

OPERATOREN IN JAVA

Ausdrücke und Operatoren im Überblick

Ausdrücke

- einfachste Form von Ausdrücken in JAVA → **einfache Ausdrücke**
 - Konstante
 - Variable
- ein Ausdruck besitzt immer einen **Wert**
 - Konstante → Wert der Konstanten
 - Variable → Inhalt (Wert) der Variablen



- der Wert eines Ausdrucks wird oft auch als **Rückgabewert** des Ausdrucks bezeichnet
- in JAVA stellt **alles**, was **einen Wert zurückliefert**, einen Ausdruck dar
- weitere Formen von Ausdrücken
 - ein **Methodenaufruf**, falls die Methode einen Rückgabewert liefert
 - Verknüpfung von **Operanden** durch **Operatoren**
 - **komplexe Ausdrücke**
 - ◇ ein Operand kann ein einfacher Ausdruck sein
 - ◇ ein Operand kann aber auch wieder ein komplexer Ausdruck sein
 - ▷ $a+b*c$ hier ist der zweite Operand der Addition selbst wieder ein Ausdruck → $a*b$

- wichtige Kriterien bei der Auswertung (Berechnung) eines Ausdrucks
 - **Auswertungsreihenfolge** der Operanden eines Operators
 - **Prioritäten** der einzelnen Operatoren
 - **Seiteneffekte** bei der Auswertung eines Ausdrucks

Operatoren

- die Operatoren können über die Anzahl der benötigten Operanden klassifiziert werden
- JAVA kennt folgende Arten von Operatoren
 - **einstellig** (*unär, monadisch*) → ein Operand
 - **zweistellig** (*binär, dyadisch*) → zwei Operanden
 - **dreistellig** (*ternär, tryadisch*) → drei Operanden

Seiteneffekte

- ein **Seiteneffekt** (*Nebeneffekt, Nebenwirkung*) tritt bei der Bearbeitung eines Ausdrucks auf, wenn neben der Berechnung des Ausdruckwertes z.B. noch Variablen verändert werden
- Verursacher von Seiteneffekten
 - Operatoren
 - Methodenaufrufe

Ausdrucksanweisungen

- eine (wichtige) Abgrenzung zwischen **Ausdrücken** und **Anweisungen**
 - ein Ausdruck hat immer einen Rückgabewert
 - eine Anweisung hat **keinen** Rückgabewert
- Ausdrucksanweisung
 - bestimmte Arten von Ausdrücken können durch das **Anhängen** eines **Semikolons** zu einer Anweisung werden
→ **Ausdrucksanweisung**
- die folgenden Ausdrücke können zu einer Anweisung werden
 - Zuweisungen → a = b
 - die Anwendung eines Inkrement- oder Dekrement-Operators auf eine Variable → a++
 - Methodenaufruf
 - Ausdrücke, die mit **new** ein Objekt erzeugen

Arithmetische Operatoren

Arithmetische Operatoren auf numerischen Typen

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo-Operation

Tab. OPE-1

Binäre arithmetische Operatoren

- die in Tab. OPE-1 aufgeführten **binären Operatoren** sind für **alle numerischen** Typen (**byte, short, int, char, long, float** und **double**) gültig
- das Ergebnis einer Modulo-Operation ist der **Rest**, der bei einer **Division** auftritt → 13 % 5=3 -7 % -2=-1
- zusätzlich sind folgende **unäre Operatoren** definiert
 - unäres - für die **Negation**
 - ◇ Beispiel: Vorzeichenwechsel einer Zahl wert = -wert
 - unäres +
 - ◇ nur aus Symmetriegründen vorhanden

Integer-Arithmetik

- die Integer-Arithmetik ist eine reine **Zweierkomplement-Arithmetik**
 - überschreitet das Ergebnis einer Operation den Wertebereich seines Typs, so wird der Ergebniswert auf den darstellbaren Bereich **reduziert**
 - es tritt weder ein **Über-** noch ein **Unterlauf** auf
 - Beispiel: die Addition der **byte**-Werte 120 und 35 liefert das Ergebnis -101
- Integer-Division
 - **Abschneiden** der Nachkommastellen des Ergebnisses
 - ◊ Beispiele: $7/2=3$ $-7/2=-3$ $8/3=2$ $-8/3=-2$
 - damit gilt folgende Gleichung: $(x/y)*y + x\%y == x$
- Division oder Modulo-Operationen mit 0 sind unzulässig
 - Auslösen einer `ArithmeticException`

Gleitpunktarithmetik

- Verwendung des *IEEE 754-1985* Standards zur **Gleitpunktarithmetik** für die Zahlenbereiche **float** und **double**
- wichtige Eigenschaften
 - **Überschreitung** des Zahlenbereichs liefert ∞ oder $-\infty$
 - **Unterschreitung** des Zahlenbereichs liefert 0.0 oder -0.0
 - Ergebnisse von **ungültigen** Operationen liefern den Wert NaN
 - (**Not a Number**)
 - der Vergleich von -0.0 und 0.0 liefert den Wert *true*
 - arithmetische Operationen, die einen NaN-Operanden besitzen, liefern NaN
- Erzeugung der Werte ∞ bzw. $-\infty$
 - durch eine entsprechende Operation → vgl. Tab. OPE-2 (S. 7)
 - ◊ in der Tabelle stellt a eine **double**-Zahl mit $a>0$ dar
 - Verwendung von
 - ◊ `Double.POSITIVE_INFINITY`
 - ◊ `Double.NEGATIVE_INFINITY`
 - ◊ `Float.POSITIVE_INFINITY`
 - ◊ `Float.NEGATIVE_INFINITY`

x	y	x+y	x-y	x*y	x/y	x%/y	-x
∞	∞	∞	NaN	∞	NaN	NaN	-∞
∞	-∞	NaN	∞	-∞	NaN	NaN	-∞
-∞	∞	NaN	-∞	-∞	NaN	NaN	∞
-∞	-∞	-∞	NaN	∞	NaN	NaN	∞
a	∞	∞	-∞	∞	0.0	a	-a
a	-∞	-∞	∞	-∞	-0.0	a	-a
-a	∞	∞	-∞	-∞	-0.0	-a	a
-a	-∞	-∞	∞	∞	0.0	-a	a
∞	a	∞	∞	∞	∞	NaN	-∞
∞	-a	∞	∞	-∞	-∞	NaN	-∞
-∞	a	-∞	-∞	-∞	-∞	NaN	∞
-∞	-a	-∞	-∞	∞	∞	NaN	∞
0.0	∞	∞	-∞	NaN	0.0	0.0	-0.0
0.0	-∞	-∞	∞	NaN	-0.0	0.0	-0.0
-0.0	∞	∞	-∞	NaN	-0.0	-0.0	0.0
-0.0	-∞	-∞	∞	NaN	0.0	-0.0	0.0
∞	0.0	∞	∞	NaN	∞	∞	-∞
∞	-0.0	∞	∞	NaN	-∞	-∞	-∞
-∞	0.0	-∞	-∞	NaN	-∞	-∞	∞
-∞	-0.0	-∞	-∞	NaN	∞	∞	∞
a	0.0	a	a	0.0	∞	a	-a
a	-0.0	a	a	-0.0	-∞	a	-a
-a	0.0	-a	-a	-0.0	-∞	-a	a
-a	-0.0	-a	-a	0.0	∞	-a	a
0.0	a	a	-a	0.0	0.0	0.0	-0.0
0.0	-a	-a	a	-0.0	-0.0	0.0	-0.0
-0.0	a	a	-a	-0.0	-0.0	-0.0	0.0
-0.0	-a	-a	a	0.0	0.0	-0.0	0.0
0.0	0.0	0.0	0.0	0.0	NaN	0.0	-0.0
0.0	-0.0	0.0	0.0	-0.0	NaN	0.0	-0.0
-0.0	0.0	0.0	-0.0	-0.0	NaN	-0.0	0.0
-0.0	-0.0	-0.0	0.0	0.0	NaN	-0.0	0.0

Tab. OPE-2

Gleitpunktarithmetik in JAVA

String-Verkettung

- Zeichenketten können mit + verkettet werden
- die Ausgabe des Programms Prg. OPE-1 liefert:
juhuh!

```
public class StringVerkettung {
    public static void main ( String[] args ) {
        String ju = "ju";
        String schrei = ju + "huh";
        schrei = schrei + "!";
        System.out.println ( schrei );
    } // main
} // StringVerkettung
```

Prg. OPE-1

Beispiel für die String-Verkettung


- **implizite Typumwandlung** bei der Verwendung des Operators +
 - besitzt ein Operand den Typ `String` und der zweite Operand einen **einfachen Typ**, so wird für diesen Operanden automatisch eine Konvertierung in einen `String` vorgenommen
 - die Ausgabe des Programms Prg. OPE-2 liefert:
Pi hat den Wert 3.14

```
public class AusgabePI {
    public static void main ( String[] args ) {
        double pi = 3.14;
        String ausgabe = "Pi hat den Wert ";
        ausgabe = ausgabe + pi;
        System.out.println ( ausgabe );
    } // main
} // AusgabePI
```

Prg. OPE-2

Beispiel für eine implizite Typumwandlung

Inkrement- und Dekrement-Operatoren

- Operatoren
 - Inkrement-Operator: ++
 - Dekrement-Operator: --
 - die Operatoren sind für alle **numerischen Typen** (`byte`, `short`, `int`, `char`, `long`, `float` und `double`) definiert
 - **Postfix**-Anwendung der Operatoren (auf der Variablen `index`)
 - `index++` bzw. `index--`
 - Bedeutung
 - ◊ der (alte) Wert von `index` wird gelesen und als Ergebnis des Ausdrucks zurückgeliefert
 - ◊ danach wird `index` inkrementiert (Addition von 1) bzw. dekrementiert (Subtraktion von 1)
 - **Präfix**-Anwendung der Operatoren (auf der Variablen `index`)
 - `++index` bzw. `--index`
 - Bedeutung
 - ◊ `index` wird inkrementiert bzw. dekrementiert
 - ◊ danach wird der (neue) Wert von `index` gelesen und als Ergebnis des Ausdrucks zurückgeliefert
-  das Inkrementieren bzw. Dekrementieren stellt einen **Seiteneffekt** der Ausdrucksauswertung dar

- Beispiel: Postfix- und Präfix-Anwendung in Ausdrucksanweisungen
 - siehe Prg. OPE-3
 - das Programm liefert die folgende Ausgabe:


```
index=8
index=7
```
 - die `System.out.println`-Anweisungen enthalten jeweils eine implizite Typumwandlung

```
public class InkrementDekrementBeispiel {
    public static void main ( String[] args ) {
        int index = 7;
        index++; // Postfix-Anwendung
        System.out.println ( "index=" + index );
        --index; // Präfix-Anwendung
        System.out.println ( "index=" + index );
    } // main
} // InkrementDekrementBeispiel
```

Prg. OPE-3

Anwendung der Inkrement- und Dekrement-Operatoren

- Anwendung auf einer **char**-Variablen
 - ++ und -- können auch auf **char**-Variablen angewendet werden, um zum nächsten bzw. vorherigen UNICODE-Zeichen zu gelangen
 - die Ausgabe des Programms Prg. OPE-4 liefert:


```
buchstabe=d
buchstabe=c
```

```
public class InkrementDekrementChar {
    public static void main ( String[] args ) {
        char buchstabe = 'c';
        buchstabe++;
        System.out.println ( "buchstabe=" + buchstabe );
        --buchstabe;
        System.out.println ( "buchstabe=" + buchstabe );
    } // main
} // InkrementDekrementChar
```

Prg. OPE-4

Anwendung der Inkrement- und Dekrement-Operatoren auf einer **char**-Variablen

- bei den bisherigen Beispielen war es unerheblich, ob die Prä- oder Postfix-Anwendung des Inkrement- bzw. Dekrement-Operators gewählt wurde
- dies gilt **nicht** mehr, wenn das **Ergebnis** des Inkrement- bzw. Dekrement-Ausdrucks noch innerhalb der Anweisung **weiterverwendet** wird

- Beispiel: Auswertung von Ausdrücken → siehe Prg. OPE-5 (S. 14)
- das Programm liefert die folgende Ausgabe (mit jeweils a=5):
 - b = a++ : b=5 a=6
 - b = ++a : b=6 a=6
 - b = a++ -a : b=-1 a=6
- Verarbeitung von b = a++;
 - ◇ Lesen des Wertes von a (=5) und internes Zwischenspeichern
 - ◇ Inkrementieren von a ⇒ a=6
 - ◇ Zuweisung des intern zwischengespeicherten Wertes an b ⇒ b=5
- Verarbeitung von b = ++a;
 - ◇ Inkrementieren von a ⇒ a=6
 - ◇ Lesen des Wertes von a und Zuweisung an b ⇒ b=6
- Verarbeitung von b = a++ -a;
 - ◇ Lesen des Wertes von a (=5) und internes Zwischenspeichern
 - ◇ Inkrementieren von a ⇒ a=6
 - ◇ Subtraktion des neuen Wertes von a (=6) von dem intern zwischengespeicherten Wert → Ergebnis -1
- ◇ Zuweisung dieses Ergebnisses an b ⇒ b=-1

```

public class InkrementAusdruck {
public static void main ( String[] args ) {
int b, a=5;
b = a++;
System.out.println ( "b = a++ : b="
+ b + " a=" + a );
a=5;
b = ++a;
System.out.println ( "b = ++a : b="
+ b + " a=" + a );
a=5;
b = a++ -a;
System.out.println ( "b = a++ -a : b="
+ b + " a=" + a );
} // main
} // InkrementAusdruck

```

Prg. OPE-5

Auswertung von Ausdrücken

Vergleichsoperatoren

- Vergleichsoperatoren in JAVA → siehe Tab. OPE-3
 - das Ergebnis eines Vergleichs ist ein **Boolescher** Wert

Operator	Bedeutung
>	größer
>=	größer oder gleich
<	kleiner
<=	kleiner oder gleich
==	ist gleich
!=	ungleich

Tab. OPE-3

Vergleichsoperatoren in JAVA

- Bemerkungen
 - hat mindestens einer der Operanden in $x \text{ op } y$ ($\text{op} \in \{<, <=, ==, >=, >\}$) den Wert NaN, so ist das Ergebnis *false*
 - ◊ insbesondere gilt: `Double.NaN==Double.NaN` liefert *false*
 - hat mindestens einer der Operanden in $x \text{ != } y$ den Wert NaN, so ist das Ergebnis *true*
 - ◊ insbesondere gilt: `Double.NaN!=Double.NaN` liefert *true*

Logische Operatoren

- **Logische** Operatoren in JAVA → siehe Tab. OPE-4

Operator	Bedeutung
!	Negation
&&	bedingtes UND
	bedingtes ODER

Tab. OPE-4

Logische Operatoren in JAVA

- Bemerkungen
 - bei `x && y` wird `y` nur ausgewertet, wenn `x` den Wert *true* besitzt
 - bei `x || y` wird `y` nur ausgewertet, wenn `x` den Wert *false* besitzt
- die Eigenschaft der **bedingten Auswertung** spielt eine Rolle, wenn `y` ein Ausdruck ist, der einen **Seiteneffekt** besitzt
 - Prg. OPE-7 (S. 19) zeigt ein Beispiel für einen Seiteneffekt
- Beispiel: Erzeugung einer Wahrheitstabelle für den `&&`-Operator → siehe Prg. OPE-6 (S. 17)



```

public class WahrheitstabelleUnd {
    public static void main ( String[] args ) {
        System.out.println ( "a      b      a&&b" );
        System.out.println ( "-----" );
        boolean a = true, b = true, c = a&&b;
        System.out.println ( a + "   " + b + "   " + c );
        b = false;
        c = a&&b;
        System.out.println ( a + "   " + b + "   " + c );
        a = false; b = true;
        c = a&&b;
        System.out.println ( a + "   " + b + "   " + c );
        b = false;
        c = a&&b;
        System.out.println ( a + "   " + b + "   " + c );
    } // main
} // WahrheitstabelleUnd
    
```

Prg. OPE-6

Wahrheitstabelle für den &&-Operator

- das Programm liefert die folgende Ausgabe:

```

a      b      a&&b
-----
true   true   true
true   false  false
false  true   false
false  false  false
    
```

Bitoperatoren

Logische Bitoperatoren

- Tab. OPE-5 zeigt die von JAVA unterstützten **logischen** (auch: *booleschen*) Bitoperatoren

Operator	Bedeutung
&	binärer Bitoperator UND
	binärer Bitoperator (Inklusives) ODER
^	binärer Bitoperator Exklusives ODER
~	unärer Bitoperator Komplement (Negation)

Tab. OPE-5

Logische Bitoperatoren

- Bitoperatoren sind nur für den **boolean**- und die Integer-Datentypen definiert

- Anwendung bei **boolean**-Operanden
 - im Gegensatz zu den *bedingten logischen Operatoren* `&&` und `||` werden bei den *logischen Bitoperatoren* `&` und `|` immer *beide* Operanden ausgewertet
 - Prg. OPE-7 zeigt das Auftreten eines *Seiteneffekts* bei der Verwendung des `&`-Operators und liefert folgendes Ergebnis:

```
&-Operator: b=false i=12
&&-Operator: b=false i=11
```

```
public class Seiteneffekt {
    public static void main ( String[] args ) {
        int i = 11;
        boolean b = 1<0 & 2<i++;
        System.out.println ( "&-Operator: b="
            + b + " i=" + i );
        i = 11;
        b = 1<0 && 2<i++;
        System.out.println ( "&&-Operator: b="
            + b + " i=" + i );
    } // main
} // Seiteneffekt
```

Prg. OPE-7

Logisches und binäres UND

- Anwendung bei Integer-Operanden
 - die Operatoren werden *bitweise* auf der *Binärdarstellung* der Integer-Operanden ausgeführt
 - bei der Verwendung von **int**-Operanden werden somit 32 Einzeloperationen ausgeführt
 - Prg. OPE-8 zeigt die Verwendung der logischen Bitoperatoren auf dem Datentyp **byte** und liefert folgendes Ergebnis:

```
i&j = 2
i|j = 27
i^j = 25
~i = -11
```

```
public class Bitoperatoren {
    public static void main ( String[] args ) {
        byte i = 10;
        byte j = 19;
        System.out.println ( "i&j = " + (i&j) );
        System.out.println ( "i|j = " + (i|j) );
        System.out.println ( "i^j = " + (i^j) );
        System.out.println ( "~i = " + (~i) );
    } // main
} // Bitoperatoren
```

Prg. OPE-8

Anwendung der logischen Bitoperatoren



innerhalb der `System.out.println`-Anweisungen muß z.B. `i|j` eingeklammert werden, da sonst ein Übersetzungsfehler auftritt

Verschiebeoperatoren

- Tab. OPE-6 zeigt die von JAVA unterstützten **Verschiebe-Operatoren** (*Shift-Operatoren*)

Operator	Bedeutung
<<	Verschieben des Operanden bitweise nach links Auffüllen mit 0-Bits von rechts
>>	Verschieben des Operanden bitweise nach rechts Auffüllen mit dem höchstwertigen Bit (Vorzeichen) von links
>>>	Verschieben des Operanden bitweise nach rechts Auffüllen mit 0-Bits von links

Tab. OPE-6

Shift-Operatoren

- Bemerkungen
 - Shift-Operatoren können nur auf Integer-Typen angewendet werden
 - die Anzahl der Stellen, um die ein n -Bit-Integer-Operand verschoben wird, sollte maximal $n-1$ sein
- Beispiel:
 - Verschieben eines Operandens um 3 Stellen nach links
 - dies entspricht einer Multiplikation mit 8


```
int zahl = -5;
int produkt = zahl<<3;
```

Der Bedingungsoperator ? :

- der **Bedingungsoperator** (*conditional operator*) kann verwendet werden, um bestimmte **if-else**-Anweisungen kürzer zu formulieren
- Beispiel:


```
if ( userSetIt )
    value = usersValue;
else
    value = defaulttValue;
ist äquivalent zu
value = userSetIt ? usersValue : defaulttValue ;
```
- Bemerkung
 - der **? : -Operator** ist der einzige **ternäre Operator** (d.h. mit drei Operanden) in JAVA

Zuweisungsoperatoren

- das einfache = ist die Grundform des Zuweisungsoperators → a=b
- Semantik des Zuweisungsoperators (z.B. a=b)
 - der Zuweisungsoperator wird als **binärer Operator** betrachtet, d.h. die Zuweisung stellt einen **Ausdruck** dar
 - der **Rückgabewert** der Zuweisungsoperation ist der Wert des **rechten Operandens** (hier b)
 - zusätzlich wird – als **Seiteneffekt** – dem **linken Operanden** (hier: a) der Wert des rechten Operanden zugewiesen → sonst wäre es ja keine Zuweisung im klassischen Sinne
- ⚠ da in JAVA die Zuweisung als Ausdruck betrachtet wird, sind z.B. Anweisungen der folgenden Form möglich:
 - d = 2 + b + c + (b=c)
- von der Verwendung derartiger Anweisungen wird **dringend abgeraten**

- der Zuweisungsoperator kann mit jedem **arithmetischen** Operator oder **binären Bitoperator** verbunden werden → **kombinierter Zuweisungsoperator**

x += y; entspricht x = x + y;
 a <<= 3; entspricht a = a << 3;

- ⚠ die Verwendung eines kombinierten Zuweisungsoperators führt zu einem **impliziten Setzen von Klammern**

a *= b+1; entspricht a = a*(b+1); und nicht a = a*b+1;

Typumwandlung bei einfachen Typen: der cast-Operator

Der Typ eines Ausdrucks

- Tab. OPE-7 zeigt die wichtigsten Regeln für die Festlegung des Typs eines Ausdrucks

Typ des Ausdrucks	Operand 1	Operator	Operand 2
int	int, short, byte	arithmetisch, Bitoperator	int, short, byte
long	long	arithmetisch, Bitoperator	long, int, short, byte
float	float	arithmetisch	float, long, int, short, byte
double	double	arithmetisch	double, float, long, int, short, byte
String	String	+	String, einfacher Typ

Tab. OPE-7 Festlegung des Typs eines Ausdrucks

- wird ein **char**-Wert in einem Ausdruck verwendet, so wird dieser in einen **int**-Wert umgewandelt
- die oberen 16 Bits werden auf Null gesetzt

Implizite Typumwandlungen

- **implizite Typumwandlungen** (*Typkonvertierung*) finden automatisch statt
- Tab. OPE-7 (S. 24) enthält die wichtigsten Fälle impliziter Typumwandlungen
- Bemerkung
 - eine implizite Typumwandlung kann zu einem **Genauigkeitsverlust** führen
 - Beispiel: Umwandlung von **long** nach **float**
- Umwandlung eines **einfachen Typs** in einen **String**
 - bei der Verwendung des String-Verkettungsoperators + wird der Wert eines **einfachen Typs** automatisch in einen **String** umgewandelt

Explizite Typumwandlungen

- eine **explizite** Typumwandlung wird auch als **Casting** bezeichnet
- syntaktische Struktur einer Typumwandlung: (Typname) Ausdruck
 - (Typname) wird auch als **cast-Operator** bezeichnet
- Beispiel:


```
double a = 7.99;
long   b = (long) a; // ⇒ b=7
```
- Bemerkungen
 - bestimmte Typumwandlungen sind nicht erlaubt, z.B. **boolean** nach **int**
 - eine Typumwandlung liefert einen (neuen) Wert eines neuen Typs, der die beste verfügbare Darstellung des (alten) Wertes im alten Typ ist
 - ◊ bei der Umwandlung einer **Gleitkommazahl** in eine **Integer-Zahl** werden die **Nachkommastellen abgeschnitten**
 - ◊ bei der Umwandlung von **double** nach **float** kann im Extremfall der Wert ∞ entstehen
 - ◊ ein Integer-Typ wird in einen kleineren Integer-Typ durch das Abschneiden der **oberen Bits** umgewandelt
 - ▷ dies kann zu einer **Vorzeichenänderung** führen
 - ▷ Beispiel: **short** s = -134; b = (**byte**) s; liefert b=122
 - ein **char**-Ausdruck kann in jeden **Integer**-Typ umgewandelt werden und umgekehrt

Auswertungsreihenfolge, Priorität und Assoziativität von Operatoren

Auswertungsreihenfolge

- JAVA garantiert eine Auswertungsreihenfolge der **Operanden** eines **Operators** von *links* nach *rechts*
 - Beispiel: $x+y$
 - Auswertung von x
 - Auswertung von y
 - Addition
 - die Auswertungsreihenfolge ist dann wichtig, wenn x oder y **Seiteneffekte** besitzen (wenn also x oder y beispielsweise Methodenaufrufe darstellen)
 - mit Ausnahme der Operatoren $\&\&$, $||$ und $?:$ wird **jeder** Operand eines Operators **vor** der Ausführung des Operators ausgewertet

Prioritäten von Operatoren

- Operatorprioritäten spielen eine Rolle, wenn in einem Ausdruck **mehrere Operatoren** auftreten
 - die Prioritäten entscheiden, in welcher **Reihenfolge** die einzelnen **Operationen** durchgeführt werden
 - bei Operatoren gleicher Priorität entscheidet deren **Assoziativität** über die Reihenfolge der Operationsausführung
- Beispiel: $x + y * z \rightarrow$ die Priorität der Multiplikation ist höher als die der Addition
 - Auswertung von x
 - Auswertung des zweiten Operanden der Additionsoperation \rightarrow dieser Operand ist selbst wieder ein Ausdruck: $y * z$
 - ◇ Auswertung von y
 - ◇ Auswertung von z
 - ◇ Ausführung der Multiplikation
 - Ausführung der Addition
- durch die Verwendung von **Klammern** kann die Ausführungsreihenfolge der Operationen verändert werden
 - Beispiel: $(x+y) * z$
 - Auswertung des linken Operanden der Multiplikation: $x+y$
 - ◇ Auswertung von x
 - ◇ Auswertung von y
 - ◇ Ausführung der Addition
 - Auswertung von z
 - Ausführung der Multiplikation

- Tab. OPE-8 definiert die Prioritäten der Operatoren
 - die höchste Priorität ist 1, die niedrigste 14
 - die Operatorprioritäten sind [GWR00, S. 203] entnommen

1	Array-Index Methodenaufruf Attributzugriff	[] () .
2	Prä- oder Postinkrement Prä- oder Postdekrement Vorzeichen (unär) bitweise Negation logische Negation 	++ -- - + ~ ! (type) new
3	Multiplikation, Division, Rest	* / %
4	Addition, String-Verkettung, Subtraktion	+ -
5	Verschiebeoperatoren	<< >> >>>
6	Vergleichsoperatoren Typüberprüfung eines Objekts	< > >= <= instanceof
7	Gleichheit, Ungleichheit	== !=
8	Bitoperator UND	&
9	Bitoperator Exklusives ODER	^
10	Bitoperator (Inklusives) ODER	
11	logisches UND	&&
12	logisches ODER	
13	bedingte Auswertung	? :
14	Wertzuweisung kombinierte Zuweisungsoperatoren	= = += -= *= /= %= >>= <<= >>>= &= ^= =

Tab. OPE-8

Prioritäten der Operatoren

Assoziativität von Operatoren

- alle **binären** Operatoren **außer** den Zuweisungsoperatoren sind **links-assoziativ**
 - bei gleicher Priorität werden somit die Operationen von links nach rechts ausgeführt
 - a op b op c ist äquivalent zu (a op b) op c
- ⚠ (3.0/4.0)/5.0=0.15 und 3.0/(4.0/5.0)=3.75
- die **Zuweisungsoperatoren** sind **rechts-assoziativ**
 - a=b=c ist äquivalent zu a=(b=c)
- ⚠ ein derartiger Ausdruck sollte **nicht benutzt** werden

Literatur

- [AGH00] ARNOLD, KEN, JAMES GOSLING und DAVID HOLMES: *The Java Programming Language, Third Edition*. Addison Wesley Publishing Company, 3. Auflage, 2000. ISBN 0-201-70433-1.
- [Bal99a] BALZERT, HEIDE: *Lehrbuch der Objektmodellierung: Analyse und Entwurf*. Spektrum Akademischer Verlag, 1999. ISBN 3-8274-0285-9.
- [Bal99b] BALZERT, HELMUT: *Lehrbuch Grundlagen der Informatik*. Spektrum Akademischer Verlag, 1999. ISBN 3-8274-0358-8.
- [Eng93] ENGESSER, HERMANN (Herausgeber): *DUDEN Informatik: Ein Sachlexikon für Studium und Praxis*. Dudenverlag, 1993. ISBN 3-411-05232-5.
- [GJSB00] GOSLING, JAMES, BILL JOY, GUY STEELE und GILAD BRACHE: *The Java Language Specification, Second Edition*. Addison Wesley Publishing Company, 2. Auflage, 2000. ISBN 0-201-31008-2.
- [GWR00] GOLL, JOACHIM, CORNELIA WEISS und PETER ROTHLÄNDER: *Java als erste Programmiersprache: Java 2 Plattform*. B. G. Teubner Stuttgart, 2000. ISBN 3-519-12642-7.
- [Ste88] STETTER, FRANZ: *Grundbegriffe der Theoretischen Informatik*. Springer-Verlag Berlin, Heidelberg, 1988.