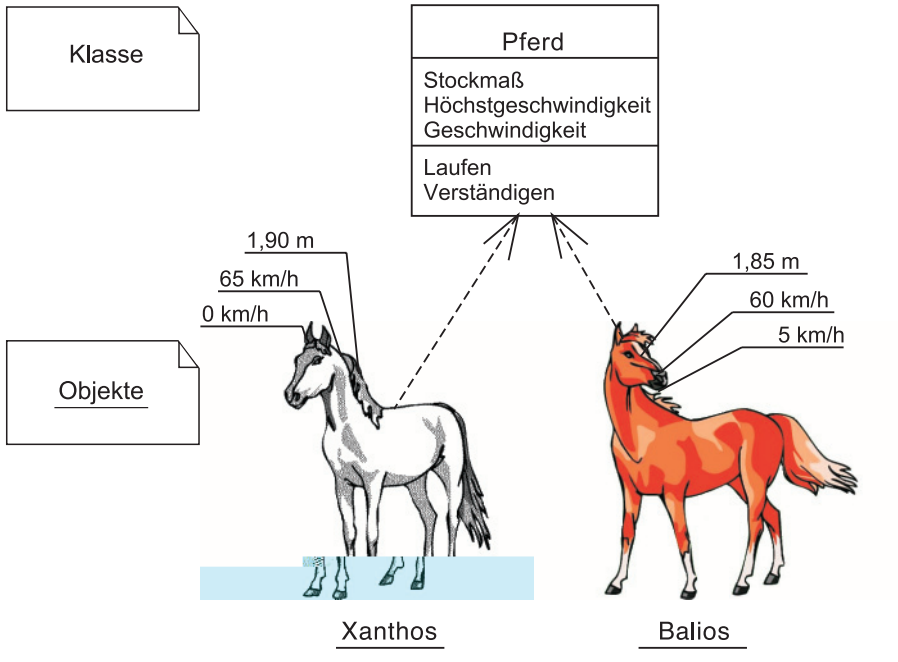


Abbildung 3.3 Die Klasse «Pferd« liefert den Bauplan für Pferde-Objekte.

Konstanten

Beispielsweise soll Xanthos über folgende Attribute verfügen: Stockmaß 1,90 m, Höchstgeschwindigkeit 65 km/h, Geschwindigkeit 0 km/h. Balios hingegen soll 1,85 m groß sein, maximal 60 km/h schnell laufen können und momentan gerade 5 km/h laufen. Obwohl beide Pferde nach dem gleichen Bauplan erzeugt worden sind, sind zwei deutlich unterschiedliche Objekte entstanden: Beide besitzen ein Stockmaß, sind aber unterschiedlich groß, beide können laufen, aber unterschiedlich schnell, beide besitzen eine Geschwindigkeit, aber ein Pferd steht und das andere bewegt sich langsam.

Zustände

Es ist Ihnen vielleicht aufgefallen, dass bei den bisherigen Attributen der beiden Pferde einige mit festen Werten belegt waren, andere hingegen mit veränderlichen Werten. Die flexiblen Attribute beschreiben den Zustand des Objekts. Zum Beispiel beschreibt die *Geschwindigkeit*, wie schnell sich Xanthos zur Zeit bewegt. Der Zustand eines Objekts kann sich im Lauf der Zeit ändern.

Kennungen

Was würde passieren, wenn man Xanthos und Balios so erzeugen würde, dass sie das gleiche *Stockmaß*, die gleiche *Höchstgeschwindigkeit* und die gleiche momentane *Geschwindigkeit* besitzen? Wie könnte man sie dann unterscheiden? In diesem Fall haben beide Objekte zwar individuelle Werte für ihre Attribute bekommen, aber diese sind zufällig gleich. Damit gleichen sich auch die Objekte in einem Programm wie eineiige Zwillinge.

Um die Pferde zu unterscheiden, benötigt man so etwas wie einen genetischen Fingerabdruck. In der Programmierung vergibt der Entwickler eine so genannte Kennung. Diese Kennung ist ein zusätzliches Attribut, bei dem darauf geachtet wird, dass es *eindeutig* ist. Objekte der *gleichen Klasse* besitzen also die gleichen Attribute, aber mit individuellen Werten. Erst die Kennung eines Objekts sorgt dafür, dass das Programm unterschiedliche Exemplare auch dann unterscheiden kann, wenn ihre Attribute zufällig die gleichen Werte besitzen.

3.3.2 Methoden

Angenommen, Sie wollen dem Objekt *Balios* mitteilen, dass es nun springen soll. Im wirklichen Leben geben Sie ihm dazu ein Zeichen. In der objektorientierten Programmierung müssen Sie stattdessen eine Methode des Objekts *Xanthos* aufrufen. Statt Methode werden Sie auch öfter auf die Begriffe Botschaft (Smalltalk), Nachricht oder Operation stoßen, die das Gleiche bedeuten sollen.

Abbildung 3.4 Objekte verständigen sich durch den Austausch von Nachrichten.

Egal, wie der Begriff nun bei den verschiedenen Programmiersprachen und in der Literatur genannt wird, eines ist gleich: Verhaltensweisen wie *Laufen* und *Verständigen* bestimmen die Fähigkeit eines Objekts zu kommunizieren und Aufgaben zu erledigen. Objekte verständigen sich also über Methoden.

Es existiert nicht nur eine Art von Methoden, sondern folgende fünf Grundtypen: *Konstruktoren* (»Erbauer«), *Destruktoren* (»Zerstörer«), *Mutatoren* (»Veränderer«), *Accessoren* (»Zugriffsmethoden«) und *Funktionen* (»Tätigkeiten«).

Konstruktoren

Die wichtigsten Methoden sind die, die ein Objekt erzeugen. Sie werden demzufolge auch Konstruktoren genannt, denn sie konstruieren, das heißt erschaffen ein Objekt.

Destruktoren

Methoden, die ein Objekt zerstören, nennen sich in der objektorientierten Programmierung Destruktoren. In Programmiersprachen wie C++ können Sie diese Destruktoren auch aufrufen und damit unmittelbar ein Objekt zerstören. In Java hat man hingegen aus Sicherheitsgründen darauf verzichtet, Destruktoren direkt aufzurufen. Hier wird ein Objekt automatisch zerstört, wenn es nicht mehr benötigt wird.

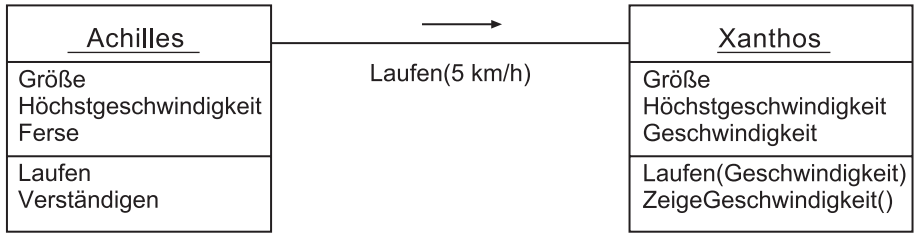


Abbildung 3.5 Die Methode »Laufen« verändert den Zustand von Xanthos

Mutatoren

Methoden, die den Wert eines Attributs verändern, nennen sich Mutatoren. Sie verändern den Zustand des Objekts. Mit einer solchen Methode könnten Sie zum Beispiel die Geschwindigkeit des Pferds *Xanthos* ändern (→ Abbildung 3.5). Diese Methode verfügt über einen so genannten Parameter, der den neuen Zustand (Geschwindigkeit des Pferds) vorgibt.

Accessoren

Accessoren sind Zugriffsmethoden, die nur ein bestimmtes Attribut abfragen, ohne etwas am Zustand des Objekts zu ändern. Eine solche Methode wäre zum Beispiel die Abfrage der momentanen Geschwindigkeit des Pferds *Xanthos* (→ Abbildung 3.6). Diese Methode besitzt einen so genannten Rückgabewert, die aktuelle Geschwindigkeit, mit der sich *Xanthos* fortbewegt.

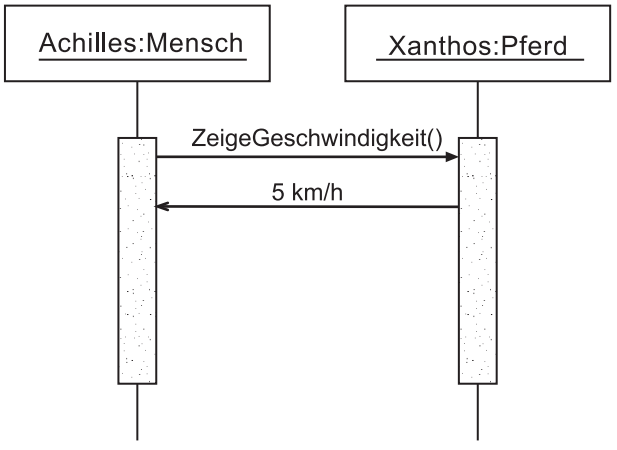


Abbildung 3.6 Die Methode »ZeigeGeschwindigkeit()« gibt den Zustand (die momentane Geschwindigkeit) von Xanthos zurück.

Funktionen

Methoden, die zum Beispiel nur eine Rechenoperation durchführen, werden häufig auch Funktionen genannt. Sie dürfen nicht verwechselt werden mit den Funktionen der klassischen Programmiersprachen, denn es bestehen erhebliche Unterschiede: So werden auch diese Methoden von Klasse zu Klasse vererbt (→ 3.5 Vererbung).

3.4 Abstraktion

Vielleicht werden Sie jetzt sagen: »Das ist doch alles Unsinn. Die Fähigkeiten und Attribute eines Pferds sind viel komplexer und können nicht auf Größe und Farbe, auf Laufen und Wiehern reduziert werden.« Das ist in der natürlichen Welt richtig, aber in der künstlichen Welt der Softwareentwicklung in der Regel völlig falsch.

Richtig wäre es nur dann, wenn man die Natur in einem Programm vollständig abbilden müsste. Aber für so eine übertriebene Genauigkeit gibt es bei der Programmierung selten einen Grund. Die objektorientierte Programmierung erleichtert eine möglichst natürliche Abbildung der realen Welt und fördert damit gutes Softwaredesign.

Sie verführt damit auch zu übertriebenen Konstruktionen. Die Kunst besteht darin, dem entgegen zu steuern und die Wirklichkeit so genau wie nötig, aber so einfach wie möglich abzubilden. Wie Sie später bei größeren Beispielprogrammen sehen werden, bereitet gerade die Analyse der für das Programm wesentlichen und richtigen Bestandteile unter Umständen große Probleme.

Wenn man innerhalb eines Programms nur die für die Funktionalität wesentlichen Teile programmiert, dann hat das praktische Gründe: Das Programm lässt sich schneller entwickeln, es wird billiger und schlanker. Somit benötigt es weniger Speicherplatz und es wird in der Regel schneller ablaufen als ein Programm, das mit unnötigen Informationen überfrachtet ist.

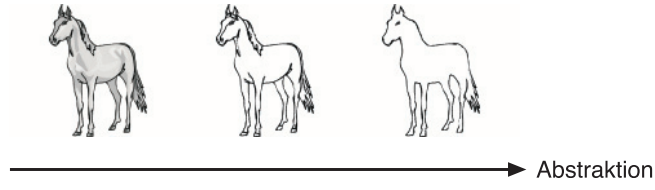


Abbildung 3.7 Durch Abstraktion erhält man das Wesentliche einer Klasse.

Um diese Kompaktheit zu erreichen, ist es notwendig, die meist extrem komplizierten natürlichen Objekte und deren Beziehungen so weit es geht zu abstrahieren, also zu vereinfachen. Der Fachbegriff für diese Technik nennt sich demzufolge auch *Abstraktion* (→ Abbildung 3.7).

3.5 Vererbung

Nach der Einführung von Klassen, Objekten, Methoden und Attributen ist es an der Zeit, diese neuen Begriffe in den Zusammenhang mit dem Begriff der *Vererbung* zu stellen. Vererbung gestattet es, Verhalten zwischen Klassen und damit auch zwischen Objekten mit Hilfe des Bauplans zu übertragen.

Ein Beispiel: Pferd und Zebra sind eng verwandt (→ Abbildung 3.8), in mancherlei Hinsicht aber doch sehr verschieden. Diese Unterschiede sind von anderer Güte als die Unterschiede zwischen zwei Pferden: Pferde und Zebras haben eine deutlich unterschiedliche Gestalt.

Dass Zebras im Sinne der Formenlehre eine andere Gestalt besitzen als Pferde, wird deutlich, wenn Sie überlegen, welche Farbe man einem Zebra zuordnen müsste: Schwarz oder Weiß? Zebras haben alle verschiedene Muster. Die Muster unterscheiden sich wie Fingerabdrücke beim Menschen. Das Muster des Fells ist eines der Merkmale, die ein Zebra von einem Pferd unterscheiden (es gibt noch andere).

Es ist also in den Fällen, in denen es auf die Unterschiede zwischen Farbe und Muster ankommt, immer besser, einem Zebra die Eigenschaft *Muster* zu geben und es einer anderen Klasse zuzuordnen (→ Abbildung 3.9). In allen anderen Fällen genügt eine gemeinsame Klasse.

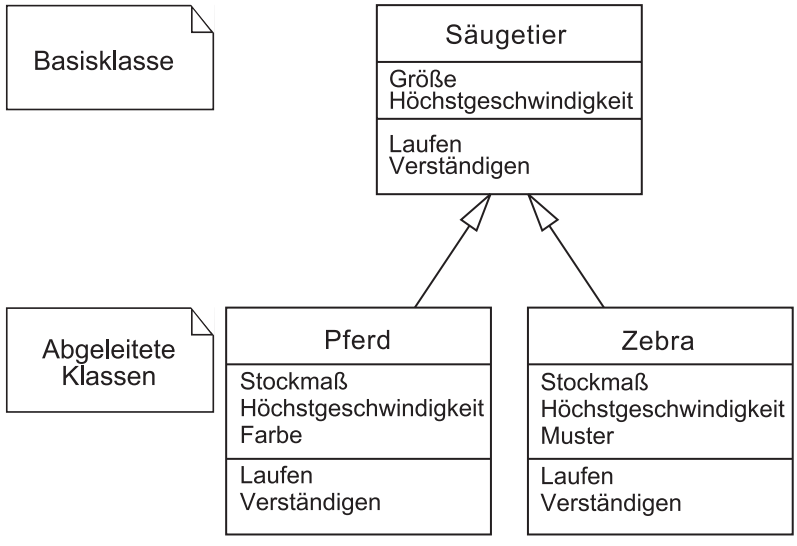


Abbildung 3.9 Die Basisklasse überträgt Basis-Eigenschaften und -Verhalten.

3.5.2 Abgeleitete Klassen

Angenommen, Sie möchten gern eine neue Klasse namens *Wal* auf Basis der Klasse *Säugetier* erzeugen. In der objektorientierten Programmierung spricht man davon, von *Säugetier* eine neue Klasse namens *Wal abzuleiten*.

Die neue Klasse erbt wie die Klassen *Pferd* und *Zebra* die Attribute *Größe* und *Höchstgeschwindigkeit* sowie die Methoden *Laufen* und *Verständigen* – Moment mal: *Laufen*? Fast alle Säugetiere können laufen, Wale jedoch nicht. Hier ist genau das passiert, was tagtäglich zu den Problemen der objektorientierten Programmierung gehört: Die Funktionalität der Basisklasse ist nicht ausreichend analysiert worden. Platt gesagt: Hier liegt ein Designfehler vor.

3.5.3 Designfehler

Sie können sich vielleicht vorstellen, dass es sehr unangenehm ist, wenn die Basisklasse aufgrund eines Designfehlers geändert werden muss. Die Änderungen pflanzen sich lawinenartig in alle Programmteile fort, in denen Objekte des Typs *Pferd* und *Zebra* mit der Methode *Laufen* verwendet wurden. Überall dort, wo die Methode *Laufen* der Klasse *Säugetier* verwendet wurde, muss sie durch die Methode *Fortbewegen* ersetzt werden.

Im Fall von Designfehlern stellt sich die Technik der Vererbung als großer Nachteil heraus. Vererbung hat neben diesem Manko auch den Nachteil, dass sich nicht nur Designfehler, sondern alle anderen vorzüglich gestalteten, aber unerwünsch-

ten Teile der Basisklasse in die abgeleiteten Klassen in Form von Ballast übertragen: Die Nachkommen solcher übergewichtiger Klassen werden immer fetter und fetter. Daher sollten Sie Vererbung stets kritisch sehen, sparsam einsetzen und wirklich nur dort verwenden, wo sie sinnvoll ist.

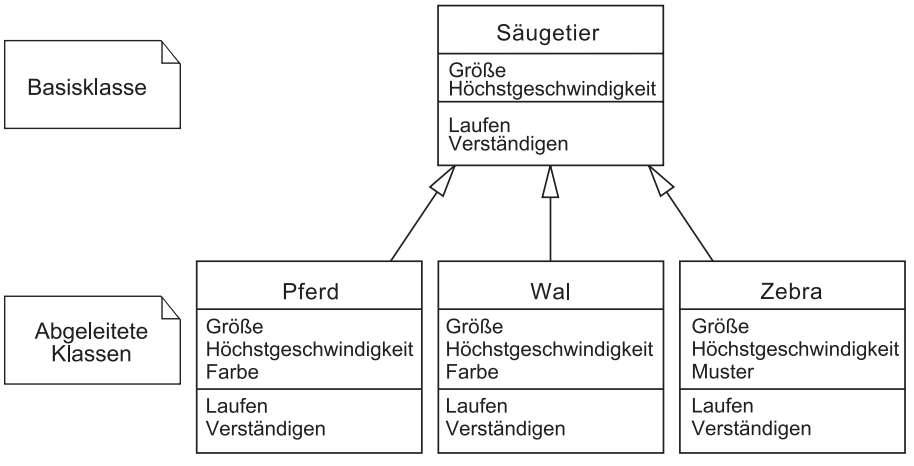


Abbildung 3.10 Durch Vererbung vererben sich auch Designfehler.

3.5.4 Umstrukturierung

Aber zurück zu den Designfehlern. Wie geht man mit Fehlern dieser Art um? Sie sind trotz der Vererbung heute kein so großes Problem mehr wie noch vor ein paar Jahren. Es gibt mittlerweile moderne Softwareentwicklungswerkzeugen, mit denen es relativ leicht ist, die notwendige Umstrukturierung vorzunehmen. Allerdings sollten Sie Software möglichst nur während der Analyse- und Designphase der Software (→ Kapitel 5) umstrukturieren. Als Regel gilt: Je später Änderungen vorgenommen werden, desto höher ist der damit verbundene Aufwand.

3.5.5 Mehrfachvererbung

In der Natur ist sie üblich, in Java und Smalltalk nicht erlaubt: die Mehrfachvererbung. Sie wäre dann praktisch, wenn Sie zwei Klassen verschmelzen wollten, zum Beispiel die Klasse *Pferd* mit der Klasse *Esel*. Die neue Kreuzung *Muli* würde Attribute und Verhalten beider Basisklassen erben (→ Abbildung 3.11). Aber welche Attribute und welches Verhalten? Sollen sich Mulis verständigen und laufen wie Pferde oder wie Esel?

Bei derartigen Szenarien kommt die Softwareentwicklung an die Grenze des technisch Sinnvollen. Es ist nicht sinnvoll, Erbinformationen nach dem Zufallsprinzip zu übertragen, um die Natur zu imitieren. Der Anwender wünscht sich im Regel-

fall Programme, die über definierte Eigenschaften verfügen und deren Verhalten vorhersehbar ist.

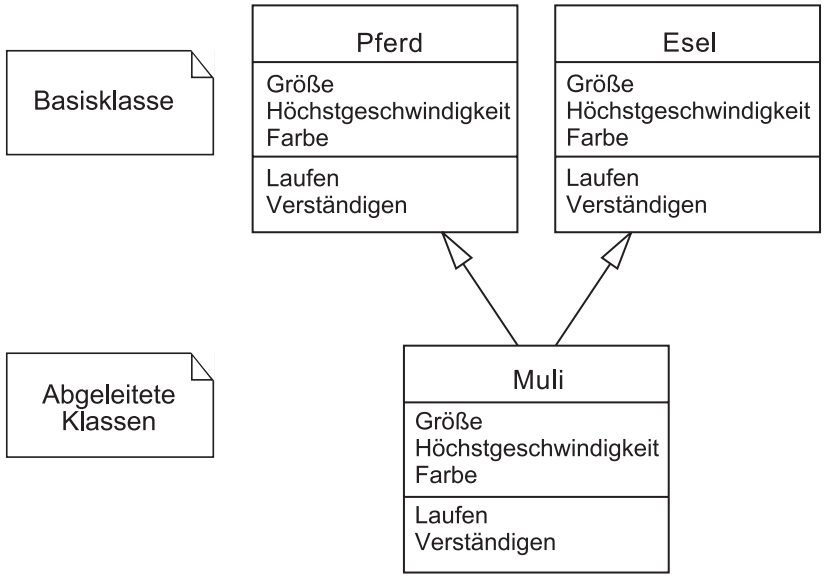


Abbildung 3.11 Mehrfachvererbung am Beispiel einer Kreuzung

Aus den genannten Gründen haben sich die Entwickler der Programmiersprache Java bewusst gegen die konventionelle Mehrfachvererbung entschieden, wie sie in C++ realisierbar ist. Wie Sie trotzdem mehrere Basisklassen ohne Nebenwirkungen miteinander verbinden können, erfahren Sie in → Kapitel 4 (Abschnitt 4.2.2 Klassen – Interfaces).

3.6 Kapselung

Eines der wichtigsten Merkmale objektorientierter Sprachen ist der Schutz von Klassen und Attributen vor unerwünschtem Zugriff. Jedes Objekt besitzt eine Kapsel, welche Daten und Methoden des Objekt schützt (→ Abbildung 3.12). Die Kapsel versteckt die Teile des Objekts, die von außen nicht oder nur durch bestimmte andere Objekte erreichbar sein sollen. Die Stellen, an denen die Kapsel durchlässig ist, nennt man *Schnittstellen*.

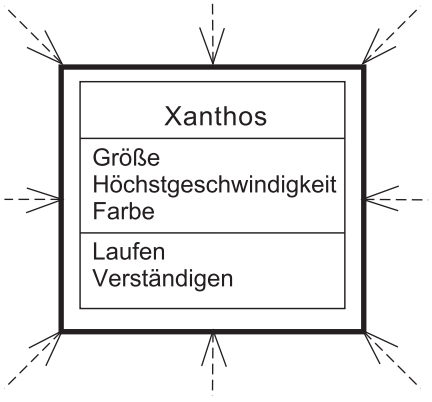


Abbildung 3.12 Die Kapsel schützt das Objekt vor unerwünschten Zugriffen.

Die wichtigste Schnittstelle der Klasse *Pferd* ist sein Konstruktor. Über diese spezielle Methode lässt sich ein Objekt der Klasse *Pferd* erzeugen. Ein anderes Beispiel für eine solche Schnittstelle ist die Methode *Laufen* der Klasse *Pferd*. Das Objekt *Xanthos* besitzt eine solche Methode *Laufen* und *Achilles*, ein Objekt der Klasse Mensch, kann diese Methode verwenden. Er kommuniziert mit *Xanthos* über diese Schnittstelle (→ Abbildung 3.13) und teilt darüber *Xanthos* mit, dass er laufen soll. Das Objekt *Achilles* darf nicht alle Daten von *Xanthos* verändern. Zum Beispiel soll es ihm selbstverständlich nicht erlaubt sein, die Größe des Pferds zu ändern. Gäbe es eine öffentlich zugängliche Methode wie zum Beispiel *Wachsen*, so könnte er *Xanthos* damit verändern.

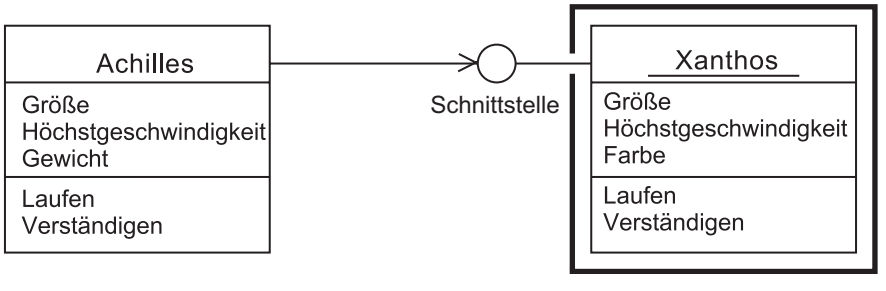


Abbildung 3.13 Objekte kommunizieren nur über Schnittstellen.

In technischer Hinsicht wollten die Entwickler der objektorientierten Programmiersprachen verhindern, dass ein Programmteil einen anderen – absichtlich oder unabsichtlich – verändert. Dies gehört zu den eklatanten Schwächen einiger klassischer Programmiersprachen.

3.7 Beziehungen

Wenn ein Objekt die Schnittstelle eines anderen Objekts verwendet, das heißt, wenn Objekt eine Methode wie einen Konstruktor eines anderen Objekts aufruft, so besteht zwischen den beiden Objekten bereits eine Beziehung. Diese Beziehungen vererben sich über Klassen. Aus diesem Grund ist es auch völlig egal, ob man sagt »Objekt A steht in Beziehung zu Objekt B« oder »Klasse A steht in Beziehung zu Klasse B«.

Die objektorientierte Programmierung ist bei Beziehungen wieder sehr feinsinnig und unterscheidet gleich drei verschiedene Arten: einfache *Assoziationen* (Verknüpfungen), *Aggregationen* (Zusammenlagerungen) sowie *Kompositionen* (Zusammensetzungen).

3.7.1 Assoziation

Assoziation ist die einfachste Form einer Beziehung zwischen Klassen und Objekten. Wenn ein Objekt namens *Achilles* einem Objekt namens *Xanthos* die Botschaft Springen sendet, besteht zwischen *Achilles* und *Xanthos* eine Assoziation (→ Abbildung 3.14).

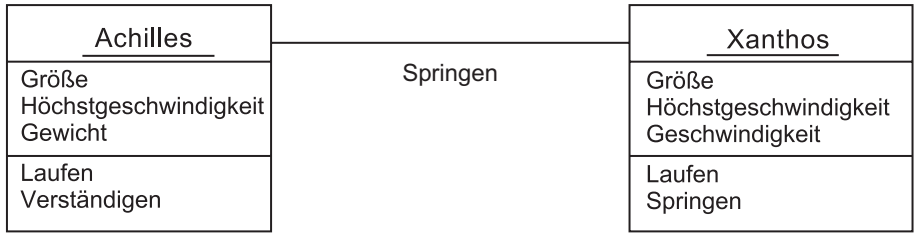


Abbildung 3.14 Eine einfache Assoziation zwischen Mensch und Pferd

3.7.2 Aggregation

Eine Aggregation besteht dann, wenn ein Objekt aus anderen Objekten besteht. Zum Beispiel soll Pferdefutter aus einer nicht näher bestimmten Anzahl von Karotten bestehen (→ Abbildung 3.15). Das bedeutet zum Beispiel, dass Pferdefutter eine Beziehung zur Karotte hat.

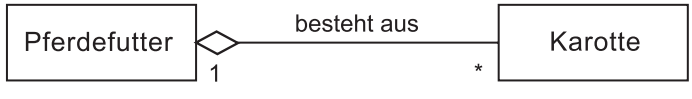


Abbildung 3.15 Aggregation zwischen Pferdefutter und Karotten

Diese Beziehung ist aber von einer völlig anderer Qualität als im vorhergehenden Beispiel zwischen einem Mensch und einem Pferd. Während Pferd und Mensch

allein und unabhängig voneinander existieren können, setzt sich das Pferdefutter (unter anderem) aus Karotten zusammen. Wichtig ist dabei, dass jedes Karotten-Objekt auch allein lebensfähig ist, was dieses Beispiel von der Komposition unterscheidet.

3.7.3 Komposition

Die stärkste Form der *Assoziation* stellt die *Komposition* dar. Wie bei der Aggregation liegt wieder eine »Besteht-aus-Beziehung« vor, sie ist aber im Gegensatz zur Aggregation abermals verschärft.

Ein Beispiel für eine Komposition ist das Verhältnis zwischen einem Pferd und seinen vier Beinen. Hier besteht eine enge Beziehung, denn ein Bein ist – im Gegensatz zur Karotte – als selbstständiges Objekt vollkommen sinnlos. Bei der Erzeugung eines Pferde-Objekts bekommt dieses automatisch vier Beine, die im Zusammenhang mit anderen Klassen nicht verwendet werden können.

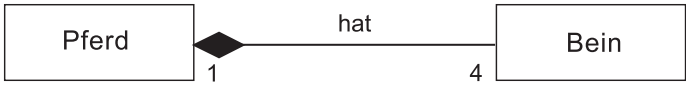


Abbildung 3.16 Pferd und seine vier Beine als Komposition

Pferdebeine sind ohne ein geeignetes Objekt der Klasse Pferd nicht lebensfähig. Wenn ein Pferde-Objekt stirbt, so sterben auch seine Pferdebeine.

3.8 Persistenz

Ein Programm erzeugt Objekte, die an ihrem Lebensende wieder zerstört werden. Diese Objekte bezeichnet man als transient, also flüchtig. Manchmal ist aber ein »Leben nach dem Tod« auch für Objekte erstrebenswert. Sie sollen auch dann wieder zum Leben erweckt werden, wenn das Programm beendet ist und der Anwender des Programms nach Hause geht. Am nächsten Tag startet der Anwender das Programm erneut und möchte mit dem gleichen Objekt weiterarbeiten.

Solche »unsterblichen« Objekte bezeichnet man als persistent (dauerhaft). Das bedeutet nichts anderes, als dass sie in geeigneter Form gespeichert werden. Sie befinden sich dann in einer Art Tiefschlaf auf einer Festplatte in einer Datei oder im Verbund mit anderen Objekten in einer Datenbank.

3.9 Polymorphie

Der Name Polymorphie kommt aus dem Griechischen und bedeutet so viel wie Vielgestaltigkeit, Verschiedengestaltigkeit. Der Begriff klingt mehr nach Mineralienkunde als nach Informatik und so wundert es Sie vielleicht auch nicht, dass der

Chemiker Mitscherlich die Polymorphie bei Mineralien Anfang des 19. Jahrhunderts entdeckte. Er stellte fest, dass manche Mineralien wie Kalziumcarbonat ($CaCO_3$) unterschiedliche Kristallformen annehmen können ohne ihre chemische Zusammensetzung zu ändern. Das bedeutet, sie können je nach Druck und Temperatur eine verschiedene Gestalt annehmen.

Alles sehr schön bis jetzt, aber was hat das mit objektorientierter Programmierung zu tun? – Das bedeutet auf keinen Fall, dass ein Objekt wie Xanthos so radikal seine Form verändern kann wie ein Mineral. Es bedeutet, dass Xanthos bei geschickter »Programmierung« in situationsbedingt verschieden reagieren kann. Klingt wie Zauberei, ist es aber nicht.

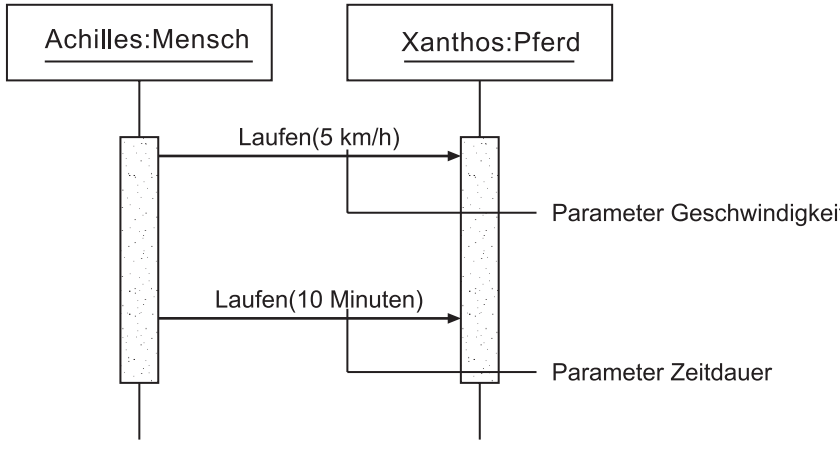


Abbildung 3.17 Xanthos verfügt über zwei verschieden gestaltete Methoden namens Laufen. Sie unterscheiden sich durch ihre Parameter.

3.9.1 Statische Polymorphie

Stellen Sie sich vor, das Objekt *Achilles* teilt dem Objekt *Xanthos* mit, dass er laufen soll, und zwar mit der Geschwindigkeit 5 km/h. Was wird passieren? – *Xanthos* wird sich mit dieser Geschwindigkeit fortbewegen. Offensichtlich ist die Richtung ebenso egal wie die Dauer. Was würde passieren, wenn *Achilles* abermals *Xanthos* mitteilt, er solle laufen, und zwar 10 Minuten? *Xanthos* würde 10 Minuten lang mit 5 km/h laufen und danach stehen bleiben.

Damit *Xanthos* den etwas wirr klingenden Anweisungen von *Achilles* Folge leisten kann, benötigt er Methoden »unterschiedlicher Gestalt«. Er benötigt eine Methode, die auf den Parameter Geschwindigkeit reagiert, und eine Methode, die auf den Parameter Zeitdauer reagiert. Obwohl die Methoden den gleichen Namen tragen, führen sie zu einer unterschiedlichen Verarbeitung durch das Objekt Xanthos. Der Fachausdruck für diese Technik heißt *Überladen*.

3.9.2 Dynamische Polymorphie

Anders als bei der Mehrfachvererbung sieht es aus, wenn man Eigenschaften der Basisklasse bei der Vererbung bewusst umgehen möchte. Dazu möchte ich nochmals auf das Beispiel der Basisklasse *Säugetier* zurückgreifen. Angenommen, Sie möchten in der abgeleiteten Klasse *Pferd* bestimmen, auf welche Weise sich Pferde-Objekte verständigen. Dazu *überschreiben* Sie die Methode *Verständigen* und legen die Art und Weise des Wieherns in der Klasse *Pferd* für die abgeleiteten Objekte fest.

Das Überschreiben von Methoden ist ein sehr mächtiges Mittel der objektorientierten Programmierung. Es erlaubt Ihnen, unerwünschte Erbinformationen teilweise oder ganz zu unterdrücken und damit eventuelle Designfehler – in Grenzen – auszugleichen beziehungsweise Lücken in der Basisklasse zu füllen. Dabei ist die Technik extrem simpel. Es reicht es aus, eine identische Methode in der abgeleiteten Klasse *Pferd* zu beschreiben, damit sich Objekte wie *Xanthos* »plötzlich« anders verhalten.

3.10 Designregeln

Auch wenn die objektorientierte Softwareentwicklung im Vergleich zur konventionellen Programmierung gutes Softwaredesign besser unterstützt, sie ist auf keinen Fall eine Garantie für sauber strukturierte und logisch aufgebaute Programme. Die objektorientierte Programmierung erleichtert zwar gutes Softwaredesign, sie erzwingt es jedoch nicht. Da man trotz Objektorientierung schlechte Programme entwickeln kann, sollten Sie einige Grundregeln beachten:

- Vermeiden Sie Vererbung.
- Reduzieren Sie die Anforderungen auf das Wesentliche.
- Kapseln Sie alle Attribute und Methoden, die nicht sichtbar sein müssen.
- Arbeiten Sie bei großen Projekten mit einem Modell.
- Verwenden Sie einen Prototypen.

3.11 Zusammenfassung

Die objektorientierte Programmierung war eine Antwort auf die Softwarekrise in der Mitte der 60er Jahre des letzten Jahrhunderts. Durch Objektorientierung lässt sich die natürliche Welt leichter in Computerprogrammen umsetzen. Diese objektorientierten Computerprogramme bestehen aus einer Sammlung von einem oder mehreren Objekten.

Ein Objekt lässt sich mit einem natürlichen Lebewesen vergleichen und verfügt über eine Gestalt und Fähigkeiten. Die Gestalt prägen Attribute, während die

Fähigkeiten von Methoden bestimmt sind. Beide Bestandteile eines Objekts sind in der Klasse festgelegt, von der ein Objekt stammt. Sie liefert den Bauplan für gleichartige Objekte.

Objektorientierte Programmierung ist kein Allheilmittel. Sie unterstützt gutes Design, ohne es zu erzwingen. Es ist deshalb notwendig, auf sauberes Design zu achten, wenn man mit objektorientierter Programmierung erfolgreich sein will.

3.12 Aufgaben

Versuchen Sie, folgende Aufgaben zu lösen:

3.12.1 Fragen

1. Worin unterscheiden sich Klassen von Objekten?
2. Wie unterscheiden sich Objekte der gleichen Klasse voneinander?
3. Was bedeutet der Begriff »Basisklasse«?
4. Was bedeutet der Begriff »abgeleitete Klasse«?
5. Wie verständigen sich Objekte untereinander?
6. Welche Arten von Beziehungen gibt es und wie unterscheiden sie sich?
7. Worin liegt die Gefahr bei Vererbungsbeziehungen?

3.12.2 Übungen

1. Zeichnen Sie zur Abbildung 3.18 eine Klasse mit Klassennamen, Attributen und Methoden.

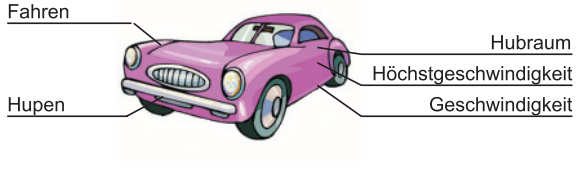


Abbildung 3.18 Ein Objekt mit verschiedenen Merkmalen und Fähigkeiten

2. Zeichnen Sie zur Abbildung 3.19 eine gemeinsame Basisklasse mit Klassennamen, Attributen und Methoden.

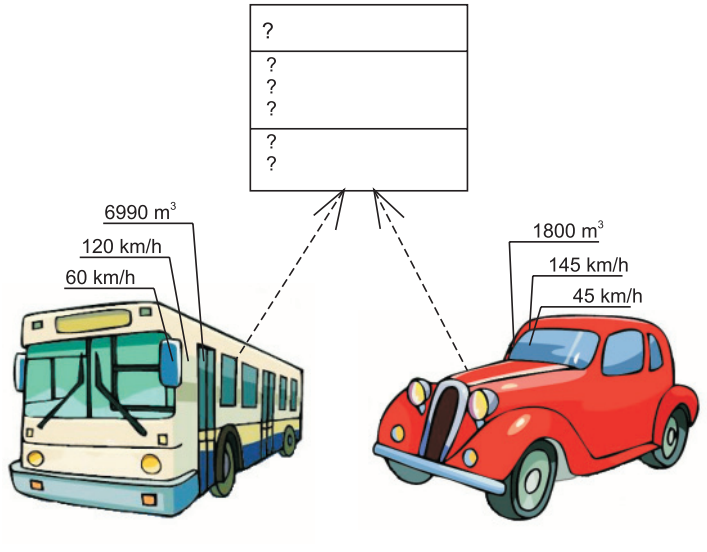


Abbildung 3.19 Zwei verschiedene Objekte

3. Zeichnen Sie zur Abbildung 3.20 ein Klassendiagramm mit einer Basisklasse und drei abgeleiteten Klassen, die in Beziehung zur Basisklasse stehen.

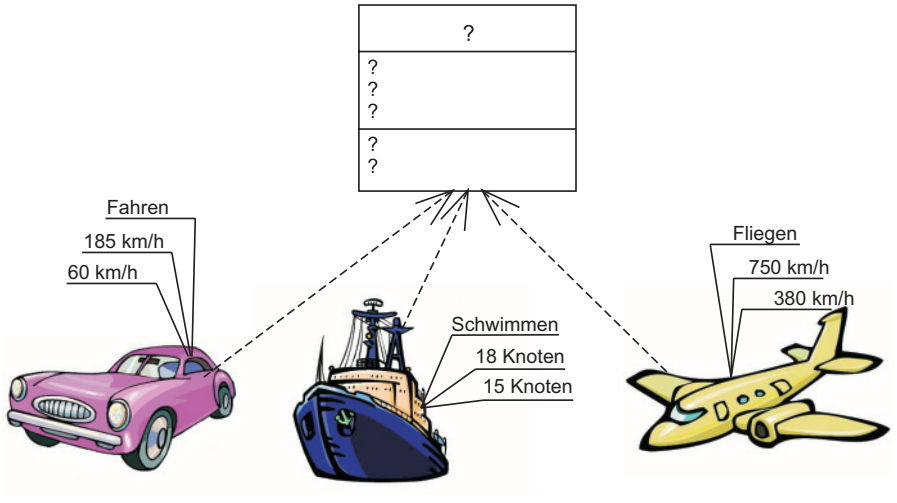


Abbildung 3.20 Drei veschiedene Objekte

Die Lösungen zu den Aufgaben finden Sie in → Teil IV, Kapitel 17, ab Seite 441.

Teil II
Java, Java, Java ...

Der vorhergehende Teil dieses Buchs stellte Ihnen die grundlegenden Konzepte der Datenverarbeitung, der Programmiersprachen und der objektorientierten Programmierung vor.

Dieser Teil baut direkt auf diesen Konzepten auf und setzt das Buch mit den drei Säulen der Java-Technologie fort:

- Sprache Java (→ Kapitel 4)
- Plattform Java (→ Kapitel 6)
- Java-Klassenbibliotheken (→ Kapitel 8)

Sie erfahren zwischen diesen Eckpfeilern der Java-Technologie, welche Entwicklungsprozesse ablaufen (→ Kapitel 5), welche Gesetzmäßigkeiten gelten (→ Kapitel 7), was Algorithmen sind und wozu sie der Java-Programmierer benötigt (→ Kapitel 9).

4 Sprache

»Sprache ist das Mittel, mit dem wir unsere Gedanken ausdrücken, und die Beziehung zwischen diesen Gedanken und unserer Sprache ist heimtückisch und verzwickt.« (William Wulf)

4.1 Einleitung

Das → Kapitel 3 hat Ihnen die Grundlagen zur objektorientierten Programmierung aus dem naiven Blickwinkel der natürlichen Welt vermittelt. Nun folgt die Umsetzung der objektorientierten Programmierung in Java. Sie werden feststellen, dass hier ein anderer Blickwinkel notwendig ist. Das liegt daran, dass Java zwar enorm viele Möglichkeiten bietet, ein Programm aufzubauen, aber in manchen Aspekten von der reinen Lehre der objektorientierten Programmierung abweicht.

4.1.1 Geschichte

Einige Programmierer der Firma Sun Microsystems entwickelten 1990 eine objektorientierte Sprache namens Oak (Object Application Kernel). Im Mittelpunkt des Projekts stand die Programmierung von Haushaltsgeräten. Da diese Geräte weder leistungsfähig noch einheitlich aufgebaut waren, mussten Oak-Programme sowohl kompakt als auch plattformunabhängig sein.

Die Sprache Oak schien sich nicht nur für Haushaltsgeräte gut zu eignen, sondern ebenso gut zur Internet-Programmierung, was der Firma Sun erfolgversprechender schien – kurzerhand wurde das Projekt neu ausgerichtet. Das Team sollte jetzt den ersten graphischen Internetbrowser auf Basis von Oak entwickeln.

Ungefähr ein Jahr später war der neue Browser in der Lage, kleine Programme (Applets) in HTML-Seiten darzustellen. Zusammen mit der Programmiersprache Java[1] erblickte er unter dem Name HotJava im Jahr 1995 das Licht der Welt.

1 Slangausdruck für Kaffee. Laut James Gosling, einem der Java-Autoren, verbrachte sein Team viele Stunden mit Brainstorming, um einen guten Namen für die neue Programmiersprache zu finden, bis ihnen in einer Kaffeebar die zündende Idee kam ...

4.1.2 Beschreibung mittels Text

Java-Programme werden in einer oder mehreren Unicode-Textdateien beschrieben. Unicode (→ Kapitel 1) bedeutet, dass Sie auch nationale Sonderzeichen verwenden könnten. Jede der Textdateien muss den Namen der Klasse tragen, die darin definiert ist, und die Endung `java` besitzen. Ein Beispiel: Wenn Sie ein Programm namens *Rectangle* mit einer Hauptklasse gleichen Namens schreiben möchten, so speichern Sie es einfach in einer Textdatei mit dem Namen `Rectangle.java` (→ Listing 4.1) ab.

```
package ch04.rectangle;
//CD/examples/ch04/ex01
class Rectangle {
public static void main(String[] arguments) {
        int height;
        int width;
        int area;
        height = 1;
        width = 5;
        area = height * width;
        System.out.println("Flaeche = " + area + " m^2");
    }
}
```

Listing 4.1 Das Java-Programm »Rectangle« als Textdatei Rectangle.java

Das hier vorgestellte Beispielprogramm *Rectangle* erzeugt die Ausgabe `Fläche = 5 m^2`. Das Beispielprogramm ist zwar sehr kurz, es weist aber trotzdem schon fast alle typischen Sprachelemente eines Java-Programms auf.

4.1.3 Überblick über die Sprachelemente

Paket und Klasse

Das Beispiel (→ Abbildung 4.1) besteht aus einem Paket (Punkt 1 und 2) mit einer Klasse namens *Rectangle* (Punkt 3). Diese Klasse enthält eine Startmethode namens *main* (Punkt 5). Sie setzt sich aus drei Typdeklarationen (Punkt 6), drei Zuweisungen mehrerer Werte (Punkt 7 und 8) und dem Aufruf einer Methode (Punkt 9) zusammen.

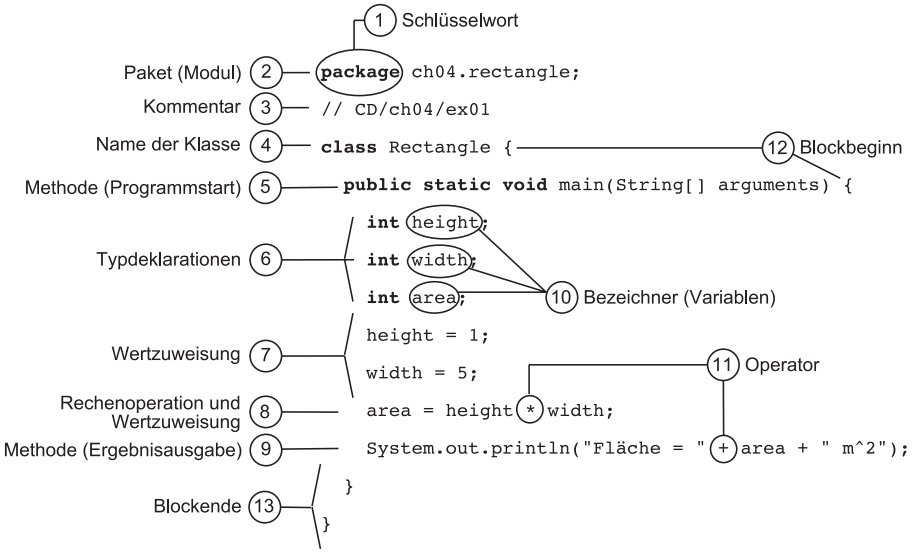


Abbildung 4.1 Übersicht über die wichtigsten Java-Sprachelemente

Programmstart

Das Package und die Klasse sollen Sie zunächst nicht interessieren, sondern nur der Programmstart, die Methode *main*. Sie beginnt mit drei Typdeklarationen. Diese Deklarationen legen fest, welche Datentypen die Variablen *height*, *width* und *area* bekommen, und reserviert für diese Speicherplatz. Die Wertzuweisungen im Anschluss daran definieren den aktuellen Wert der ersten beiden Variablen (Punkt 7), während die letzte Anweisung (Punkt 8) eine Wertzuweisung mit einer Rechenoperation kombiniert.

Rechenoperation

Die Rechenoperation multipliziert die Höhe des Rechtecks (Variable *height*) mit der Breite des Rechtecks (Variable *width*) und weist das Ergebnis der Fläche (Variable *area*) zu. An der Stelle 9 gibt die Klasse das Ergebnis mit Hilfe der Methode *println* (Punkt 9) aus.

Zusammenfassung

Zusammengefasst besteht das kurze Beispiel aus der Berechnung eines Rechtecks mit den Seiten *height* und *width* sowie der Ausgabe der Fläche *area* mit Hilfe der Methode *println*. Das Programm enthält folgende Java-Sprachelemente:

- Schlüsselwörter (zum Beispiel Punkt 1)
- Datentypen (zum Beispiel 4 und 6)

- Methoden (zum Beispiel 5 und 9)
- Operatoren (zum Beispiel Punkt 11)
- Anweisungen (zum Beispiel Punkt 7)
- Kommentare (zum Beispiel Punkt 3)

Soweit die – zugegebenermaßen sehr kurze – Analyse des Beispiels. Ich möchte im weiteren Verlauf des Kapitels Stück für Stück die Teile dieses Beispielprogramms ausführlicher beleuchten und mit den Schlüsselwörtern beginnen.

4.2 Schlüsselwörter

Bei der näheren Betrachtung des Programmbeispiels (→ Abbildung 4.1) fallen Ihnen eine ganze Reihe von fett gedruckten Begriffen auf, die eine reservierte Bedeutung in Java besitzen. Diese Wörter nennen sich Schlüsselwörter, von denen laut Java-Sprachdefinition zurzeit 49 existieren (→ Tabelle 4.1).

abstract	boolean	break	byte	case
catch	char	class	const[2]	continue
default	do	double	else	enum[3]
extends	final	finally	float	for
goto1	if	implements	import	instanceof
Int	interface	long	native	new
Package	private	protected	public	return
short	static	strictfp	super	switch
synchronized	this	throw	throws	transient
try	void	volatile	while	

Tabelle 4.1 Schlüsselwörter der Sprache Java

Das Schlüsselwort *enum* können Sie mit der aktuellen Java-Version 1.4 leider noch nicht verwenden. Es ist jedoch schon relativ sicher, dass es in der Java-Version 1.5 zur Verfügung stehen wird. Die Schlüsselwörter *const* und *goto* können im Gegensatz dazu niemals verwendet werden. Die Java-Erfinder haben sie reserviert, aber ihre Verwendung nicht gestattet – warum?

2 Ist reserviert, wird aber nicht benutzt
3 Für JDK 1.5 geplant

Sie dürfen nicht verwendet werden, um Problemen aus dem Weg zu gehen, die diese Schlüsselwörter bewirken können. Goto-Anweisungen führen zum Beispiel häufig zu schlechtem Programmdesign. Dass man sie dennoch reservierte, liegt daran, dass sie in C und C++ zum Sprachumfang gehören. Wenn ein C-/C++-Programmierer seine ersten Java-Programme schreibt, soll er sich nicht wundern, dass sein anscheinend korrektes Programm nicht wunschgemäß funktioniert. Er bekommt stattdessen schon bei der Übersetzung des Programms eine aussagekräftige Fehlermeldung.

Die Schlüsselwörter (→ Tabelle 4.1) besitzen verschiedene Funktionen in Java. Sie sind unter anderem für Folgendes reserviert:

- Einfache Datentypen (zum Beispiel *int*)
- Erweiterte Datentypen (zum Beispiel *enum*)
- Benutzerdefinierte Datentypen (zum Beispiel *class*)
- Klassenbeziehungen (zum Beispiel *extends*)
- Methodentypen (zum Beispiel *static*)
- Operatoren (zum Beispiel *new*)
- Anweisungen (zum Beispiel *for*)
- Module (zum Beispiel *package*)
- Fehlerbehandlung (zum Beispiel *try*)

Neben dem Schlüsselwort *class*, das den so genannten benutzerdefinierten Datentypen vorbehalten ist, und dem zusammengesetzten Datentyp *enum* befinden sich noch weitere »Typen« unter den Schlüsselwörtern, die *einfache Datentypen* genannt werden.

4.3 Einfache Datentypen

Die einfachen Datentypen sind Restbestände aus der verwandten Programmiersprache C. Während es in rein objektorientierten Sprachen wie Smalltalk (→ Kapitel 3) keine derartigen Datentypen gibt, haben sich die Erfinder von Java aus mehreren Gründen entschieden, einfache Datentypen zur Verfügung zu stellen.

Der erste Grund war, dass diese primitiven Java-Datentypen nur *reine* Daten ohne Methoden enthalten. Sie belegen daher wenig Speicherplatz – ganz im Gegensatz zu Objekten, die aus Daten *und* Methoden bestehen. Der zweite Grund war, dass die Java-Erfinder es den C- und C++-Programmierern erleichtern wollten, auf Java umzusteigen.

4.3.1 Grundlagen

Einfache Datentypen sollten Sie immer dann verwenden, wenn es nur darum geht, primitive Zahlenwerte im Programm zu speichern. Dazu müssen Sie dem Computer mitteilen, von welchem Datentyp eine Variable sein soll.

Eigenschaften bezeichnen

Der Vorgang, einer Variable einem Typ zuzuordnen, wird Deklaration genannt. Im Englischen spricht man auch von »to declare« (bezeichnen), weshalb sich der Begriff Deklaration bei der Übersetzung englischer Fachbücher etabliert hat. Durch die Deklaration sind zwei Eigenschaften der Variable unveränderlich festgelegt:

- Wertebereich
- Rechenoperationen

Java ist eine streng typisierte Programmiersprache. Das bedeutet, dass ein einmal festgelegter Datentyp für die Programmlebensdauer unveränderlich ist. Auch die Rechenoperationen, die an den Bezeichner gebunden sind, sind unveränderlich.

Aufbau der Deklaration

Wie die Deklaration aufgebaut ist, zeigt → Abbildung 4.2: Zunächst folgt der Datentyp und danach die Variable. Diese Deklaration wird durch ein Semikolon abgeschlossen und ist immer Teil einer Klasse, zum Beispiel in Form eines *Attributs*. Das heißt, Sie können eine Variable wie *height* niemals losgelöst von einer Klasse verwenden.

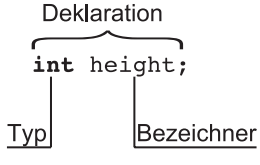


Abbildung 4.2 Deklaration der Variablen »height«

Übersicht der Datentypen

In → Tabelle 4.2 sehen Sie eine Übersicht über alle einfachen Java-Datentypen. Sie unterscheiden sich im Wertebereich, der die Größe des reservierten Speicherplatzes für den Bezeichner (Variable) bestimmt.

Typ	Speicherplatz [byte]	Wertebereich	Standardwert
`boolean`	1	`true, false`	`false`
`char`	2	Alle Unicode-Zeichen	\u0000
`byte`	1	$-2^7 \ldots 2^7-1$	0
`short`	2	$-2^{15} \ldots 2^{15}-1$	0
`int`	4	$-2^{31} \ldots 2^{31}-1$	0
`long`	8	$-2^{63} \ldots 2^{63}-1$	0
`float`	4	$\pm 3.40282347 * 10^{38}$	0.0
`double`	8	$\pm 1.79769313486231570 * 10^{308}$	0.0

Tabelle 4.2 Übersicht der einfachen Java-Datentypen

Bezeichner

Die Variable wird im Fachjargon auch Bezeichner genannt. Sie *bezeichnet* einen Programmteil, der vom Programmierer festgelegt wird. Ein Bezeichner kann zum Beispiel eine primitive Variable sein, aber auch eine Klasse, ein Objekt oder eine Methode. Allen Bezeichnern ist gemeinsam, dass sie nicht den Namen eines der Java-Schlüsselwörter (→ Tabelle 4.1) tragen dürfen.

Genauigkeit und Wertebereich

Der Computer reserviert so viel Speicherplatz, wie für einen bestimmten Datentyp festgelegt ist. Zum Beispiel reserviert er für den Datentyp *int* eine 4 Byte »große« Speicherzelle (→ Abbildung 4.3). 4 Byte sind identisch mit 32 Bit (→ Kapitel 1).

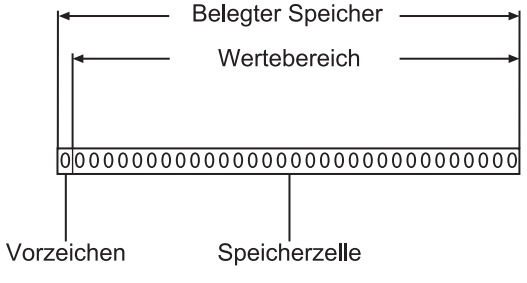


Abbildung 4.3 Wertebereich und belegter Speicher bei Zahlendatentypen am Beispiel des Datentyps »int«

Der reservierte Speicherbereich ist auf allen Computersystemen, auf denen Ihr Programm läuft, identisch. Sie müssen also nicht wie bei anderen Programmier-

sprachen zittern, wenn Ihr Programm auf ein anderes, fremdes Computersystem übertragen und dort ausgeführt werden soll.

Der reservierte Speicherbereich ist nicht nur auf allen Computersystemen, sondern auch für die gesamte Programmlaufzeit konstant. Der Speicherbereich ist bei den Zahlendatentypen aber nicht identisch mit dem Wertebereich. Das liegt daran, dass alle Java-Zahlendatentypen über ein Vorzeichen verfügen, das ebenfalls kodiert werden muss. Es vermindert den Wertebereich um ein Bit (→ Abbildung 4.3). Im Fall von *int* bedeutet das, dass »nur« 31 Bit nutzbar sind – jetzt können Sie sich auch den merkwürdigen Wertebereich der Zahlendatentypen in → Tabelle 4.2 erklären.

Im Fall von *int* ergibt sich der negative Wertebereich aus 2^7, der positive Wertebereich aus 2^7-1. Dass die Zahlendatentypen ein Vorzeichen besitzen, ist leider nicht immer praktisch. Für viele Fälle wären nur positive »natürliche« Zahlen mit einem Wertebereich von 0 bis 255 notwendig, den der Datentyp *byte* nicht bietet.

Der Wertebereich der Zahlendatentypen orientiert sich am maximalen Wert, der auf unterschiedlichen Computersystemen realisierbar ist. Sie erinnern sich: Java erlaubt es, portable Programme zu schreiben (→ Kapitel 2). Um eine Portabilität der Programme zu erreichen, mussten die Erfinder der Sprache darauf Rücksicht nehmen, was auf verschiedenen Rechnersystemen realisierbar ist.

Die höchste darstellbare Informationsmenge (Rechnerunendlich) liegt bei vielen Computersystemen bei 64 Bit, weshalb dies auch den Grenzwert der Java-Zahlendatentypen markiert. Das Rechnerunendlich bei PC-Systemen (mit mathematischem Coprozessor) beträgt allerdings 80 Bit und bleibt Java-Programmen leider verschlossen.

Mit dem maximal darstellbaren Wertebereich ist auch eine gewisse Genauigkeit verknüpft, da kein Computersystem Zahlen beliebig exakt zu verarbeiten vermag. Bei ganzzahligen Datentypen ist die Genauigkeit innerhalb der zugesicherten Grenzen stets optimal, da diese Zahlen immer vollständig gespeichert werden können, solange sie sich im Wertebereich befinden.

Gleitkommazahlen haben im Gegensatz zu Ganzzahlen prinzipiell nur eine beschränkte Genauigkeit, auch wenn sie sich im Wertebereich des Datentyps befinden. Das liegt daran, dass der Computer solche Zahlen nur dann vollständig speichern kann, wenn sie über eine beschränkte Zahl von Nachkommastellen verfügen.

Beim Speichern einer Gleitkommazahl zerlegt der Computer diese in zwei Teile. Der erste Teil ist der Exponent und der zweite Teil ist die Mantisse; beide werden binär gespeichert. Die Dezimaldarstellung (→ Abbildung 4.4) zeigt, dass eine sol-

che Zahl nur bis zu einer gewissen Nachkommastelle exakt ist, alles andere fällt unter den Tisch. Man spricht in diesem Fall von so genannten Rundungsfehlern.

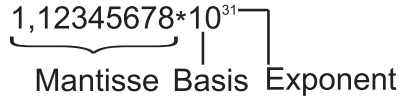


Abbildung 4.4 Die Stellen der Mantisse bestimmen die Genauigkeit.

Überschreiten des Wertebereichs

Sie müssen bei der Deklaration entscheiden, ob Ihnen der reservierte Wertebereich und die Genauigkeit für eine Variable im Laufe des Programms ausreichen. Ist das nicht der Fall, und die Variable überschreitet irgendwann ihren maximal gültigen Wert oder ist zu ungenau, kommt es zu Programmfehlern. Das kann sich unterschiedlich äußern.

Im günstigsten Fall fällt dies durch einen so genannten Überlauf auf. In manchen Fällen kann es bei der Sprache Java jedoch passieren, dass das Programm verrückt spielt und völlig falsche Werte produziert. In → Kapitel 7 erfahren Sie genau, in welchen Fällen dies passiert und wie Sie Ihr Programm vor solchen Zuständen schützen können.

Auswahl des Datentyps

Was bedeutet die Gefahr von Fehlern für die Auswahl eines Datentyps? – Das bedeutet zunächst, dass der Softwareentwickler schon bei der Programmierung sehr genau abschätzen sollte, welchen einfachen Datentyp mit welchem Wertebereich er verwendet.

Strategie 1: Ist der Entwickler zu sicherheitsbewusst und benutzt stets zu »große« Datentypen, läuft sein Programm zwar sicher, es verbraucht aber zu viel Speicher. **Strategie 2:** Ist er zu sparsam, braucht es wenig Speicherplatz, aber es wird nicht richtig funktionieren. Es liegt auf der Hand, dass Sie in Zweifelsfällen die erste Strategie bevorzugen sollten.

Programmtest

In jedem Fall muss der Entwickler in Bezug auf einfache Datentypen sorgfältig abwägen, welcher Datentyp für welchen Programmteil am besten geeignet ist, und sein Programm in Bezug auf einfache Datentypen ganz besonders sorgfältig testen (→ Kapitel 7). Wie Sie die Datentypen verwenden, zeigen einige kleine Anwendungsbeispiele in den folgenden Abschnitten.

4.3.2 Festkommazahlen

Festkommazahlen besitzen im Gegensatz zu Gleitkommazahlen (→ 4.3.3) keine Nachkommastellen. Diese Ganzzahlen dienen dazu, Zahlenwerte aus der natürlichen Zahlenmenge darzustellen. Java stellt die vier Ganzzahltypen *byte*, *short*, *int* und *long* zur Verfügung, die sich nur durch ihren Wertebereich unterscheiden.

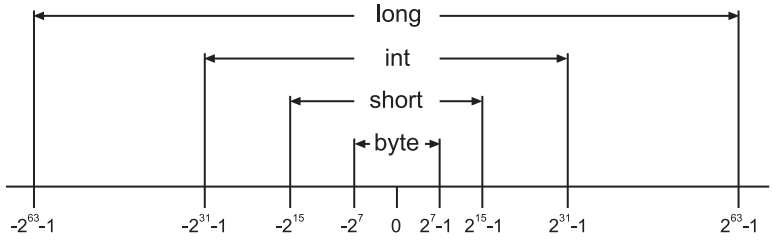


Abbildung 4.5 Wertebereich der Ganzzahltypen (nichtlineare Darstellung)

byte

Der Datentyp *byte* verfügt wie alle Ganzzahltypen über ein Vorzeichen. Er besitzt einen Wertebereich von einem Byte (daher der Name des Datentyps). Für das Beispielprogramm *Rectangle* hätte es also völlig ausgereicht, diesen Datentyp zu verwenden, weil sich die verwendeten Werte innerhalb des Wertebereichs (→ Abbildung 4.5) von *byte* befinden.

```
//CD/examples/ch04/ex02
(...)
        byte  height;
        byte  width;
        byte  area;
        height  =  1;
        width  =  5;
        area  =  (byte)(height  *  width);
        System.out.println("Fläche  =  "  +  area  +  "  m^2");
(...)
```

Listing 4.2 Variation des Beispiels »Rectangle« mit dem Datentyp »byte«

Das hier dargestellte Beispielprogramm *Rectangle* erzeugt die Ausgabe

```
Fläche = 5 m^2.
```

short

Für den Datentyp *short* gilt: Er hat auf allen Plattformen die gleiche Länge, verfügt über ein Vorzeichen und einen Wertebereich von 2 Byte. Im Vergleich zu den anderen Datentypen ist der Wertebereich relativ klein, daher auch die Bezeichnung *short*.


```
//CD/examples/ch04/ex03
(...)
        short  height;
        short  width;
        short  area;
        height  =  1;
        width  =  5;
        area  =  (short)(height  *  width);
        System.out.println("Fläche  =  "  +  area  +  "  m^2");
  (...)
```

Listing 4.3 Variation des Beispiels »Rectangle« mit dem Datentyp »short«

Auch dieses Beispiel erzeugt die Ausgabe

```
Fläche  =  5  m^2.
```

int

Der Datentyp *int* verdoppelt nochmals den Wertebereich des Vorgängers und besitzt ansonsten dessen genannte Eigenschaften. Er ist der am häufigsten eingesetzte Datentyp für Ganzzahlen in Java-Programmen.

```
//CD/examples/ch04/ex04
(...)
        int  height;
        int  width;
        int  area;
        height  =  1;
        width  =  5;
        area  =  height  *  width;
        System.out.println("Fläche  =  "  +  area  +  "  m^2");
  (...)
```

Listing 4.4 Ausschnitt aus dem Programmbeispiel »Rectangle«

Hier entsteht ebenfalls die Ausgabe

```
Fläche  =  5  m^2.
```

long

Dieser Datentyp erhöht nochmals den Wertebereich auf das Doppelte des Vorgängers und bietet mit 8 Byte (32 Bit) das Maximum an Wertebereich für Ganzzahlen innerhalb eines Java-Programms.

```
//CD/examples/ch04/ex05
(...)
        long  height;
        long  width;
        long  area;
        height  =  1L;
        width  =  5L;
        area  =  height  *  width;
        System.out.println("Fläche  =  "  +  area  +  "  m^2");
  (...)
```

Listing 4.5 Variation des Beispiels »Rectangle« mit dem Datentyp »long«

Auch dieses Beispiel verändert die Ausgabe des Programms im Vergleich zu den vorher genannten Beispielen nicht.

4.3.3 Gleitkommazahlen

Daten mit dem komischen Namen Gleitkommazahlen haben im Gegensatz zu Festkommazahlen eine variable Anzahl von Nachkommastellen (daher der Name). Sie dienen dazu, Zahlenwerte aus der rationalen Zahlenmenge zu verarbeiten. Sie können zum Beispiel durch eine Bruchrechnung entstehen.

$$\pi = \frac{\text{Kreisumfang}}{2 * \text{Kreisradius}} \approx 3{,}14$$

Abbildung 4.6 Beispiel für die Entstehung einer Gleitkommazahl

Java verfügt über die zwei Datentypen *float* und *double*, die sich durch ihre Wertebereiche unterscheiden (→ Abbildung 4.7).

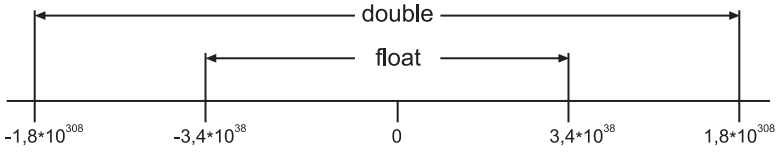


Abbildung 4.7 Wertebereich der Gleitkommatypen (gerundete Werte, nichtlineare Darstellung)

float

Der Typ *float* ist der Standardtyp für Gleitkommazahlen unter Java mit so genannter einfacher Genauigkeit (4 Byte). Einfache Genauigkeit reicht jedoch auch nur für einfache Rechenoperationen aus, weil die Anzahl der gespeicherten Nachkommastellen gering ist. Eine Anwendung zeigt → Listing 4.6.


```
//CD/examples/ch04/ex06
(...)
        float  height;
        float  width;
        float  area;
        height  =  1.1F;
        width  =  5.1F;
        area  =  height  *  width;
        System.out.println("Fläche  =  "  +  area  +  "  m^2");
  (...)
```

Listing 4.6 Variation des Beispiels »Rectangle« mit dem Datentyp »float«

Aufgrund der anderen Werte für Breite und Höhe entsteht auch eine andere Ausgabe:

```
Fläche  =  5.61  m^2.
```

double

Der Typ *double* beschließt den Abschnitt über Gleitkommazahlen. Sie benötigen diesen Typ immer dann, wenn mit höchstmöglichem Wertebereich und maximaler Genauigkeit bei den Nachkommastellen gerechnet werden muss. Das ist zum Beispiel bei Finanzdienstleistungssoftware, Flugsteuerungssoftware, medizinischen Anwendungen, Navigationssystemen oder Taschenrechnern der Fall. Ein Beispiel für die Verwendung zeigt → Listing 4.7.

```
//CD/examples/ch04/ex07
(...)
        double  height;
        double  width;
        double  area;
        height  =  1.1;
        width  =  5.1;
        area  =  height  *  width;
```

```
        System.out.println("Fläche  =  "  +  area  +  "  m^2");
  (...)
```

Listing 4.7 Variation des Beispiels »Rectangle« mit dem Datentyp »double«

Das Programm gibt ebenfalls `5.61 m^2` als Endergebnis aus.

4.3.4 Wahrheitswerte

Der in Java vordefinierte Datentyp für Wahrheitswerte kann die Werte *true* oder *false* annehmen. Das Verständnis von Wahrheitswerten ist von grundlegender Bedeutung für die Java-Programmierung. Wie Sie später sehen werden, steuern Sie mit Hilfe solcher Wahrheitswerte den Ablauf des Programms.

```
//CD/examples/ch04/ex08
(...)
        boolean  passwordChecked;
        boolean  userAuthorized;
        passwordChecked  =  true;
        userAuthorized  =  false;
        System.out.println("Passwort  überprüft  =  "  +
            passwordChecked);
        System.out.println("Benutzer  berechtigt  =  "  +
          userAuthorized);
  (...)
```

Listing 4.8 Programmbeispiel mit Wahrheitswerten

Dieses Beispiel erzeugt folgende Ausgabe:

```
Passwort  überprüft  =  true
Benutzer  berechtigt  =  false
```

4.3.5 Zeichen

Der Zeichentyp *char* ist mit einem Wertebereich von 2 Byte ausgestattet worden und basiert auf dem Unicode-Zeichensatz. *Char*-Typen sind im Gegensatz zu *String*-Typen (→ Kapitel 8) mit einfachen Hochkommata zu initialisieren (→ Listing 4.9). Der Zeichentyp ist zur Ausgabe einzelner Zeichen gedacht. Um Wörter auszugeben, müssen Sie einzelne Zeichen mit Hilfe von Arrays zu Zeichenketten zusammensetzen (→ 4.4.1 Arrays).

```
//CD/examples/ch04/ex09
(...)
char yesKey = 'J';
```

```
char cancelKey = 'A';
char helpKey = '?';
System.out.println("Soll der Vorgang fortgesetzt werden?");
System.out.println("<Ja> .......... [" + yesKey + "]");
System.out.println("<Abbrechen> ... [" + cancelKey + "]");
System.out.println("<Hilfe> ....... [" + helpKey + "]");
(...)
```

Listing 4.9 Ein Beispiel für die Verwendung des Char-Typs

Das Beispielprogramm sorgt für folgende Ausgabe:

```
Soll der Vorgang fortgesetzt werden?
<Ja> .......... [J]
<Abbrechen> ... [A]
<Hilfe> ....... [?]
```

4.4 Erweitere Datentypen

4.4.1 Arrays

Arrays zählen zu den erweiterten Datentypen. Es sind Felder, in denen Zahlen- oder Objektmengen gespeichert werden. Anders als in manchen anderen Programmiersprachen sind Arrays auch in Java *Objekte*. Das bedeutet, dass Arrays aus einer entsprechenden Klasse erzeugt werden.

Das → Listing 4.10 zeigt noch einmal einen Minimaldialog, aber diesmal unter Verwendung eines Char-Arrays. In diesem Fall erzeugt das Programm zwei Felder, einmal mit zwei Elementen und einmal mit neun Elementen.

```
//CD/examples/ch04/ex10
(...)
    char yesKey[]; // Deklaration ohne feste Laenge
    yesKey = new char [2]; // Erzeugung
    yesKey[0] = 'J'; // Zuweisung
    yesKey[1] = 'A'; // Zuweisung
    char cancelKey []; // Deklaration ohne feste Laenge
    cancelKey = new char [9]; // Erzeugung mit fester Laenge
    cancelKey[0] = 'A'; // Zuweisung
    cancelKey[1] = 'B'; // Zuweisung
    cancelKey[2] = 'B'; // Zuweisung
    cancelKey[3] = 'R'; // Zuweisung
    cancelKey[4] = 'E'; // Zuweisung
    cancelKey[5] = 'C'; // Zuweisung
```

```
    cancelKey[6] = 'H'; // Zuweisung
    cancelKey[7] = 'E'; // Zuweisung
    cancelKey[8] = 'N'; // Zuweisung
    char helpKey = '?';
    System.out.println("Wollen Sie eine Frage stellen?");
    System.out.println("<Ja> ..... [" + yesKey + "]");
    System.out.println("<Abbrechen> ... [" + cancelKey + "]");
    System.out.println("<Hilfe> ..... [" + helpKey + "]");
(...)
```

Listing 4.10 Ein Beispiel für ein Char-Array

Das Beispielprogramm erzeugt folgende Ausgabe:

```
Wollen Sie eine Frage stellen?
<Ja> ..... [JA]
<Abbrechen> ... [ABBRECHEN]
<Hilfe> ..... [?]
```

Arrays können eine Dimension oder mehrere Dimensionen besitzen. Die Anzahl der Elemente eines Arrays muss nicht zum Zeitpunkt der Deklaration feststehen. Wenn ein Array erzeugt wird, besitzt es jedoch eine feste Länge; Arrays sind infolgedessen halbdynamisch.

```
//CD/examples/ch04/ex11
(...)
    int numberArray [][];// Deklaration ohne feste Laenge
    numberArray = new int [1][2]; // Erzeugung mit fester Laenge
    numberArray[0][0] = 4; // Zuweisung
    numberArray[0][1] = 2; // Zuweisung
    System.out.println("Die Antwort lautet " +
                       numberArray[0][0] +
                       numberArray[0][1] );
(...)
```

Listing 4.11 Ein Programmdialog mit einem Int-Array

Das Beispiel erzeugt folgende Ausgabe:

```
Die Antwort lautet 42
```

Die bisherigen Beispiele waren nicht gerade sehr elegant, da sie Deklaration und Erzeugung trennten. Das nächste Beispiel fasst Deklaration und Erzeugung zusammen:

```
//CD/examples/ch04/ex12
(...)
  public static void main(String[] arguments) {
    // Deklaration und Erzeugung mit fester Laenge:
    int numberArray [][] = new int [1][2];
    numberArray[0][0] = 4; // Zuweisung
    numberArray[0][1] = 2; // Zuweisung
    System.out.println("Die Antwort lautet immer noch " +
                       numberArray[0][0] +
                       numberArray[0][1] );
(...)
```

Listing 4.12 Kombination von Deklaration und Erzeugung

Ebenso können Sie auch gleich die Wertemenge als Aufzählung übergeben:

```
//CD/examples/ch04/ex13
(...)
   public static void main(String[] arguments) {
      char yesKey[] = {'J', 'A'};
      char cancelKey [] =
         {'A', 'B', 'B', 'R', 'E', 'C', 'H', 'E', 'N'};
      char helpKey = '?';
      System.out.println("Noch 'ne Frage?");
      System.out.println("<Ja> ..... [" + yesKey + "]");
      System.out.println("<Abbrechen> ... [" + cancelKey + "]");
      System.out.println("<Hilfe> ..... [" + helpKey + "]");
(...)
```

Listing 4.13 Direkte Zuweisung der Zeichenkette

Der Index eines Arrays muss ein ganzzahliger Wert vom Typ *int*, *short*, *byte* oder *char* sein. Die Anzahl der Elemente können Sie über die Variable *length* ermitteln, die jedes Objekt eines Array-Typs besitzt.

```
//CD/examples/ch04/ex14
(...)
      char yesKey[] = {'J', 'A'};
      char cancelKey [] =
         {'A', 'B', 'B', 'R', 'E', 'C', 'H', 'E', 'N'};
      char helpKey = '?';
      System.out.println("Noch 'ne Frage?");
      System.out.println("<Ja> ..... [" + yesKey + "]");
      System.out.println("<Abbrechen> ... [" + cancelKey + "]");
```

```
      System.out.println("<Hilfe> ....... [" + helpKey + "]");
      System.out.println("Die Tasten haben " +
        (yesKey.length + cancelKey.length + 1) + " Zeichen");
(...)
```

Listing 4.14 Dieses Programm ermittelt die Länge der Zeichenketten.

Das Programm ermittelt die Länge der Zeichenketten, addiert sie und gibt folgenden Text aus:

```
<Ja> ..........  [JA]
<Abbrechen> ...  [ABBRECHEN]
<Hilfe> .......  [?]
Die Tasten haben 12 Zeichen
```

4.4.2 Aufzählungstyp

Zum Zeitpunkt der Drucklegung dieses Buchs gab es leider noch keine hundertprozentig verlässlichen Informationen über den neuen Aufzählungstyp, außer dass er das Schlüsselwort *enum* besitzen wird. Der Abdruck des Beispiels erfolgt also ohne Gewähr, dass sich das Programm mit dem kommenden JDK 1.5 übersetzen lässt.

```
//CD/ch04/ex15
(...)
public enum Week {
        monday, tuesday, wednesday,
        thursday, friday, saturday, sunday
}
(...)
```

Listing 4.15 Beispiel für die Verwendung des neuen enum-Typs

4.5 Benutzerdefinierte Datentypen

Klassen zählen zu den so genannten benutzerdefinierten Datentypen. Das bedeutet, dass Sie im Gegensatz zu den einfachen vordefinierten Datentypen wie *int* oder *double* bestimmen können, wie sich der neue Datentyp zusammensetzt. In der Ausprägung einer Klasse trifft die exakte Welt des Computers (→ Kapitel 1) auf die menschliche Sichtweise der natürlichen Welt (→ Kapitel 3).

Dreiklassengesellschaft

Es gibt drei Arten von Klasse in Java:

- Konkrete Klasse
- Abstrakte Klasse
- Interface

Klassenvariablen

Klassenvariablen deklariert man durch das Schlüsselwort *static*. Sie sind nicht an ein Objekt gebunden, sondern existieren ab dem Zeitpunkt, an dem eine Klasse geladen wird, bis zur Beendigung des Programms. Statische Variablen können praktisch sein, da sie so lange leben wie das Programm.

Objektvariablen

Die normale Form einer Variablen ist eine Objektvariable. Sie wird beim Erzeugen eines Objekts zum Leben erweckt und mit ihm wieder zerstört.

Konstanten

Konstanten sind – das klingt paradox – für Java nichts anderes als Variablen mit festem Wert. Sie werden ebenfalls durch das Schlüsselwort *final* gekennzeichnet. Will man eine Kostante erzeugen, die für alle Klassen gilt, kombiniert man das Schlüsselwort *static* mit *final*. Konstanten schreibt man in Versalien (Großbuchstaben). Wie eingangs erwähnt, existiert zwar das spezielle Schlüsselwort *const*, es darf aber nicht verwendet werden.

4.5.1 Konkrete Klasse

Wie der Name schon andeutet, können Sie von einer konkreten Klasse auch konkrete Exemplare (Objekte) erzeugen. Wenn Ihnen diese Aussage seltsam erscheint, müssen Sie bedenken, dass man von den beiden anderen Ausprägungen einer Klasse, den abstrakten Klassen und Interfaces, keine Objekte erzeugen kann. Eine konkrete Klasse kennen Sie bereits vom Anfang dieses Kapitels: die Klasse *Rectangle*. Sie verfügt über die Attribute *height* und *width*, die ihre Objektvariablen sind.

```
//CD/examples/ch04/ex16
package  ch04.rectangle;
public  class  Rectangle  {
```

```
    int  height;
    int  width;
}
```

Listing 4.16 Die Klasse Rechteck mit ihren Attributen

Objekte erzeugen

Ein neues Exemplar (Objekt) des Typs *Rectangle*, ein neues Rechteck, erzeugen Sie wie im Listing angegeben mit dem so genannten new-Operator (→ 4.7 Operatoren).

```
//CD/examples/ch04/ex17
package  ch04.rectangle;
public  class  TestApp  {
    public  static  void  main(String[]  arguments)  {
        Rectangle  rect;  //  Deklaration  des  Objekts  rect
        rect  =  new  Rectangle();//  Erzeugung  des  Objekts
    }
}
```

Listing 4.17 Ein neues Rechteck entsteht

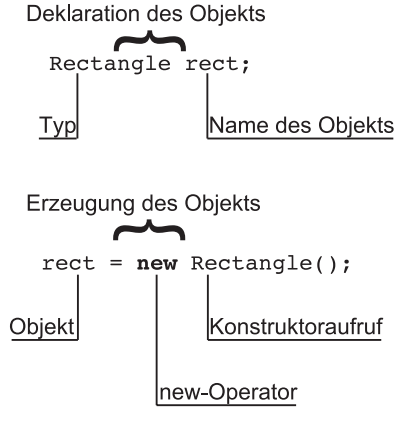


Abbildung 4.8 Deklaration und Erzeugung des Objekts »rect«

Wie Sie das von einfachen Datentypen kennen gelernt haben, muss die neue Variable *rect* des Typs *Rectangle* zuerst deklariert werden. Danach erfolgt die Erzeugung. Der Konstruktor wird wie eine normale Methode aufgerufen (→ Abbildung 4.8). Damit das Objekt erzeugt wird, ist es erforderlich, den new-Operator einzusetzen.

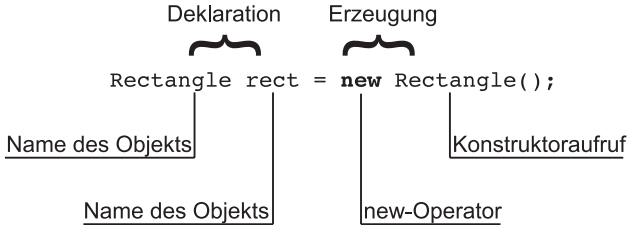


Abbildung 4.9 Kombination von Deklaration und Erzeugung eines Objekts

Die Deklaration und Erzeugung eines Objekts lässt sich auch kombinieren und in eine Zeile schreiben (→ Abbildung 4.9).

Lokale Klassen

Lokale Klassen definiert man innerhalb einer anderen Klasse. Sie können auch nur von dieser Klasse verwendet werden. Im Gegensatz zu anonymen Klassen besitzen sie einen konkreten Namen.

```
//CD/examples/ch04/ex18
package ch04.rectangle;
class Rectangle {
    private int height;
    private int width;
    public void Rectangle() {
        new Pattern(); // Erzeugung des Objekts
    }
    class Pattern { // Lokale Klasse
        private int dummy;
    }
}
```

Listing 4.18 Die innere Klasse »Pattern«

Innere Klassen sind vor allem bei der Programmierung graphischer Oberflächen nützlich, wo sie als Hilfsklassen dienen.

Anonyme Klassen

Eine weitere Form von Hilfsklassen, die innerhalb einer anderen Klasse definiert wurden, nennt sich innere Klassen. Im Gegensatz zu den eben erwähnten lokalen Klassen besitzt diese Spezies keinen Namen.

```
//CD/examples/ch04/ex19
package ch04.rectangle;
```

```
class Rectangle {
    private int height;
    private int width;
    public Rectangle() {
        new Pattern() { // Anonyme Klasse
            private int dummy;
        };
    }
}
```

Listing 4.19 Beispiel für die Verwendung einer anonymen Klasse

Das → Listing 4.19 zeigt eine aus der Klasse *Pattern* erzeugte Klasse, die nur über ein Attribut, aber nicht über einen Namen verfügt.

Vererbung

Wenn man eine Klasse vererben (ableiten) möchte, erweitert man sie um bestimmte Eigenschaften. Das Schlüsselwort für die Vererbung heißt entsprechend *extends*.

Folgendes Beispiel: Es soll eine Klasse *Shape* definiert werden, die als Basisklasse für geometrische Figuren dient. Diese Klasse wird von *Rectangle* erweitert.

```
//CD/examples/ch04/ex20
package ch04.shapes;
class Rectangle extends Shape {
    public Rectangle() {
    }
}
```

Listing 4.20 Rectangle erweitert die Klasse Shape

4.5.2 Abstrakte Klassen

Die Klasse *Shape* des Beispiels signalisiert schon durch ihren Namen, dass von ihr keine konkreten Objekte erzeugt werden sollen. Von einer konkreten Klasse lässt sich dies nicht verbieten. Dazu muss man eine Klasse als *abstrakt* definieren (→ Listing 4.21).

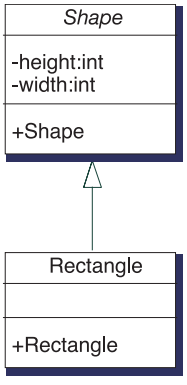


Abbildung 4.10 Die Klasse »Rectangle« erweitert »Shape«.

Der zugehörige Java-Code sieht so aus:

```
//CD/examples/ch04/ex21
package ch04.shapes;
abstract public class Shape {
    private int height;
    private int width;
    public Shape() {
    }
}
```

Listing 4.21 Die Klasse »Shape« als abstrakte Klasse

Vererbung

Wie bei einer konkreten Klasse erfolgt die Vererbung mit dem Schlüsselwort *extends*.

4.5.3 Interfaces

Die Schnittstelle (Interface) ist eine spezielle Form der Klasse, mit der eine Art von Mehrfachvererbung realisiert werden kann. Ein Interface ist eine Sammlung von abstrakten Methoden und Konstanten. Es enthält keine Konstruktoren und daher gibt es auch keine Objekte davon. Von einem Interface wird stets eine abgeleitete Klasse benötigt, die *alle* Methoden des Interfaces implementieren muss.

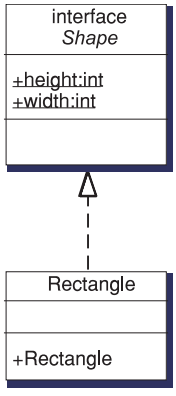


Abbildung 4.11 Die Klasse »Rectangle« implementiert das Interface »Shape«

Es gib drei wichtige Gründe, Interfaces einzusetzen:

- Kapselung von Komponenten
- Realisierung von Mehrfachvererbung
- Zusammenfassung identischer Methoden

Komponenten bilden eine Kapsel um mehrere Klassen, deren Schnittstellen nicht vollständig nach außen gelangen sollen. Eine Schnittstelle bietet hier eine Untermenge der inneren Schnittstellen.

Mehrfachvererbung ist in Java aufgrund der in Kapitel 3 erwähnten Nachteile nicht realisiert worden. Dennoch kann es aus architektonischen Gründen wichtig sein, eine Methodendeklaration von mehr als einer Klasse zu erben. Genau dies ist der Sinn von Interfaces.

```
//CD/examples/ch04/ex22
package  ch04.shapes;
public  interface  Shape  {
    int  height  =  1;
    int  width  =  5;
}
```

Listing 4.22 Die Klasse »Shape« als Interface

Die Klasse *Shape* ist hier nur in einer Minimalausführung zu sehen. Normalerweise verfügt eine solche Klasse über eine Reihe von abstrakten Methoden, die die abgeleitete Klasse mit Leben füllt. Ein Interface ist die maximale Steigerung einer abstrakten Klasse.

```
//CD/examples/ch04/ex22
package  ch04.shapes;
```

```
class Rectangle implements Shape {
    public Rectangle() {
    }
}
```

Listing 4.23 Die Klasse »Rectangle« implementiert »Shape«.

Vererbung

In den gerade eingeführten Interfaces gibt es eigentlich nichts zu erben, da sie nur eine Summe von Schnittstellen anbieten, die zu implementieren sind. Entsprechend heißt dort das Schlüsselwort für die Vererbung auch *implements* (→ Listing 4.23).

4.6 Methoden

Das → Kapitel 3 hat Methoden als die Fähigkeit eines objektorientierten Programms beschrieben, zu kommunizieren und Aufgaben zu erledigen. Methoden sind das objektorientierte Äquivalent zu Funktionen einer prozeduralen Programmiersprache.

Methodenarten

Für verschiedene Zwecke besitzt Java verschiedene Arten von Methoden:

- Konstruktoren
- Destruktoren
- Accessoren (Getter-Methoden)
- Mutatoren (Setter-Methoden)
- Funktionen

Klassenmethoden

Klassenmethoden kennzeichnen Sie durch das Schlüsselwort *static*. Sie sind wie Klassenvariablen nicht an ein Objekt gebunden. Sie existieren ab dem Zeitpunkt, an dem eine Klasse geladen wird, bis zur Beendigung des Programms. Die bekannteste Methode ist die Startmethode *main* eines Programms.

Objektmethoden

Die übliche Form einer Methode ist die, die an ein Objekt gebunden ist. Alle Methoden gehören immer zu einer Klasse und bestehen aus einem Kopf und einem Rumpf. Der Kopf setzt sich aus der Angabe der Sichtbarkeit, des Typs des

Rückgabewerts sowie aus der Signatur zusammen. Der Rumpf besteht aus Anweisungen.

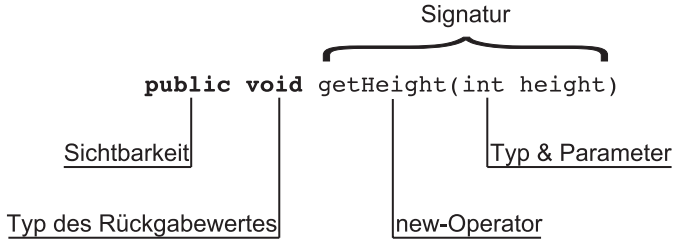


Abbildung 4.12 Eine Methode im Detail

Sichtbarkeit einer Methode

Die Sichtbarkeit von Klassen, Methoden und Variablen behandelt ausführlich → Kapitel 7. An dieser Stelle ist nur wichtig, dass es vier Stufen gibt, die Kapselung einer Methode festzulegen: *public*, *protected*, *default* und *private*. Wie schon in Kapitel 3 erwähnt, dient die Kapselung dazu, das Objekt vor Zugriffen anderer zu schützen.

Typ des Rückgabewertes

Alle Methoden außer Konstruktoren besitzen in Java einen bestimmten Typ des Rückgabewertes. Man unterscheidet hier zwei Fälle:

Fall 1: Falls sie einen konkreten Wert zurückliefern, dann entspricht der Typ des Rückgabewerts dem Typ der Methode.

Fall 2: Sie geben keine konkreten Werte zurück. Dann sind sie vom Typ *void* (engl. für: leer, unbesetzt).

Konstruktoren geben zwar keine konkreten Werte zurück, sie dürfen aber trotzdem nicht mit *void* gekennzeichnet werden, um sie von normalen Methoden zu unterscheiden. Eine Definition in der Art

```
void Rectangle(int height, int width) { ... }
```

wird als normale Methode interpretiert und hat eine völlig andere Wirkung als der Konstruktor:

```
Rectangle(int height, int width) { ... }
```

Falls Sie eine Klasse *Rectangle* definieren, die nur eine Methode *Rectangle* des Typs *void* enthält, wird diese klaglos ausgeführt. Bei der Erzeugung eines Objekts der Klasse *Rectangle* ruft das Programm jedoch nicht die Methode *Rectangle* auf,

sondern den Standardkonstruktor gleichen Namens. Sie erhalten somit möglicherweise einen ganz anderen Programmablauf.

Signatur

Die Signatur einer Methode ist ihr Name und die Parameterliste (→ Abbildung 4.12).

Rumpf einer Methode

Der Rumpf einer Methode besteht aus Anweisungen, der eigentlichen Implementierung also.

4.6.1 Konstruktoren

Die speziellen Methoden zum Erzeugen von Klassen nennen sich Konstruktoren (Erbauer). Sie dienen dazu, ein Exemplar zu erzeugen und eventuell sogleich mit definierten Werten zu belegen. In der Klasse »*Rectangle*« könnte die Übergabe der Attribute *height* und *width* als Parameter so erfolgen:

```
//CD/examples/ch04/ex23
package ch04.shapes;
class Rectangle implements Shape {
    private int height;
    private int width;

    //Konstruktor:
    public Rectangle(int height, int width) { //Parameter-
übergabe
        this.height = height;
        this.width = width;
    }
}
```

Listing 4.24 Die Klasse »Rectangle« mit Konstruktor

Das Programmbeispiel (→ Listing 4.24) füllt die Klasse beim Erzeugen gleich mit Werten für die Höhe und Breite. Es ist sinnvoll, eine Klasse mit einer Vielzahl von solchen Konstruktoren auszustatten, die den unterschiedlichsten Einsatzbereichen genügen. Die Technik nennt sich Überladen von Methoden und wird in → Kapitel 7 beschrieben (7.5.2 Überschreiben von Methoden).

Standardkonstruktor

Es ist übrigens nicht notwendig, einen Konstruktor zu definieren. Wird kein Konstruktor bei der Klassendefinition angegeben, erzeugt der Compiler beim Übersetzen der Klasse automatisch einen leeren Konstruktor (Standardkonstruktor). Dieser hat allerdings nur eine Funktion: ein Objekt zu erzeugen.

4.6.2 Destruktoren

Destruktoren (Zerstörer) im Sinne von C++ gibt es in Java nicht. Sie werden wegen der in Java eingebauten automatischen Speicherverwaltung nicht benötigt. Es gibt aber sehr wohl eine Methode mit dem Namen *finalize*, über die alle Java-Klassen automatisch verfügen.

```
//CD/examples/ch04/ex24
(...)
class Rectangle implements Shape {
protected void finalize() { Methode finalize
        // Anweisungen ...
    }
}
```

Listing 4.25 Die Methode finalize

Groteskerweise ist der Aufruf dieses Pseudo-Destruktors in Java nicht garantiert. Sie sollten also die üblichen Aufräumarbeiten beim Zerstören eines Objekts nicht in diese Methode integrieren. Kritische Abläufe, die am Ende eines Programms erledigt werden müssen, sind an einer anderen Stelle besser aufgehoben.

4.6.3 Accessoren

Will man Informationen von einer Klasse erhalten, muss man ihr eine Botschaft zukommen lassen. Diese Botschaften liefern Werte zurück und greifen auf Informationen der Klasse zu. Im Englischen bezeichnet man sie deshalb als »Accessors« oder »Accessor Methods«. Im Deutschen hat sich der Name Zugriffsmethoden, Getter-Methoden oder Accessoren etabliert.

```
//CD/examples/ch04/ex25
package ch04.shapes;
public class Rectangle implements Shape {
    private int height;
    private int width;
    public Rectangle(int height, int width) {
        this.height = height;
```

```
        this.width  =  width;
    }
    public  int  getHeight()  {
        return  this.height;
    }
    public  int  getWidth()  {
        return  this.width;
    }
}
```

Listing 4.26 Die Accessoren der Klasse »Rectangle«

Die Accessoren sind so aufgebaut, dass vor dem Methodennamen der Typ des Rückgabewerts stehen muss. Die Methode gibt das Ergebnis über die *Anweisung return this.height* zurück. Das Schlüsselwort *this* ist momentan nicht wichtig. Wichtig ist das Schlüsselwort *return*. Es bewirkt die Rückgabe des darauf folgenden Wertes.

4.6.4 Mutatoren

Methoden, die auf Daten eines Objekts zugreifen, werden Setter-Methoden, Zugriffsmethoden oder Mutatoren genannt. Man nennt sie auch Mutatoren, weil sie die Daten des Objekts mutieren (verändern). Die entsprechenden Methoden für die Klasse *Rectangle* sehen folgendermaßen aus:

```
//CD/examples/ch04/ex26
(...)
public  void  setHeight(int  height)  {
        this.height  =  height;
    }
    public  void  setWidth(int  width)  {
        this.width  =  width;
    }
  (...)
```

Listing 4.27 Die Mutatoren der Klasse »Rectangle«

Die Mutatoren geben keine Werte zurück, sondern übernehmen einen oder mehrere Werte als so genannte Parameter. Parameter sind Werte, die nach dem Namen der Methode innerhalb eines Klammerpaars übergeben werden. Da die Parameter deklariert werden müssen, erfolgt auch die Übergabe wie eine Deklaration stets nach dem Schema *Typ Bezeichner*.

Dadurch, dass Mutatoren keine Werte zurückliefern, ist der Typ der Methode kein Datentyp und keine Klasse. Die Methode ist von einem Typ, der keinen konkreten Wert zurückliefert. Wie schon erwähnt, kennzeichnet man derartige Methoden in Java mit dem Schlüsselwort *void*.

4.6.5 Funktionen

Funktionen wie das Ausrechnen von Zinsen oder das Starten eines Programms gehören zur dritten Art von Methoden, die Sie in einem Java-Programm antreffen können. Wie die spezialisierten Getter- und Setter-Methoden können sie Rückgabewerte besitzen oder nicht. Sie sind praktisch identisch aufgebaut.

```
//CD/examples/ch04/ex27
(...)
public  static  void  main(String[]  arguments)  {
//  Anweisungen
(...)
```

Listing 4.28 Die Startmethode eines Programms

4.7 Operatoren

Operatoren verknüpfen Variablen, Attribute und Objekte zu Ausdrücken (→ Seite 50). Folgende Operatoren sind verfügbar:

- Arithmetische Operatoren
- Vergleichende Operatoren
- Logische Operatoren
- Bitweise Operatoren
- Zuweisungsoperatoren
- Fragezeichenoperatoren
- New-Operator

4.7.1 Arithmetische Operatoren

Die klassischen mathematischen Operatoren Addition, Subtraktion, Division und Multiplikation sind auch in Java verfügbar. Daneben gibt es auch die Operatoren, die von C/C++ stammen.

Operator	Bezeichnung	Beispiel	Erläuterung
+	Positives Vorzeichen	+i	Synonym für i
-	Negatives Vorzeichen	-i	Vorzeichenumkehr von i
+	Summe	i + i	Führt eine Addition durch
-	Differenz	i - i	Führt eine Subtraktion durch
*	Produkt	i * i	Führt eine Multiplikation durch
/	Quotient	i / i	Führt eine Division durch
%	Divisionsrest (Modulo)	i % i	Ermittelt den Divisionsrest
++	Präinkrement	j = ++i	1. Schritt: i = i + 1 2. Schritt: j = i
++	Postinkrement	j = i++	1. Schritt: i = j 2. Schritt: i = i + 1
--	Prädekrement	j = --i	1. Schritt: i = i – 1 2. Schritt: j = i
--	Postdekrement	j = i--	1. Schritt: i = j 2. Schritt: i = i - 1

Tabelle 4.3 Arithmetische Operatoren

Positives Vorzeichen

Ein positives Vorzeichen ist stets optional, das heißt, es muss nicht verwendet werden, da ein Zahlenwert ohne Vorzeichen immer positiv belegt ist.

```
//CD/ch04/ex28
(...)
int  height;
int  width;
int  area;
height  =  +1;
width  =  +5;
  (...)
```

Listing 4.29 Die Variablen height und width besitzen positive Vorzeichen.

Negatives Vorzeichen

Ein negatives Vorzeichen bewirkt im Gegensatz dazu einen Vorzeichenwechsel. Die Multiplikation zweier negativer Zahlen ergibt – wie von der Mathematik bekannt – wieder eine positive Zahl.

```
//CD/ch04/ex29
(...)
        int  height;
        int  width;
        int  area;
        height  =  -1;
        width  =  -5;
  (...)
```

Listing 4.30 Die Variablen height und width mit negativen Vorzeichen

Summe

Der Additionsoperator bewirkt die Summenbildung der benachbarten Variablen.

```
//CD/ch04/ex30
(...)
        int  height;
        int  width;
        int  sum;
        height  =  1;
        width  =  5;
        sum  =  height  +  width;
        System.out.println("Summe  zweier  Seiten  =  "  +  sum
+  "  m");
(...)
```

Listing 4.31 Der Summenoperator verknüpft zwei Summanden mit einer Addition.

Das Beispielprogramm kalkuliert die Summe zweier Rechteckseiten und gibt Folgendes aus:

Summe zweier Seiten = 6 m

Differenz

Mit dem Differenzoperator verknüpfen Sie die benachbarten Variablen durch eine Subtraktion.

```
//CD/examples/ch04/ex31
        int  height;
        int  width;
        int  diff;
        height  =  1;
        width  =  5;
```

```
        diff = height - width;
        System.out.println("Differenz zweier Seiten = " +
                                              diff + " m");
  (...)
```

Listing 4.32 Differenzbildung zweier Variablen

Das Ergebnis des Beispielprogramms lautet:

```
Differenz zweier Seiten = -4 m
```

Produkt

Der Produktoperator verknüpft die benachbarten Variablen mit einer Multiplikation.

```
//CD/examples/ch04/ex32
(...)
height = 1;
width = 5;
area = height * width;
(...)
```

Listing 4.33 Das Produkt zweier Variablen

Das Ergebnis des Beispielprogramms ist die mehrfach verwendete Fläche eines Rechtecks.

Quotient

Bei der Verwendung des Divisionsoperators ist zu beachten, dass Java-Programme Zwischenergebnisse als *int*-Werte speichern. Aus diesem Grund muss der Typ des Ergebnisses konvertiert werden. Auf dieses Thema geht → Kapitel 7 ausführlich ein.

```
//CD/examples/ch04/ex33
(...)
int height;
int width;
float div;
height = 1;
width = 5;
div = (float) height // width;
System.out.println("Division = " + div + " m");
(...)
```

Listing 4.34 Die Berechnung einer Division

Um diese Konvertierung durchzuführen, verwenden Sie den Cast-Operator (→ 4.7.8). Das bedeutet, dass der neue Typ des Ergebnisses vor die Division gesetzt wird.

Divisionsrest

Der Divisionsrestoperator (Modulo-Operator) ermittelt den Rest einer ganzzahligen Division. Bei nachfolgendem Beispiel 5 : 3 = 1 ergibt sich ein Divisionsrest von 2, den das Beispiel auch anzeigt.

```
//CD/examples/ch04/ex34
(...)
        height  =  5;
        width  =  3;
        modulo  =  height  %  width;  //  5  :  3  =  1  =>
Rest  2
        System.out.println("Divisionsrest  =  "  +  modulo);
(...)
```

Listing 4.35 Der Modulo-Operator

Präinkrement

Die folgenden vier Operatoren sind ein Erbe von C++. Sie kombinieren Zuweisungen und Berechnungen. Der Präinkrement-Operator erhöht erst den Wert der Variable *height* und weist ihn danach der Variablen *result* zu. Präinkrement bedeutet vorher inkrementieren (erhöhen).

```
//CD/examples/ch04/ex35
(...)
height  =  1;
result = ++height; // 1.) height = height + 1; 2.) result =
result + 1
System.out.println("Höhe  =  "  +  height  +  "  m");
System.out.println("Ergebnis    =  "  +  result  +  "  m");
(...)
```

Listing 4.36 Der Präinkrement-Operator

Das Programm gibt Folgendes aus:

```
Höhe = 2 m
Ergebnis = 2 m
```

Postinkrement

Beim Postinkrement-Operator verhält es sich entgegengesetzt. Er weist im ersten Schritt den Wert der Variablen *height* der Variablen *result* zu und erhöht danach im zweiten Schritt den Wert von *height*.


```
//CD/examples/ch04/ex36
(...)
height = 1;
result= height++; // 1.) result = height; 2.) height =
height + 1
System.out.println("Höhe = " + height + " m");
System.out.println("Ergebnis = " + result + " m");
(...)
```

Listing 4.37 Der Postinkrement-Operator

Aus diesem Grund ergeben sich für die Variable *height* und für *result* auch andere Werte:

Höhe = 2 m

Ergebnis = 1 m

Prädekrement

Der Prädekrement-Operator setzt erst den Wert der Variablen *height* herab und weist ihn anschließend der Variablen *result* zu.

```
//CD/examples/ch04/ex37
(...)
int height;
int result;
height = 1;
result = --height; // 1.) height = height - 1; 2.) result =
height
System.out.println("Höhe = " + height + " m");
System.out.println("Ergebnis = " + result + " m");
(...)
```

Listing 4.38 Der Prädekrement-Operator

Aus diesem Grund sind beide Werte gleich, und das Ergebnis für beide Werte ist 0.

Postdekrement

Der Postinkrement-Operator verhält sich wieder entgegengesetzt. Er setzt im ersten Schritt den Wert der Variable *height* herab und weist das Ergebnis im zweiten Schritt der Variablen *result* zu.

```
//CD/examples/ch04/ex38
  (...)
int  height;
int  result;
height  =  1;
result  =  height--;  //  1.)  result  =  height  2.)  height  =
height  -  1;
System.out.println("Höhe  =  "  +  height  +  "  m");
System.out.println("Ergebnis  =  "  +  result  +  "  m");
  (...)
```

Listing 4.39 Der Postdekrement-Operator

Das Ergebnis des Programms sind auch diesmal unterschiedliche Werte:

```
Höhe = 0 m
Ergebnis = 1 m
```

4.7.2 Vergleichende Operatoren

Die relationalen (vergleichenden) Operatoren dienen, wie der Name es andeutet, dazu, Ausdrücke miteinander zu vergleichen. Auch hier wieder zunächst eine Übersicht über die verfügbaren Operatoren:

Operator	Bezeichnung	Beispiel	Erläuterung
==	Gleich	i == j	Vergleich auf Gleichheit
!=	Ungleich	-i	Vergleich auf Ungleichheit
<	Kleiner	i + i	Vergleich auf kleiner
<=	Kleiner gleich	i - i	Vergleich auf kleiner oder gleich
>	Größer	i * i	Vergleich auf größer
>=	Größer gleich	i / i	Vergleich auf größer oder gleich

Tabelle 4.4 Vergleichende Operatoren

Vergleich auf Gleichheit

Die einfachste Operation ist es zu prüfen, ob zwei Ausdrücke identisch sind. Das Ergebnis der Operation ist ein Wahrheitswert. Falls zwei Werte identisch sind, ergibt sich *true*, falls nicht, *false*.


```
//CD/examples/ch04/ex39
(...)
int  height;
int  width;
int  area;
height  =  1;
width  =  5;
area  =  height  *  width;
System.out.println(height  ==  width);
System.out.println(area  ==  width);
(...)
```

Listing 4.40 Überprüfung zweier Werte auf Gleichheit

Das Programm erzeugt die Ausgabe:

```
false
true
```

Zuerst wird die Multiplikation durchgeführt, die das Endergebnis 5 erzielt. Dieses Endergebnis bekommt die Variable *area* zugewiesen und ist damit identisch mit der Variablen *width*.

Vergleich auf Ungleichheit

Wenn man überprüfen möchte, ob zwei Werte nicht identisch sind, verwendet man den Ungleichheitsoperator.

```
//CD/examples/ch04/ex40
(...)
int  height;
int  width;
int  area;
height  =  1;
width  =  5;
area  =  height  *  width;
System.out.println(height  !=  width);
```

```
System.out.println(area  !=  width);
(...)
```

Listing 4.41 Überprüfung zweier Werte auf Ungleichheit

Wie zu erwarten, erzeugt das Programm diesmal die umgekehrte Ausgabe:

```
true
false
```

Vergleich auf kleiner

Um herauszufinden, ob ein Ausdruck oder Wert kleiner als ein anderer ist, verwenden Sie diesen relationalen Operator. Dazu wieder ein Beispiel:

```
//CD/examples/ch04/ex41
(...)
int  height;
int  width;
int  area;
height  =  1;
width  =  5;
area  =  height  *  width;
System.out.println(height  <  width);
System.out.println(area  <  width);
(...)
```

Listing 4.42 Überprüfung, ob ein Wert kleiner als ein anderer ist

Wie zu erwarten, erzeugt das Programm auch diesmal folgende Ausgabe:

```
true
false
```

Die Variable *height* ist kleiner als *width*. Dies ist also eine wahre Aussage. Die Variable *area* ist aber nicht kleiner als *width*, sondern identisch. Dies ist also eine falsche Aussage.

Vergleich auf kleiner oder gleich

Anders sieht das vorhergehende Beispiel aus, wenn Sie überprüfen wollen, ob die Werte kleiner oder gleich sind. Es reicht also schon aus, dass gleiche Werte miteinander verglichen werden, damit die Aussage wahr ist.

```
//CD/examples/ch04/ex42
(...)
int  height;
```

```
int  width;
int  area;
height  =  1;
width  =  5;
area  =  height  *  width;
System.out.println(height  <=  width);
System.out.println(area  <=  width);
(...)
```

Listing 4.43 Vergleich, ob ein Wert kleiner oder gleich einem anderen ist

Das Programm erzeugt die folgende Ausgabe:

```
true
true
```

Die Variable *height* ist kleiner als *width*. Dies ist also eine wahre Aussage. Die Variable *area* ist mit *width* identisch. Dies ist also eine wahre Aussage.

Vergleich auf größer

Um herauszufinden, ob ein Ausdruck oder Wert größer als ein anderer ist, müssen Sie diesen relationalen Operator einsetzen. Das Beispiel erzeugt diesmal natürlich eine entgegengesetzte Ausgabe, da beide Vergleiche keine wahren Aussagen ergeben:

```
//CD/examples/ch04/ex43
(...)
int  height;
int  width;
int  area;
height  =  1;
width  =  5;
area  =  height  *  width;
System.out.println(height  >  width);
System.out.println(area  >  width);
(...)
```

Listing 4.44 Vergleich, ob ein Wert größer als ein anderer ist

Vergleich auf größer oder gleich

Wenn Sie überprüfen wollen, ob Werte größer oder gleich sind, verwenden Sie den Größer-Gleich-Operator. Hier reicht es ebenfalls aus, dass gleiche Werte miteinander verglichen werden, damit die Aussage wahr ist.

```
//CD/examples/ch04/ex44
(...)
int  height;
int  width;
int  area;
height  =  1;
width  =  5;
area  =  height  *  width;
System.out.println(height  >=  width);
System.out.println(area  >=  width);
(...)
```

Listing 4.45 Vergleich, ob Werte größer oder gleich sind

Der Vergleich führt zu folgendem Ergebnis:

```
false
true
```

4.7.3 Logische Operatoren

Diese Operatoren setzen Sie ein, um Wahrheitswerte (→ Kapitel 1) miteinander zu vergleichen. Folgende Operatoren sind in Java verfügbar:

Operator	Bezeichnung	Beispiel	Erläuterung
!	Nicht	!i	Negation
&&	Und	i && i	Und-Verknüpfung
||	Oder	i || i	Oder-Verknüpfung

Tabelle 4.5 Vergleichende Operatoren

Negation

Um eine wahre Aussage umzukehren, verwendet man den Nicht-Operator. Das Beispiel hierzu vergleicht zwei Variablen miteinander. Das Ergebnis dieses Vergleichs ist nicht wahr, da beide unterschiedliche Werte besitzen. Der Nicht-Operator stellt diese Aussage auf den Kopf (Inversion) und daher ist das Endergebnis *true*.

```
//CD/examples/ch04/ex45
(...)
int  height;
int  width;
```

```
int  area;
height  =  1;
width  =  5;
area  =  height  *  width;
System.out.println(!(height  ==  width));
(...)
```

Listing 4.46 Der Nicht-Operator invertiert eine Aussage

Und-Vernüpfung

Folgendes Programm vergleicht im ersten Schritt die Variable *height* und *width* miteinander. Das Ergebnis ist eine falsche Aussage. Im zweiten Schritt vergleicht es die Variablen *area* und *width* miteinander. Das Ergebnis ist eine wahre Aussage. Werfen Sie nun nochmals einen Blick auf → Kapitel 1, Abbildung 1.6. Der Und-Operator verknüpft eine wahre und eine falsche Aussage so, dass das Endergebnis *false* entsteht.

```
//CD/examples/ch04/ex46
(...)
int  height;
int  width;
int  area;
height  =  1;
width  =  5;
area  =  height  *  width;
System.out.println(((height == width) && (area == width)));
(...)
```

Listing 4.47 Eine Und-Verknüpfung zweier Aussagen

Oder-Vernüpfung

Nochmals dieselbe Konstellation, aber diesmal mit einer Oder-Verknüpfung. Das Ergebnis des ersten Ausdrucks ist eine falsche Aussage. Das Ergebnis des zweiten Ausdrucks ist eine wahre Aussage. Werfen Sie nun einen Blick auf → Kapitel 1, Abbildung 1.6. Dort erkennen Sie, dass es reicht, wenn eine Aussage wahr ist, damit eine Oder-Verknüpfung ein wahres Ergebnis zurückliefert. Aus diesem Grund entsteht das Endergebnis *true*.

```
//CD/examples/ch04/ex47
(...)
int  height;
int  width;
int  area;
```

```
height = 1;
width = 5;
area = height * width;
System.out.println(((height == width) ||
(height == area)));
(...)
```

Listing 4.48 Diese Oder-Verknüpfung liefert ein wahres Ergebnis

4.7.4 Bitweise Operatoren

Bitweise Operatoren dienen dazu, Manipulationen auf der niedrigsten Ebene einer Speicherzelle, der Bitebene, durchzuführen. Zu ihrem Verständnis ist normalerweise Erfahrung in Assemblerprogrammierung nötig. Sie sollen hier nicht weiter interessieren.

Operator	Bezeichnung	Beispiel	Erläuterung
~	Einerkomplement	~i	Bitweise Negation
|	Bitweises Oder	i | i	Bitweises Oder
&	Bitweises Und	i & i	Bitweises Und
^	Exklusives Oder	i ^ i	Bitweises exklusives Oder
>>	Rechtsschieben mit Vorzeichen	i >> 2	Rechtsverschiebung
>>>	Rechtsschieben ohne Vorzeichen	i >>> 2	Rechtsverschiebung ohne Vorzeichenwechsel
<<	Linksschieben mit Vorzeichen	i << 2	Linksverschiebung

Tabelle 4.6 Vergleichende Operatoren

4.7.5 Zuweisungsoperatoren

Zuweisungsoperatoren dienen dem Namen gemäß dazu, Werte zuzuweisen. Java besitzt im Wesentlichen die von C++ bekannten Operatoren, welche die → Tabelle 4.7 zusammenfasst.

Operator	Bezeichnung	Beispiel	Erläuterung
=	Zuweisung	i = 1	i erhält den Wert 1
+=	Additionszuweisung	i += 1	i = i + 1

Tabelle 4.7 Zuweisungsoperatoren

Operator	Bezeichnung	Beispiel	Erläuterung
-=	Subtraktionszuweisung	i -= 1	i = i - 1
*=	Produktzuweisung	i *= 1	i = i * 1
/=	Divisionszuweisung	i /= 1	i = i / 1
%=	Modulozuweisung	i %= 1	i = i % 1
&=	Und-Zuweisung	i &= 1	i = i & 1
|=	Oder-Zuweisung	i |= 1	i = i | 1
^=	Exklusiv-Oder-Zuweisung	i ^= 1	i = i ^ 1
<<=	Linksschiebezuweisung	i <<= 1	i = i << 1
>>=	Rechtsschiebezuweisung	i >>= 1	i = i >> 1
>>>=	Rechtsschiebezuweisung mit Nullexpansion	i >>>= 1	i = i >>> 1

Tabelle 4.7 Zuweisungsoperatoren (Forts.)

Die Zuweisungsoperatoren bieten hier nicht Neues, sondern kombinieren nur die bisher bekannten Operatoren und die Zuweisung, so dass man sich beim Schreiben eines Programms eine Zeile sparen kann. Die Lesbarkeit des Programms lässt jedoch zu wünschen übrig.

4.7.6 Fragezeichenoperator

Der Fragezeichenoperator ist eine extreme Kurzform einer Kontrollstruktur. Auch hier lautet meine Empfehlung: Wegen seiner schlechten Lesbarkeit sollte der einzige dreistellige Operator möglichst nicht verwendet werden. Ein Beispiel für die Überprüfung eines Ergebnisses:

```
//CD/examples/ch04/ex48
(...)
boolean  checked;
checked  =  false;  //  Nicht  geprueft
char  state;  //  Erfolgreich?
state  =  (checked)  ?  (state  =  '+')  :  (state  =  '-');  //
Kurzform
System.out.println("Status:  "  +  state);
  (...)
```

Listing 4.49 Kurz- und Langform eines Ausdrucks

Das Programm gibt folgendes Ergebnis aus:

```
Status:  -
```

Zuerst prüft das Beispielprogramm, ob die Variable *checked* den Wert *true* besitzt. Falls das der Fall ist, weist sie der Variablen *state* das Minuszeichen zu, falls nicht, das Minuszeichen.

4.7.7 New-Operator

Zum Erzeugen von Objekten dient ein Operator, den Sie auch unter den Schlüsselbegriffen finden und der im Abschnitt über Klassen bereits erwähnt wurde. Er führt eine Operation aus, die dazu dient, ein neues Objekt zu erzeugen, und gehört deswegen auch zu den Operatoren.

```
//CD/examples/ch04/ex49
(...)
public  class  TestApp  {
    public  static  void  main(String[]  arguments)  {
        Rectangle  rect  =  new  Rectangle(1,  5);  //  Neues
Rechtseck  "rect"
(...)
```

Listing 4.50 Ein Beispiel für den Aufruf eines Konstruktors

4.7.8 Cast-Operator

Das Umwandeln eines Datentyps wird ausführlich in → Kapitel 7 behandelt. An dieser Stelle möchte ich nur den dazu notwendigen Operator der Vollständigkeit halber aufführen.

```
//CD/examples/ch04/ex50
(...)
int  a  =  30000;
int  b  =  2700;
short  result;
result  =  (short)  (a  +  b);
System.out.println("Ergebnis  =  "  +  result);
(...)
```

Listing 4.51 Eine Typkonvertierung von »int« nach »short«

Eine solche Typkonvertierung konvertiert natürlich nicht eine Variable und hebt die Deklaration auf. Das würde die Typsicherheit der Sprache Java untergraben. Eine Typkonvertierung bedeutet vielmehr nur, dass der Ausdruck *(a + b)*, der hier entstanden ist, *temporär* einen anderen Typ besitzt.

Es ist die ausdrückliche Anfrage des Programmierers, hier auf eigene Gefahr ein lokal begrenzte Umwandlung vorzunehmen. Diese Erlaubnis ist notwendig, da der short-Typ *result* und die int-Typen *a* sowie *b* andere Wertebereiche besitzen und somit inkompatibel sind.

4.8 Ausdrücke

Bis jetzt wurde in diesem Kapitel nur eine Menge relativ lebloser Datentypen und Operatoren vorgestellt. Um etwas Dynamik in Ihre Programme zu bringen, müssen Sie die bisher bekannten Bausteine zu größeren Einheiten kombinieren und den Ablauf steuern. Sie benötigen Anweisungen, Zuweisungen, Schleifen – kurz all das, was man unter Ausdrücken versteht.

4.8.1 Zuweisungen

Zuweisungen haben Sie zuhauf in Programmlistings dieses Kapitels gesehen, ohne dass der Fachbegriff dafür gefallen ist. Die Zuweisung

```
height  =  1
```

bewirkt, dass der Computer die nachfolgende Zahl 1 in eine Speicherzelle mit dem Namen *height* schreibt (→ Abbildung 4.13). Das Gleichheitszeichen ist einer der Java-Operatoren und hat eine vergleichbare Wirkung wie eine Methode. Das Zeichen ist für den Computer also nichts anderes als die Kurzschreibweise einer Funktion, die in diesem Fall bewirkt, dass die Speicherzelle namens *height* den Wert 1 bekommt.

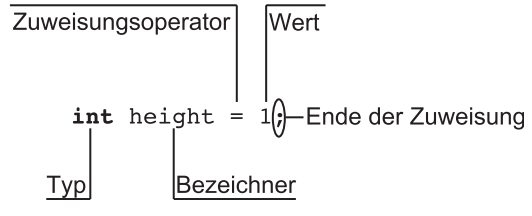


Abbildung 4.13 Die Zuweisung des Wertes 1

Java-Zuweisung ≠ mathematische Gleichung

Wenn Sie beginnen, einen Computer zu programmieren, ist es extrem wichtig, diese Form der Zuweisung genau zu verstehen. Auf der linken Seite der Zuweisung stehen immer Programmteile, die verändert werden. Auf der rechten Seite stehen die unveränderlichen Teile des Programms. Die Richtung, in der das Programm abgearbeitet wird, ist gegen alle westlichen Sitten und Gebräuche von rechts nach links (→ Abbildung 4.14).

```
height    =    1;
```

Abbildung 4.14 Die Zuweisung erfolgt von rechts nach links.

Das ist aber nicht das einzige Paradoxon. Die Zuweisung *height = 1* scheint eine mathematische Gleichung zu sein – ein Irrtum, der durch das Gleichheitszeichen hervorgerufen wird. Zum Vergleich: In der Programmiersprache Pascal sähe die Zuweisung so aus: *height := 1*.

Ist x = y gleich y = x?

In Pascal ist der Zuweisungsoperator zweistellig, weil der Erfinder der Sprache verhindern wollte, dass man den Operator mit dem mathematischen Gleichheitszeichen verwechselt. Es sollte unmöglich sein, dass jemand auf den Gedanken kommt *1 = height* statt *height = 1* zu schreiben; das ist in Java nicht gestattet, da auf der linken Seite variable Bezeichner stehen müssen.

Aber wie ist es, wenn auf beiden Seiten Variablen stehen? Ist *x = y* das Gleiche wie *y = x*? – In der Mathematik auf jeden Fall. In Java jedoch nicht. Im ersten Fall weist das Programm den Wert der Variablen *y* der Speicherzelle *x* zu. Im zweiten Fall ist es umgekehrt: Die Speicherzelle y bekommt den Wert von x vorgesetzt.

Ein Programmbeispiel (→ Listing 4.52) zeigt das deutlich. Es gibt Folgendes aus: Fall 1: x = 5; y = 5 und Fall 2: x = , y = 1. Das war nicht anders zu erwarten, weil der Computer im Fall 1 den Wert der Speicherzelle y in die Speicherzelle x kopiert hat. Im Fall 2 hingegen hat die Speicherzelle y den Wert der Speicherzelle x bekommen.

```
//CD/examples/ch04/ex51
(...)
int x; // Deklaration x
int y; // Deklaration y
// Fall 1:
x = 1; // x mit 1 initialisiert
y = 5; // y mit 5 initialisiert
x = y; // x bekommt den Wert von y
System.out.println("Fall 1: x = " + x + "; y = " + y);
// Fall 2:
x = 1; // x erneut mit 1 initialisiert
y = x; // y bekommt den Wert von x
```

```
System.out.println("Fall 2: x = " + x + "; y = " + y);
(...)
```

Listing 4.52 Der Ausdruck x = y ist keineswegs gleich y = x.

Die Sprache Java verhält sich anders als die mathematische Sprache. Mathematisch wäre die ganze Aktion vollkommen sinnlos, denn aus x = 1 und y = 5 folgt nicht x = y. Die letzte Aussage ist mathematisch gesehen nicht wahr: x ist nicht gleich y, da x = 1 gleich y = 5 eine falsche Aussage ist.

Die Quintessenz dieses Beispiels zeigt, dass sich mathematische Formeln keinesfalls 1:1 in die Programmiersprache Java übertragen lassen. Sie müssen daher eine Reihe von Gesetzmäßigkeiten beachten, die Sie jetzt noch nicht benötigen, aber in → Kapitel 7 ausführlich kennen lernen werden.

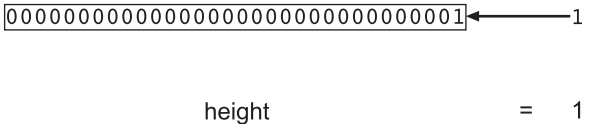


Abbildung 4.15 Zustand der Speicherzelle nach der Zuweisung des Wertes 1

Nach diesem kleinen Exkurs in die Gefilde der niederen Mathematik wieder zurück zum Programm mit der Flächenberechnung eines Rechtecks. Wie sieht die Speicherzelle *height* nach der Zuweisung aus? Sie besitzt, wie erwartet, einen neuen Wert. Der ursprüngliche Wert oooooooh ist überschrieben worden (→ Abbildung 4.15). Dass die Speicherzelle schon einen definierten Wert besitzt, unterscheidet Java von C und C++. Alle einfachen Datentypen besitzen schon einen Standardwert.

Je nachdem, wie man die Sprache Java strukturiert, kann man folgende Anweisungen unterscheiden:

- Elementare Anweisungen
- Verzweigungen
- Schleifen

4.8.2 Elementare Anweisungen

Block

Der Block ist eine Anzahl von zusammengehörenden Anweisungen. Sie werden nacheinander ausgeführt. Blöcke können lokale Variablen besitzen, die außerhalb des Blocks ihre Gültigkeit verlieren. In nachfolgenden Beispiel wird beispielsweise

ein char-Array deklariert und initialisiert. Es ist außerhalb des Blocks nicht sichtbar (→ Kapitel 7, 7.2.2).

```
//CD/examples/ch04/ex52
(...)
    {
        char block [] = {'B', 'l', 'o', 'c', 'k'};
        for(int i = 0; (i < block.length); i++)
            System.out.print(block[i]);
    }
    System.out.print("haus");
(...)
```

Listing 4.53 Ein Blockhaus

4.8.3 Verzweigungen

Verzweigungen dienen dazu, den Programmfluss zu steuern. Sie gehören daher wie die Schleifen zu den Kontrollstrukturen des Programms. Java hat aus C/C++ das Erbe der if- und der switch-Anweisung angetreten.

If-Verzweigung

Die If-Verzweigung des folgenden Beispiels kommt Ihnen vielleicht bekannt vor und ist tatsächlich fast eine Dublette des Beispiels 4.48 dieses Kapitels. Hier soll überprüft werden, ob der Wert *checked* gültig, das heißt wahr ist. Falls das der Fall ist, bekommt die Variable *state* das Pluszeichen zugewiesen, andernfalls das Minuszeichen.

```
//CD/examples/ch04/ex53
(...)
boolean checked;
checked = false; // Nicht geprueft
char state; // Erfolgreich?
if (checked)
      (state = '+');
else
      (state = '-');
System.out.println("Status: " + state);
checked = true; // Geprueft
if (checked) {
      (state = '+');
}
else {
      (state = '-');
```

```
}
System.out.println("Status:  "  +  state);
(...)
```

Listing 4.54 Zwei If-then-else-Konstrukte

Die Schleife wird in diesem Programm zweimal ausgeführt. Bei der zweiten Variante stehen die Anweisungen in geschweiften Klammern. Diese Klammern sind nur dann notwendig, wenn die nachfolgende Anweisung zu einem Block zusammengefasst werden soll.

Case-Verzweigung

Die Case-Anweisung ist praktisch, wenn man viele Möglichkeiten einer Verzweigung hat, die über eine If-Konstruktion zu umständlich zu lösen wäre. Allerdings darf die nach dem Schlüsselwort *switch* folgende Variable nur vom Typ *char*, *byte*, *short* oder *int* sein. Ein Wahrheitswert ist nicht erlaubt.

```
//CD/examples/ch04/ex54
(...)
checked  =  0;  //  Nicht  geprueft
char  state  =  '  ';  //  Erfolgreich?
switch  (checked)  {
      case  0:  state  =  '-';
          break;
      case  1:  state  =  '+';
          break;
}
(...)
```

Listing 4.55 Switch-Konstrukt

Soll eine Case-Anweisung verlassen werden, wenn eine Bedingung erfüllt ist, so *muss* sie mit einem *break* beendet werden. Das Beispielprogramm gibt ein Minuszeichen als Status aus; falls kein *break* verwendet wird, ist es ein Pluszeichen(!).

4.8.4 Schleifen

Schleifen dienen keineswegs zur Verzierung eines Java-Programms, sondern dazu, sich wiederholende Abläufe (Schleifen) zu verpacken. Es gibt drei Schleifentypen in Java:

- While-Schleife
- Do-Schleife
- For-Schleife

While-Schleife

Die Schleife gehört zum Typ der kopfgesteuerten Schleifen. Das → Listing 4.56 zeigt ein Beispiel für eine While-Schleife. Im Kopf der Schleife fragt das Programm ab, ob *lange* kleiner als *weile* ist. Ist das der Fall, wird die Schleife das erste Mal ausgeführt. Danach wiederholt sich der Vorgang, bis der Ausdruck im Schleifenkopf *true* ist.

```
//CD/examples/ch04/ex55
(...)
byte  lange,  weile;
lange  =  1;  //  Nicht  lange
weile  =  7;  //  Wie  lange
while  (lange  <  weile)  {
      lange++;
      System.out.println("Lange");
}
System.out.print("Langewhile");
(...)
```

Listing 4.56 Eine lange While-Schleife

Die While-Schleife ist abweisend, falls der Ausdruck im Schleifenkopf *true* sein sollte. Das bedeutet, dass die Schleife nicht durchlaufen wird, falls der Ausdruck im Schleifenkopf *true* ist.

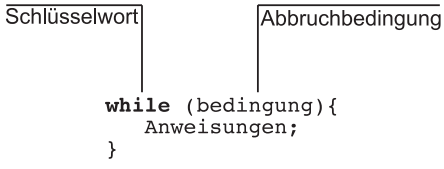


Abbildung 4.16 Aufbau der While-Schleife

Do-Schleife

Die Do-Schleife gehört zu der Schleifenart mit dem lustigen Namen »fußgesteuerte[4] Schleifen«. Sie wird folgendermaßen verwendet:

```
//CD/examples/ch04/ex56
byte  lange,  weile;
lange  =  1;  //  Nicht  lange
weile  =  1;  //  Wie  lange?
do  {
      lange++;
```

4 Ich hoffe, Programme mit diesem Schleifentyp sind trotzdem kopfgesteuert.

```
      System.out.println("Kurz");
}  while  (lange  <  weile);
System.out.print("Kurzwhile");
```

Listing 4.57 Eine kurze Do-Schleife

Dieser Schleifentyp prüft nicht vor dem ersten Durchlauf, ob der Wert des Ausdrucks *true* oder *false* ist. Obwohl im Schleifenfuß ein wahrer Ausdruck entsteht, kommt es trotzdem zu einem Durchlauf.

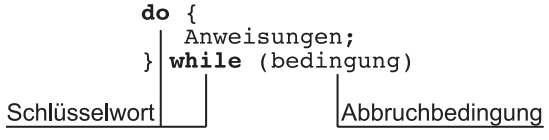


Abbildung 4.17 Aufbau der Do-Schleife

For-Schleife

Die For-Schleife gilt als die schnellste Schleifenart. In ihrem Kopf werden sämtliche Ablaufbedingungen festgelegt. Der erste Ausdruck bestimmt den Anfangswert, der zweite die Abbruchbedingung, und der dritte ist eine Anweisung zur Steuerung der Abbruchbedingung.

```
//CD/examples/ch04/ex57
byte  fort,  ran;
ran  =  5;  //  Wie  lange?
for  (fort  =  1;  fort  <=  ran;  fort++)  {
      System.out.print("For");
}
System.out.print("tran");
```

Listing 4.58 Ein Beispiel für eine For-Schleife

Wichtig ist, dass alle Schleifen mit *break* unterbrochen und mit *continue* wieder fortgesetzt werden können.

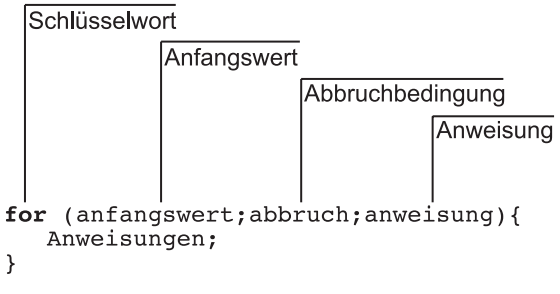


Abbildung 4.18 Aufbau der For-Schleife

4.9 Module

Um größere Softwaresysteme überschaubar zu halten, gibt es bei den verschiedenen Programmiersprachen Modulkonzepte. Ein Modul nennt sich Java Package (Paket). Es umfasst eine oder mehrere Java-Klassen.

4.9.1 Klassenimport

Diese Packages (Pakete) sind Gültigkeitsbereiche für Klassen, die sich in ihnen befinden (→ Kapitel 7, 7.2 Sichtbarkeit). Auch öffentliche Klassen sind so lange für andere Module unbekannt, bis sie über eine Importanweisung übernommen werden.

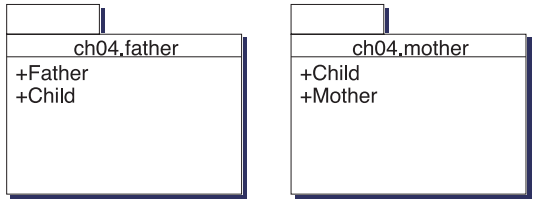


Abbildung 4.19 Die Klasse »Child« hat zwei Bedeutungen.

Ein Beispiel dazu: Stellen Sie sich eine Familie vor, die aus einer Mutter und einem Vater besteht, die in Trennung leben. In einen »Haus«, dem Package *mother*, lebt die Tochter, in dem anderen Package *father* der Sohn. Beide gehören zur Klasse *Child*.

Wie Sie an der → Abbildung 4.20 erkennen können, ist die Klasse *Child* zweimal vorhanden. Im linken Package *mother* hat sie die Bedeutung eines Kindes mit starken Beziehungen zur Mutter, im rechten Package *father* hingegen den eines Kindes mit schwachen Beziehungen zur Mutter.

In → Listing 4.60 erkennen Sie in Zeile drei eine Importanweisung. Durch diese Anweisung kann die Klasse *Child* von der Klasse *Mother* erben (*Child extends Mother*). Dazu muss das Package mit dem vollständigen Namen angegeben werden.

```
//CD/examples/ch04/ex58
package  ch04.mother;
public  class  Child  extends  Mother  {
(...)
}
```

Listing 4.59 Die Klasse »Child« also Teil des Packages »Mother«

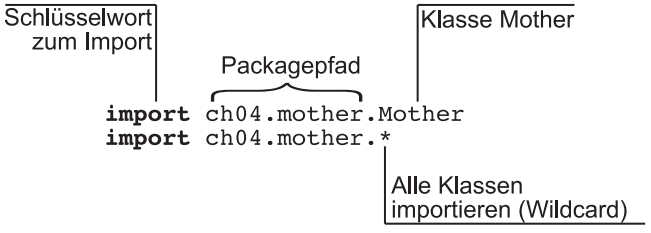


Abbildung 4.20 Aufbau der Importanweisung

Der Import von Klassen kann entweder einzeln für jede Klasse eines Packages ausgeführt werden oder für ein ganzes Package. Im letzteren Fall verwendet man eine Wildcard (→ Abbildung 4.20). Es hat einige Vorteile, jede Klasse einzeln zu importieren. Dadurch kann der Programmierer leichter nachvollziehen, welche Klasse verwendet wurde.

4.9.2 Namensräume

Java stört die doppelte Definition nicht, denn jede Klasse besitzt ihr eigenes Paket. Ein Paket nennt sich auch allgemein Namensraum. Er schränkt die Sichtbarkeit einer Klasse für andere Klassen ein (→ Kapitel 7, 7.2 Sichtbarkeit).

```
//CD/examples/ch04/ex58
package  ch04.father;
import  ch04.mother.Mother;
public  class  Child  extends  Mother  {
(...)
}
```

Listing 4.60 Die Klasse »Mother« wird importiert und erweitert.

4.10 Dokumentation

Kommentarzeichen dienen dazu, Teile des Quelltextes zu dokumentieren. Java verfügt sogar über drei verschiedene Kommentararten:

- Zeilenbezogene Kommentare
- Abschnittsbezogene Kommentare
- Dokumentationskommentare

4.10.1 Zeilenbezogene Kommentare

Dieser Kommentartyp wird durch doppelte Schrägstriche eingeleitet, die den Rest der Zeile als Kommentar markieren. Sie beziehen sich also jeweils nur auf eine einzelne Zeile (→ Listing 4.61).

```
//CD/examples/ch04/ex60
(...)
// Zeilenbezogener Kommentar vor einer Anweisung
Anweisungen    // Zeilenbezogener Kommentar hinter einer
Anweisung
(...)
```

Listing 4.61 Zeilenbezogene Kommentare

4.10.2 Abschnittsbezogene Kommentare

Im Gegensatz dazu lassen sich mit abschnittsbezogenen Kommentarzeichen weite Teile für den Compiler ausblenden und als Kommentar markieren. Sie werden wie in C mit einem Schrägstrich, der von einem Asterisk gefolgt ist, begonnen und enden in der umgekehrten Reihenfolge (→ Listing 4.62).

```
//CD/examples/ch04/ex60
(...)
//*  Dieser  Kommentar
      erstreckt
      sich  ueber
      mehrere  Zeilen  *//
(...)
```

Listing 4.62 Ein abschnittsbezogener Kommentar

Der abschnittsbezogene Kommentar kann aber auch dazu verwendet werden, mitten im Quelltext Kommentare einzufügen.

```
(...)
//* Dieser Kommentar bezieht sich auf einen Abschnitt *//
Anweisungen
(...)
```

Listing 4.63 Ein weiterer abschnittsbezogener Kommentar

4.10.3 Dokumentationskommentare

Dieser interessante Kommentartyp dient dazu, aus Kommentaren, die im Quelltext eingefügt werden, HTML-Dokumente zu erzeugen. Auch diese Kommentare können sich über mehrere Zeilen erstrecken, enden wie die abschnittsbezogene Kommentare, beginnen aber mit einem zusätzlichen Asterisk (→ Listing 4.64).

```
//**
  *  Projekt:    Transfer
  *  Beschreibung:  Backup-Programm
```

```
  *  @Copyright  (c)  2000  -  2003  by
  *  @author  Bernhard  Steppan
  *  @version  1.0
*//
```

Listing 4.64 Beispiel für einen Dokumentationskommentar

Es gibt Java-Werkzeuge, die aus den Dokumentationskommentaren vollautomatisch Java-Dokumentation erzeugen können. Einzelheiten finden Sie in → Kapitel 5 (5.3 Konstruktionsphase) und → Kapitel 21.

4.11 Zusammenfassung

Die Sprache Java verfügt über einfache Datentypen, erweiterte Datentypen und benutzerdefinierte Datentypen. Die acht einfachen Datentypen sind Datentypen sind prozedurale Restbestände aus der verwandten Programmiersprache C. Sie sind keine Klassen, sondern nur Datenbehälter ohne Methoden.

Im Gegensatz dazu, sind Arrays vordefinierte Klassen. Arrays können ohne feste Länge deklariert werden, müssen aber zur Erzeugung eine feste Länge besitzen. Sie sind also halbdynamisch.

Es gibt drei Arten von benutzerdefinierten Datentypen in Java: konkrete und abstrakte Klassen sowie Interfaces. Während man von konkreten Klassen mit Hilfe des New-Operators Objekte erzeugen kann, lassen sich abstrakte Klassen und Interfaces nur erweitern.

Ausdrücke erlauben es, Variablen zu deklarieren, Werte zuzuweisen und den Fluss des Programms zu steuern. Java besitzt darüber hinaus noch drei Schleifenarten, die dazu dienen, wiederkehrende Abläufe zu verpacken.

4.12 Aufgaben

4.12.1 Fragen

1. Wann ist die Programmiersprache Java veröffentlicht worden?
2. Über welche Sprachelemente verfügt Java?
3. Wozu dient eine Deklaration?
4. Wie ist sie aufgebaut?
5. Was sind einfache Datentypen?
6. Wie unterscheiden sie sich von Klassen?
7. Wo liegen ihre Vorteile?

8. Was ist eine streng typisierte Sprache?
9. Warum sind Java-Arrays halbdynamisch?
10. Was ist ein benutzerdefinierter Datentyp?
11. Wozu benötigt man benutzerdefinierte Datentypen?
12. Welche Arten von Klassen gibt es in Java?
13. Wie kann man verhindern, dass von Klassen Objekte erzeugt werden?
14. Wozu dient ein Konstruktor?
15. Wie unterscheidet er sich von einer normalen Methode?
16. Wieso benötigt man Accessoren und Mutatoren?
17. Welche Bedeutung hat der Cast-Operator?
18. Wo liegt der Unterschied zwischen einer mathematischen Gleichung und einer Programmzuweisung?

4.12.2 Übungen

1. Schreiben Sie auf Basis von Beispiel 20 eine Klasse namens Circle.
2. Ergänzen Sie Circle um eine Objektvariable radius.
3. Ergänzen Sie Circle um eine Klassenvariable Pi.
4. Ergänzen Sie folgende Anweisungen um ein komplette Klasse mit main-Methode und berechnen Sie, was das Programm ausgeben wird.

```
i  =  10;
j  =  10;
j  =  i++;
System.out.println(i);
i  =  10;
j  =  10;
j  =  ++i;
System.out.println(i);
```

5. Berechnen Sie, was die Anweisung ausgeben wird.

```
boolean  i  =  true;
boolean  j  =  false;
System.out.println(i  ||  j);
```

6. Berechnen Sie, was die Anweisung ausgeben wird.

```
static  final  int  i  =  10;
i++;
System.out.println(i  ||  j);
```