

Kapitel 3: Grunddatentypen, Ausdrücke und Variablen

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik

Institut für Informatik, LMU München

WS 2018/19

Stand der Folien: 31. Oktober 2018

Die Inhalte dieser Folien basieren – mit freundlicher Genehmigung – tlw. auf Folien von Prof. Dr. Rolf Hennicker aus dem WS 2017/18 und auf Folien von PD Dr. Ulrich Schöpp aus dem WS 2010/11



Grunddatentypen in Java

Eine **Datenstruktur** besteht aus

- einer Menge von **Daten** (Werten)
- charakteristischen **Operationen**

Datenstrukturen werden mit einem Namen bezeichnet, den man **Datentyp** nennt.

In Java gibt es **grundlegende** Datenstrukturen für

- Ganze Zahlen
- Gleitkommazahlen
- Zeichen
- Boolesche Werte

Grammatik für Grunddatentypen in Java

PrimitiveType = NumericType | "boolean" | "char"

NumericType = IntegralType | FloatingPointType

IntegralType = "byte" | "short" | "int" | "long"

FloatingPointType = "float" | "double"

Ganze Zahlen in Java

Typ	Größe	Wertebereich
byte	1 Byte (8 bit)	-128 bis 127 -2^7 bis $2^7 - 1$
short	2 Byte (16 bit)	-32 768 bis 32 767 -2^{15} bis $2^{15} - 1$
int	4 Byte (32 bit)	-2 147 483 648 bis 2 147 483 647 -2^{31} bis $2^{31} - 1$
long	8 Byte (64 bit)	-9 223 372 036 854 775 808 bis 9 223 372 036 854 775 807 -2^{63} bis $2^{63} - 1$

Grammatik für positive ganze Dezimalzahlen

DecimalIntegerLiteral = DecimalNumeral [IntegerTypeSuffix]

DecimalNumeral = "0" | NonZeroDigit [Digits]
| NonZeroDigit Underscores Digits

NonZeroDigit = "1" | "2" | ... | "9"

Digits = Digit | Digit [DigitsAndUnderscores] Digit

Digit = "0" | NonZeroDigit

DigitsAndUnderscores = DigitOrUnderscore { DigitOrUnderscore }

DigitOrUnderscore = Digit | "_"

Underscores = "_" | { "_" }

IntegerTypeSuffix = "l" | "L"

Beachte: Der Unterstriche wurden mit Version 7 hinzugefügt, um die Lesbarkeit langer Zahlen zu verbessern z.B. 1_000_000 statt 1000000.

Ausprobieren in Java

```
public class Zahlentest {
    public static void main(String[] args) {
        byte testByte    = 126;
        short testShort  = -32768;
        int testInt      = 2_147_483_647;
        long testLong    = 2_147_483_648L;
        System.out.println("byte:" + testByte);
        System.out.println("short:" + testShort);
        System.out.println("int:" + testInt);
        System.out.println("long:" + testLong);
    }
}
```

Binärcodierung ganzer Zahlen mit fester Folgenlänge

Eine nicht-negative ganze Zahl x im Bereich $[0, 2^n - 1]$ wird codiert durch eine Bitfolge $b_{n-1} \dots b_0$ der Länge n , so dass gilt:

$$x = b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + \dots + b_0 \cdot 2^0 = \sum_{i=0}^{n-1} b_i \cdot 2^i$$

Beispiele für $n = 3$ und damit dem Bereich $[0, 7]$:

7 wird codiert durch **111**, da $1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 4 + 2 + 1 = 7$

3 wird codiert durch **011**, da $0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 2 + 1 = 3$

Binärcodierung ganzer Zahlen mit fester Folgenlänge (2)

Darstellung positiver und **negativer** Zahlen:

- Verwende ein **zusätzliches Bit am Anfang** ($1 = -, 0 = +$).

Damit nichts verschwendet wird (doppelte 0):

- Negativer Bereich wird dargestellt durch:
 - kleinste negative Zahl ($= -2^n$): 1 gefolgt von n 0en
 - größte negative Zahl ($= -1$): 1 gefolgt von n 1en

Für $n = 3$ und einem Vorzeichenbit, Bereich $[-8, 7]$:

- | | |
|-----------------------------|------------------------------|
| • 0 wird codiert durch 0000 | • -8 wird codiert durch 1000 |
| • 1 wird codiert durch 0001 | • -7 wird codiert durch 1001 |
| • 2 wird codiert durch 0010 | • -6 wird codiert durch 1010 |
| • 3 wird codiert durch 0011 | • -5 wird codiert durch 1011 |
| • 4 wird codiert durch 0100 | • -4 wird codiert durch 1100 |
| • 5 wird codiert durch 0101 | • -3 wird codiert durch 1101 |
| • 6 wird codiert durch 0110 | • -2 wird codiert durch 1110 |
| • 7 wird codiert durch 0111 | • -1 wird codiert durch 1111 |

Binärcodierung ganzer Zahlen mit fester Folgenlänge (3)

Diese Binärcodierung ganzer Zahlen nennt man **Zweierkomplement**
Berechnung der Zweierkomplementdarstellung einer negativen Zahl Z :

- 1 Sei $0b_{n-1} \dots b_0$ die Binärdarstellung von $|Z|$
- 2 Invertiere alle Bits von $0b_{n-1} \dots b_0$
- 3 addiere 1

Binärcodierung ganzer Zahlen mit fester Folgenlänge (3)

Diese Binärcodierung ganzer Zahlen nennt man **Zweierkomplement**
 Berechnung der Zweierkomplementdarstellung einer negativen Zahl Z :

- 1 Sei $0b_{n-1} \dots b_0$ die Binärdarstellung von $|Z|$
- 2 Invertiere alle Bits von $0b_{n-1} \dots b_0$
- 3 addiere 1

Beispiel: Zweierkomplementdarstellung von -56 als 32bit Zahl

$$\begin{array}{r}
 \text{① } |-56| = 56 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011\ 1000 \\
 \text{② } \text{Invertieren} \quad 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100\ 0111 \\
 \text{③ } \text{Addiere 1} \quad 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100\ 0111 \\
 \qquad \qquad \qquad + \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad 1 \\
 \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad \qquad 111 \\
 \hline
 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100\ 1000
 \end{array}$$

```
public class ZweierkomplementTest {
    public static void main(String[] args) {
        int zbetrag = 0b0000_0000_0000_0000_0000_0000_0011_1000;
        int z       = 0b1111_1111_1111_1111_1111_1111_1100_1000;
        System.out.println(" |Z| = " + zbetrag);
        System.out.println("  -Z = " + z);
    }
}
```

Darstellung von Binärliteralen in Java:

Starte mit Präfix 0b, danach folgt die Binärzahl als Zweierkomplement (Unterstriche zur besseren Lesbarkeit erlaubt).

Compilieren und Ausführung:

```
> javac ZweierkomplementTest.java
> java ZweierkomplementTest
|Z| = 56
-Z = -56
```

Gleitkommazahlen

Typ	Größe	Wertebereich	Genauigkeit
float	4 Byte (32 bit)	ca. $\pm 10^{-45}$ bis $\pm 10^{38}$	7 Stellen (8. gerundet)
double	8 Byte (64 bit)	ca. $\pm 10^{-324}$ bis $\pm 10^{308}$	15 Stellen (16. gerundet)

Syntaktische Darstellung:

Beispiele:

- double: 36.22, 3.622E+1, 0.3633e+2, 362.2E-1, -0.73, -7.3E-1
- float: -0.73f, -7.3E-1F

Beachte: **Punkt statt Komma**

Normalformen von Gleitkommazahlen

Eine Gleitkommazahl $r \neq 0$ wird in ihrer Normalform dargestellt durch

["+" | "-"] Mantisse "E" Exponent

wobei

- Mantisse ist eine Dezimalzahl aus dem halboffenen Intervall $[1, 10[$ (mit Punkt für ein Komma)
- Exponent ist eine ganze Zahl
- und $|r| = \text{Mantisse} \cdot 10^{\text{Exponent}}$

Falls $r = 0$, ist die Normalformdarstellung 0.0

Rechnen mit Gleitkommazahlen ist ungenau!

```
public class FloatDoubleExample {
    public static void main(String[] args) {
        float   oneDollarInEuroFloat   = 1.1576f;
        float   amountEuroFloat        = 1000.37f;
        double  oneDollarInEuroDouble  = 1.1576;
        double  amountEuroDouble       = 1000.37;
        System.out.print("Dollar mit float :");
        System.out.println(oneDollarInEuroFloat*amountEuroFloat);
        System.out.print("Dollar mit double:");
        System.out.println(oneDollarInEuroDouble*amountEuroDouble);
    }
}
```

Rechnen mit Gleitkommazahlen ist ungenau!

```
public class FloatDoubleExample {
    public static void main(String[] args) {
        float   oneDollarInEuroFloat   = 1.1576f;
        float   amountEuroFloat        = 1000.37f;
        double  oneDollarInEuroDouble  = 1.1576;
        double  amountEuroDouble       = 1000.37;
        System.out.print("Dollar mit float :");
        System.out.println(oneDollarInEuroFloat*amountEuroFloat);
        System.out.print("Dollar mit double:");
        System.out.println(oneDollarInEuroDouble*amountEuroDouble);
    }
}
```

Ausführung:

```
> javac FloatDoubleExample.java
> java FloatDoubleExample
Dollar mit float :1158.0283
Dollar mit double:1158.028312
```

Arithmetische Operationen

Einige arithmetische Operationen auf **int**, **long**, **float**, **double** in Java sind:

- + Addition
- Subtraktion
- * Multiplikation
- / Division
- % Rest bei ganzzahliger Division

Bemerkungen

- Die Operationen sind zweistellig (und infix).
- Die beiden Argumente einer Operationen müssen **den gleichen Typ haben**; das Ergebnis hat dann den gleichen Typ wie die Argumente

Division und Rest für int-Zahlen n und m

- n/m entsteht durch Division und Abschneiden der Nachkommastellen
- $n \% m$ ist der Rest von n/m .

- Beispiele:

$$14/4 = 3$$

$$14 \% 4 = 2$$

- Beispiele:

$$-14/4 = -3$$

$$-14 \% 4 = -2$$

$$-15/4 = -3$$

$$-15 \% 4 = -3$$

Überlauf

Arithmetische Operationen können zu einem **Überlauf** führen!

Beispiel:

```
public class Ueberlauf {
    public static void main(String[] args) {
        int testValue = 1073741824; // 2 hoch 30;
        int resultValue = 2*testValue;
        System.out.println("testValue: " + testValue);
        System.out.println("resultValue: " + resultValue);
    }
}
```

Aufruf:

```
> java Ueberlauf
testValue: 1073741824
resultValue: -2147483648
```

Mathematische Funktionen

- Die Java-Standardbibliothek stellt eine Reihe von mathematischen Funktionen zur Verfügung
- Zum Beispiel:

```
double y = Math.sqrt(x); // Wurzel von x
int i = Math.round(y); // gerundeter Wert
double u = Math.max(z,10.0); // Maximum von z und 10.0
```

- Sie Java API-Dokumentation von **Math** für weitere Funktionen
<https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>

„Kleiner-Beziehung“ zwischen numerischen Datentypen

`byte < short < int < long < float < double`

Java konvertiert, wenn nötig, Ausdrücke automatisch in den größeren Typ.

Beispiele:

- `1 + 1.7` ist vom Type `double`
- `1 + 1.7f` ist vom Type `float`
- `1.0 + 1.7f` ist vom Type `double`

Typkonversion (2)

Type Casting:

Erzwingen der Typkonversion durch Voranstellen von (type).
(Meist ist type dann ein kleinerer Typ.)

Beispiele

(byte)3 ist vom Typ byte.

(int)(2.0 + 5.0) ist vom Typ int.

(float)1.3e-7 ist vom Typ float.



Beachte: Bei der Typkonversion kann Information verloren gehen!

Beispiele:

(int)5.6 hat den Wert 5

(int)-5.6 hat den Wert -5

Nachkommastellen werden abgeschnitten.

- Typ char (steht für **character**)
- bezeichnet die Menge der Zeichen aus dem Unicode-Zeichensatz
- char umfasst insbesondere den ASCII-Zeichensatz mit kleinen und großen Buchstaben, Ziffern, Sonderzeichen und Kontrollzeichen
- **Darstellung** von Zeichen durch Umrahmung mit Apostroph (normalerweise mit  +  zu erzeugen)
- Beispiele: 'a', 'A', '1', '9', '!', '='
- Falsch: 'aa'
- Spezialzeichen: z.B.
 - \n (Zeilenumbruch)
 - \' (Apostroph)
 - \\ (Backslash)

Exkurs: Zeichenketten

- Zeichenketten sind **Folgen von Zeichen**
- Zeichenketten werden mit doppelten Anführungszeichen umrahmt und sind vom Typ String
- String ist **kein Grunddatentyp**, sondern eine Klasse (mehr Details dazu später)
- Beispiele: "aa", "1. Januar 2000"
- Strings können mit dem Operator „+“ zusammengehängt werden.
- Wird ein Wert eines Grunddatentyps mit einem String zusammengehängt, dann wird er in einen String umgewandelt.

Beispiele:

"x" + 3 ergibt "x3"

" " + 3 ergibt "3"

3.1 + "x" ergibt 3.1x

Boolesche Werte

- Für die Steuerung des Programmablaufs benutzt man

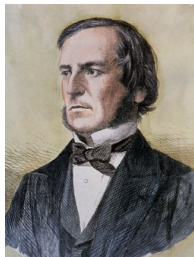
Wahrheitswerte

- Der Typ `boolean` hat genau zwei Werte: `true` und `false`.
- Vergleichtests auf Zahlen liefern Boolesche Werte als Ergebnis

<code>i < j</code>	kleiner	<code>i > j</code>	größer
<code>i <= j</code>	kleiner-gleich	<code>i >= j</code>	größer-gleich
<code>i == j</code>	gleich	<code>i != j</code>	ungleich

- Beispiele:

```
7 < 6    ergibt  false
7 != 6   ergibt  true
7 == 6   ergibt  false
```



George Boole(1815-1864)
Englischer Mathematiker
(Boolesche Algebra)

Boolesche Operationen

- ! Negation
- && Konjunktion „und“ (sequentiell)
- & Konjunktion „und“ (strikt)
- || Disjunktion „oder“ (sequentiell)
- | Disjunktion „oder“ (strikt)

Die **Negation** ist **einstellig**; das Argument muss den Typ `boolean` haben; das Ergebnis ist wieder vom Typ `boolean`

Alle **anderen Operationen** sind **zweistellig**; beide Argumente müssen vom Typ `boolean` sein, das Ergebnis ist wieder vom Typ `boolean`

Auch für Boolesche Werte gibt es den Test auf Gleichheit mit `==`

Negation

a	!a
true	false
false	true

Konjunktion und Disjunktion

&, &&	true	false
true	true	false
false	false	false

Konjunktion

,	true	false
true	true	true
false	true	false

Disjunktion

Beispiel zur Booleschen Logik

- 1 Anna sagt: „Bettina lügt.“
- 2 Bettina sagt: „Anna oder Claudia lügen.“
- 3 Claudia sagt: „Anna und Bettina lügen.“

Wer lügt denn nun?

Beispiel zur Booleschen Logik

- 1 Anna sagt: „Bettina lügt.“
- 2 Bettina sagt: „Anna oder Claudia lügen.“
- 3 Claudia sagt: „Anna und Bettina lügen.“

Wer lügt denn nun?

...

```
boolean aLuegt, bLuegt, cLuegt, alleAussagen;  
aLuegt = ...; bLuegt = ...; cLuegt = ...;
```

```
alleAussagen =  
    (!(aLuegt) == bLuegt)           // Aussage 1  
    && (!(bLuegt) == (aLuegt || cLuegt)) // Aussage 2  
    && (!(cLuegt) == (aLuegt && bLuegt)); // Aussage 3
```

...

Beispiel zur Booleschen Logik

- 1 Anna sagt: „Bettina lügt.“
- 2 Bettina sagt: „Anna oder Claudia lügen.“
- 3 Claudia sagt: „Anna und Bettina lügen.“

Wer lügt denn nun?

...

```
boolean aLuegt, bLuegt, cLuegt, alleAussagen;  
aLuegt = ...; bLuegt = ...; cLuegt = ...;  
  
alleAussagen =  
    (!(aLuegt) == bLuegt)           // Aussage 1  
    && (!(bLuegt) == (aLuegt || cLuegt)) // Aussage 2  
    && (!(cLuegt) == (aLuegt && bLuegt)); // Aussage 3
```

...

Ausprobieren aller möglichen Belegungen zeigt, dass alleAussagen nur wahr wird bei:

```
aLuegt = true; bLuegt = false; cLuegt = true;
```

Sequentielle und strikte Operatoren

- Bei sequentiellen Operatoren `&&` bzw. `||` wird von links nach rechts ausgewertet und das zweite Argument ignoriert, wenn das Ergebnis nach der ersten Auswertung schon „klar“ ist. Zum Beispiel ergibt `(false && „undefiniert“)` als Ergebnis `false`. Aber `(„undefiniert“ && false)` ergibt `(„undefiniert“)`.
- Bei strikten Operatoren `&` bzw. `|` werden immer beide Argumente ausgewertet und dann der Ausdruck. Wenn ein Argument undefiniert ist, dann ist der ganze Ausdruck undefiniert. Beispiel `(false & „undefiniert“)` ergibt `(„undefiniert“)`

Beispiele für die sequentielle und die strikte Konjunktion

`(0 == 1) && (100/0 > 1)` ergibt `false`

`(0 == 1) & (100/0 > 1)` ergibt Laufzeitfehler

`(100/0 > 1) & (0 == 1)` ergibt Laufzeitfehler

`(100/0 > 1) && (0 == 1)` ergibt Laufzeitfehler

Beispiele für die sequentielle und die strikte Disjunktion

`(0 == 0) || (100/0 > 1)` ergibt `true`

`(0 == 0) | (100/0 > 1)` ergibt Laufzeitfehler

`(100/0 > 1) | (0 == 0)` ergibt Laufzeitfehler

`(100/0 > 1) || (0 == 0)` ergibt Laufzeitfehler

Beobachten: strikt vs. sequentiell

```
public class BoolExample2 {
    public static void main(String[] args) {
        System.out.print("false && trueWithPrint() :"); // seq and
        System.out.println(false && trueWithPrint());
        System.out.print("false & trueWithPrint() :"); // strict and
        System.out.println(false & trueWithPrint());
        System.out.print("trueWithPrint() && false :"); // seq and
        System.out.println(trueWithPrint() && false);
        System.out.print("trueWithPrint() & false :"); // strict and
        System.out.println(trueWithPrint() & false);
    }
    // static method to return true, but doing a print before
    static boolean trueWithPrint() {
        System.out.print(" [Executing trueWithPrint] ");
        return true;
    }
}
```


Syntax: Expression "?" Expression ":" Expression

- Wird auch **ternärer Operator**, **?-Operator**, **?:-Operator** genannt.
- In $B ? E_1 : E_2$ muss B ein boolescher Ausdruck sein, die Typen von E_1 und E_2 sind meistens identisch, genauer gilt: der Typ von $B ? E_1 : E_2$ ist der größte gemeinsame Typ der Typen von E_1 und E_2 .
- **Wert** von $B ? E_1 : E_2$ ist
 - der Wert von E_1 , wenn B den Wert `true` hat
 - der Wert von E_2 , wenn B den Wert `false` hat

Der Bedingungsoperator (2)

Beispiele:

- $(1 > 0) ? 10 : 20$ ist gleich zu 10, da $(1 > 0)$ zu true ausgewertet.
- $(1 == 0) ? 'A' : 'B'$ ist gleich zu 'B'
- $((1 < 0) || (1 != 0)) ? false : true$ ist gleich zu false, da $((1 < 0) || (1 != 0))$ zu true ausgewertet.
- $a > b ? a : b$ liefert das Maximum aus a und b

Klammerung bei Schachtelung

Der Bedingungsoperator ist **rechts-assoziativ**:

$$a ? b : c ? d : e$$

entspricht

$$a ? b : (c ? d : e)$$

und **nicht**

$$(a ? b : c) ? d : e$$

Ausdrücke werden (vorläufig) gebildet aus

- Werten
- Variablen
- Anwendung von Operationen auf Ausdrücke
- Klammern um Ausdrücke

Beispiele:

$(-x + y) * 17$ (x, y seien Variable vom Typ int)

$x == y \ \&\& \ !b$ (b sei eine Variable vom Typ boolean)

Grammatik für Ausdrücke

Expression	= Variable Value Expression BinOp Expression UnOp Expression "(" Expression ")" Expression "?" Expression ":" Expression
Variable	= NamedVariable
NamedVariable	= Identifier
Value	= IntegerValue FloatingPointValue CharacterValue BooleanValue
BooleanValue	= "true" "false"
UnOp	= "!" "(" Type ")" "-" "+"
BinOp	= "&" " " "&&" " " "+" "-" "*" "/" "%" "==" "!=" ">" ">=" "<" "<="
Type	= PrimitiveType

Beachte: Expression, Variable, Value, Type werden später erweitert.

Ein syntaktisch korrekter Ausdruck kann trotzdem nicht gültig sein:

Nebenbedingung:

Ausdrücke müssen nicht nur syntaktisch korrekt sein (gemäß der Regeln), sondern auch **typkorrekt** gebildet werden!

Beispiel:

`true + 1` ist syntaktisch korrekt, aber **nicht typkorrekt!**

Typ eines Ausdrucks

Ein Ausdruck ist **typkorrekt**, wenn ihm ein Typ zugeordnet werden kann.

Die **Zuordnung** eines Typs erfolgt unter Beachtung:

- der Typen der in dem Ausdruck vorkommenden Werte und Variablen,
- der Argument- und Ergebnistypen der in dem Ausdruck vorkommenden Operationen,
- Klammerungen und Präzedenzen

Beispiele:

`(-3 + 12) * 17` hat den Typ `int`

`(5 == 7) && (!true)` hat den Typ `boolean`

Präzedenzen von Operatoren bestimmen deren **Bindungsstärke** (z.B. „Punkt vor Strich“) und erlauben dadurch Klammerersparnis.

Operator	Präzedenz	Operator	Präzedenz
!, unäres +, -	14	&	7
(type)	13		6
*, /, %	12	&&	4
binäres +, -	11		3
>, >=, <, <=	9	ternäres ?:	2
==, !=	8		

Höherer Präzedenzwert bedeutet:

Der Operator **zieht die Argumente an sich**:

Beispiel:

```
(3 + 5 * 4 > -2 - 2 & 4 < 1 + 2
      && true || (float)11.1 == 11.1f)
```


Präzedenzen von Operatoren bestimmen deren **Bindungsstärke** (z.B. „Punkt vor Strich“) und erlauben dadurch Klammerersparnis.

Operator	Präzedenz	Operator	Präzedenz
!, unäres +, -	14	&	7
(type)	13		6
*, /, %	12	&&	4
binäres +, -	11		3
>, >=, <, <=	9	ternäres ?:	2
==, !=	8		

Höherer Präzedenzwert bedeutet:

Der Operator **zieht die Argumente an sich**:

Beispiel:

```
(3 + 5 * 4 > (-2) - 2 & 4 < 1 + 2
      && true || (float)11.1 == 11.1f)
```

Präzedenzen von Operatoren bestimmen deren **Bindungsstärke** (z.B. „Punkt vor Strich“) und erlauben dadurch Klammerersparnis.

Operator	Präzedenz	Operator	Präzedenz
!, unäres +, -	14	&	7
(type)	13		6
*, /, %	12	&&	4
binäres +, -	11		3
>, >=, <, <=	9	ternäres ?:	2
==, !=	8		

Höherer Präzedenzwert bedeutet:

Der Operator **zieht die Argumente an sich**:

Beispiel:

`(3 + 5 * 4 > (-2) - 2 & 4 < 1 + 2`

`&& true || ((float)11.1) == 11.1f)`

Präzedenzen von Operatoren bestimmen deren **Bindungsstärke** (z.B. „Punkt vor Strich“) und erlauben dadurch Klammerersparnis.

Operator	Präzedenz	Operator	Präzedenz
!, unäres +, -	14	&	7
(type)	13		6
*, /, %	12	&&	4
binäres +, -	11		3
>, >=, <, <=	9	ternäres ?:	2
==, !=	8		

Höherer Präzedenzwert bedeutet:

Der Operator **zieht die Argumente an sich**:

Beispiel:

$(3 + (5 * 4) > (-2) - 2 \& 4 < 1 + 2$

$\&\& \text{true} \ || \ ((\text{float})11.1) == 11.1\text{f})$

Präzedenzen von Operatoren bestimmen deren **Bindungsstärke** (z.B. „Punkt vor Strich“) und erlauben dadurch Klammerersparnis.

Operator	Präzedenz	Operator	Präzedenz
!, unäres +, -	14	&	7
(type)	13		6
*, /, %	12	&&	4
binäres +, -	11		3
>, >=, <, <=	9	ternäres ?:	2
==, !=	8		

Höherer Präzedenzwert bedeutet:

Der Operator **zieht die Argumente an sich**:

Beispiel:

```
((3 + (5 * 4)) > ((-2) - 2) & 4 < (1 + 2)
&& true || ((float)11.1) == 11.1f)
```

Präzedenzen von Operatoren bestimmen deren **Bindungsstärke** (z.B. „Punkt vor Strich“) und erlauben dadurch Klammerersparnis.

Operator	Präzedenz	Operator	Präzedenz
!, unäres +, -	14	&	7
(type)	13		6
*, /, %	12	&&	4
binäres +, -	11		3
>, >=, <, <=	9	ternäres ?:	2
==, !=	8		

Höherer Präzedenzwert bedeutet:

Der Operator **zieht die Argumente an sich**:

Beispiel:

```
((3 + (5 * 4)) > ((-2) - 2)) & (4 < (1 + 2))  
&& true || ((float)11.1) == 11.1f)
```

Präzedenzen von Operatoren bestimmen deren **Bindungsstärke** (z.B. „Punkt vor Strich“) und erlauben dadurch Klammerersparnis.

Operator	Präzedenz	Operator	Präzedenz
!, unäres +, -	14	&	7
(type)	13		6
*, /, %	12	&&	4
binäres +, -	11		3
>, >=, <, <=	9	ternäres ?:	2
==, !=	8		

Höherer Präzedenzwert bedeutet:

Der Operator **zieht die Argumente an sich**:

Beispiel:

```
(( (3 + (5 * 4)) > ((-2) - 2)) & (4 < (1 + 2))  
    && true || (((float)11.1) == 11.1f))
```

Präzedenzen von Operatoren bestimmen deren **Bindungsstärke** (z.B. „Punkt vor Strich“) und erlauben dadurch Klammerersparnis.

Operator	Präzedenz	Operator	Präzedenz
!, unäres +, -	14	&	7
(type)	13		6
*, /, %	12	&&	4
binäres +, -	11		3
>, >=, <, <=	9	ternäres ?:	2
==, !=	8		

Höherer Präzedenzwert bedeutet:

Der Operator **zieht die Argumente an sich**:

Beispiel:

```
((((3 + (5 * 4)) > ((-2) - 2)) & (4 < (1 + 2)))  
    && true || (((float)11.1) == 11.1f))
```

Präzedenzen von Operatoren bestimmen deren **Bindungsstärke** (z.B. „Punkt vor Strich“) und erlauben dadurch Klammersparnis.

Operator	Präzedenz	Operator	Präzedenz
!, unäres +, -	14	&	7
(type)	13		6
*, /, %	12	&&	4
binäres +, -	11		3
>, >=, <, <=	9	ternäres ?:	2
==, !=	8		

Höherer Präzedenzwert bedeutet:

Der Operator **zieht die Argumente an sich**:

Beispiel:

```
(((((3 + (5 * 4)) > ((-2) - 2)) & (4 < (1 + 2))))  
&& true) || (((float)11.1) == 11.1f))
```


Vorgehensweise zur Typüberprüfung von E :

- 1 Den Ausdruck E von links nach rechts durchgehen und vollständig klammern unter Berücksichtigung der Präzedenzen.
- 2 Den Ausdruck E von innen nach außen durchgehen und von links nach rechts überprüfen, ob die Argumenttypen von Operationen zu den (Ergebnis-)Typen der Ausdrücke, auf die die Operationen angewendet werden, passen.

Typüberprüfung von Ausdrücken (2)

Beispiel: $7 < 8 + 3$

- 1 Klammern zu $(7 < (8 + 3))$,
da $+$ höhere Präzedenz hat als $<$.
- 2 Betrachte zunächst $8 + 3$:

8 und 3 haben Typ `int`, $+$ kann `int` als Argumenttypen nehmen und liefert als Ergebnis den Typ `int`.

Nun betrachte $7 < (8 + 3)$:

7 hat Typ `int` und $8 + 3$ hat Typ `int` und $<$ kann `int` als Argumenttyp haben und liefert dann ein Ergebnis vom Typ `boolean`.

Der Gesamtausdruck ist daher typkorrekt (vom Typ `boolean`)

Variablen und Zustände

Eine **Variable** ist ein „Behälter“ (Speicherplatz), der zu jedem Zeitpunkt (während des Programmablaufs) einen Wert eines bestimmten Datentyps enthält.

Die Syntax dazu ist (kennen wir schon):

Variable = NamedVariable
NamedVariable = Identifier

- Variablen müssen vor ihrer Benutzung **deklariert** werden.
- Bei der Deklaration wird ihnen ein **Typ** zugeordnet.
- Die Variable kann bei ihrer Deklaration **initialisiert** werden mit einem Ausdruck passenden Typs.

Variablendeklaration

Syntax der Variablendeklaration:

VariableDeclaration = Type VariableDeclarator { ", " VariableDeclarator } ";"
VariableDeclarator = NamedVariable ["=" Expression]

Beispiel:

```
int total = -5;  
int quadrat = total * total;  
boolean aussage = false;
```

Bemerkung:

Variablen müssen nicht sofort bei der Deklaration initialisiert werden, aber vor ihrer ersten Benutzung (wird vom Compiler überprüft).




Zustand

- Ein **Zustand** ist eine Belegung der (zum aktuellen Zeitpunkt) deklarierten Variablen mit Werten.
- Ein Zustand wird **abstrakt dargestellt** durch eine **Liste von Paaren**, bestehend aus einem **Variablennamen** und einem (zugehörigen) **Wert**.

Beispiel: Abstrakte Darstellung eines Zustands σ :

$$\sigma = [(total, -5), (quadrat, 25), (aussage, false)]$$

Im Speicher:

aussage	F002	false
quadrat	F001	25
total	F000	-5
		
Variable	Adresse	Wert

Gegeben sei ein typkorrekter Ausdruck E und ein Zustand σ für die in E vorkommenden Variablen.

Vorgehensweise zur Auswertung von E unter σ

- 1 Den Ausdruck E von links nach rechts durchgehen und vollständig klammern unter Berücksichtigung der Präzedenzen.
- 2 Den Ausdruck E von innen nach außen und links nach rechts durchgehen und die Operationen auswerten. Der Wert von Variablen ist dabei durch den Zustand σ bestimmt.

Auswertung von Ausdrücken (2)

Beispiel:

Der Zustand nach Deklaration `int x = 8;` ist $\sigma = [(x, 8)]$

Auswertung von `7 < x + 3` unter σ :

- 1 Vollständige Klammerung: `7 < (x + 3)`
- 2 $7 < (x + 3)$
 $=_{\sigma} 7 < (8 + 3)$
 $=_{\sigma} 7 < 11$
 $=_{\sigma} \text{true}$

Beachte: Jeder Auswertungsschritt wird mit $=_{\sigma}$ bezeichnet.

- Grunddatentypen für Ganzzahlen in Java:
byte,short,int,long
- Grunddatentypen für Fließkommazahlen in Java float und double
- Arithmetische Operationen
- char ist der Grunddatentyp für Zeichen
- boolean für Wahrheitswerte
- Vergleichstests und Boolesche Operatoren (insbesondere auch strikt vs. sequentiell)
- Ausdrücke: Typisierung und Auswertung
- $=_{\sigma}$ zur Auswertung