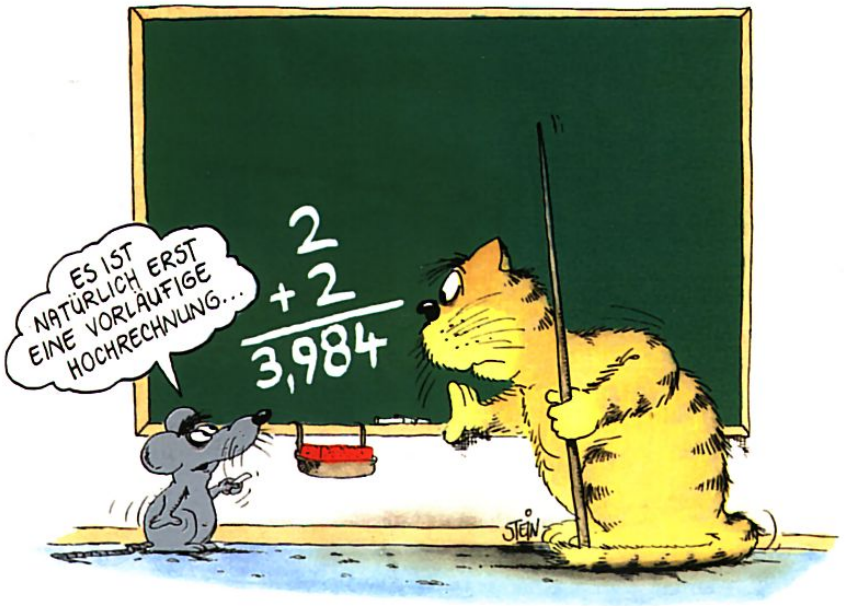
Grundlagen der Programmierung

VL 03: Einfache Datentypen

Prof. Dr. Samuel Kounev,
M.Sc. Norbert Schmitt



Danksagung

- Vorlesungsmaterialien von Prof. Dr. Detlef Seese wurden als Basis verwendet
- Unterstützung bei der technischen und inhaltlichen Gestaltung des Vorlesungsmaterials leisteten:

  Jóakim v. Kistowski

  Dietmar Ratz, Joachim Melcher, Roland Küstermann, Jana Weiner, Hagen Buchwald, Matthes Elstermann, Oliver Schöll, Niklas Kühl, Tobias Diederich

Inhalt

- Datentypen und ihre Literale
- Zuweisungen und Typumwandlungen
- Ausdrücke und Operatoren
- Auswertungsreihenfolge
- Konversionsregeln
- Konstanten

Einfache Datentypen - Grundfragestellungen

- Warum ist es sinnvoll Typen von Daten zu unterscheiden?
- Was sind einfache Datentypen?
- Wie müssen wir mit diesen Datentypen umgehen?

Datentypen (data types)

- Jedem deklarierten Bezeichner muss genau ein **Typ** zugeordnet sein

➔ Dadurch wird implizit festgelegt:

- welche Operationen für den Bezeichner definiert sind
- wie viel Speicherplatz zu reservieren ist
- welche Werte dem jeweiligen Speicherinhalt entsprechen

Datentypen (2)

- Mit jedem Typ ist ein Wertebereich (domain) festgelegt
  - Menge der Werte, welche eine Variable des Typs annehmen kann
- Java/C sind Sprachen mit strenger Typprüfung (type-checking)
  - Jeder Variable und jedem Ausdruck ist ein zur Übersetzungszeit bekannter Typ zugeordnet
- Generell wird zw. “strongly typed” und “weakly typed” Programmiersprachen unterschieden
  - Keine einheitliche Definition
    - siehe http://en.wikipedia.org/wiki/Strong_and_weak_typing

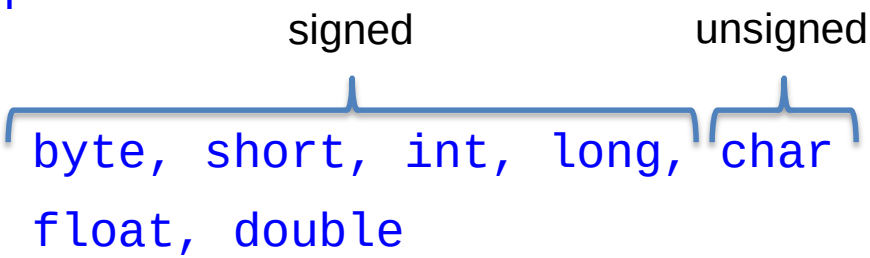
Datentypen (3)

Java

- Java unterscheidet

1. Einfache (elementare) Datentypen (primitive data types)

- der logische Typ `boolean`
- numerische Typen
  - ganzzahlige Typen
  - Gleitkommatypen



2. Referenztypen (reference types)

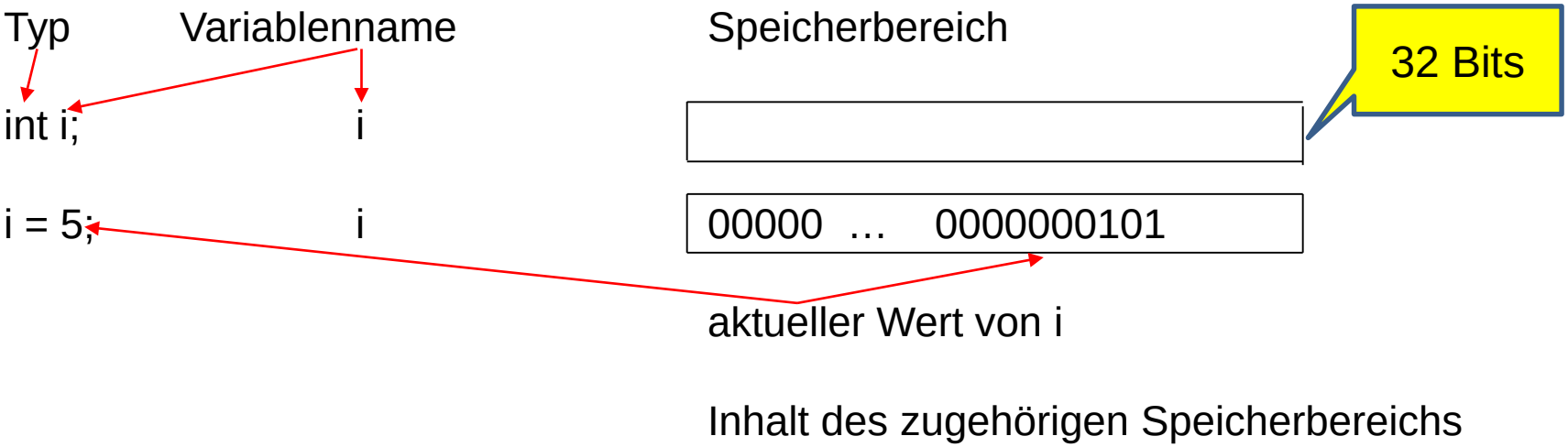
- Klassen, Interfaces und Felder (später)

- Bemerkung
  - `char` ist gleichzeitig auch ein Zeichentyp
  - Die Klasse `String` spielt eine Sonderrolle für Zeichenketten

Datentypen (4)

Java

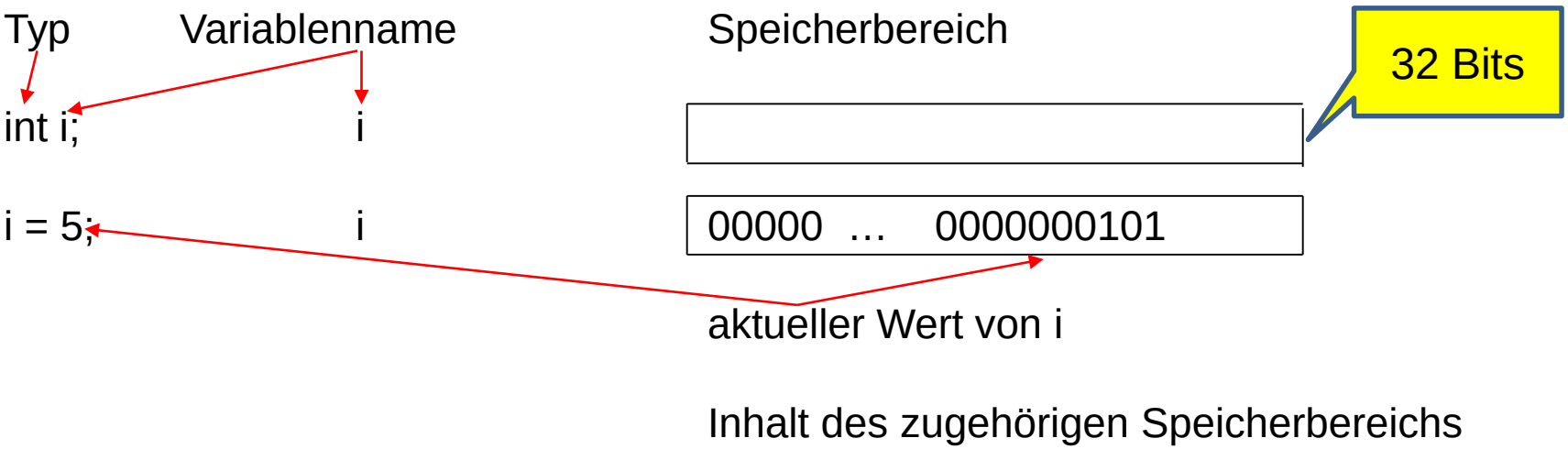
Einfache (elementare) Datentypen



Datentypen (4)

Java

Einfache (elementare) Datentypen



Referenztypen (z.B. Klassen, Interfaces und Felder)

Ganzzahlige Datentypen

Java

- Ganzzahlige Datentypen (integer data types)

Typname	kleinster Wert	größter Wert	Länge
`byte`	`-128`	`127`	8 Bits
`short`	`-32768`	`32767`	16 Bits
`int`	`-2147483648`	`2147483647`	32 Bits
`long`	`-9223372036854775808`	`9223372036854775807`	64 Bits

- Ganzzahlige Literalkonstanten
  - für `byte`, `short` und `int` Ziffernfolgen der Form: `8` `123` `16787`
  - für `long` Ziffernfolgen mit l oder L am Ende: `17L` `29874987349L`

- Darstellung ganzzahliger Datentypen
  - Binär-, Oktal- bzw. Hexadezimal-Darstellungen sind möglich → siehe Anlage
    - Verwendung von 0b bzw. 0B (für Binär), 0 (für Oktal) bzw. 0x (für Hexa) am Anfang des Literals

Java 7 erlaubt bei der Literaleingabe auch noch den Unterstrich _ als Trennzeichen.

Gleitkommatypen

C/C++ Java

- für Fließkommazahlen / Gleitpunktzahlen (floating point numbers)
- Java realisiert hier den IEEE-754-Standard für die Typen `float` (einfache Genauigkeit realisiert mit 4 Byte) und `double` (doppelte Genauigkeit realisiert mit 8 Byte)
- Darstellung von Gleitkommatypen → siehe Anlage
- Gleitkommatypen (Länge in C/C++ ist Plattform-abhängig):

Typname	größter positiver Wert	kleinster positiver Wert	Länge
float	≈3.40282347E38	≈1.4E-45	32 Bits
double	≈1.79769313486231 57E308	≈4.9E-324	64 Bits

Gleitkommatypen (2)

Java

C/C++

- Gleitkomma-Literalkonstanten
  - bestehen aus Mantisse Exponential-Anteil Typ-Suffix
  - nur die Mantissen-Angabe ist zwingend
  - für `float`

    `1.5f` `1.f` `.35f` `1e7f` `1.3e7f`
  - für `double`

    `1.5` `1.` `.35` `1e7` `1.3e7d`

    e bzw. E mit anschließender Zahl X steht für die Multiplikation mit 10^X

    Beispiele: 1.78e4 entspricht $1.78*10^4 = 1.78*10000 = 17800$

    2.4e-3 entspricht $2.4*10^{-3} = 2.4*0.001 = 0.0024$
- Negative Zahlen werden erzeugt, indem man vor die entsprechende Zahl ein Minuszeichen setzt. Dies ist dann ein Ausdruck und kein Literal.
- Achtung: Mit 32 oder auch 64 Bits kann nicht jede Zahl zwischen +3.4028235E38 und -3.4028235E38 exakt dargestellt werden ➔ Rundungsfehler

Gleitkommatypen (3)

Java

C/C++

- Ein Gleitkommaliteral ist vom Typ double, es sei denn es endet mit f oder F. In diesem Fall besitzt es den Typ float.

Beispiel:	`3.141592E38f`	➔ Typ float
	`3.141592e40F`	➔ Typ float
	`3.141592E300`	➔ Typ double
	`3.141592e300d`	➔ Typ double
	`3.141592e300D`	➔ Typ double
	`.0E+3d`	➔ Typ double

- Aber nachfolgend liegen keine Gleitkommaliterale vor:

0,235	3.0d+3
3.5.7	22.+2
+.17	-1.43

Die meisten sind Ausdrücke

Logischer Datentyp

- Logischer Datentyp
  - Typname **`boolean`** (Java)
  - Typname **`bool`** (C++)
  - Länge ist typischerweise 1 Byte
  - Literalkonstanten: **`false`** und **`true`**

Java

C++

In C gibt es generell keinen Typ „boolean“.

Es wird meistens integer verwendet (0 = false, 1 = true).

C

Zeichen-Datentypen

Java

C/C++

- Zeichen-Datentypen (character data types)
  - einzelne Zeichen
    - Typname **`char`**
    - Länge 16 Bits (2 Bytes in Java, meist 1 Byte in C)
    - Literalkonstanten (in Hochkommas eingeschlossen)

```
'a'      'x'      '+'      'ä'
'\n'     Ersatzdarstellung für 'Zeilenende'
'\t'     Ersatzdarstellung für 'Tabulator'
'\''     Ersatzdarstellung für '''  (da unzulässig)
'\"'     Ersatzdarstellung für '"'  (aber zulässig)
'\\'     Ersatzdarstellung für '\'  (da unzulässig)
```

Zeichen-Datentypen (2)

Java

`'\u007B'` Ersatzdarstellung für das Zeichen mit der Nummer **`007B`** (hexadezimal) im **Unicode** (ab C11 auch in C)

Wertebereich des Typs char

`\u0000` bis `\uffff` — 65536 Zeichen, siehe Anlage

Unicode: entsteht aus einer 4-stelligen Hexadezimalzahl durch Voranstellen von `\u`

`'\u0061'` entspricht dem Zeichen `'a'`

`'\u007B'` entspricht dem Zeichen `'{'`

`'\u0020'` entspricht dem Leerzeichen `' '`

Inwischen, ab Version 2.0 enthält Unicode wesentlich mehr Zeichen, die nicht alle in Java realisiert sind

Details siehe Unicode Consortium: http://unicode.org/

Zeichen-Ketten (Strings)

Java

C++

- eigentlich kein einfacher Datentyp, siehe später: **Klasse `String`**
- Literalkonstanten (in Doppelhochkommas eingeschlossen)

```
"HalliHallo"
"öakljölkj"
"Ach du lieber Himmel"
"Ach\ndu\nlieber\nHimmel„
```

(eigentlich mehrzeilig)

C

Bemerkung: In C gibt es keine Strings!

Es weren Felder (Arrays) vom Typ **`char`** verwendet.

Default-Werte für einfache Datentypen

Java

Bisher haben wir nur lokale Variablen in der Methode **`main`** verwendet.
Für diese muss stets explizit ein Wert festgelegt werden.

In vielen Fällen wird später Variablen automatisch ein Wert zugewiesen, auch wenn keine explizite Wertzuweisung erfolgt - siehe später **Instanzvariablen**.

Solche Werte werden Default-Werte genannt. Für die bisher behandelten Typen sind die folgenden Default-Werte festgelegt:

Default-Werte

- 0 bei numerischen Datentypen
- (ganzzahlig 0, Fließkomma 0.0)
- **`false`** bei **`boolean`**
- \u0000 bei **`char`**

Bemerkung: In C gibt es keine Default-Werte!

C

Inhalt

- Datentypen und ihre Literale
- Zuweisungen und Typumwandlungen
- Ausdrücke und Operatoren
- Auswertungsreihenfolge
- Konversionsregeln
- Konstanten

Zuweisung, Typumwandlung

Java

- Beispielhafte Deklarationen mit Initialisierungen

```
byte b = 5;
short s = 50;
int i = 500;
long l = 5000;
float f = 1.5f;
double d = 2.5;
char c = ‘a';
boolean x = true;
```

char – 16 Bits

- Frage: Welche der nachfolgenden Zuweisungen ist (un-)zulässig?

```
i = b; //   erlaubt
f = l; //   erlaubt aber Datenverlust möglich
i = c; //   erlaubt
s = f; //   unzulaessig!
i = l; //   unzulaessig!
i = x; //   unzulaessig!
```

Typumwandlung (type cast / type casting)

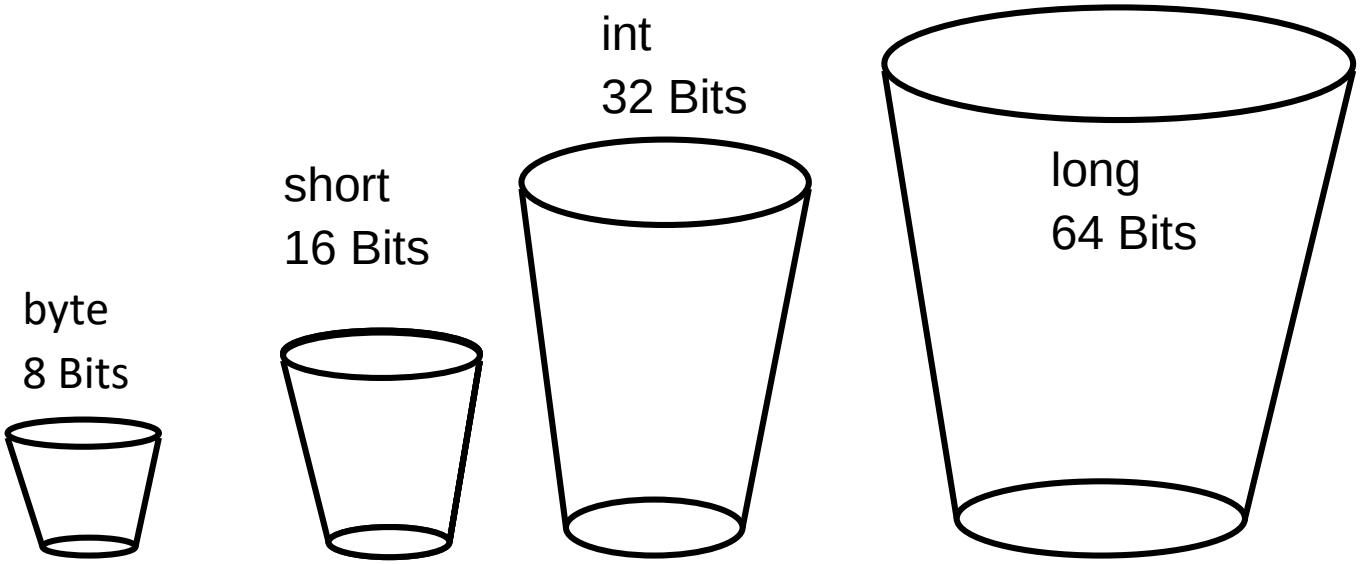
Java

Typen kann man sich als Becher, oder auch als Schablonen vorstellen.

Aber: Wenn Sie den Inhalt eines großen Bechers in einen kleinen schütten, so geht oft etwas verloren.

Die Größe eines Bechers bezieht sich auf den Wertebereich (value range).

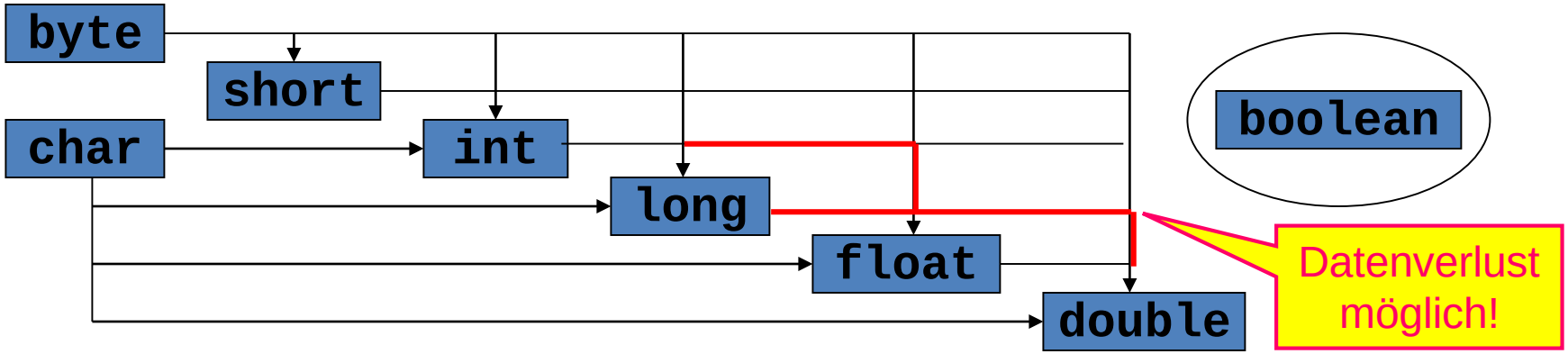
Beachten Sie, dass **`int`** und **`float`** beide 32 Bits haben und **`long`** und **`double`** beide 64 Bits haben. Die Darstellung der betreffenden Zahlen ist aber eine andere – Idee Schablone



Typumwandlung (type cast / type casting)

Java

1. **Regel für die automatische (implizite) Typumwandlung**



2. **Explizite Typumwandlung** (eventuell mit Datenverlust!)
   - durch Voranstellen des eingeklammerten Typnamens
   - in den Beispielen:

```
s = (short) f;
i = (int) l;
i = (int) x; // unzulässig, auch umgekehrt!
```

Inhalt

- Datentypen und ihre Literale
- Zuweisungen und Typumwandlungen
- Ausdrücke und Operatoren
- Auswertungsreihenfolge
- Konversionsregeln
- Konstanten

Ausdrücke und Operatoren

Java

C/C++

- Beispiel für einen Ausdruck:
  - `3.5 * x - 2 / (4 + y)`
- Ein Ausdruck (expression) setzt sich zusammen aus
  - Operanden (Konstanten, Variablen, Methodenaufrufe)
  - Operatoren (+, -, *, ...)

  und *liefert stets einen Wert ab*!
- Abhängig vom *Typ der Operanden* sind nur bestimmte Operatoren zulässig, z.B.
  - arithmetische Operatoren für die *numerischen Datentypen*
  - Vergleichs-Operatoren für die *numerischen Datentypen*
  - logische Operatoren für den *logischen Datentyp*
  - der Operator + für Zeichenketten (*Typ* `String`)
- Man unterscheidet
  - einstellige (monadische, unäre) Operatoren (z.B. +, -, ++, --)
  - zweistellige (dyadische, binäre) Operatoren (z.B. +, *, &&, <, <=)
  - dreistellige (triadische, ternäre) Operatoren (nur einer: ?:)

Stellung und Schreibweise

Java

C/C++

1. **Präfix Notation**

   <Operator><Operand>

   Beispiel: **-a**

2. **Postfix Notation**

   <Operand><Operator>

   Beispiel: **n++**

3. **Infix Notation**

   <Operand><Operator><Operand>

   Beispiel **a+b**

   oder

   <Operand>**?**<Operand>**:**<Operand>

   Beispiel: bedingte Zuweisung 3-stellig **`max = (a>b)?a:b;`**

Ausdrücke und Operatoren

Java

- Elementare (also einfachste) Ausdrücke sind
  - nur ein Operand und kein Operator
    - z.B.
      - `x`
      - `1.74`
      - `Math.PI`
      - `Math.E`
  - nur ein einfacher Methodenaufruf
    - z.B.
      - `Math.sin(x)`
      - `Math.sqrt(2)`

- Die Argumente von Methodenaufrufen können natürlich selbst wieder Ausdrücke sein

- Elementarfunktionen der Mathematik (z.B. `exp`, `sin`, `cos`, `tan`, ...) sowie Näherungen für die Konstanten *e* und π sind in Java in die Klasse `Math` ausgelagert

Ausdrücke und Operatoren (2)

Java

Methoden aus der Klasse Math

Funktion	*Argument-Typ*	*Ergebnis-Typ*	*mathem. Bedeutung*
`abs(x)`	`double`	`double`	Betrag
	`float`	`float`	
	`int`	`int`	
	`long`	`long`	
`acos(x)`	`double`	`double`	Arcus Cosinus
`asin(x)`	`double`	`double`	Arcus Sinus
`atan(x)`	`double`	`double`	Arcus Tangens
`atan2(x,y)`	`double`	`double`	`arctan(x/y)`
`cos(x)`	`double`	`double`	Cosinus
`exp(x)`	`double`	`double`	e^x
`log(x)`	`double`	`double`	`ln x`
`pow(x,y)`	`double`	`double`	x^y
`sin(x)`	`double`	`double`	Sinus
`sqrt(x)`	`double`	`double`	Quadratwurzel
`tan(x)`	`double`	`double`	Tangens
`max(x,y)`	`double`	`double`	Maximum
	`float`	`float`	
	`int`	`int`	
	`long`	`long`	
`min(x,y)`	(analog)		Minimum

Einstellige numerische Operatoren

Java

C/C++

- Die Operatoren:
  - `+` Identität (ohne Wirkung)
  - `-` Negation
  - `++` Inkrement (Erhöhung um 1)
  - `--` Dekrement (Erniedrigung um 1)
  - Verwendung von `+`, `-`, `++` und `--` in Präfix-Notation (vor-gestellt)
  - Verwendung von `++` und `--` in Postfix-Notation (nach-gestellt)

- Beispiele für `int x = 5`

`x` vor der Auswertung	5	5	5	5	5	5
Ausdruck	`+x`	`-x`	`++x`	`--x`	`x++`	`x--`
Wert des Ausdrucks	5	-5	6	4	5	5
`x` nach der Auswertung	5	5	6	4	6	4

Einstellige numerische Operatoren (2)

Java

C/C++

- Beispiel
  - Programmfragment:

```
int a, b, c, d;
a = 5;
a++;
b = ++a;
c = a++;
d = a--;
a = a++;
```

```
a ← 5
a ← 6
a ← 7, Wert rechts: 7, b ← 7
Wert rechts: 7, c ← 7, a ← 8
Wert rechts: 8, d ← 8, a ← 7
Wert rechts: 7, a ← 7, a ← 8
```

  - Welche Werte haben a b c d am Ende?

- Für Gleitkommatypen analog

Zweistellige numerische Operatoren

Java

C/C++

- Die Operatoren:

`+`	Addition
`-`	Subtraktion
`*`	Multiplikation
`/`	Division
`%`	Rest (bei Division mit ganzzahligem Ergebnis)

- Beispiele

```
– int i = 7 / 3;                  // ==> i =
– int j = 7 % 3;                  // ==> j =
– double d = 8.2 / 4.0;           // ==> d =
– double e = 8.2 % 4.0;           // ==> e =
– int x = 5;
– int y = (x++) + (x++);          // ==> y =
                                  // ==> x =
```

2

1

2.05

0.2
= 0.1999999999999993

11

7

Zweistellige numerische Operatoren (2)

Java

- Der Operator + für Strings
  - bewirkt das Aneinanderhängen von Zeichenketten

- Zusammengesetzte Zuweisungsoperatoren
  (compound assignment operators)
  - zur Abkürzung von Zuweisungen mit einfachen Operationen
    - die Anweisung `a += b;` // entspricht der Anweisung `a = a + b;`
    - die Anweisung `x |= y;` // entspricht der Anweisung `x = x | y;`
  - **Vorsicht:** Die Lesbarkeit eines Programms verschlechtert sich!

Vergleichs-Operatoren

Java

C/C++

- Die Operatoren:
  - `<` Test auf "kleiner"
  - `>` Test auf "größer"
  - `<=` Test auf "kleiner oder gleich"
  - `>=` Test auf "größer oder gleich"
  - `==` Test auf "gleich"
  - `!=` Test auf "ungleich"

  – Beispiel:

Java

```
int x = …;
boolean b = (x <= 5);
System.out.println("5.2 <= 4 is " + (5.2 <= 4));
```

    - Ausgabe: `5.2 <= 4 is false`

Logische Operatoren

Java

C/C++

- Die Operatoren:

&	logisches UND
|	logisches ODER
^	logisches exklusives ODER (entweder oder)
&&	logisches UND
||	logisches ODER
!	logische NEGATION (Nicht)

  - Doppelzeichen **&&** und **||** ermöglichen vorzeitigen Abbruch der Ausdrucks-Auswertung, sobald das Ergebnis feststeht
  - Beispiel:

Java

```
int a = InputHelper.readInteger("a = ");
int b = InputHelper.readInteger("b = ");
boolean wahrOderFalsch = (a != 0) & (b/a > 5);
```

Logische Operatoren

Java

C/C++

- Die Operatoren:

&	logisches UND
|	logisches ODER
^	logisches exklusives ODER (entweder oder)
&&	logisches UND
||	logisches ODER
!	Logische NEGATION (Nicht)

  - Doppelzeichen **&&** und **||** ermöglichen vorzeitigen Abbruch der Ausdrucks-Auswertung, sobald das Ergebnis feststeht
  - Beispiel:

Java

```
int a = InputHelper.readInteger("a = ");
int b = InputHelper.readInteger("b = ");
boolean wahrOderFalsch = (a != 0) & (b/a > 5);
```

    - Was passiert bei Eingabe von **0** für **a**?

      Programmabsturz wegen Division durch 0

    - Was passiert, wenn statt **&** ein **&&** verwendet wird?

      alles ok: (**a!=0**) liefert **false**, und (**b/a>5**) wird nicht ausgewertet

Logische Operatoren (2)

&	logisches UND	
	`true`	`false`
`true`	`true`	`false`
`false`	`false`	`false`

&&	logisches UND	
	`true`	`false`
`true`	`true`	`false`
`false`	`false`	`false`

Unterschied:

A & B : beide Ausdrücke A und B werden ausgewertet

A && B : liefert **A** den Wert `false`,
so wird **B nicht mehr ausgewertet**

Logische Operatoren (3)

	logisches ODER	
	`true`	`false`
`true`	`true`	`true`
`false`	`true`	`false`

		logisches ODER	
		`true`	`false`
`true`		`true`	`true`
`false`		`true`	`false`

Unterschied:

A | B : beide Ausdrücke A und B werden ausgewertet

A || B : liefert A den Wert `true`,
so wird **B nicht mehr ausgewertet**

Logische Operatoren (4)

^	exklusives ODER	
	true	false
true	false	true
false	true	false

|	logisches ODER	
	true	false
true	true	true
false	true	false

!	logische NEGATION
true	false
false	true

Bitoperatoren…

- …lassen ganzzahlige Operanden zu, arbeiten aber nicht mit dem ganzzahligen, eigentlichen Wert der Operanden, sondern nur mit deren Bits
- **Anwendung**
  - bei der Arbeit mit Bitketten in selbstdefinierten Datentypen

- Die **Negation** ~ liefert bitweise stets das Komplement des Operanden

a	~a
0	1
1	0

Beispiel:

```
byte a;
a = -55;
```

1	1	0	0	1	0	0	1	a
0	0	1	1	0	1	1	0	~a

a = -55 = -128 + 64 + 8 + 1

~a = 54 = 32 + 16 + 4 + 2

Bitoperatoren (2)

- Neben der Negation existieren noch drei binäre Operationen: **Und** (**&**), **Oder** (**|**) und **exklusives Oder** (**^**)

a	b	a&b	a|b	a^b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

- Bei den bitweisen Verknüpfungen zweier ganzzahliger Operanden werden diese Bitoperationen auf mehrere Bits (**Stelle für Stelle**) **gleichzeitig** angewendet

Bitoperatoren (3)

Java

C/C++

Beispiel:

```
byte a, b;
a = 11;                        //     00001011 =    11
b = 13;                        //     00001101 =    13
System.out.println(~a);        //     11110100 =    -12
System.out.println(a&b);       //     00001001 =    9
System.out.println(a|b);       //     00001111 =    15
System.out.println(a^b);       //     00000110 =    6
```

?

Grund: Zweierkomplement-Darstellung (siehe Anhang)
-128+64+32+16+4 = -12

Bitoperatoren (4)

- Daneben existieren noch drei Schiebeoperatoren, die alle Bits eines ganzzahligen Wertes um eine vorgegebene Anzahl von Stellen nach links bzw. rechts schieben

  a << b Schiebt die Bits in a um b Stellen nach links und füllt mit 0-Bits auf

  ```
  i = 2 << 3 // entspricht für int i gleich 16
  ```

  a >> b Schiebt die Bits in a um b Stellen nach rechts und füllt mit dem höchsten Bit von a auf

  ```
  i = -1>>6 // entspricht für int i = -1
  ```

  a >>> b Schiebt die Bits in a um b Stellen nach rechts und füllt mit 0-Bits auf

  ```
  i = -1>>>30 // entspricht für int i gleich 3
  ```

- Das Schieben um eine Stelle nach links bzw. rechts entspricht dabei einer Multiplikation bzw. Division des Wertes mit bzw. durch 2

Zuweisung als Ausdruck

Java

C/C++

- Die Wertzuweisung als Ausdruck
  - Normalerweise ist die Wertzuweisung eine Anweisung

```
a = b + 3 / c;
```

Variable — Ausdruck

  - In Java: Die Zuweisung ist selbst wieder ein Ausdruck. Der Wert eines Zuweisungsausdrucks ist der neue Wert der jeweils linken Seite

  - Daher ist folgende **Mehrfachzuweisung** erlaubt:

```
d = e = f = g = h = a = b + 3 / c;
```

Variable

Ausdruck

Ausdruck

Ausdruck

Inhalt

- Datentypen und ihre Literale
- Zuweisungen und Typumwandlungen
- Ausdrücke und Operatoren
- Auswertungsreihenfolge
- Konversionsregeln
- Konstanten

Auswertungsreihenfolge

Java

C/C++

- Regeln für die Auswertungsreihenfolge in Ausdrücken
  - Operatoren mit höherer Priorität *vor* Operatoren mit niedrigerer Priorität
  - mehrere zweistellige Operatoren mit gleicher Priorität von links nach rechts (links-assoziativ)
  - mehrere einstellige Operatoren und Zuweisungsoperatoren von rechts nach links (rechts-assoziativ)
  - elementare Ausdrücke und Ausdrücke in Klammern *vor* den Operatoren, die diese weiterverknüpfen
  - linker Operanden-Ausdruck immer *vor* dem rechten Operanden-Ausdruck
  - Beispiel: Wie wird der folgende Ausdruck ausgewertet?

Java

```
Math.sqrt(3.5+x)*5/3-(x+10)*(x-4.1) < 0
```

Operator-Prioritäts-Stufen

Java

C/C++

Postfixoperatoren	`[] . () ++ --`
einst. Operatoren	`++ -- + - ~ !`
Erzeugung/Cast	`new (Typname)`
multiplikativ	`* / %`
additiv	`+ -`
shift	`<< >> >>>`
relational	`< > <= >= instanceof`
Gleichheit	`== !=`
logisches/bitweises UND	`&`
logisches/bitweises XOR	`^`
logisches/bitweises ODER	`|`
logisches UND	`&&`
logisches ODER	`||`
Bedingung	`? :`
Zuweisung	`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

höhere

niedrigere

Merkregel: einstellig vor multiplikativ vor additiv vor vergleichend vor logisch vor zuweisend

Konversionsregeln in Ausdrücken

Java

- `String`-Konversion
  - Ist `s` vom Typ `String` und `x` von einem beliebigen anderen Typ, so wird bei Auswertung des Operators + in

    `x + s` oder `s + x`

    `x` implizit nach `String` gewandelt

- Numerische Typ-Konversion
  - Werden zweistellige Operatoren (z.B. `+`, `-`, `*`, `<`, `==`) auf numerische Operanden angewandt, so wird wie folgt vorgegangen:
    - Falls ein Operand vom Typ `double` ist, so wird der andere nach `double` konvertiert
    - Andernfalls, falls ein Operand vom Typ `float` ist, so wird der andere nach `float` konvertiert
    - Andernfalls, falls ein Operand vom Typ `long` ist, so wird der andere nach `long` konvertiert
    - Andernfalls werden beide Operanden nach `int` konvertiert

Konversionsregeln in Ausdrücken (2)

Java

Welche der nachfolgenden Zeilen sind fehlerhaft? Welche Werte haben die Variablen auf der linken Seite in den korrekten Zeilen?

```
int      v1     = 3/2;
double   v2     = 3/2;
double   v3     = (3/2)*2;
double   v4     = (3.0/2)*2;
double   v5     = 3.0/2;
int      v6     = 3/2.0;
int      v7     = (int)3/2.0;
int      v8     = (int)3.0/2;
int      v9     = (int)(3/2.0);
float    v10    = 3.0/2.0;
short    v11    = 2;
short    v12    = v11 + v11;
int      v13    = true;
int      v14    = 'x' + 3;
String   v15    = "Hilfe" + true + 3;
String   v16    = true + 3 + "Hilfe";
```

```
v1 ← 1
v2 ← 1.0
v3 ← 2.0
v4 ← 3.0
v5 ← 1.5
unzulässig!
unzulässig!
v8 ← 1
v9 ← 1
unzulässig!
v11 ← 2
unzulässig!
unzulässig!
v14 ← 123
v15 ← "Hilfetrue3"
unzulässig!
```

Konstanten

Java

Syntaxregel für `final`-Variablen / symbolische Konstanten

`final` <VARIABLENTYP> <VARIABLENBEZEICHNER > = <AUSDRUCK>;

Beispiel:

```
final double WECHSELKURS = 0.723;
```

Unzulässig wäre dann: **`WECHSELKURS = 1.47;`**

Konstanten (2)

C/C++

Syntaxregel für `const`-Variablen

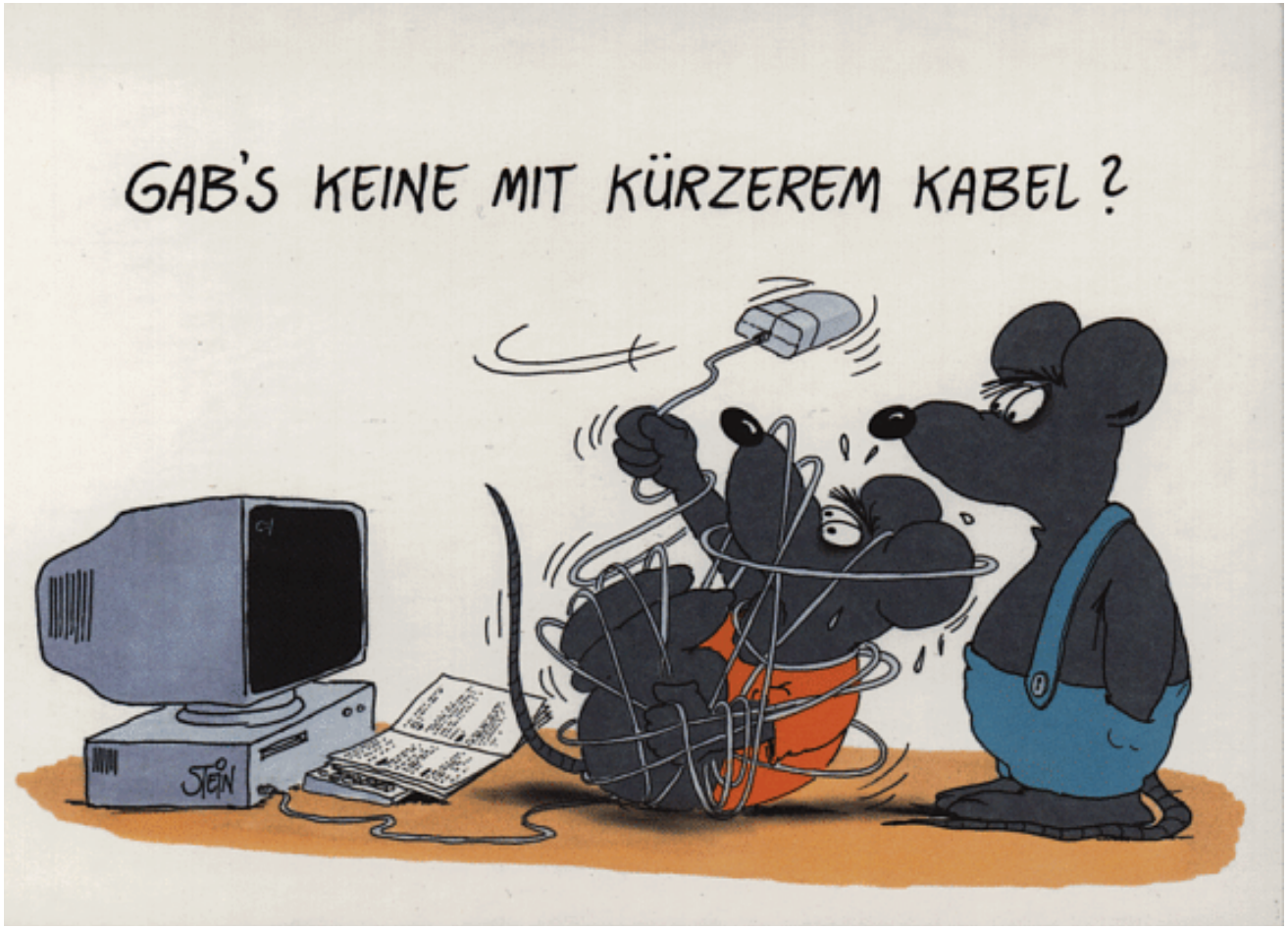
`const` <VARIABLENTYP> <VARIABLENBEZEICHNER > = <AUSDRUCK>;
oder
<VARIABLENTYP> `const` <VARIABLENBEZEICHNER > = <AUSDRUCK>;

Beispiel:

```
const double WECHSELKURS = 0.723;
```

Unzulässig wäre dann: **`WECHSELKURS = 1.47;`**

Fragen?



Anlage

Darstellung ganzzahliger Datentypen

`'\u007B'` Ersatzdarstellung für das Zeichen mit der Nummer **`007B`** (hexadezimal, dezimal = 123) im **Unicode**

Wertebereich des Typs char **`\u0000`** bis **`\uffff`**

Darstellung ganzzahliger Datentypen (2)

Einschub Binär-, Oktal- und Hexadezimalsystem

Erinnerung Dezimalsystem:

Ziffern

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

$12345 = 1*10^4+2*10^3+3*10^2+4*10^1+ 5*10^0$

Allgemeine Form, hier für p=10:

$$a_0 a_1 a_2 ... a_{l-1} a_l = \sum_{i=0}^{l} a_i * p^{l-i}$$

Hier also {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Hierbei sind die a_i aus der Menge $\{z_0, z_1, ..., z_{p-2}, z_{p-1}\}$ der Ziffern, gewählt, wobei $a_0 \neq 0$ angenommen wird.

Darstellung ganzzahliger Datentypen (3)

Einschub Binär-, Oktal- und Hexadezimalsystem

Binärsystem p=2

Ziffern 0 und 1.

Beispiel:

Ab Java 7: Binärdarstellung in Java: 0b oder 0B am Anfang. danach nur 0 oder 1 als Ziffern.
Der Binärwert von 361 in Java wäre dann:
0b101101001 oder
0B101101001

$$101101001 = 1*2^8 + 0*2^7 + 1*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0$$
$$= 256+0+64+32+0+8+0+1 = 361$$

Binärdarstellung

dezimale Darstellung

Darstellung ganzzahliger Datentypen (4)

Einschub Binär-, Oktal- und Hexadezimalsystem

Oktalsystem p=8

Ziffern

0, 1, 2, 3, 4, 5, 6, 7

dezimale Darstellung

$551 = 5*8^2 + 5*8^1 + 1*8^0 = 5*64 + 40 + 1 = 361$

Oktal in Java **`0551`**

Darstellung ganzzahliger Datentypen (5)

Einschub Binär-, Oktal- und Hexadezimalsystem

Hexadezimalsystem p=16

Ziffern

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

dezimale Darstellung

$1C8 = 1*16^2 + 12*16^1 + 8*16^0 = 256 + 192 + 8 = 456$

hexadezimal in Java `0x1C8`

Darstellung ganzzahliger Datentypen (6)

Wie werden ganze Zahlen gespeichert?

Möglich wäre:

- als eine Folge binärer Zeichen (also 0 oder 1) etwa: 100101
- jede Stelle repräsentiert dabei eine ganzzahlige Potenz der Zahl 2

$1*2^5+0*2^4+0*2^3+1*2^2+0*2^1+1*2^0 =$
$1*32+0*16+0*8+1*4+0*2+1*1 = 37$

So nicht!

➔ der Zahl 37 könnte also die Binärfolge 100101 entsprechen

Aber zur Kennzeichnung des Vorzeichens benötigt man ein weiteres Bit.

Darstellung ganzzahliger Datentypen (7)

Wie werden ganze Zahlen wirklich dargestellt?

- Ganzzahlige Datentypen sind vorzeichenbehaftet und benutzen die **Zweierkomplement-Darstellung (two's complement format)**

- Die binäre Folge $a_{n-1}a_{n-2}...a_1a_0$ repräsentiert in Zweierkomplement-Darstellung die Zahl

$$-a_{n-1}2^{n-1}+\sum_{i=0}^{n-2}a_i 2^i$$

Beispiel: Für n=8 (1 Byte):

größte darstellbare Zahl:	01111111 = 127
kleinste darstellbare Zahl:	10000000 = -128

Darstellung ganzzahliger Datentypen (8)

Weitere Beispiele:

00000000 = 0
00000001 = 1
00000010 = 2
00000011 = 3
...
01111110 = 126
01111111 = 127

10000000 = -128
10000001 = -127
10000010 = -126
10000011 = -125
..
11111111 = -1

➔ dies liefert den Datentyp byte

Datentyp byte

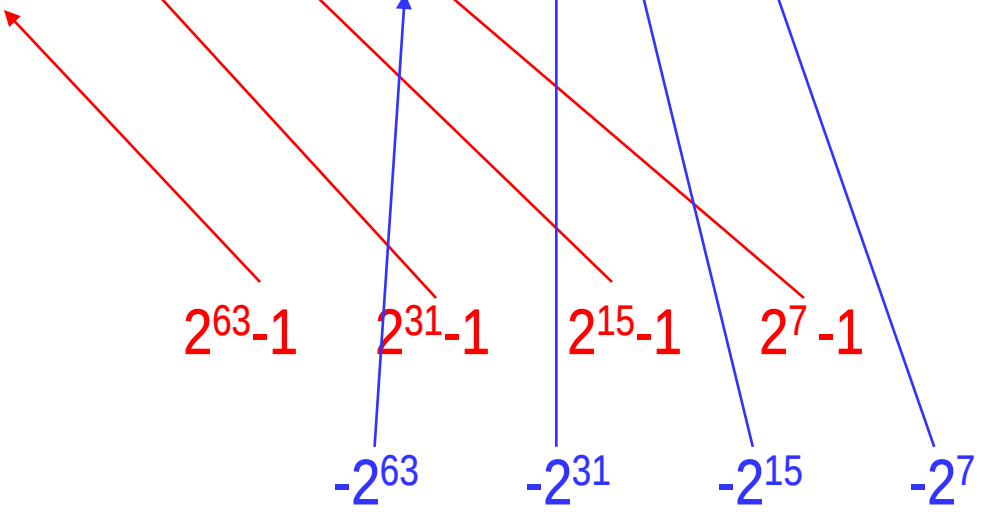
Größe/Länge: 1 Byte
Wertebereich: -128 bis 127

entspricht mit 24 vorangestellten Nullen der Darstellung für int = 128 = 129

Darstellung ganzzahliger Datentypen (9)

- Die übrigen ganzzahligen Typen werden analog behandelt.

Typname	Größter Wert	kleinster Wert	Länge
`byte`	127	-128	8
`short`	32767	-32768	16
`int`	2147483647	-2147483648	32
`long`	9223372036854775807	-9223372036854775808	64



Die Zweierkomplementdarstellung erklärt die Asymmetrie der Werteverteilung.

Darstellung von Gleitkommatypen

- Die exakte Darstellung hat nach IEEE Standard for Binary Floating-Point Arithmetic, ANSI / IEEE Standard 754-1985 (IEEE, New York) die Form:

float: $s*m*2^e$ 32 Bits

mit s = +1 oder -1

m positive ganze Zahl kleiner als 2^{24}

e eine ganze Zahl zwischen -149 und 104 (einschließlich)

double: $s*m*2^e$ 64 Bits

mit s = +1 oder -1

m positive ganze Zahl kleiner als 2^{53}

e eine ganze Zahl zwischen -1075 und 970 (einschließlich)

[http://de.wikipedia.org/wiki/Gleitkommazahl#Darstellung]