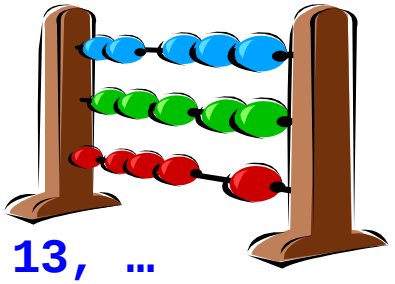
Daten und Operationen

Bits, Bytes, Binärzahlen, Hex-Zahlen, Dezimalzahlen, Konversionen, cast, this, Würfel, Boolesche Werte, Zeichen, Unicode, Fonts

Operationen auf ganzen Zahlen



- Grundoperationen
  - Zahlkonstanten (Literale)
    `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, …`
  - einstellige Operationen
    `+, -` plus, minus
    - Beispiel: **`+5`**, **`-17`**, **`-(3-4)`** = …
  - zweistellige Operationen
    `+, -, *` Addition, Subtraktion, Multiplikation
    - Beispiele: **`-3 + 17, 5 + -17, 3*(4-7)`**

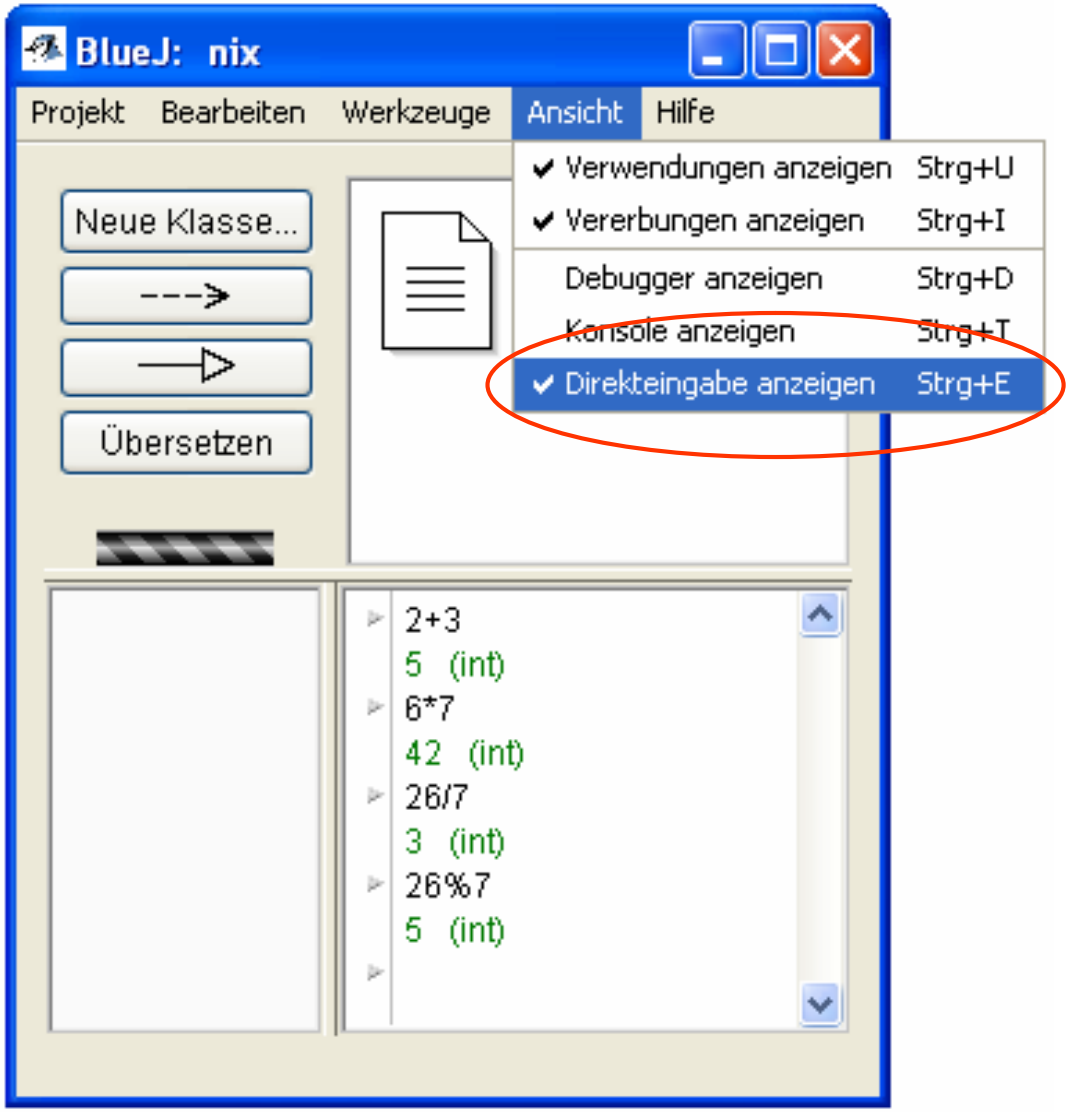
    `/` Ganzzahl-Division, mathematisch: *div*)
    - Beispiele: **`17/4`** = … , **`-1/-4`** = …, **`-3/2`**= …, **`(-3)/2`**= …

    `%` Rest bei Ganzzahl-Division, mathematisch: *mod*)
    - Beispiele: **`17%4`** = …, **`-1%-4`** = … , **`-3%2`**= …

BlueJ als Java-Taschenrechner

- Direkteingabe aktivieren

- Im Calculator rechnen
  - **`26/7 = 3`**
  - **`26%7 = 5`**
  - **`etc.`**



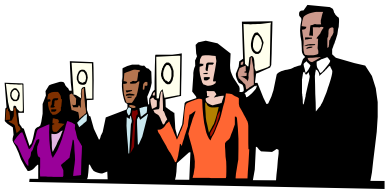
div und mod

- div : Division ohne Rest
  - 23 div 7 = 3
- mod: Rest bei der Division
  - 23 mod 7 = 2
- Eigenschaften von *div* und *mod*
  - Seien n und d natürliche Zahlen, k > 0, dann gilt:
    - $(n \mathit{div} d)*d + (n \mathit{mod} d) = n$
    - $0 \leq (n \mathit{mod} d) < d$
  - Sei $z=(d_{k-1}d_{k-2}...d_1d_0)_{10}$ die Dezimaldarstellung von z:
    - $0 \leq d_i < 10$ für i=0,...,(k-1)
    - $(d_{k-1}d_{k-2}...d_1)_{10} = z \mathit{div} 10$
    - $d_0 = z \mathit{mod} 10$
  - Beispiel:
    - $2002 = (2\ 0\ 0\ 2)_{10}$
    - $2002 \mathit{div} 10 = 200 = (2\ 0\ 0)_{10}$
    - $2002 \mathit{mod} 10 = 2$

Wichtig !

Computer verstehen nur 0 und 1 …

- Eine physikalische Speicherzelle kann nur einen von zwei Zuständen haben
  - Strom/kein Strom, Ladung/keine Ladung, ein/aus, positiv/negativ
  - Interpretiere einen dieser Zustände als 0, den anderen als 1



0 oder 1

- Eine Gruppe von Speicherzellen kann viele Kombinationen von Zuständen speichern:
  - Eine Gruppe von zwei Speicherzellen kann 4 Zustände speichern
    - 00, 01, 10, 11

00 , 01 , 10 , 11

  - Eine Gruppe von 4 Speicherzellen kann 16 Zustände speichern
    - 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111

0000 , 0001 … 1111

  - Eine Gruppe von 8 Speicherzellen kann 256 = 2^8 Zustände speichern:
    - 00000000, 00000001, … 11111110, 11111111

00000000 … 11111111

Byte, Wort, Doppelwort

- Kein Computer der Welt gibt sich mit einzelnen Speicherzellen ab
  - Die kleinste Einheit, mit der Computer rechnen ist ein *Byte*, das sind 8 Bit
  - Die kleinste Einheit, die ein Computer speichern oder lesen kann ist ein Byte
- 16-Bit-Computer rechnen intern mit zwei Byte großen Daten
  - zwei Byte nennt man dann ein *Wort*
  - Vier Byte heißen ein Doppelwort, 8 Byte ein Quad-Wort
- 32 Bit Computer rechnen intern mit 4 Byte großen Daten
  - Ob dann 4 Byte ein Wort oder ein *Doppelwort* heißen, ist uneinheitlich
  - Die kleinste im Speicher adressierbare Einheit kann 1 Byte, 2 Byte aber auch 4 Byte sein
- 64-Bit Computer …

```
  0 0 1 1 0 1 0 1 0 1 0 1 0 1 1 1
+ 0 1 0 1 0 1 1 1 0 0 1 1 0 1 0 1
-----------------------------------
= 1 0 0 0 1 1 0 0 1 0 0 0 1 1 0 0
```

Können Computer auch Zahlen speichern ?

- **Man kann den Speicherzuständen Zahlen zuordnen**
  - Zum Beispiel positive Zahlen
    - **0000** = 0, **0001** = 1, **0010** = 2, **0011** = 3, **0100** = 4, **0101** = 5, **0110** = 6, **0111** = 7, **1000** = 8, **1001** = 9, **1010** =10, **1011** = 11, **1100** = 12, **1101** = 13, **1110** = 14, **1111** = 15
  - oder ganze (positive und negative) Zahlen
    - **0000** = 0, **0001** = 1, **0010** = 2, **0011** = 3, **0100** = 4, **0101** = 5, **0110** = 6, **0111** = 7, **1000** = -8, **1001** = -7, **1010** = -6, **1011** = -5, **1100** = -4, **1101** = -3, **1110** = -2, **1111** = -1

- **In einem Byte kann man speichern**
  - eine *positive Zahl* zwischen 0 und 255 d.h. zwischen 0 und 2^8, oder
  - eine *ganze Zahl* aus dem Bereich -128 .. +127 , d.h. -2^7 .. 2^7-1.

- **In zwei Byte passt**
  - eine positive Zahl zwischen 0 und 65535, d.h. zwischen 0 und 2^{16} , oder
  - eine ganze Zahl aus dem Bereich -32768 .. +32767 , d.h. -2^{15} .. $2^{15}-1$.

- **In 4 Byte passt**
  - eine positive Zahl zwischen 0 und 4294967295 d.h. zwischen 0 und 2^{32}, oder
  - eine ganze Zahl aus dem Bereich -2147483648 .. +2147483647 , d.h. -2^{31} .. $2^{31}-1$.

Bitfolge	Positive Zahl	Ganze Zahl
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Binärsystem

- Binärziffern:
  - 0 und 1
- Binärzahl:
  - Folge $b_n b_{n-1} \ldots b_1 b_0$ von Binärziffern
    - Beispiel: 10110111
  - Falls Missverständnis möglich, Index 2 verwenden
    - Beispiel : $(10110111)_2$
- Zahlenwert
  - $W = b_n*2^n+b_{n-1}*2^{n-1}+\ldots+b_1*2^1+b_0 = (b_n b_{n-1} \ldots b_1 b_0)_2$
    - Beispiel: $10110111 = 2^7+2^5+2^4+2^2+2^1+2^0 = 183$
- Beobachtung
  - $b_0 = 1 \Leftrightarrow$ W ungerade bzw. $b_0 = 0 \Leftrightarrow$ W gerade
  - 0 an Binärzahl anhängen $\Leftrightarrow$ Wert mit 2 multiplizieren
    - Beispiel $(1011)_2 = 11$ und $(10110)_2 = 22$

Binär - Dezimal

- Zahlenwert
  - $W = b_n * 2^n + b_{n-1} * 2^{n-1} + \ldots + b_1 * 2^1 + b_0 = (b_n\, b_{n-1} \ldots\, b_1\, b_0)_2$
- $W = 2 * (b_n * 2^{n-1} + b_{n-1} * 2^{n-2} + \ldots + b_1 * 2^0) + b_0$

  $= 2 * (b_n\, b_{n-1} \ldots\, b_1)_2 + b_0$ und $0 \leq b_0 < 2$

  W div 2 $\quad$ W mod 2
- Also von Dezimal nach Binär
  - $W = (b_n b_{n-1} \ldots b_1 b_0)_2$

    $\Leftrightarrow \quad W \text{ div } 2 = (b_n b_{n-1} \ldots b_1)_2$

    und $\quad W \bmod 2 = b_0$

Binärsystem

- Binärziffern:
  - 0 und 1
- Binärzahl:
  - Folge $b_n b_{n-1} \ldots b_1 b_0$ von Binärziffern
    - Beispiel: 10110111
  - Falls Missverständnis möglich, Index 2 verwenden
    - Beispiel : $(10110111)_2$
- Zahlenwert
  - $W = b_n*2^n+b_{n-1}*2^{n-1}+\ldots+b_1*2^1+b_0$
    - Beispiel: $10110111 = 2^7+2^5+2^4+2^2+2^1+2^0 = 183$
- Von Dezimal nach Binär
  - $W = (b_n b_{n-1} \ldots b_1 b_0)_2$
    $\Leftrightarrow$ $W \text{ div } 2 = (b_n b_{n-1} \ldots b_1)_2$
    und $W \bmod 2 = b_0$

Umwandlung ins Binärsystem

W	= 2* (W div 2)	+	W mod 2
n 2002	= 2 * 1001	+	0
n 1001	= 2 * 500	+	1
n 500	= 2 * 250	+	0
n 250	= 2 * 125	+	0
n 125	= 2 * 62	+	1
n 62	= 2 * 31	+	0
n 31	= 2 * 15	+	1
n 15	= 2 * 7	+	1
n 7	= 2 * 3	+	1
n 3	= 2 * 1	+	1
n 1	= 2 * 0	+	1

$(2002)_{10} = (1\ 1\ 1\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 0)_2$

Hexadezimalsystem

- Im Hexadezimalsystem verwendet man
  - 16 Ziffern :
    - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
  - Positionsbewertungen
    - F53CA = $F*16^4+5*16^3+3*16^2+C*16^1+A*16^0 = (1004490)_{10}$
- Umrechnung vom Dezimalsystem ins Hexadezimalsystem:
  - $(2002)_{10} = (d_2\ d_1\ d_0)_{16}$

$$= d_2*16^2+d_1*16^1 + d_0*16^0$$

$$= 16*(d_2*16^1+d_1*16^0) + d_0, \quad \text{wobei } 0 \le d_0 < 16$$

2002 div 16 **2002 mod 16**

- Konsequenz:

$\Rightarrow$ $d_0 = 2002 \bmod 16 = 2$

  - $2002 \text{ div } 16 = 125 = (d_2\ d_1)_{16}$ $\Rightarrow$ $d_1 = 125 \bmod 16 = D$
  - $125 \text{ div } 16 = 7 = (d_2)_{16}$ $\Rightarrow$ $d_2 = 7 \bmod 16 = 7$

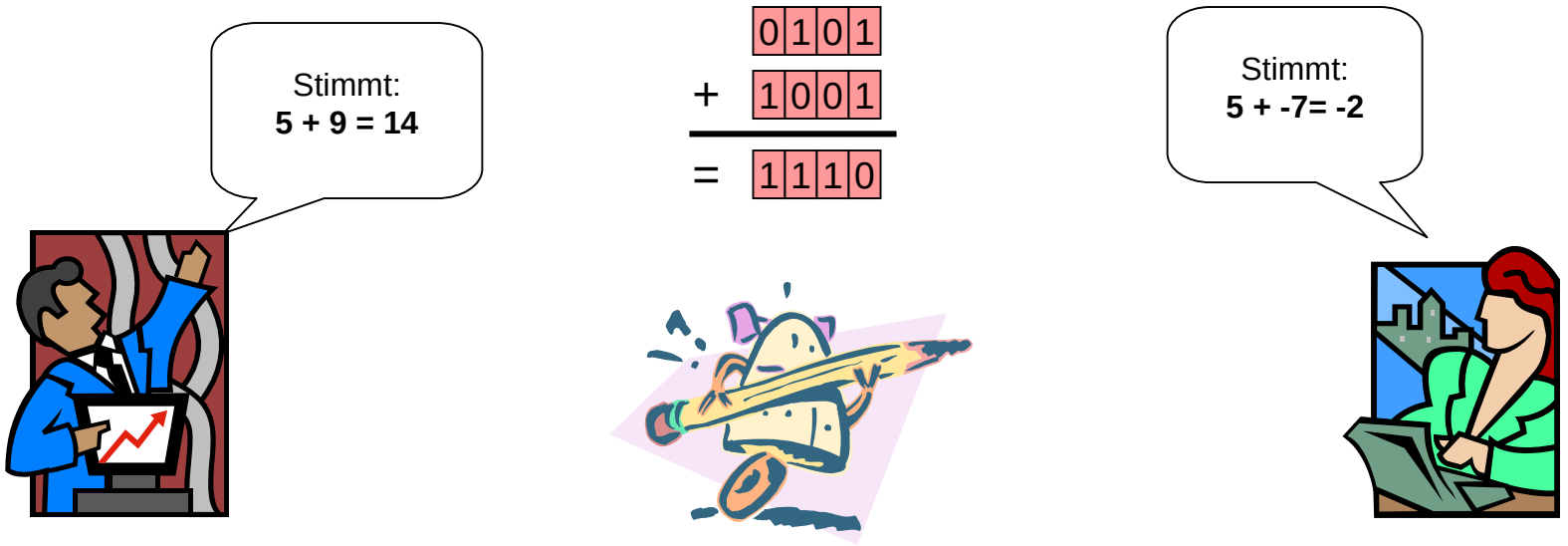
- Ergebnis:
  - $2002 = (7D2)_{16}$

Darstellung von Bytes

- Ein Byte wie z.B. 10111110 kann aufgefasst werden als
  - 8-stellige Zahl im *Binärsystem*
    - $(10111110)_2 = 1*2^7+0*2^6+1*2^5+1*2^4+1*2^3+1*2^2+1*2^1+0*2^0$
  - 3 stellige Zahl im *Oktalsystem* (mit führender Oktalziffer < 4)
    - $(10\ 111\ 110)_2 = 2*8^2+7*8^1+6*8^0 = \mathbf{(276)_8}$
  - 2-stellige Zahl im *Hexadezimalsystem*
    - $1011\ 1110 = 11*16^1+14*16^0 = \mathbf{(BE)_{16}}$

Woher weiß der Computer, ob eine positive oder eine negative Zahl gemeint ist ?

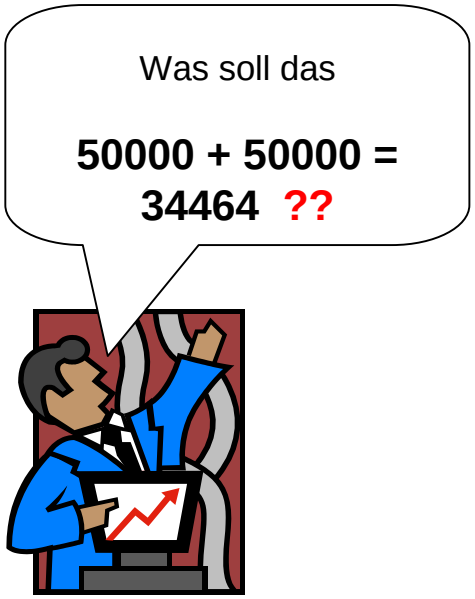
- Er weiss es nicht
  - Dank der cleveren Codierung (Zweierkomplement) braucht er es nicht zu wissen
  - Nur der Programmierer muss wissen, was seine Daten bedeuten

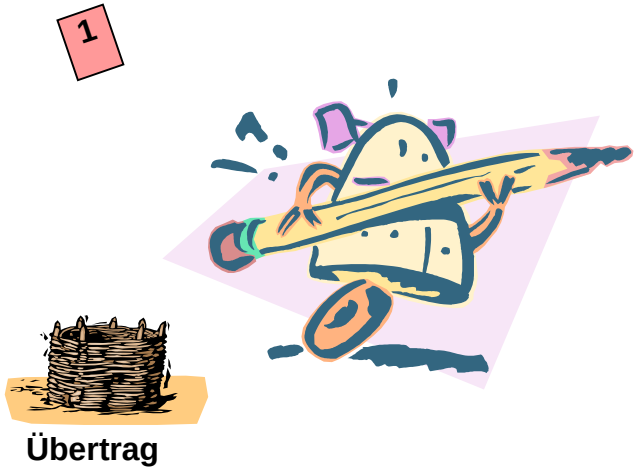


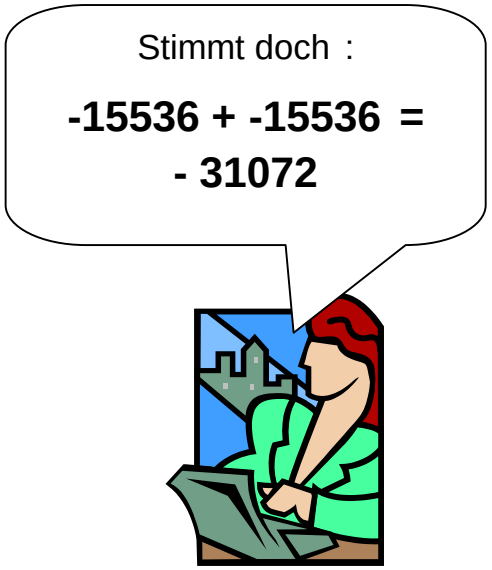
Bereichsfehler

- Operationen auf Zahlen können den festgelegten Bereich überschreiten
  - z.B. Addition zweier großer positiver Zahlen
    - Im 2-Byte-Bereich 0 .. 2^{16}-1 die Addition 50000+50000:
      1100001101010000 + 1100001101010000 =1100001101010100000

	1	1	0	0	0	0	1	1	0	1	0	1	0	0	0	0
+	1	1	0	0	0	0	1	1	0	1	0	1	0	0	0	0
=	1	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0

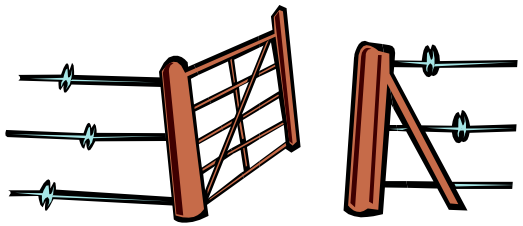






Bereichsüberschreitung

- Auch bei ganz-zahligen Werten ist Bereichsüberschreitung möglich
  - z.B. bei der Addition großer Zahlen,
  - z.B. bei Subtraktion kleiner Zahlen,
  - bei der Multiplikation.



- Beispiel aus dem Bank-Projekt
  - Legen Sie ein Konto für "**`Bill Gates`**" an.
  - Zahlen Sie sein Geld ein: **2147483600 €.**
  - Überweisen Sie ihm nun das Geld für Ihr Windows-Update: **60 €.**
  - Überprüfen Sie mit *Inspect,* wie viele „Miese" er jetzt hat.

- Java speichert int-Werte in 4 Bytes ab
  - Bereich: - 2147483648 ... 2147483647
  - Für Bereichsüberschreitungen ist der Programmierer verantwortlich

Ganze Zahlen in Java

- Java bietet vier Varianten von ganz-zahligen Wertetypen
  - byte
    - 1 Byte = 8 bit
    - Wertebereich $-2^7 = -128 \ldots +127 = 2^7 - 1$
  - short
    - 2 Byte = 16 bit
    - Wertebereich $-2^{15} = -32768 \ldots +32767 = 2^{15} - 1$
  - int
    - 4 Byte = 32 bit
    - Wertebereich $-2^{31} = -2147483648 \ldots +2147483647 = 2^{31} - 1$
  - long
    - 8 Byte = 64 bit
    - Wertebereich $-2^{63} \ldots 2^{63}-1$, d.h. -9223372036854775808 … 9223372036854775807

Zahlenliterale

- Zahlenkonstanten (Literale)
  - standardmässig als *int* interpretiert
- int-Bereich
  -2147483648 … +2147483647
- Interpretation als *long* durch Anhängen von L oder l

```
42
42  (int)
-123
-123  (int)
2000000000
2000000000  (int)
2000000000 + 2000000000
-294967296  (int)
2000000000L
2000000000  (long)
2000000000L + 2000000000L
4000000000  (long)
3000000000
Error: integer number too large: 3000000000
3000000000L
3000000000  (long)
```

Konversionen

- Werte aus einem Bereich können in Werte des anderen konvertiert werden
  - engl.: *to cast*
  - neudeutsch: *casten, gecastet, …*
- Verknüpfung von Werten aus verschiedenen Bereichen konvertiert automatisch in den größeren
  - z.B. (short)+(long) = (long)
- Konvertierung in zu kleinen Bereich wird ausgeführt
  - auch wenn es nicht passt
  - Fehler

```
42
42  (int)
(byte)42
42  (byte)
(long)42
42  (long)
(short)-42
-42  (short)
(short)23 + (long)19
42  (long)
(byte)127
127  (byte)
(byte)128
-128  (byte)
(byte)129
-127  (byte)
(byte)255
-1  (byte)
(byte)256
0  (byte)
```

Operationen auf Dezimalzahlen

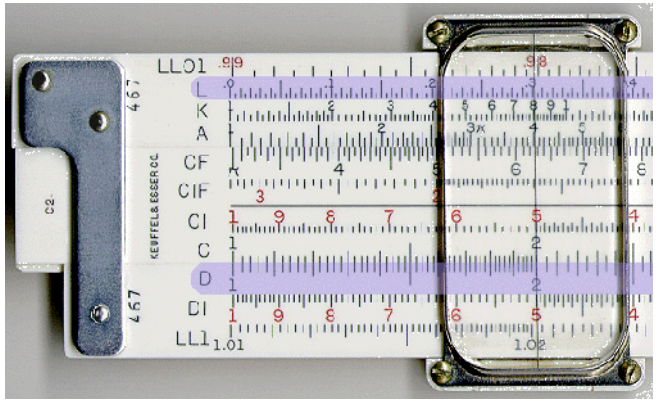
- Grundoperationen

  - Zahlkonstanten (Literale)
    - Zahlen mit Dezimalpunkt
      - Beispiel: 0.0124, 3.1415, 1234.0, .23
    - Wissenschaftliche Notation
      - Beispiele: 234E-12 für $2.34*10^{-12}$, 1.5E17
  - Einstellige Operationen

    +, - plus, minus
  - Zweistellige Operationen

    +, -, *, / Addition, Subtraktion, Multiplikation, Division

Dezimalzahlen



- Dezimalzahlen sind Zahlen mit Komma. Man speichert
  - das Vorzeichen
  - die Folge der Ziffern (Mantisse)
  - die Position des Kommas (Exponent)

… allerdings im Binären Zahlensystem

-7141,59270595 =

-0,71415927059 5 * 10^4

Vorzeichen: **1**

Mantisse **71415927**

Exponent: **4**

Viele Bits für Mantisse: hohe Genauigkeit
Viele Bits für Exponenten: großer Bereich

Java Dezimalzahlen

- Java hat zwei Varianten von Dezimalzahlen
  - float (floating point number, single precision)
    - 32 Bit
  - double (double precision)
    - 64 Bit

- Das genaue Format ist in der IEEE Norm 745 festgelegt:

IEEE 745	Sign	Exponent	Mantisse
single	1 bit	8 bit	23 bit
Double	1 bit	11 bit	52 bit

Dezimale Literale

- standardmäßig *double*
- float durch Anhängen von F oder f
- gemischte Operationen konvertieren in größeren Bereich (*double*)
- Konversionen, casts
  - (float)
  - (double)
- Konversion nach (int)
  - schneidet Stellen ab
  - nicht dasselbe wie runden

```
0.1
0.1  (double)
3.14
3.14  (double)
3.
3.0  (double)
.3
0.3  (double)
3.14F
3.14  (float)
3.14F + 3.14D
6.2800001049041 75  (double)
(float) 3.14
3.14  (float)
3.14/2
1.57  (double)
3.14/3.14
1.0  (double)
(int)3.14
3  (int)
(int)-3.14
-3  (int)
(int)-3.99
-3  (int)
```

Fehler bei Dezimalzahlarithmetik

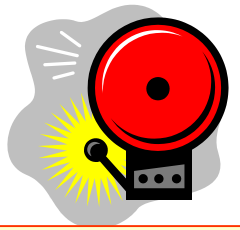
- Dezimalzahlarithmetik ist nicht exakt
  - Werte werden approximiert
  - Wir addieren 0.1 und 0.2 und erhalten:

```
0.1 + 0.2
0.30000000000000004  (double)
0.1 + 0.2 == 0.3
false  (boolean)

1.80 + .30 + .25 + .25
2.6  (double)
1.80 + .25 + .25 + .30
2.5999999999999996  (double)
```

- Vergleiche sind nicht 100% verlässlich:
  `0.1 + 0.2 ≠ 0.3`
- Addition ist nicht kommutativ

Vergleichsoperationen

- Vergleichsoperationen liefern einen von zwei Werten
  - **`true`** oder **`false`**
- Sie sind auf allen Zahlen (ganze, dezimale) definiert
  - < , > (kleiner, größer)
    - Beispiel: 2 < 3, 1 < 1.2, kontoStand > 0, …
  - <=, >= (größer oder gleich, kleiner oder gleich)
    - Beispiel: 5 <= 7, getKontostand() >= betrag,
  - == (gleich)
    - 7 == 8, kontoStand == 0
  - != (ungleich)



```
kontoStand = 0
// setzt den kontoStand auf 0 !!!
```

Bibliotheksfunktionen

- In der Klasse Math gibt es zusätzliche Operationen
  - Konstanten
    `E, PI`
  - Trigonometrische Funktionen
    `sin(), cos(), tan(), asin(), acos() atan()`,
  - Exponentialfunktionen
    `exp(), log(), abs(), sqrt(), pow()`
  - Rundungsfunktionen
    `floor(), ceil(), round(), rint()`
  - Zufallszahlengenerator
    `random()` liefert eine zufällige Zahl zwischen `0` und `1`

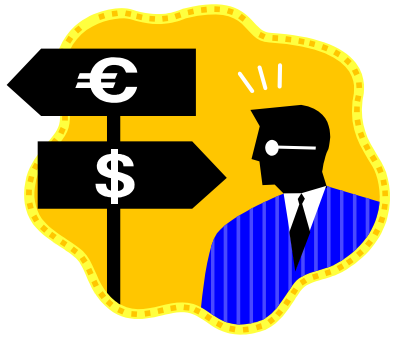
- Die meisten dieser Funktionen nutzen den Datentyp `double`

Experimente

- Konstanten und Funktionen gehören zur Klasse *Math*
  - Konstanten immer ganz in Großbuchstaben
- Voranstellen des Klassennamens notwendig
  - statische Felder/Methoden
- Trig. Funktionen approximativ
  - Math.sin(π) $\approx$ 0
- Logarithmus zur Basis e
- Math.exp(x) = e^x
- Math.random() liefert Zufallszahl
  - jedesmal eine neue
  - Werte zwischen 0 und 1

```
PI
Error: not a statement
Math.PI
3.1415926535897793  (double)
Math.sin(30)
-0.9880316240928618  (double)
Math.sin(Math.PI)
1.2246467991473532E-16  (double)
Math.E
2.7182818284590045  (double)
Math.log(100)
4.6051701859880092  (double)
Math.exp(1)
2.7182818284590455  (double)
Math.exp(4.605170)
99.999998140119261  (double)
Math.sqrt(2)
1.4142135623730951  (double)
Math.floor(-3.7)
-4.0  (double)
Math.ceil(-3.7)
-3.0  (double)
Math.random()
0.76102291008570333  (double)
Math.random()
0.96468595846594  (double)
```

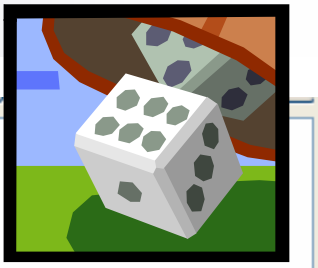
Konversion



- Automatische Konversion
  - Automatische Umwandlung in gemeinsamen Typ
    - 3/2.0 = 1.5 :
      - 3 (int) wird zu 3.0 (double) umgewandelt, dann geteilt.
      - Das Ergebnis ist 1.5 (double)
    - 3/2 = 1 :
      - 3 und 2 sind vom Typ int.
      - „/“ wird als Integer Division aufgefasst.
  - Umwandlung in den Typ, der die meisten Bits benutzt
    - int → float → double
- Explizite Konversion durch
  - Aufrufen mathematischer Funktionen
    - Math.round
  - Voranstellen des Typs in Klammern sogenannter „cast“
    - (short)Zinssatz, (int)3.14, (int)(Math.random()*6)

… dabei kann Präzision verloren gehen !!!

Ein Würfel



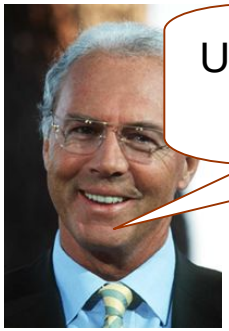
Würfel
Konstruktor: Wuerfel() Methoden: int getAugenzahl() void wuerfele()

Anwendung von *Math.random()*

§ skalieren auf Intervall 0 < x < 6

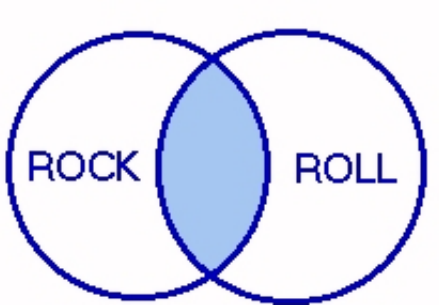
§ (int)-cast liefert y ∈ {0,1,2,3,4,5}

§ 1 addieren




```
/** Beschreibung der Klasse Wuerfel.
 * @author H.P.Gumm
 * @version 7.9.06
 */
public class Wuerfel {
//  ################    Objekt-Felder  ################

    /** Was oben liegt */
    private int augenZahl;

//   ##############  Konstruktoren    ###################

    /** Default-Konstruktor für Objekte der Klasse Wuerfel */
    public Wuerfel(){
      wuerfele();
    }

//   ############   Objekt-Methoden    ###################

    /** Würfele eine neue Augenzahl */
    public void wuerfele(){
       augenZahl = (int)(6*Math.random())+1;
    }
    /** Ergebnis des letzten Wurfes
     *  @return augenzahl die Augenzahl, die oben liegt
     */
    public int getAugenZahl(){
      return augenZahl;
    }
}// Ende der Klasse Wuerfel
```

Java Datentyp: boolean



- Boolesche Werte: **`false`** , **`true`**.
  - Operationen
    - Literale

      `false`, **`true`**
    - Einstellige Operation

      `!` (not)
      - Wird als Präfix verwendet: **`!true`**, !(2<1), !(zinsSatz<3)
    - Zweistellige Operationen

      `&&` (and)
      - Wird kurz ausgewertet: **`(2 < 1) &&`** … == **`false`**

      || (or)
      - Wird kurz ausgewertet: **`(1 < 2) ||`** … **`== true`**

Rechnen mit booleschen Werten

- Boolesche Operatoren binden schwächer als Vergleichsoperationen
  - jene schwächer als arithmetische Operationen
- || und && werden verkürzt (lazy) ausgewertet
  - nur wenn nötig
  - schneller
  - kann benutzt werden um vor Fehlern zu schützen

```
true
true   (boolean)
! true
false   (boolean)
! ! true
true   (boolean)
false || ! false
true   (boolean)
2 == 3 || 2 < 2
false   (boolean)
5/0 == 1
Error: ArithmeticException:
/ by zero
2 < 3 || 5/0 == 1
true   (boolean)
5/0 == 1 || 2 < 3
Error: ArithmeticException:
/ by zero
3 > 4 && 1/0 ==1
false   (boolean)
```

Boolesche Algebra

- Algebra der Logik
  - Logische Aussagen müssen entweder Wahr (W) oder Falsch (F) sein
  - Logische Operationen
    - ¬ not,
    - ∧ and
    - ∨ or
    - ⇒ wenn...dann
    - ⇔ genau dann ... Wenn
    - ⊕ xor, bzw. entweder ... oder
  - Basisoperationen durch Tabellen definiert
    - ¬ , ∧ , ∨
  - Andere Operationen lassen sich herleiten

¬	
W	F
F	W

∧	F	W
F	F	F
W	F	W

Beachte: F ∧ ... = F

∨	F	W
F	F	W
W	W	W

Beachte: W ∨ ... = W

George Boole. 1815-1864
Engl. Mathematiker

Boolesche Gleichungen

$x \vee x = x$ $x \vee y = y \vee x$ $x \vee (y \vee z) = (x \vee y) \vee z$	idempotent kommutativ assoziativ	$x \wedge x = x$ $x \wedge y = y \wedge x$ $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
$x \vee (x \wedge y) = x$	absorptiv	$x \wedge (x \vee y) = x$
$\neg(x \vee y) = \neg x \wedge \neg y$ $x \vee \neg x = W$	deMorgan Komplement	$\neg(x \wedge y) = \neg x \vee \neg y$ $x \wedge \neg x = F$
$\neg\neg x = x$	Doppelte Negation	

Abgeleitete Operationen

- Zusätzliche Operationen lassen sich aus den vorhandenen definieren:

$$x \Rightarrow y \;:=\; \neg x \vee y$$

$$x \Leftrightarrow y \;:=\; (x \wedge y) \vee (\neg x \wedge \neg y)$$

$$x \oplus y \;:=\; (x \wedge \neg y) \vee (\neg x \wedge y)$$

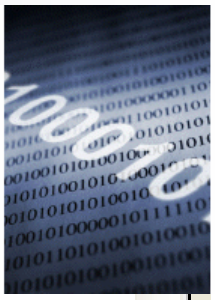
$\Rightarrow$	F	W
F	W	W
W	F	W

$\Leftrightarrow$	F	W
F	W	F
W	F	W

$\oplus$	F	W
F	F	W
W	W	F

(2=1) $\Rightarrow$ Russel ist Papst

Bertrand Russell, 1872-1970
Englischer Philosoph und Logiker



Bitoperationen

- Varianten der booleschen Operationen arbeiten auf den Bits von Werten
  - 13 & 6 = 4
  - 13 | 6 = 15
  - 13 ^ 6 = 11
- Auf booleschen Werten stimmt das mit den normalen Operationen überein
  - aber nicht lazy
  - daher kaum benutzt

```
13 & 6
4  (int)
13 | 6
15  (int)
13 ^ 6
11  (int)
true | false
true  (boolean)
true | 1/0 == 1
Error: ArithmeticException:
/ by zero
13 >> 2
3  (int)
13 << 2
52  (int)
```

$$\begin{array}{r} (1101)_2 \\ \& \ (0110)_2 \\ \hline = \ (0100)_2 \end{array}$$

$$\begin{array}{r} (1101)_2 \\ | \ (0110)_2 \\ \hline = \ (1111)_2 \end{array}$$

$$\begin{array}{r} (1101)_2 \\ \wedge \ (0110)_2 \\ \hline = \ (1011)_2 \end{array}$$

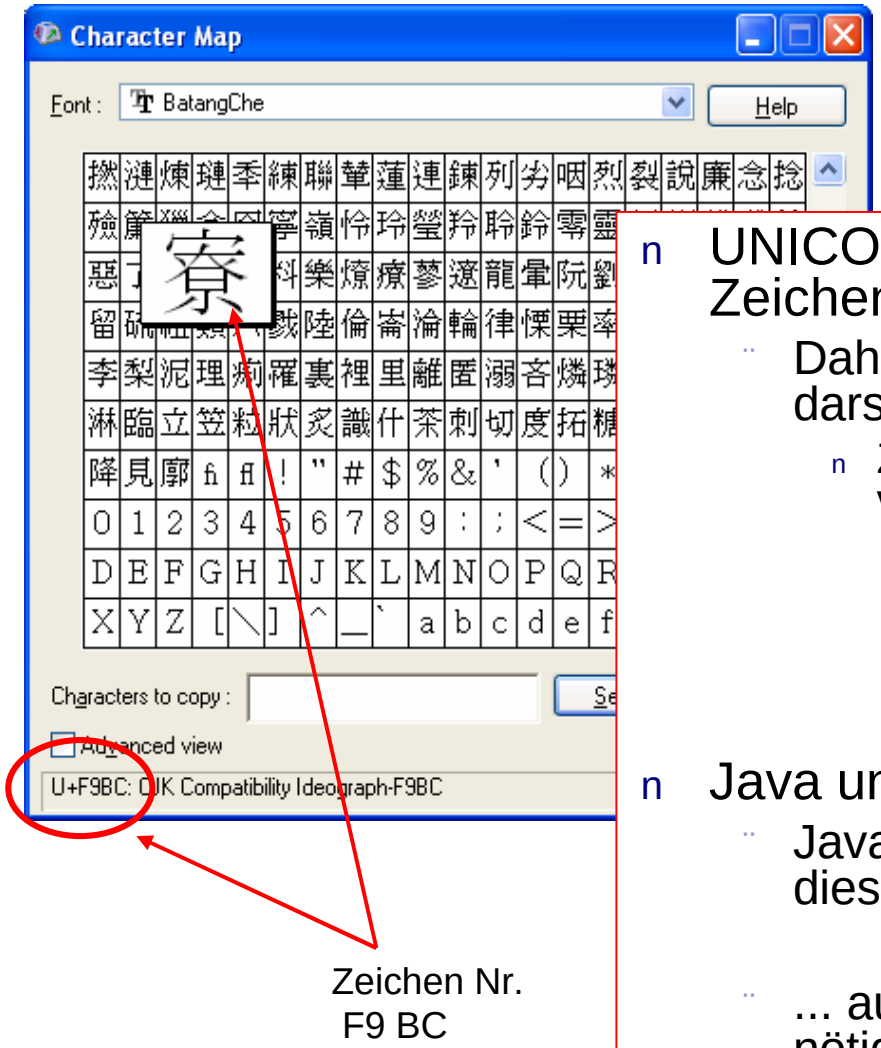
$$(1101)_2 >> 2 = (0011)_2$$

$$(1101)_2 << 2 = (110100)_2$$

Zeichen

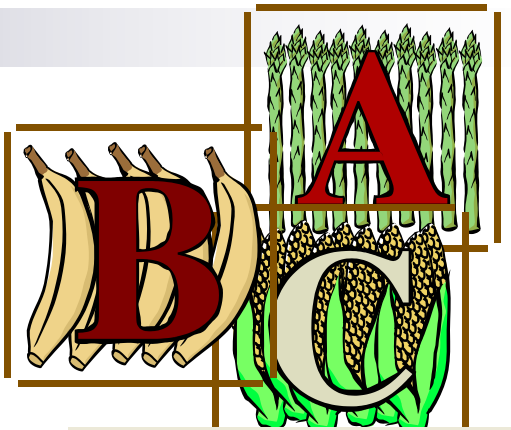
- Zeichen dienen (nicht nur) zur Darstellung von Text
  - Buchstaben
    - Großbuchstaben
      - A B C D ... X Y Z
    - Kleinbuchstaben
      - a b c d ... x y z
  - Ziffern
    - 0 1 2 3 4 5 6 7 8 9
  - Sonderzeichen
    - , ; . : - _ ? ! # + - * / \ % § $ ...
  - Nichtdruckbare Zeichen
    - CR (Zeilenvorschub), TAB (Tabulator), Bell, Abbruch
    - Einige dieser Zeichen dienen zur Formatierung von Text

UNICODE



- UNICODE benutzt eine 2-Byte Codierung für Zeichen
  - Daher kann man in Unicode 65536 Zeichen darstellen
    - Zu den europäische Zeichen kommen auch noch viele Zeichen aus anderen Sprachen, u.a.
      - Griechische Φ, Kyrillische Ю Я,
      - Arabische ك,
      - Hiragana, Katakana,
      - Chinesisch/Kanji, Koreanisch, ...
- Java unterstützt UNICODE
  - Java Entwicklungssysteme, z.B. Editoren tun dies meist noch nicht
  - ... auch Betriebssysteme haben nicht immer die nötigen Zeichen für die Bildschirmdarstellung

Java Datentyp: char



- Literale:
  - Zeichen in Apostrophen:
    - ‘a‘, ‘A‘, ‘7‘, ‘:‘
  - Sonderzeichen mit backslash-escape :
    - ‘\n‘ (neue Zeile), ‘\t‘ (Tab), ‘\b‘ (backslash), ‘\r‘ (return), ‘\\‘ (backslash)
  - Unicode char : \u gefolgt von 4 Hex-Ziffern:
    - \uF9BC =
  - Oktal Wert : \ gefolgt von 3 Oktal-Ziffern
    - \376 = ÿ
- Relationen
  - **==, !=, <, >, <=, >=** sind auf **char** definiert
- Konversionen
  - **char** wird bei Bedarf automatisch zu **int** konvertiert

```
'a'
a  (char)
'a'+1
98  (int)
(char)('a'+1)
b  (char)
'a'+'b'
195  (int)
(char)('q' + 'A' - 'a')
Q  (char)
'a' < 'A'
false  (boolean)
(int)'a'
97  (int)
(int)'A'
65  (int)
'a' < 'b'
true  (boolean)
```

Operationen mit char

- *char* und *int* gleichwertig
  - automatische Konversion nach *int*
  - Rückkonversion durch cast
- Lateinisches Alphabet
  - ‘a‘ < ‘b‘ < … <‘z‘
  - ‘A‘ < ‘B‘ < … <‘Z‘
- Unicode oft nicht darstellbar
  - z.B. auf Konsolen

```
'a'
a  (char)
'a'+1
98  (int)
(char)('a'+1)
b  (char)
'a'+'b'
195  (int)
(char)('q' + 'A' - 'a')
Q  (char)
'a' < 'A'
false  (boolean)
(int)'a'
97  (int)
(int)'A'
65  (int)
'a' < 'b'
true  (boolean)
'\361'
ñ  (char)
'\F3C1'
Error: illegal escape character
'\uF3C1'
□  (char)
```

Zeichen und Fonts

- Fonts legen fest, wie druckbare Zeichen aussehen
  - Arial:
    - Das ist ein cooler Font
  - Haettenschweiler
    - Das ist ein cooler Font
  - **Courier**
    - Das ist ein cooler Font
  - WingDings:
    - Das ist ein cooler Font
- Jeden Font gibt es in drei Varianten:
  - Normal
  - **Fett** **(bold)**
  - *Kursiv* *(italic)*
  - ***Fett+Kursiv*** ***(bold-italic)***
- Unterscheidungsmerkmale von Fonts
  - Serifen oder Serifenlos (Sans Serif)
  - Proportional oder `fixed width`
  - Ligaturen: Schafft das ß ab !