

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

primitive types

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

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

Gleitkommazahlen

$\pm 0.0\dots01$
323 und 0

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

nach IEEE-754-Standard (1985)

Syntaktische Darstellung:

Beispiele:

▪ **double:** 36.22, 3.622×10^1 , $3.622E+1$, 0.3622e+2, 362.2E-1

-0.73, -7.3×10^{-1} , $-7.3E-1$ Normalformen

▪ **float:** -0.73f, -7.3E-1F

Normalform von Gleitkommazahlen

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

$$("+ | "-) \textit{Mantisse} "E" \textit{Exponent}$$

wobei

Mantisse eine Dezimalzahl aus dem halboffenen Intervall $[1,10[$ ist
(mit Punkt für Komma)

Exponent eine ganze Zahl ist und

$$r = \textit{Mantisse} * 10^{\textit{Exponent}}$$

Falls $r = 0$, ist die Normalformdarstellung 0.0

Arithmetische Operationen

- Arithmetische Operationen auf **int**, **long**, **float** und **double**
 - + Addition
 - Subtraktion
 - * Multiplikation
 - / Division
 - % Rest bei ganzzahliger Division
- 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: $14/4 = 3$, $14\%4 = 2$
 - Beispiel: $-14/4 = -3$, $-14\%4 = -2$, $-15/4 = -3$, $-15\%4 = -3$
- Arithmetische Operationen können zu **Überlauf** führen.

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:

<code>1 + 1.7</code>	ist vom Typ <code>double</code>
<code>1 + 1.7f</code>	ist vom Typ <code>float</code>
<code>1.0 + 1.7f</code>	ist vom Typ <code>double</code>

Typkonversion (2)

Type Casting:

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

Beispiele:

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

Beachte: Bei der Typkonversion kann Information verloren gehen!

Beispiel:

<code>(int) 5.6</code>	<code>==</code>	<code>5</code>
<code>(int) -5.6</code>	<code>==</code>	<code>-5</code>

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"`
- Strings können mit der Operation „+“ zusammengehängt werden.
- Wird ein Wert eines Grunddatentyps mit einem String zusammengehängt, dann wird er in einen String umgewandelt.

`"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**.
- Vergleichstest 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
- && 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 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)

Wahrheitstabellen

Konjunktion

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

Disjunktion

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

- 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“.
- 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“.

Beispiele

Beispiele für die sequentielle und die strikte Konjunktion

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

$(-x + y) * 17$

(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* ")"

Variable = *NamedVariable*

NamedVariable = *Identifier*

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

BooleanValue = "true" | "false"

- *Identifier* 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!

! ? syntaktisch korrekt nach der Grammatik
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

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

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

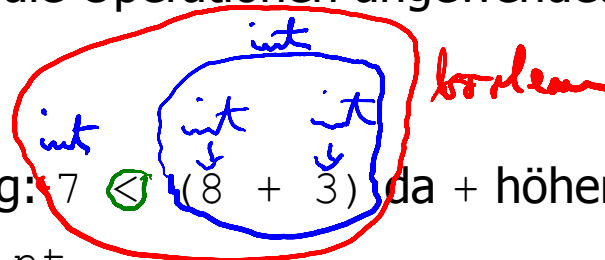
Typüberprüfung von Ausdrücken

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 = *Identifizier*

- 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*]

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.

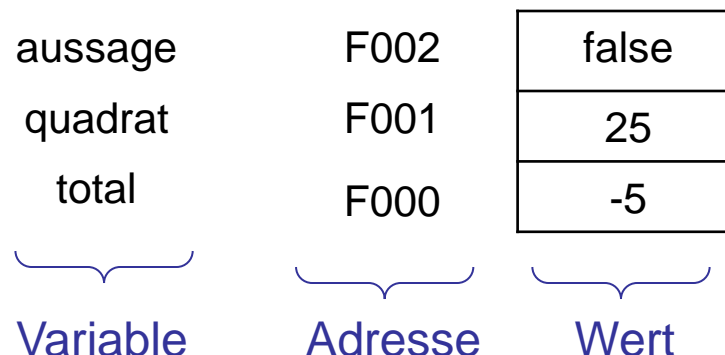
Beispiel:

Abstrakte Darstellung eines Zustand σ :

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

Im Speicher:

aussage	F002	false
quadrat	F001	25
total	F000	-5



Variable Adresse Wert

Auswertung von Ausdrücken

Gegeben sei ein typkorrekt 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 nochmals von links nach rechts durchgehen und, unter Berücksichtigung der Klammern, die Operationen auswerten. Der Wert von Variablen ist dabei durch den Zustand σ bestimmt.

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

Auswertung von `7 < x + 3` unter σ :

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

Beachte:

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