

# Kapitel 3

---

## Grunddatentypen, Ausdrücke und Variable

## 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
- Gleitpunktzahlen
- 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

Typ	Größe	Wertebereich	
▪ <b>byte</b>	1 Byte = 8 Bit	-128 $-2^7$	bis 127 bis $2^7-1$ } $2^8 = 256$ Zahlen
▪ <b>short</b>	2 Byte	-32768 $-2^{15}$	bis 32767 bis $2^{15}-1$
▪ <b>int</b>	4 Byte = 32 Bit	-2 147 483 648 $-2^{31}$	bis 2 147 483 647 bis $2^{31}-1$ } $2^{32}$ Zahlen
▪ <b>long</b>	8 Byte	-9 223 372 036 854 775 808 $-2^{63}$	bis 9 223 372 036 854 775 807 bis $2^{63}-1$

Syntaktische Darstellung: Worte von *IntegerValue* (vgl. Kap. 2)

## Binärcodierung ganzer Zahlen

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

$$x = b_{n-1} * 2^{n-1} + \dots + b_0 * 2^0$$

Beispiele für  $n = 3$  und den Bereich  $[0, 2^3 - 1] = [0, 7]$ :

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

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

Für die Codierung aller ganzer Zahlen wird ein Bit für die Darstellung des Vorzeichens hinzugenommen (1 für -, 0 für +).

Beispiele: Integer.MIN\_VALUE wird codiert durch 1 gefolgt von 31 mal 0.

Integer.MAX\_VALUE wird codiert durch 0 gefolgt von 31 mal 1.

# Spiel "Zahl erraten"

Gegeben sind 6 Karten. Auf jeder Karte befindet sich eine Auswahl ganzer Zahlen im Bereich 1 bis 63.

Ein Mitspieler wird aufgefordert, sich eine ganze Zahl im Bereich 0 bis 63 auszudenken.

Für jede Karte soll er angeben, ob sich die Zahl auf der Karte befindet. Damit erhält der Spieler 6 Informationseinheiten (Bits).

Die Karten seien von 0 bis 5 durchnummeriert und so organisiert, dass eine Zahl  $x$  genau dann auf der Karte  $i \in \{0, \dots, 5\}$  ist, wenn in der Binärcodierung  $x = b_5 \cdot 2^5 + \dots + b_0 \cdot 2^0$  gilt:  $b_i = 1$ .  
Damit kann die Zahl  $x$  ausgerechnet werden.

Folgende Zahlen befinden sich auf den Karten:

Karte 0

1	3	5	7	9	11	13	15
17	19	21	23	25	27	29	31
33	35	37	39	41	43	45	47
49	51	53	55	57	59	61	63

Karte 1

2	3	6	7	10	11	14	15
18	19	22	23	26	27	30	31
34	35	38	39	42	43	46	47
50	51	54	55	58	59	62	63

## Karte 2

4	5	6	7	12	13	14	15
20	21	22	23	28	29	30	31
36	37	38	39	44	45	46	47
52	53	54	55	60	61	62	63

## Karte 3

8	9	10	11	12	13	14	15
24	25	26	27	28	29	30	31
40	41	42	43	44	45	46	47
56	57	58	59	60	61	62	63



## Karte 4

16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

## Karte 5

32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

# Gleitkommazahlen

+ 0,0...01  
 - 324 Nullen

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

nach IEEE-754-Standard (1985)

+/- 1 0...0  
 308 Nullen

## Syntaktische Darstellung:

Beispiele:

- double:** 36.22,  $3.622 \times 10^{+1}$ ,  $0.3622 \times 10^2$ ,  $362.2 \times 10^{-1}$   
 -0.73,  $-7.3 \times 10^{-1}$
- float:** -0.73f,  $-7.3 \times 10^{-1}F$

Normalform-Darstellung

## Normalform von Gleitkommazahlen

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

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

wobei

$- 7.3 E - 1$

*Mantisse* eine Dezimalzahl aus dem halboffenen Intervall  $[1, 10[$  ist

(mit Punkt für Komma)

z.B.  $7.3$

$9.999$

*Exponent* eine ganze Zahl ist und

$$|r| \neq \text{Mantisse} * 10^{\text{Exponent}}$$

z.B.  $| -0,73 | = 7.3 * 10^{-1}$

Falls  $r = 0$ , ist die Normalformdarstellung  $0.0$  *ist vom Typ Double*

$0$  *ist eine ganze Zahl*

## Arithmetische Operationen

- Arithmetische Operationen auf **int**, **long**, **float** und **double**

+ Addition

- Subtraktion

\* Multiplikation

/ Division

% Rest bei ganzzahliger Division

*"modulo"*

- Die Operationen sind zweistellig. Die beiden Argumente einer Operation **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  $n/m$ .

- Beispiel:  $14/4 = 3$ ,  $14\%4 = 2$   $14 = 4 * 3 + 2$

- Beispiel:  $-14/4 = -3$ ,  $-14\%4 = -2$ ,  $-15/4 = -3$ ,  $-15\%4 = -3$

- Arithmetische Operationen können zu **Überlauf** führen.

$$-14 = 4 * (-3) + (-2)$$

$$-16/4 = -4$$

$$-16\%4 = 0$$

## Mathematische Funktionen

- Die Java Standardbibliothek stellt eine Reihe von mathematischen Funktionen zur Verfügung.
- Beispiele:

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

- Siehe die Java API-Dokumentation von **Math** für weitere Funktionen.

## Typkonversion (1)

„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 Typ **double**  
 $1 + 1.7f$  ist vom Typ **float**  
 $1.0 + 1.7f$  ist vom Typ **double**

*Handwritten notes:*  
A blue line is drawn under  $1 + 1.7$  with an arrow pointing to  $1.0$ .  
A blue arrow points from  $1.7$  to  $1.0f$ .  
A blue arrow points from  $1.7$  to  $1.7$ .

$14/4 = 3$

$14/4.0$

$14.0/4.0 = 3.5$

$14.0/4 = 3.5$

$4.7 + 3$

*Handwritten notes:*  
A blue arrow points from  $3$  to  $3.0$ .  
A blue arrow points from  $4.0$  to  $4.0$ .

## Typkonversion (2)

### Type Casting:

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

### Beispiele:

<code>(byte)3</code>	ist vom Typ <b>byte</b>
<code>(int)(2.0 + 5.0)</code>	ist vom Typ <b>int</b>
<code>(float)1.3e-7</code>	ist vom Typ <b>float</b>

**Beachte:** Bei der Typkonversion kann Information verloren gehen!

**Beispiel:** `(int) 5.6` hat den Wert `5`  
`(int)-5.6` hat den Wert `-5`  
Nachkommastellen werden abgeschnitten

## Zeichen

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



## Exkurs: Zeichenketten

- Zeichenketten werden mit hochgestellten Anführungszeichen umrahmt und sind vom Typ **String**.
- **String** ist kein Grunddatentyp sondern eine Klasse (vgl. später).
- Beispiele: "aa", "1. Januar 2000" *z.B. "Hal" + "lo" ergibt*
- Strings können mit der Operation „+“ zusammengehängt werden.
- Wird ein Wert eines Grunddatentyps mit einem String *"Hallo"* zusammengehängt, dann wird er in einen String umgewandelt.

*↑ "3"*  
"x" + 3      ergibt    "x3"

" " + 3      ergibt    " 3"

3.1 + "x"    ergibt    "3.1x"

*↓*  
"3.1"

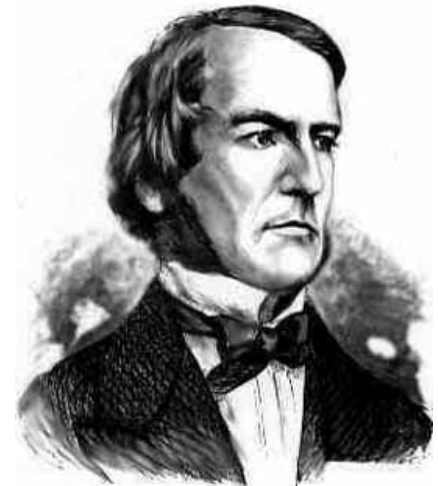
## Boolesche Werte

- Für die Steuerung des Programmablaufs benutzt man Wahrheitswerte.
- Der Typ **boolean** hat genau zwei Werte: **true** und **false**.
- Vergleichstests auf Zahlen liefern Boolesche Werte als Ergebnis

$i < j$	kleiner	$i > j$	größer
$i \leq j$	kleiner-gleich	$i \geq j$	größer-gleich
$i == j$	gleich	$i != j$	ungleich

- Beispiel:

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



**George Boole**  
1815-1864  
Engl. Mathematiker  
Boolesche Algebra  
der Aussagenlogik

## Boolesche Operationen

!	Negation	! true ist false
&&	Konjunktion „und“ (sequentiell)	! false ist true
&	Konjunktion „und“ (strikt)	
	Disjunktion „oder“ (sequentiell)	! (3 == 4)
	Disjunktion „oder“ (strikt)	

- Die Negation ist einstellig; das Argument muss den Typ `boolean` haben; das Ergebnis hat wieder den Typ `boolean`.
- Alle anderen Operationen sind zweistellig; beide Argumente müssen den Typ `boolean` haben; das Ergebnis hat wieder den Typ `boolean`.

Außerdem gibt es auch für Boolesche Werte: `==` (Test auf Gleichheit)

*(true == false) hat den Wert false*

## Wahrheitstabellen

Konjunktion

<code>&amp;&amp;, &amp;</code>	true	false
true	true	false
false	false	false

Disjunktion

<code>  ,  </code>	true	false
true	true	true
false	true	false

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

## Beispiele

### Beispiele für die sequentielle und die strikte Konjunktion

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

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

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

`true && (100/0 > 1)` ergibt einen Laufzeitfehler (undefiniert)

### Beispiele für die sequentielle und die strikte Disjunktion

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

`true | (1/0 == 1)` ergibt einen Laufzeitfehler (undefiniert)

`(1/0 == 1) || true` ergibt einen Laufzeitfehler (undefiniert)

`false || (1/0 == 1)` ergibt einen Laufzeitfehler (undefiniert)

## Ausdrücke

**Ausdrücke** werden (vorläufig) gebildet aus

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

**Beispiele:**

$(\ominus x + y) * 17$

$-x$                        $+x$   
 $\neg -x$                        $\neg +x$

( $x, y$  seien Variable vom Typ `int`)

$x == y \ \&\& \ !b$   
boolean                      boolean  
boolean

( $b$  sei eine Variable vom Typ `boolean`)

## Grammatik für Ausdrücke

*Expression* = *Variable* |  
*Value* |  
*Expression BinOp Expression* |  
*UnOp Expression* |  
"(" *Expression* ")"

*Variable* = *NamedVariable*

*NamedVariable* = *Identifizier*

*Value* = *IntegerValue* | *FloatingPointValue* | *CharacterValue* |  
*BooleanValue*

*BooleanValue* = "true" | "false"

- *Identifizier* und *IntegerValue* wurden in Kap. 2 definiert. Die restlichen Werte wurden an Beispielen erläutert.
- *Expression*, *Variable* und *Value* werden später erweitert.

## Grammatik für Ausdrücke (Fortsetzung)

$UnOp = "!" \mid "(" \textit{Type} ")" \mid "-" \mid "+"$

$BinOp = "&" \mid "|" \mid "&&" \mid "||" \mid "+" \mid "-" \mid "*" \mid$   
 $"/" \mid "\%" \mid "==" \mid "!=" \mid ">" \mid ">=" \mid "<" \mid "<="$

$Type = \textit{PrimitiveType}$  (*Type* wird später erweitert.)

### 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:**

*int int int int int*  
 $(-3 + 12) * 17$

hat den Typ `int`

*int int boolean boolean boolean*  
 $(5 == 7) \ \&\& \ (!\text{true})$

hat den Typ `boolean`

~~$\Rightarrow (8 + \text{false})$~~

*int*  $\Rightarrow (8 + 3)$   
 typkorrekt?

## Präzedenzen

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

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

## Typüberprüfung von Ausdrücken

Gegeben sei ein gemäß der Regeln syntaktisch korrekter Ausdruck E.

**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 nochmals von links nach rechts durchgehen und unter Berücksichtigung der Klammern überprüfen, ob die Argumenttypen von Operationen zu den (Ergebnis-)Typen der Ausdrücke, auf die die Operationen angewendet werden, passen.

**Beispiel:**  $7 < 8 + 3$



1. Vollständige Klammerung:  $7 < (8 + 3)$  da + höhere Präzedenz hat als <.
2. Die Zahl 7 hat den Typ `int`,  
8 und 3 haben den Typ `int` =>  $(8 + 3)$  hat den Typ `int`,  
< kann auf Argumente vom Typ `int` angewendet werden  
und hat den Ergebnistyp `boolean` =>  
 $7 < (8 + 3)$  hat den Typ `boolean` und ist damit auch typkorrekt.

## Variable und Zustände

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

Syntax (Wdh.):

*Variable = NamedVariable*

*NamedVariable = Identifier*

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

Syntax:

*VariableDeclaration =*

*Type VariableDeclarator { " " VariableDeclarator } " ; "*

*VariableDeclarator = NamedVariable [ " = " Expression ]*

*int x = 7, y;*

## Variablendeklaration

### *Beispiel:*

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

### *Bemerkung:*

Variablen müssen nicht sofort bei ihrer Deklaration initialisiert werden jedoch 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 Zustand  $\sigma$ :

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

Im Speicher:

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

## Auswertung von Ausdrücken

Gegeben sei ein typkorrekter Ausdruck E **und** ein Zustand  $\sigma$  für die in E vorkommenden Variablen.

**Vorgehensweise** zur **Auswertung von E unter  $\sigma$** :

1. Den Ausdruck E von links nach rechts durchgehen und vollständig klammern unter Berücksichtigung der Präzedenzen.
2. Den Ausdruck E nochmals von links nach rechts durchgehen und, unter Berücksichtigung der Klammern, die Operationen auswerten. Der Wert von Variablen ist dabei durch den Zustand  $\sigma$  bestimmt.

**Beispiel:** `int x = 8;` Zustand  $\sigma = [(x, 8)]$

Auswertung von `7 < x + 3` unter  $\sigma$ :

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

**Beachte:**

Jeder Auswertungsschritt wird mit  $=_{\sigma}$  bezeichnet!