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

5 Ausdrücke

5.1 Eigenschaften von Ausdrücken

Wie in den meisten anderen Programmiersprachen gehören auch in Java *Ausdrücke* zu den kleinsten ausführbaren Einheiten eines Programms. Sie dienen dazu, Variablen einen Wert zuzuweisen, nummerische Berechnungen durchzuführen oder logische Bedingungen zu formulieren.

Ein Ausdruck besteht immer aus mindestens einem Operator und einem oder mehreren Operanden, auf die der Operator angewendet wird. Nach den Typen der Operanden unterscheidet man *nummerische*, *relationale*, *logische*, *bitweise*, *Zuweisungs-* und *sonstige* Operatoren. Jeder Ausdruck hat einen Rückgabewert, der durch die Anwendung des Operators auf die Operanden entsteht. Der Typ des Rückgabewerts bestimmt sich aus den Typen der Operanden und der Art des verwendeten Operators.

Neben der Typisierung ist die *Stelligkeit eines Operators* von Bedeutung. Operatoren, die lediglich ein Argument erwarten, nennt man *einstellig*, solche mit zwei Argumenten *zweistellig*. Beispiele für einstellige Operatoren sind das unäre Minus (also das negative Vorzeichen) oder der logische Nicht-Operator. Arithmetische Operatoren wie Addition oder Subtraktion sind zweistellig. Darüber hinaus gibt es in Java – wie in C – auch den dreistelligen Fragezeichenoperator.

Für die richtige Interpretation von Ausdrücken muss man die *Bindungs-* und *Assoziativitätsregeln* der Operatoren kennen. Bindungsregeln beschreiben die Reihenfolge, in der verschiedene Operatoren innerhalb eines Ausdrucks ausgewertet werden. So besagt beispielsweise die bekannte Regel »Punktrechnung vor Strichrechnung«, dass der Multiplikationsoperator eine höhere Bindungskraft hat als der Additionsoperator und demnach in Ausdrücken zuerst ausgewertet wird. Assoziativität beschreibt die Auswertungsreihenfolge von Operatoren derselben Bindungskraft, also beispielsweise die Auswertungsreihenfolge einer Kette von Additionen und Subtraktionen. Da Summationsoperatoren linksassoziativ sind, wird beispielsweise der Ausdruck `a-b+c` wie `(a-b)+c` ausgewertet und nicht wie `a-(b+c)`.

Neben ihrer eigentlichen Funktion, einen Rückgabewert zu produzieren, haben einige Operatoren auch *Nebeneffekte*. Als Nebeneffekt bezeichnet man das Verhalten eines Ausdrucks, auch ohne explizite Zuweisung die Inhalte von Variablen zu verändern. Meist sind diese Nebeneffekte erwünscht, wie etwa bei der Verwendung der Inkrement- und Dekrementoperatoren. Komplexere Ausdrücke können wegen der oftmals verdeckten Auswertungsreihenfolge der Teilausdrücke jedoch unter Umständen schwer verständliche Nebeneffekte enthalten und sollten deshalb mit Vorsicht angewendet werden.

Im Gegensatz zu vielen anderen Programmiersprachen ist in Java die *Reihenfolge der Auswertung* der Operanden innerhalb eines Teilausdrucks wohldefiniert. Die Sprachdefinition schreibt explizit vor, den linken Operanden eines Ausdrucks vollständig vor dem rechten Operanden auszuwerten. Falls also beispielsweise `i` den Wert 2 hat, dann ergibt der Ausdruck `(i=3) * i` in jedem Fall den Wert 9 und nicht 6.

Neben der natürlichen Auswertungsreihenfolge, die innerhalb eines Ausdrucks durch die Bindungs- und Assoziativitätsregeln vorgegeben wird, lässt sich durch eine explizite Klammerung jederzeit eine andere Reihenfolge erzwingen. Während das Ergebnis des Ausdrucks `4+2*5` wegen der Bindungsregeln 14 ist, liefert `(4+2)*5` das Resultat 30. Die Regeln für die Klammerung von Teilausdrücken in Java gleichen denen aller anderen Programmiersprachen und brauchen daher nicht weiter erläutert zu werden.

In Java gibt es ein Konzept, das sich *Definite Assignment* nennt. Gemeint ist damit die Tatsache, dass jede lokale Variable vor ihrer ersten Verwendung definitiv initialisiert sein muss. Das wünscht sich eigentlich zwar auch jeder Programmierer, aber in Java wird dies durch den Compiler sichergestellt! Dazu muss im Quelltext eine Datenflussanalyse durchgeführt werden, die jeden möglichen Ausführungspfad von der Deklaration einer Variablen bis zu ihrer Verwendung ermittelt und sicherstellt, dass kein Weg existiert, der eine Initialisierung auslassen würde.

Die folgende Methode `Test` lässt sich beispielsweise deshalb nicht fehlerfrei kompilieren, weil `k` vor der Ausgabeanweisung nicht initialisiert wird, wenn `i` kleiner 2 ist:

Listing 5.1: Fehler beim Kompilieren durch unvollständige Initialisierung

```
001 public static void Test(int i)
002 {
003   int k;
004   if (i >= 2) {
005     k = 5;
006   }
007   System.out.println(k);
008 }
```

Ein solches Verhalten des Compilers ist natürlich höchst wünschenswert, denn es zeigt einen *tatsächlichen* Fehler an, der sonst unbemerkt geblieben wäre. Dass die Datenflussanalyse allerdings auch Grenzen hat, zeigt das folgende Beispiel, bei dem ebenfalls ein Compiler-Fehler auftritt:

Listing 5.2: Fehler beim Kompilieren durch unvollständige Datenflussanalyse

```
001 public static void Test(int i)
002 {
003   int k;
004   if (i < 2) {
005     k = 5;
```

```
006   }
007   if (i >= 2) {
008     k = 6;
009   }
010   System.out.println(k);
011 }
```

Listing 5.2: Fehler beim Kompilieren durch unvollständige Datenfluss-analyse (Forts.)

Natürlich kann hier `k` nicht uninitialisiert bleiben, denn eine der beiden Bedingungen ist immer wahr. Leider gehen die Fähigkeiten der Compiler noch nicht so weit, dies zu erkennen.

Es wäre in diesem Fall natürlich vernünftiger gewesen, den `else`-Zweig an die erste Verzweigung anzuhängen und damit die zweite ganz einzusparen.

In Wirklichkeit funktioniert die Datenflussanalyse bei vernünftigem Programmierstil recht gut. Die wenigen Fälle, in denen der Compiler sich irrt, können in der Regel durch explizite Initialisierungen aufgelöst werden.

5.2 Arithmetische Operatoren

Java kennt die üblichen arithmetischen Operatoren der meisten imperativen Programmiersprachen, nämlich die *Addition*, *Subtraktion*, *Multiplikation*, *Division* und den *Restwertoperator*. Zusätzlich gibt es die einstelligen Operatoren für positives und negatives Vorzeichen sowie die nebeneffektbehafteten *Prä-* und *Postinkrement-* und *Prä-* und *Postdekrement-*Operatoren.

Die arithmetischen Operatoren erwarten nummerische Operanden und liefern einen nummerischen Rückgabewert. Haben die Operanden unterschiedliche Typen, beispielsweise `int` und `float`, so entspricht der Ergebnistyp des Teilausdrucks dem größeren der beiden Operanden. Zuvor wird der kleinere der beiden Operanden mithilfe einer erweiternden Konvertierung in den Typ des größeren konvertiert.

Tabelle 5.1 gibt eine Übersicht der in Java verfügbaren arithmetischen Operatoren.

Tabelle 5.1: Arithmetische Operatoren

Operator	Bezeichnung	Bedeutung
+	Positives Vorzeichen	+n ist gleichbedeutend mit n
-	Negatives Vorzeichen	-n kehrt das Vorzeichen von n um
+	Summe	a + b ergibt die Summe von a und b

Tabelle 5.1: Arithmetische Operatoren (Forts.)

Operator	Bezeichnung	Bedeutung
-	Differenz	a - b ergibt die Differenz von a und b
*	Produkt	a * b ergibt das Produkt aus a und b
/	Quotient	a / b ergibt den Quotienten von a und b
%	Restwert	a % b ergibt den Rest der ganzzahligen Division von a durch b. In Java lässt sich dieser Operator auch auf Fließkommazahlen anwenden.
++	Präinkrement	++a ergibt a+1 und erhöht a um 1
++	Postinkrement	a++ ergibt a und erhöht a um 1
--	Prädekrement	--a ergibt a-1 und verringert a um 1
--	Postdekrement	a-- ergibt a und verringert a um 1

5.3 Relationale Operatoren

Relationale Operatoren dienen dazu, Ausdrücke miteinander zu vergleichen und in Abhängigkeit davon einen logischen Rückgabewert zu produzieren. Java stellt den *Gleichheits-* und *Ungleichheitstest* sowie die Vergleiche *Größer* und *Kleiner* sowie *Größer gleich* und *Kleiner gleich* zur Verfügung. Die relationalen Operatoren arbeiten auf beliebigen – auch gemischten – nummerischen Typen. Im Fall von Gleichheit und Ungleichheit funktionieren sie auch auf Objekttypen. Tabelle 5.2 gibt eine Übersicht der in Java verfügbaren relationalen Operatoren.

Tabelle 5.2: Relationale Operatoren

Operator	Bezeichnung	Bedeutung
==	Gleich	a == b ergibt true, wenn a gleich b ist. Sind a und b Referenztypen, so ist der Rückgabewert true, wenn beide Werte auf dasselbe Objekt zeigen.
!=	Ungleich	a != b ergibt true, wenn a ungleich b ist. Sind a und b Objekte, so ist der Rückgabewert true, wenn beide Werte auf unterschiedliche Objekte zeigen.
<	Kleiner	a < b ergibt true, wenn a kleiner b ist.
<=	Kleiner gleich	a <= b ergibt true, wenn a kleiner oder gleich b ist.
>	Größer	a > b ergibt true, wenn a größer b ist.
>=	Größer gleich	a >= b ergibt true, wenn a größer oder gleich b ist.

5.4 Logische Operatoren

Logische Operatoren dienen dazu, *boolesche* Werte miteinander zu verknüpfen. Im Gegensatz zu den relationalen Operatoren, die durch Vergleiche erst Wahrheitswerte produzieren, werden logische Operatoren zur Weiterverarbeitung von Wahrheitswerten verwendet.

Java stellt die Grundoperationen *UND*, *ODER* und *NICHT* zur Verfügung und bietet darüber hinaus die Möglichkeit, das Auswertungsverhalten der Operanden zu beeinflussen. Anders als die meisten anderen Programmiersprachen, stellt Java die UND- und ODER-Verknüpfungen in zwei verschiedenen Varianten zur Verfügung, nämlich mit *Short-Circuit-Evaluation* oder ohne.

Bei der Short-Circuit-Evaluation eines logischen Ausdrucks wird ein weiter rechts stehender Teilausdruck nur dann ausgewertet, wenn er für das Ergebnis des Gesamtausdrucks noch von Bedeutung ist. Falls in dem Ausdruck `A && B` also bereits `A` falsch ist, wird zwangsläufig immer auch `A && B` falsch sein, unabhängig von dem Resultat von `B`. Bei der Short-Circuit-Evaluation wird in diesem Fall `B` gar nicht mehr ausgewertet. Analoges gilt bei der Anwendung des ODER-Operators.

Der in Java ebenfalls verfügbare *EXKLUSIV-ODER-Operator* muss natürlich immer in der langen Variante ausgewertet werden. Tabelle 5.3 gibt eine Übersicht der logischen Operatoren.

Tabelle 5.3: Logische Operatoren

Operator	Bezeichnung	Bedeutung
!	Logisches NICHT	!a ergibt false, wenn a wahr ist, und true, wenn a falsch ist.
&&	UND mit Short-Circuit-Evaluation	a && b ergibt true, wenn sowohl a als auch b wahr sind. Ist a bereits falsch, so wird false zurückgegeben und b nicht mehr ausgewertet.
||	ODER mit Short-Circuit-Evaluation	a || b ergibt true, wenn mindestens einer der beiden Ausdrücke a oder b wahr ist. Ist bereits a wahr, so wird true zurückgegeben und b nicht mehr ausgewertet.
&	UND ohne Short-Circuit-Evaluation	a & b ergibt true, wenn sowohl a als auch b wahr sind. Beide Teilausdrücke werden ausgewertet.
|	ODER ohne Short-Circuit-Evaluation	a | b ergibt true, wenn mindestens einer der beiden Ausdrücke a oder b wahr ist. Beide Teilausdrücke werden ausgewertet.
^	Exklusiv-ODER	a ^ b ergibt true, wenn beide Ausdrücke einen unterschiedlichen Wahrheitswert haben.

5.5 Bitweise Operatoren

Mithilfe der bitweisen Operatoren kann auf die Binärdarstellung von nummerischen Operanden zugegriffen werden. Ein nummerischer Datentyp wird dabei als Folge von Bits angesehen, die mithilfe der bitweisen Operatoren einzeln abgefragt und manipuliert werden können.

Java hat dieselben bitweisen Operatoren wie C und C++ und stellt daher *Schiebeoperationen*, *logische Verknüpfungen* und das *Einerkomplement* zur Verfügung. Da alle nummerischen Typen in Java vorzeichenbehaftet sind, gibt es einen zusätzlichen *Rechtsschiebeoperator* >>>, der das höchstwertige Bit nach der Verschiebung auf 0 setzt – und zwar auch dann, wenn es vorher auf 1 stand. Tabelle 5.4 gibt eine Übersicht über die bitweisen Operatoren in Java.

Tabelle 5.4: Bitweise Operatoren

Operator	Bezeichnung	Bedeutung
~	Einerkomplement	~a entsteht aus a, indem alle Bits von a invertiert werden.
|	Bitweises ODER	a | b ergibt den Wert, der entsteht, wenn die korrespondierenden Bits von a und b miteinander ODER-verknüpft werden.
&	Bitweises UND	a & b ergibt den Wert, der entsteht, wenn die korrespondierenden Bits von a und b miteinander UND-verknüpft werden.
^	Bitweises Exklusiv-ODER	a ^ b ergibt den Wert, der entsteht, wenn die korrespondierenden Bits von a und b miteinander Exklusiv-ODER-verknüpft werden.
>>	Rechtsschieben mit Vorzeichen	a >> b ergibt den Wert, der entsteht, wenn alle Bits von a um b Positionen nach rechts geschoben werden. Falls das höchstwertige Bit gesetzt ist (a also negativ ist), wird auch das höchstwertige Bit des Resultats gesetzt.
>>>	Rechtsschieben ohne Vorzeichen	a >>> b ergibt den Wert, der entsteht, wenn alle Bits von a um b Positionen nach rechts geschoben werden. Dabei wird das höchstwertige Bit des Resultats immer auf 0 gesetzt.
<<	Linksschieben	a << b ergibt den Wert, der entsteht, wenn alle Bits von a um b Positionen nach links geschoben werden. Das höchstwertige Bit (also das Vorzeichen) erfährt keine besondere Behandlung.

5.6 Zuweisungsoperatoren

Auch die Zuweisungsoperatoren in Java entsprechen im großen und ganzen den Zuweisungsoperatoren von C und C++. Ebenso gilt die Zuweisung nicht als *Anweisung*, sondern als *Ausdruck*, der einen Rückgabewert erzeugt.

Die Verwechslung der relationalen Operatoren *Zuweisung* und *Gleichheitstest* (= und ==) war in C eines der Kardinalprobleme, in Java kann sie nicht mehr so leicht passieren. Sind beispielsweise `a` und `b` vom Typ `int`, so hat zwar der Ausdruck `a = b` einen definierten Rückgabewert wie in C. Er darf jedoch nicht als Kontrollausdruck einer Schleife oder Verzweigung verwendet werden, da er nicht vom Typ `boolean` ist. Anders als in C, wo boolesche Werte durch Ganzzahlen simuliert werden, schließt Java diese Art von Fehler also von vorneherein aus. Nur wenn `a` und `b` vom Typ `boolean` sind, wird das Verwechseln von Zuweisung und Gleichheitstest vom Compiler nicht bemerkt.

Ebenso wie in C können auch in Java nummerische bzw. bitweise Operatoren mit der Zuweisung kombiniert werden. Der Ausdruck `a+=b` addiert `b` zu `a`, speichert das Ergebnis in `a` und liefert es ebenfalls als Rückgabewert zurück. Tabelle 5.5 gibt eine Übersicht der in Java verfügbaren Zuweisungsoperatoren.

Tabelle 5.5: Zuweisungsoperatoren

Operator	Bezeichnung	Bedeutung
=	Einfache Zuweisung	a = b weist a den Wert von b zu und liefert b als Rückgabewert.
+=	Additionszuweisung	a += b weist a den Wert von a + b zu und liefert a + b als Rückgabewert.
-=	Subtraktionszuweisung	a -= b weist a den Wert von a - b zu und liefert a - b als Rückgabewert.
*=	Multiplikationszuweisung	a *= b weist a den Wert von a * b zu und liefert a * b als Rückgabewert.
/=	Divisionszuweisung	a /= b weist a den Wert von a / b zu und liefert a / b als Rückgabewert.
%=	Modulozuweisung	a %= b weist a den Wert von a % b zu und liefert a % b als Rückgabewert.
&=	UND-Zuweisung	a &= b weist a den Wert von a & b zu und liefert a & b als Rückgabewert.
|=	ODER-Zuweisung	a |= b weist a den Wert von a | b zu und liefert a | b als Rückgabewert.
^=	Exklusiv-ODER-Zuweisung	a ^= b weist a den Wert von a ^ b zu und liefert a ^ b als Rückgabewert.

Tabelle 5.5: Zuweisungsoperatoren (Forts.)

Operator	Bezeichnung	Bedeutung
<<=	Linksschiebezuweisung	a <<= b weist a den Wert von a << b zu und liefert a << b als Rückgabewert.
>>=	Rechtsschiebezuweisung	a >>= b weist a den Wert von a >> b zu und liefert a >> b als Rückgabewert.
>>>=	Rechtsschiebezuweisung mit Nullexpansion	a >>>= b weist a den Wert von a >>> b zu und liefert a >>> b als Rückgabewert.

5.7 Sonstige Operatoren

Neben den bisher vorgestellten Operatoren stellt Java noch eine Reihe weiterer Operatoren zur Verfügung, die in diesem Abschnitt erläutert werden sollen.

5.7.1 Weitere Operatoren für primitive Typen

Fragezeichenoperator

Der Fragezeichenoperator `?:` ist der einzige dreistellige Operator in Java. Er erwartet einen logischen Ausdruck und zwei weitere Ausdrücke, die beide entweder nummerisch, von einem Referenztyp oder vom Typ `boolean` sind.

Bei der Auswertung wird zunächst der Wert des logischen Operators ermittelt. Ist dieser wahr, so wird der erste der beiden anderen Operanden ausgewertet, sonst der zweite. Das Ergebnis des Ausdrucks `a ? b : c` ist also `b`, falls `a` wahr ist, und `c`, falls `a` falsch ist. Der Typ des Rückgabewerts entspricht dem Typ des größeren der beiden Ausdrücke `b` und `c`.

Type-Cast-Operator

Ebenso wie in C gibt es auch in Java einen Type-Cast-Operator, mit dessen Hilfe explizite Typumwandlungen vorgenommen werden können. Der Ausdruck `(type) a` wandelt den Ausdruck `a` in einen Ausdruck vom Typ `type` um. Auch wenn `a` eine Variable ist, ist das Ergebnis von `(type) a` ein Ausdruck, der nicht mehr auf der linken, sondern nur noch auf der rechten Seite eines Zuweisungsoperators stehen darf.

Wie in Kapitel 4 auf Seite 93 erklärt, gibt es verschiedene Arten von Typkonvertierungen in Java. Mithilfe des Type-Cast-Operators dürfen alle legalen Typkonvertierungen vorgenommen werden. Der Type-Cast-Operator wird vor allem dann angewendet, wenn der Compiler keine impliziten Konvertierungen vornimmt; beispielsweise bei der Zuweisung von größeren an kleinere nummerische Typen oder bei der Umwandlung von Objekttypen.

5.7.2 Operatoren für Objekte

Es gibt in Java einige Ausdrücke und Operatoren, die mit Objekten arbeiten oder Objekte produzieren. Die meisten von ihnen können erst dann erläutert werden, wenn die entsprechenden Konzepte in späteren Kapiteln eingeführt wurden. Der Vollständigkeit halber sollen sie dennoch an dieser Stelle erwähnt werden.

String-Verkettung

Der +-Operator kann nicht nur mit nummerischen Operanden verwendet werden, sondern auch zur Verkettung von Strings. Ist wenigstens einer der beiden Operatoren in `a + b` ein String, so wird der gesamte Ausdruck als String-Verkettung ausgeführt. Hierzu wird gegebenenfalls zunächst der Nicht-String-Operand in einen String umgewandelt und anschließend mit dem anderen Operanden verkettet. Das Ergebnis der Operation ist wieder ein String, in dem beide Operanden hintereinanderstehen.

In Java ist die Konvertierung in einen String für nahezu jeden Typen definiert. Bei primitiven Typen wird die Umwandlung vom Compiler und bei Referenztypen durch die Methode `toString` ausgeführt. Die String-Verkettung ist daher sehr universell zu verwenden und ermöglicht (beispielsweise zu Ausgabezwecken) eine sehr bequeme Zusammenfassung von Ausdrücken unterschiedlichen Typs. Ein typisches Beispiel für die Verwendung der String-Verkettung ist die Ausgabe von nummerischen Ergebnissen auf dem Bildschirm:

Listing 5.3: String-Verkettung

```
001 /* Listing0503.java */
002
003 public class Listing0503
004 {
005   public static void main(String[] args)
006   {
007     int a = 5;
008     double x = 3.14;
009
010     System.out.println("a = " + a);
011     System.out.println("x = " + x);
012   }
013 }
```

Die Ausgabe des Programms lautet:

```
a = 5
x = 3.14
```

Etwas Vorsicht ist geboten, wenn sowohl String-Verkettung als auch Addition in einem Ausdruck verwendet werden sollen, da die in diesem Fall geltenden Vorrang- und Assoziativitätsregeln zu unerwarteten Ergebnissen führen können. Das folgende Programm gibt daher nicht `3 + 4 = 7`, sondern `3 + 4 = 34` aus.

Listing 5.4: Vorsicht bei der String-Verkettung!

```
001 /* Listing0504.java */
002
003 public class Listing0504
004 {
005   public static void main(String[] args)
006   {
007     System.out.println("3 + 4 = " + 3 + 4);
008   }
009 }
```

Um das gewünschte Ergebnis zu erzielen, müsste der Teilausdruck `3 + 4` geklammert werden:

Listing 5.5: Korrekte String-Verkettung bei gemischten Ausdrücken

```
001 /* Listing0505.java */
002
003 public class Listing0505
004 {
005   public static void main(String[] args)
006   {
007     System.out.println("3 + 4 = " + (3 + 4));
008   }
009 }
```

Referenzgleichheit und -ungleichheit

Die Operatoren == und != können auch auf Objekte, also auf Referenztypen, angewendet werden. In diesem Fall ist zu beachten, dass dabei lediglich die Gleichheit oder Ungleichheit der Referenz getestet wird. Es wird also überprüft, ob die Objektzeiger auf ein und dasselbe Objekt zeigen, und nicht, ob die Objekte *inhaltlich* übereinstimmen.

Ein einfaches Beispiel ist der Vergleich zweier Strings `a` und `b`, die beide den Inhalt "hallo" haben:

Listing 5.6: Vergleichen von Referenzen

```
001 /* Listing0506.java */
002
003 public class Listing0506
004 {
005   public static void main(String[] args)
006   {
007     String a = new String("hallo");
```

```
008     String b = new String("hallo");
009     System.out.println("a == b liefert " + (a == b));
010     System.out.println("a != b liefert " + (a != b));
011   }
012 }
```

Listing 5.6: Vergleichen von Referenzen (Forts.)

Werden sie zur Laufzeit angelegt (wie in diesem Beispiel), liefert das Programm das erwartete Ergebnis, denn `a` und `b` sind Referenzen auf unterschiedliche Objekte, also Zeiger auf unterschiedliche Instanzen derselben Klasse.

```
a == b liefert false
a != b liefert true
```

Dies ist das erwartete Verhalten für fast alle Objektreferenzen. Werden die Strings als Literale dagegen zur Compile-Zeit angelegt und damit vom Compiler als *konstant* erkannt, sind sie genau dann Instanzen derselben Klasse, wenn sie tatsächlich inhaltlich gleich sind. Dies liegt daran, dass String-Literale mithilfe der Methode `String.intern` angelegt werden, die einen Puffer von `String`-Objekten verwaltet, in dem jede Zeichenkette nur einmal vorkommt. Wird ein bereits existierender String noch einmal angelegt, so findet die Methode den Doppelgänger und liefert einen Zeiger darauf zurück. Dieses Verhalten ist so nur bei Strings zu finden, andere Objekte besitzen keine konstanten Werte und keine literalen Darstellungen. Die korrekte Methode, Strings auf inhaltliche Übereinstimmung zu testen, besteht darin, die Methode `equals` der Klasse `String` aufzurufen (siehe Listing 5.7).

```
001 /* Listing0507.java */
002
003 public class Listing0507
004 {
005   public static void main(String[] args)
006   {
007     String a = new String("hallo");
008     String b = new String("hallo");
009     System.out.println("a.equals(b) liefert " + a.equals(b));
010   }
011 }
```

Listing 5.7: Vergleichen von Strings mit equals

Der instanceof-Operator

Der `instanceof`-Operator kann verwendet werden, um herauszufinden, zu welcher Klasse ein bestimmtes Objekt gehört. Der Ausdruck `a instanceof b` liefert genau dann `true`, wenn `a` und `b` Referenztypen sind und `a` eine Instanz der Klasse `b` oder einer ihrer Unterklassen ist. Falls das Ergebnis des `instanceof`-Operators nicht bereits zur Compile-Zeit ermittelt werden kann, generiert der Java-Compiler Code, um den entsprechenden Check zur Laufzeit durchführen zu können.

Der new-Operator

In Java werden Objekte und Arrays mithilfe des `new`-Operators erzeugt. Sowohl das Erzeugen eines Arrays als auch das Erzeugen eines Objekts sind Ausdrücke, deren Rückgabewert das gerade erzeugte Objekt bzw. Array ist.

Member-Zugriff

Der Zugriff auf Klassen- oder Instanzvariablen wird mithilfe des *Punkt*-Operators ausgeführt und hat die Form `a.b`. Dabei ist `a` der Name einer Klasse bzw. der Instanz einer Klasse, und `b` ist der Name einer Klassen- oder Instanzvariable. Der Typ des Ausdrucks entspricht dem Typ der Variable, und zurückgegeben wird der Inhalt dieser Variable.

Methodenaufruf

In Java gibt es keine Funktionen, sondern nur *Methoden*. Der Unterschied zwischen beiden besteht darin, dass Methoden immer an eine Klasse oder die Instanz einer Klasse gebunden sind und nur in diesem Kontext aufgerufen werden können. Die Syntax des Methodenaufrufs gleicht der anderer Programmiersprachen und erfolgt in der (etwas vereinfachten) Form `f()` bzw. `f(parameterliste)`. Der Typ des Ausdrucks entspricht dem vereinbarten Rückgabetyp der Methode. Der Wert des Ausdrucks ist der von der Methode mithilfe der `return`-Anweisung zurückgegebene Wert.

Methoden können selbstverständlich Nebeneffekte haben und werden in vielen Fällen ausschließlich zu diesem Zweck geschrieben. Ist dies der Fall, so sollte eine Methode als `void` deklariert werden und damit anzeigen, dass sie keinen Rückgabewert produziert. Die einzig sinnvolle Verwendung einer solchen Methode besteht darin, sie innerhalb einer *Ausdrucksanweisung* (siehe Kapitel 6 auf Seite 129) aufzurufen.

Da die Realisierung der Methodenaufrufe in Java recht kompliziert ist (die Sprachspezifikation widmet diesem Thema mehr als 10 Seiten), werden wir in Kapitel 7 auf Seite 151 noch einmal ausführlich darauf eingehen.

Zugriff auf Array-Elemente

Wie in anderen Programmiersprachen erfolgt auch in Java der Zugriff auf Array-Elemente mithilfe eckiger Klammern in der Form `a[b]` (bzw. `a[b][c]`, `a[b][c][d]` usw. bei mehrdimensionalen Arrays). Dabei ist `a` der Name eines Arrays oder ein Ausdruck, der zu einem Array ausgewertet wird, und `b` ein Ausdruck, der zu einem `int` evaluiert werden kann. Der Typ des Ausdrucks entspricht dem Basistyp des Arrays, zurückgegeben wird der Inhalt des Array-Elements, das sich an Position `b` befindet. Wie in C und C++ beginnt die Zählung mit dem ersten Element bei Position 0.

5.7.3 Welche Operatoren es nicht gibt

Da es in Java keine expliziten Pointer gibt, fehlen auch die aus C bekannten Operatoren `*` zur Dereferenzierung eines Zeigers und `&` zur Bestimmung der Adresse einer Variablen. Des weiteren fehlt ein `sizeof`-Operator, denn da alle Typen eine genau spezifizierte Länge haben, ist dieser überflüssig. Der Kommaoperator von C ist ebenfalls nicht vorhanden, er taucht aber als syntaktischer Bestandteil der `for`-Schleife wieder auf und erlaubt es, im Initialisierungsteil der Schleife mehr als eine Zuweisung vorzunehmen.

5.8 Operator-Vorrangregeln

Tabelle 5.6 listet alle Operatoren in der Reihenfolge ihrer Vorrangregeln auf. Weiter oben stehende Operatoren haben dabei Vorrang vor weiter unten stehenden Operatoren. Innerhalb derselben Gruppe stehende Operatoren werden entsprechend ihrer Assoziativität ausgewertet.

Die Spalte *Typisierung* gibt die möglichen Operandentypen an. Dabei steht »N« für nummerische, »I« für integrale (also ganzzahlig nummerische), »L« für logische, »S« für String-, »R« für Referenz- und »P« für primitive Typen. Ein »A« wird verwendet, wenn alle Typen in Frage kommen, und mit einem »V« wird angezeigt, dass eine Variable erforderlich ist.

Tabelle 5.6: Operator-Vorrangregeln

Gruppe	Operator	Typisierung	Assoziativität	Bezeichnung
1	++	N	R	Inkrement
	--	N	R	Dekrement
	+	N	R	Unäres Plus
	-	N	R	Unäres Minus
	~	I	R	Einerkomplement
	!	L	R	Logisches NICHT
	(type)	A	R	Type-Cast
2	*	N,N	L	Multiplikation
	/	N,N	L	Division
	%	N,N	L	Modulo
3	+	N,N	L	Addition
	-	N,N	L	Subtraktion
	+	S,A	L	String-Verkettung
4	<<	I,I	L	Linksschieben
	>>	I,I	L	Rechtsschieben

Tabelle 5.6: Operator-Vorrangregeln (Forts.)

Gruppe	Operator	Typisierung	Assoziativität	Bezeichnung
	>>>	I,I	L	Rechtsschieben mit Null-expansion
5	<	N,N	L	Kleiner
	<=	N,N	L	Kleiner gleich
	>	N,N	L	Größer
	>=	N,N	L	Größer gleich
	instanceof	R,R	L	Klassenzugehörigkeit
6	==	P,P	L	Gleich
	!=	P,P	L	Ungleich
	==	R,R	L	Referenzgleichheit
	!=	R,R	L	Referenzungleichheit
7	&	I,I	L	Bitweises UND
	&	L,L	L	Logisches UND mit vollständiger Auswertung
8	^	I,I	L	Bitweises Exklusiv-ODER
	^	L,L	L	Logisches Exklusiv-ODER
9	|	I,I	L	Bitweises ODER
	|	L,L	L	Logisches ODER mit vollständiger Auswertung
10	&&	L,L	L	Logisches UND mit Short-Circuit-Evaluation
11	||	L,L	L	Logisches ODER mit Short-Circuit-Evaluation
12	?:	L,A,A	R	Bedingte Auswertung
13	=	V,A	R	Zuweisung
	+=	V,N	R	Additionszuweisung
	-=	V,N	R	Subtraktionszuweisung
	*=	V,N	R	Multiplikationszuweisung
	/=	V,N	R	Divisionszuweisung
	%=	V,N	R	Restwertzuweisung
	&=	N,N u. L,L	R	Bitweises-UND-Zuweisung und Logisches-UND-Zuweisung
	|=	N,N u. L,L	R	Bitweises-ODER-Zuweisung und Logisches-ODER-Zuweisung

Gruppe	Operator	Typisierung	Assoziativität	Bezeichnung
	^=	N,N u. L,L	R	Bitweises-Exklusiv-ODER-Zuweisung und Logisches-Exklusiv-ODER-Zuweisung
	<<=	V,I	R	Linksschiebezuweisung
	>>=	V,I	R	Rechtsschiebezuweisung
	>>>=	V,I	R	Rechtsschiebezuweisung mit Nullexpansion

Tabelle 5.6: Operator-Vorrangregeln (Forts.)

Etwas unschön ist die Tatsache, dass die bitweisen Operatoren schwächer binden als die relationalen Operatoren. Da sie auf einer Stufe mit den zugehörigen logischen Operatoren stehen, gibt es beim Übersetzen des folgenden Programms den Fehler »Incompatible type for &. Can't convert int to boolean«:

Listing 5.8: Bindungsprobleme bei den bitweisen Operatoren

```
001 /* Listing0508.java */
002
003 public class Listing0508
004 {
005   public static void main(String[] args)
006   {
007     int i = 55;
008     int j = 97;
009     if (i & 15 < j & 15) {
010       System.out.println("LowByte(55) < LowByte(97)");
011     } else {
012       System.out.println("LowByte(55) >= LowByte(97)");
013     }
014   }
015 }
```

Bei der Verwendung der bitweisen Operatoren sind also zusätzliche Klammern erforderlich. Die korrekte Version des Programms zeigt Listing 5.9 (verbessert wurde Zeile 009):

Listing 5.9: Korrekte Klammerung von bitweisen Operatoren

```
001 /* Listing0509.java */
002
003 public class Listing0509
004 {
005   public static void main(String[] args)
006   {
007     int i = 55;
008     int j = 97;
009     if ((i & 15) < (j & 15)) {
010       System.out.println("LowByte(55) < LowByte(97)");
011     } else {
```

Listing 5.9: Korrekte Klammerung von bitweisen Operatoren (Forts.)

```
012       System.out.println("LowByte(55) >= LowByte(97)");
013     }
014   }
015 }
```

Die Ausgabe des Programms ist nun erwartungsgemäß:

```
LowByte(55) >= LowByte(97)
```

5.9 Zusammenfassung

In diesem Kapitel wurden folgende Themen behandelt:

- Grundlegende Eigenschaften von Ausdrücken
- Die arithmetischen Operatoren +, -, *, /, %, ++ und --
- Die relationalen Operatoren ==, !=, <, <=, > und >=
- Die logischen Operatoren !, &&, ||, &, | und ^
- Die bitweisen Operatoren ~, |, &, ^, >>, >>> und <<
- Die Zuweisungsoperatoren =, +=, -=, *=, /=, %, &=, |=, ^=, <<=, >>= und >>>=
- Der Fragezeichenoperator ?:, der Operator für Typumwandlungen und der String-Verkettungs-Operator +
- Die Operatoren `new` und `instanceof`
- Die Operatoren für Member- und Array-Zugriff und der Operator zum Aufrufen von Methoden
- Die Operator-Vorrangregeln von Java