

Inhalt

| | | |
|-------|--|------|
| 3 | Grundelemente der Java-Programmierung..... | 3-2 |
| 3.1 | <i>Alphabet</i> | 3-2 |
| 3.2 | <i>Bezeichner (Identifizier)</i> | 3-3 |
| 3.3 | <i>Kommentare</i> | 3-3 |
| 3.4 | <i>Elementardatentypen</i> | 3-5 |
| 3.5 | <i>Konstanten (Literele)</i> | 3-7 |
| 3.5.1 | Unbenannte Konstanten | 3-7 |
| 3.5.2 | Benannte Konstanten..... | 3-8 |
| 3.6 | <i>Variablen</i> | 3-9 |
| 3.7 | <i>Ausdrücke</i> | 3-11 |
| 3.7.1 | Arithmetische Ausdrücke | 3-11 |
| 3.7.2 | Bitoperatoren | 3-11 |
| 3.7.3 | Wertzuweisungen | 3-12 |
| 3.7.4 | Inkrementieren und Dekrementieren..... | 3-13 |
| 3.7.5 | Logische Ausdrücke (Bedingungen) | 3-14 |
| 3.7.6 | Bedingte Ausdrücke | 3-14 |
| 3.8 | <i>Zusammenfassung</i> | 3-16 |

3 Grundelemente der Java-Programmierung

3.1 Alphabet

Unicode

Da der **8-Bit-Zeichensatz** des **ASCII-Code**¹ *nicht alle nationalen Besonderheiten* berücksichtigt ($2^8 = 256$ mögliche Zeichen), gibt es verschiedene lokale Versionen. Entsprechend der Betriebssystemkonfiguration wird die jeweils aktive Version festgelegt.

Um **Plattformunabhängigkeit** zu erreichen, bedient sich Java, im Gegensatz zu anderen Programmiersprachen, des **16-Bit-Zeichensatzes** des **Unicode**². Durch die Verschlüsselung der Zeichen in 2 Bytes ergeben sich insgesamt $2^{16} = 65\,536$ mögliche Zeichen. Alle wichtigen Sprachen und deren Besonderheiten können berücksichtigt werden. Derzeit sind nicht ganz 40 000 Zeichen belegt. Der ASCII-Codes ist eine *Teilmenge* des Unicode.

Intern arbeitet Java komplett mit Unicode. Dazu sind entsprechende Umwandlungsfunktionen implementiert, welche plattformabhängig in 8-Bit-Zeichen umwandelt und umgekehrt.

Steuerzeichen:

Sie dienen der **Steuerung von Ausgabegeräten** wie Bildschirm und Drucker. Folgende Escapesequenzen (analog C) sind definiert:

| | |
|-----------------|--|
| <code>\b</code> | Versetzen um eine Position nach links (backspace) |
| <code>\f</code> | Seitenvorschub (formfeed) |
| <code>\n</code> | Zeilenvorschub (linefeed, new line) |
| <code>\r</code> | Positionierung am Zeilenanfang (carriage return) |
| <code>\t</code> | Horizontaler Tabulator (horizontal tab) |

Die folgenden Anweisungen erzeugen dieselbe Konsolenausgabe:

```
System.out.println( "Hallo Welt!");  
System.out.print( "Hallo Welt!\n");
```

Entwerter:

Weitere Escapesequenzen erzeugen **druckbare Zeichen** und dienen der **Entwertung von Metazeichen**:

| | |
|-----------------|--|
| <code>\'</code> | Entwerten des Abschlusssymbol für Zeichenkonstante |
| <code>\"</code> | Entwerten des Abschlusssymbol für Zeichenkettenkonstante |
| <code>\\</code> | Entwerten des Backslashes als Escapesequenz |

Groß- und Kleinbuchstaben sind *signifikant*.

¹ ASCII - *American Standard Code for Information Interchange*

² Unicode-Konsortium <http://www.unicode.org/>

3.2 Bezeichner (Identifizier)

Für die Bezeichnung der *Objekte*, ihrer *Klassen*, *Attribute* und *Methoden* werden **Namen** benötigt:

Namen sind frei wählbare, beliebig lange Wörter aus *Buchstaben* beliebiger Sprachen, *Ziffern*, dem *Unterstrich* und dem *Dollarzeichen*, die nicht mit einer Ziffer beginnen dürfen und keinem der 48 *Schlüsselwörter* der Sprache (`int`, `if`, `public`, ...) entsprechen.

Bezeichner sind *ein* oder aber auch *mehrere Namen*, verbunden durch einen *Punkt* (`setLampe`, `System.out.println`).

Es gelten allgemeine *Regeln* für die Wahl von Namen:

- Grundsätzlich sollten Namen den Inhalt beschreiben (*sprechende Namen*).
- *Klassennamen* beginnen mit einem **Großbuchstaben**. Ist dieser aus mehreren Worten zusammengesetzt, so beginnt jedes neue Wort ebenfalls mit einem Großbuchstaben (**H**allo**W**elt**A**pplikation).
- *Variablen- und Methodennamen* beginnen mit einem **Kleinbuchstaben**. Ist dieser aus mehreren Worten zusammengesetzt, so beginnt jedes neue Wort mit einem Großbuchstaben (**r**ead**I**nt).
- *Konstantennamen* bestehen aus nur **Grossbuchstaben**. Ist dieser aus mehreren Worten zusammengesetzt, so beginnt jedes neue Wort mit einem Unterstrich (**MAX**, **MAX_ANZAHL**).

3.3 Kommentare

Kommentare dienen der Lesbarkeit und Verständlichkeit von Programmquelltexten. Sie werden vom Compiler ignoriert, haben somit keinen Einfluss auf Größe und Geschwindigkeit des ausführbaren Programms. Man unterscheidet Kommentare für **interne** und **externe Dokumentationen**. Java kennt drei verschiedene Arten von Kommentaren.

Interne Dokumentation

1. `// Kommentar`

Alle dem Kommentarzeichen `//` folgende Zeichen bis Zeilenende werden vom Compiler ignoriert.

2. `/* Kommentar */`

Alle Zeichen zwischen `/*` und `*/` werden als Kommentar behandelt. Dieser kann über mehrere Zeilen gehen, darf aber *nicht* geschachtelt auftreten und wird zum *Auskommentieren von Programmteilen* vor allem in der *Testphase* verwendet.

Externe Dokumentation

3. `/** doc-Kommentar */`

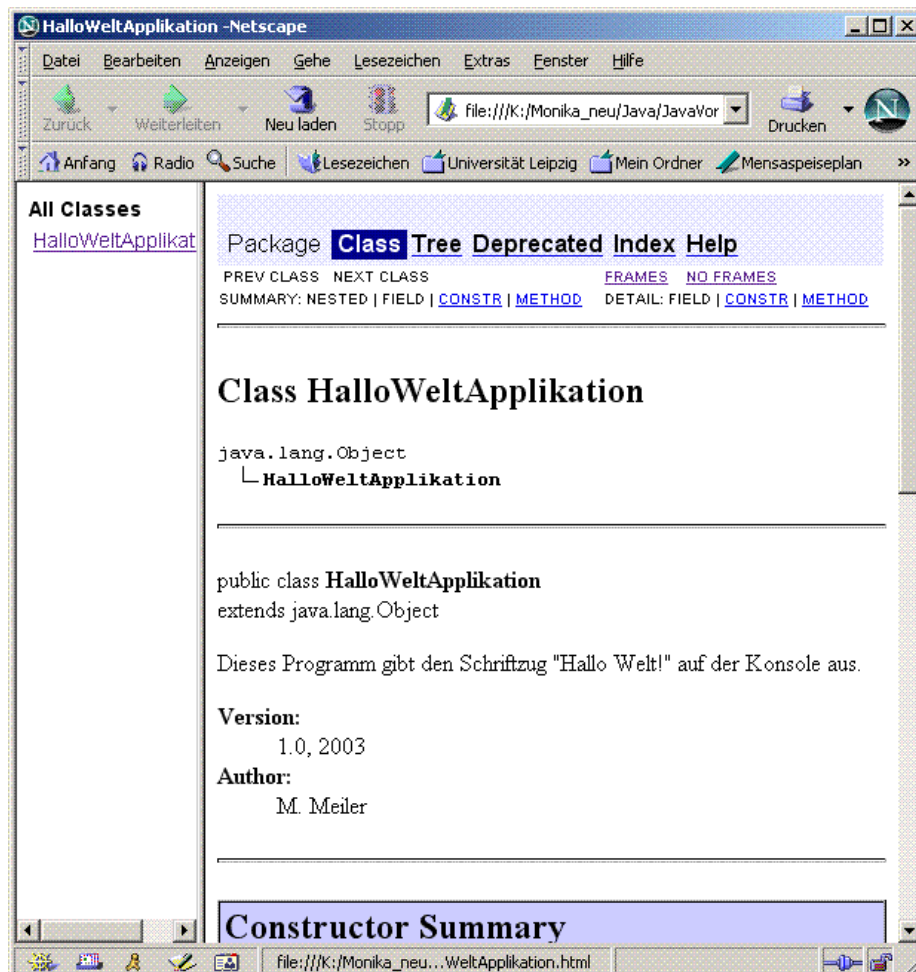
Aus Kommentaren, die mit `/**` und `*/` eingeschlossen werden, können automatisch HTML-Seiten generiert werden. Das *Sun-Programm* `javadoc` generiert anhand der Struktur einer Klasse und den in ihr enthaltenen *Dokumentationskommentaren* eine *Online-Dokumentationen* zu Klassen und legt diese im selben Verzeichnis ab. Deshalb stehen diese Kommentare stets unmittelbar vor den dokumentierten Klassen, ihren Attributen und Methoden.

HalloWeltApplication.java

```
// HalloWeltApplikation.java
/*                               Mein erstes Programm                               */

/**
 * Dieses Programm gibt den Schriftzug "Hallo Welt!"
 * auf der Konsole aus.
 * @author M. Meiler
 * @version 1.0, 2003
 */
public class HalloWeltApplikation
{
    /**
     * Hauptmethode erzeugt Bildschirmausschrift
     */
    public static void main( String[] args)
    {
        // Konsolenausgabe
        System.out.println( "Hallo Welt!");
    }
}
```

```
$ javadoc -author -version HalloWeltApplikation.java
```



3.4 Elementardatentypen

Java kennt vordefinierte **Elementardatentypen** für **ganze Zahlen**, **Gleitpunktzahlen** und **logische Werte**. Aus diesen können komplexere **strukturierte Datentypen** zusammengesetzt werden.

Die Wertebereiche der Datentypen ist maschinenunabhängig.

Ganze Zahlen

| Typ | kleinster Wert (MIN) | größter Wert (MAX) | Byte | Codierung |
|--------------|---|---|----------|-------------------|
| byte | -128 (-2^7) | 127 (2^7-1) | 1 | Komplement |
| short | -32'768 (-2^{15}) | 32'767 ($2^{15}-1$) | 2 | Komplement |
| int | -2'147'483'648 (-2^{31}) | 2'147'483'647 ($2^{31}-1$) | 4 | Komplement |
| long | -9'223'372'036'854'775'808 (-2^{63}) | 9'223'372'036'854'775'807 ($2^{63}-1$) | 8 | Komplement |
| char | 0 | 65535 ($2^{16}-1$) | 2 | Unicode |

Die Speicherung für *negative* ganze Zahlen erfolgt in *Komplementdarstellung*, sonst in *Dualdarstellung*. Der Datentyp **char** wird speziell für **Zeichen** genutzt, die *intern als ganze Zahlen im Unicode* dargestellt werden. Er gehört also zu den *ganzen Zahlen*.

Gleitpunktzahlen

| Typ | kleinster Wert (MIN) | größter Wert (MAX) | Byte | Codierung |
|---------------|---|--|----------|-----------------------------|
| float | $\pm 1.40239846 \text{ E } -45$ | $\pm 3.40282347 \text{ E } +38$ | 4 | IEEE 754³ |
| double | $\pm 4.940656458412465 \text{ E } -324$ | $\pm 1.797693138462315750 \text{ E } +308$ | 8 | IEEE 754 |

Logische Werte

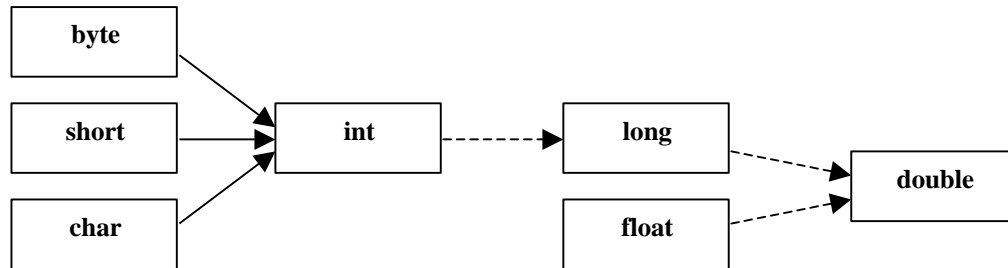
Der Datentyp **boolean** kennt nach der Booleschen Logik nur zwei Werte, **true** für wahr und **false** für falsch.

Implizite Typumwandlungen

Die auszuführende Operation ist vom jeweiligen **Operator** und den **Operandentypen** abhängig. Es gibt feste Regel zur **Typenkonvertierung** eines oder beider Operanden:

1. Eine Umwandlung von **boolean** in einen anderen Datentyp ist generell *nicht* möglich.
2. Mit dem Typen **byte**, **short** und **char** wird *nicht* gerechnet. Die Operanden werden in den Datentyp **int** umgewandelt.
3. Sind beide Operanden von unterschiedlichem Typ, so wird einer der beiden Operanden in den Typ des anderen nach der folgenden Grafik umgewandelt.

³ <http://www.h-schmidt.net/FloatApplet/IEEE754de.html>



Die Konvertierung erfolgt immer in Richtung des Pfeils. *Durchgezogene Pfeile* deuten an, dass diese Typumwandlung grundsätzlich vorgenommen wird. Die durch *gestrichelte Pfeile* dargestellten Typumwandlungen werden bei unterschiedlichen Operandentypen in arithmetischen Ausdrücken ausgeführt.

4. Sind beide Operanden vom selben Typ (oder wurden sie in diesen umgewandelt), so besitzt das Resultat ebenfalls diesen Typ.

Explizite Typumwandlungen (Casting)

Implizite Typumwandlungen werden immer dann vom Compiler automatisch durchgeführt, wenn *keine* Datenverluste auftreten, sonst erfolgt eine Fehlermeldung. Möchte man eine Typumwandlung *erzwingen* und evtl. Datenverluste hinnehmen, so kann man das explizit durch **typecast** erreichen.

(Typ) Ausdruck
(int) 3.14 => 3

3.5 Konstanten (Literele)

Konstanten haben einen *Typ* und einen *unveränderbaren Wert*, können einen *Namen* haben (**benannte Konstanten**) oder auch nicht (**unbenannte Konstanten**).

3.5.1 Unbenannte Konstanten

Unbenannte Konstanten sind **explizit** angegebene *Daten* im Programm. Diese werden durch bloßes **Hinschreiben** definiert, besitzen einen *Typ*, der sich aus der Schreibweise der Konstanten ergibt.

Ganzzahlige Konstanten (*integer-constant*)

29 = 035 = 0x1D = 0x1d

3 000 000 000

42l

-3554163L

Datentyp: **int**

Datentyp: **long**

decimal-constant $\{ 1, 2, 3, 4, 5, 6, 7, 8, 9 \} \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}^{*4}$

Beispiel: **1234**

octal-constant $0 \{ 0, 1, 2, 3, 4, 5, 6, 7 \}^*$

Beispiel: **0123** = $(123)_8 = 1 \cdot 8^2 + 2 \cdot 8 + 3 = 64 + 16 + 3 = 83$

hexa-decimal-constant $0 \{ x \mid X \} \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, a, B, b, C, c, D, d, E, e, F, f \}^{+5}$

Beispiel: **0x10Fa** = $(10fa)_{16} = 1 \cdot 16^3 + 15 \cdot 16 + 10 = 4096 + 240 + 10 = 4346$

Gleitpunktkonstanten (*floating-constant*)

18. = 18e0 = 1.8e1 = .18E2 = 0.018E3 = +18000E-3

18F

0.0012F

1.2E-3f

Datentyp: **double**

Datentyp: **float**

fractional-constant $Z . \{ 0 \}^* Z$ oder $. \{ 0 \}^* Z$ oder $Z .$
(*Festpunktschreibweise*) (Dezimalpunkt! *Z fractional-constant*)
Beispiel: **1.01234 .1234 1.**

floating-constant $\{ F \mid Z \} \{ e \mid E \} \{ | + | - \} Z$
(*Gleitpunktschreibweise*) (*F fractional-constant*)
Beispiel: **1.e1** = $1.0 \cdot 10^1 = 10.$
 .12e+3 = $0.12 \cdot 10^{+3} = 120.$
 5E-4 = $5 \cdot 10^{-4} = 0.0005$

^{4*} Iteration über eine Menge: Menge aller Wörter dieser Menge, **einschließlich** dem leeren Wort.

⁵⁺ Iteration über eine Menge: Menge aller Wörter dieser Menge, **ausschließlich** dem leeren Wort.

Als besondere Gleitpunktkonstanten sind definiert:

| Wert | Bedeutung | Beispiel |
|-------------------|----------------------|----------|
| POSITIVE_INFINITY | Infinity, $+\infty$ | 1 / 0 |
| NEGATIVE_INFINITY | -Infinity, $-\infty$ | -1 / 0 |
| NaN | Not a Number | 0 / 0 |

Zeichenkonstanten (*character-constant*)

Zeichenkonstanten werden in Apostrophe eingeschlossen und als ganze Zahlen durch ihren **Unicode** verschlüsselt.

| Zeichen | dezimal | hexadezimal | Bitfolge |
|---------|---------|-------------|---------------------|
| 'a' | 97 | \u0061 | 0000 0000 0110 0001 |
| '1' | 49 | \u0031 | 0000 0000 0011 0001 |
| '\n' | 10 | \u000A | 0000 0000 0000 1010 |

Explizite Angabe des Unicods eines Zeichens erfolgt **hexadezimal**. Gefolgt auf die Zeichen \u stehen vier Hexadezimalziffern (\u0000 bis \uffff).

Zeichenkettenkonstanten (*string-literal*)

Zeichenketten werden in Java als *Objekte einer Klasse* namens `String` behandelt. **Konstanten** dieses Typs werden als Folge von Zeichen aus dem verfügbaren Zeichensatz, eingeschlossen in Ausführungszeichen, dargestellt.

"Hallo Welt!"

Zeichenketten gehören nicht zu den einfachen Datentypen.

3.5.2 Benannte Konstanten

Sollen sich Werte von Variablen im Verlauf eines Programms nicht verändern, so richtet man sogenannte **final**-Variablen ein, auch als **benannte** bzw. **symbolische, Konstanten** bezeichnet. Dabei wird der *Typ* und der *Wert* der Konstanten direkt angegeben und das Schlüsselwort **final** vorangestellt.

```
final int MAX = 100;
```

Diese Schreibweise entspricht der üblichen *Variablen-Deklaration* (s. nächster Abschnitt) mit Initialisierung. *Jeder ändernde Zugriff* auf MAX wäre unzulässig.

3.6 Variablen

Beim Euklidischen Algorithmus werden als mathematischen Objekte natürliche Zahlen verarbeitet. Diese müssen im Rechner abgespeichert werden und auch wieder aufgefunden werden. Drei Fragen sind zu klären:

Wo befindet sich im Speicher die Zahl?
Wie viel Speicherplatz benötigt sie?
Wie wird sie abgespeichert?

Daten werden als **Bitfolgen** abgespeichert, d.h. Folgen aus 0 und 1. **Variablen** dienen als *Platzhalter* im Speicher für die Daten und besitzen *drei Grundbestandteile*:

Name symbolischer Bezeichner für die Anfangsadresse der Bitfolge
Typ Länge der Bitfolge und ihre Interpretationsvorschrift
Wert die interpretierte Bitfolge

Arbeitsspeicher

| Name <i>Symbolische Adresse</i> | Adresse <i>Adresse im Speicher</i> | Wert <i>Inhalt der Speicherzellen</i> | Typ <i>Interpretations- vorschrift</i> |
|--|---|--|---|
| | ... | ... | |
| b | 5e | 00 | short (2 Byte) |
| | 5f | 6a | |
| | ... | ... | |

Adresse *b* \Rightarrow $\&b = (5e)_{16} = (0101\ 1110)_2 = 64 + 16 + 8 + 4 + 2 = 94$

Wert von *b* \Rightarrow $b = (00\ 6a)_{16} = (0000\ 0000\ 0110\ 1010)_2 = 64 + 32 + 8 + 2 = 106$

Wäre die obige Bitfolge als *Zeichen*, d.h. als Wert vom Typ `char` (2 Byte), zu interpretieren, so hätte *b* den Wert 'k'.

Variablendeklaration und Initialisierer:

Alle Variablen müssen vor dem ersten Zugriff **deklariert** werden, d. h. dem Compiler muss der *Name* und der *Typ* bekannt gegeben werden, damit er den entsprechenden *Speicherplatz* zur Verfügung stellen kann und die *Interpretationsvorschrift* kennt.

Gleichzeitig ist eine **Initialisierung** der Variablen möglich. Ohne explizite Angabe eines *Initialisierers* haben die Variablen der Zahlendatentypen den **Wert 0** und der boolean-Datentypen den Wert **false**.

Typ Variablenname [= Ausdruck], Variablenname [= Ausdruck], ... ;

```
int Anzahl;           // Variable hat der Wert 0
float Zahl, Summe;    // Variable hat der Wert 0.0
float a = 1e10, b = 1e-10; // Initialisierung
float c = a + b; // möglich, da a und b deklariert
```

Die folgenden Programmzeilen liefern einen Fehler:

```
short a = 1, b = 2, c = a + b;
```

Fehlermeldung:

```
Ausdruecke.java:22: possible loss of precision
found   : int
required: short
    short a = 1, b = 2, c = a + b;
                                ^
1 error
```

Wegen der 1. Regel der Typumwandlung liefert das Ergebnis einen Wert vom Datentyp `int`. Da die Wertzuweisung an einen `short`-Typ Datenverluste nach sich ziehen könnte, muss die Typkonvertierung explizit durch *typecast* erzwungen werden:

```
short a = 1, b = 2, c = (short) ( a + b );
```

3.7 Ausdrücke

Entsprechend der üblichen Syntax, gegebenenfalls unter Verwendung der Klammer „(“ und „)“, lassen sich *Variable* und *Konstanten* mittels *Operatoren* zu **Ausdrücken** verknüpfen. Je nach dem **Hauptverknüpfungsoperator** unterscheidet man:

Java-Ausdrücke

Arithmetische Ausdrücke
 Bitausdrücke
 Wertzuweisungen
 Inkrementieren und Dekrementieren
 Logische Ausdrücke (Bedingungen)
 Bedingte Ausdrücke

3.7.1 Arithmetische Ausdrücke

| | |
|--------------------|--|
| Operatoren: | $+$ $-$ (unär) $*$ $/$ $\%$ $+$ $-$ (binär) |
| Operanden: | byte, short, int, long, float, double, char |
| Syntax: | wie üblich, mit Klammern. |

Wegen der 3. Regel zur Typumwandlung ist für die ganzzahlige Division kein gesondertes Operationszeichen notwendig.

| | | | |
|-------------|---------------|--------------------|--------|
| $7 / 3$ | \Rightarrow | 2 | int |
| $7.0 / 3.0$ | \Rightarrow | 2.3333333333333335 | double |
| $7.0 / 3$ | \Rightarrow | 2.3333333333333335 | double |
| $1 / 2 * 2$ | \Rightarrow | 0 | int |

Explizite Typeumwandlungen

| | | | |
|--------------------------|---------------|--------------------|--------|
| $7 / 3f$ | \Rightarrow | 2.333333 | float |
| $(\text{float}) 7 / 3.$ | \Rightarrow | 2.3333333333333335 | double |
| $(\text{float})(7 / 3.)$ | \Rightarrow | 2.333333 | float |
| $(\text{float})(7 / 3)$ | \Rightarrow | 2.0 | float |

3.7.2 Bitoperatoren

| | | |
|------------------------------|---|--|
| Operatoren: | \ll \gg \ggg Verschieben $\&$ $ $ \wedge Und Oder Exklusiv-Oder \sim Negation | (binär) (binär) (unär) |
| Operanden ganzzahlig: | byte, short, int, long, char | |
| Syntax: | wie üblich, mit Klammern. | |

\sim bitweise Negation
 $\&$ bitweise Verknüpfung durch ein logisches **Und**
 $|$ bitweise Verknüpfung durch ein logisches **Oder**

- \wedge bitweise Verknüpfung durch ein logisches **Exklusiv-Oder**
- \ll bitweise Verschieben um angegebene Stellenzahl nach links, Auffüllen mit 0-Bits
- \gg bitweise Verschieben um angegebene Stellenzahl nach rechts, Auffüllen mit dem höchsten Bits (vorzeichenerhaltend, arithmetisches Verschieben)
- \ggg bitweise Verschieben um angegebene Stellenzahl nach rechts, Auffüllen mit 0-Bits(nicht vorzeichenerhaltend, logisches Verschieben)

| | $\&$ | | $ $ | | \wedge | | \sim |
|---|------|---|-----|---|----------|---|--------|
| | 0 | 1 | 0 | 1 | 0 | 1 | |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |

In den folgenden Beispielen seien Operanden und Ergebnisse vom Datentyp `byte`.

Negation

```

~ 1      : ~ 0000 0001      => 1111 1110  => -2
~~ 1     : ~ -2              => 1

```

Und

```

1 & 3    : 0000 0001 & 0000 0011 => 0000 0001  => 1

```

Oder

```

1 | 3    : 0000 0001 | 0000 0011 => 0000 0011  => 3

```

Exklusiv-Oder

```

1 ^ 3    : 0000 0001 ^ 0000 0011 => 0000 0010  => 2

```

Verschieben

```

-1 << 3  : 1111 1111 << 3      => 1111 1000  => -8
-1 >> 3   : 1111 1111 >> 3      => 1111 1111  => -1
-1 >>> 3  : 1111 1111 >>> 3     => 0001 1111  ⚡ -1?

```

Man beachte (1. Regel der Typumwandlung), dass intern nur mit `int` oder `long` gerechnet wird:

```
-1 >>> 3: 1111 1111 >>> 3
```

```
=> Konvertierung nach int
```

```
=> 1111 1111 1111 1111 1111 1111 1111 1111 >>> 3
```

```
=> 0001 1111 1111 1111 1111 1111 1111 1111
```

```
=> 536'870'911
```

```
=> Konvertierung nach byte
```

```
=> 1111 1111
```

```
=> -1!
```

3.7.3 Wertzuweisungen

Operatoren: `= *= /= %= += -= &= ^= |= <=> >>= >>>=` (binär)

Syntax: *einfache Wertzuweisung*

Variable = Ausdruck

zusammengesetzte Wertzuweisung

Variable °= Ausdruck mit $\circ \in \{*, /, \%, +, -, <, >, \&, \wedge, | \}$

```
int x, y, z;
```

```
x = y = 5;
```

```
=> x = 5, y = 5
```

```
x += y + ( z = 1 );
```

=> $z = 1, x = 11, y = 5$

$x = \mathbf{Aus}$

1. Der Variablen x wird der Wert des Ausdrucks **Aus** zugewiesen.
2. Der komplexe Ausdruck $x = \mathbf{Aus}$ erhält den Wert des rechten Ausdrucks **Aus**.

$x \circ = \mathbf{Aus}$ ist äquivalent mit $x = x \circ \mathbf{Aus}$, Unterschied: Anzahl der Speicherzugriffe

Konvertierung

Können in Wertzuweisungen auf Grund des Datentyps der Variable Datenverluste eintreten, wird dies als Fehler ausgewiesen. Wünscht man trotzdem eine Typkonvertierung, so muss diese explizit durch *typecast* erzwungen werden:

```
short a, b, c;  
c = (short) ( a + b );
```

Nebeneffekte

```
x = 1; x = x + ( x += 10 );           => 1 + 11 => x = 12  
x = 1; x = ( x += 10 ) + x;         => 11 + 11 => x = 22
```

In Ausdrücken sollten die Variablen mit Wertzuweisung an keiner weiteren Stelle auftreten.

besser:

```
x = 1; x += 10; x *= 2;                => 11 * 2 => 22
```

3.7.4 Inkrementieren und Dekrementieren

| | | |
|--------------------|-------------------------|------------------|
| Operatoren: | ++ Inkrementator | (unär) |
| | -- Dekrementator | (unär) |
| Syntax: | ++ Variable | (präfix) |
| | -- Variable | |
| | Variable ++ | (postfix) |
| | Variable -- | |

```
y = z = 5;  
x = --y + 5;           => x = 9, y = 4  
x *= y++ + ++z;       => x = 90, y = 5, z = 6
```

Semantik:

| | | | |
|--------------|-------------|-----------------------------|------------------|
| ++ x | x ++ | ist äquivalent mit | x = x + 1 |
| -- x | x -- | ist äquivalent mit | x = x - 1 |
| Unterschied: | | Anzahl der Speicherzugriffe | |

Postfix-Schreibweise ($x++$, $x--$)

Keine unmittelbare Wirkung auf den Wert des komplexen Ausdrucks, x hat während der Berechnung an dieser Stelle noch den alten Wert.

```
x = 1; y = 3 * x++;           => x = 2, y = 3
```

Präfix-Schreibweise (`++ x`, `-- x`)

Vor der Auswertung des komplexen Ausdrucks wird der Wert von `x` verändert.

`x = 1; y = 3 * ++x;` $\Rightarrow x = 2, y = 6$

Nebeneffekte

`x = 1; y = x++ + x;` $\Rightarrow 1 + 2 \Rightarrow x = 2, y = 3$

`x = 1; y = ++x + x;` $\Rightarrow 2 + 2 \Rightarrow x = 2, y = 4$

In Ausdrücken sollten inkrementierte bzw. dekrementierte Variablen an keiner weiteren Stelle auftreten.

besser:

`x = 1; ++x; y = x + x;`

3.7.5 Logische Ausdrücke (Bedingungen)

| | | | |
|------------------------------|--|------------|---------|
| Vergleichsoperatoren: | <code>< <= > >= == !=</code> | Vergleiche | (binär) |
| logische Operatoren: | <code>& && </code> | Und Oder | (binär) |
| | <code>!</code> | Negation | (unär) |

Operanden bei logische Operatoren und Ergebnistyp: boolean

Syntax: wie üblich, mit Klammern.

```
int x = 2; boolean y, z;
y = 0 < x & x <= 2;           => true & true => y = true
z = x == 4;                   => z = false
```

| | &, && | | , | | ! |
|-------|-------|-------|-------|------|-------|
| | false | true | false | true | |
| false | false | false | false | true | true |
| true | false | true | true | true | false |

Optimierung

Die logischen Operatoren `&&` und `||` werden grundsätzlich optimierend ausgewertet:

- **&&**: Ist der erste Operand falsch, so wird der zweite Operand nicht ausgewertet.
- **||**: Ist der erste Operand wahr, so wird der zweite Operand nicht ausgewertet.

```
// Division durch 0 wird vermieden.
double x = 1, y = 0, toleranz = 0.1; ...
if( y == 0 || x / y > toleranz ) ...
```

3.7.6 Bedingte Ausdrücke

Operatoren: `? :`

Syntax: *Bedingung ? Ausdruck : Ausdruck*

Wenn die *Bedingung* wahr ist, wird der erste *Ausdruck* ausgewertet, sonst wird, falls vorhanden, der zweite *Ausdruck* abgearbeitet. Verschachtelungen sind möglich.

Mit dem Programmausschnitt wird der Quadrant eines Punktes im kartesischen Koordinatensystem bestimmt.

```
double x = 1, y = -1; int Quadrant;  
  
Quadrant = x >= 0? y >=0? 1: 4: y >= 0? 2: 3;  
  
System.out.println  
( "Der Punkt (" + x + ", " + y + ") liegt im Quadrant "  
  + Quadrant + "!" );  
  
⇒ Der Punkt (1.0,-1.0) liegt im Quadrant 4!
```

Das Maximum zweier Zahlen x und y wird wie folgt durch einen bedingten Ausdruck bestimmt.

```
max = ( x > y ) ? x : y;
```

3.8 Zusammenfassung

Die Menge aller **Ausdrücke** wird wie folgt zusammengefasst:

1. Konstanten, Variablen und Methodenaufrufe sind Ausdrücke
2. Ausdrücke verknüpft mit Operatoren ergeben wieder Ausdrücke, wobei die übliche Klammerung erlaubt sind und folgende Vorrangregeln gelten:

Java-Operatoren mit Rangfolge und Assoziativitätsrichtung:

| | | | |
|----|---------------------|--|---|
| 15 | () [] . | Ausdrucksgruppierung Auswahl der Feldkomponenten Auswahl der Klassenkomponenten | → |
| 14 | ! ~ ++ -- + - | Negation (logisch, bitweise) Inkrementation, Dekrementation (Präfix oder Postfix) Vorzeichen | ← |
| 13 | (Typ) | explizite Typumwandlung | ← |
| 12 | * / % | Multiplikation, Division Rest bei ganzzahliger Division | → |
| 11 | + - | Summe, Differenz | → |
| 10 | << >> >>> | bitweise Verschiebung nach links, rechts | → |
| 9 | < <= > >= | Vergleich auf kleiner, kleiner oder gleich Vergleich auf größer, größer oder gleich | → |
| 8 | == != | Vergleich auf gleich, ungleich | → |
| 7 | & | Und (bitweise, logisch) | → |
| 6 | ^ | exklusives Oder (bitweise) | → |
| 5 | | inklusive Oder (bitweise, logisch) | → |
| 4 | && | Und (logisch) | → |
| 3 | | inklusive Oder (logisch) | → |
| 2 | ? : | bedingte Auswertung (paarweise) | ← |
| 1 | = °= | Wertzuweisung zusammengesetzte Wertzuweisung (* =, / =, % =, + =, - =, & =, ^ =, =, < < =, > > =, > > > =) | ← |