



# Praktikum zu LV 7328 - Ruby:

## Übung 04

Vererbung  
Mixins, Module



## Organisatorisches



- Arbeitsverzeichnis:  
`~/lv/ruby/04/`
  - Dateinamen:  
`04-modnum.rb` # neu erstellen & abgeben
  - Werkzeuge:  
`ruby` # Der Interpreter  
`irb` # Interactive Ruby-Shell  
`emacs` # mit Ruby-Mode  
`scite` # Ein portabler Editor, auch  
# mit Ruby-Mode  
`ri` # Ruby-Dokumentation auf der Kommando-  
# zeile
- 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:
  - $K_n = (\{0, 1, \dots, n-1\}, +, *)$  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  $b > 0$ :
    - $/(a, b) := *(a, \text{inv}(b))$
    - mit  $\text{inv}(b)$  so, dass  $b * \text{inv}(b) == 1$



## Die Aufgabe



- Beispiele zum Modulo-Rechnen:
  - $K_n$  mit  $n=10$  (kein Körper!):
    - $2+3 = 5;$        $5+5 = 0;$        $5+6 = 1;$        $9+9 = 8$
    - $2-3 = 2+7 = 9;$        $5-6 = 5+4 = 9$
    - $2*3 = 6;$        $5*5 = 5;$        $5*6 = 0;$        $9*9 = 1$
    - $\text{recip}(3) = 7;$        $\text{recip}(2)$  ex. nicht!
  - $K_n$  mit  $n=11$  (Körper!):
    - $2+3 = 5;$        $5+5 = 10;$        $5+6 = 0;$        $9+9 = 7$
    - $2-3 = 2+8 = 10;$        $5-6 = 5+5 = 10$
    - $2*3 = 6;$        $5*5 = 3;$        $5*6 = 8;$        $9*9 = 4$
    - $\text{recip}(3) = 4;$        $\text{recip}(2) = 6,$
    - 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



## Rechnen im $K_{10}$

## Rechnen im $K_{11}$

a	b	a + b	-b	a - b	a * b	rec ip (b)	a / b
2	3	5	7	9	6	7	-
5	5	0	5	0	5	-	-
5	6	1	4	9	0	-	-
9	9	8	1	0	1	9	-

a	b	a + b	-b	a - b	a * b	rec ip (b)	a / b
2	3	5	8	10	6	4	8
5	5	10	6	0	3	9	1
5	6	0	5	10	8	2	10
9	9	7	2	0	4	5	1

$K_{10}$  ist kein Körper!



# Die Aufgabe



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

```

initialize(value, base)    --> 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 "mischen" Sie das Modul "Comparable" in Ihre Klasse ein mittels

```
include Comparable
```

direkt unterhalb "class ...". Übernehmen Sie die Ordnungsrelation aus N.

E: Tests: Stehen Ihnen nun die üblichen Vergleichsoperatoren <, >, == etc auch in den Klassen Modnum10 und Modnum11 zur Verfügung?

Was ergibt z.B.  $a_2 < a_3$ ,  $b_9 < b_6$ , was passiert bei  $b_{10} \geq a_9$  ?

Ist das sinnvoll??

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 nun 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?
```



- Material, Hinweise:
  - In einer Methode rufen Sie die gleichnamige Methode der Basisklasse ggf. auf mit "**super**" (anstelle des Methodennamens).
  - Tipp 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. Alternativ bilden Sie eine Testsuite mittels „test/unit“.
  - 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 **04-modnum.rb** sollte ausführbar sein und frei von Syntaxfehlern.