Wo sind wir?

- Java-Umgebung
- Lexikale Konventionen
- Datentypen
- Kontrollstrukturen
- **Ausdrücke**
- Klassen, Pakete, Schnittstellen
- JVM
- Exceptions
- Java Klassenbibliotheken
- Ein-/Ausgabe
- Collections
- Threads
- Applets, Sicherheit
- Grafik
- Beans
- Integrierte Entwicklungsumgebungen

Ausdruck

Aus Konstanten, Variablen und verbindenden Operatoren lassen sich Ausdrücke angeben, die von der JVM ausgewertet werden. Das Ergebnis der Auswertung ist ein Wert.

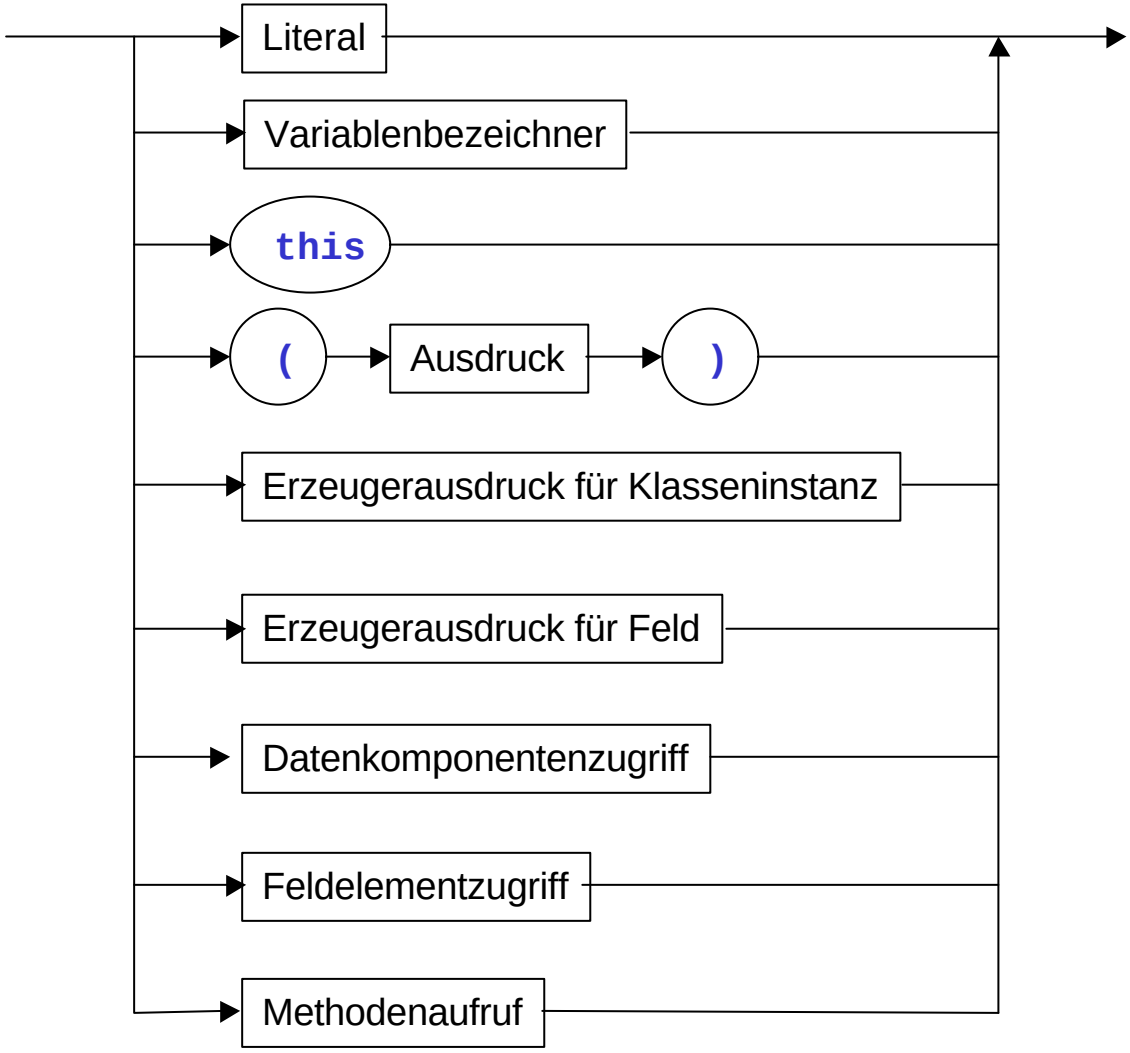
Unterscheidung:

Ausdruck:	Wert
Anweisung:	Kontrollfluss

Primäre Ausdrücke sind die "Grundbausteine" von Ausdrücken, mit Operatoren lassen sich daraus komplexere Ausdrücke erzeugen.

Übersicht primärer Ausdruck

Primärer Ausdruck



Primäre Ausdrücke (1)

- Literale
  schon behandelt (verschiedene Typen von Literalen)

  Beispiele:
  ```
  3   4.0     true    "Mein String"    null
  ```

- Variablenbezeichner
  Vorerst: Der Wert der Variablen wird genommen.
  Die genaue Semantik ist komplizierter und wird
  im Zusammenhang mit Klassen behandelt.

  Beispiel:
  ```
  int i, j=2, k=3;
  i = j + k;            // entspricht i = 2 + 3;
  ```

- Geklammerter Ausdruck
  Aus jedem Ausdruck entsteht durch Klammerung ein neuer Ausdruck.

  Beispiele:
  ```
  (3)          (4.0 + 3)
  ```

Primäre Ausdrücke (2)

- Schlüsselwort this
  Nur in Instanzenmethoden, Konstruktoren oder Initialisierern von Instanzenvariablen
  Der Wert dieses Ausdrucks ist eine Referenz auf das Bezugsobjekt

  Beispiel:

```
class Beispiel {
    int x;
    ...
    // Instanzenmethode (kein static)
    boolean equals(int y) {
        if (this.x == y)      // Datenkomponente x dieses Objektes
            return true;
        else
            return false;
    }
}
```

Primäre Ausdrücke (3)

- Erzeugerausdruck für Klasseninstanz

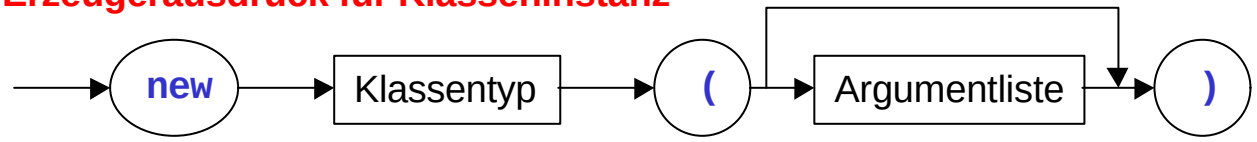
  Beispiele:

```
MeineKlasse a = new MeineKlasse();
MeineKlasse b = new MeineKlasse(1, 2, 3);
```

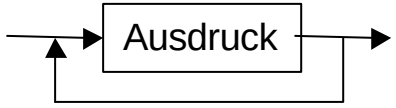
  Interne Schritte bei der Auswertung eines new-Operators:

  - Speicherplatz für Klasseninstanz besorgen
  - Speicherplaz für Datenfelder der Klasse und aller Oberklassen besorgen und ggf. initialisieren
  - Argumentliste des Konstruktors von links nach rechts auswerten
  - Konstruktor mit diesem Werten aufrufen

Erzeugerausdruck für Klasseninstanz



Argumentliste



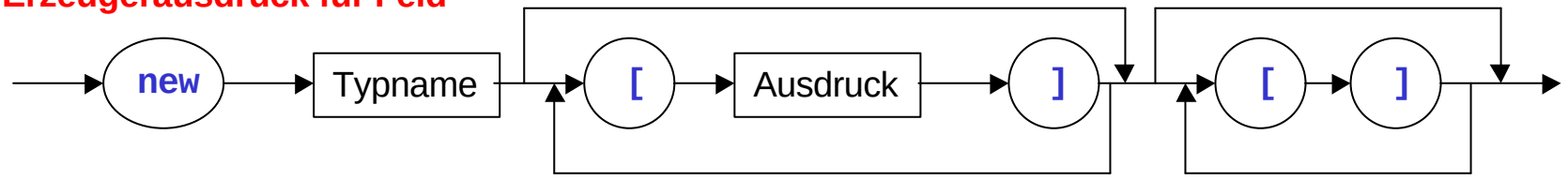
Primäre Ausdrücke (4)

- Erzeugerausdruck für Feld
  schon behandelt

  Beispiele:

```
int[] a = new int[4];
MeineKlasse[][][] b = new MeineKlasse[4][3][];
```

Erzeugerausdruck für Feld



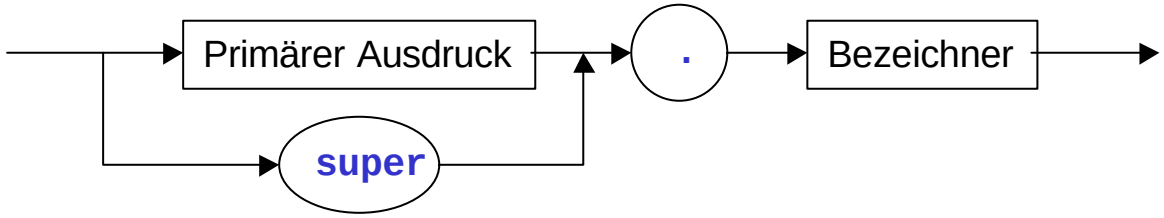
Primäre Ausdrücke (5)

- Datenkomponentenzugriff

  Beispiel:

```
class MeineKlasse {
    int x;
    MeineKlasse(int y) {
        this.x = y;
    }
}
```

Datenkomponentenzugriff



- Der primäre Ausdruck muss einen Referenzwert liefern.
- `super` darf nur dort verwendet werden, wo auch `this` stehen dürfte (siehe `this`).
- Im Zusammenhang mit Klassen genauer

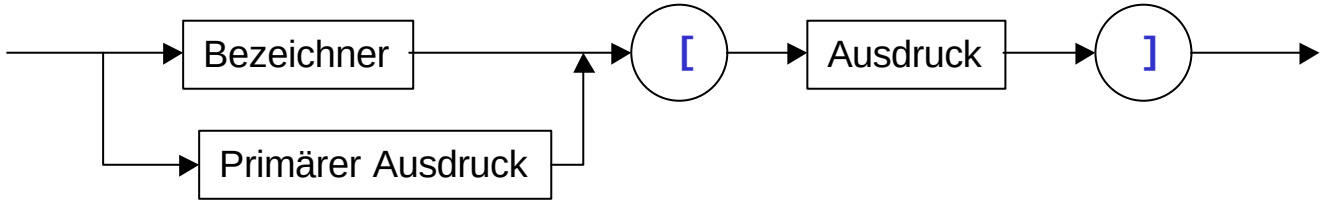
Primäre Ausdrücke (6)

- Feldelementzugriff

Beispiel:

```
int[] a = new int[4];
int[][] b = new int[2][3];
...
a[2] = a[3];
b[1][2] = b[0][1];
```

Feldelementzugriff



- Der Indexausdruck muss einen ganzzahligen Typen außer `long` haben
- Der Typ des Bezeichners/Ausdrucks vor `[` muss ein Feldtyp sein
- Der Wert des Ausdrucks muss im Indexraum enthalten sein
- Der primäre Ausdruck muss einen Referenzwert (auf ein Feld) liefern.

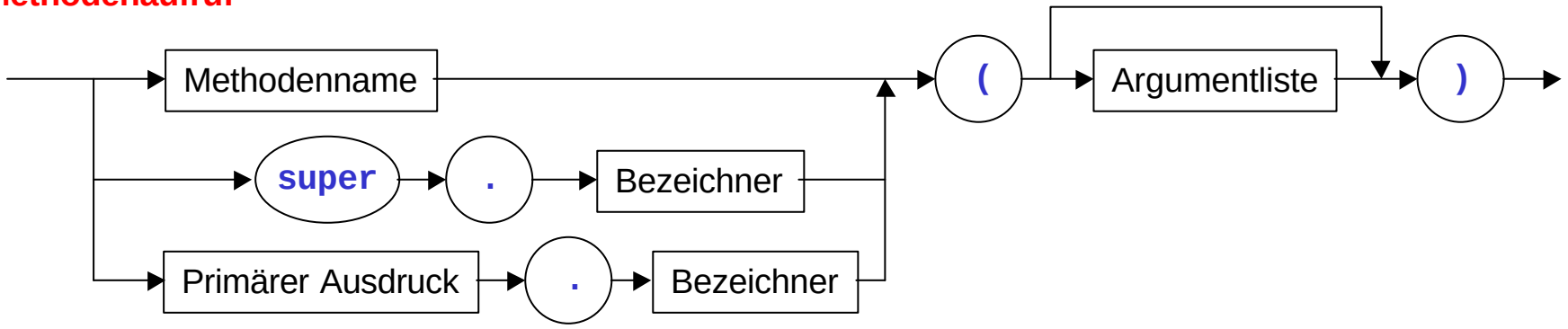
Primäre Ausdrücke (7)

- Methodenaufruf

Beispiele:

```
methodenname();
super.methode(4711, 31415);
feld[5].methode(4711);
```

Methodenaufruf



Semantik eines Methodeaufrufs

- Auswerten der aktuellen Argumente des Methodenaufrufs (von links nach rechts)
- Erzeugen neuer Variablen mit dem Namen und Typ der formalen Parameter
- Initialisieren der neuen Variablen mit den Werten der aktuellen Argumente
- Ausführen des Rumpfs der Methode
- Zurückgabe eines Ergebniswertes (falls vereinbart)

Die Übergabe der aktuellen Argumente erfolgt nach der call-by-value-Strategie. Auch bei Referenztypen erfolgt die Übergabe call-by-value (!), jedoch können durch den indirekten Zugriff in der Methode die Werte des Originalobjekts verändert werden.

Zusammengesetze Ausdrücke

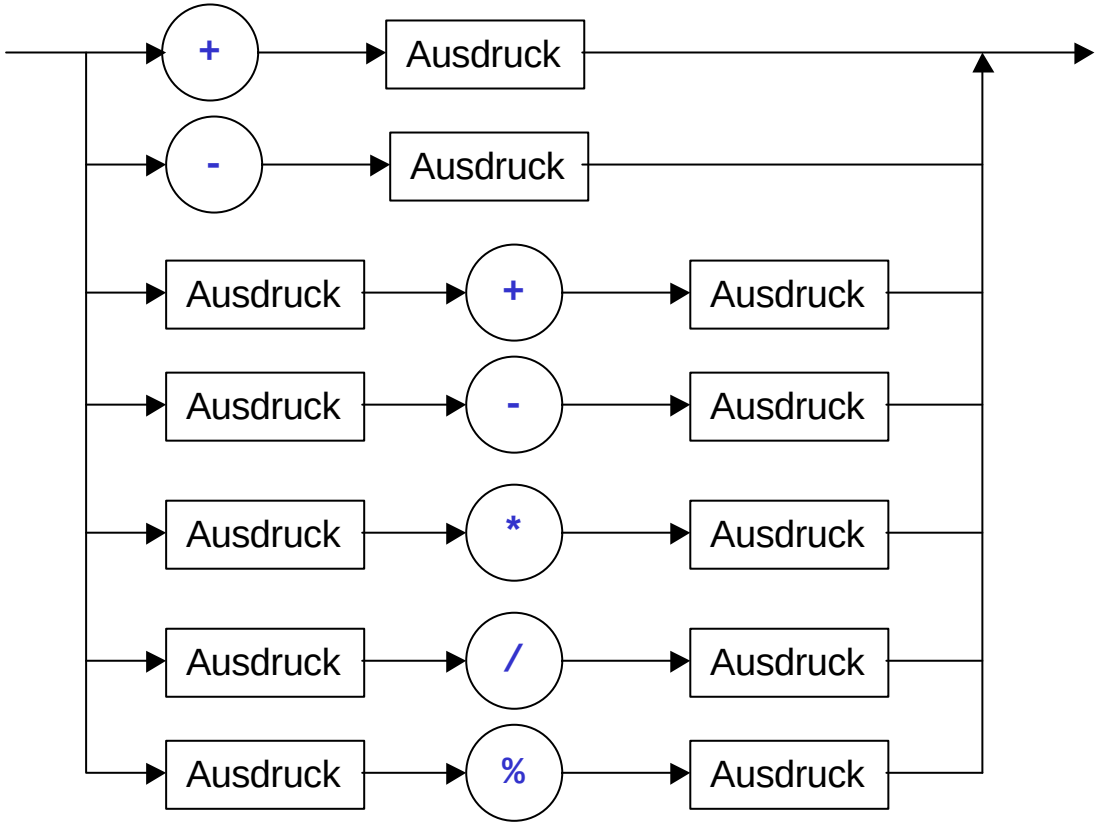
Aufbauend auf den primären Ausdrücken als Grundbaustein werden beliebige Ausdrücke (rekursiv) durch die Anwendung von Operatoren definiert.

Man unterscheidet nach der Stelligkeit (Anzahl Argumente) der Operatoren:

- Unäre Operatoren (einstellig). Beispiel: -5
- Binäre Operatoren (zweistellig). Beispiel: 3 + 4
- Ternäre Operatoren (dreistellig). Beispiel: (3 < 4) ? 5 : 6

Arithmetische Operatoren

Arithmetischer Ausdruck



Anmerkung: Wenn Sie in Büchern Grammatiken zu Java sehen, so sind dort eventuell Ausdrücke komplizierter angegeben. Dies geschieht dann deshalb, um Prioritäten der Operatoren in der Grammatik ausdrücken zu können.

Arithmetische Operatoren (1)

Unäres Minus, unäres Plus:
Auswertung wie gewohnt ; verfügbar für alle arithmetischen Typen

Beispiele:

```
-3          = -3
+7.5        = +7.5
```

Binäre Addition (+), Subtraktion (-), Multiplikation (*):
Auswertung wie gewohnt; verfügbar für alle arithmetischen Typen.
Keine Überprüfung auf Überlauf (alle numerischen Operationen)!

Beispiele:

```
3 + 4       = 7
7.0 * 3.5   = 24.5
```

Arithmetische Operatoren (2)

Division (/):

Wenn einer der beiden Operanden eine Fließkommawert ist, so wird die Fließkommadivision ausgeführt, ansonsten die ganzzahlige Division.

Ganzzahlige Division:

Rest wird weggelassen, Division durch 0 liefert ArithmeticException

Beispiele:

`9 / 4`	= 2
`-9 / 4`	= -2

Fließkommandivision:

Mit Rest, Division durch 0 liefert Unendlich oder NaN

Beispiele:

`9.0 / 4.0`	= 2.25
`7.0 / 0.0`	= +∞
`0.0 / 0.0`	= NaN

Arithmetische Operatoren (3)

Modulobildung (%):

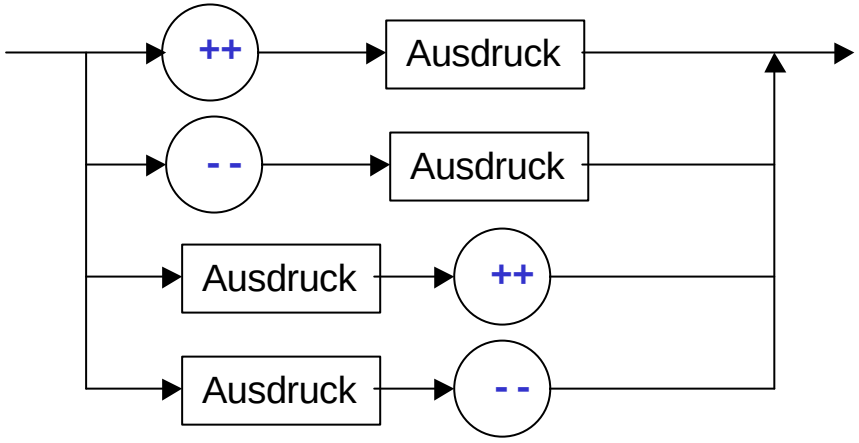
Rest bei Division. Das Vorzeichen ist gleich dem Vorzeichen des ersten Operanden. Anders als in anderen Programmiersprachen ist die Modulobildung auch für Fließkommawerte definiert.

Beispiele:

```
9 % 4          = 1
-9 % 4         = -1
4.3 % 2.1      = 0.1
```

Inkrement-/Dekrement-Operatoren

Inkrement/Dekrement-Ausdruck



Der Ausdruck muss eine Variable eines numerischen Typs bezeichnen.
Der Ausdruck wird ausgerechnet, indem der Wert der Variablen genommen wird, dieser um 1 inkrementiert (++) bzw. dekrementiert (--) wird und dieser neue Wert in der Variablen abgespeichert wird (Seiteneffekt; später mehr).
Das Ergebnis des Ausdrucks ist der neue Wert der Variablen bei den Formen ++a und --a und ist der alte Wert der Variablen bei den Formen a++ bzw. a--.

Häufige Verwendung:

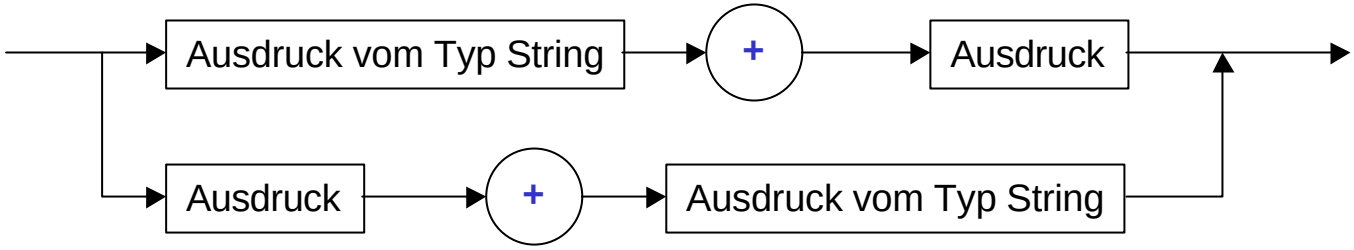
```
for(int i=0; i < 100; i++) ...
```

Beispiele

```
int i, j, k;

i = 0;
++i;              // i wird um 1 erhöht, d.h. neuer Wert von i ist 1.
                  // Der Wert des Ausdrucks wird ignoriert.

i = 0;
j = ++i;          // i um 1 erhöht, der neue Wert (1) genommen und j zugewiesen

i = 0;
j = i++;          // i um 1 erhöht, der alte Wert (0) genommen und j zugewiesen

i = 0;
j = 0;
k = ++i + j++;    // i und j haben anschließend den Wert 1. k hat den Wert 1.

i = 0;
j = 0;
k = ++i + ++j;    // i und j haben anschließend den Wert 1. k hat den Wert 2.
```

String-Operatoren

String-Ausdruck



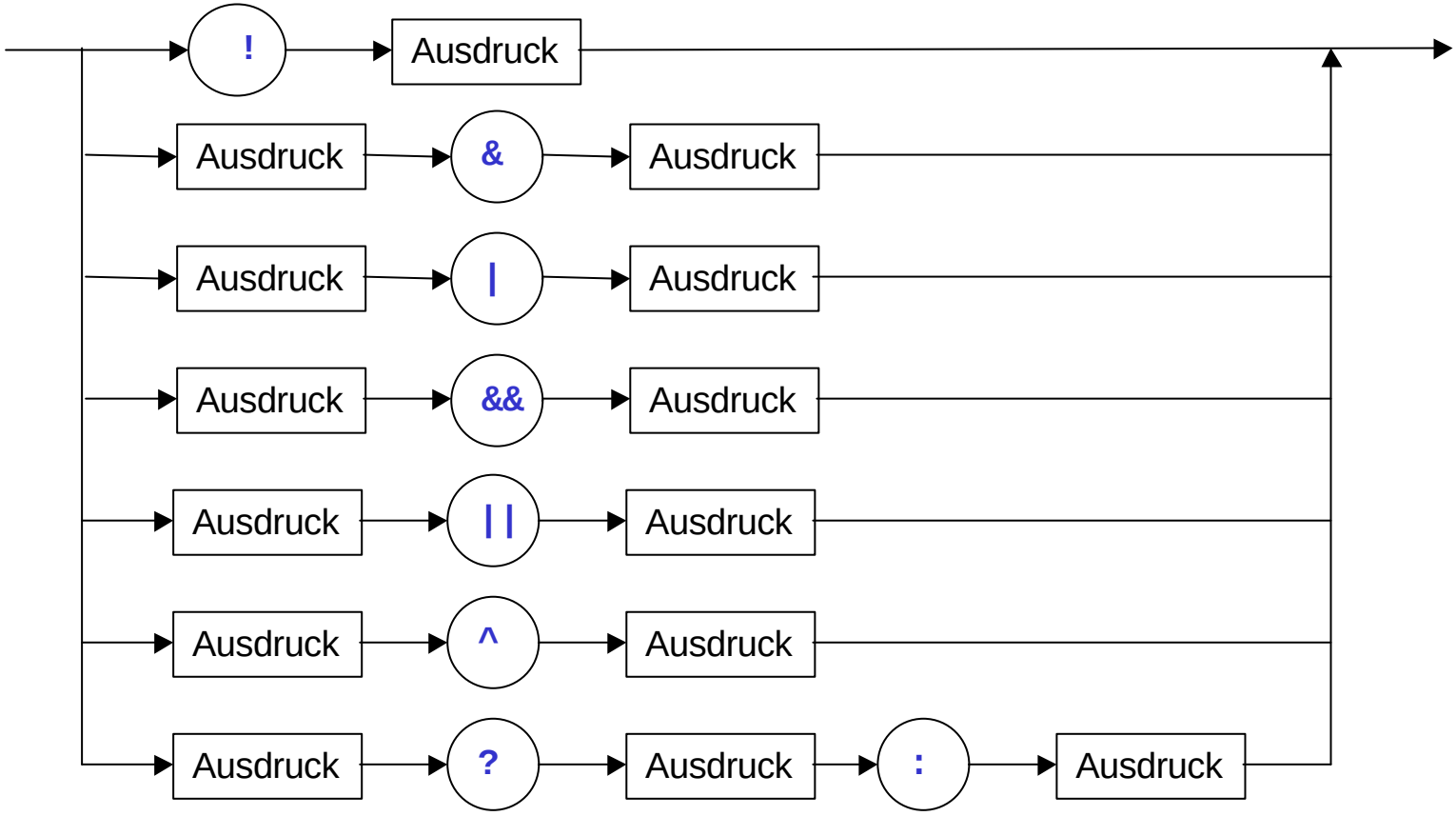
Ist mindestens einer der beiden Operanden vom Typ String, so bedeutet der +-Operator eine Verknüpfung von String (Konkatenation).
Ist einer beiden Ausdrücke vom Typ T ≠ String, so wird dieser zuerst in einen String umgewandelt. Dazu wird intern die Methode `toString()` zur Laufzeit aufgerufen, die für jedes Objekt (inkl. aller Primitivwerte) existiert.
`toString()` ist eine Methode der Klasse `Object` und wird somit an alle Klassen vererbt, kann aber in einer Klasse überschrieben werden.

Beispiele:

```
"Aber " + "Hallo"          // ergibt "Aber Hallo"
"Jetzt schägts " + 13      // ergibt "Jetzt schlägts 13"

MeineKlasse o = new MeineKlasse();
"Mein Name ist " + o       // ergibt "Mein Name ist xxx" mit xxx=String
```

Boolsche Operatoren

Boolscher Ausdruck



Alle Argumente der boolschen Operatoren bis auf ? : müssen vom Typ `boolean` sein.

Boolsche Operatoren

- !e
  Logische Negation des Wertes von e (`true` → `false`, `false` → `true`)

- e1 & e2
  Logische Und-Verknüpfung der beiden Werte

- e1 | e2
  Logische Oder-Verknüpfung der beiden Werte

- e1 ^ e2
  Das Ergebnis ist `true`, falls beide Operandenwerte verschieden sind.
  Ansonsten ist das Ergebnis `false`.

Boolsche Operatoren

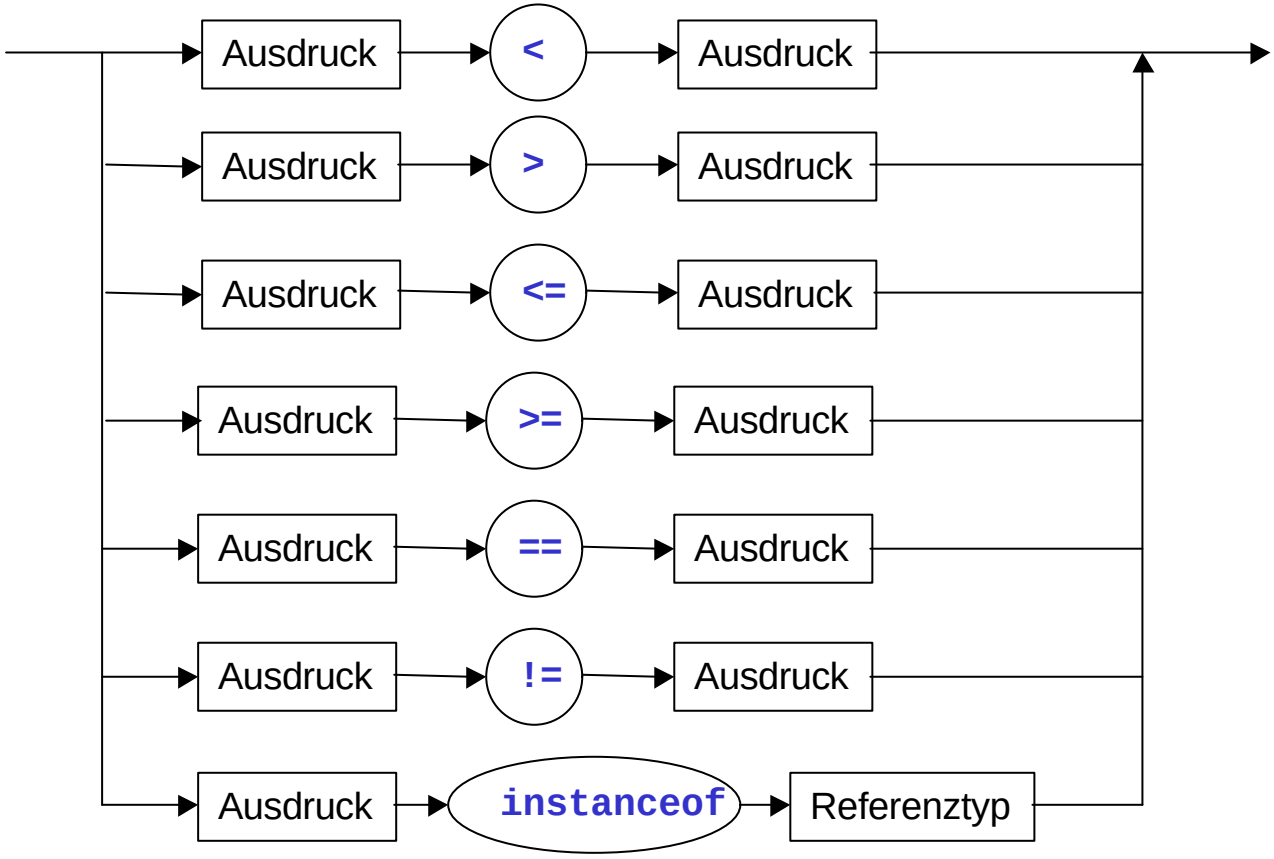
- e1 && e2
  Der Ausdruck e1 wird ausgewertet. Falls der Wert gleich `false`, so wird e2 nicht mehr ausgewertet und der Wert des gesamten Ausdrucks ist `false`. Ansonsten wird e2 noch ausgewertet und die beiden Werte mit Und verknüpft.
- e1 || e2
  Der Ausdruck e1 wird ausgewertet. Falls der Wert gleich `true`, so wird e2 nicht mehr ausgewertet und der Wert des gesamten Ausdrucks ist `true`. Ansonsten wird e2 noch ausgewertet und die beiden Werte mit Oder verknüpft.
- e1 ? e2 ? e3
  e1 muss ein Ausdruck vom Typ boolean sein. e2 und e3 müssen entweder beide boolean, beide numerisch oder beide einen Referenztyp haben.
  Zuerst wird e1 ausgewertet. Ist der Ergebniswert `true`, so wird e2 ausgewertet und der Wert des Gesamtausdrucks ist der Wert von e2. Ansonsten (e1 ist false) wird e3 ausgewertet und der Wert des Gesamtausdrucks ist der Wert von e3.
  Dieser Ausdruckstyp entspricht einem if-the-else in Ausdrucksform, d.h. ein Wert wird erzeugt, den man weiterverwenden kann.

Beispiele

```
boolean b, b1, b2;
int i;

b1 = true;
b2 = !b1;                    // false

b = b1 & b2;                 // true & false = false
b = b1 | b2;                 // true | false = true
b = b1 ^ b2;                 // true ^ false = true

b = b2 && methode(10000); // weil b2 false ist, wird methode(10000)
                             // nicht mehr ausgewertet
b = b1 || methode(10000); // weil b1 false ist, wird methode(10000)
                             // nicht ausgewertet

i = (b1 & b2) ? 3 : 4;    // (true & false) ? 3 : 4 = false ? 3 : 4 = 4
```

Vergleichsoperatoren

Vergleichsausdruck



Der (Ergebnis-) Typ eines Vergleichsausdrucks ist immer vom Typ `boolean`.

Vergleichsoperatoren

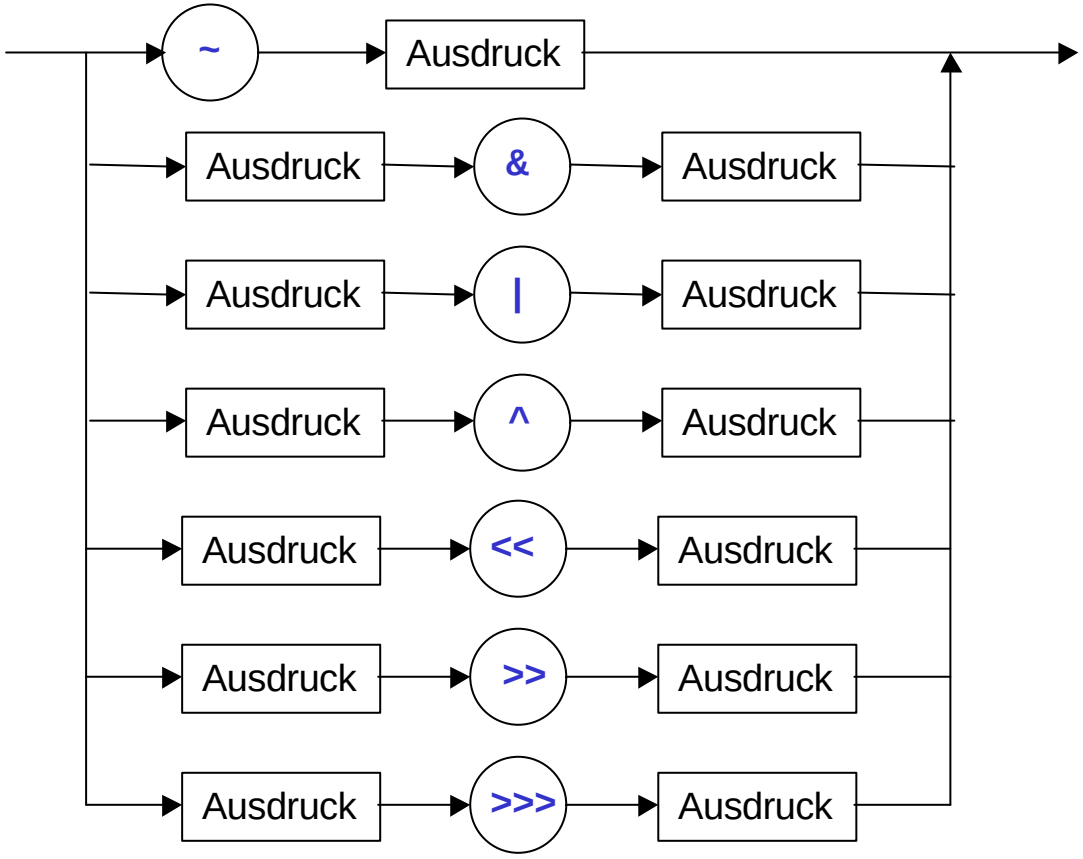
- Die Vergleichsoperatoren `<, <=, >, >=` sind nur für numerische Werte definiert.
- Die Vergleichsoperatoren `==` (gleich) und `!=` (ungleich) vergleichen entweder zwei numerische Werte, zwei boolsche Werte oder zwei Referenzwerte (bzw. `null`). Bei Referenzen wird der Zeiger verglichen, nicht der Inhalt! D.h. zwei Objekte können als ungleich gelten (verschiedene Zeiger), obwohl ihre Inhalte gleich sind.
- `instanceof`-Operator später

Beispiele

```
boolean b, b1=true, b2 = false;
MeineKlasse o1 = new MeineKlasse();
MeineKlasse o2 = o1;

b = 1 < 4;                  // true
b = 4.0 > 3;                // true; hier findet vorher eine Typumwandlung statt

b = 4.0 == 4;               // true; hier findet vorher eine Typumwandlung statt
b = b1 != b2;               // true
b = o1 == o2;               // true
```

Bit-Operatoren

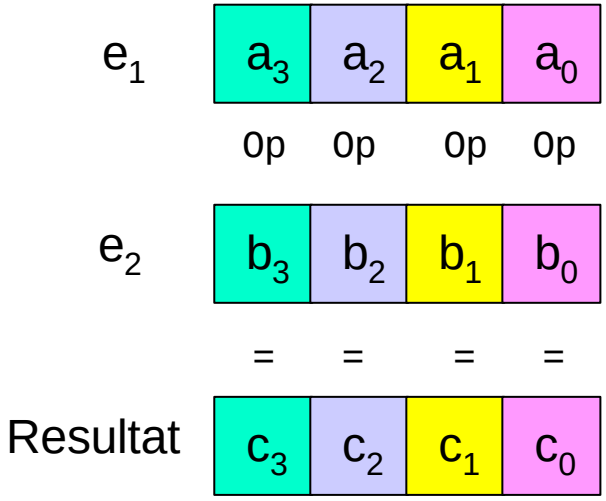
Bit-Ausdruck



Die Argumente der Bit-Operatoren müssen von einem ganzzahligen Typen sein.

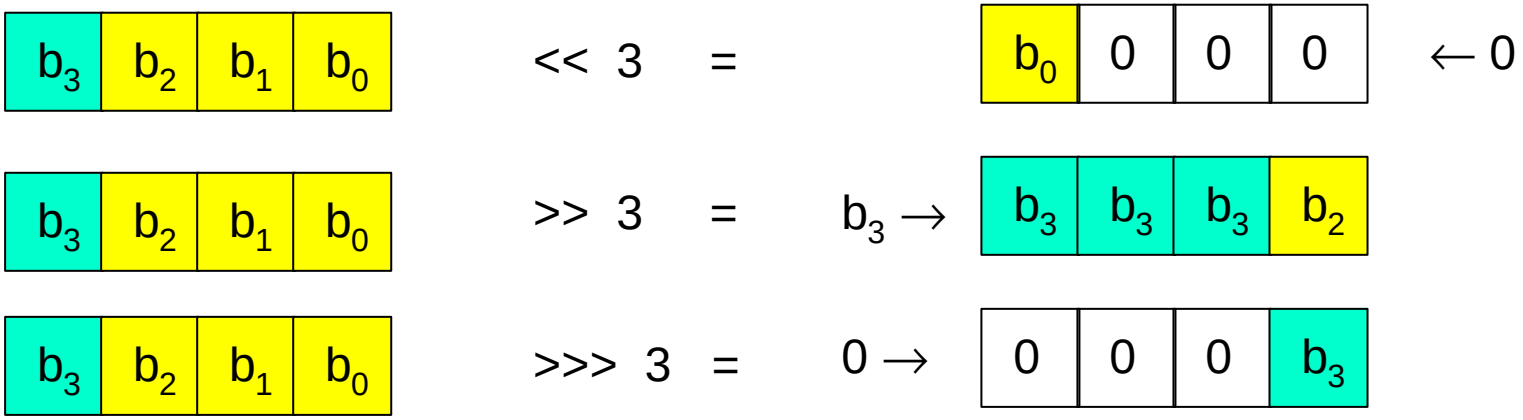
Bit-Operatoren (1)

- ~e
  Bitweise Komplementärbildung (0→1, 1→0)
- e_1 & e_2
  Bitweise Und-Verknüpfung (0&0=0, 0&1=0, 1&0=0, 1&1=1)
- e_1 | e_2
  Bitweise Oder-Verknüpfung (0&0=0, 0&1=1, 1&0=1, 1&1=1)
- e_1 ^ e_2
  Bitweise Exklusiv-Oder-Verknüpfung (0&0=0, 0&1=1, 1&0=1, 1&1=0)



Bit-Operatoren (1)

- $e_1 << e_2$
  Bitweises Verschieben des linken Wertes um e_2 Positionen nach links (0 wird nachgeschoben). Dies entspricht einer Multiplikation mit 2^{e2}.
- $e_1 >> e_2$
  Bitweises Verschieben des linken Wertes um e_2 Positionen nach rechts (Vorzeichenbit wird nachgeschoben). Dies entspricht einer Division durch 2^{e2}.
- $e_1 >>> e_2$
  Bitweises Verschieben des linken Wertes um e_2 Positionen nach rechts (0 wird nachgeschoben). Für positive Werte von e_1 ist dies gleich $e_1 >> e_2$.



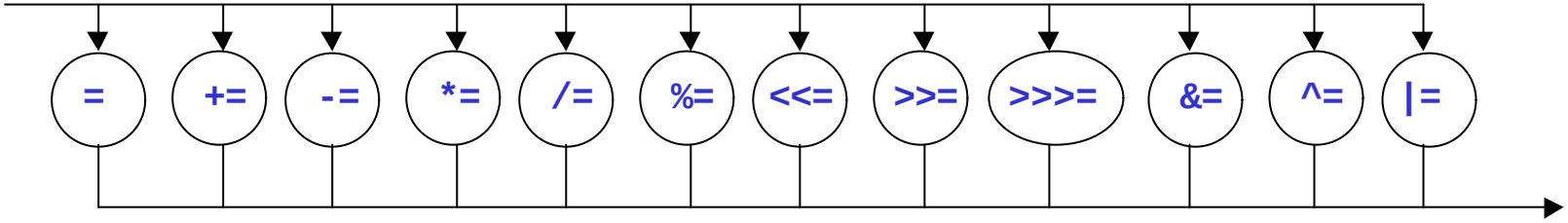
Beispiele

```
byte b, b1, b2;
int i1, i2, i3;

b = 1;            // = 00000001
b = ~b;           // ~00000001 = 11111110

b1 = 3;           // = 00000011
b2 = 5;           // = 00000101
b = b1 & b2;      // = 00000001 = 1(10)
b = b1 | b2;      // = 00000111 = 7(10)
b = b1 ^ b2;      // = 00000110 = 6(10)

i1 = 0x80000001; // = 10000....00001
i2 = 2;
i = i1 << i2;     // = 0000...000100
i = i1 >> i2;     // = 1110000...000
i = i1 >>> i2;    // = 0010000...000

i1 = 0x40000001; // = 010000....00001
i2 = 2;
i = i1 >> i2;     // = 0010000...000
```

Zuweisungsoperatoren

Zuweisungsausdruck

Zuweisungsoperator



Ein L-Wert ist ein Ausdruck, der auf der linken Seite einer Zuweisung stehen kann, d.h. einen Speicherplatz bezeichnet.

Beispiele:

```
int a;
int[] b = new int[10];
// L-Werte:
a = 1;
b[3] = 5;
```

Zuweisungsoperatoren

Ein Ausdruck der Form

L-Wert Zuweisungsoperator Ausdruck

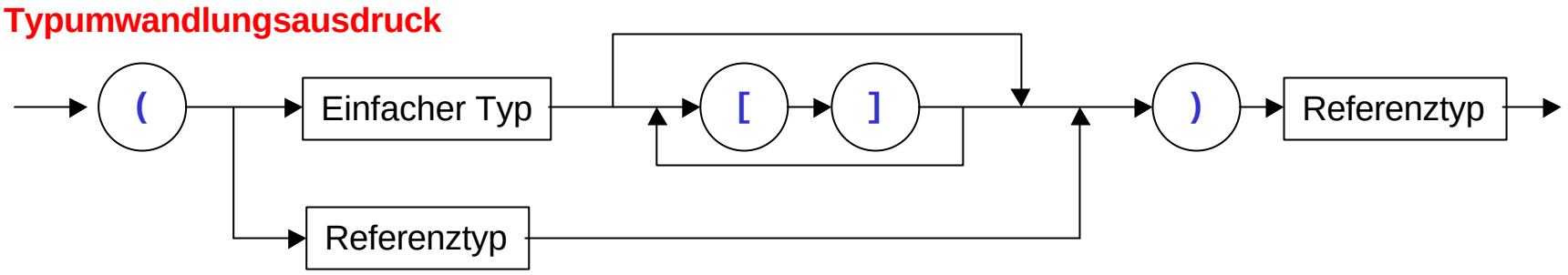
ist eine Kurzform für

L-Wert = L-Wert Operator Ausdruck

Beispiele:

```
int i = 0;

i += 3;          // anschließend i = 3
i *= 5;          // anschließend i=15
i &= 2;          // anschließend i=1
```

Typumwandlungsausdruck



Eine Typumwandlung nennt man auch cast (oder cast-Operation).
Mit solch einem Ausdruck erzwingt man explizit die Typumwandlung (wenn möglich) des Ausdrucks in den Wunschtyp (später mehr zu Typumwandlungen).

Beispiele:

```
int i = 1;
float f;

f = (float) i;
i = (int) f;
f = (float) (long) (byte) 3.0;
```

Seiteneffekte

Bei der Auswertung eines Ausdrucks gewinnt man einen Wert. Ein Operator kann aber auch Seiteneffekte verursachen.

Beispiel 1:

```
int a = 2;
int b = ++a + a;
```

Der ++-Operator hat als Seiteneffekt, dass der Wert der Variablen a erhöht wird.

Beispiel 2:

```
class MeineKlasse1 { ... }
class MeineKlasse2 { ... }

MeineKlasse1 a = new MeineKlasse1();
MeineKlasse2 b = new MeineKlasse2();

// Hier kann eventuell der Zustand von b verändert werden
a.methode(b);
```

Vorrang, Assoziativität, Auswertereihenfolge

Vorrang von Operatoren:

Beispiel Punkt-vor-Strichrechnung: `a + b * c`
Der *-Operator hat höhrere Priorität als der +-Operator.

Assoziativität:

Fall: Ein Ausdruck hat mehrere Operatoren mit gleicher Priorität.
Die meisten Operatoren haben eine links-nach-rechts-Assoziativität. Lediglich die Zuweisungsoperatoren und Zuweisungsoperatoren haben eine Assoziativität von rechts nach links.
Beispiel 1: `a + b + c` (links nach rechts, d.h. `(a + b) + c`)
Beispiel 2: `a = b = c` (rechts nach links, d.h. `a = (b = c))`

Die Reihenfolge der Anwendung von Operatoren kann durch Klammerung von Ausdrücken überschrieben werden: `(a + b) * c`

Auswertereihenfolge:

```
a + methode(x, y)
```

Was wird zuerst ausgewertet? Seiteneffekte!

Priorität und Assoziativität der Operatoren (1)

Priorität	Assoziativität	Operator	Bemerkung
15	links	`.  [ ]`	Datenkomponente, Feldelement
		`(args)`	Methodenaufruf
		`++  --`	Postinkrement/-Dekr.
14	rechts	`++  --`	Präinkement/-Dekr.
		`+  -`	Unäres +,-
		`~`	Bitweises Komplement
		`!`	Boolsches Nicht
13	rechts	`new`	Objekterzeugung
		`( type )`	Typanpassung (cast)
12	links	`*  /  %`	Multiplikative Operatoren
11	links	`+  -`	Additive Operatoren
		`+`	String-Konkatenation
10	links	`<<  >>  >>>`	Shift-Operatoren

Priorität und Assoziativität der Operatoren (2)

Priorität	Assoziativität	Operator	Bemerkung
9	links	`<  <=  >  >=`	Vergleichsoperatoren
		`instanceof`	Typvergleich
8	links	`==  !=`	Vergleichsoperatoren
7	links	`&`	Bitweises oder boolsches Und
6	links	`^`	Bitweises oder boolsches XOR
5	links	`|`	Bitweises oder boolsches Oder
4	links	`&&`	Bedingtes Und
3	links	`||`	Bedingtes Oder
2	rechts	`?:`	Konditionaler Operator
1	rechts	`=  *=  /=  %=  +=  -=`	Zuweisungen mit Operatoren
		`<<=  >>=  >>>=`	
		`&=  ^=  |=`	

„Problemfälle“ bei Priorität

Die Priorität entspricht im Wesentlichen der „üblichen“ Vorstellung von Operatorpriorität. Ausnahmen sind:

- Cast kombiniert mit Objektelementzugriff
```
((Integer) o).intValue();
```
- Zuweisung kombiniert mit Vergleich
```
while((line = in.readLine()) != null) ...
```
- Bitweise Operatoren kombiniert mit Vergleich
```
if((flags1 &flags2) != 0) ...
```

Auswertereihenfolge

In Java gibt es eine klar definierte Reihenfolge in der Auswertung von Ausdrücken:

- Auswertung von links nach rechts bei binären Operatoren
- Auswertung aller Operanden vor Anwendung eines Operators (Ausnahme: `&&, ||, ?:`)
- Auswertung aller Argumente eines Methodenaufrufs von links nach rechts, bevor die Methode angewendet wird
- Berücksichtigung von Klammerung sowie Vorrang und Assoziativität von Operatoren

Beispiele

- Auswertung von links nach rechts

```
int a = 2;
int b = ++a + a;
```

  Auswertung: 3 + 3

- Auswertung aller Operanden

```
int a = 2;
int b = ++a + ++a * ++a;
```

  Auswertung: Zuerst alle Operanden auswerten, dann erst die Multiplikation vor der Addition ausführen. D.h.: 3 + 4 * 5

- Auswertung aller Argumente

```
int a = 2;
double x = sqrt(++a, ++a);
```

  Aufruf: `sqrt(3, 4)`

- Klammerung sowie Vorrang und Assoziativität
  siehe 2.

Konstante Ausdrücke

An manchen Stellen ist ein konstanter Ausdruck notwendig. Ein konstanter Ausdruck ist ein Ausdruck, dessen Wert von einem einfachen Typ oder String-Typ ist und dessen Wert zur Übersetzungszeit vom Compiler berechnet werden kann.

Konstante Ausdrücke werden konstruiert aus:

- Literale einfacher Typen und Strings
- Cast-Operationen in einfachen Typ oder String
- Unäre Operatoren `+ - ~ !`
- Arithmetische Operatoren `+ - * / %`
- Shift-Operatoren `<< >> >>>`
- Vergleichoperatoren `< <= > >= == !=`
- Bitoperatoren und logische Operatoren `& ^ |`
- Bedingte Operatoren `&& ||`
- Bedingungsoperator `?:`
- Namen final deklarierter Variablen mit konstanten Initialisierungsausdrücken (sowohl einfache Namen als auch qualifizierte Namen der Form Typname.Bezeichner)

Beispiele

```
true

3.0 + (float)2

(short) (1*2*3*4*5)

Integer.MAX_VALUE / 2

2.0 * Math.PI

"Die Integerzahl " + Long.MAX_VALUE + " ist ganz schön groß."
```

Typumwandlung

Was passiert bei der Auswertung von 3 + 4.0?

Weder eine Ganzzahladdition noch eine Fließkommaaddition kann durchgeführt werden, da unterschiedliche Datenformate vorliegen (Zweierkomplementdarstellung für Ganzzahlen bzw. IEEE-754-Darstellung für Fließkommazahlen).

Lösung:
Umwandlung eines Operanden, so dass beide Operanden vom gleichen Typ sind, d.h. entweder

	3 + 4	(Ganzzahladdition)
oder		
	3.0 + 4.0	(Fließkommaaddition)

Typumwandlung

Typumwandlungen können auf zweierlei Weise stattfinden:

- Implizite Typumwandlungen nimmt der Compiler automatisch vor
  Beispiel: 3 + 4.0

- Explizite Umwandlungen werden über den cast-Operator durch den Programmierer "erzwungen".
  Beispiel: 3 + (int)4.0

Typumwandlungen sind in allen Programmiersprachen ein schwieriges Thema, weil eine Reihe von Problemen dabei auftreten können. Man unterscheidet zwischen erweiternden Umwandlungen (weniger ein Problem) und einengenden Umwandlungen (kann zu massiven Problemen führen).

Zuerst wird vorgestellt, welche Kategorien von Typumwandlungen prinzipiell möglich sind. Anschließend wird erläutert, in welchen Fällen (z.B. in einer Zuweisung, in einem Ausdruck usw.) die einzelnen Kategorien erlaubt sind.

Auf Umwandlungen mit Referenztypen wird später näher eingegangen.

Prinzipielle Probleme bei Typumwandlungen

- Umwandlung int nach long (Erweiternde Umwandlung einfacher Typen)
  Kein Problem, da alle Werte des Datentyps int in long darstellbar sind

- Umwandlung long nach int (Einengende Umwandlung einfacher Typen)
  Problem: Was soll mit den Werten gemacht werden, die nicht in int darstellbar sind?

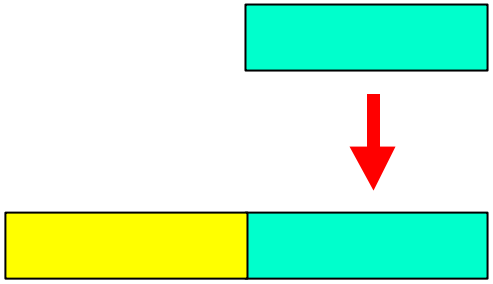
- Umwandlung von long nach float
  Problem: long-Wert eventuell nicht exakt darstellbar in float (Genauigkeitsverlust)

- Umwandlung eines Wertes vom Typ Object zum Typ MeineKlasse
  Problem: später mehr.

Erweiternde Umwandlung einfacher Typen

Folgende Umwandlungen sind erweiternde Umwandlungen:

- byte nach short, int, long, float, double
- short nach int, long, float, double
- char nach int, long, float, double
- int nach long, float, double
- long nach float, double
- float nach double



Bei einer der obigen Umwandlungen zwischen ganzzahligen Typen findet kein Informationsverlust statt, ebenso nicht bei float nach double.
Bei einer Umwandlung von int oder long nach float oder long nach double kann es zu einem Genauigkeitsverlust führen.

Beispiel:

```
int big = 1234567890;
float approx = big;
System.out.println(big – (int)approx);
```

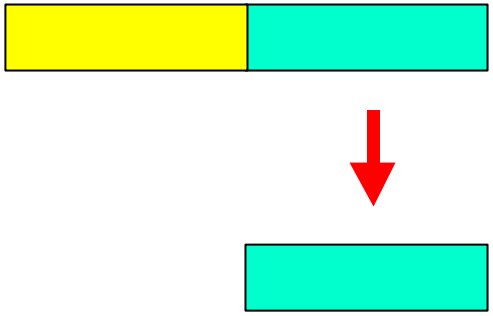
Ausgabe:

```
-46
```

Einengende Umwandlung einfacher Typen (1)

Folgende Umwandlungen sind einengende Umwandlungen:

- byte nach char
- short nach byte, char
- char nach byte, short
- int nach byte, short, char
- long nach byte, short, char, int
- float nach byte, short, char, int, long
- double nach byte, short, char, int, long, float



Einengende Umwandlungen können zu Informations- und Genauigkeitsverlust führen. Mit unserem "normalen" Verständnis von Zahlen kann also ein falscher Wert entstehen!

Beispiel:

```
byte b = (byte)1234567890;
```

Einengende Umwandlung einfacher Typen (2)

- Umwandlung eines ganzzahligen Wertes in einen kleineren ganzzahligen Typen (Beispiel: int nach short):
  Die niederwertigsten n Bits werden genommen, der Rest weggelassen (abgeschnitten). Dabei kann es sogar zu einem Vorzeichenwechsel kommen!

  Beispiel:

```
int i1 = 0x100;       // = 0000 ... 0001 0000 0000 = 256
byte b1 = (byte)i1;   // =               0000 0000 = 0
int i2 = 0x8001;      // = 1000 0000 ... 0000 0001 = -32769
byte b2 = (byte)i2;   // =               0000 0001 = 1
```

- Umwandlung char in einen ganzzahligen Typ:
  Die niederwertigsten n Bits (der Unicode-Codierung) werden genommen, der Rest weggelassen. Dabei kann auch eine negative Zahl entstehen.

  Beispiel:

```
byte i = (byte) 'a';
```

Einengende Umwandlung einfacher Typen (3)

Umwandlung eines Fließkommawertes in einen ganzzahligen Typen T in zwei Schritten:

- Umwandlung von Fließkomma nach long (T=long) oder Fließkomma nach int (T=byte, short, int char):
  - Ist Fließkommazahl=NaN, so ist das Ergebnis 0l bzw. 0
  - Andernfalls findet eine Rundung gegen 0 statt. Ist der Wert in long bzw. int darstellbar, ist dieser Schritt fertig.
    Ansonsten ist entweder der Wert zu klein oder zu groß für long bzw. int. Dann wird der kleinste bzw. größte darstellbar Wert von long bzw. int genommen.

- Umwandlung innerhalb ganzzahliger Typen
  Ist T=int oder long, so ist man fertig. Ansonsten (T=byte, short, char) findet eine weitere einengende Umwandlung statt (int nach T), wie sie bereits beschrieben wurde.

  Beispiel:

```
int i = (int)3.0;
```

Einengende Umwandlung einfacher Typen (4)

Umwandlung von double nach float:

Zuerst Rundung nach IEEE-754. Dann:

- Ist der Wert zu klein, um in float dargestellt zu werden, ist das Ergebnis +0.0f bzw. –0.0f.
- Ist der Wert zu groß, um in float dargestellt zu werden, ist das Ergebnis +∞ bzw. -∞.
- NaN wird zu NaN.

Beispiel:

```
float f = 3.1;
```

Übersicht über Typumwandlungen bei einfachen Typen

nach ® ¯ von	boolean	byte	short	char	int	long	float	double
boolean								
byte			>	<	>	>	>	>
short		<		<	>	>	>	>
char		<	<		>	>	>	>
int		<	<	<		>	>*	>
long		<	<	<	<		>*	>*
float		<	<	<	<	<		>
double		<	<	<	<	<	<	

< einengende Umwandlung (cast notwendig)

> weitende Umwandlung (automatisch)

>* weitende Umwandlung, evtl. Verlust von unteren Bits (automatisch)

Kategorien von Umwandlungen

In Java werden 5 Kategorien unterschieden, in welchem Zusammenhang Typumwandlungen stattfinden können (alle bis auf explizite Umwandlung finden implizit statt):

- Umwandlung in Zuweisungen
  Beispiel: `int i = 3.0;`
- Umwandlung in Methodenaufrufen
  Beispiel:
  ```
  static void meineMethode(double x) {...}
  public static void main(String[] args) { meineMethode(3); }
  ```
- Explizite Umwandlung
  Beispiel: `int i = (int)3.0;`
- Zeichenkettenumwandlung (schon behandelt)
  Beispiel: `System.out.println("Nummer " + i);`
- Anpassung numerischer Typen
  Beispiel: `int i = 3 + 4.0;`

Umwandlung in Zuweisungen

Folgende Fälle sind erlaubt:

- Erweiternde Umwandlung einfacher Typen
- Erweiternde Umwandlung von Referenztypen (später)
- Einengende Umwandlung einfacher Typen, wenn alle drei Bedingungen erfüllt sind:
  - Der Ausdruck ist ein konstanter Ausdruck vom Typ int.
  - Die Variable ist vom Typ byte, short oder char.
  - Der Wert des Ausdrucks (bekannt, da konstant) ist im Typ der Variablen darstellbar.

Beispiele:

```
long l = 3;       // Erweiternde Umwandlung von int nach long
byte b = 4;       // Einengende Umwandlung von int nach byte
```

Umwandlung in Methodenaufrufen

Die folgenden Umwandlungen werden auf jeden Argumentwert einzeln angewandt. Mögliche Umwandlungen sind:

- Erweiternde Umwandlung einfacher Typen
- Erweiternde Umwandlung von Referenztypen (später)

Es ist also hierbei keine einengende Umwandlung möglich.

Beispiel:

```
class MeineKlasse {
    void meineMethode1(long x) { ... }
    void meineMethode2(short x) { ... }

    void aufrufendeMethode(long x, byte y, int z) {

        meineMethode1(z);          // möglich (erweiternd)
        meineMethode1(y);          // möglich (erweiternd)
        meineMethode2(y);          // möglich (erweiternd)
        meineMethode2(z);          // Fehler: nicht möglich
    }
}
```

Explizite Umwandlung

Eine explizite Umwandlung durch einen cast-Ausdruck ist für alle Kategorien (bis auf String-Umwandlung) prinzipiell möglich, d.h.:

- Erweiternde Umwandlung einfacher Typen
- Erweiternde Umwandlung von Referenztypen
- Einengende Umwandlung einfacher Typen
- Einengende Umwandlung von Referenztypen

Zur Übersetzungszeit und zur Laufzeit wird jedoch überprüft, ob diese prinzipiell zulässige Umwandlung auch konkret ausgeführt werden darf. Probleme kann es dabei jedoch nur bei Referenztypen geben (später).

Beispiele:

```
byte b = (byte)1234567890;          // einengende Umwandlung
int i = (int)b;                     // erweiternde Umwandlung
```

Anpassung numerischer Typen

Die Anpassung numerischer Typen wird verwendet, um die Operanden eines numerischen Operators in einen gemeinsamen Typ umzuwandeln, so dass eine Operation ausgeführt werden kann.

Dabei wird zwischen der Anpassung für unäre und binäre Operationen unterschieden.

Umwandlung für unäre Operationen

Ist der Operand vom Typ byte, short oder char, so wird eine erweiternde Umwandlung nach int vorgenommen.

Diese Umwandlung findet statt bei:

- Dimensionsausdruck bei der Erzeugung von Felder
  Beispiel: `int[] feld = new int['z'];` `// Umwandlung von 'z' nach int`
- Indexausdruck bei Feldzugriffen
  Beispiel: `feld['a'] = 5;` `// Umwandlung von 'a' nach int`
- Operand von unärem + und –
  Beispiel: `byte b = 5; int i = -b;` `// Umwandlung von b nach int`
- Operand des bitweisen Komplementäroperators ~
  Beispiel: `int i = ~b;` `// Umwandlung von b nach int`
- Jeder Operand einzeln (!) der Shift-Operatoren <<,>>,>>>
  Beispiel: `int i = b >> b;` `// 2x Umwandlung von nach int`

Umwandlung für binäre Operationen

Regeln der Reihe nach anwenden:

- Einer der Operanden double: andere auch nach double
- Einer der Operanden float: andere auch nach float
- Einer der Operanden long: andere auch nach long
- Ansonsten, beide Operanden nach int umwandeln

Faustregel: Immer zum größeren Typen hin umwandeln.

Die Umwandlung findet statt bei:

- +, -, *, /, %
- Numerische Vergleichoperatoren <, <=, >, >=, ==, !=
- Bitoperatoren &, |, ^
- Bedingungsoperator ?: (etwas komplizierter, wird nicht behandelt)

Beispiele:

```
byte b = 3;
int i= 3 * b;              // Umwandlung von b nach int
double d = 3.0 + 4;        // Umwandlung von 4 nach double
```