

Softwareentwicklung I (IB)

— Arithmetik und Variablen —

Prof. Dr. Oliver Braun

Letzte Änderung: 11.08.2017 20:35

Numerische Ausdrücke

Numerale

- ▶ **Numeral** = im Programmtext genannte Zahlenkonstante
- ▶ Schreibweise ganzzahliger Numerale: Folge von Dezimalziffern
- ▶ Ziffern zur besseren Lesbarkeit wahlweise gruppiert mit _ (Underscores)
 - ▶ nicht als erstes oder letztes Zeichen
 - ▶ nicht mehrmals direkt nacheinander
- ▶ Positive und negative Werte: Vorzeichen + oder -
 - ▶ Vorzeichen **optional**: fehlendes Vorzeichen = +
- ▶ Beispiele

```
0
23
+23
-4_000
-0
+230859160
+230_859_160
```

Numerale mit anderer Zahlenbasis

- ▶ Normale Numerale dezimal = Zahlenbasis 10
- ▶ Syntax für andere Zahlenbasen:

- ▶ **Hexadezimal** (Basis 16): Präfix „0x“:

0x23

0x1000

- ▶ **Oktal** (Basis 8): Führende Ziffer „0“:

023

01000

Irrtümlich gesetzte führende Null schwer auffindbarer Fehler

- ▶ **Binär** (Basis 2): Präfix „0b“:

0b1000

0b1_010_000

Beispiel

```
public class Numbase {
    public static void main(String[] args) {
        System.out.println(23);           // 23
        System.out.println(1000);        // 1000
        System.out.println(0x23);        // 35
        System.out.println(0x1000);      // 4096
        System.out.println(023);         // 19
        System.out.println(01000);      // 512
        System.out.println(0b1000);     // 8
        System.out.println(0b1_010_000); // 80
    }
}
```

Grundrechenarten

- ▶ Schreibweise arithmetischer Ausdrücke ähnlich zur Mathematik („Prinzip der geringsten Verwunderung“)
- ▶ Einfache Verknüpfung: Zwei Numerale und **Operator**
- ▶ Operatoren für die Grundrechenarten:

+	Addition
-	Subtraktion
*	Multiplikation
/	Division

- ▶ Multiplikationsoperator * muss immer angegeben werden

Berechnung und Ausgabe

- ▶ Ausgabe von Text

```
System.out.println("2");           // Ausgabe 2
```

- ▶ Ausgabe eines Numerals:

```
System.out.println(2);             // Ausgabe 2
```

- ▶ Ausgabe eines Ausdrucks:

```
System.out.println(1 + 2);         // Ausgabe 3
```

Text (in Gänsefüßchen) wird immer unverändert ausgegeben:

```
System.out.println("1 + 2");       // Ausgabe 1 + 2
```

- ▶ Schreibweise von ganzen Zahlen nicht eindeutig:

```
System.out.println(+2);             // Ausgabe 2
```

```
System.out.println(02);             // Ausgabe 2
```

```
System.out.println(+00002);         // Ausgabe 2
```

Ganzzahlige Division

- ▶ Arithmetik (bisher) ausschließlich ganzzahlig
- ▶ Ganzzahlige Division schneidet Nachkommaanteil des Ergebnisses ab
- ▶ **kein Runden:**

$$\begin{array}{rcl} \hline 11/4 & \rightarrow & 2 \text{ (nicht } 2.75) \\ -11/4 & \rightarrow & -2 \text{ (nicht } -2.75) \\ \hline \end{array}$$

- ▶ Auf den ersten Blick fragwürdig, in der Praxis nützlich

Modulus

- ▶ Fünfte „Grundrechenart“: Divisionsrest = **Modulus**
- ▶ Operatorzeichen %
- ▶ Beispiele:
 - $11\%4 \rightarrow 3$
 - $7\%2 \rightarrow 1$
 - $19\%19 \rightarrow 0$
 - $1\%19 \rightarrow 1$
- ▶ Vorsicht bei negativen Operanden
 - $11\%-4 \rightarrow 3$
 - $-11\%4 \rightarrow -3$
 - $-11\%-4 \rightarrow -3$
- ▶ Faustregel: Vorzeichen des Ergebnisses = Vorzeichen des linken (ersten) Operanden

Mathematischer Divisionsrest

- ▶ % implementiert nicht den mathematischen Divisionsrest (mod, „Modulo“)
 - ▶ *Mathematik*: $x \bmod y$ ist b in:
 $x = a \cdot y + b$, mit $0 \leq b < m$
 - ▶ *Java*: $x\%y$ wird berechnet mit Hilfe ganzzahliger Division:
 $x\%y = x - (x/y)*y$

$x\%y$ kann negativ sein, mathematischer Divisionsrest nicht

Beispiel für $y = 3$

x	Mathematik $x \bmod 3$	Java $x \% 3$
4	1	1
3	0	0
2	2	2
1	1	1
0	0	0
-1	2	-1
-2	1	-2
-3	0	0
-4	2	-1

Zusammengesetzte Ausdrücke

- ▶ Ausdrücke können kombiniert werden
- ▶ Induktive Definition eines Ausdrucks:
 - ▶ Numeral („elementarer Ausdruck“) oder
 - ▶ zwei Ausdrücke + Operator („zusammengesetzter Ausdruck“)
- ▶ Beispiele für zusammengesetzte Ausdrücke:

$$3 + 2*4$$

$$2*3 + 4*5$$

$$100 - 10 - 20$$

$$+4*+5$$

$$-2--1$$

- ▶ Leerzeichen dienen nur der besseren Lesbarkeit und werden vom Compiler ignoriert

Auswertung

Semantik arithmetischer Ausdrücke

- ▶ Syntax von Ausdrücken liegt fest (Grammatik), aber die **Semantik** (= Berechnung der Werte) ist zu klären
- ▶ Ergebnis abhängig von der Reihenfolge der Operatorenanwendung
- ▶ Beispiel

$$2 + 3 * 4$$

Reihenfolge:

Addition vor Multiplikation:	$2 + 3 * 4 \rightarrow$	$5 * 4 \rightarrow$	20
Multiplikation vor Addition:	$2 + 3 * 4 \rightarrow$	$2 + 12 \rightarrow$	14

- ▶ Nur ein Ergebnis kann „richtig“ sein

Priorität

- ▶ Auswertungsreihenfolge folgt **Priorität** (= „Bindungsstärke“, „Operatorenvorrang“) der Operatoren
- ▶ Priorität der „Punkt-Operatoren“ ($*$, $/$, $\%$) höher als die der „Strich-Operatoren“ ($+$, $-$)
- ▶ Wie in der Grundschule gelernt: „Punkt vor Strich“
- ▶ Damit gilt (Multiplikation vor Addition):
 $2 + 3 * 4 \rightarrow 2 + 12 \rightarrow 14$
- ▶ Wieder: „Prinzip der geringsten Verwunderung“ (*principle of least astonishment*)

Klammern

- ▶ **Runde Klammern** (...) erzwingen eine bestimmte Auswertungsreihenfolge
- ▶ Eingeklammerte Teilausdrücke werden immer zuerst ausgerechnet

$$(2 + 3) * 4 \rightarrow 5 * 4 \rightarrow 20$$

- ▶ Klammern sind um jeden Ausdruck erlaubt:

$$2 + (3 * 4) \rightarrow 2 + 12 \rightarrow 14$$

$$(2 + 3) \rightarrow 5$$

$$(((2))) \rightarrow 2$$

Es spielt keine Rolle, ob Klammern die Auswertungsreihenfolge beeinflussen oder nicht

Unäre Vorzeichenoperatoren

- ▶ **Unäre** (= „einstellige“) **Vorzeichenoperatoren** + und - vor dem (einzigen) Operanden
- ▶ - tauscht das Vorzeichen, + aus Symmetriegründen
- ▶ Priorität der unären Operatoren **höher** als die der binären Operatoren
- ▶ Beispiele:

$-(1 + 2)$	$\rightarrow -(3)$	$\rightarrow -3$
$3*-4$	$\rightarrow 3*(-4)$	$\rightarrow -12$
$-3+-4$	$\rightarrow (-3)+(-4)$	$\rightarrow -7$
$-(2 + -3)$	$\rightarrow -(-1)$	$\rightarrow 1$

Assoziativität

- ▶ Priorität regelt Vorrang bei unterschiedlichen Operatoren
- ▶ Aber: Auch bei mehreren gleichrangigen Operatoren gibt es Alternativen
- ▶ Beispiel $8 - 3 - 2$:

linkes Minus zuerst: $8 - 3 - 2 \rightarrow 5 - 2 \rightarrow 3$

rechtes Minus zuerst: $8 - 3 - 2 \rightarrow 8 - 1 \rightarrow 7$

- ▶ Operatoren haben eine charakteristische **Assoziativität** (= „Bindungsrichtung“)
 - ▶ links-assoziativ: Der am weitesten links stehende Operator wird zuerst ausgewertet
 - ▶ rechts-assoziativ: Der am weitesten rechts stehende Operator wird zuerst ausgewertet
- ▶ Alle binären arithmetischen Operatoren sind links-assoziativ. Demnach:

$$8 - 3 - 2 \rightarrow 5 - 2 \rightarrow 3$$

Operatortabelle

► Zusammenstellung der Eigenschaften

Op	Priorität	Assoz.	Ops	Wirkung
+	1 (hoch)	rechts	1	positives Vorzeichen
-		rechts	1	negatives Vorzeichen
*	2	links	2	Multiplikation
/		links	2	Division
%		links	2	Modulus = Divisionsrest
+	3 (niedrig)	links	2	Addition
-		links	2	Subtraktion

Variablen und Wertzuweisungen

- ▶ Arithmetische Ausdrücke liefern Werte
- ▶ Variablen speichern Werte zur späteren Verwendung
- ▶ Modell: Variable = Box, in der ein Wert aufgehoben werden kann

Variablennamen

- ▶ Variablen haben Namen
- ▶ Variablennamen sind *Identifier*
- ▶ Variablennamen beginnen **per Konvention** mit kleinen Buchstaben
- ▶ Beispiele für Variablennamen:
 - i
 - counter
 - theUltimateAnswer
 - track2
- ▶ Variablennamen sind per Konvention englisch und verwenden keine deutschen Umlaute (ä, ö, ü, ß)

Definition von Variablen

- ▶ Variablen müssen definiert werden
- ▶ Beispiele für **Variablendefinitionen**:

```
int i;  
int counter;  
int theUltimateAnswer;  
int track2;
```

- ▶ Vor dem Namen steht der **Typ** der Variablen
- ▶ Typ „int“ = ganze Zahl
- ▶ Definition = Anweisung (engl. statement)
- ▶ Ein Variable darf nur einmal definiert werden

Wertzuweisung

- ▶ Eine **Wertzuweisung** (engl. assignment) gibt einer Variablen einen Wert
- ▶ Beispiele:

```
absoluteZero = -273;  
daysPerYear = 31 + 28 + 31 + 30 + 31 + 30 +  
               31 + 31 + 30 + 31 + 30 + 31;  
theUltimateAnswer = 179 % 18 * 5 / 2;
```

- ▶ Mehrfache Wertzuweisungen **überschreiben** vorhergehende Werte

```
int counter;  
counter = 1;  
counter = -(6 - 8);  
counter = 11 % 4;      // counter hat jetzt den Wert 3
```

- ▶ Eine Variable hat kein Gedächtnis

Unveränderliche Variablen

- ▶ Variablenwerte werden manchmal einmal zugewiesen und sollen sich dann nicht mehr ändern
- ▶ Bezeichnung „unveränderliche Variable“ = **Konstante**
- ▶ Schutz gegen Änderungen mit dem **Modifier** `final`

```
final int speedOfLight;  
speedOfLight = 299_793_218;
```

- ▶ „Modifier“ allgemein: optionale Zusätze zur Definition

Der Compiler hilft

- ▶ `final` erlaubt eine einzige Wertzuweisung

```
final int speedOfLight;  
speedOfLight = 299_793_218;  
speedOfLight = 0;           // Fehler!
```

- ▶ `final`-Definitionen helfen bei der Entwicklung eines Programms, tragen aber nicht zur Funktionalität bei
 - ▶ **großer Vorteil:** Der Compiler erinnert uns daran, dass wir die Variable als Konstante nutzen wollen
- ▶ **Variablen immer `final` definieren, außer Sie haben einen guten Grund es nicht zu tun!**

Variablen in Ausdrücken

- ▶ Variablen können auf beiden Seiten einer Wertzuweisung stehen:

links **Zuweisen** eines neuen Wertes, „Schreiben“ einer Variablen:

```
// fahrenheit schreiben  
fahrenheit = 91;
```

rechts **Verwenden** des alten Wertes, „Lesen“ einer Variablen

```
// fahrenheit lesen, celsius schreiben  
celsius = 4 * fahrenheit / 7 - 32;
```

- ▶ Variablen spielen je nach **Kontext** unterschiedliche Rollen
- ▶ „Kontext“ hier: linke oder rechte Seite einer Wertzuweisung

Variablen in Java und in der Mathematik

- ▶ Variable in Java \neq Variable in der Mathematik
- ▶ Beispiele

`i = i + 1;`

- ▶ Mathematik: Widerspruch: i kann keinen Wert haben, der gleichzeitig um 1 größer ist.
- ▶ Java: Wertzuweisung

`i = 2; i = 3;`

- ▶ Mathematik: Widerspruch: i kann nicht gleichzeitig die Werte 2 und 3 annehmen.
- ▶ Java: Zwei Wertzuweisungen nacheinander

Gleichheitszeichen:

Mathematik: Zustand ohne zeitliche Dimension

Javaprogramm: Abfolge zeitlich getrennter, nacheinander abgewickelter Teilschritte:

1. Ausdruck auf der rechten Seite komplett ausrechnen
 2. Ergebnis der Berechnung an die Variable auf der linken Seite zuweisen
- ▶ Textuell ähnlich, aber konzeptionell verschieden
 - ▶ In der Programmierung heisst = deshalb **Zuweisungsoperator**

Nicht initialisierte Variablen

- ▶ Eine neu definierte Variable hat keinen Wert, sie ist „nicht initialisiert“
- ▶ Eine nicht initialisierte Variable kann nicht gelesen, nur geschrieben werden
- ▶ Beispiel (fehlerhaft):

```
int i;  
int j;  
j = 2*i;    // Fehler - i ist nicht initialisiert
```

- ▶ Compiler prüft Initialisierung, übersetzt das Programm gegebenenfalls nicht

Initialisierung

- ▶ Variable kann bei der Definition sofort mit einem Startwert versorgt werden = **Initialisierung**

```
int fahrenheit = 91;  
int celsius = 4 * fahrenheit / 7 - 32;  
final int speedOfLight = 299_793_218;
```

- ▶ Kurzform für Definition & Wertzuweisung
- ▶ Getrennte Definition und Wertzuweisung ...

```
int a;  
a = 1;
```

... äquivalent zu:

```
int a = 1;
```

- ▶ Variablen möglichst **immer** initialisieren

Anweisungsarten

- ▶ Arten von Anweisungen (statements) bisher

Definition	<code>int i;</code>
Wertzuweisung	<code>i = 23;</code>
Ausgabe	<code>System.out.println(i);</code>

(Initialisierung weggelassen)

- ▶ Programm = Liste von Anweisungen
- ▶ Reihenfolge von Anweisungen im Programm beliebig, aber (bezüglich einer Variablen) ...
 1. Definition
 2. Schreiben (links in einer Wertzuweisung)
 3. Lesen (als Ausdruck oder Teil eines Ausdrucks)

Floatingpoint-Zahlen

Motivation

- ▶ Oft werden gebraucht:
 - ▶ gebrochene Werte (zum Beispiel $\pi = 3.141592\dots$)
 - ▶ sehr große oder sehr kleine Werte (zum Beispiel 10^{23} , 10^{-34})
- ▶ Mit ganzen Zahlen umständlich oder überhaupt nicht ausdrückbar
- ▶ Zweiter numerischer Typ **Floatingpoint-Zahlen** (= „Gleitkommazahlen“, „Fließkommazahlen“)
- ▶ Bezeichnung mit reserviertem Wort `double`, gleichberechtigt zu `int`

Numerische Datentypen

Java kennt weitere numerische Typen außer `int` und `double`, die seltener gebraucht werden:

<code>byte</code>	ganze Zahlen zwischen -128 und $+127$
<code>short</code>	ganze Zahlen zwischen -32768 und $+32767$
<code>long</code>	ganze Zahlen zwischen -9223372036854775808 und $+9223372036854775807$
<code>float</code>	Floatingpoint-Zahlen mit geringerer Genauigkeit

Floatingpoint-Numerale

- ▶ Schreibweise von Floatingpoint-Numeralen: *Wenigstens* eines der folgenden Merkmale:
 - ▶ **Nachkommaanteil**, mit einem Dezimalpunkt abgetrennt:
3.14, 0.001, -123.04, 21_200.0
 - ▶ **Zehnerexponent**, mit E oder e markiert (E kann gelesen werden als „mal-zehn-hoch“):
1E23 ($1 \cdot 10^{23}$)
1e-34 ($1 \cdot 10^{-34}$)
6.670E-11 ($6.670 \cdot 10^{-11}$)
-4.17e-4 ($-4.17 \cdot 10^{-4}$)
Zehnerexponenten immer ganzzahlig, mit optionalem Vorzeichen
 - ▶ **Floatingpoint-Suffixe** D oder d erklären jedes Numeral zum Floatingpoint-Numeral:
1D, -234d, 0.001D, 1e-34d

Floatingpoint-Numerale (2)

- ▶ Mehrere Schreibweisen des gleichen Wertes möglich (Gegensatz zu `int`):

20.5

0.0205E3

205_000E-4

- ▶ Beispiele für Typen von Numeralen:

20 `int`

20.0 `double`

20E0 `double`

20.E0 `double`

20D `double`

- ▶ Rechnerisch gleiche `double`- und `int`-Numerale im Quelltext **nicht** beliebig austauschbar

Floatingpoint-Variablen

- ▶ Variablen können ganze Zahlen oder Floatingpoint-Werte speichern
- ▶ Typ einer Variablen bei der Definition festgelegt:
`int i;`
`double d;`
- ▶ Der Typ kann sich nicht ändern, der Wert sehr wohl
- ▶ Modifier `final` und Initialisierung unabhängig vom Typ, bei allen Variablen, auch `double`:
`final double pi = 3.14;`

Polymorphismus

- ▶ Arithmetischen Operatoren arbeiten mit `int`- und `double`-Operanden
- ▶ Typ des Ergebnisses abhängig von den Operanden:
 $20/8 \rightarrow 2$
 $20.0/8.0 \rightarrow 2.5$
- ▶ Der gleiche Operator (hier `/`) löst intern unterschiedliche Mechanismen aus: **Polymorphismus** (“Vielgestaltigkeit”)
- ▶ Allgemeines Phänomen, taucht an vielen Stellen in vielen Programmiersprachen auf

int vs. double

- ▶ Floatingpoint-Arithmetik rechnerisch genauer, wozu überhaupt ganzzahlige Arithmetik?
- ▶ int-Arithmetik dennoch wichtig:
 - ▶ double-Arithmetik langsamer als int-Arithmetik
 - ▶ double-Werte brauchen mehr Platz als int-Werte
 - ▶ double-Arithmetik macht unvorhersehbare Rundungsfehler
- ▶ Beispiel Rundungsfehler:

```
// ergibt 1000.0  
System.out.println(1000.0 / 50.0 * 50.0);  
// ergibt 1000.00000000000001  
System.out.println(1000.0 / 60.0 * 60.0);
```

- ▶ **Regel:** int wenn möglich, double wenn nötig

Implizite Typkonversion $\text{int} \rightarrow \text{double}$

- ▶ Zwei Operanden gleichen Typs: Operandentyp = Ergebnistyp
- ▶ Gemischten Operandentypen: double-Ergebnis:
 - $1 + 2 \rightarrow 3$ (int)
 - $1.0 + 2 \rightarrow 3.0$ (double)
 - $1 + 2.0 \rightarrow 3.0$ (double)
 - $1.0 + 2.0 \rightarrow 3.0$ (double)
- ▶ Im zweiten und dritten Beispiel: Erst Umwandlung des int-Operanden in double, dann weiter mit zwei Operanden gleichen Typs
- ▶ Zweites Beispiel im Detail:
 - $1.0 + 2 \rightarrow 1.0 + 2.0 \rightarrow 3.0$
- ▶ Allgemein: **Implizite Typkonversion** = automatische (stillschweigende) Umwandlung eines Typs in einen anderen
- ▶ Implizite Typkonversion $\text{int} \rightarrow \text{double}$ immer dann, wenn int verfügbar, aber double gebraucht

Keine implizite Typkonversionen `double`→`int`

- ▶ Zu jedem `int`-Wert gibt es einen äquivalenten `double`-Wert, zum Beispiel

`2` → `2.0`

`-1000` → `-1000.0`

- ▶ Aber: Zu vielen `double`-Werten gibt es keinen äquivalenten `int`-Wert

`3.141_592`

`1E100`

- ▶ Deshalb: keine implizite Typkonversion von `double`→`int`
- ▶ Beispiele:

- ▶ Zulässig wegen impliziter Typkonversion `int`→`double`:

```
// implizite Typkonversion 2 → 2.0, dann Zuweisung  
double d = 2;
```

- ▶ Fehler mangels impliziter Typkonversion:

```
int i = 2.0; // Fehler!
```

Kompatibilität

- ▶ Allgemein:

Ein Typ `T` ist **kompatibel** zu einem anderen Typ `U`, wenn ein Wert vom Typ `T` einer Variablen vom Typ `U` zugewiesen werden kann

Code schematisch:

```
T t = ...;  
U u = t;    // zulässig falls T kompatibel zu U
```

`int` kompatibel zu `double` wegen impliziter Typkonversion

- ▶ Aber: `double` nicht kompatibel zu `int`
→ Kompatibilitätsbeziehung nicht symmetrisch

Explizite Typkonversionen

- ▶ Typkonversion `double`→`int` kann erzwungen werden mit einer **expliziten Typkonversion** (engl. type cast)
- ▶ Syntax allgemein `(type)expression`
- ▶ Formal ein **unärer Operator**
- ▶ Hohe Priorität, wie andere unäre Operatoren:
`(int)2.5*3` → `2*3` → 6
`(int)-2.5` → -2
`-(int)2.5` → -2
Ggf. Klammern setzen für andere Auswertungsreihenfolge:
`(int)2.5*3` → `2*3` → 6
`(int)(2.5*3)` → `(int)7.5` → 7
- ▶ Typecast liefert immer ein Ergebnis, auch wenn unsinnige und völlig falsche Ergebnisse herauskommen
`(int)1e100` → 2147483647
- ▶ Typecasts **auf Mindestmaß beschränken oder ganz vermeiden**

Wertebereiche

- ▶ Werte verbrauchen Speicherplatz.
Speicherplatz endlich \Rightarrow darstellbare Wertebereiche begrenzt
- ▶ Größe für jeden Typ fixiert

Typ	Platzbedarf
<code>int</code>	4 Byte (32 Bit)
<code>double</code>	8 Byte (64 Bit)

Wertebereich int

- ▶ Grenzwerte für int:

Grenze / Vordefinierte Variable	Wert
größter negativer Wert	
Integer.MIN_VALUE	$-2147483648 = -2^{31}$
größter positiver Wert	
Integer.MAX_VALUE	$+2147483647 = +2^{31} - 1$

Wertebereich double

- ▶ Grenzwerte für double:

Grenze	Wert	Vordefinierte Variable
größter negativer Wert	$-1.79769 \cdot 10^{308}$	<code>-Double.MAX_VALUE</code>
kleinster negativer Wert	$-4.94065 \cdot 10^{-324}$	<code>-Double.MIN_VALUE</code>
kleinster positiver Wert	$+4.94065 \cdot 10^{-324}$	<code>Double.MIN_VALUE</code>
größter positiver Wert	$+1.79769 \cdot 10^{308}$	<code>Double.MAX_VALUE</code>

- ▶ Größen und Grenzwerte unabhängig vom zugrunde liegenden System

Rechengenauigkeit

- ▶ `double` speichert etwa 16 Dezimalstellen
- ▶ Maximale Anzahl Ziffern unabhängig von der Position des Dezimalpunktes
- ▶ Weitere niederwertige Ziffern werden abgeschnitten
→ Rundungsfehler
- ▶ Beispiele: Zwischenergebnis nach der ersten (linken) ungenau:
 $1.0 + 1\text{E-}14 - 1.0 \rightarrow 9.992007221626409\text{E-}15$
 $1.0 + 1\text{E-}15 - 1.0 \rightarrow 1.1102230246251565\text{E-}15$
 $1.0 + 1\text{E-}16 - 1.0 \rightarrow 0.0$
 $1\text{E}15 + 1 - 1\text{E}15 \rightarrow 1.0$
 $1\text{E}16 + 1 - 1\text{E}16 \rightarrow 0.0$

Bereichsüberschreitung

- ▶ Ergebnisse bei Verlassen des Wertebereichs?
- ▶ Ganze Zahlen (`int`)
 - ▶ Überlauf liefert kommentarlos falsches Ergebnis:
`2_147_483_647 + 1` \rightarrow `-2147483648`
Resultat des wrap-around des `int`-Wertebereichs
- ▶ Floatingpoint-Zahlen (`double`)
 - ▶ Überlauf liefert **Fluchtwerte**:
`Double.POSITIVE_INFINITY` = $+\infty$
`Double.NEGATIVE_INFINITY` = $-\infty$
Beispiel:
`1E308 + 1E308` \rightarrow `Double.POSITIVE_INFINITY`
 - ▶ Unterlauf liefert null. Beispiel:
`5E-324/2` \rightarrow `0.0`

Division durch null

- ▶ Division durch null verhält sich unterschiedlich bei `int` und `double`
- ▶ `int`-Division durch null ist unzulässig, ebenso Modulus mit zweitem Operanden null
- ▶ Programm wird sofort abgebrochen (Exception-Handling)

```
System.out.println(1/0);    // ebenso 1%0
```

Ausgabe des Programms:

```
Exception in thread "main" java.lang.ArithmeticException:  
    at DivisionByZero.main(DivisionByZero.java:5)
```

Division durch null (2)

- ▶ double-Division durch null liefert Fluchtwerte:

```
System.out.println(1.0/0.0);    // Infinity  
System.out.println(-1.0/0.0);  // -Infinity  
System.out.println(0.0/0.0);   // NaN
```

- ▶ Fluchtwert `Double.NaN` (not a number) = undefinierter Wert
- ▶ Mit `Infinity` kann (sehr eingeschränkt) weiter gerechnet werden, aber nicht mit `NaN`
- ▶ Meist fehlerhaftes Programm, kaum jemals sinnvoll

Kommandozeilenparameter

Übergabe von Kommandozeilenparameter

- ▶ Um bei der **Programmausführung** Werte zu variieren, können Kommandozeilenparameter genutzt werden
- ▶ Kommandozeilenparameter werden beim Programmstart hinter dem Klassennamen angegeben, z.B.

```
$ java Hello Heinrich
```

oder

```
$ java Max 23 15 81
```

- ▶ nach dem Kommando java kommt immer der Klassenname
- ▶ darauf folgende "Wörter" sind Kommandozeilenparameter, z.B. Heinrich oder 23, 15 und 81
- ▶ Getrennt werden Kommandozeilenparameter durch ein oder mehrer Leerzeichen
- ▶ Soll ein einzelner Parameter ein Leerzeichen enthalten, können Sie Anführungszeichen verwenden, z.B.

```
$ java Hello "Oskar Wilde"
```

Verwendung von Kommandozeilenparameter

- ▶ Kommandozeilenparameter werden über das Array `args` an die `main`-Funktion übergeben

```
public static void main(String[] args)
```

- ▶ `args` ist der Name
 - ▶ `[]` steht für Array
 - ▶ `String[]` besagt, dass die Elemente des Arrays Strings (Zeichenketten) sind
- ▶ Um die einzelnen Werte des `args`-Array verwenden zu können, nutzen wir einen Index
- ▶ **Achtung:** Die InformatikerIn zählt ab 0 nicht ab 1
- ▶ Der erste Kommandozeilenparameter hat den Namen `args[0]`, der zweite `args[1]`, usw.

Beispiel

```
class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hallo " + args[0]);  
    }  
}
```

```
$ java Hello Hannelore  
Hallo Hannelore
```

```
$ java Hello Rudi Joop  
Hallo Rudi
```

```
$ java Hello "Rudi Joop"  
Hallo Rudi Joop
```

Kommandozeilenparameter umwandeln

- ▶ Kommandozeilenparameter sind immer Strings
- ▶ Manchmal brauchen wir z.B. eine Zahl
- ▶ Nutzen von Bibliotheksmethoden zum Umwandeln, z.B.

```
int age = Integer.parseInt(args[0]);
```

oder

```
double radius = Double.parseDouble(args[0]);
```


Ausgabe

Formatierte Ausgabe

- ▶ Neben `System.out.print` und `System.out.println` gibt es noch die *formatierte Ausgabe* mit `System.out.printf`
- ▶ `printf` bekommt als ersten Parameter einen Format-String bei dem Platzhalter für die Ausgabe verwendet werden können.
- ▶ weitere Parameter sind dann die Werte, die in die Platzhalter eingesetzt werden

Beispiel — Code

```
class Bill {
    public static void main(String[] args) {
        String product = args[0];
        double price = Double.parseDouble(args[1]);
        int amount = Integer.parseInt(args[2]);

        System.out.print(
            "Anzahl\tProdukt\tEinzelpreis\tGesamt\n");
        System.out.print(
            "=====\t=====\t=====\t=====\n");

        System.out.printf("%6d\t%s\t%11.2f\t%6.2f\n",
            amount, product, price, price * amount);
    }
}
```

Beispiel — Ausführung

```
$ java Bill Schal 12.95 3
```

Anzahl	Produkt	Einzelpreis	Gesamt
=====	=====	=====	=====
3	Schal	12.95	38.85

Bibliotheksmethoden

- ▶ Mathematische Funktionen oft gebraucht (beispielsweise Quadratwurzel, Logarithmen, trigonometrischen Funktionen)
- ▶ entsprechende Algorithmen vordefiniert, in der Laufzeitbibliothek bereitgestellt
- ▶ Laufzeitbibliothek auf allen Systemen in der gleichen Form zur Verfügung
- ▶ Funktionsumfang der Laufzeitbibliothek = **Java-API** (Java Application Programming Interface)

- ▶ Beispiele für mathematische Bibliotheksmethoden:

Funktion	Mathematik	Java
Quadratwurzel	\sqrt{x}	<code>Math.sqrt(x)</code>
Natürlicher Logarithmus	$\ln(x)$	<code>Math.log(x)</code>
Logarithmus zur Basis 10	$\log_{10}(x)$	<code>Math.log10(x)</code>
e-Funktion	e^x	<code>Math.exp(x)</code>
Sinus	$\sin(x)$	<code>Math.sin(x)</code>
Arcus-Tangens	$\arctan(c)$	<code>Math.atan(x)</code>

Argumente und Ergebnis

- ▶ Bibliotheksmethoden erwarten Argumente, liefern Funktionswert zurück
- ▶ **Argumente** in runden Klammern angeben
- ▶ Beispiel: Sinus von 0.5 (alle Winkel im Bogenmaß):
`Math.sin(0.5)` → 0.479425538604203
- ▶ Vordefinierte Konstanten für die Kreiszahl π und die Eulerzahl e :
`Math.PI`
`Math.E`
- ▶ Beispiel: Sinus von $45^\circ = \frac{1}{4}\pi$:
`Math.sin(Math.PI / 4)` → 0.7071067811865475
`Math.sin(45 * Math.PI / 180)` → 0.7071067811865475

Mehrere Argumente

- ▶ Mehrere Argumente mit Komma getrennt aufzählen
z.Beispiele:

Funktion	Mathematik	Java
Potenz	x^y	<code>Math.pow(x, y)</code>
„Pythagoras“	$\sqrt{a^2 + b^2}$	<code>Math.hypot(x, y)</code>
Vektorrichtung	$\arctan(xy)$	<code>Math.atan2(y, x)</code>

- ▶ Berechnen von 3^7 : `Math.pow(3, 7) → 2187.0`
 - ▶ Vorsicht: `Math.pow` rechnet immer mit `double`-Werten → schwer vorhersagbare Rundungsfehler
 - ▶ `Math.pow` aufwändig, damit langsam → nur bei großen oder gebrochenen Exponenten sinnvoll
- `Math.pow(2, 135) → 4.3556142965880123E40`
`Math.pow(2, 1.0/7.0) → 1.1040895136738123`

Mehrere Argumente (2)

- ▶ Länge der Hypotenuse c aus den Längen a und b der Katheten eines rechtwinkligen Dreiecks („Pythagoras“):

$$c = \sqrt{a^2 + b^2}$$

`Math.hypot(1, 2)` → 2.23606797749979

- ▶ Richtungswinkel σ (Bogenmaß!) des Vektors (x, y) :

`Math.atan2(2, 1)` → 1.1071487177940904

- ▶ Korrekt auch für $x = 0$
- ▶ Ergebnis im Intervall $-\pi \leq \sigma \leq \pi$

Geschachtelte Aufrufe

- ▶ Argumente für Bibliotheksmethoden: beliebige Ausdrücke
- ▶ Ganzer Aufruf einer Bibliotheksmethode selbst ein Ausdruck
- ▶ Geschachtelte Aufrufe zulässig:

3^{3^3}	<code>Math.pow(3, Math.pow(3, 3))</code>
$(3^3)^3$	<code>Math.pow(Math.pow(3, 3), 3)</code>
$\sin(0.5^2)$	<code>Math.sin(0.5 * 0.5)</code>
$\sin(0.5)^2$	<code>Math.sin(0.5) * Math.sin(0.5)</code>
$\sin^2(0.5) = \sin(\sin(0.5))$	<code>Math.sin(Math.sin(0.5))</code>

Bibliotheksmethoden vs. einfache Implementierung

- ▶ Einfachere Konstrukte oft effizienter \Rightarrow Bibliotheksmethoden mit Bedacht einsetzen
- ▶ Beispiele:

Ausdruck	langsam	schnell	Verhältnis
4.1^3	<code>Math.pow(4.1, 3)</code>	<code>4.1*4.1*4.1</code>	~ 100
$10^{0,5}$	<code>Math.pow(10, 0.5)</code>	<code>Math.sqrt(10)</code>	~ 20