PM1/PT1 Ruby

Zahlen und arithmetische Ausdrücke

Einführung

- Wir unterscheiden in Ruby zwischen
  - → Ganzen Zahlen
  - → Gleitkommazahlen

- **Zahlen** sind in Ruby **Objekte** der zugehörigen Zahlenklassen, wir rufen wie auch für andere Objekte **Methoden** auf Zahlen auf, insbesondere auch dann, wenn wir mit ihnen arithmetische Operationen ausführen.

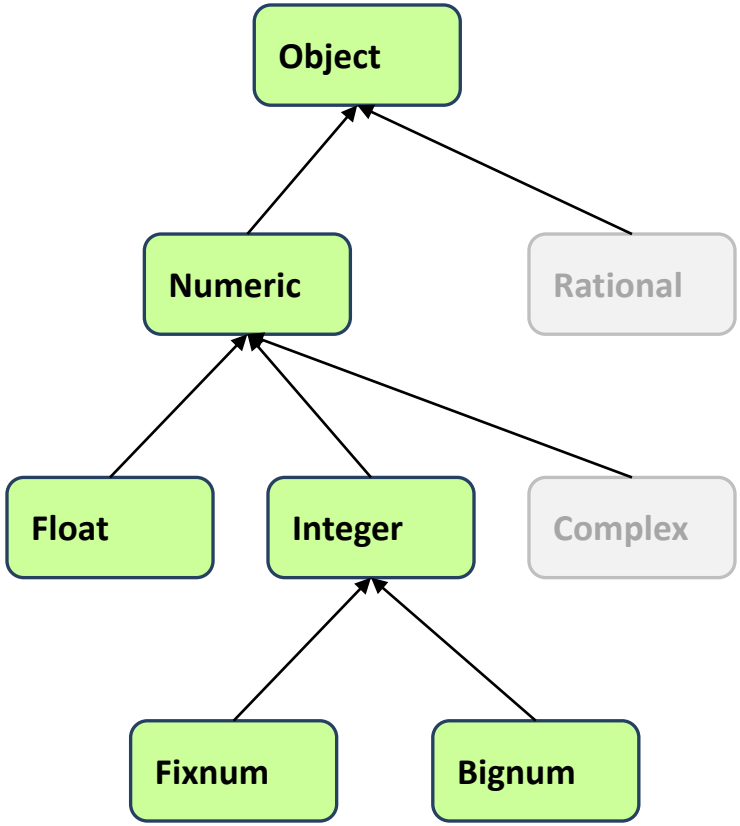
- Ganze Zahlen (***Fixnum***) und Gleitkommazahlen (***Float***) haben einen definierten Wertebereich. Jenseits dieses Wertebereichs sind diese Zahlen nicht darstellbar.

```
5
-205
99999999999999999999999999999
999999
0

54.321
0.001
-205.3884
0.0
0.987E-12
0.76e7
```

Hierarchie der Zahlenklassen

- ***Numeric*** repräsentiert die Zahlen in Ruby
- ***Integer*** ist eine gemeinsame Superklasse der ganzen Zahlen
  - ***Fixnum*** sind die ganzen Zahlen, die in einer Prozessorarchitektur dargestellt werden können.
  - ***Bignum*** sind alle ganzen Zahlen, die kleiner der kleinsten Fixnum oder größer der größten Fixnum sind. Die Umwandlung von Fixnum in Bignum und zurück wird Ruby-intern übernommen.
- ***Float:*** Gleitkommazahlen sind Näherungen für die reellen Zahlen.



Zahlen und ihre Wertebereiche

Klasse	Wertebereich / Konstanten / Darstellung	Bemerkung
Fixnum	-2^{31} … $+2^{31}-1$ -2^{63} … $+2^{63}-1$	
Bignum	Zahlen größer / kleiner Fixnum	Softwaredarstellung für Zahlen Konvertierung zwischen Fixnum und Bignum durch den Ruby implizit
Float		Double in der jeweiligen Prozessorarchitektur

Implizites Konvertieren zwischen ***Bignum*** und ***Fixnum***

- Überschreitet eine Zahl die maximal / minimal darstellbare ***Fixnum***, dann konvertiert Ruby diese Zahl in eine ***Bignum***.

- Liegt die ***Bignum*** im Wertebereich der ***Fixnum***, dann wird die ***Bignum*** intern wieder in eine ***Fixnum*** konvertiert.

```
num = 8

7.times do
  puts("#{num.class} #{num}")
  num *= num
end

7.times do
  puts("#{num.class} #{num}")
  num =
   Integer(Math::sqrt(num))
end
```

Darstellen von Zahlen

- ***Numeric*** ist eine Abstraktion für Zahlen: es gibt keine Darstellung von ***Numeric-Objekten***

- ***Integer***: abhängig von der Zahlenbasis unterschiedliche Darstellungen:

  1456 oder ***-896761*** (dezimal Basis 10)

  0b1001 oder ***-0b1001*** (binär Basis 2)

  01456 oder ***-07346*** (oktal Basis 8)

  0x9AF3 oder ***-0x65CDE3*** (hexadezimal Basis 16).

- Zahlen werden ausschließlich durch **Zahlenliterale** erzeugt. Es gibt für die Zahlenklassen keine Methode ***new()***

- ***Float***: Es existieren verschiedene Schreibweisen für Zahlenliterale

  12.34

  -.1234e2

  1234e-2

  1234.e3 Fehler: Hier wird versucht die Methode e3 auf 1234 aufzurufen.

Konvertieren zwischen Zahlen / zwischen Zahlen und String

- Mit den Methoden ***Integer*** und ***Float*** können wir aus Strings Zahlen erzeugen.

```
puts Integer("89") #=> 89
puts Float("1.78") #=> 1.78
```

- Die Konvertierung von Strings zu Zahlen benötigen wir, wenn wir
  - Benutzereingaben von der Konsole oder aus Eingabefeldern der ***Toolbox*** lesen
  - Dateien lesen.
- Eingaben werden immer als String gelesen.
- Erwarten wir eine Zahl, dann müssen wir den ***String*** zunächst in eine Zahl wandeln.
- Achtung: Wenn der String keine gültigen Wert für ***Integer*** / ***Float*** enthält, dann erzeugen diese Konvertierungen einen Fehler und brechen das Programm ab.

Strings in Zahlen konvertieren

```
while true
  puts("Geben Sie \'end\' ein, um das Programm zu beenden")
  puts("Bitte eine ganze Zahl eingeben")
  z_string = gets().chomp()

  if (z_string.downcase() =="end")
    break
  end

  z1= Integer(z_string)
  puts("Bitte einen Operator eingeben")
  op = gets().chop()
  puts("Bitte eine ganze Zahl eingeben")
  z_string2 = gets().chomp()
  z2 = Integer(z_string2)
  puts(z1.method(op.to_sym()).call(z2) )
end
```

Arithmetische Ausdrücke

- Durch Verknüpfung von Zahlen mit mathematischen Operatoren entstehen **arithmetische Ausdrücke**.
- Der **Wert** eines arithmetische Ausdrucks ist immer eine Zahl.
- Daher können arithmetische Ausdrücke auch beliebig **hintereinander** angewendet werden und beliebig **geschachtelt** sein.
- In **Ruby** sind arithmetische Ausdrücke das Verketten mehrerer auch hintereinandergeschalteter Methodenaufrufe auf Zahlenobjekten.
- Die mathematische Operationen sind als Methodenimplementierungen realisiert.

- **Ü3-d-1: *5+8* ist ein einfacher arithmetischer Ausdruck. Schreiben Sie einen komplexen arithmetischen Ausdruck mit 5 verschiedenen Operatoren und einer Schachtelungstiefe von 3.**

Alle mathematischen Operatoren sind Methoden

- **+, -, *, /, ** ** sind Methoden

- Die Schreibweisen rechts zeigen den Aufruf dieser Methoden auf Zahlenobjekten
  - in der Kurzform ohne Klammerung der Parameter und ohne den für den Methodenaufruf typischen Punkt
  - mit Klammerung der Parameter

- Die Schreibweisen sind äquivalent.

```
5+8.9
5.+(8.9)

5-8.9
5.-(8.9)

5*8.9
5.*(8.9)

5/8.9
5./(8.9)

5**8.9
```

Modulo Operator

div und *mod (%)*

- Für die Definition von ***div*** sind zwei Varianten gebräuchlich.
- Die Variante nach Donald Knuth:
  x div y = floor(x/y)
- Die Variante nach Nikolaus Wirth:
  x div y = trunc(x/y)
- Beide Varianten liefern das gleiche Ergebnis, wenn ***x*** und ***y*** das gleiche Vorzeichen haben. Allerdings unterscheiden sie sich im Ergebnis, wenn eines der beiden Argumente ***x*** oder ***y*** negativ ist.
- Abhängig von der ***div*** Definition ist das Ergebnis von ***mod*** unterschiedlich.
- ***mod*** oder ***%*** ist für beide Varianten gleich definiert

  x mod y = x – y*(x div y)
  x % y = x – y*(x div y)
- In Ruby ist für ***div*** und ***(/)*** die Knuthsche Variante implementiert. In anderen Sprachen z.B. in Java ist die Wirthsche Variante implementiert.
- Ruby bietet neben ***mod*** noch die Methode ***remainder***, diese liefert die Ergebnisse der Wirtschen Variante

divmod und *div (/)* und *mod (%)* und *remainder*

x	y	x.divmod(y)	x / y	x.modulo(y)	x.remainder(y)
13	4	[3, 1]	3	1	1
13	-4	[-4, -3]	-4	-3	1
-13	4	[-4, 3]	-4	3	-1
-13	-4	[3, -1]	3	-1	-1
11.5	4	[2, 3.5]	2.875	3.5	3.5
11.5	-4	[-3, -0.5]	-2.875	-0.5	3.5
-11.5	4	[-3, 0.5]	-2.875	0.5	-3.5
-11.5	-4	[2, -3.5]	2.875	-3.5	-3.5

*x mod y= x – y*floor(float(x)/float(y))*
x remainder y = x – y (trunc(float(x)/float(y))

Methoden für Zahlen

Beschreibung	Methoden
arithmetisch unäre	+, -, abs
arithmetisch binär	+,-,*, div (nutzt die „/“ Methode von Integer und Float)
"Runden“	ceil, floor, round, truncate
Prüfen auf Gleichheit	==, eql?, <=>
Tests	integer?, nonzero?, zero?
Divisionsrest	%,modulo, remainder
Konvertierung	to_int
Iteratoren	step (***später***)

Methoden

Methode	Erklärung
anum.ceil	liefert die kleinste ganzzahlige Zahl größer als anum
anum.floor	liefert die größte ganzzahlige Zahl kleiner als anum
anum.round	rundet anum
anum.truncate	schneidet die Nachkommazahlen von anum ab
anum1 <=> anum2	Spaceship Operator: Ergebnis ist 0 wenn anum1 == anum2 , und nil sonst
a_num % a_int a_num. modulo(a_int)	Divisionsrest (siehe Beispiel)

Methoden für Zahlen

Methode	Beispiel
anum.ceil	1.ceil #=> 1 1.2.ceil #=> 2 (-1.2).ceil #=> -1 (-1.0).ceil #=> -1
anum.floor	1.floor #=> 1 (-1.2).floor #=> -2
anum.truncate	1.2.truncate #=> 1 (-1.2).truncate #=> -1
a_num % (a_int) a_num modulo(a_int)	liefert den ganzzahligen Divisionrest (siehe nächste Folie)
a_num .divmod(a_int)	liefert eine Array aus Quotient und Divisionsrest. Enthält für den Quotienten immer die größte Zahl kleiner der Floating-Point Division x/y

Methoden für **Integer**

Beschreibung	Methoden
Methoden, die die Zahl selbst zurückgeben: Da die Konvertierung in einen Integer immer den Integer selbst ergibt und eine ganze Zahl „gerundet“ immer die Zahl selbst ergibt.	to_i, to_int, floor, ceil, round, truncate
Konvertierung	chr (wandelt einen Integer in die Characterdarstellung um)
Iteratoren	downto, upto, times (später)

Methoden für **Integer**

Methoden	Beispiele
to_i, to_int, floor, ceil, round, truncate	34.to_i #=> 34 34.to_int #=> 34 34.floor() #=> 34 34.truncate() #=> 34
an_int.chr	65.Chr #=> "A" ?a.chr #=> "a" 230.chr #=> "\346"

Methoden für **Fixnum**

Beschreibung	Methoden
arithmetisch unäre	+, -, abs
arithmetisch binär	+,-,*, /, /(num), ** (Exponentialfunktion)
Vergleich	<, <=, >, >=, <=> (Spaceship-Operator x <=> y ist -1 wenn x < y, ist 0 wenn x == y, ist 1 wenn x > y)
Gleichheit	== (gleiche Werte), eql? (gleiche Werte und gleiche Klasse)
Divisionsrest	%,modulo, remainder
Konvertierung	to_f, to_s(basis)

Methoden für **Fixnum**

Methode	Beispiel
fix/(num) fix.quo(num)	4/(3) #=> 1.33333… 4.quo(3) #=> 1.33333…
fix**num fix.power!(num)	2**3 #=> 8 2.power!(3) #=> 8
fix <=> anum	2 <=> -1 #=> 1 2 <=> 2.0 #=> 0 2 <=> 4 #=> -1
fix == anum	2 == 2.0 #=> true
fix eql? anum	2.eql?(2.0) #=> false
fix.to_f	2.to_f #=> 2.0
fix.to_s(basis=10)	2.to_s(2) #=> "10" 2.to_s() #=> "2" 15.to_s(16) #=> "F"

Methoden für **Float**

Beschreibung	Methoden
arithmetisch unäre	+, -, abs
arithmetisch binär	+,-,*, /, **
"Runden“	ceil, floor, round, truncate und to_i, to_int haben den selben Effekt)
Gleichheit	== (gleiche Werte), eql? (gleiche Werte und gleiche Klasse)
Vergleich	<, <=,>,>=, <=>
Tests	finite?, infinite?, nan?, zero?
Divisionsrest	%, modulo, remainder
Konvertierung	to_f, to_s, to_int, to_i, truncate

Methoden für **Float**

Methode / Beschreibung	Beispiel
a_float.ceil (kleinste ganzzahle Wert größer a_float)	12.3.ceil() #=> 13 -12.3.ceil() #=> -12
a_float.floor (größte ganzzahle Wert kleiner a_float)	12.3.ceil() #=> 12 -12.2.floor() #=> -13
a_float.truncate a_float.to_i a_float.to_int (Abschneiden Nachkommastellen)	12.3.truncate() #=> 12 -12.3.truncate() #=> -12
a_float.finite? (nicht infinite? und nan? ist false)	12.3.finite? #=> true 0.0/0.0.finite? #> false
a_float.infinite? (nil, wenn a_float finit ist, -1, wenn a_float - ∞ wenn a_float +∞)	(0.0).infinite? #=> nil (-1.0/0.0).infinite? #=> -1 (+1.0/0.0).infinite? #=> 1
a_float.nan? (true, wenn a_float keine gültige Zahl ist)	a = -1.0 #=> -1.0 a.nan? #=> false a = 0.0/0.0 #=> NaN a.nan? #=> true

Rundungsfehler bei Rechnen mit Float

- ***Float*** ist nur eine Näherung für die reellen Zahlen, im Rechner.
- Die Rechengenaugkeit ist durch das kleinste darstellbare Intervall zwischen zwei Zahlen bestimmt.
- Wir z.B. bei einer Division diese Genauigkeit unterschritten, dann liefert die anschließende Multiplikation mit dem Wert durch den zuvor dividiert wurde zwar eine Zahl nahe 1, aber != 1.

```
1.upto(99) do |i|
   z = 1.0 / i
   puts("z=" +z.to_s)
   if (z * i != 1.0)
     puts("Fehler fuer " + i.to_s)
     break
   end
end
```

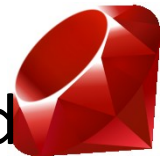
Rundungsfehler bei Rechnen mit Float

- Daher muss beim Vergleich von ***Float***-Zahlen immer auf einen **Epsilon**-Bound verglichen werden.

- Ein **Epsilon**-Bound ist eine sehr kleine Zahl, die eine Abweichung zwischen zwei ***Float-Zahlen*** beschreibt.

```
1.upto(99) do |i|
   z = 1.0 / i
   if (z * i != 1.0)
     puts("Fehler fuer " + i.to_s)
     z = z *i
     puts(1.0-z)
     break
   end
end
puts
puts("und nun mit epsilon-bound")
1.upto(99) do |i|
   z = 1.0 / i
   if ((z * i) < 2e-16)
     puts("Fehler fuer " + i.to_s)
     z = z *i
     puts(1.0-z)
     break
   end
end.
```

Trigonometrische und Transzendente Methoden und mathematische Konstanten im Math Modul



- das **Math** Modul enthält Klassenmethoden und Klassenkonstanten (**das später genauer)**
- Die Methoden werden verwendet, indem der Methode / Konstante der Name des Moduls **Math** vorangestellt wird (Beispiele siehe nächste Folie)
- Konstanten werden verwendet, indem der Konstante er Modulname mit zweifachem Doppelpunkt vorangestellt wird.
- das **Math** Modul gehört zum Standardsprachumfang in Ruby und kann direkt genutzt werden.

Kategorie	Methode
trigonometrische	sin, cos, asin, acos, tan, atan etc…
transzendente	exp, log, log10, sqrt etc…
Konstanten	PI, E

Das *Math* Objekt

- **::** ist der Scope Operator (später).
- ***Math*** kann intuitiv, wie gezeigt benutzt werden

```
puts(Math::PI)                  3.14159265358979
puts(Math::E)                   2.71828182845905
puts(Math.cos(Math::PI/3))      0.5
puts(Math.tan(Math::PI/4))      1.0
puts(Math.log(Math::E**2))      2.0
puts((1 + Math.sqrt(5))/2)      1.61803398874989
```

(Pseudo)Zufallszahlen

```
puts rand                                          0.62586024289918
puts rand                                          0.886284096103342
puts rand                                          0.917206740280718
puts(rand(100))                                    92
puts(rand(100))                                    74
puts(rand(100))                                    88
puts(rand(1))                                      0
puts(rand(1))                                      0
puts(rand(1))                                      0
puts('There is a '+rand(101).to_s+'% chance of rain.')   There is a 56% chance of rain.
```

- Pseudozufallszahlen werden nach einem deterministischen mathematischen Verfahren erzeugt. Die Zahlenfolge hängt vom Startwert (Saat engl. **seed**) ab. Bei gleichem Startwert wird immer die gleiche Folge von Zufallszahlen erzeugt.

- rand: erzeugt Pseudozufallszahlen mit zufällig gewähltem Startwert.
- rand erzeugt Pseudozufallszahlen x im Intervall: 0.0 <= x < 1.0.
- rand(*grenze*) erzeugt Pseudozufallszahlen x im Intervall: 0 <= x < grenze
- *grenze* ist nicht im Wertebereich der generierten Zahlen. Verwende *grenze + 1*

Wiederholbare Folgen von (Pseudo)Zufallszahlen

```
srand 1776
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts ''
srand 1776
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
```

ergibt

```
24
35
36
58
70

24
35
36
58
70
```

- um gleiche Folgen von Zufallszahlen zu erzeugen, muss der Startwertangegeben werden.

  → verwende **srand(*startwert*)**

- ***srand(0)***

  → setzt das Verhalten des Zufallszahlengenerators auf das normale Verhalten zurück

Übungen

- **Ü3-d-1:** Lesen Sie solange ganze Zahlen von der Konsole ein, bis der Benutzer „exit“ eingibt. Addieren Sie die eingegebenen Zahlen sukzessive und geben Sie das Ergebnis nach Eingabe von „exit“ auf der Konsole aus. Es sollen alle Schreibweisen von „exit“ erkannt werden. Führende und endende Leerzeichen sollen für Zahlen und die Eingabe von „exit“ möglich sein. Wird weder „exit“ noch eine korrekte ganze Zahl eingegeben, dann soll das Programm abbrechen.
- **Ü3-d-2:** Modifizieren Sie **Ü-3-d-1** wie folgt. Eine Zufallszahl zwischen 0 und 4 (inkl.) bestimmt die arithmetische Methode, die auf das letzte Ergebnis und die aktuelle Eingabe angewendet wird. Es soll keine ganzzahlige Division verwendet werden.