
Bernhard Steppan
Inkl. Java-Profi-Buch auf CD
Einstieg in Java
Aktuell zum JDK 5
• Für Programmiereinsteiger und Java-Neulinge
• Mit vielen Beispielen und kommentierten Lösungen
• Verstehen, anwenden und nachschlagen
2. Auflage
Galileo Computing

Auf einen Blick

Inhalt

Vorwort

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

Liebe Leserin, lieber Leser!

Ich möchte mich bei allen Leserinnen und Lesern für die vielen Zuschriften, die konstruktiven Verbesserungsvorschlägen und Anregungen bedanken. Sie haben geholfen, dass die erste Auflage dieses Buchs ein großer Erfolg wurde und in diese Neuauflage viele Verbesserungen und Anregungen eingeflossen sind.

Neben diesen Verbesserungen habe ich die wesentlichen Neuerungen von Java 5 (alias JDK 1.5) und einige neue Beispiele in das vorliegende Buch eingearbeitet. Alle Beispielprogramme wurden neben den anderen Entwicklungsumgebungen (JBuilder, Together) zudem mit der neuesten Version 3.0 von Eclipse mit den Erweiterungen zu Java 5 übersetzt.

Aufbau des Buchs

Dieses Buch besteht aus fünf Teilen. 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 im Quelltext und als ausführbare Dateien für Win-

dows, 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. Um Ihnen die Arbeit mit Entwicklungs-Tools etwas zu erleichtern, enthält dieses Buch trotzdem im Anhang ein Kapitel zu 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 anfordern. 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
Java-Bezeichner wie Variablen, Methoden und Klassen	*Kursivschrift*
Programmbeispiel auf der CD: hier Kapitel 04, Beispiel 1	`//CD/examples/ch04/ex01`
Dateinamen, Pfadangaben und Programmausgaben	`Schrift mit fester Zeichenbreite`
Querverweise auf andere Buchteile (Abbildungen, Listings, Kapitel etc.)	→

Tabelle 1 Verwendete Schriftkonventionen

Manche Quelltexte sind aus Platzgründen nicht komplett im Buch abgedruckt, sondern nur die zum Verständnis wichtigen Teile. Die Stellen der Quelltexte, bei denen Teile ausgelassen wurden, habe ich mit einem Scherensymbol (✂) gekennzeichnet. Sie finden die Programme vollständig auf der beiliegenden CD.

Errata

Leider lässt sich trotz größter Sorgfalt nicht immer vermeiden, dass der ein oder andere Fehler im Buch oder in den Beispielprogrammen verbleibt. Aus diesem Grund finden Sie auf der Website des Verlags und meiner Website (**http://www.steppan.net**) wie schon bei der Erstauflage eine Liste der bekannten Fehler (Errata) und (falls notwendig) aktualisierte Beispielprogramme. Sie können diese Aktualisierungen selbstverständlich kostenfrei herunterladen.

Danksagung

Ich möchte mich herzlich bei meiner Frau Christiane bedanken, die mich auch bei dieser Neuauflage wieder sehr motiviert hat. Meinem Sohn Jonathan danke ich für die Vorschläge zu den neuen Graphiken des Beispielprogramms »Memory«, meiner Lektorin Judith Stevens-Lemoine für die unkomplizierte Zusammenarbeit, Dr. Rainer Noske für das Korrekturlesen des Manuskripts sowie Katrin Volkmann für die sorgfältige Fachkorrektur und die zahlreichen wertvollen Verbesserungsvorschläge.

Schreiben Sie mir, ...

wenn Sie Fragen, konstruktive Kritik oder Verbesserungsvorschläge haben. Jedes Buch lebt vom Dialog mit seinen Lesern. Deshalb sind mir Ihre Anregungen sehr wichtig. Richten Sie bitte Ihre Post an **bernhard@steppan.net** oder den Verlag Galileo Press. Ich wünsche Ihnen nun viel Spaß bei der Lektüre des vorliegenden Buchs und viel Erfolg bei der Entwicklung Ihrer Java-Programme!

Bernhard Steppan
Bad Homburg, im September 2004

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

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.

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

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 ...

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("Fl\u00e4che = " + 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 4). 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 einer Textausgabe mit der Verwendung von Unicode (Punkt 9) zusammen.

Programmstart

Das Package und die Klasse sollen Sie zunächst nicht interessieren, sondern nur der Programmstart, der von der Methode *main()* eingeleitet wird. Diese Methode beginnt mit drei Typdeklarationen. Diese Deklarationen legen fest, welche Datentypen die Variablen *height*, *width* und *area* bekommen, und reserviert für diese Speicherplatz.

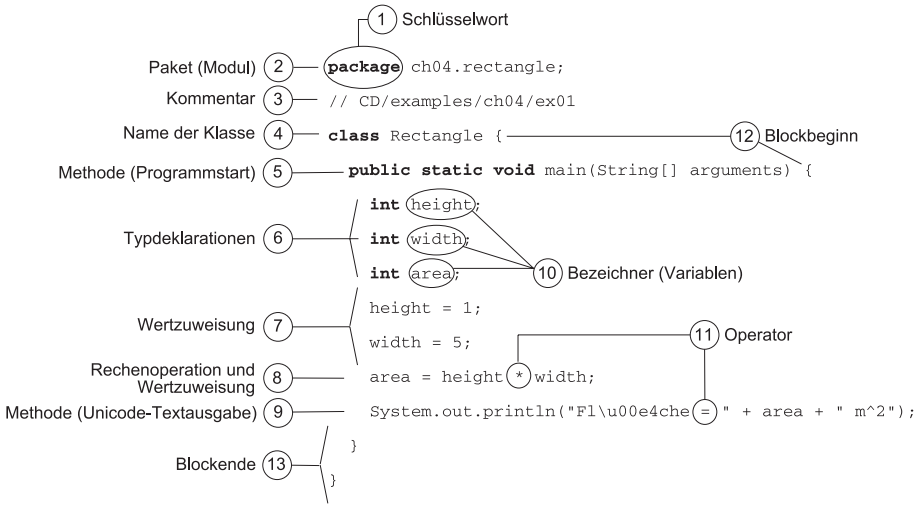

Abbildung 4.1 Übersicht über die wichtigsten Java-Sprachelemente

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. Damit unter allen Betriebssystemen der deutschen Umlaut des Wortes Fläche korrekt ausgegeben wird, musste das »ä« als Unicode kodiert werden.

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)

So weit 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 53 existieren (→ Tabelle 4.1).

abstract	assert	boolean	break	byte
case	catch	char	class	const[1]
continue	default	do	double	else
enum	extends	false	final	finally
float	for	goto1	if	implements
import	instanceof	int	interface	long
native	new	null	package	private
protected	public	return	short	static
strictfp	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while		

Tabelle 4.1 Schlüsselwörter der Sprache Java

Das Schlüsselwort *enum* können Sie erst ab der neuen Java-Version 5.0 (JDK 1.5) verwenden. Im Gegensatz dazu lassen sich die Schlüsselwörter *const* und *goto* niemals verwenden. Die Java-Erfinder haben sie reserviert, aber ihre Verwendung nicht gestattet – warum?

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

1 Ist reserviert, wird aber nicht benutzt.

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:

- Überwachung des Programmzustands mit Vor- und Nachbedingungen (*assert*)
- 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.

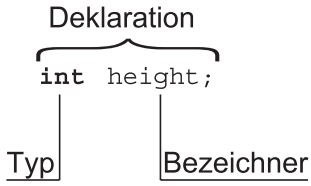

Abbildung 4.2 Deklaration der Variablen »height«

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

Tabelle 4.2 Übersicht der einfachen Java-Datentypen

Typ	Speicherplatz [byte]	Wertebereich	Standardwert
`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 (Forts.)

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).

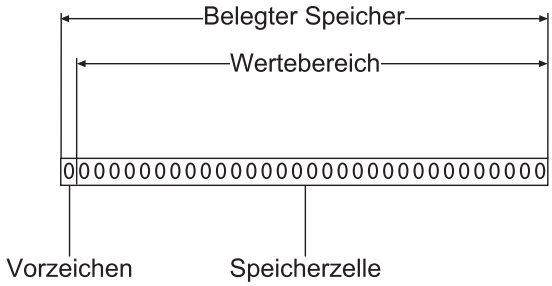

Abbildung 4.3 Wertebereich und belegter Speicher bei Zahlendatentypen am Beispiel des Datentyps »int«

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

Anders sieht es beim Datentyp *byte* aus. Hier 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 solche Zahl nur bis zu einer gewissen Nachkommastelle exakt ist, alles

andere fällt unter den Tisch. Man spricht in diesem Fall von so genannten Rundungsfehlern.

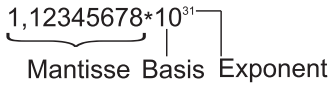

Abbildung 4.4 Die Stellen der Mantisse bestimmen die Genauigkeit.

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.

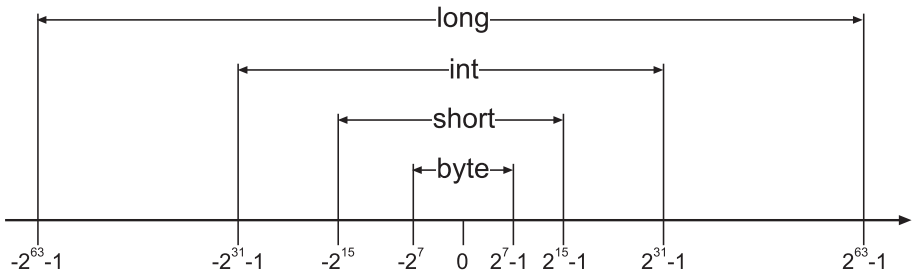

Abbildung 4.5 Wertebereich der Ganzzahltypen (nichtlineare Darstellung)

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("Fl\u00e4che = " + 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("Fl\u00e4che = " + 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("Fl\u00e4che = " + 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("Fl\u00e4che  =  "  +  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).

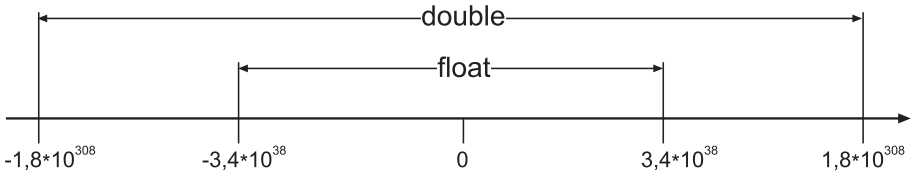

Abbildung 4.7 Wertebereich der Gleitkommatypen (gerundete Werte, nichtlineare Darstellung)

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("Fl\u00e4che = " + 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("Fl\u00e4che = " + 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 \u00fcberpr\u00fcft = " +
   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 Erweiterte 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
    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[0] + yesKey[1] + "]");
    System.out.println("<Abbrechen> ... [" +
            cancelKey[0] + cancelKey[1] + cancelKey[2] +
            cancelKey[3] + cancelKey[4] + cancelKey[5] +
            cancelKey[6] + cancelKey[7] + cancelKey[8] + "]");
    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
✂
// 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
✂
char  yesKey[]  =  {'J',  'A'};
char  cancelKey  []  =
   {'A',  'B',  'B',  'R',  'E',  'C',  'H',  'E',  'N'};
char  helpKey  =  '?';
System.out.println("<Ja> ..... [" +
     yesKey[0] + yesKey[1] + "]");
System.out.println("<Abbrechen> ..... [" +
     cancelKey[0] + cancelKey[1] + cancelKey[2] +
     cancelKey[3] + cancelKey[4] + cancelKey[5] +
     cancelKey[6] + cancelKey[7] + cancelKey[8] + "]");
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("<Ja> ..... [" +
     yesKey[0] + yesKey[1] + "]");
System.out.println("<Abbrechen> ..... [" +
     cancelKey[0] + cancelKey[1] + cancelKey[2] +
     cancelKey[3] + cancelKey[4] + cancelKey[5] +
     cancelKey[6] + cancelKey[7] + cancelKey[8] + "]");
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

In Java-Kreisen wurde lange über die Notwendigkeit eines Aufzählungstyps diskutiert. Nun ist es Tatsache: Ab Java 5.0 (JDK 1.5) verfügt die Sprache endlich auch über diesen Datentyp mit dem Namen *enum*. Er dient vorwiegend dazu, Sammlungen von Konstanten zusammenzufassen, zum Beispiel die Tage einer Woche, wie folgendes Beispiel zeigt.

```
//CD/examples/ch04/ex15
package ch04.week;
public class Week {
    private enum DaysOfTheWeek {
       Montag, Dienstag, Mittwoch,
       Donnerstag, Freitag, Samstag, Sonntag}
    public static void main(String[] args) {
       System.out.println("Die Tage einer Woche:");
       for (DaysOfTheWeek day : DaysOfTheWeek.values()) {
           System.out.println(day);
       }
    } // main
} // Week
```

Listing 4.15 Beispiel für die Verwendung des neuen enum-Typs

Das Programm legt eine neue Aufzählung namens Woche mit sieben Konstanten (*Montag*, *Dienstag* etc.) an. Danach werden die Namen der Tage über eine so genannte Schleife nach und nach ausgegeben. Die genaue Erklärung der For-Schleife erfolgt in einem späteren Abschnitt dieses Kapitels (→ 4.10.4 Schleifen). An dieser Stelle soll nur wichtig sein, dass Sie hiermit folgende Liste erhalten:


```
Die Tage einer Woche:
Montag
Dienstag
Mittwoch
Donnerstag
Freitag
Samstag
Sonntag
```

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).

Vierklassengesellschaft

Es gibt vier Arten von Klassen in Java:

- Konkrete Klassen
- Abstrakte Klassen
- Interfaces
- Generische Klassen

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 (→ 4.7.7).

```
//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

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.

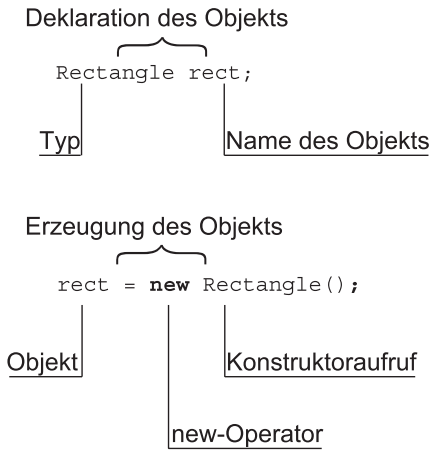

Abbildung 4.8 Deklaration und Erzeugung des Objekts »rect«

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

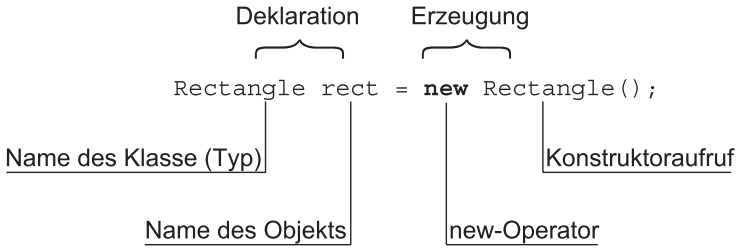

Abbildung 4.9 Kombination von Deklaration und Erzeugung eines Objekts

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 werden, 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).

Der zugehörige Java-Code sieht so aus:

```
//CD/examples/ch04/ex21
package ch04.shapes;
```

```
public abstract class  Shape  {
    private  int  height;
    private  int  width;
    public  Shape()  {
    }
}
```

Listing 4.21 Die Klasse »Shape« als abstrakte Klasse

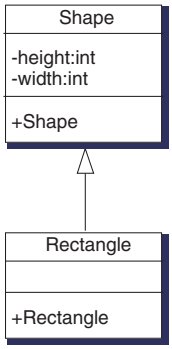

Abbildung 4.10 Die Klasse »Rectangle« erweitert »Shape«.

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.

Es gibt 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.

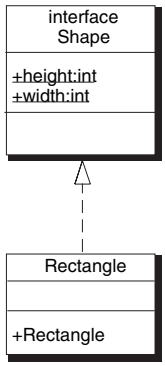

Abbildung 4.11 Die Klasse »Rectangle« implementiert das Interface »Shape«.

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).

4.5.4 Generische Klassen

Waren abstrakte Klassen und Interfaces für den Einstieg in die Java-Programmierung eigentlich schon mysteriös genug, so stellen die generischen Klassen (Generics) noch eine weitere Steigerung dar. Diese Klassen sind vorwiegend dazu da, beliebige Objekte aufnehmen zu können – eine Art universeller Behälter also.

Die Berechtigung für solche Klassen liegt auf der Hand. Angenommen, Sie wollen einen Tresor konstruieren, der alle Arten von Wertsachen aufbewahren kann, wie müssen Sie vorgehen? Ungeschickt wäre, für jede Wertsache eine eigene Klasse zu entwickeln. Stattdessen benötigen Sie einen generisch (universellen) verwendbaren Behälter – daher auch die englische Bezeichnung *Generics*.

```
//CD/examples/ch04/ex23
package  ch04.safe;
class  Safe<T> {
    private T valueable;
    public  setValueable(T valueable)  {
      this.valueable = valueable;
    }
}
✂
```

Listing 4.24 Die Klasse »Safe« nimmt Wertsachen der unterschiedlichsten Arten auf.

Beim Beispiel in → Listing 4.24 sehen Sie, wie eine solche Klassen definiert ist. Nach der Klassenbezeichnung folgt in gespitzten Klammern der Name des Stellvertreters für einen beliebigen Objekttyp. Erst bei der Erzeugung legen Sie sich fest, wozu Sie den Tresor verwenden wollen.

```
//CD/examples/ch04/ex23
✂
Safe<Money> safe = new Safe<Money>();
moneyBox.setValueable(new Money("400,53 Euro"));
✂
```

Listing 4.25 Die Verwendung der generischen Klasse »Safe«.

4.6 Variablen

Objektvariablen

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

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.

4.7 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 Konstante 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.

4.8 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 Zeit-

punkt, 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.

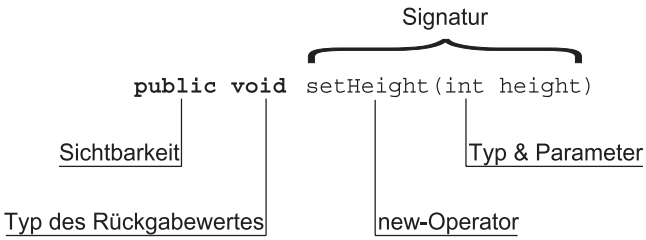

Abbildung 4.12 Die Signatur einer Methode

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

Unter der Signatur einer Methode versteht man ihren Namen und ihre Parameterliste (→ Abbildung 4.12).

Rumpf einer Methode

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

4.8.1 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/ex24
package ch04.shapes;
class Rectangle implements Shape {
    private int height;
    private int width;

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

Listing 4.26 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.

4.8.2 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/ex25
✂
class Rectangle implements Shape {
  protected void finalize() {
    // Anweisungen ...
  }
}
```

Listing 4.27 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.

4.8.3 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 haben sich die Ausdrücke Zugriffsmethoden, Getter-Methoden oder Accessoren etabliert.

```
//CD/examples/ch04/ex26
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 4.28 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 Bezeichners.

4.8.4 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/ex27
✂
public void setHeight(int height) {
        this.height = height;
    }
    public void setWidth(int width) {
        this.width = width;
    }
✂
```

Listing 4.29 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*.

4.8.5 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.

```
public static void main(String[] arguments) {
// Anweisungen
```

Listing 4.30 Die Startmethode eines Programms

4.9 Operatoren

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

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

4.9.1 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: j = i 2. Schritt: i = i + 1
––	Prädekrement	j = ––i	1. Schritt: i = i – 1 2. Schritt: j = i
––	Postdekrement	j = i––	1. Schritt: j = i 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/examples/ch04/ex28
✂
int  height;
int  width;
int  area;
height  =  +1;
width  =  +5;
✂
```

Listing 4.31 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/examples/ch04/ex29
✂
int  height;
int  width;
int  area;
height  =  -1;
width  =  -5;
✂
```

Listing 4.32 Die Variablen »height« und »width« mit negativen Vorzeichen

Summe

Der Additionsoperator bewirkt die Summenbildung der benachbarten Variablen.

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

Listing 4.33 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 4.34 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 4.35 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 einer Division ganzer Zahlen als *int*-Werte speichern, wenn die Variablen (Operanden) nicht ausdrücklich anders deklariert wurden. In diesem Fall muss der Typ des Ergebnisses konvertiert werden. Auf dieses Thema geht → Kapitel 7 ausführlich ein.

```
//CD/examples/ch04/ex33
✂
float div;
div = 1 / 5; // Fehler durch interne Verarbeitung als int-Wert
System.out.println("Division (Fall 1) = " + div + " m");
div = 1F / 5F; // Korrekt durch Deklaration
System.out.println("Division (Fall 2) = " + div + " m");
div = (float) 1 / 5; // Korrekt durch Casting
```

```
System.out.println("Division (Fall 3) = " + div + " m");
✂
```

Listing 4.36 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 4.37 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 = height
System.out.println("H\u00f6he = " + height + " m");
System.out.println("Ergebnis = " + result + " m");
✂
```

Listing 4.38 Der Präinkrement-Operator

Das Programm gibt Folgendes aus:

```
Höhe = 2 m
Ergebnis = 2 m
```

Postinkrement

Beim Postinkrement-Operator verhält es sich entgegengesetzt. Er weist den Wert der Variablen *height* im ersten Schritt 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("H\u00f6he  =  "  +  height  +  "  m");
System.out.println("Ergebnis    =  "  +  result  +  "  m");
✂
```

Listing 4.39 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 im ersten Schritt 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("H\u00f6he  =  "  +  height  +  "  m");
System.out.println("Ergebnis    =  "  +  result  +  "  m");
✂
```

Listing 4.40 Der Prädekrement-Operator

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

Postdekrement

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

```
//CD/examples/ch04/ex38
✂
int  height;
int  result;
height  =  1;
result  =  height--; // 1.) result = height 2.) height =
height - 1;
System.out.println("H\u00f6he  =  "  +  height  +  "  m");
System.out.println("Ergebnis  =  "  +  result  +  "  m");
✂
```

Listing 4.41 Der Postdekrement-Operator

Das Ergebnis des Programms sind auch diesmal unterschiedliche Werte:

```
Höhe = 0 m
Ergebnis = 1 m
```

4.9.2 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 != j	Vergleich auf Ungleichheit
<	Kleiner	i < j	Vergleich auf kleiner
<=	Kleiner gleich	i <= j	Vergleich auf kleiner oder gleich
>	Größer	i > j	Vergleich auf größer
>=	Größer gleich	i >= j	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 4.42 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 4.43 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 4.44 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 4.45 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 4.46 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 4.47 Vergleich, ob Werte größer oder gleich sind

Der Vergleich führt zu folgendem Ergebnis:

```
false
true
```

4.9.3 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 4.48 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 1.7. 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 4.49 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 4.50 Diese Oder-Verknüpfung liefert ein wahres Ergebnis.

4.9.4 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 Bitweise Operatoren

4.9.5 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
-=	Subtraktionszuweisung	i -= 1	i = i – 1
*=	Produktzuweisung	i *= 1	i = i * 1
/=	Divisionszuweisung	i /= 1	i = i / 1
%=	Modulozuweisung	i %= 1	i = i % 1

Tabelle 4.7 Zuweisungsoperatoren

Operator	Bezeichnung	Beispiel	Erläuterung
&=	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.

4.9.6 Fragezeichenoperator

Der Fragezeichenoperator ist eine extreme Kurzform einer Verzweigung (→ 4.8.3). 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 = '-');
System.out.println("Status: " + state);
✂
```

Listing 4.51 Der Fragezeichenoperator ersetzt eine Verzweigung.

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. Die Langform des Programms finden Sie in → Listing 4.56 auf Seite 135.

4.9.7 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 Rechteck "rect"
✂
```

Listing 4.52 Ein Beispiel für den Aufruf eines Konstruktors

4.9.8 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 4.53 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 eine 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.

4.10 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.

4.10.1 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.

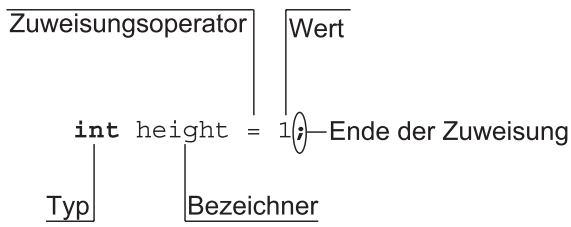

Abbildung 4.13 Die Zuweisung des Wertes 1

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).

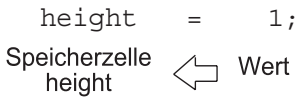

Abbildung 4.14 Die Zuweisung erfolgt von rechts nach links.

Das ist aber nicht das einzige Paradox. 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 4.54) zeigt das deutlich. Es gibt Folgendes aus: Fall 1: x = 5; y = 5 und Fall 2: x = 1, 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 4.54 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.

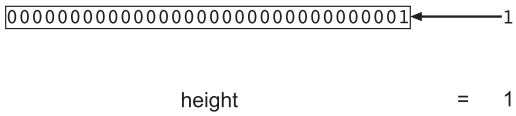

Abbildung 4.15 Zustand der Speicherzelle nach der Zuweisung des Wertes 1

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 00000000h 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

4.10.2 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 nachfolgendem 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 4.55 Ein Blockhaus

4.10.3 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 4.51 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 4.56 Zwei If-then-else-Konstrukte

Die Anweisungen werden in diesem Programm zweimal mit unterschiedlichem Ergebnis ausgeführt, da der Wert der Variable *checked* vor jeder Ausgabe verändert wird.

Case-Verzweigung

Wenn man sehr viele Möglichkeiten einer Programmverzweigung hat, werden If-Konstruktionen als Lösung schnell unübersichtlich. Als Ersatz bietet sich dann die Case-Anweisung an. Allerdings darf die nach dem Schlüsselwort *switch* folgende Variable nur vom Typ *char*, *byte*, *short*, *int* oder *enum* sein. Ein Wahrheitswert ist beispielsweise nicht erlaubt.

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

Listing 4.57 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 (!).

4.10.4 Schleifen

Schleifen dienen keineswegs zur Verzierung eines Java-Programms, sondern dazu, sich wiederholende Abläufe, so genannte 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 4.58 Eine lange While-Schleife

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

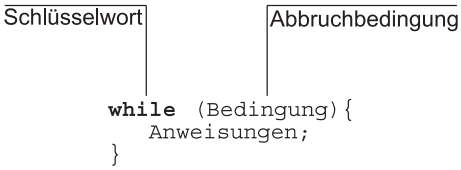

Abbildung 4.16 Aufbau der While-Schleife

Do-Schleife

Die Do-Schleife gehört zu der Schleifenart mit dem lustigen Namen »fußgesteuerte[1] Schleifen«. Sie wird folgendermaßen verwendet:

```
//CD/examples/ch04/ex56
✂
byte lange, weile;
lange = 1; // Nicht lange
weile = 1; // Wie lange?
do {
     lange++;
     System.out.println("Kurz");
} while (lange < weile);
System.out.print("Kurzwhile");
✂
```

Listing 4.59 Eine kurze Do-Schleife

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

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

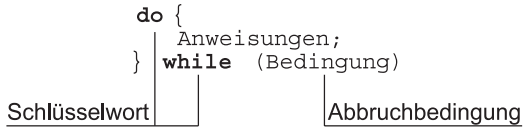

Abbildung 4.17 Aufbau der Do-Schleife

Einfache 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 4.60 Ein Beispiel für eine einfache For-Schleife

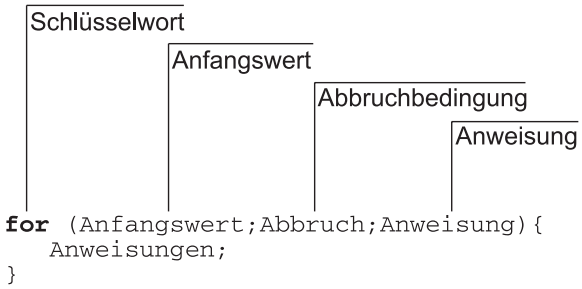

Abbildung 4.18 Aufbau der einfachen For-Schleife

Erweiterte For-Schleife

Manchen war die Programmierung einer For-Schleife in bestimmten Fällen zu umständlich. Das gilt vor allen dann, wenn Felder abgegrast werden sollen. Aus

diesem Grund gibt es ab Java 5.0 (JDK 1.5) die erweiterte For-Schleife, die Sie schon bei den Aufzählungstypen kennen gelernt haben. An dieser Stelle möchte ich das Beispiel nochmals aus der Perspektive der Schleife beleuchten.


```
//CD/examples/ch04/ex58
package ch04.week;
public class Week {

    private enum DaysOfTheWeek {
       Montag, Dienstag, Mittwoch,
       Donnerstag, Freitag, Samstag, Sonntag}

    public static void main(String[] args) {
       System.out.println("Die Tage einer Woche:");
       for (DaysOfTheWeek day : DaysOfTheWeek.values()) {
           System.out.println(day);
✂
```

Listing 4.61 Ein Beispiel für eine erweiterte For-Schleife

Die erweiterte Form der Schleife benötigt keinen Index mehr. Es besteht nur aus Typ, Bezeichner und Feld. Übersetzt lautet die Schleife aus Listing 4.61: »Für alle Tage der Woche innerhalb der Liste aller Tage der Woche gebe den Namen ihres Tags aus.«

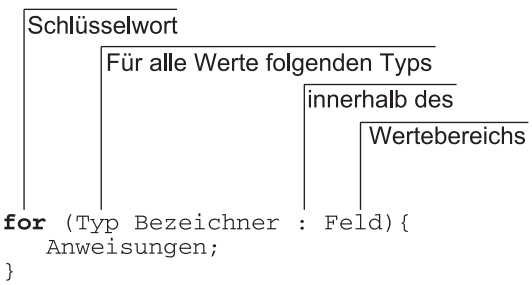

Abbildung 4.19 Aufbau der erweiterten For-Schleife

Als abschließende Bemerkung zu diesem Abschnitt ist wichtig, dass die For-Schleife wie alle Schleifen mit *break* unterbrochen und mit *continue* wieder fortgesetzt werden können.

4.11 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.

4.11.1 Klassenimport

Dynamische Importe

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.

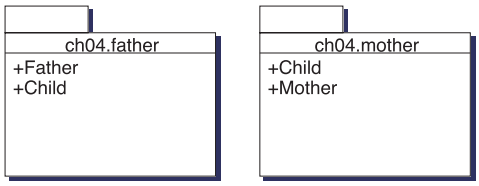

Abbildung 4.20 Die Klasse »Child« hat zwei Bedeutungen.

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/ex59
package  ch04.father;
import  ch04.mother.Mother;
public  class Child extends Mother {
✂
}
```

Listing 4.62 Die Klasse »Child« ist also Teil des Packages »Mother«.

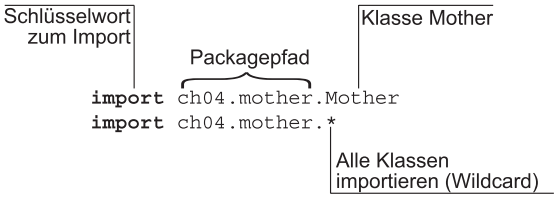

Abbildung 4.21 Aufbau der konventionellen Importanweisung

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.

Statische Importe

Bei den konventionellen Importanweisungen muss der Klassenbezeichner auch dann immer bei der Verwendung einer Methode vorangestellt werden, wenn diese statisch ist. Will man zum Beispiel mathematische Funktionen wie die Wurzeloperation anwenden, stört dies, wie folgendes Beispiel zeigt:

```
//CD/examples/ch04/ex60
package  ch04.imports;
import  java.lang.Math;
✂
  result = Math.sqrt(radicant);
✂
```

Listing 4.63 Der konventionelle Import einer Klasse

Der Grund für diese Schreibweise ist klar: Java gestattet nur die Definition von Methoden, die an eine Klasse gebunden sind. Methoden können nicht losgelöst von einer Klasse existieren. Und da es keine globalen Methoden gibt, muss auf sie immer in Verbindung mit der Klasse zugegriffen werden.

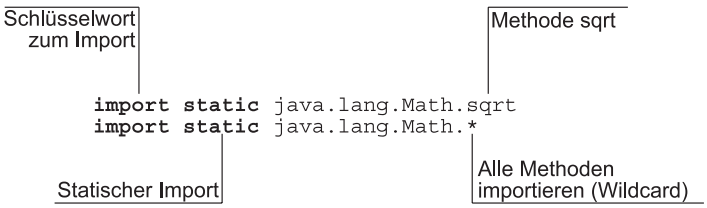

Abbildung 4.22 Aufbau eines statischen Imports

Das hat auch weiter Gültigkeit, nur dass sich die Schreibweise seit Java 5.0 durch statische Importe vereinfacht hat. Jetzt lassen sich auch Methoden einzeln oder über Wildcards statisch importieren.

```
//CD/examples/ch04/ex61
package  ch04.imports;
import static  java.lang.Math.sqrt;
✂
  result = sqrt(radicant);
✂
```

Listing 4.64 Der statische Import einer Methode verkürzt die Schreibweise.

4.11.2 Namensräume

Java stört die doppelte Definition von Klassen 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 7.7.1 Sichtbarkeit).

```
//CD/examples/ch04/ex62
package  ch04.father;
import  ch04.mother.Mother;
public  class  Child  extends  Mother  {
✂
}
```

Listing 4.65 Die Klasse »Mother« wird importiert und erweitert.

4.12 Dokumentation

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

- Zeilenbezogene Kommentare
- Abschnittsbezogene Kommentare
- Dokumentationskommentare

4.12.1 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 4.66).

```
✂
// Zeilenbezogener Kommentar vor einer Anweisung
Anweisungen  // Zeilenbezogener Kommentar hinter einer Anweisung
✂
```

Listing 4.66 Zeilenbezogene Kommentare

4.12.2 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, gefolgt von einem Asterisk, begonnen und enden in der umgekehrten Reihenfolge (→ Listing 4.67).

```
/* Dieser Kommentar
   erstreckt
   sich ueber
   mehrere Zeilen */
```

Listing 4.67 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 4.68 Ein weiterer abschnittsbezogener Kommentar

4.12.3 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 abschnittsbezogenen Kommentare, beginnen aber mit einem zusätzlichen Asterisk (→ Listing 4.69).

```
/**
  * Projekt: Transfer
  * Beschreibung: Backup-Programm
  * @Copyright (c) 2000 - 2004 by
  * @author  Bernhard Steppan
  * @version  1.1
*/
```

Listing 4.69 Beispiel für einen Dokumentationskommentar

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

4.13 Zusammenfassung

Die Sprache Java verfügt über einfache Datentypen, erweiterte Datentypen und benutzerdefinierte Datentypen. Die acht einfachen 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 vier Arten von benutzerdefinierten Datentypen in Java: konkrete und abstrakte Klassen, Interfaces sowie seit Java 5 generische Klassen (Generics). 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.

4.14 Aufgaben

4.14.1 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. Worin liegt der Unterschied zwischen einer mathematischen Gleichung und einer Programmzuweisung?
19. Was sind statische Importe und wozu verwendet man sie?

4.14.2 Übungen

1. Schreiben Sie auf Basis des Beispiels 20 eine Klasse namens Circle.
2. Ergänzen Sie *Circle* um eine Objektvariable *radius*.
3. Ergänzen Sie Circle um eine Konstante *Pi*.
4. Ergänzen Sie folgende Anweisungen um ein komplette Klasse mit einer Methode *main()* 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);
```

Die Lösungen zu den Aufgaben finden Sie in → Teil IV, Kapitel 19, ab Seite 513.

Index

D

E

K

L

M

S

T

U

V

W

X

Z