Einführung in die Programmierung

Objektorientierte Programmierung mit Java

02 : Ausdrücke und Operatoren

Dr. Rudolf Scheurer

rudolf.scheurer@hefr.ch

(Skript: Jacques Bapst)

Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Ausdrücke [1]

- **Ausdrücke** bestehen aus Literalen, Variablen, Methodenaufrufen und Operatoren. Beispiel: `4 + a / f(x)`
- **Ausdrücke** sind syntaktische Elemente zur Bestimmung eines Wertes aus anderen Werten
- Die **Auswertung eines Ausdrucks** gibt einen **Wert** und einen **Typ** zurück
- **Primärer Ausdruck**: Literal oder Variable
- Der resultierende **Wert** der Auswertung eines primären Ausdrucks ist der literale Wert oder der Wert, der in der Variable gespeichert ist. Der resultierende **Typ** ist derjenige des Literals oder der Variablen
- Beispiele :

```
141     // ganzzahliges Literal (Typ int, Wert 141)
true    // boolesches Literal (Typ boolean, Wert true)
total   // Variable  (Typ je nach Deklaration der Variablen,
                     aktueller Wert von total)
```

Ausdrücke [2]

- **Komplexe Ausdrücke** sind aus **primären Ausdrücken** (**Operanden**) und **Operatoren** zusammengesetzt.
- Beispiele :

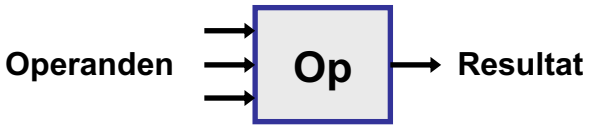
```
total = 141       // Zuweisungs-Ausdruck
1 + 2             // Ausdruck mit Additions-Operator
total = 7 + b     // Ausdruck bestehend aus einer Addition
                  // und einer Zuweisung
r = p.min(a, b)   // Ausdruck bestehend aus einem Methodenaufruf
```

- **Wert** und **Typ** eines komplexen Ausdrucks werden sowohl durch die Werte und Typen der Operanden als auch durch die Operatoren bestimmt, die im Ausdruck vorkommen (die Regeln folgen später).
- Gewisse Operatoren haben **Nebeneffekte**: sie ändern den Wert der Operanden und verändern somit den Zustand des Programms (Zuweisung, Inkrementation, Dekrementation, Methodenaufruf, Objekterstellung)

```
a = 2
i++
```

Operatoren

- **Operatoren** sind syntaktische Elemente, welche unter Verwendung ihrer **Operanden** (Parameter der Operation) gewisse Operationen durchführen



- **Unäre Operatoren** (einwertig) *1 Operand*

```
-b              // einwertiger Operand minus
a++             // einwertiger Operand Post-Inkrementation
```

- **Binäre Operatoren** (zweiwertig) *2 Operanden*

```
a * b           // der Multiplikationsoperator ist binär
y = z           // die Zuweisung ist auch binär
(int)y          // Typenkonversion (casting)
```

- **Ternäre Operatoren** *3 Operanden*
  - Ein einziger ternärer Operator in *Java* (nicht zu empfehlen)

```
x > y ? x : y   // gibt den grösseren Wert von x und y zurück
```

Liste der Operatoren [1]

P	A	Operator	Operandentyp(en)	Operation

15	L	`.`
		`[]`
		`( params )`
		`++, --`
			Objekt, Element
			Array, int
			Methode, Parameterliste
			Variable
				Zugriff auf ein Element eines Objekts
				Zugriff auf ein Element eines Arrays
				Methodenaufruf
				Post-Inkrementation/Dekrementation

14	R	`++, --`
		`+, -`
		`~`
		`!`
			Variable
			Zahl
			Ganzzahl
			Boolescher Wert
				Pre-Inkrementation/Dekrementation
				Einwertiges plus, einwertiges minus
				Binäres Komplement
				Logisches NICHT

13	R	`new` `( type ) expr`	Klasse, Parameterliste Typ, beliebig	Erstellung (Instanzierung) eines Objekts Typenwandlung
12	L	`*, /, %`	Zahl, Zahl	Multiplikation, Division, Modulo
11	L	`+, -` `+`	Zahl, Zahl Beliebige Zeichenkette	Addition, Subtraktion Verkettung
10	L	`<<, >>, >>>`	Ganzzahl, Ganzzahl	Verschiebung links/rechts, ohne Vorzeichen

9	L	`<, <=`
		`>, >=`
		`instanceof`
			Zahl, Zahl
			Zahl, Zahl
			Referenz, Typ
				Kleiner, kleiner gleich
				Grösser, grösser gleich
				Typenvergleich

8	L	`==`
		`!=`
		`==`
		`!=`
			Einfach, einfach
			Einfach, einfach
			Referenz, Referenz
			Referenz, Referenz
				Gleichheit (gleiche Werte)
				Ungleichheit
				Gleichheit (Referenz zum selben Objekt)
				Ungleichheit

P : Präzedenz A : Assoziativität

Liste der Operatoren [2]

P	A	Operator	Operandentyp(en)	Operation
7	L	`&` `&`	Ganzzahl, Ganzzahl boolescher Wert, boolescher Wert	Binäres UND (Bit für Bit) Logisches UND
6	L	`^` `^`	Ganzzahl, Ganzzahl boolescher Wert, boolescher Wert	Binäres EXCUSIV-ODER (Bit für Bit) Logisches EXCUSIV-ODER
5	L	`|` `|`	Ganzzahl, Ganzzahl boolescher Wert, boolescher Wert	Binäres ODER (Bit für Bit) Logisches ODER
4	L	`&&`	boolescher Wert, boolescher Wert	Bedingtes[1)], logisches UND
3	L	`||`	boolescher Wert, boolescher Wert	Bedingtes[1)], logisches ODER
2	R	`? :`	boolescher Wert, beliebig, beliebig	Ternärer Bedingungs-Operator
1	R	`=` `*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=`	Variable, beliebig Variable, beliebig	Zuweisung Zuweisung mit Operation

P : Präzedenz A : Assoziativität

[1)] *Der Operand auf der rechten Seite wird nur wenn nötig evaluiert*

Präzedenz und Assoziativität

- **Präzedenz** *(siehe Spalte* **P** *in der Operatorenliste)*
  - Definiert die Reihenfolge, in welcher die Operatoren ausgeführt werden (wenn keine Klammern verwendet werden)
  - Je höher der Präzedenzgrad ist, desto stärker sind die Operanden an den betreffenden Operator gebunden

    `a+b * c` → `a+(b*c)`

  - **Klammern** definieren explizit die Präzedenz der Operationen
  - ***Tipp*** *: Ausser in Trivialfällen sollten in komplexen Ausdrücken* ***immer*** *Klammern verwendet werden, um jeden Interpretationszweifel auszuschliessen.*

- **Assoziativität** *(siehe Spalte* **A** *in der Operatorenliste)*
  - Falls mehrere Ausdrücke vom selben Präzedenzgrad vorkommen, definiert die Assoziativität, die Reihenfolge, in welcher die Operationen durchgeführt werden
    - Von Links nach Rechts (**L**) `a+b+c` → `(a+b)+c`
    - Von Rechts nach Links (**R**) `a=b=c` → `a=(b=c)`

Arithmetische Operatoren

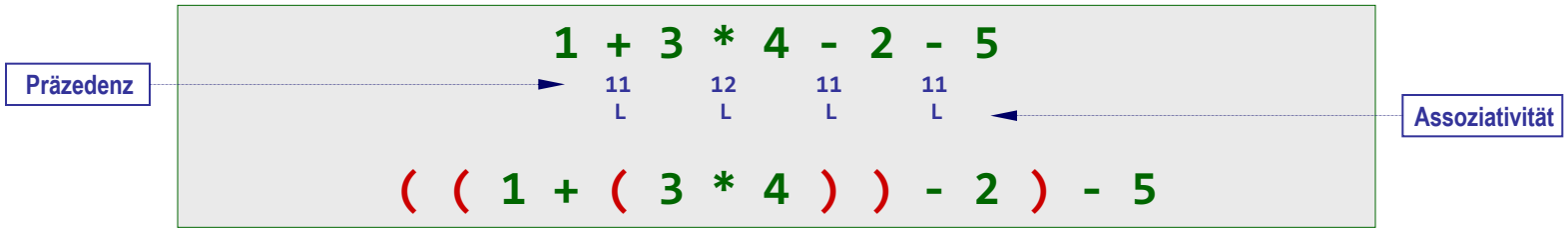
- Führen **Arithmetische Operationen** aus und geben numerische Werte zurück
- Der Typ des Resultats hängt vom Typ der Operanden ab
- Wenn nötig wird eine (automatische) erweiternde Wandlung der Operanden vor Ausführung der Operation durchgeführt.

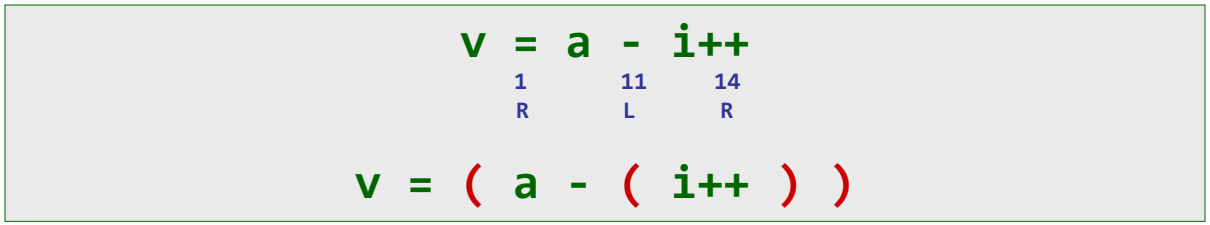
`+ -`	Einstelliges (unäres) Plus, Minus
`+ -`	Addition, Subtraktion (zweistellig, binär)
`++ --`	Pre/Post-Inkrementation, Dekrementation
`* / %`	Multiplikation, Division, Modulo (Rest der ganzzahligen Division)

```
-2.7
i++
--j - 2
a * x*x + 4.2
5.6/3
17 % 5    // nicht mit 17/5 zu verwechseln
```

Ausführungsreihenfolge der Operationen

- Um die Reihenfolge der Ausführung klarzustellen können Klammern verwendet werden, welche die Präzedenz und die Assoziativität der verwendeten Operatoren aufzeigen:





- Nach dem Setzen der Klammern sollten die Operatoren nur noch Ausdrücke in Klammern, eventuell ein Symbol oder einen einzelnen, literalen Wert als Operanden haben.
- Unvollständige Auswertung : `u = (a * b) - (c * d) + f(e);`

Auswertungsreihenfolge der Operanden

- In komplexeren (bzw. etwas „vermurksten“) Ausdrücken ist es erforderlich, dass gut zwischen der **Auswertungs**reihenfolge der **Operanden** (von links nach rechts) und der **Ausführungs**reihenfolge der **Operationen** (abhängig von der Präzedenz und Assoziativität) unterschieden wird.
- Solche Ausdrücke müssen **vermieden** werden (der Code muss unbedingt eindeutiger formuliert werden).

```
int i = 10;
i = i + (i=5);
```

```
int i=2;
i = ++i + ++i * ++i;
```

```
int i=1, j=2;
i = i+++j;
```

Ergänzung

Auswerten der Operationen [1]

- Wenn einer der Operanden eine Gleitkommazahl ist, wird die **Dezimalstellen-Arithmetik** (reelle Arithmetik) verwendet (Berechnung in Gleitkommaarithmetik)
- Wenn beide Operanden ganze Zahlen sind, wird die **Ganzzahl-arithmetik** verwendet (wichtig bei Divisionen und für den möglichen Wertebereich)
- Zeichen (`char`) werden wie Ganzzahlen vom Typ `short` behandelt (der *Unicode*-Wert wird verwendet)
- Die **Ganzzahlarithmetik ist „zirkulär“** und produziert nie einen Überlauf (eine Division durch Null hingegen generiert einen Ausnahmefehler *ArithmeticException*)
- Die **Dezimalstellen-Arithmetik** (Gleitkommaarithmetik) kann Spezialwerte generieren (*unendlich, -unendlich, negative Null, NaN*) **erzeugt aber nie einen Ausnahmefehler** (*Exception*)

Auswerten der Operationen [2]

- Das Resultat eines **Ganzzahlausdrucks** ist vom Typ `int`[1)] ausser wenn einer der Operanden vom Typ `long` ist. In diesem Fall ist der Ausdruck vom Typ `long`
- Das Resultat eines **Dezimalstellen-Ausdrucks** (beinhaltet mindestens einen Gleitkomma-Operand) ist vom Typ `float`[1)] ausser wenn einer der Operanden vom Typ `double` ist. In diesem Fall ist der Ausdruck vom Typ `double`

[1)] *durch automatische, erweiternde Operandenumwandlung*

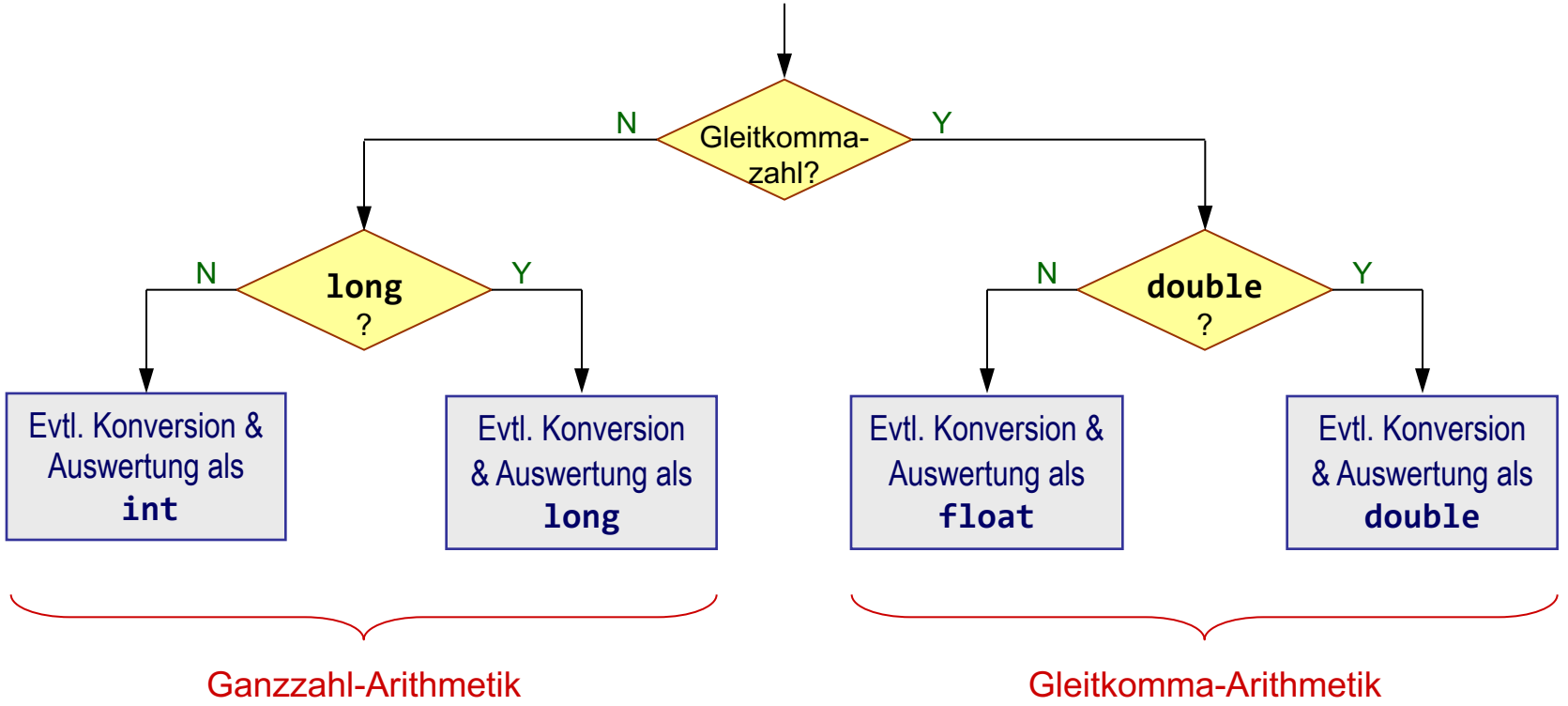
```
byte  b = 12;
short s = 167;
float f = 1.2f;

int i = s - b;          // Ganzzahliger Ausdruck, s und b werden
                        // vor der Operation in int  umgewandelt

double d = f * 2.5;     // Reeller Ausdruck, f wird vor der Operation
                        // in double umgewandelt
```

Auswertung arithmetischer Ausdrücke

- Für jeden arithmetischen Operator werden vorgängige Konversionen abhängig von den Operandentypen durchgeführt:



Beispiel : **17 / 3 + 1.0**

Vergleichsoperatoren

- Führen **Vergleichsoperationen** aus und geben boolesche Resultate zurück.

`==` `!=`	Gleichheit, Ungleichheit
`<` `>`	Kleiner, grösser
`<=` `>=`	Kleiner gleich, grösser gleich

```
i < 5
if (alter >= 18) {...}
while (i<100)  {...}
if (objekt1 != objekt2)  {...}    // Vergleich der Referenzen
```

Achtung : Bei Referenztypen (Objekte und Arrays) vergleichen die Gleichheits- und Ungleichheitsoperatoren die Referenzen und nicht die Inhalte der referenzierten Werte.

Operatoren und Gleitkommazahlen

- Die Gleitkommazahlen (`float` und `double`) sind lediglich **Näherungswerte von reellen Zahlen,** denn sie sind kodiert (mit Basis 2) mit einer limitierten Anzahl Bits für die Mantisse und den Exponenten (Darstellung gemäss IEEE 754, siehe z.B. www.binaryconvert.com oder babbage.cs.qc.edu/IEEE-754).
- Sogar innerhalb eines vorgesehenen Wertebereichs und mit einer Anzahl Dezimalstellen grösser als die maximale Präzision, kann eine grosse Anzahl reeller Zahlen nicht genau repräsentiert werden.
- Deshalb muss man bei der Verwendung von Vergleichsoperatoren im Zusammenhang mit Gleitkommazahlen sehr vorsichtig sein und die Rundungsproblematik berücksichtigen.

```
float a=1.0f, b=0.1f, c=0.2f;
(a+(b+c)) != ((a+b)+c)    //!!! true !!!
(a+(b+c)) <  ((a+b)+c)    //!!! true !!!

float d=2.0E7f;
d == (d+1)                //!!! true !!!
```

Logikoperatoren

- Führen **logische Operationen** (mit zwei booleschen Operanden) aus und geben boolesche Werte zurück.

!	Logisches NICHT
&	Logisches UND
&&	Bedingtes logisches UND (der 2te Operand wird nur ausgewertet, wenn der erste wahr ist)
|	Logisches ODER
||	Bedingtes logisches ODER (der 2te Operand wird nur ausgewertet, wenn der erste falsch ist)
^	Logisches EXCLUSIV-ODER

```
if (!found)  {...}
if (size>1.4 || age>12)  {...}
if (t!=null && t.length>=2 && t[1]!=4)  {...}
```

Bit-orientierte Operatoren

- Führen **bitweise Operationen** (mit zwei ganzzahligen Operanden) aus und geben Ganzzahlen (mit Vorzeichen) zurück.

`~`	Inversion (1er Komplement)
`&`	Bitweises UND
`|`	Bitweises ODER
`^`	Bitweises EXCLUSIV-ODER *nicht mit Potenzierung verwechseln*
`<<`	Linksverschiebung (Nullen rechts angehängt)
`>>`	Rechtsverschiebung mit Vorzeichen (Vorzeichenbit wird links hinzugefügt)
`>>>`	Rechtsverschiebung ohne Vorzeichen (Nullen links hinzugefügt)

```
byte b = ~12  //           ~00001100 => 11110011 (-13)
10 & 7        // 00001010 & 00000111 => 00000010 (  2)
10 | 7        // 00001010 | 00000111 => 00001111 ( 15)
10 ^ 7        // 00001010 ^ 00000111 => 00001101 ( 13)
10 << 2       //            00001010 => 00101000 ( 40)
10 >> 2       //            00001010 => 00000010 (  2)
-10 >> 2      //            11110110 => 11111101 ( -3)
-10 >>>2      //            11110110 => 00111101 ( 61)
```

Zuweisung

- Speichert im linken Operand (Variable) den Wert des rechten Operanden (Ausdrucks)
- Der Typ des linken Operanden muss mit dem Typ des rechten Ausdrucks kompatibel sein (wenn notwendig, automatische erweiternde Konversion)
- **Rückgabe-Typ und -Wert** des Ausdrucks entspricht dem Typ der Variable und deren Wert nach der Zuweisung
- **Achtung** : Den Zuweisungsoperator (=) nicht mit dem Gleichheitsoperator (==) verwechseln

  « ***übernimmt den Wert*** » und nicht « ***ist gleich wie*** »

```
a = b + c;          // Die Variable a bekommt als Wert die Summe (b+c)
a = b = c;          // Assoziativität rechts ⇒  a = (b = c)
t[i] = circle.center();
```

Zuweisung mit Operator

- Kombination der Zuweisung mit einem Operator (arithmetisch oder bitweise)
- Der Typ des linken Operanden (Variable) muss mit dem Typ des rechten Ausdrucks kompatibel sein

var **op=** *expr* ist dasselbe wie *var* **=** *var* **op** **(** *expr* **)**

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

```
i += 2;           // i = i + 2;
a *= z + 4;       // a = a * (z + 4);
flag |= mask;     // flag = flag | mask;
                  // setzt gewisse Bits entsprechend der Maske
```

Operator instanceof

- Ermöglich das Testen, ob ein Ausdruck (ein Referenztyp) kompatibel (bzw. wandelbar) zu einem gegebenen Typ ist.
- Gibt **`true`** zurück, wenn der linke Operand (welcher ein Ausdruck vom Typ Objekt oder Array sein muss) eine Instanz des Typs des Operanden auf der rechten Seite ist (welcher ein Referenztyp sein muss).

```
"Text" instanceof String;          // true
"Text" instanceof Object;          // true
null   instanceof String;          // false
new int[] {1} instanceof Object    // true
new int[] {1} instanceof byte[]    // false  Array von int ⇔ Array von byte

//  Nützlich vor Typumwandlungen
if (form instanceof Polygon) {
  Polygon p1 = (Polygon)form;
}
```

Spezielle Operatoren

- Folgende spezielle Operatoren werden manchmal als **syntaktische Elemente** bezeichnet, manchmal als **Operatoren** :

  - **Zugriff auf ein Element eines Objekts (.)**
    ```
    obj.x
    obj.f()
    ```
  - **Zugriff auf ein Element eines Arrays ([])**
    ```
    t[2]
    ```
  - **Methodenaufruf (())**
    ```
    rectangle.move(x, y)
    ```
  - **Erstellung eines Objektes (new)**
    ```
    new Point(4.0, -2.5);
    new ArrayList();
    ```
  - **Typenumwandlung (())**
    ```
    (float)position
    (int)(a*1.414f)
    (Circle)shape
    ```

Ergänzung