LV 4342
Skriptsprachen-Praktikum
Übung 04

Vererbung

Mixins, Module

Organisatorisches

- Arbeitsverzeichnis:

  `~/lv/skriptspr/04/`

- Dateinamen:

```
04-arithmetik.rb      # neu erstellen & abgeben
```

- Werkzeuge:

```
ruby                  # Der Interpreter
irb                   # Interactive Ruby-Shell
emacs, xemacs         # mit Ruby-Mode
scite                 # Ein portabler Editor, auch
                      # mit Ruby-Mode
ri                    # Ruby-Dokumentation auf der Kommando-
                      # zeile
komodo                # IDE für Ruby u.a. Skriptsprachen
```

- Vorlagen:

  `(keine)`

Die Aufgabe

- Rund um das Thema "Grundrechenarten zu beliebigen Zahlenbasen" üben wir:
  - Ableitung neuer (Zahlen-)Klassen per Vererbung,
  - Nutzung gemeinsamer Methoden,
  - Verwendung von Mixins

  (Beachten Sie die Ähnlichkeiten mit "Complexnum" aus der Vorlesung!)

- Hinweise zum Rechnen in beliebigen Stellenwertsystemen:
  - Intern repräsentieren Sie die Zahl einfach als Integer
  - Falls die Zahl als String-Repräsentation und nicht bereits als ein Integer-Objekt übergeben wird, müssen Sie geeignet umwandeln (Horner-Schema!)
  - Die gewünschten Rechenoperationen delegieren Sie an die gewöhnlichen Integer-Operationen
  - Achten Sie darauf, dass z.B. die Addition zweier IntN-Objekte wieder ein IntN-Objekt ergeben muss – und nicht etwa ein Fixnum-Objekt!
  - Die Implementierung der Methode `to_s` erfordert ebenfalls Code für einen Darstellungswechsel, bei dem erneut ein Horner-Schema hilfreich ist.

Die Aufgabe

A: Implementieren Sie Klasse "IntN" mit den Standardmethoden:

```
initialize(value, base)     --> aIntN
+(op)                       --> aIntN
-(op)                       --> aIntN
*(op)                       --> aIntN
/(op)                       --> aIntN
%(op)                       --> aIntN
+@                          --> aIntN
-@                          --> aIntN
to_i                        --> anInteger
to_s                        --> aString
```

B: Leiten Sie von Klasse "IntN" die Klassen "Int2" und "Int16" ab. Sie sollen dieselben Operatoren unterstützen wie "IntN", aber mit voreingestellter Basis 2 bzw. 16 arbeiten:

```
initialize(value)           --> aInt2 / aInt16
```

Die Aufgabe

C: Testen Sie die Korrektheit der Grundrechenarten

```
a0 = Int2.new(0), ..., a9 = Int2.new(9)
b0 = Int16.new(0), ..., b9 = Int16.new(9)
puts a5+a6    # 1011_2
puts b5+b6    # B_16
puts b9+b9    # 12_16   (usw. ...)
```

D: Implementieren Sie in Klasse "IntN" den Vergleichsoperator:

```
<=>(op)                          --> -1 oder 0 oder +1
```

"Mixen" Sie nun das Modul "Comparable" in Ihre Klasse ein mittels

```
include Comparable
```

direkt unterhalb "class ...", oder leiten Sie die Klasse "IntN" ab von der Klasse "Numeric" (mit analoger Wirkung und noch mehr)

Auf diese Weise übernehmen Sie die Ordnungsrelation aus $\mathbb{Z}$.

```
# Test:
a5+a6  == 11  # true
```

Die Aufgabe

- **`initialize(value, base)`**
  - base: Ein Integer aus dem Bereich 2..36 (10 Ziffern + 26 Buchst.)
  - value: Entweder Integer oder ein String
    Falls String: Erlaubtes Format (Reg. Ausdruck) =
    `/^[+-]?[0-9A-Za-z]+$/`
- **`to_s`**
  - Grundlage: Stellenwertsystem,
    mit Ziffernwerten 0..9 = 0..9, A=10, B=11, … Z=35
  - Vorzeichen ggf. als Präfix
  - Basis (Dezimal) als Suffix anhängen, getrennt mit '_'
  - Beispiele:

```
Int2.new(13).to_s          # "1101_2"
Int2.new(-1).to_s          # "-1_2"
IntN.new(1000, 20).to_s    # "2A0_20"
IntN.new("-j6", 20).to_i   # -386
```

Die Aufgabe

E: Tests: Stehen Ihnen nun die üblichen Vergleichsoperatoren <, >, >= etc. auch in den Klassen Int2 und Int16 zur Verfügung?

Was ergibt z.B. a2 < a3, b9 < b6 ?

F: Mischfälle:

Was passiert bei b9 >= a8 ? Ist das sinnvoll?

Was ergibt b9 + 2 ? Was sollte es ergeben?

Was passiert bei 2 + b9?

G: Sortieren

Funktioniert `[a8, b4, a7, b1, b2].sort` ?

G: Testen Sie in Klasse "IntN" sowie in den abgeleiteten Klassen die Division und Modulorechnung (/ und %)

Ist die Methode „divmod“ verfügbar? Falls ja: Warum?

Die Aufgabe

H: Ergänzen Sie ein eigenes kleines Modul "Extras"

Es enthält als einzige Methode "fact" - Berechnung der Fakultät einer Zahl. Hier der "Rahmen":

```
module Extras # Analog zu "class ..."
  def fact
# ... Hier Ihre Implementierung.
# Hinweis/Vereinfachung: Ignorieren Sie Fall "0"
  end
end
```

Mixen Sie es mittels "include" (analog zu "Comparable") in die Klassen "IntN" und "Integer" ein!

Testen Sie nun das Konzept des "*implementation sharing*":

```
puts 5.fact   # Sollte 120 ergeben
puts a5.fact  # Sollte 1111000_2 ergeben
puts b5.fact  # Was erwarten Sie … ?
```

Hinweise

- Material, Hinweise:
  - In einer Methode rufen Sie die gleichnamige Methode der Basisklasse ggf. auf mit "`super`" (anstelle des Methodennamens).
  - Die Fakultät n! einer natürlichen Zahl implementiert man häufig rekursiv, denn 1! = 1, n! = n * (n-1)!
- Abgabe
  - Schreiben Sie erst den Code für Klassen und Module in Ihre Datei.
  - Am Dateiende schreiben Sie Code, der die Ergebnisse der o.g. Tests sowie weiterer Tests nach Ihrer Wahl in lesbarer, aber einfacher Form ausgibt. Alternativ bilden Sie eine Testsuite mittels „test/unit“, s.u.
  - Leerzeilen und Zeilen mit Titeln für neue Abschnitte machen Ihre Testausgaben lesbarer. ;-)
  - Bemerkungen, Fragen etc. schreiben Sie einfach als Kommentarabschnitte in Ihre Datei.
  - Ihre abgegebene Datei **04-arithmetik.rb** sollte ausführbar sein und frei von Syntaxfehlern. Für diese Übung gibt es **2 Punkte**!

Hinweise

- **Beispielcode zu Testsuites (Unit tests):**

```
if __FILE__ == $0
  require "test/unit"

   class MyTests < Test::Unit::TestCase
    def setup
      @a3, @a4 = Int2.new(3), Int2.new(4)
      @b3, @b4 = Int16.new(3), Int16.new(4)
     end

    def test_strichrechnung
      assert_equal Int2.new(7), @a3+ @a4
      assert_equal "-4_16", (-@b4).to_s  # etc.
     end
     # Weitere Testmethoden …
  end
end
```

- **Weitere „*assertions*“:**
  - **assert, assert_not_equal, assert_raise, assert_nothing_raised, assert_instance_of, …**