Praktikum zu LV 7328 - Ruby:
Übung 03

Vererbung
Mixins, Module

Organisatorisches

- Arbeitsverzeichnis:
  `~/kurse/ruby/03/`
- Dateinamen:
  `03-modnum.rb` `# neu erstellen & abgeben`
- Werkzeuge:
  `ruby` `# Der Interpreter`
  `emacs` `# mit Ruby-Mode. NICHT X-Emacs`
  `scite` `# Ein portabler Editor, auch`
  `# mit Ruby-Mode`
- Vorlagen:
  `(keine)`

Die Aufgabe

- Rund um das Thema "Grundrechenarten in endlichen Körpern" 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 Modulo-Rechnen:
  - K_n = ([0, ..., n-1], +, *) bildet einen endlichen Körper, wenn man definiert:

```
+(a, b) := (a + b) % n      # "modulo n"
*(a, b) := (a * b) % n      # "modulo n"
```

  - Subtraktion ist definiert mittels des inversen Elements der Addition:

```
-(a, b) := +(a, -b)
  mit -b so, dass b + (-b) == 0
```

  - Division ist definiert mittels des inversen Elements der Multiplikation (Kehrwert), falls n prim und b > 0:

```
/(a, b) := *(a, inv(b))
  mit inv(b) so, dass b * inv(b) == 1
```


Die Aufgabe

- Beispiele zum Modulo-Rechnen:
  - K_n mit n=10 (kein Körper!):

```
2+3=5; 5+5=0; 5+6=1; 9+9=8
2-3=2+7=9; 5-6=5+4=9
2*3=6; 5*5=5; 5*6=0; 9*9=1
recip(3) = 7; recip(2) ex. nicht!
```

  - K_n mit n=11 (Körper!):

```
2+3=5; 5+5=10; 5+6=0; 9+9=7
2-3=2+8=10; 5-6=5+5=10
2*3=6; 5*5=3; 5*6=8; 9*9=4
recip(3)=4; recip(2)=6,
  daher: 5/3=5*4=9, 10/2=10*6=5
```

  - Nutzen Sie diese Beispiele zum Auffrischen Ihrer Kenntnisse über Modulo-Rechnen und zum Testen Ihrer Implementierungen.

Die Aufgabe

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

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

B: Leiten Sie von Klasse "Modnum" die Klassen "Modnum10" und "Modnum11" ab. Sie sollen dieselben Operatoren unterstützen wie "Modnum", aber mit voreingestellter Basis 10 bzw. 11 arbeiten:

```
initialize(value)           --> aModnum
```

C: Tests: Reproduzieren Sie die Ergebnisse der Seite zuvor:

```
a2 = Modnum10.new(2), ..., a9=Modnum10.new(9)
b2 = Modnum11.new(2), ..., b10=Modnum11.new(10)
puts a5+a5    # 0
puts b5+b5    # 10    (usw. ...)
```


Die Aufgabe

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

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

und "mixen" Sie das Modul "Comparable" in Ihre Klasse ein mittels

```
include Comparable
```

direkt unterhalb "class ...".

E: Tests: Stehen Ihnen nun die üblichen Vergleichsoperatoren <, >, == etc auch in den Klassen Modnum10 und Modnum11 zur Verfügung?
Was ergibt z.B. a2 < a3, b9 < b6, was passiert bei b10 >= a9 ??

F(*): Implementieren Sie in Klasse "Modnum11" Kehrwert und Division:

```
recip                       --> aModnum11
/(op)                       --> aModnum11
```

und testen Sie die Wirkung

Hinweis: Es genügt hier, den Kehrwert per linearer Suche schlicht durch Ausprobieren zu ermitteln. Ein optimierter Algorithmus ist nicht gefragt.

Die Aufgabe

G: Schreiben 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 "Modnum" und "Integer" ein!

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

```
puts 5.fact  # Sollte 120 ergeben
puts a5.fact # Sollte 0 ergeben
puts b5.fact # Weder 0 noch 120, sondern?
```


Hinweise

- Material, Hinweise:
  - In einer Methode rufen Sie die gleichnamige Methode der Basisklasse ggf. auf mit "`super`" (anstelle des Methodennamens).
  - Tip zu Teil F: "Range" besitzt eine Methode "find"...
  - 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.
  - Leerzeilen und Zeilen mit Titeln für neue Abschnitte machen Ihre Testausgaben lesbarer. ;-)
  - Bemerkungen, Fragen etc. schreiben Sie einfach als Kommentarabschnitte in Ihre Datei.
  - Ihre Datei **03-modnum.rb** sollte ausführbar sein und frei von Syntaxfehlern.