Operatoren und Ausdrücke

- Priorität und Assoziativität von Operatoren
- Auswertungsreihenfolge der Operanden
- Implizite Typkonvertierung
- Arithmetische Operatoren
- Bitoperatoren
- Vergleichsoperatoren
- Der Bedingungsoperator `?:`
- Explizite Typkonvertierung
- Zuweisungsoperatoren
- Konstantenausdrücke

Operatoren und Ausdrücke

Ein **Ausdruck** besteht aus

- Operanden,
- Operatoren und
- Klammern,

die zusammen eine Berechnungsvorschrift beschreiben.

Operanden können

- Variablen,
- Konstanten sowie
- Methodenaufrufe

sein.

Bei den **Operatoren** gibt es verschiedene Stelligkeiten:

- **Unäre** Operatoren wirken auf einen Operanden.
- **Binäre** Operatoren stehen zwischen ihren beiden Operanden.
- **Ternäre** Operatoren haben drei Operanden.

Operatoren und Ausdrücke

Priorität und Assoziativität von Operatoren

Die **Priorität** (oder der Vorrang) von Operatoren bestimmt ihre Auswertungsreihenfolge.

Beispiel: *„Punktrechnung vor Strichrechnung“*

- Der Ausdruck `5 * 3 - 3` ergibt den Wert `12` und nicht `0`. Der Operatorvorrang kann durch Klammerung geändert werden: `5 * (3 - 3)` ergibt `0`.

Erscheinen zwei Operatoren gleicher Priorität nebeneinander, entscheidet die **Assoziativität** (die Auswertungsrichtung) des Operators, der zuerst ausgeführt wird.

Beispiel:

- Weil Addition und Subtraktion linksassoziativ sind, ist der Ausdruck `9 - 5 + 2` äquivalent zu `(9 - 5) + 2` und ergibt den Wert 6 und nicht 2.

Operatoren und Ausdrücke

Priorität und Assoziativität von Operatoren

In der folgenden Tabelle sind alle verfügbaren Operatoren in absteigender Priorität aufgeführt.

Die Tabelle enthält

- der Priorität (*Prio.*),
- die Stelligkeit (*Stl.*) und
- die Auswertungsrichtung (*Rtg.*)

der Operatoren.

Bei der Auswertungsrichtung bedeutet

⇐ „*von rechts nach links*“ (rechtsassoziativ) und
⇒ „*von links nach rechts*“ (linksassoziativ).

Operatoren und Ausdrücke

Priorität und Assoziativität von Operatoren

Prio.	Operation	Stl.	Operator	Rtg.

15	Zugriff auf Objektkomponente
	Zugriff auf Feldelement
	Methodenaufruf
	Post-Inkrement
	Post-Dekrement
		unär	`.`
			`[]`
			`( args )`
			`++`
			`--`
				⇒

14	Prä-Inkrement
	Prä-Dekrement
	positives Vorzeichen
	negatives Vorzeichen
	Bitkomplement
	logische Negation
		unär	`++`
			`--`
			`+`
			`-`
			`~`
			`!`
				⇐

13	Objekterzeugung Typumwandlung (cast)	unär	`new` `( typ )`	⇐

12	Multiplikation
	Division
	Rest
		binär	`*`
			`/`
			`%`
				⇒

11	Addition
	Subtraktion
	String-Konkatenation
		binär	`+`
			`-`
			`+`
				⇒

10	Linksschieben
	Rechtsschieben mit Vorzeichen
	Rechtsschieben mit Nullen
		binär	`<<`
			`>>`
			`>>>`
				⇒

9	Vergleiche Typvergleich	binär	`<, >, <=, >=` `instanceof`	⇒
8	Gleichheit Ungleichheit	binär	`==` `!=`	⇒

Operatoren und Ausdrücke

Priorität und Assoziativität von Operatoren

Prio.	Operation	Stl.	Operator	Rtg.
7	bitweises UND logisches UND	binär	**&**	⇒
6	bitweises exklusives ODER logisches exklusives ODER	binär	**^**	⇒
5	bitweises ODER logisches ODER	binär	**|**	⇒
4	bedingtes UND	binär	**&&**	⇒
3	bedingtes ODER	binär	**||**	⇒
2	bedingter Ausdruck	ternär	**? :**	⇐
1	Zuweisungen **op** ∈ { ***, /, %, +, -,** **&, ^, |, <<, >>, >>>** }	binär	**=** **op=**	⇐

Operatoren und Ausdrücke

Auswertungsreihenfolge der Operanden

Die Operanden der Operatoren werden von links nach rechts ausgewertet.

Im Beispiel `x + y + z` wertet der Compiler erst x und y aus, addiert sie, wertet z aus und addiert dieses zum vorherigen Ergebnis.

Die Auswertungsreihenfolge spielt eine Rolle, wenn x, y und z in irgendeiner Form **Seiteneffekte** erzeugen.

Wenn sie z. B. Methoden verwenden, welche den Zustand eines Objekts ändern oder etwas ausdrucken, würde es sofort auffallen, wenn sie in einer anderen Reihenfolge ausgewertet würden.

Mit Ausnahme der Operatoren `&&`, `||`, und `?:` wird jeder Operand eines Ausdrucks vor der Ausführung des Operators ausgewertet.

Operatoren und Ausdrücke

Implizite Typkonvertierung

Alle Ausdrücke besitzen einen **Typ**.

Der Typ eines Ausdrucks wird von seinen Bestandteilen und der Semantik seiner Operatoren bestimmt.

Die meisten binären Operatoren verlangen als linken und rechten Operanden Werte desselben Typs.

Der Java-Compiler führt bei Operanden unterschiedlicher Datentypen intern eine Typkonvertierung durch, bei der die Operanden auf den Typ des „kompliziertesten“ Operanden erweitert werden.

Operatoren und Ausdrücke

Implizite Typkonvertierung

Wird ein arithmetischer oder ein Bitoperator auf ganzzahlige Werte angewandt, so ist das Ergebnis vom Typ `int`; es sei denn einer oder beide Operanden sind vom Typ `long`, dann ist auch das Ergebnis vom Typ `long`.

Alle Operationen auf ganzzahligen Werten werden entweder auf `int` oder auf `long` ausgeführt, so dass die kleineren Integer-Typen `byte` und `short` (und auch `char`) vor der Berechnung immer mindestens nach `int` umgewandelt werden.

Ist einer der Operanden ein reeller Wert, so wird die Operation in Gleitkommaarithmetik durchgeführt.
Solche Operationen werden in `float` durchgeführt, es sei denn, einer der Operanden ist vom Typ `double`, dann wird für die Berechnung sowie für das Ergebnis `double` verwendet.

Operatoren und Ausdrücke

Im Folgenden wird für die einzelnen Operatoren angegeben, von welchen Datentypen die Operanden sein dürfen und welchen Datentyp dann das Ergebnis hat.

- Mit **ganz** werden die Datentypen `long`, `int`, `short`, `byte` und `char` bezeichnet.

  Der Ergebnistyp ist `long`, wenn mindestens ein Operand vom Typ `long` ist; sonst ist der Ergebnistyp immer `int`.

- Mit **reell** werden die Datentypen `float` und `double` bezeichnet.

  Der Ergebnistyp ist `double`, wenn mindestens ein Operand vom Typ `double` ist; sonst ist der Ergebnistyp immer `float`.

Operatoren und Ausdrücke

Arithmetische Operatoren

Prio	Operator	Operanden	Ergebnis	Bemerkung
14	Vorzei-chen **+**	ganz reell	**int**, **long** reell	positives Vorzeichen
11	**+**	ganz, ganz reell, reell String, String	**int**, **long** reell String	Summe Summe Konkatenation
14	Vorzei-chen **-**	ganz reell	**int**, **long** reell	negatives Vorzeichen
11	**-**	ganz, ganz reell, reell	**int**, **long** reell	Differenz Differenz
12	*****	ganz, ganz reell, reell	**int**, **long** reell	Produkt Produkt
12	**/**	ganz, ganz reell, reell	**int**, **long** reell	ganzzahliger Quotient Quotient
12	**%**	ganz, ganz reell, reell	**int**, **long** reell	ganzzahliger Rest reeller Rest

Bei der Division `/` und der Modulo-Operation `%` darf der zweite Operand nicht Null sein.

Falls dies doch der Fall ist, wird bei ganzzahligen Operanden der Programmlauf mit einer entsprechenden Ausnahme abgebrochen; bei reellen Operanden ist das Ergebnis unendlich (`Infinity`).

Operatoren und Ausdrücke

Arithmetische Operatoren

Beispiel:

```
public class ArithmetischeOperatoren
{
  public static void main(String[] argv)
  {
    float quotient, z=12.34E5f, q=-3e1f, rest, wert;
    int zaehler=13, ganz;

    System.out.println(zaehler + " / 4 = " +
      zaehler/4 + "  Rest = " + zaehler%4);

    quotient = zaehler / 4;
    System.out.println("Reeller Quotient = " +
      quotient + " ist ganzzahlig");

    quotient = zaehler / 4.0F;
    System.out.println(
      "Jetzt ist das Ergebnis wirklich reell : " +
      quotient);

    rest = z % q;
    System.out.println(
      "Rest fuer reelles modulo: " + rest);
  }
}
```

Ausgabe:

```
13 / 4 = 3  Rest = 1
Reeller Quotient = 3.0 ist ganzzahlig
Jetzt ist das Ergebnis wirklich reell : 3.25
Rest fuer reelles modulo: 10.0
```

Operatoren und Ausdrücke

Arithmetische Operatoren

Beispiel:

```
public class Zahlenueberlauf
{
  public static void main(String[] argv)
  {
    float zaehler = 2.5E27f, nenner = 3.4e-12f;
    int oben = 12, unten = 0;

    System.out.println("Zahlenueberlauf bei float : "
      + zaehler/nenner);

    System.out.println("Division durch Null    : "
      + oben/unten);
  }
}
```

Ausgabe:

```
Zahlenueberlauf bei float : Infinity
java.lang.ArithmeticException: / by zero
      at Zahlenueberlauf.main(Zahlenueberlauf.java:32)
```

Operatoren und Ausdrücke

Arithmetische Operatoren

ganzzahlige Division und Modulo-Operation

Beispiele:

- `int m = 33, n = 7;`
  `n/m` ergibt `0`, `n%m` ergibt `7`

- Ganzzahlige Division kann zur Umrechnung von Maßeinheiten verwendet werden.

  - Länge in Fuß und Zoll
    76 Zoll sind 6 Fuß und 4 Zoll

    ```
    76/12 == 6    76%12 == 4
    ```

  - Zeit in Stunden und Minuten
    140 Minuten sind 2 Stunden und 20 Minuten.

    ```
    140/60 == 2    140%60 == 20
    ```

- Wenn eine ganze Zahl `m` durch 2 geteilt wird, ist der Rest entweder `0` oder `1`. Das Ergebnis ist `0`, wenn `m` gerade ist, und `1`, wenn `m` ungerade ist.

- Eine ganze Zahl `m` ist durch `n` teilbar, wenn `m%n` den Wert `0` ergibt.

Operatoren und Ausdrücke

Bitoperatoren

Schiebeoperatoren

Prio	Operator	Operanden	Ergebnis	Bemerkung
10	**<<**	ganz, ganz	**`int`**, **`long`**	Linksschieben
10	**>>**	ganz, ganz	**`int`**, **`long`**	Rechtsschieben mit Vorzeichen
10	**>>>**	ganz, ganz	**`int`**, **`long`**	Rechtsschieben mit Nullen

Das Bitmuster des ersten Operanden wird um so viele Stellen nach links bzw. rechts geschoben, wie im zweiten Operanden angegeben ist.

Beim Linksschieben werden von rechts Nullen nachgezogen.

Beim Rechtsschiebe-Operator >> wird von links das Vorzeichenbit nachgezogen. Das heißt:

- Nullen, falls der erste Operand positiv ist,
- Einsen, falls der erste Operand negativ ist.

Beim Rechtsschiebeoperator >>> werden von links immer Nullen nachgezogen.

Operatoren und Ausdrücke

Schiebeoperatoren

Beispiel:

```
public class SchiebeOperatoren
{
  public static void main(String[] argv)
  {
    int a, b;

    // Rechts-Schieben um n Stellen
    // entspricht Division durch 2 hoch n
    a = -12;
    b = a >> 2;
    System.out.println(a + " >> 2 = " +  b
      + "   also: -12/(2 hoch 2) = -12/4 = -3");

    a = -12;
    b = a >>> 2;  // vorzeichenlos behandelt
    System.out.println(a + " >>> 2 = " + b
      + "  vorzeichenlos behandelt\n"
      + "\tNullen kommen von links,\n"
      + "\tdaher positiv und so gross\n"
      + "\t(32 Bit)");

    // Links-Schieben um n Stellen
    // entspricht Multiplikation mit 2 hoch n
    a = 12;
    b = a << 4;   // 12*16
    System.out.println(a + " << 4 = " + b
      + "  12*(2 hoch 4) = 192");
  }
}
```

Operatoren und Ausdrücke

Schiebeoperatoren

Ausgabe:

```
-12 >> 2 = -3   also: -12/(2 hoch 2) = -12/4 = -3
-12 >>> 2 = 1073741821  vorzeichenlos behandelt
        Nullen kommen von links,
        daher positiv und so gross
        (32 Bit)
12 << 4 = 192  12*(2 hoch 4) = 192
```

Operatoren und Ausdrücke

Bitoperatoren

logische und bitweise Operatoren

Für einzelne Bits sind die logischen Operatoren wie folgt definiert:

A	B	NOT A	A AND B	A OR B	A XOR B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Interpretiert man `1` als `true` und `0` als `false`, so hat man die logischen Verknüpfungen für die Werte vom Typ `boolean`.

Erweitert man die logischen Operatoren auf Folgen von Bits, so erhält man die bitweisen Operatoren.

Operatoren und Ausdrücke

logische und bitweise Operatoren

Prio	Operator	Operanden	Ergebnis	Bemerkung
14	**~**	ganz	**int**, **long**	Bitkomplement
14	**!**	**boolean**	**boolean**	logisches NOT
7	**&**	ganz, ganz **boolean**, **boolean**	**int**, **long** **boolean**	bitweises AND logisches AND
6	**^**	ganz, ganz **boolean**, **boolean**	**int**, **long** **boolean**	bitweises XOR logisches XOR
5	**|**	ganz, ganz **boolean**, **boolean**	**int**, **long** **boolean**	bitweises OR logisches OR
4	**&&**	**boolean**, **boolean**	**boolean**	logisches AND
3	**||**	**boolean**, **boolean**	**boolean**	logisches OR

Beim booleschen & werden immer beide Operanden ausgewertet. Dagegen wird bei && der zweite Operand nicht mehr ausgewertet, wenn der erste `false` ergibt - dann ist das Ergebnis nämlich unabhängig vom zweiten Operanden sicher `false`.

Beim booleschen | werden immer beide Operanden ausgewertet. Dagegen wird bei || der zweite Operand nicht mehr ausgewertet, wenn der erste `true` ergibt - dann ist das Ergebnis nämlich unabhängig vom zweiten Operanden sicher `true`.

Operatoren und Ausdrücke

Einschub: Das Modul `utilities`

Das Modul `utilities` enthält Klassen, die nützliche Methoden bereitstellen, wie z.B. zum Einlesen von Daten von der Tastatur oder zur formatierten Ausgabe auf den Monitor.

Die Klasse `FormatierteAusgabe` enthält u.a. die folgende Methode:

```
public static void binaerAusgabe(
  long zahl, int breite);
  // gibt zahl als Binärmuster aus;
  // nach je vier Dualstellen folgt ein
  // Leerzeichen;
  // breite gibt an, wie viele Dualstellen
  // gezeigt werden
```

Ein Programm, das die obige Methode benutzt, enthält als erste Zeile

```
import utilities.FormatierteAusgabe;
```

Operatoren und Ausdrücke

Einschub: Das Modul `utilities`

Die Klasse `TastaturEingabe` enthält u.a. die folgenden Methoden:

```
public static String readString(String prompt);
  // liefert die eingelesene Zeichenkette
public static char readChar(String prompt);
  // liefert das erste Zeichen der gelesenen
  // Zeichenkette; der Rest wird ignoriert
public static int readInt(String prompt);
  // liefert eingelesenen int-Wert
public static long readLong(String prompt);
  // liefert eingelesenen long-Wert
public static double readDouble(String prompt);
  // liefert eingelesenen double-Wert
public static void warte();
  // wartet auf <Return>
```

Die Zeichenkette `prompt` wird als Eingabeaufforderung auf den Monitor ausgegeben.

Ein Programm, das die obigen Methoden benutzt, enthält als erste Zeile

```
import utilities.TastaturEingabe;
```

Operatoren und Ausdrücke

logische und bitweise Operatoren

Beispiel:

```
import utilities.FormatierteAusgabe;

public class LogBitOperatoren1
{
  public static void main(String[] argv)
  {
    int zahl = 0x59B7, maske = 0X03F8;
    System.out.println(
      "Teil von \"zahl\" wird auf 1 gesetzt");
    FormatierteAusgabe.binaerAusgabe(zahl, 16);
    System.out.println("\tzahl");
    FormatierteAusgabe.binaerAusgabe(maske, 16);
    System.out.println("\tmaske");
    FormatierteAusgabe.binaerAusgabe(
      zahl | maske, 16);
    System.out.println("\t<<== zahl | maske");
    System.out.println(".... ..-- ---- -...");
  }
}
```

Ausgabe:

```
Teil von "zahl" wird auf 1 gesetzt
0101 1001 1011 0111     zahl
0000 0011 1111 1000     maske
0101 1011 1111 1111     <<== zahl | maske
.... ..-- ---- -...
```

Operatoren und Ausdrücke

logische und bitweise Operatoren

Beispiel:

```
import utilities.FormatierteAusgabe;

public class LogBitOperatoren2
{
  public static void main(String[] argv)
  {
    int zahl = 0x59B7, maske = 0X03F8;
    System.out.println(
      "Teil von \"zahl\" wird auf 0 gesetzt");
    FormatierteAusgabe.binaerAusgabe(zahl, 16);
    System.out.println("\tzahl");
    FormatierteAusgabe.binaerAusgabe(maske, 16);
    System.out.println("\tmaske");
    FormatierteAusgabe.binaerAusgabe(
      zahl & ~maske, 16);
    System.out.println("\t<<== zahl & ~maske");
    System.out.println(".... ..-- ---- -...");
  }
}
```

Ausgabe:

```
Teil von "zahl" wird auf 0 gesetzt
0101 1001 1011 0111     zahl
0000 0011 1111 1000     maske
0101 1000 0000 0111     <<== zahl & ~maske
.... ..-- ---- -...
```

Operatoren und Ausdrücke

logische und bitweise Operatoren

Beispiel:

```
import utilities.FormatierteAusgabe;

public class LogBitOperatoren3
{
  public static void main(String[] argv)
  {
    int zahl = 0x59B7, maske = 0X03F8;
    System.out.println(
      "Teil von \"zahl\" wird herausgeschnitten");
    FormatierteAusgabe.binaerAusgabe(zahl, 16);
    System.out.println("\tzahl");
    FormatierteAusgabe.binaerAusgabe(maske, 16);
    System.out.println("\tmaske");
    FormatierteAusgabe.binaerAusgabe(
      zahl & maske, 16);
    System.out.println("\t<<== zahl & maske");
    System.out.println(".... ..-- ---- -...");
    FormatierteAusgabe.binaerAusgabe(
      (zahl & maske) >> 3, 16);
    System.out.println(
      "\t<<== zahl & maske um 3 Stellen " +
      "\n\t\t\tnach rechts verschoben");
  }
}
```

Ausgabe:

```
Teil von "zahl" wird herausgeschnitten
0101 1001 1011 0111     zahl
0000 0011 1111 1000     maske
0000 0001 1011 0000     <<== zahl & maske
.... ..-- ---- -...
0000 0000 0011 0110     <<== zahl & maske um 3 Stellen
                        nach rechts verschoben
```

Operatoren und Ausdrücke

logische und bitweise Operatoren

Beispiel:

```
import utilities.FormatierteAusgabe;

public class LogBitOperatoren4
{
  public static void main(String[] argv)
  {
    int zahl = 0x59B7, maske = 0X03F8;
    System.out.println(
      "exklusives ODER  \"zahl ^ maske\" ");
    FormatierteAusgabe.binaerAusgabe(zahl, 16);
    System.out.println("\tzahl");
    FormatierteAusgabe.binaerAusgabe(maske, 16);
    System.out.println("\tmaske");
    FormatierteAusgabe.binaerAusgabe(
      zahl ^ maske, 16);
    System.out.println("\t<<== zahl ^ maske");
  }
}
```

Ausgabe:

```
exklusives ODER  "zahl ^ maske"
0101 1001 1011 0111     zahl
0000 0011 1111 1000     maske
0101 1010 0100 1111     <<== zahl ^ maske
.... .--- ---- -...
```

Operatoren und Ausdrücke

Vergleichsoperatoren

Prio	Operator	Operanden	Ergebnis	Bemerkung
9	**<**	ganz, ganz reell, reell	**boolean**	kleiner als
9	**<=**	ganz, ganz reell, reell	**boolean**	kleiner oder gleich
9	**>**	ganz, ganz reell, reell	**boolean**	größer als
9	**>=**	ganz, ganz reell, reell	**boolean**	größer oder gleich
8	**==**	ganz, ganz reell, reell **boolean**, **boolean**	**boolean**	gleich
8	**!=**	ganz, ganz reell, reell **boolean**, **boolean**	**boolean**	ungleich

Achtung:

Der Test auf Gleichheit wird mit zwei Gleichheitszeichen geschrieben (== statt =) !

Operatoren und Ausdrücke

Der Bedingungsoperator ? :

dient zur Formulierung eines Ausdrucks, der abhängig von einem booleschen Ausdruck einen von zwei Werten zurückliefert:

Bedingung ? Ausdruck$_1$: Ausdruck$_2$

Falls die *Bedingung* wahr ist, ergibt sich der Wert von *Ausdruck$_1$*, andernfalls der Wert von *Ausdruck$_2$*.

Beispiel:

```
public class BedingterAusdruck
{
  public static void main(String[] argv)
  {
    int a = -12;
    int betrag = a < 0 ? -a : a;
    System.out.println("Betrag von " + a +
      " ist " + betrag);
  }
}
```

Ausgabe:

```
Betrag von -12 ist 12
```

Operatoren und Ausdrücke

Der Bedingungsoperator `?:`

Der bedingte Ausdruck

```
min = x <= y ? x : y;
```

ist äquivalent zur `if`-Anweisung

```
if(x <= y)
  min = x;
else
  min = y;
```

Operatoren und Ausdrücke

Der Bedingungsoperator `?:`

In einem bedingten Ausdruck

***Bedingung* `?` *Ausdruck*$_1$ `:` *Ausdruck*$_2$**

müssen beide Ausdrücke zuweisungskompatible Typen haben. Der Typ des einen Ausdrucks muss dem Typ des anderen ohne explizite Typumwandlung zugewiesen werden können. Der Typ des Ergebnisses ist der allgemeinere der beiden Typen.

Im folgenden Beispiel:

```
double scale = halveIt ? 0.5 : 1;
```

sind die beiden Ausdrücke vom Typ `int` (`1`) bzw. `double` (`0.5`). Daher wird die `1` in `1.0` umgewandelt, und das Ergebnis des Bedingungsoperators ist vom Typ `double`.

Operatoren und Ausdrücke

Explizite Typkonvertierung

Wenn ein Typ einem anderen Typ nicht mit einer impliziten Typumwandlung zugewiesen werden kann, kann er oft explizit in einen anderen Typ umgewandelt werden (engl. *cast*).

Beispiel:

Die Deklaration

```
double d = 7.99;
long l = d;
```

führt zu einer Fehlermeldung des Compilers:

Incompatible type for declaration.
Explicit cast needed to convert double to long.

Abhilfe schafft die explizite Typkonvertierung:

```
double d = 7.99;
long l = (long) d;
```

Operatoren und Ausdrücke

Explizite Typkonvertierung

größer ganzzahlig => kleiner ganzzahlig

- Es werden lediglich so viele Bits von rechts her übernommen, wie der kleinere Typ aufnehmen kann. Dabei kann sich neben dem Zahlenwert auch das Vorzeichen ändern.

`double` => `float`

- liefert entweder den Näherungswert oder - falls der `double`-Wert nicht als `float` darstellbar ist - einen der folgenden Werte:

    - die positive bzw. negative Null, wenn der Wert betragsmäßig zu klein und positiv bzw. negativ ist;

    - positiv bzw. negativ unendlich, wenn der Wert betragsmäßig zu groß und positiv bzw. negativ ist;

    - `NaN` (Not a Number) vom Typ `double` wird in `NaN` vom Typ `float` konvertiert.

Operatoren und Ausdrücke

Explizite Typkonvertierung

reell => ganzzahlig

- `NaN` wird in den ganzzahligen Wert 0 konvertiert.

- Ist der Zieltyp kleiner als `int`, wird als temporärer Zieltyp `int` gewählt.

- Der reelle Wert wird in Richtung 0 gerundet. Ist dieser Wert im (temporären) Zieltyp darstellbar, wird dieser Wert genommen. Ist er zu groß bzw. zu klein, so wird der größte bzw. kleinste darstellbare Wert des (temporären) Zieltyps gewählt.

- War der ursprüngliche Zieltyp kleiner als `int`, so wird der temporäre Zieltyp auf den ursprünglichen Zieltyp so gewandelt, dass wieder nur so viele Stellen von rechts her übernommen werden, wie der kleinere Typ aufnehmen kann (siehe größer ganzzahlig => kleiner ganzzahlig).

Operatoren und Ausdrücke

Explizite Typkonvertierung

Beispiel:

```
import utilities.FormatierteAusgabe;

public class Casting
{
  public static void main(String[] argv)
  {
    double langeZahl = 1.3e8;
    System.out.print("64-Bit binaer : ");
    FormatierteAusgabe.binaerAusgabe(
      (long)langeZahl, 64);
    System.out.println();
    System.out.println("double : " + langeZahl);
    System.out.println("float  : " + (float)langeZahl);
    System.out.println("long   : " + (long)langeZahl);
    System.out.println("int    : " + (int)langeZahl);
    System.out.println("short  : " + (short)langeZahl);
    System.out.println("byte   : " + (byte)langeZahl);
    System.out.println("char   : " + (char)langeZahl);
  }
}
```

Ausgabe:

```
64-Bit binaer : 0000 0000 0000 0000 0000 0000 0000 0000
                0000 0111 1011 1111 1010 0100 1000 0000
double : 1.3E8
float  : 1.3E8
long   : 130000000
int    : 130000000
short  : -23424
byte   : -128
char   : ?
```

Operatoren und Ausdrücke

Explizite Typkonvertierung

Beispiel:

```
public class ReellerQuotient
{
  public static void main(String[] argv)
  {
    float quotient;
    int zaehler = 14, nenner = 4;
    quotient = zaehler / nenner;
    System.out.println(
      "ganzzahliges Ergebnis : " + quotient);
    quotient = (float)zaehler / nenner;
    System.out.println(
      "reelles Ergebnis      : " + quotient);
    quotient = (float)(zaehler / nenner);
    System.out.println(
      "ganzzahliges Ergebnis : " + quotient);
  }
}
```

Ausgabe:

```
ganzzahliges Ergebnis : 3.0
reelles Ergebnis      : 3.5
ganzzahliges Ergebnis : 3.0
```

Operatoren und Ausdrücke

Zuweisungsoperatoren

der einfache Zuweisungsoperator =

Prio.	Operator	Operanden	Ergebnis	Bemerkung
1	=	linke Seite, beliebig	Typ von linke Seite	Wertzuweisung

Wirkung von $E_1 = E_2$:

- Der rechte Operand E_2, der i. a. ein Ausdruck ist, wird ausgewertet.

- Der Datentyp von E_2 wird, falls nötig, in den Datentyp von E_1 "nach oben" erweitert.
  Der so erhaltene Wert wird E_1 zugewiesen.

  Ist der Datentyp von E_2 allgemeiner als der von E_1, muss der cast-Operator aus dem vorigen Abschnitt verwendet werden; andernfalls gibt der Compiler eine Fehlermeldung aus.

- Die Zuweisung liefert als Wert den neuen Wert von E_1.

Operatoren und Ausdrücke

Zuweisungsoperatoren

Beispiel:

```
public class Zuweisung
{
  public static void main(String[] argv)
  {
    int x, y = 5, z = 7, u = -15;
    System.out.print("x = " + (x = y + z));
    // Wert von x nach Zuweisung ausgeben
    System.out.println(" , nochmal x = " + x);
    // und noch einmal ausgeben
    y = (u = x + y) + z;
    // u wird die Summe x + y zugewiesen
    // danach wird y der Wert u + z zugewiesen
    System.out.println("u = " + u + ", y = " + y);
    x = y = z = 15;
    // Mehrfachzuweisung: x, y, z erhalten den Wert 15
    System.out.println("x = " + x);
  }
}
```

Ausgabe:

```
x = 12 , nochmal x = 12
u = 17, y = 24
x = 15
```

Operatoren und Ausdrücke

Zuweisungsoperatoren

kombinierte Zuweisungsoperatoren

Jeder Operator

op ∈ { ***, /, %, +, -, &, ^, |, <<, >>, >>>*** }

kann mit = verknüpft werden. Auf diese Weise erhält man die sogenannten kombinierten Zuweisungs-operatoren.

Prio	Operator	Operanden	Ausdruck	entspricht
1	**op=**	Opd, Wert	Opd **op=** Wert	Opd **=** Opd **op** Wert

Beispiel:

```
feld[wo()] += 12;
```

ist dasselbe, wie

```
feld[wo()] = feld[wo()] + 12;
```

außer, dass der Ausdruck auf der linken Seite der Zuweisung auch nur einmal ausgewertet wird.

Operatoren und Ausdrücke

Zuweisungsoperatoren

kombinierte Zuweisungsoperatoren

Sind `var` eine Variable vom Typ `typ`, `expr` ein Ausdruck und *op* ein binärer Operator, so ist

```
var op= expr
```

äquivalent zu

```
var = (typ)((var) op (expr))
```

außer, dass `var` nur einmal ausgewertet wird.

Das bedeutet, dass *op*= nur gültig ist, wenn *op* auch für die beteiligten Typen gültig ist.

Beachten Sie die obige Klammerung. Der Ausdruck

```
a *= b + 1
```

ist gleichwertig mit

```
a = a * (b + 1)
```

aber nicht mit

```
a = a * b + 1
```

Operatoren und Ausdrücke

Zuweisungsoperatoren

Inkrement- und Dekrement-Operatoren

Häufig muss man den Wert einer Variablen um 1 erhöhen (inkrementieren) oder erniedrigen (dekrementieren). Hierzu gibt es den Inkrement-Operator ++ bzw. den Dekrement-Operator `--`.
Beide Operatoren sind unär und können entweder *vor* dem Operanden (Präfixform) oder *hinter* dem Operanden (Postfixform) stehen.

Prio	Operator	Operanden	Ergebnis	Bemerkung
15	**++**	linke Seite	Typ von linker Seite	Post-Inkrement
15	**--**	linke Seite	Typ von linker Seite	Post-Dekrement
14	**++**	linke Seite	Typ von linker Seite	Prä-Inkrement
14	**--**	linke Seite	Typ von linker Seite	Prä-Dekrement

- Der Wert des Operanden wird bei ++ um 1 erhöht, bei `--` um 1 erniedrigt.

- Der Wert des Ausdrucks ist wie folgt festgelegt:

  - in der *Postfix*-Form der ursprüngliche Wert des Operanden,

  - in der *Präfix-F*orm der um 1 erhöhte bzw. erniedrigte Wert des Operanden.

Operatoren und Ausdrücke

Zuweisungsoperatoren

Inkrement- und Dekrement-Operatoren

Beispiel:

```
public class Inkrement_Dekrement {
  public static void main(String[] argv) {
    int a = 12;
    System.out.print(a++ + " , ");
    System.out.print(++a + " , ");
    System.out.print(--a + " , ");
    System.out.print(a-- + " , ");
    System.out.println(--a);
  }
}
```

Ausgabe:

```
12 , 14 , 13 , 13 , 11
```

12 alter Wert von a, danach wird a erhöht
14 Wert 13 wird erst erhöht, dann ausgegeben
13 Wert 14 wird erst erniedrigt, dann ausgegeben
13 alter Wert von a, danach wird a erniedrigt
11 Wert 12 wird erst erniedrigt, dann ausgegeben

Operatoren und Ausdrücke

Konstantenausdrücke

Java erwartet an manchen Stellen sogenannte Konstantenausdrücke. Das sind Ausdrücke, deren Operanden konstante Werte oder benannte Konstanten sind. Konstantenausdrücke werden bereits bei der Übersetzung berechnet.

Beispiel:

```
public class KonstanterAusdruck
{
  public static void main(String[] argv)
  {
    final int a = 12, b = -14, c = 3;
    int i = 3 * (a * c + b);
    System.out.println(i);
  }
}
```

Der Compiler erzeugt folgende Deklaration:

```
int i = 66;
```