II.1.2 Einfache Datentypen

Montag, 24. Oktober 2016 17:10

- Für jede Variable muss der dazugehörige Datentyp deklariert werden. Var kann erst nach ihrer Deklaration im Block ihrer Deklaration benutzt werden:

```
{
  ⋮
  int x;        ← Hier kann die Var. x benutzt werden.
  ⋮
}
```

`int x;` bedeutet: reserviere Speicherplatz mit Größe für einen int-Wert

`x=10;`

Ganze Zahlen

Typen: byte, short, int, long

Hiermit lassen sich die ganzen Zahlen von $\{-2^{8\cdot n-1}, \dots, 2^{8\cdot n-1}-1\}$ darstellen, wobei

$n=1$ bei byte, d.h. $\{-2^7, \dots, 2^7-1\} = \{-128, \dots, 127\}$ ← benötigt 8 Bit = 1 Byte

$n=2$ bei short ← benötigt 16 Bit = 2 Byte

$n=4$ bei int, d.h. $\{-2^{31}, \dots, 2^{31}-1\}$ ← benötigt 32 Bit = 4 Byte

$n = 2$ bei short ← benötigt 16 Bit = 2 Byte
$n = 4$ bei int, d.h. $\{-2^{31}, \dots, 2^{31}-1\}$ ← benötigt 32 Bit = 4 Byte
$n = 8$ bei long

Grund für die Zahlenbereiche: Zahlen werden im Dualsystem repräsentiert, wobei zur Darstellung negativer Zahlen das Zweierkomplement benutzt wird.

Mit 3 Bit kann man die Zahlen von $\{\underbrace{-2^2}_{-4}, \dots, \underbrace{2^2-1}_{3}\}$ darstellen.

	2^1	2^0
3	0	1
2	0	0
1	0	1
0	0	0
-1	1	1
-2	1	0
-3	1	1
-4	1	0

Mit m Bit kann man $\{-2^{m-1}, \dots, 2^{m-1}-1\}$ darstellen:

Stelle die negative Zahl $-z$ dar als $2^m - z$

Bsp: Darstellg. von -4 als $2^3 - 4 = 4$

Darstl. von -3 als $2^3 - 3 = 5$

z.B. Darstlg. von -2:

Darst. von 2: 0 1 0

⇓ vertausche 0 und 1

1 0 1

⇓ addiere 1

1 1 0

Alternative:

Um $-z$ darzustellen,
nimm die Dualzahl für z,
vertausche 0 und 1,
addiere 1.

Was passiert bei Überschreitung des Wertebereichs?

$2^{31}-1$

int x = 2 147 483 647; | 0 1 1

```
int x = 2 147 483 647;
int y = 1;
System.out.print(x+y);
```

```
  0 1 ..... 1
+           1
-------------
  1 0 .... 0 0
```

Ergebnis ist $\underbrace{-2\,147\,483\,648}_{-2^{31}}$

- Operationen auf ganzen Zahlen: +, −, *, /, % (modulo)

  / : Ganzzahl-Division, Rest wird abgeschnitten

- Operationen linksassoziierend: 5 − 4 − 3 ist −2
- +, −, ... auf byte oder short liefert int
- long-Zahlen werden durch nachgestelltes l deutlich gemacht

  5l
  5L

Gleitkommazahlen

Datentypen float und double

float: 32 Bit, double: 64 Bit

Gleitkommazahlen haben die Form

Vorzeichen · Mantisse · 2^{Exponent} ← im Rechner

Bspe: -1.5

1.0 } gleich
1. }

0.5 } gleich
.5 }

1.2e-23 $\hat{=}$ $1{,}2 \cdot 10^{-23}$

1.5f ← Typ float

gleich { 1.5d ← Typ double
1.5

Wahrheitswerte

- Datentyp boolean
  mit 2 Elementen true und false
- Vordef. Operationen

&&	und
||	oder
!	nicht

Bsp: boolean x = true || false;
System.out.print(x); ← gibt true aus

&& und || werden "lazy"

aus, d.h., es wird zunächst das 1. Argument ausgewertet. Das 2. Arg. wird nur ausgewertet, falls das Ergebnis noch nicht feststeht.

Weitere vordef. Operationen mit Erg. v. Typ boolean:

== Gleichheit

!= Ungleichheit

<, >, <=, >= etc.

x = y Anweisung Zuweisung, x bekommt den Wert von y

x == y Ausdruck true oder false

Zeichen

Datentyp char

Zeichen werden in Java in Apostrophe gesetzt, d.h.

'a', ..., 'A', 'ü', ...,

'$', '1', '@',

'\n', ...

↑ Steuerzeichen, \n steht für newline

Java erlaubt 2^{16} ver-

Schiedene Zeichen (Unicode)

Vergleichsoperatoren für Zeichen:

==, !=

<, >, <=, >= : vergleichen Zeichen anhand ihrer Nummer im Unicode

z.B. 'a' hat Unicode 97

'b' — " — 98

boolean x = 'a' < 'b'; ← x bekommt den Wert true

Strings

- Kein primitiver Datentyp, Objekte sind aus char-Zeichen zusammengesetzt
- Vereinfachte Schreibweise mit " ... ": "hallo"
- Konkatenation mit +
- Vergleich von 2 Strings str_1, str_2
  - str_1 == str_2 nicht empfehlenswert
  - str_1.equals(str_2)

besser
(genaue Erklärung später)

Typkonversion

Java ist eine getypte Sprache,
d.h. jeder Ausdruck hat einen Typ.
Funktionen können nur auf Argumente d. passenden
Typen angewendet werden.

z.B. `2 + 3` ✓ (int, int)

`true + false` ↯ (boolean)

`2.2 + 3.5` ✓ (double)

`"hal" + "lo"` ✓ (String)

`2 + 3.5` (int, double)

erlaubt: Hierzu wird 2 zunächst
in eine double-Zahl 2.0
umgewandelt und dann
2.0 + 3.5 berechnet.

Java verwendet eine
implizite Datentypanpassung
vom Untertyp (spezielleren Typ)
zum Obertyp (allgemeineren Typ).

Bsp `int x = 'a' + 1;` (char, int) ↖ x hat den Wert 98

Man kann Typkonversionen auch explizit erzwingen:

(int) 9.5
ergist 9

wird auch als Type Cast bezeichnet.

```
int X = 'a' + 1;
char Y = (char) X;   ← Y hat den Wert 'b'
```

1.0f — wird implizit in 2.0f konvertiert

(float) 1 / 2 ergist 0.5f

(float) (1 / 2) ergist 0.0f

(int, int) → int

Eine Operation, die Argument v. Typ t_1 erwartet, darf auch auf Arg. v. Typ t_2 angewendet werden, falls es implizite Typkonversion von t_2 nach t_1 gibt.