Blocs, Proc, Modules

Plan

Blocs et portée de variables

- Les variables ne sont pas explicitement définies en ruby
- Mais la portée est une portée de bloc de code (jusqu'à `}` ou `end`).
- Un nom de variable a une portée dans le bloc dans lequel elle est initialisée et dans ses sous blocs.

```
1 1.times { i = 1 }
2 puts i
3 #`<main>': undefined local variable or method `i' for main:Ob
```

```
1 i=0
2 1.times { i = 1 }
3 puts i     # 1
```

Bloc et arguments

On peut passer des arguments à un bloc

```
1 2.times { |i| puts i }
2 #=> 0
3 #=> 1
```

On peut passer un bloc qui ignore certains des arguments

```
1 2.times { puts 'lapin' }
2 #=> lapin
3 #=> lapin
```

On peut passer un bloc qui a trop d'arguments

```
1 2.times { |i, j, k| puts j.inspect }
```

Blocs : réponse à des questions de TP

- Peut être marqué par `begin`/`end` ou `{}` sans différence (`{}` si le bloc tient sur une ligne, `begin`/`end` sinon).
- Ne peut faire de `return` (ce n'est pas une méthode).
  - Dernière valeur calculée renvoyée.

Passage de bloc en argument *implicitement* et appel

Compteur de temps

```ruby
def compte()
  t = Time.now
  yield
  puts "Duree: #{Time.now - t}"
end

compte { sleep 2 }
```

Répéter

```ruby
def repeat(n)
  n.times { |i| yield(i) }
end
repeat(5) { |i|  puts i }
repeat(2) { puts 'Hello' }
```

Passage de bloc en argument *explicitement*

```ruby
def repeat(n, &code)
  n.times { |i| code.yield(i) }
end

repeat(5) do |i|
  puts i
end

repeat(2) { puts 'Hello' }
```

Transmettre un bloc de code

Compteur de temps avec répétitions

```ruby
def repeat(n)
  n.times { |i| yield(i) }
end

def compte(n, &code)
  t = Time.now
  repeat(n, &code)
  Time.now - t
end

v = []
puts compte(100000) { v << nil }
puts compte(100000) { v.push(nil) }
```

Mais de quel type est *code* ?

```
1 def foo(&code)
2   p code
3 end

4 foo { puts i }

5 # #<Proc:0x0000000266841 8>
```

Un bloc de code est passé sous forme de `Proc`.

Proc

Qu'est-ce qu'un objet Proc ?

1. Un bloc de code.
2. Un contexte capturé (Ensemble de variables locales connues au moment de la création du bloc).

```ruby
def gen_adder()
  accumulator = 0
  return Proc.new { |value| accumulator += value }
end

ac1 = gen_adder(); ac2 = gen_adder()
puts "#{ac1} #{ac2}"#=> #<Proc:0xd98> #<Proc:0xd48>

(1..3).each { |v| puts ac1.call(v) }  # 1 3 6
[10, 20].each { |v| puts ac2.call(v) } # 10 30
```

Appeler une Proc

Deux moyens équivalents :

```
1 a_proc = Proc.new { |a, *b| b.collect { |i| i * a } }
2 a_proc.call(9, 1, 2, 3)    #=> [9, 18, 27]
3 a_proc[9, 1, 2, 3]         #=> [9, 18, 27]
```

Construire une proc

Peut passer plus d'arguments à l'appel

```
1 pl = Proc.new { |x| x + 1 }
2 pl.call(1, 1) #  2
```

Vérification du nombre d'argument

```
1 pl = lambda { |x| x + 1 }
2 pl.call(1, 1) #  wrong number of arguments
3 pf = ->(x) { x + 1 }
4 pf.call(1, 1) #  wrong number of arguments
```

Symboles

- Un symbole est le nom de quelque-chose pour Ruby.
- Un symbole s'écrit `:nom`, où *nom* est le nom du symbole.
- Tous les symboles de même nom sont le *même* symbole :

```
1 s1 = :troll;                          s2 = :troll
2 puts "#{s1.object_id}                 #{s2.object_id}"
3 #=>          423848                        423848
4 ch1 = "troll";                        ch2 = "troll"
5 puts "#{ch1.object_id}                #{ch2.object_id}"
6 #=>        15950520                      15950500
```

- Ruby utilise des symboles pour identifier ses classes, méthodes, variables...
- `Symbol` est une classe (what else ?).

Utilisations des symboles

Une sorte de chaîne de caractère plus légère mais constante (enum sans valeur ?)

```
1 degre = { :duvel => 8.5,        :chimay_bleue => 9,
2           :chimay_rouge => 7, :rochefort_10 => 11.3 }
3 puts degre[:rochefort_10]
```

Construire un bloc (`to_proc`)

```
1 puts degre.keys.map(&:to_s).map(&:capitalize)
```

Introspection et réflexivité

- Parce que sert à désigner les méthodes, classes...
- Vu dans un cours à part.

Speed Test : String vs Symbol

```ruby
def process(&build)
  start = Time.now
  10000000.times { build.call }
  Time.now - start
end

puts process { 0 }          #=>    1.719175028
puts process { :troll  }    #=>    1.74779423
puts process { "troll" }    #=>    2.703107185
```

Ceci dit, si vous cherchez la vitesse pure, vous n'êtes pas sur le bon outil !

Le module comme espace de nommage

```ruby
module Clv
  class Player
    def initialize
      @state = Action::STOP
    end

    def play(track)
      @state = Action::START
       # ...
    end
  end

  module Action
    STOP = :stop
    START = :start
  end
end
```

Le module comme espace de nommage (suite)

En précisant le module à chaque fois

```
1 player = Clv::Player.new
2 player.play('El vals del obrero.ogg')
```

En utilisant `include`

```
1 include Clv
2 player = Player.new
3 player.play('El vals del obrero.ogg')
```

Le module comme espace de nommage (observation)

- Les modules peuvent s'imbriquer (module défini dans module).
- Un module peut être défini sur plusieurs fichiers
- `CRIEZ_PAS` est une constante. Majuscules partout.
  - Le casse des variables définit leur "type" !
- Notez l'usage des symboles dans l'exemple.
- L'abus de `include` augmente la probabilité de collision de nom.

Le module comme *mixin* (simple)

```ruby
module Bavard
  def to_ls
    "#{self.class}: #{self.to_s}"
  end
end

class Blob
  include Bavard
  def initialize(nom, pv)  @nom, @pv = nom, pv  end
  def to_s                 "#{@nom}(#{@pv})"    end
end

gaston = Blob.new("King", 128)
puts gaston.to_ls          #=> Blob: King(128)
```

Mixin simple

Le Module :

Définit des méthodes, des attributs, des constantes.

La classe :

`include` le Module.

L'instance :

Est alors dotées des méthodes, classes, attributs du module.

Modules prédéfinis (En mixin) : Comparable

```ruby
class Troll
  include Comparable
  attr_reader :nom, :mouches
  def initialize(nom, nombre_mouches)
    @nom, @mouches = nom, nombre_mouches
  end
  def <=>(other)
    return @mouches - other.mouches
  end
  def to_s
    "#{@nom} (#{mouches} mouches)"
  end
end
# Comparable fournit <, >, <=, >=, !=, ==
puts Troll.new("Arg", 23) < Troll.new("Zog", 3)

tribu = [];  nom = 'Grub'
10.times { tribu << Troll.new(nom, rand(100)); nom = nom.next }
puts tribu.sort
```

Modules prédéfinis (En mixin) : Enumerable

Prérequis de la classe qui inclus :

- Possède une méthode `each` qui `yield` en passant en argument chacun des éléments de la collection.
- Si on veut utiliser `min`, `max`, `sort`, il faut une méthode `<=>`.

Le module `Enumerable` fournit (entre autres !) :

- `all?` `any?`, `none`, `one?`, `count`
- `chunk` : découpe en *n* partitions suivant prédicat, `partition` : découpe en deux selon prédicat.
- `collect|map`, `collect_concat`
- `drop(n)` (début), `drop_while`
- `find` (premier élément qui...), `find_all`, `find_index`, `include?`
- `grep`, `select`, `group_by`, `reject`
- `reduce`
- `min`, `max`, `min`, `max_by`, `min_by`, `minmax`, `minmax_by`, `sort`, `sort_by`

Un peu plus de Mixin Issu de Brian Schröeder -immersive programming course-

```ruby
module Observable
  def register(event=nil, &callback)
    @observers ||= Hash.new
    @observers[event] ||= []
    @observers[event] << callback
    self
  end

  protected
  def signal_event(event = nil, *args)
    @observers ||= Hash.new
    @observers[event] ||= []
    @observers[event].each do |callback|
      callback.call(self, *args)
    end
  end
end
```

Suite de l'exemple de l'observable

```ruby
class Observed
  include Observable
  def foo = (a_foo)
    signal_event(:changed, @foo, a_foo)
    @foo = a_foo
  end
end

observed = Observed.new
observed.register(:changed) do |o, old, new|
  puts "#{old} -> #{new}"
end

observed.foo = 'Yukihiro'
observed.foo = 'Yukihiro Matsumoto'
observed.foo = 'Matz'
```

Observations sur l'observable

- Les méthodes du module peuvent accéder aux attributs de l'objet qui l'`include`.
- `a ||= b` réalise l'affectation si `a` n'est pas défini.
- Notez la syntaxe du setter de la propriété `foo` :

```
1 def foo=(a_foo)
2   signal_event(:changed, @foo, a_foo)
3   @foo = a_foo
4 end
```

- L'utilisateur du module doit signaler explicitement le changement. Possible d'automatiser en ajoutant automatiquement un setter par attribut (méta-programmation).

TD, ou TP, ou juste un peu de réflexion pour le plaisir...

Un arbre binaire

Écrire conteneur type *arbre binaire trié*. À l'insertion dans un sous-arbre, on compare l'élément à la racine : si celui-ci est plus petit, on insère à gauche, sinon on insère à droite.

- À quoi va ressembler un nœud de l'arbre ?
- Comment bénéficier de toutes les méthodes pour les conteneurs ?
- Quelles méthode doit donc implémenter l'arbre ? Comment l'écrire (algo, pas ruby...) ?
- Comment être sûr de gagner du temps sur les recherches ?

Qu'est ce que self ?

La valeur de *self* dépend du contexte.

```
1 class SuperLapin
2   def quel_self
3     p self
4   end
5   p self
6 end                       # => SuperLapin
7 sl = SuperLapin.new
8 sl.quel_self              # => #<SuperLapin:0x000000019d4878>
```

Et dans une définition de méthode ?

```
1 class SuperLapin
2   def self.la_classe()
3     puts "La classe!"
4   end
```

Retour sur les modules (exemple du *Caffeinated Crash Course*)

```ruby
module Encryptor
  def self.gen_key
    rand(255)
  end
  def self.encrypt str, key
    str.bytes.map{|byte| sprintf "%02x", (byte ^ key)}
      .join
  end
  def self.decrypt str, key
    str.scan(/../).map{|byte| (byte.to_i(16) ^ key).chr}
      .join
  end
end
key = Encryptor.gen_key                # => 171
c = Encryptor.encrypt "hello", key     # => "c3cec7c7c4"
puts Encryptor.decrypt c, key          # => "hello"
```

Encryptor, version *Mixin*

```ruby
module Encryptor
  def gen_key
    rand(255)
  end
  def encrypt (key)
    neo=self.bytes
       .map{|byte| sprintf "%02x", (byte ^ key)}.join
    self.replace(neo)
  end
  def decrypt (key)
    neo=self.scan(/../)
       .map{|byte| (byte.to_i(16) ^ key).chr}.join
    self.replace(neo)
  end
end
class String
  include Encryptor
end
```

Version *Mixin* : usages

Étendons une classe

```
1 secret = "Je suis un troll"
2 key = secret.gen_key
3 secret.encrypt(key) # =>  "5a7530636579633065 7e3064627f7c7c"
4 secret.decrypt(key) # => "Je suis un troll"
```

Étendons un seul objet ! `Object::extend`

```
1 secret = "Je suis un troll"
2 secret.extend(Encryptor)
3 secret.encrypt(42)
```

Résumé des leçons du Mixin

- Un module peut contenir des méthodes.
- Quand il est *inclus* (`include`) dans une classe, ces méthodes deviennent des méthodes d'instance.
- Il peut donc créer des attributs pour l'instance.
- On accède, dans les méthodes du module, à l'objet qui l'inclut par `self`.
- Un module peut étendre un objet via `extend`.

Comment ajouter une méthode de classe via un Module ?

- La solution est sur ce transparent
- Note à moi même : peut-être introduire ça après la réflexivité.

Ajouter une méthode de classe via un module - V1

Version exclusive

```
1 module PasInspire
2   def une_methode
3   end
4 end

5 class A
6   extend PasInspire
7 end
```

- C'est la classe qui étend le module !
- Pb : pas possible de faire méthode de classe et d'instance dans le même module ?

Ajouter une méthode de classe via un module - V2

Version à la main

- *Callback* `included` : appelé quand le module est inclus.

```ruby
module Modulaire
  def self.included(base)
    base.extend(ClassMethods)
  end
  module ClassMethods
    def une_methode
    end
    def une_autre_methode
    end
  end
end
```