Operatoren und Ausdrücke

Im vorherigen Abschnitt haben wir *Variablen* und *Datentypen* kennengelernt. Wir können Variablen deklarieren und initialisieren und ihnen neue Werte zuweisen. Nun führen wir Operationen ein, die wir in den jeweiligen Datentypen verwenden können, um neue Werte zu erzeugen. Wir beginnen bei den arithmetischen Operatoren, d.h. mit den Operatoren, die wir für die ganzzahligen Datentypen und die Gleitkomma-Datentypen verwenden können.

Arithmetische Operatoren

Arithmetische Operationen kennen wir natürlich schon. Die einfachsten arithmetischen Operatoren sind die *unären* Operatoren, auch Vorzeichenoperatoren genannt. Ansonsten gibt es die Addition +, die Subtraktion -, die Multiplikation \* und die Division /. Außerdem gibt es auch einen Restwertoperator (auch *modulo* genannt), der bei der ganzzahligen Division den verbleibenden Rest als Ergebnis ermittelt (%).

Ganz **wichtig** ist, dass das Divisionssymbol / eine unterschiedliche Bedeutung hat, je nachdem, ob wir ganze Zahlen dividieren oder Gleitkommazahlen. Bei der Division von ganzen Zahlen ist das Ergebnis der Division der ganzzahlige Wert (also z.B. 5 / 4 ergibt 1), aber bei der Division von Gleitkommazahlen ein Gleitkommawert (also z.B. 5.0 / 4.0 ergibt 1.25).

Der Restwertoperator wird wirklich sinnvoll eigentlich nur für ganze Zahlen verwendet (also z.B. 7 % 4 ergibt 3 - der verbleibende Rest der ganzzahligen Divsion ist 3). Trotzdem kann der Restwertoperator auch auf Gleitkommazahlen angewendet werden (obwohl dort ja eigentlich kein Rest bleibt). So ergibt 7.0 % 4.0 auch 3.0 und/aber 7.5 % 4.0 ergibt 3.5.

| Operatorsymbol(e) | Bedeutung | Beispiel |
| --- | --- | --- |
| **Unäre Operatoren** | | |
| + | Wird als Vorzeichen vor ganzen Zahlen (z.B. +5) oder vor Gleikommazahlen (z.B. +5.5) verwendet. + ändert nichts und kann immer weggelassen werden. Verhalten für ganze Zahlen und Gleitkommazahlen gleich. | int x = -5; x = +x; Wert von x ist -5 |
| - | Wird als Vorzeichen vor ganzen Zahlen (z.B. -5) oder vor Gleikommazahlen (z.B. -5.5) verwendet. -x dreht das Vorzeichen von x um. Verhalten für ganze Zahlen und Gleitkommazahlen gleich. | int x = -5; x = -x; Wert von x ist +5 double y = -5.5; y = -y; Wert von y ist +5.5 |
| **Binäre Operatoren** | | |
| + | Addition von Zahlen. Verhalten für ganze Zahlen und Gleitkommazahlen gleich. | int x = -5 + 6; Wert von x: +1 double y = -5.5 + 6.5; Wert von y: +1.0 |
| - | Subtraktion von Zahlen. Verhalten für ganze Zahlen und Gleitkommazahlen gleich. | int x = -5 - 6; Wert von x ist -11 double y = -5.5 - 6.5; Wert von y: -12.0 |
| \* | Multiplikation von Zahlen. Verhalten für ganze Zahlen und Gleitkommazahlen gleich. | int x = -5 \* 6; Wert von x: -30 double y = -5.0 \* 6.0; Wert von y: -30.0 |
| / | Division von Zahlen. **Achtung! Sie müssen unterscheiden, ob Sie ganze Zahlen dividieren oder Gleitkommazahlen!** Bei ganzen Zahlen handelt es sich um die **ganzzahlige** Division! | int x = 5 / 4; Wert von x: +1 double y = 5.0 / 4.0; Wert von y: +1.25 |
| % | Restwertoperator. Ergebnis ist der Rest, der bei ganzzahliger Division übrig bleibt. Wirklich sinnvoll nur bei ganzzahligen Werten, geht aber auch bei Gleitkommazahlen. | int x = 7 % 4; Wert von x ist 3 double y = 6.5 % 5.0; Wert von y ist 1.5 |

Prä- und Postfix-Operatoren

Darüber hinaus gibt es noch besondere Operatoren, die eingeführt wurden, weil sie eine häufig vorkommende Operation in der Schreibweise verkürzen.

Angenommen, wir haben eine Variable int x = 5;. Da es (insbesondere später in Schleifen) häufig vorkommt, dass der Wert dieser Variablen um 1 erhöht werden soll, hat man dafür einen eigenen Operator eingeführt: ++. Dieser Operator steht für die Erhöhung des Wertes um den Summanden 1. Er kann sowohl als Präfix-Operator (++x) als auch als Postfix-Operator eingesetzt werden. Man spricht hier von Prä- und Postfix-**Inkrement**-Operatoren.

|  |  |
| --- | --- |
| 1  2  3  4  5 | int x = 5;  System.out.println(x++); // 5  System.out.println(x); // 6  System.out.println(++x); // 7  System.out.println(x); // 7 |

Der Unterschied zwischen den beiden wird im obigen Beispiel deutlich. x hat nach der Initialisierung den Wert 5. In Zeile 2 geben wir den Wert aus und erhöhen x **danach** um 1. Ausgegeben wird also der Wert 5, aber nach der Ausgabe ist der Wert um 1 erhöht auf 6 (siehe Zeile 3). Bei dem Prefix-Operator (Zeile 4) wird der Wert **erst** erhöht und dann ausgelesen. Es wird also bereits der neue Wert 7 ausgegeben. Im Prinzip entsprechen beide Operatoren aber der Anweisung x = x + 1;.

Den äquivalenten Operator gibt es auch für die Subtraktion minus 1. Dies wird mit dem Prefix-Operator --x; sowie mit dem Postfix-Operator x--; erreicht. Diese heißen Prä- und Postfix-**Dekrement**-Operatoren.

Verkürzte Schreibweisen für arithmetische Operatoren

Aus dieser Verwendung haben sich weitere Schreibweisen etabliert:

| Operatoren (Beispiele) | Bedeutung |
| --- | --- |
| x+=7; | x = x + 7; |
| x-=7; | x = x - 7; |
| x\*=7; | x = x \* 7; |
| x/=7; | x = x / 7; |
| x%=7; | x = x % 7; |

Vergleichsoperatoren

Werden die oben genannten Operatoren auf arithmetische Datentypen angewendet, so ist das Ergebnis selbst wieder von einem arithmetischen Datentyp (int oder double). Bei den folgenden *Vergleichsoperatoren* ist das anders. Die Operanden sind zwar auch wieder von arithmetischen Datentypen, aber das Ergebnis ist ein boolean, das heißt, der Vergleich resultiert in einem Wert true oder in einem Wert false.

| Vergleichsoperator | Beispiel | Bedeutung |
| --- | --- | --- |
| == | x == y | Vergleich auf Gleichheit. Ergibt true, wenn x den gleichen Wert hat wie y, sonst false |
| < | x < y | Kleiner als. Ergibt true, wenn x einen echt kleineren Wert hat als y, sonst false |
| > | x > y | Größer als. Ergibt true, wenn x einen echt größeren Wert hat als y, sonst false |
| <= | x <= y | Kleiner gleich. Ergibt true, wenn x einen kleineren als oder den gleichen Wert wie y hat, sonst false |
| >= | x >= y | Größer gleich. Ergibt true, wenn x einen größeren als oder den gleichen Wert wie y hat, sonst false |
| != | x != y | Ungleich. Ergibt true, wenn x einen anderen Wert hat als y, sonst false |

Wertzuweisungsoperator

Einen weiteren Operator kennen wir bereits, den Wertzuweisungsoperator =. Auf der linken Seite der Zuweisung steht immer eine Variable und auf der rechten Seite ein Wert, der sich auch aus einem Ausdruck ergeben kann, z.B. int x = 5 + 6;. Dann wird xder Wert 11zugeweisen.

**Beachte**

Der Zuweisungsoperator ist zunächst gewöhnungsbedürftig, denn eine Zuweisung der Form x = x+1; sieht ja aus mathematischer Sicht komisch aus, da in der Mathematik das Symbol = für die Gleichheit verwendet wird. In der (Java-)Programmierung hat dieses Symbol aber eine andere Bedeutung, nämlich die Zuweisung des Wertes auf der rechten Seite zur Variablen auf der linken Seite. In der Zuweisung x = x+1; wird also erst der Wert x+1 berechnet und dieser Wert dann der Variablen x zugeordnet (die dann einen um 1 höheren Wert hat als zuvor). Der Operator für die Gleichheit ist übrigens ==, wie wir gleich sehen werden.

Logische Operatoren

Die arithmetischen Operatoren werden auf Operanden angewendet, die von einem ganzzahligen Datentyp oder von einem Gleitkomma-Datentyp sind. Jetzt lernen wir Operatoren kennen, die auf Operanden vom Typ boolean angewendet werden.

Nehmen wir in der folgenden Tabelle an, dass die Variablen a und b jeweils vom Typ boolean sind. Sie wurden also wie folgt deklariert: boolean a; und boolean b;.

| Logischer Operator | Beispiel | Bedeutung |
| --- | --- | --- |
| && | a && b | UND-Operator. Ergibt true, wenn a den Wert true hat **UND** b auch den Wert true, sonst false |
| || | a || b | ODER-Operator. Ergibt true, wenn a den Wert true hat **ODER** b den Wert true (oder beide), sonst false |
| ^ | a ^ b | EXCLUSIVES ODER. Ergibt true, wenn ENTWEDER a den Wert true hat **ODER** b den Wert true (aber NICHT beide), sonst false |
| ! | !a | NEGATION (unärer Operator). Ergibt true, wenn a den Wert false hat, ergibt false, wenn a den Wert true hat |

Es sein angemerkt, dass es für die Operatoren && und || jeweils auch die Operatoren & und | gibt. Das logische Prinzip ist das gleiche (also UND und ODER). Es gibt nur jeweils eine Unterscheidung und diese ist im folgenden Beispiel dargestellt:

false && a // a wird nicht mehr geprüft, Ausdruck ist false

false & a // a wird geprüft, Ausdruck ist false

true || b // b wird nicht mehr geprüft, Ausdruck ist true

true | b // b wird geprüft, Ausdruck ist true

Ist beim logischen UND bereits der erste Operand false, dann kann das Ergebnis nicht mehr true sein, sondern ist false. Egal, welchen Wert der zweite Operand hat. Bei dem Operator && wird dann der zweite Operand auch gar nicht mehr geprüft. Bei dem Operator & aber doch. && ist somit effizienter.

Ist beim logischen ODER bereits der erste Operand true, dann ist das Ergebnis bereits true, egal, welchen Wert der zweite Operand hat. Bei dem Operator || wird dann der zweite Operand auch gar nicht mehr geprüft. Bei dem Operator | aber doch. || ist somit effizienter. Wir verwenden deshalb immer && und || anstatt & und |.

**Wahrheitstabellen**

Hier nochmal eine Veranschaulichung der obigen logischen Operatoren in einer Wahrheitstabelle. Wir nehmen wieder an, dass die Variablen a und b jeweils vom Typ boolean sind.

| a | b | a && b | a || b | a ^ b | !a |
| --- | --- | --- | --- | --- | --- |
| true | true | true | true | false | false |
| true | false | false | true | true | false |
| false | true | false | true | true | true |
| false | false | false | false | false | true |

Übung Exclusives Oder

Ausdrücke und Anweisungen

*Anweisungen* haben wir bereits kennengelernt. Zum Beispiel sind Deklarationen Anweisungen und Initialisierungen auch. Neben Anweisungen gibt es im Programmcode auch *Ausdrücke*. Ausdrücke unterscheiden sich von Anweisungen dahingehened, dass sie einen Wert (genauer einen *Rückgabewert*) haben. Jeder Wert in Java hat einen Typ. Wir schauen uns zunächst die einfachsten Ausdrücke an, die es gibt, nämlich sogenannte *Literale*.

**Literale**

Ein *Literal* ist ein konstanter Wert im Java-Quellcode. Literale sind z.B.

* ganze Zahlen (z.B. 123 oder -123),
* Gleikommazahlen (z.B 5.5 oder -6.0),
* Wahrheitswerte (true oder false),
* ein Character (z.B. 'a' oder 'A')

Wir werden später noch andere Literale kennenlernen:

* Zeichenketten (Datentyp String), z.B. "Hallo FIW!"
* die leere Referenz null

Jedes Literal ist ein Ausdruck. Der Rückgabewert des Literals ist der Wert des Literals. Jedes Literal ist von einem konkreten Datentyp. Die folgenden Anweisungen werden auch *Ausdrucksanweisung* genannt, da in der Anweisung ein Ausdruck verwendet wird und der Wert dieses Ausdrucks als *Nebeneffekt* zum Tragen kommt (nur die ersten beiden sind Ausdrucksanweisungen - die folgenden drei sind **Fehler**):

int x = 123; // auf der rechten Seite des Zuweisungsoperators steht ein Ausdruck;

// der Wert des Ausdrucks wird der Wert der Variablen x

x++; // x++ ist ein Ausdruck - durch das Semikolon wird der Ausdruck hier zu einer Ausdrucksanweisung

x++ // der Ausdruck selbst kann so nicht stehen (Compilerfehler)

// da steht nur ein Wert mit dem nichts gemacht wird

5 + 6 // gleicher Fehler wie oben; nur ein Ausdruck, kann so nicht alleine stehen

5 + 6; // geht auch nicht; hier gibt es keinen Nebeneffekt (keine Zuweisung)

// keine gültige Ausdrucksanweisung

**Ausdrücke sind**

* Literale
* Literale, die mit (passenden) Operatoren verknüpft sind
* Ausdrücke, die mit passenden Operatoren verknüpft sind

**Ausdrücke mit Operatoren**

Verbinden wir Literale mit Operatoren, entstehen dadurch Ausdrücke. Ausdrücke können wiederum mithilfe von Operatoren mit weiteren Literalen oder auch mit weiteren Ausdrücken verbunden werden. Beachten Sie, dass Ausdrücke

* immer einen Wert haben,
* der Wert ermittelt wird, indem der Ausdruck *aufgelöst* wird, d.h. die Operatoren angewendet werden. Dies erfolgt in folgender Reihenfolge:
  + wenn runde Klammern um einen Ausdruck gesetzt sind, wird zunächst der geklammerte Ausdruck aufgerufen,
  + unäre (also einelementige) Operatoren binden stärker als binäre (also zweielementige) Operatoren, d.h. es werden zunächst die einelementigen Operatoren angewendet,
  + bei arithmetischen Operatoren gilt "Punkt vor Strich-Rechnung",
  + bei logischen Operatoren gilt folgende Reihenfolge (stärkere Bindung von links nach rechts) ==, != -> & → ^ → | → && → || (lernen Sie das aber keinesfalls auswendig und verlassen sich dann auf Ihr Gedächtnis, sondern verwenden Sie Klammern!)
  + der Zuweisungsoperator (=) sowie +=, -=, /=, \*=, %= binden am schwächsten
  + dann erfolgt die Auflösung von links nach rechts

Achten Sie darauf, dass Ausdrücke einen anderen Typ haben können als die Literale (bzw. Ausdrücke), die man im Ausdruck mithilfe von Operatoren miteinander verbindet, z.B. 3 == 4 ist vom Typ boolean, aber 3 und 4 sind jeweils vom Typ int.

Beispiele für Ausdrücke sind:

5 + 6 - 8 // Ergebnis (Wert) ist ein int

true && false // Ergebnis (Wert) ist ein boolean

5 < 6 // Ergebnis (Wert) ist ein boolean

7 >= 7 // Ergebnis (Wert) ist ein boolean

6 == 6 // Ergebnis (Wert) ist ein boolean

6 != 6 // Ergebnis (Wert) ist ein boolean

5.5 \* 2.0 // Ergebnis (Wert) ist ein double

7.0 / 4.0 // Ergebnis (Wert) ist ein double

(7.0 / 4.0 ) > 1.0 // Ergebnis (Wert) ist ein boolean

Übung AusdruckÜbung Durchschnitt berechnen

Bei der Verwendung des Zuweisungsoperators gibt es auf der linken Seite immer eine Variable und auf der rechten Seite immer einen Ausdruck. Bevor die Zuweisung erfolgt, wird der Wert des Ausdrucks auf der rechten Seite ausgewertet. Beispiele:

int x = 0; // Wert von x ist 0

x = 7 + 4 / 2; // Wert von x ist 9

int y = 7 % 4; // Wert von y ist 3

x = y++; // Achtung Wert von x ist 3 (Postfix); Wert von y ist 4

x = y; // Wert von x ist 4

boolean a = (7 > 4); // Wert von a ist true

a = !a; // Wert von a ist false

a = (a || true); // Wert von a ist true

a = !!a && true; // Wert von a ist true

**Beachte**

Generell gilt bei der Zuweisung **immer**, dass der Ausdruck auf der rechten Seite erst vollständig ausgerechnet wird und der berechnete Wert dann der Variablen auf der linken Seite zugewiesen wird. "Leider" macht der Postfix-Operator dabei eine **Ausnahme**. Bei diesem Operator erfolgt erst die Zuweisung und dann die Berechnung:

int x = 3; // x hat den Wert 3

int y = x++; // y hat den Wert 3 und x den Wert 4

int a = 3; // a hat den Wert 3

int b = a--; // b hat den Wert 3 und a den Wert 2

Das gilt nicht für den Präfix-Operator:

int x = 3; // x hat den Wert 3

int y = ++x; // y hat den Wert 4 und x den Wert 4

int a = 3; // a hat den Wert 3

int b = --a; // b hat den Wert 2 und a den Wert 2