Inhalt

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 des **16-Bit-Basiszeichensatzes UTF-16** des **Unicodes**[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. Der ASCII-Code ist eine *Teilmenge* dieses Zeichensatzes.

Intern arbeitet Java komplett mit *Unicode*. Dazu sind entsprechende *Umwandlungsfunktionen (Konvertierung)* 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:

`\b`	Versetzen um eine Position nach links (**backspace**)
`\f`	Seitenvorschub (**form feed**)
`\n`	Zeilenvorschub (**line feed, new line**)
`\r`	Positionierung am Zeilenanfang (**carriage return**)
`\t`	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*:

`\'`	Entwerten des Abschlusssymbols für Zeichenkonstante
`\"`	Entwerten des Abschlusssymbols für Zeichenkettenkonstante
`\\`	Entwerten des Backslashs als Escapesequenz

[1] ASCII - ***A**merican **S**tandard **C**ode for **I**nformation **I**nterchange*
[2] Unicode-Konsortium http://www.unicode.org/

3.2 Bezeichner

Für die Bezeichnung (Identifier) der *Objekte*, ihrer *Klassen*, *Attribute* und *Methoden*, *Variablen* und *Konstanten* werden **Namen** benötigt:

Schlüsselwörter sind spezielle Bezeichner der Sprache Java (`int`, `if`, `public`, ...).

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* entsprechen.

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

Groß- und Kleinbuchstaben sind *signifikant*.

Faustregeln Namensgebung

- ⇨ **Namen beschreiben den Inhalt: *"Sprechende" Namen*.**
- ⇨ ***Klassennamen* beginnen mit Großbuchstaben: `HalloWeltApplikation`.**
- ⇨ ***Variablen-, Objekt-, Attribut-* und *Methodennamen* beginnen mit Kleinbuchstaben: `lampe`.**
- ⇨ **Bei *zusammengesetzten Namen* beginnen weitere Wörter jeweils mit einem Großbuchstaben: `setLampe`.**
- ⇨ ***Konstanten* werden mit Großbuchstaben bezeichnet: `MAX`.**
- ⇨ **Bei *zusammengesetzten* Konstanten werden weitere Worte jeweils durch einen Unterstrich getrennt: `MAX_VALUE`.**

3.3 Kommentare

Kommentare dienen der *Lesbarkeit* und *Verständlichkeit* von Programmen und ihren Quelltexten. *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.

HalloWeltApplication.java

```
// HalloWelt.java                                MM 2009
/*               Mein erstes Programm                  */

/**
 * Konsolenausgabe des Schriftzugs "Hallo Welt!".
 */
public class HalloWelt
{
/**
 * Hauptmethode, erzeugt Bildschirmausschrift.
```

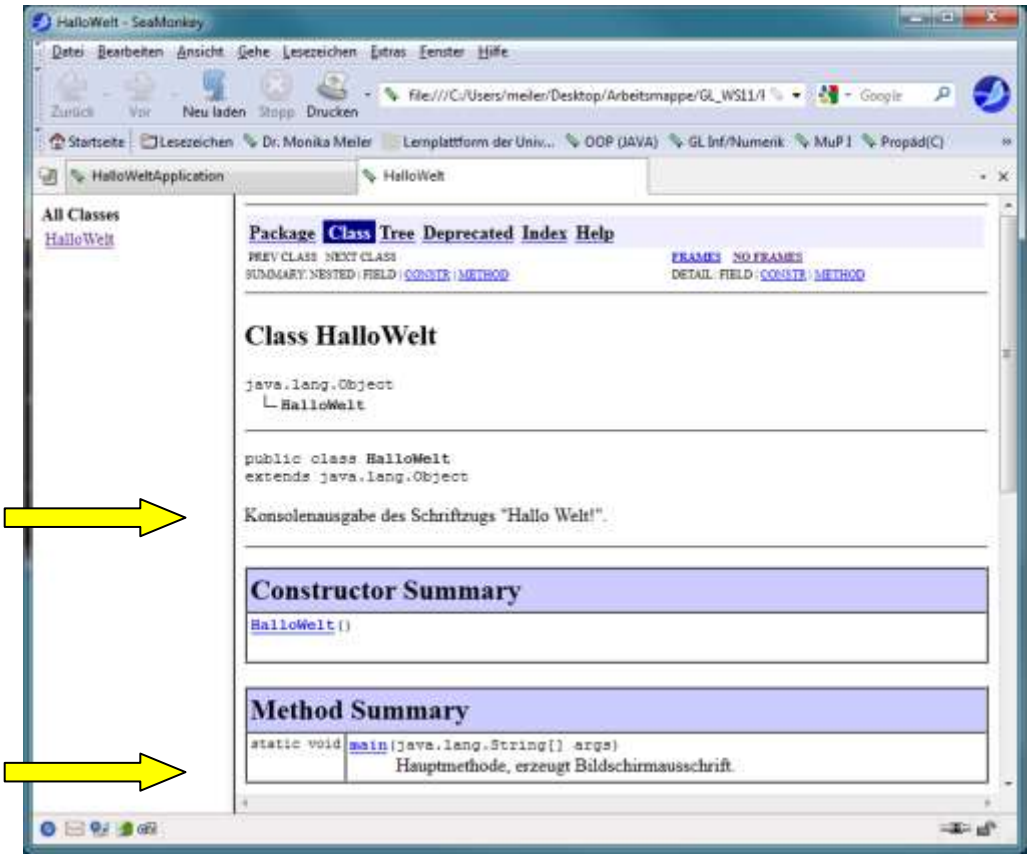
```
   */
    public static void main( String[] args)
    {                                          // Konsolenausgabe
      System.out.println( "Hallo Welt!");
    }
  }
```

Interne Dokumentation

1. `// Zeilenkommentar` (*Programmbeschreibung*)
   Alle dem Kommentarzeichen `//` folgende Zeichen bis Zeilenende werden vom Compiler ignoriert. Sie dienen vor allem dem Programmierer zur *Programmbeschreibung*.
2. `/* Kommentar */` (*Auskommentieren von Testcode*)
   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 */` (*Online-Dokumentation*)
   Aus Kommentaren, die mit `/**` und `*/` eingeschlossen werden, können automatisch HTML-Seiten generiert werden. Das *Java-Programm* `javadoc` generiert anhand der Struktur einer Klasse und den in ihr enthaltenen *Dokumentationskommentaren* eine *Online-Dokumentationen* und legt diese im aktuellen Verzeichnis ab. Externe Kommentare stehen stets unmittelbar vor den dokumentierten Klassen, ihren Attributen und Methoden.
   `$ javadoc HalloWeltApplication.java`



Java Plattform, All Classes

3.4 Elementardatentypen

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

Ganze Zahlen

Typ	kleinster Wert (MIN_VALUE)	größter Wert (MAX_VALUE)	Byte	Codierung
`byte`	-128 (-2^7)	127 (2^7-1)	**1**	**Direktcode / Komplement**
`short`	-32´768 (-2^{15})	32´767 (2^{15}-1)	**2**	
`int`	-2´147´483´648 (-2^{31})	2´147´483´647 (2^{31}-1)	**4**	
`long`	-9´223´372´036´854´775´808 (-2^{63})	9´223´372´036´854´775´807 (2^{63}-1)	**8**	
`char`	0	65535 (2^{16}-1)	**2**	**UTF-16**

Der Datentyp **`char`** wird speziell für **Zeichen** genutzt, intern als *ganze Zahlen* im Unicode UTF-16 dargestellt. Er gehört also zu den *ganzen Zahlen*.

Gleitpunktzahlen

Typ	kleinster Wert (MIN_VALUE)	größter Wert (MAX_VALUE)	Dezimal-stellen	Byte	Codierung
`float`	±1.40239846 E -45	±3.40282347 E +38	7	**4**	**IEEE 754[3] single**
`double`	±4.940656458412465 E -324	±1.797693138462315750 E +308	15	**8**	**IEEE 754 double**

Logische Werte

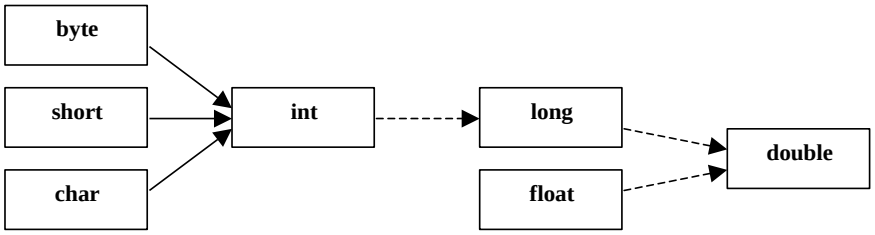
Der Datentyp **`boolean`** kennt entsprechend der Booleschen Logik nur zwei Werte, **`true`** für *wahr* und **`false`** für *falsch*.

Wert	*Bedeutung*
`true`	wahr
`false`	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 den 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 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 x = Math.PI; ⇒  Fehler
int x = (int)Math.PI; ⇒  3
```

Standard für rationalen Zahlenbereiche

(IEEE – *Standard, Institute of Electrical and Electronics Engineers***)**

Bezeichnung	Byteanzahl	*s* [Bit]	*M* [Bit]	*E* [Bit]	*r*	*R*	*C*
`single`	4	1	24	8	-126	127	127
`double`	8	1	53	11	-1022	1023	1023

s Vorzeichenbit: 0 ... +, 1 ... -
M Länge der Mantisse, einschließlich *hidden bit*
E Anzahl der Exponentenbit
r kleinster Exponent
R größter Exponent
C Verschiebungskonstante

***Sonderbehandlung*:**

	$M' = 0$	$M' \neq 0$
$E' = 0$ $(E = r - 1)$	$z = 0.0$	$z = (-1)^s * 2^r * (0.M')_2$; zusätzliche Zahlen im Unterlauf
$E' = E'_{max}$ $(E = R + 1)$	$z = (-1)^s * \infty$; `-Infinity, Infinity`	`NaN` (Not a Number); keine gültige Gleitpunktzahl

***IEEE*:**

Typ	`single`	`double`
MAX < 2^{R+1}	±3.40282347 E +38	±1.79769313486231570 E +308
MIN = 2^r	±1.17549435 E -38	±2.22507385850720138 E -308
MIN Sonderbehandlung	±1.40239846 E -45	±4.940656458412465 E -324
gültige Dezimalstellen	7	15

3.5 Konstanten (Literale)

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 und besitzen einen *Typ*, der sich aus der Schreibweise der Konstanten ergibt.

Ganzzahlige Konstanten (integer-constant)

```
29 = 035 = 0x1D = 0x1d                          Datentyp: int
3 000 000 000, 42l, -3554163L, (long)18         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*8^2+2*8+3 = 64+16+3 = 83
hexa-decimal-constant	0{ x | 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*16^3+15*16+10 = 4096+240+10 = 4346

Gleitpunktkonstanten (floating-constant)

```
18. = 18e0 = 1.8e1 = .18E2 = +18000E-3          Datentyp: double
18F = 18f = (float)18                           Datentyp: float
0.0012F, 1.2E-3f
```

fractional-constant (Festpunktschreibweise)	*Z* . { 0 }[*] *Z* oder . { 0 }[*] *Z* oder *Z* . (Dezimalpunkt! *Z* ***decimal-constant***)
Beispiel:	**1.01234 .1234 1.**
floating-constant (Gleitpunktschreibweise)	{ *F* | *Z* }{ e | E }{ | + | - }*Z* (*F* ***fractional-constant***)
Beispiel:	**1.e1** = 1.0*10^1 = 10. **.12e+3** = 0.12*10^+3 = 120. **5E-4** = 5*10^-4 = 0.0005

[4] ***** Iteration über eine Menge: Menge aller Wörter dieser Menge, **ein**schließlich dem leeren Wort.
[5] **+** Iteration über eine Menge: Menge aller Wörter dieser Menge, **aus**schließlich dem leeren Wort.

Als besondere Gleitpunktkonstanten sind definiert:

Wert	*Bedeutung*	*Beispiel*
`Inf`	$+\infty$	`1 / 0`
`-Inf`	$-\infty$	`-1 / 0`
`NaN`	Not a Number	`0 / 0`

Zeichenkonstanten (character-constant)

Zeichen werden in Apostrophe eingeschlossen und durch ihren **Unicode** verschlüsselt.

Zeichen	*dezimal*	*hexadezimal*	*Unicode*	*Bitfolge*
`'a'`	97	0x61	`'\u0061'`	0000 0000 0110 0001
`'1'`	49	0x31	`'\u0031'`	0000 0000 0011 0001
`'\n'`	10	0x0A	`'\u000A'`	0000 0000 0000 1010

Explizite Angabe des UTF-16-Codes eines Zeichens erfolgt **hexadezimal**. Gefolgt auf die Zeichen `\u` stehen vier Hexadezimalziffern (`'\u0000'` bis `'\uffff'`).

```
char a = 'a', b = 98, c = 0x63, d = '\u0064';
```

Logische Konstanten (boolean-constant) `true`, `false`

Zeichenkettenkonstanten (string-literal)

Zeichenketten sind in Java *Objekte einer Klasse* `String`. Darauf können wir hier noch nicht genauer eingehen. **Konstanten** dieses Typs werden als Folge von Zeichen aus dem verfügbaren Zeichensatz, eingeschlossen in Ausführungszeichen, dargestellt.

```
String hallo = "Hallo Welt!";
```

Zeichenketten gehören *nicht* zu den *Elementardatentypen*.

3.5.2 Benannte Konstanten

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

Jeder ändernde Zugriff auf `MAX` wäre unzulässig: **`final`** `int MAX = 100;`

Für jeden Elementardatentyp sind zum Beispiel das Maximum und das Minimum vordefiniert.

`long`: **`Long.MAX_VALUE, Long.MIN_VALUE`**

3.6 Variablen

Beim Euklidischen Algorithmus werden als mathematische 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

```
short b = 106;
```

Arbeitsspeicher

Name *Symbolische Adresse*	**Adresse** *Adresse im Speicher*	**Wert** *Inhalt der Speicherzellen*	**Typ** *Interpretations-vorschrift*
	...	...	
b	*5e*	00	`short` (2 Byte)
	5f	6a	
	...	...	

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

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

```
char b = 'j';
```

Wäre die obige Bitfolge als *Zeichen* zu interpretieren, d.h. als Wert vom Typ `char` (2 Byte), so wäre unter `b` das Zeichen `'j'` abgespeichert.

Variablendeklaration und Initialisierer

Alle Variablen müssen *vor der ersten Verwendung* **deklariert** werden, d. h. dem Compiler muss *Name* und *Typ* angezeigt 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 den Wert 0
float zahl, summe;       // Variablen haben den Wert 0.0
float a = 1e10, b = 1e-10;           // Initialisierung
float c = a + b;     // moeglich, da a und b deklariert
```

Eine Variablendeklaration ist an jeder Stelle im Programm erlaubt, jedoch vor der ersten Verwenden der Variablen notwendig.

Die folgenden Programmzeilen liefern einen Fehler:

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

Fehlermeldung (Hinweis auf mögliche Einbuße von Genauigkeit):

```
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 *Variablen* 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**

Einige Beispiele sollen die Besonderheiten der Ausdrucksbildung in Java dokumentieren.

Arithmetische Ausdrücke **{ +, -, *, /, %}**
Wegen der 3. Regel zur Typumwandlung ist für die ganzzahlige Division kein gesondertes Operationszeichen notwendig.

```
    7 / 3   ⇒                    2          int
7.0 / 3.0   ⇒  2.3333333333333335       double
  7.0 / 3   ⇒  2.3333333333333335       double
1 / 2 * 2   ⇒                    0          int
```

Bitausdrücke **{ |, ^, &, <<, >>, >>>, ~}**

	&		|		^		~
	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    :  ~ 1111 1110          ⇒ 0000 0001 ⇒  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

- `<<` bitweises Verschieben um angegebene Stellenzahl nach links, Auffüllen mit 0-Bits
- `>>` bitweises Verschieben um angegebene Stellenzahl nach rechts, Auffüllen mit dem höchsten Bit (vorzeichenerhaltend, arithmetisches Verschieben)
- `>>>` bitweises Verschieben um angegebene Stellenzahl nach rechts, Auffüllen mit 0-Bits (nicht vorzeichenerhaltend, logisches 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!**

Wertzuweisungen **{ =, °=}**

```
x = y = 5;                          ⇒ x = 5, y = 5
x += y + ( z = 1 );            ⇒ z = 1, x = 11, y = 5
```

Inkrementieren und Dekrementieren **{ ++, --}**

```
y = z = 5;                          ⇒ y = 5, z = 5
x = --y + 5;                        ⇒ x = 9, y = 4
x *= y++ + ++z;                ⇒ x = 90, y = 5, z = 6
--x;                                      ⇒ x = 89
```

Vergleiche **{ ==, !=, <, <=, >, >=}**

Logische Ausdrücke (Bedingungen) **{ ||, &&, |, &, !}**

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

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

Optimierung bei &&, ||

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

Bedingte Ausdrücke **{ ?, :}**

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

3.8 Zusammenfassung

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

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

Java-Operatoren mit Rangfolge und Assoziativitätsrichtung:

15	**( )** **[ ]** **.**	Ausdrucksgruppierung Auswahl der Feldkomponenten Auswahl der Klassenkomponenten	→
14	**! ~** **++ --** **+ -**	Negation (logisch, bitweise) Inkrementieren, Dekrementieren (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	|	inklusives Oder (bitweise, logisch)	→
4	**&&**	Und (logisch)	→
3	||	inklusives Oder (logisch)	→
2	**? :**	bedingte Auswertung (paarweise)	←
1	**=** **°=**	Wertzuweisung zusammengesetzte Wertzuweisung (*=, /=, %=, +=, -=, &=, ^=, |=, <<=, >>=, >>>=)	←