PEARSON Education

ADDISON-WESLEY

Guido Krüger

Handbuch der Java-Programmierung

4. Auflage

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

4 Datentypen

4.1 Lexikalische Elemente eines Java-Programms

Bevor wir uns in diesem Kapitel mit den Datentypen von Java befassen, sollen zunächst einmal die wichtigsten lexikalischen Eigenschaften der Sprache vorgestellt werden. Hierzu zählen der Eingabezeichensatz, die Kommentare und die Struktur von Bezeichnern.

4.1.1 Eingabezeichen

Ein Java-Programm besteht aus einer Folge von Unicode-Zeichen. Der Unicode-Zeichensatz fasst eine große Zahl internationaler Zeichensätze zusammen und integriert sie in einem einheitlichen Darstellungsmodell. Da die 256 verfügbaren Zeichen eines 8-Bit-Wortes bei weitem nicht ausreichen, um die über 30.000 unterschiedlichen Zeichen des Unicode-Zeichensatzes darzustellen, ist ein Unicode-Zeichen 2 Byte, also 16 Bit, lang. Der Unicode ist mit den ersten 128 Zeichen des ASCII- und mit den ersten 256 Zeichen des ISO-8859-1-Zeichensatzes kompatibel.

Die Integration des Unicode-Zeichensatzes geht in Java so weit, dass neben `String`- und `char`-Typen auch die literalen Symbole und Bezeichner der Programmiersprache im Unicode realisiert sind. Es ist daher ohne weiteres möglich, Variablen- oder Klassennamen mit nationalen Sonderzeichen oder anderen Symbolen zu versehen.

4.1.2 Kommentare

Es gibt in Java drei Arten von Kommentaren:

- *Einzeilige Kommentare* beginnen mit `//` und enden am Ende der aktuellen Zeile.
- *Mehrzeilige Kommentare* beginnen mit `/*` und enden mit `*/`. Sie können sich über mehrere Zeilen erstrecken.
- *Dokumentationskommentare* beginnen mit `/**` und enden mit `*/` und können sich ebenfalls über mehrere Zeilen erstrecken.

Kommentare derselben Art sind nicht schachtelbar. Ein Java-Compiler akzeptiert aber einen einzeiligen innerhalb eines mehrzeiligen Kommentars und umgekehrt.

Dokumentationskommentare dienen dazu, Programme im Quelltext zu dokumentieren. Mithilfe des Tools `javadoc` werden sie aus der Quelle extrahiert und in ein HTML-Doku-

ment umgewandelt (siehe Kapitel 50 auf Seite 1219). Kapitel 18 der Sprachspezifikation erklärt die Verwendung von Dokumentationskommentaren ausführlich. Wir wollen uns hier lediglich auf ein kleines Beispiel beschränken, das besagter Beschreibung entnommen wurde:

Listing 4.1: Verwendung eines Dokumentationskommentars im Java-API

```
001 /**
002  * Compares two Objects for equality.
003  * Returns a boolean that indicates whether this Object
004  * is equivalent to the specified Object. This method is
005  * used when an Object is stored in a hashtable.
006  * @param  obj     the Object to compare with
007  * @return         true if these Objects are equal;
008  *                 false otherwise.
009  * @see            java.util.Hashtable
010  */
011 public boolean equals(Object obj)
012 {
013   return (this == obj);
014 }
```

Dokumentationskommentare stehen immer *vor* dem Element, das sie beschreiben sollen. In diesem Fall ist das die Methode `equals` der Klasse `Object`. Der erste Satz ist eine Überschrift, dann folgt eine längere Beschreibung der Funktionsweise. Die durch `@` eingeleiteten Elemente sind Makros, die eine besondere Bedeutung haben. `@param` spezifiziert Methodenparameter, `@return` den Rückgabewert und `@see` einen Verweis. Daneben gibt es noch die Makros `@exception`, `@version` und `@author`, die hier aber nicht auftauchen.

Weitere Informationen zu `javadoc` und den anderen Hilfsprogrammen des JDK finden Sie in Kapitel 50 auf Seite 1219.

4.1.3 Bezeichner

Ein Bezeichner ist eine Sequenz von Zeichen, die dazu dient, die Namen von Variablen, Klassen oder Methoden zu spezifizieren. Ein Bezeichner in Java kann beliebig lang sein, und alle Stellen sind signifikant. Bezeichner müssen mit einem *Unicode-Buchstaben* beginnen (das sind die Zeichen 'A' bis 'Z', 'a' bis 'z', '_' und '$') und dürfen dann weitere Buchstaben oder Ziffern enthalten. Unterstrich und Dollarzeichen sollen nur aus historischen Gründen bzw. bei maschinell generiertem Java-Code verwendet werden.

Ein Buchstabe im Sinne des Unicode-Zeichensatzes muss nicht zwangsläufig aus dem lateinischen Alphabet stammen. Es ist auch zulässig, Buchstaben aus anderen Landessprachen zu verwenden. Java-Programme können daher ohne weiteres Bezeichner enthalten, die nationalen Konventionen folgen. Java-Bezeichner dürfen jedoch nicht mit Schlüsselwörtern, den booleschen Literalen `true` und `false` oder dem Literal `null` kollidieren.

4.1.4 Weitere Unterschiede zu C

Nachfolgend seien noch einige weitere Unterschiede zu C und C++ aufgelistet, die auf der lexikalischen Ebene von Bedeutung sind:

- Es gibt keinen Präprozessor in Java und damit auch keine `#define`-, `#include`- und `#ifdef`-Anweisungen.
- Der Backslash `\` darf nicht zur Verkettung von zwei aufeinander folgenden Zeilen verwendet werden.
- Konstante Strings, die mit `+` verkettet werden, fasst der Compiler zu einem einzigen String zusammen.

4.2 Primitive Datentypen

Java kennt acht elementare Datentypen, die gemäß Sprachspezifikation als *primitive Datentypen* bezeichnet werden. Daneben gibt es die Möglichkeit, Arrays zu definieren (die eingeschränkte Objekttypen sind), und als objektorientierte Sprache erlaubt Java natürlich die Definition von Objekttypen.

Im Gegensatz zu C und C++ gibt es die folgenden Elemente in Java jedoch nicht:

- explizite Zeiger
- Typdefinitionen (typedef)
- Recordtypen (struct und union)
- Bitfelder

Was auf den ersten Blick wie eine Designschwäche aussieht, entpuppt sich bei näherem Hinsehen als Stärke von Java. Der konsequente Verzicht auf zusätzliche Datentypen macht die Sprache leicht erlernbar und verständlich. Die Vergangenheit hat mehrfach gezeigt, dass Programmiersprachen mit einem überladenen Typkonzept (zum Beispiel PL/I oder ADA) auf Dauer keine Akzeptanz finden.

Tatsächlich ist es ohne weiteres möglich, die unverzichtbaren Datentypen mit den in Java eingebauten Hilfsmitteln nachzubilden. So lassen sich beispielsweise Zeiger zur Konstruktion dynamischer Datenstrukturen mithilfe von Referenzvariablen simulieren, und Recordtypen sind nichts anderes als Klassen ohne Methoden. Der Verzicht auf Low-Level-Datenstrukturen, wie beispielsweise Zeigern zur Manipulation von Speicherstellen oder Bitfeldern zur Repräsentation von Hardwareelementen, ist dagegen gewollt.

Alle primitiven Datentypen in Java haben eine feste Länge, die von den Designern der Sprache ein für allemal verbindlich festgelegt wurde. Ein `sizeof`-Operator, wie er in C vorhanden ist, wird in Java daher nicht benötigt und ist auch nicht vorhanden.

Ein weiterer Unterschied zu C und den meisten anderen Programmiersprachen besteht darin, dass Variablen in Java immer einen definierten Wert haben. Bei Membervariablen (also Variablen innerhalb von Klassen, siehe Kapitel 7 auf Seite 151) bekommt eine Variable einen Standardwert zugewiesen, wenn dieser nicht durch eine explizite Initialisierung geändert wird. Bei *lokalen* Variablen sorgt der Compiler durch eine Datenflussanalyse dafür, dass diese vor ihrer Verwendung explizit initialisiert werden. Eine Erläuterung dieses Konzepts, das unter dem Namen *Definite Assignment* in der Sprachdefinition beschrieben wird, ist Bestandteil von Kapitel 5 auf Seite 113. Tabelle 4.1 listet die in Java verfügbaren Basistypen und ihre Standardwerte auf:

Tabelle 4.1: Primitive Datentypen

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

4.2.1 Der logische Typ

Mit `boolean` besitzt Java einen eigenen logischen Datentyp und beseitigt damit eine oft diskutierte Schwäche von C und C++. Der `boolean`-Typ muss zwangsweise dort verwendet werden, wo ein logischer Operand erforderlich ist. Ganzzahlige Typen mit den Werten 0 oder 1 dürfen nicht als Ersatz für einen logischen Typen verwendet werden.

Literale

Der Datentyp `boolean` kennt zwei verschiedene Werte, nämlich `true` und `false`. Neben den vordefinierten Konstanten gibt es keine weiteren Literale für logische Datentypen.

4.2.2 Der Zeichentyp

Java wurde mit dem Anspruch entworfen, bekannte Schwächen bestehender Programmiersprachen zu vermeiden, und der Wunsch nach Portabilität stand ganz oben auf der Liste der Designziele. Konsequenterweise wurde der Typ `char` in Java daher bereits von Anfang an 2 Byte groß gemacht und speichert seine Zeichen auf der Basis des Unicode-Zeichensatzes. Als einziger integraler Datentyp ist `char` nicht vorzeichenbehaftet.

Da das Sprachdesign und das Java-API so gestaltet wurden, dass die Verwendung des Unicode-Zeichensatzes weitgehend transparent bleibt, ergeben sich für die meisten Entwickler zunächst kaum Umstellungsprobleme. Ein `char` oder `String` kann in Java genauso intuitiv benutzt werden wie in Sprachen, die auf dem ASCII-Zeichensatz aufbauen. Unterschiede werden vor allem dann deutlich, wenn Berührungspunkte zwischen der internen Unicode-Darstellung und der Repräsentation auf Systemebene entstehen, beispielsweise beim Lesen oder Schreiben von Textdateien.

Literale

`char`-Literale werden grundsätzlich in einfache Hochkommata gesetzt. Daneben gibt es `String`-Literale, die in doppelten Hochkommata stehen. Ähnlich wie C stellt Java eine ganze Reihe von Standard-Escape-Sequenzen zur Verfügung, die zur Darstellung von Sonderzeichen verwendet werden können:

Zeichen	Bedeutung
`\b`	Rückschritt (Backspace)
`\t`	Horizontaler Tabulator
`\n`	Zeilenschaltung (Newline)
`\f`	Seitenumbruch (Formfeed)
`\r`	Wagenrücklauf (Carriage return)

Tabelle 4.2: Standard-Escape-Sequenzen

Tabelle 4.2: Standard-Escape-Sequenzen (Forts.)

Zeichen	Bedeutung
\"	Doppeltes Anführungszeichen
\'	Einfaches Anführungszeichen
\\	Backslash
\nnn	Oktalzahl nnn (kann auch kürzer als 3 Zeichen sein, darf nicht größer als oktal 377 sein)

Weiterhin können beliebige Unicode-Escape-Sequenzen der Form `\uxxxx` angegeben werden, wobei `xxxx` eine Folge von bis zu 4 hexadezimalen Ziffern ist. So steht beispielsweise `\u000a` für die Zeilenschaltung und `\u0020` für das Leerzeichen.

Eine wichtiger Unterschied zu Standard-Escape-Sequenzen besteht darin, dass Unicode-Escape-Sequenzen an beliebiger Stelle im Programm auftauchen dürfen, also auch außerhalb von `char`- oder `String`-Literalen. Wichtig ist außerdem, dass diese bereits *vor* der eigentlichen Interpretation des Quelltextes ausgetauscht werden. Es ist also beispielsweise nicht möglich, ein `char`-Literal, das ein Anführungszeichen darstellen soll, in der Form '\u0027' zu schreiben. Da die Unicode-Sequenzen bereits vor dem eigentlichen Compiler-Lauf ausgetauscht werden, würde der Compiler die Sequenz ''' vorfinden und einen Fehler melden.

4.2.3 Die integralen Typen

Java stellt vier ganzzahlige Datentypen zur Verfügung, und zwar `byte`, `short`, `int` und `long`, mit jeweils 1, 2, 4 und 8 Byte Länge. Alle ganzzahligen Typen sind vorzeichenbehaftet, und ihre Länge ist auf allen Plattformen gleich.

Anders als in C sind die Schlüsselwörter `long` und `short` bereits Typenbezeichner und nicht nur Modifier. Es ist daher nicht erlaubt, `long int` oder `short int` anstelle von `long` bzw. `short` zu schreiben. Auch den Modifier `unsigned` gibt es in Java nicht.

Literale

Ganzzahlige Literale können in Dezimal-, Oktal- oder Hexadezimalform geschrieben werden. Ein oktaler Wert beginnt mit dem Präfix `0`, ein hexadezimaler Wert mit `0x`. Dezimale Literale dürfen nur aus den Ziffern `0` bis `9`, oktale aus den Ziffern `0` bis `7` und hexadezimale aus den Ziffern `0` bis `9` und den Buchstaben `a` bis `f` und `A` bis `F` bestehen.

Durch Voranstellen eines `-` können negative Zahlen dargestellt werden, positive können wahlweise durch ein `+` eingeleitet werden. Ganzzahlige Literale sind grundsätzlich vom Typ `int`, wenn nicht der Suffix `L` oder `l` hinten angehängt wird. In diesem Fall sind sie vom Typ `long`.

4.2.4 Die Fließkommazahlen

Java kennt die beiden IEEE-754-Fließkommatypen `float` (einfache Genauigkeit) und `double` (doppelte Genauigkeit). Die Länge beträgt 4 Byte für `float` und 8 Byte für `double`.

Literale

Fließkommaliterale werden immer in Dezimalnotation aufgeschrieben. Sie bestehen aus einem Vorkommateil, einem Dezimalpunkt, einem Nachkommateil, einem Exponenten und einem Suffix. Um ein Fließkommaliteral von einem integralen Literal unterscheiden zu können, muss mindestens der Dezimalpunkt, der Exponent oder der Suffix vorhanden sein. Entweder der Vorkomma- oder der Nachkommateil darf ausgelassen werden, aber nicht beide. Vorkommateil und Exponent können wahlweise durch das Vorzeichen `+` oder `-` eingeleitet werden. Weiterhin ist der Exponent, der durch ein `e` oder `E` eingeleitet wird, optional. Auch der Suffix kann weggelassen werden, wenn durch die anderen Merkmale klar ist, dass es sich um eine Fließkommazahl handelt. Der Suffix kann entweder `f` oder `F` sein, um anzuzeigen, dass es sich um ein `float` handelt, oder `d` oder `D`, um ein `double` anzuzeigen. Fehlt er, so ist das Literal (unabhängig von seiner Größe) vom Typ `double`.

Gültige Fließkommazahlen sind:

- 3.14
- 2f
- 1e1
- .5f
- 6.

Neben diesen nummerischen Literalen gibt es noch einige symbolische in den Klassen `Float` und `Double` des Pakets `java.lang`. Tabelle 4.3 gibt eine Übersicht dieser vordefinierten Konstanten. `NaN` entsteht beispielsweise bei der Division durch 0, `POSITIVE_INFINITY` bzw. `NEGATIVE_INFINITY` sind Zahlen, die größer bzw. kleiner als der darstellbare Bereich sind.

Tabelle 4.3: Symbolische Fließkommaliterale

Name	Verfügbar für	Bedeutung
`MAX_VALUE`	`Float`, `Double`	Größter darstellbarer positiver Wert
`MIN_VALUE`	`Float`, `Double`	Kleinster darstellbarer positiver Wert
`NaN`	`Float`, `Double`	Not-A-Number
`NEGATIVE_INFINITY`	`Float`, `Double`	Negativ unendlich
`POSITIVE_INFINITY`	`Float`, `Double`	Positiv unendlich

4.3 Variablen

4.3.1 Grundeigenschaften

Variablen dienen dazu, Daten im Hauptspeicher eines Programms abzulegen und gegebenenfalls zu lesen oder zu verändern. In Java gibt es drei Typen von Variablen:

- *Instanzvariablen*, die im Rahmen einer Klassendefinition definiert und zusammen mit dem Objekt angelegt werden.
- *Klassenvariablen*, die ebenfalls im Rahmen einer Klassendefinition definiert werden, aber unabhängig von einem konkreten Objekt existieren.
- *Lokale Variablen*, die innerhalb einer Methode oder eines Blocks definiert werden und nur dort existieren.

Daneben betrachtet die Sprachdefinition auch Array-Komponenten und die Parameter von Methoden und Exception-Handlern als Variablen.

Eine Variable in Java ist immer typisiert. Sie ist entweder von einem primitiven Typen oder von einem Referenztypen. Mit Ausnahme eines Spezialfalls bei Array-Variablen, auf den wir später zurückkommen, werden alle Typüberprüfungen zur Compile-Zeit vorgenommen. Java ist damit im klassischen Sinne eine typsichere Sprache.

Um einer Variablen vom Typ `T` einen Wert `X` zuzuweisen, müssen `T` und `X` zuweisungskompatibel sein. Welche Typen zuweisungskompatibel sind, wird am Ende dieses Kapitels in Abschnitt 4.6 auf Seite 108 erklärt.

Variablen können auf zwei unterschiedliche Arten verändert werden:

- durch eine Zuweisung
- durch einen Inkrement- oder Dekrement-Operator

Beide Möglichkeiten werden in Kapitel 5 auf Seite 113 ausführlich erklärt.

4.3.2 Deklaration von Variablen

Die Deklaration einer Variable erfolgt in der Form

```
Typname Variablenname;
```

Dabei wird eine Variable des Typs `Typname` mit dem Namen `Variablenname` angelegt. Variablendeklarationen dürfen in Java an beliebiger Stelle im Programmcode erfolgen. Das fol-

gende Beispielprogramm legt die Variablen `a`, `b`, `c` und `d` an und gibt ihren Inhalt auf dem Bildschirm aus:

Listing 4.2: Einfache Variablen ausgeben

```
001 /* Listing0402.java */
002
003 public class Listing0402
004 {
005   public static void main(String[] args)
006   {
007     int a;
008     a = 1;
009     char b = 'x';
010     System.out.println(a);
011     double c = 3.1415;
012     System.out.println(b);
013     System.out.println(c);
014     boolean d = false;
015     System.out.println(d);
016   }
017 }
```

Die Ausgabe des Programms ist:

```
1
x
3.1415
false
```

Wie in diesem Beispiel zu sehen ist, dürfen Variablen gleich bei der Deklaration initialisiert werden. Dazu ist einfach der gewünschte Wert hinter einem Zuweisungsoperator an die Deklaration anzuhängen:

Listing 4.3: Initialisieren von Variablen

```
char b = 'x';
double c = 3.1415;
boolean d = false;
```

4.3.3 Lebensdauer/Sichtbarkeit

Die Sichtbarkeit lokaler Variablen erstreckt sich von der Stelle ihrer Deklaration bis zum Ende der Methode, in der sie deklariert wurden. Falls innerhalb eines Blocks lokale Variablen angelegt wurden, sind sie bis zum Ende des Blocks sichtbar. Die Lebensdauer einer lokalen Variable beginnt, wenn die zugehörige Deklarationsanweisung ausgeführt wird. Sie

endet mit dem Ende des Methodenaufrufs. Falls innerhalb eines Blocks lokale Variablen angelegt wurden, endet ihre Lebensdauer mit dem Verlassen des Blocks. Es ist in Java nicht erlaubt, lokale Variablen zu deklarieren, die gleichnamige lokale Variablen eines weiter außen liegenden Blocks verdecken. Das Verdecken von Klassen- oder Instanzvariablen ist dagegen zulässig.

Instanzvariablen werden zum Zeitpunkt des Erzeugens einer neuen Instanz einer Klasse angelegt. Sie sind innerhalb der ganzen Klasse sichtbar, solange sie nicht von gleichnamigen lokalen Variablen verdeckt werden. In diesem Fall ist aber der Zugriff mithilfe des `this`-Zeigers möglich: `this.name` greift immer auf die Instanz- oder Klassenvariable `name` zu, selbst wenn eine gleichnamige lokale Variable existiert. Mit dem Zerstören des zugehörigen Objektes werden auch alle Instanzvariablen zerstört.

Klassenvariablen leben während der kompletten Laufzeit des Programms. Die Regeln für ihre Sichtbarkeit entsprechen denen von Instanzvariablen.

4.4 Arrays

Arrays in Java unterscheiden sich dadurch von Arrays in anderen Programmiersprachen, dass sie *Objekte* sind. Obwohl dieser Umstand in vielen Fällen vernachlässigt werden kann, bedeutet er dennoch:

- dass Array-Variablen *Referenzen* sind
- dass Arrays Methoden und Instanz-Variablen besitzen
- dass Arrays zur Laufzeit erzeugt werden

Dennoch bleibt ein Array immer eine (möglicherweise mehrdimensionale) Reihung von Elementen eines festen Grundtyps. Arrays in Java sind *semidynamisch*, d.h. ihre Größe kann zur Laufzeit festgelegt, später aber nicht mehr verändert werden.

4.4.1 Deklaration und Initialisierung

Die Deklaration eines Arrays in Java erfolgt in zwei Schritten:

- Deklaration einer Array-Variablen
- Erzeugen eines Arrays und Zuweisung an die Array-Variable

Die Deklaration eines Arrays entspricht syntaktisch der einer einfachen Variablen, mit dem Unterschied, dass an den Typnamen eckige Klammern angehängt werden:

```
001 int[] a;
002 double[] b;
003 boolean[] c;
```

Listing 4.4: Deklaration von Arrays

Wahlweise können die eckigen Klammern auch hinter den Variablennamen geschrieben werden, aber das ist ein Tribut an die Kompatibilität zu C/C++ und sollte in neuen Java-Programmen vermieden werden.

Zum Zeitpunkt der Deklaration wird noch nicht festgelegt, wie viele Elemente das Array haben soll. Dies geschieht erst später bei seiner Initialisierung, die mithilfe des `new`-Operators oder durch Zuweisung eines Array-Literals ausgeführt wird. Sollen also beispielsweise die oben deklarierten Arrays 5, 10 und 15 Elemente haben, würden wir das Beispiel wie folgt erweitern:

```
001 a = new int[5];
002 b = new double[10];
003 c = new boolean[15];
```

Listing 4.5: Erzeugen von Arrays

Ist bereits zum Deklarationszeitpunkt klar, wie viele Elemente das Array haben soll, können Deklaration und Initialisierung zusammen geschrieben werden:

```
001 int[] a = new int[5];
002 double[] b = new double[10];
003 boolean[] c = new boolean[15];
```

Listing 4.6: Deklaration und Initialisierung von Arrays

Alternativ zur Verwendung des `new`-Operators kann ein Array auch *literal* initialisiert werden. Dazu werden die Elemente des Arrays in geschweifte Klammern gesetzt und nach einem Zuweisungsoperator zur Initialisierung verwendet. Die Größe des Arrays ergibt sich aus der Anzahl der zugewiesenen Elemente:

```
001 int[] x = {1,2,3,4,5};
002 boolean[] y = {true, true};
```

Listing 4.7: Initialisierung mit literalen Arrays

Das Beispiel generiert ein `int`-Array `x` mit fünf Elementen und ein `boolean`-Array `y` mit zwei Elementen. Anders als bei der expliziten Initialisierung mit `new` muss die Initialisierung in diesem Fall unmittelbar bei der Deklaration erfolgen.

4.4.2 Zugriff auf Array-Elemente

Bei der Initialisierung eines Arrays von `n` Elementen werden die einzelnen Elemente von `0` bis `n-1` durchnummeriert. Der Zugriff auf jedes einzelne Element erfolgt über seinen numerischen Index, der nach dem Array-Namen in eckigen Klammern geschrieben wird. Das nachfolgende Beispiel deklariert zwei Arrays mit Elementen des Typs `int` bzw. `boolean`, die dann ausgegeben werden:

Listing 4.8: Deklaration und Zugriff auf Arrays

```
001 /* Listing0408.java */
002
003 public class Listing0408
004 {
005   public static void main(String[] args)
006   {
007     int[] prim = new int[5];
008     boolean[] b = {true,false};
009     prim[0] = 2;
010     prim[1] = 3;
011     prim[2] = 5;
012     prim[3] = 7;
013     prim[4] = 11;
014
015     System.out.println("prim hat "+prim.length+" Elemente");
016     System.out.println("b hat "+b.length+" Elemente");
017     System.out.println(prim[0]);
018     System.out.println(prim[1]);
019     System.out.println(prim[2]);
020     System.out.println(prim[3]);
021     System.out.println(prim[4]);
022     System.out.println(b[0]);
023     System.out.println(b[1]);
024   }
025 }
```

Die Ausgabe des Programms ist:

```
prim hat 5 Elemente
b hat 2 Elemente
2
3
5
7
11
true
false
```

Der Array-Index muss vom Typ `int` sein. Aufgrund der vom Compiler automatisch vorgenommenen Typkonvertierungen sind auch `short`, `byte` und `char` zulässig. Jedes Array hat eine Instanzvariable `length`, die die Anzahl seiner Elemente angibt. Indexausdrücke werden vom Laufzeitsystem auf Einhaltung der Array-Grenzen geprüft. Sie müssen größer gleich 0 und kleiner als `length` sein.

4.4.3 Mehrdimensionale Arrays

Mehrdimensionale Arrays werden erzeugt, indem zwei oder mehr Paare eckiger Klammern bei der Deklaration angegeben werden. Mehrdimensionale Arrays werden als Arrays von Arrays angelegt. Die Initialisierung erfolgt analog zu eindimensionalen Arrays durch Angabe der Anzahl der Elemente je Dimension.

Der Zugriff auf mehrdimensionale Arrays geschieht durch Angabe aller erforderlichen Indizes, jeweils in eigenen eckigen Klammern. Auch bei mehrdimensionalen Arrays kann eine literale Initialisierung durch Schachtelung der Initialisierungssequenzen erreicht werden. Das folgende Beispiel erzeugt ein Array der Größe 2 * 3 und gibt dessen Elemente aus:

Listing 4.9: Zugriff auf mehrdimensionale Arrays

```
001 /* Listing0409.java */
002
003 public class Listing0409
004 {
005   public static void main(String[] args)
006   {
007     int[][] a = new int[2][3];
008
009     a[0][0] = 1;
010     a[0][1] = 2;
011     a[0][2] = 3;
012     a[1][0] = 4;
013     a[1][1] = 5;
014     a[1][2] = 6;
015     System.out.println(""+a[0][0]+a[0][1]+a[0][2]);
016     System.out.println(""+a[1][0]+a[1][1]+a[1][2]);
017   }
018 }
```

Die Ausgabe des Programms ist:

```
123
456
```

Da mehrdimensionale Arrays als geschachtelte Arrays gespeichert werden, ist es auch möglich, *nicht-rechteckige* Arrays zu erzeugen. Das folgende Beispiel deklariert und initialisiert ein zweidimensionales dreieckiges Array und gibt es auf dem Bildschirm aus. Gleichzeitig zeigt es die Verwendung der `length`-Variable, um die Größe des Arrays oder Sub-Arrays herauszufinden.

Listing 4.10: Ein nicht-rechteckiges Array

```
001 /* Listing0410.java */
002
003 public class Listing0410
004 {
005   public static void main(String[] args)
006   {
007     int[][] a = { {0},
008                   {1,2},
009                   {3,4,5},
010                   {6,7,8,9}
011                 };
012     for (int i=0; i<a.length; ++i) {
013       for (int j=0; j<a[i].length; ++j) {
014         System.out.print(a[i][j]);
015       }
016       System.out.println();
017     }
018   }
019 }
```

Die Arbeitsweise des Programms ist trotz der erst in Kapitel 6 auf Seite 129 vorzustellenden `for`-Schleife unmittelbar verständlich. Die Ausgabe des Programms lautet:

```
0
12
345
6789
```

4.5 Referenztypen

4.5.1 Beschreibung

Referenztypen sind neben den primitiven Datentypen die zweite wichtige Klasse von Datentypen in Java. Zu den Referenztypen gehören Objekte, Strings und Arrays. Weiterhin gibt es die vordefinierte Konstante `null`, die eine *leere* Referenz bezeichnet.

Eigentlich sind auch Strings und Arrays Objekte, aber es gibt bei ihnen einige Besonderheiten, die eine Unterscheidung von normalen Objekten rechtfertigen:

- Sowohl bei Strings als auch bei Arrays kennt der Compiler Literale, die einen expliziten Aufruf des `new`-Operators überflüssig machen.
- Arrays sind »klassenlose« Objekte. Sie können ausschließlich vom Compiler erzeugt werden, besitzen aber keine explizite Klassendefinition. Dennoch haben sie eine öffentliche Instanzvariable `length` und werden vom Laufzeitsystem wie normale Objekte behandelt.
- Die Klasse `String` ist zwar wie eine gewöhnliche Klasse in der Laufzeitbibliothek von Java vorhanden. Der Compiler hat aber Kenntnisse über den inneren Aufbau von Strings und generiert bei Stringoperationen Code, der auf Methoden der Klassen `String` und `StringBuffer` zugreift. Eine ähnlich enge Zusammenarbeit zwischen Compiler und Laufzeitbibliothek gibt es auch bei Threads und Exceptions. Wir werden auf diese Besonderheiten in den nachfolgenden Kapiteln noch einmal zurückkommen.

Das Verständnis für Referenztypen ist entscheidend für die Programmierung in Java. Referenztypen können prinzipiell genauso benutzt werden wie primitive Typen. Da sie jedoch lediglich einen Verweis darstellen, ist die Semantik einiger Operatoren anders als bei primitiven Typen:

- Die Zuweisung einer Referenz kopiert lediglich den Verweis auf das betreffende Objekt, das Objekt selbst dagegen bleibt unkopiert. Nach einer Zuweisung zweier Referenztypen zeigen diese also auf ein und dasselbe Objekt. Sollen Referenztypen kopiert werden, so ist ein Aufruf der Methode `clone` erforderlich (siehe Abschnitt 8.1.2 auf Seite 174).
- Der Gleichheitstest zweier Referenzen testet, ob beide Verweise gleich sind, d.h. auf ein und dasselbe Objekt zeigen. Das ist aber eine strengere Forderung als inhaltliche Gleichheit. Soll lediglich auf inhaltliche Gleichheit getestet werden, kann dazu die `equals`-Methode verwendet werden, die von den meisten Klassen implementiert wird (ebenfalls in Abschnitt 8.1.2 auf Seite 174 erläutert). Analoges gilt für den Test auf Ungleichheit.

Anders als in C und C++, wo der *-Operator zur Dereferenzierung eines Zeigers nötig ist, erfolgt in Java der Zugriff auf Referenztypen in der gleichen Weise wie der auf primitive Typen. Einen expliziten Dereferenzierungsoperator gibt es dagegen nicht.

4.5.2 Speichermanagement

Während primitive Typen lediglich deklariert werden, reicht dies bei Referenztypen nicht aus. Sie müssen mithilfe des `new`-Operators oder – im Falle von Arrays und Strings – durch Zuweisung von Literalen zusätzlich noch explizit erzeugt werden.

Listing 4.11: Erzeugen eines Objekts mit dem new-Operator

```
Vector v = new Vector();
```

Java verfügt über ein automatisches Speichermanagement. Dadurch braucht man sich als Java-Programmierer nicht um die Rückgabe von Speicher zu kümmern, der von Referenzvariablen belegt wird. Ein mit niedriger Priorität im Hintergrund arbeitender *Garbage Collector* sucht periodisch nach Objekten, die nicht mehr referenziert werden, um den durch sie belegten Speicher freizugeben.

4.6 Typkonvertierungen

4.6.1 Standardkonvertierungen

Es gibt diverse Konvertierungen zwischen unterschiedlichen Datentypen in Java. Diese werden einerseits vom Compiler automatisch vorgenommen, beispielsweise bei der Auswertung von nummerischen Ausdrücken. Andererseits können sie verwendet werden, um mithilfe des Type-Cast-Operators (siehe Kapitel 5 auf Seite 113) eigene Konvertierungen vorzunehmen.

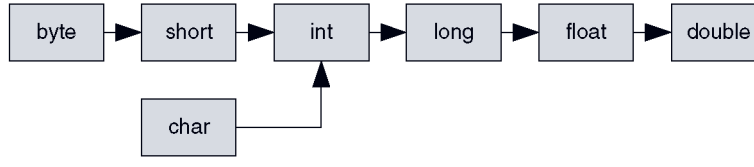
Java unterscheidet prinzipiell zwischen *erweiternden* und *einschränkenden Konvertierungen* und diese noch einmal nach primitiven Typen und Referenztypen. Zunächst zu den Referenztypen:

- Als erweiternde Konvertierung eines Referenztyps `T` wird vor allem die Umwandlung eines Objekts vom Typ `T` in eine seiner Vaterklassen angesehen.
- Als einschränkende Konvertierung eines Referenztyps `T` wird vor allem die Umwandlung eines Objekts vom Typ `T` in eine der aus `T` abgeleiteten Klassen angesehen.

Daneben gibt es noch eine ganze Reihe weiterer Regeln zur Definition von erweiternden und einschränkenden Konvertierungen von Referenztypen. Die Bedeutung von Vaterklassen und den daraus abgeleiteten Unterklassen wird in Kapitel 8 auf Seite 173 ausführlich erläutert.

Konvertierungen auf primitiven Datentypen sind etwas aufwendiger zu erklären. Wir benutzen dazu Abbildung 4.1 auf Seite 108:

Abb. 4.1: Konvertierungen auf primitiven Datentypen



Jede Konvertierung, die in Pfeilrichtung erfolgt, beschreibt eine erweiternde Konvertierung, und jede Konvertierung, die entgegen der Pfeilrichtung erfolgt, beschreibt eine einschränkende Konvertierung. Andere Konvertierungen zwischen primitiven Datentypen sind nicht erlaubt. Insbesondere gibt es also keine legale Konvertierung von und nach `boolean` und auch keine Konvertierung zwischen primitiven Typen und Referenztypen.

Welche Bedeutung haben nun aber die verschiedenen Konvertierungen zwischen unterschiedlichen Typen? Wir wollen uns an dieser Stelle lediglich mit den Konvertierungen zwischen primitiven Typen beschäftigen. Wie aus Abbildung 4.1 auf Seite 108 ersichtlich ist, beschränken sich diese auf Umwandlungen zwischen nummerischen Typen. Die Anwendung einer erweiternden Konvertierung wird in folgenden Fällen vom Compiler automatisch vorgenommen:

- Bei einer Zuweisung, wenn der Typ der Variablen und des zugewiesenen Ausdrucks nicht identisch ist.
- Bei der Auswertung eines arithmetischen Ausdrucks, wenn Operanden unterschiedlich typisiert sind.
- Beim Aufruf einer Methode, falls die Typen der aktuellen Parameter nicht mit denen der formalen Parameter übereinstimmen.

Es ist daher beispielsweise ohne weiteres möglich, ein `short` und ein `int` gemeinsam in einem Additionsausdruck zu verwenden, da ein `short` mithilfe einer erweiternden Konvertierung in ein `int` verwandelt werden kann. Ebenso ist es möglich, ein `char` als Array-Index zu verwenden, da es erweiternd in ein `int` konvertiert werden kann. Auch die Arithmetik in Ausdrücken, die sowohl integrale als auch Fließkommawerte enthalten, ist möglich, da der Compiler alle integralen Parameter erweiternd in Fließkommawerte umwandeln kann.

Es ist dagegen nicht ohne weiteres möglich, einer `int`-Variablen einen `double`-Wert zuzuweisen. Die hierzu erforderliche einschränkende Konvertierung nimmt der Compiler nicht selbst vor; sie kann allerdings mithilfe des Type-Cast-Operators manuell durchgeführt werden. Auch die Verwendung eines `long` als Array-Index verbietet sich aus diesem Grund.

Bei den einschränkenden Konvertierungen kann es passieren, dass ein Wert verfälscht wird, da der Wertebereich des Zielobjekts kleiner ist. Aber auch erweiternde Konvertierungen sind nicht immer gefahrlos möglich. So kann zwar beispielsweise ein `float` mindestens genauso große Werte aufnehmen wie ein `long`. Seine Genauigkeit ist aber auf ca. 8 Stellen beschränkt, und daher können größere Ganzzahlen (z.B. 1000000123) nicht mehr mit voller Genauigkeit dargestellt werden.

4.6.2 Vorzeichenlose Bytes

In Java sind alle nummerischen Datentypen vorzeichenbehaftet. Das ist in vielen Fällen sinnvoll, kann aber bei der Handhabung von 8-Bit-Bytes hinderlich sein. Wird ein Byte als Repräsentation eines 8-Bit langen Maschinenworts angesehen, will man meist den Wertebereich von 0 bis 255 zur Verfügung haben. Als vorzeichenbehafteter Datentyp kann `byte` aber nur Werte von -128 bis 127 darstellen. Ein Wert größer oder gleich 128 erfordert also mindestens ein `short` oder ein `int`. Deren Länge beträgt aber 2 bzw. 4 Byte.

Das Dilemma lässt sich dadurch auflösen, dass man zwischen der programminternen Verarbeitung eines Bytes und seiner äußeren Repräsentation unterscheidet. Die Repräsentation nach außen erfolgt dabei mit dem Datentyp `byte`. Zur Verarbeitung im Programm wird er dagegen in ein `int` konvertiert, so dass alle Werte von 0 bis 255 dargestellt werden können. Konvertierungsmethoden erlauben es, zwischen beiden Darstellungen zu wechseln.

Natürlich gibt es keinerlei automatischen Schutz gegen Wertebereichsüberschreitungen, wenn ein Byte als `int` verarbeitet wird. Dafür ist ausschließlich die Anwendung selbst verantwortlich.

Das folgende Listing zeigt eine einfache Klasse `ByteKit`, mit der zwischen beiden Darstellungen gewechselt werden kann:

Listing 4.12: Umwandlung zwischen int, byte und char

```
001 /**
002  * ByteKit
003  *
004  * Einfache Klasse zur Umwandlung zwischen int, char und
005  * vorzeichenlosen Bytes.
006  */
007 public class ByteKit
008 {
009   /**
010    * Wandelt value (0 <= value <= 255) in ein byte um.
011    */
012   public static byte fromUnsignedInt(int value)
013   {
014     return (byte)value;
015   }
016
017   /**
018    * Wandelt c in ein byte um. Das High-Byte wird ignoriert.
019    */
020   public static byte fromChar(char c)
021   {
022     return (byte)(c & 0xFF);
023   }
```

Listing 4.12: Umwandlung zwischen int, byte und char (Forts.)

```
024
025   /**
026    * Betrachtet value als vorzeichenloses byte und wandelt
027    * es in eine Ganzzahl im Bereich 0..255 um.
028    */
029   public static int toUnsignedInt(byte value)
030   {
031     return (value & 0x7F) + (value < 0 ? 128 : 0);
032   }
033
034   /**
035    * Betrachtet value als vorzeichenloses byte und wandelt
036    * es in ein Unicode-Zeichen mit High-Byte 0 um.
037    */
038   public static char toChar(byte value)
039   {
040     return (char)toUnsignedInt(value);
041   }
042
043   /**
044    * Liefert die Binaerdarstellung von value.
045    */
046   public static String toBitString(byte value)
047   {
048     char[] chars = new char[8];
049     int mask = 1;
050     for (int i = 0; i < 8; ++i) {
051       chars[7 - i] = (value & mask) != 0 ? '1' : '0';
052       mask <<= 1;
053     }
054     return new String(chars);
055   }
056 }
```

Eine einfache Anwendung der Klasse `ByteKit` zeigt das folgende Programm:

Listing 4.13: Anwendung der Klasse ByteKit

```
001 /* Listing0413 */
002
003 public class Listing0413
004 {
005   public static void main(String[] args)
006   {
007     for (int i = 0; i < 256; ++i) {
008       System.out.print("i=" + i);
009       byte b = ByteKit.fromUnsignedInt(i);
010       System.out.print(" b=" + ByteKit.toBitString(b));
011       char c = ByteKit.toChar(b);
012       System.out.print(" c=" + (c >= 32 ? c : '.'));
```

Listing 4.13: Anwendung der Klasse ByteKit (Forts.)

```
013         System.out.println();
014       }
015     }
016 }
```

4.7 Zusammenfassung

In diesem Kapitel wurden folgende Themen behandelt:

- Der Unicode-Zeichensatz
- Namenskonventionen für Bezeichner
- Kommentare in Java
- Die primitiven Datentypen `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` und `double`
- Die boolschen Literale `true` und `false` sowie die Fließkommaliterale `MAX_VALUE`, `MIN_VALUE`, `NaN`, `NEGATIVE_INFINITY` und `POSITIVE_INFINITY`
- Die Standard-Escape-Sequenzen `\b`, `\t`, `\n`, `\f`, `\r`, `\"`, `\'`, `\\` und `\nnn`
- Deklaration, Lebensdauer und Sichtbarkeit von Variablen
- Deklaration, Initialisierung und Zugriff auf Arrays
- Referenztypen und automatisches Speichermanagement
- Typkonvertierungen