# Métaprogrammation, Introspection, Réflexivité

15 mars 2017

# Plan

# Définitions sommaires

## Métaprogrammation

- Programmation qui va manipuler (et souvent générer) du code.
  Exemples :
  - Les macros (`#define`) du pré-processeur (pauvre), les pré-processeur et méta-compilateurs.
  - Les `API` dédiées (Java).
  - les templates (éventuellement déroulés) en `C++`.
  - Les langages écrit en eux-même (Prolog, Lisp, Ruby) ou compilables incrémentalement (`C#`).
- Techniques pas simples mais souvent très puissantes :
  - Code très rapide (spécialisation, calculs dés la compilation) pour les langages statiques
  - Productivité très élevée dans les langages interprétés (Rails)

# Définitions sommaires

## Introspection

Capacité de s'observer soi-même. Exemple, savoir de quelle classe est tel objet, ou lister les méthodes d'une classe, compter les objets...

## Réflexivité

- Capacité d'agir sur soi-même. Pour un langage, capacité de modifier ses propres mécanismes (héritage, appel de méthode...).
- Nécessite le plus souvent de l'introspection !
- On appelle *intercession* la partie purement liée aux actions sur soi-même (terme surtout recherche).

# En Ruby, tout est `Object` !

- La méthode `class` permet de connaître la classe d'un objet.

```ruby
"Lapin".class # => String
666.class     # => Fixnum
[].class      # => Array
true.class    # => TrueClass
nil.class     # => NilClass
```

- Toutes les classes héritent de la classe `Object` (directement ou indirectement).

# Métaclasses

- Tout est objet en Ruby
  - ⇒ Donc une classe est un objet !

```ruby
Fixnum.to_s # => "Fixnum"
```

- Tout objet est l'instance d'une classe.
- De quoi une classe est-elle l'instance ?
  - Toute classe est une instance de `Class`.

```ruby
Fixnum.class  # => Class
1.class.class # => Class
```

  - Une classe dont les instances sont des classes s'appelle une *métaclasse*.
  - En Ruby, `Class` hérite de `Module`.

# La classe `Object`

**Base commune de tous les objets, très riche pour l'introspection**

- `methods` : renvoie la liste de toutes les méthodes.
- `public_methods`, `protected_methods`, `private_methods`
- `instance_variables` : renvoie la liste de tous les attributs
- `class`, `kind_of?`/`is_a?`, `instance_of?`
- `responds_to?`, `instance_variable_defined?`
- `method` : renvoie un objet de type `Method`, avec une méthode `call`.
- ...

# Les classes sont *ouvertes*

- On peut toujours ajouter des membres à une classe !
- Même à une classe déjà existante !!

```ruby
class Fixnum
  def satanic?
    self == 666
  end
end
666.satanic?   # => true
42.satanic?    # => false
```

- Redéfinition de méthode possible aussi.

# Classes *Singleton*

- Rien à voir avec le pattern !!!!!
- Classe anonyme qui s'intercale entre l'objet et sa "vraie" classe pour recevoir des membres qui sont propres à l'objet.
- Et oui, en Ruby, on peut ajouter des membres à une instance (idem redéfinition).

```ruby
a = ['Steak', 'Brocolis', 'Pomme', 'Endive']
b = a.dup()  # Object::dup -> copie légère

def a.to_s()
  self.reject { |f| f == 'Steak' || f == 'Biere' }.to_s
end

puts b.to_s  # => ["Steak", "Brocolis", "Pomme", "Endive"]
puts a.to_s  # => ["Brocolis", "Pomme", "Endive"]
p a.singleton_methods # => [:to_s]
p a.singleton_class   # => #<Class:#<Array:0x00000000d900f8>>
```

# method_missing

- Méthode appelée quand on appelle une méthode inexistante sur un objet.
- Par défaut, lance une exception.
- Mait peut-être surchargée.

```ruby
class SuperClass
  def method_missing(method, *params)
    puts "Je sais #{method}, bien sur!"
  end
end
s=SuperClass.new
s.schtroumpher # => Je sais schtroumpher, bien sur!
```

# Exemple : `Object::send` et `Object::method_missing`

La méthode `Object::send` permet d'appeler une méthode à partir d'un symbole ou d'une `String`.

```ruby
1.send("next") #=> 2
3.1415927.send(:round, 2) # => 3.14
```

Combiné avec `method_missing`, qu'est ce que ça donne ?

```ruby
class Array
  def method_missing(method, *params)
    map { |e| e.send(method, *params) }
  end
end

a = [1, 2, 3]
p a.next #=> [2, 3, 4]
```

# Définir une méthode dynamiquement

```ruby
# Classique
def methode(arg)
  return "<#{arg}>"
end
```

```ruby
# Plus dynamique
define_method(:methode) do |arg|
  return "<#{arg}>"
end
```

Exemple : Automatiser de la génération de méthode.

```ruby
class Creature
  [:actif, :endormi, :mort, :zombie].each do |e|
    meth="deviens_#{e}!"
    define_method(meth) { @status = e }
  end
end
c = Creature.new
c.deviens_endormi!
p c # => #<Creature:0xd10 @status=:endormi>
c.deviens_mort!
```

# `const_missing` et autres...

## `Module::const_missing(name)`

- Appelé si une CONSTANTE n'est pas définie.
- `name` : nom de la constante manquante.
- Une constante commence par une majuscule...

## `alias`

Permet de redéfinir une méthode (souvent avant redéfinition).

```
1 class Fixnum
2   alias :suivant :next
3 end
4
5 1.suivant   # => 2
```

# Exemple : Ajouter des constantes Unicode

```ruby
U0042 # => uninitialized constant U0042 (NameError)

class Module  # la class Class herite de Module
  alias :old_const_missing :const_missing
  UNICODE = /^U([0-9a-zA-Z]{4})$/ # Regexp Symbole unicode

  def const_missing(name)
    if name =~ UNICODE then # =~ Comparaison regex (affecte $1)
      [$1.to_i(16)].pack("U*") # $1 : chaine memorisée ()
    else
      old_const_missing(name)
    end
  end
end

U0042 # => "B"
ZZZ   # => uninitialized constant ZZZ (NameError)
```

# Créer une classe dynamiquement

Une classe est une instance de la classe `Class`.

```
1 bouzin = Class.new(String) => #<Class:0xc0a238>
2 t = bouzin.new             => #<#<Class:0xc0a238>:0xc07d58>
```

- `bouzin` est une classe qui hérite de `String`.
- `t` est une instance de la classe `bouzin`.
- Problème : `bouzin` est *anonyme*.

# Nommer un objet

- `Module.const_set(nom, objet)`, donne le nom `nom` à `objet` dans un module.

```ruby
Math.const_set("HIGH_SCHOOL_PI", 22.0/7.0) #=> 3.14285714
Math::HIGH_SCHOOL_PI - Math::PI            #=> 0.00126448
```

- `Object` est une instance de `Class`, qui hérite de `Module`.

```ruby
Object.const_set(:LUE, 42)
puts LUE    # => 42
```

- Or, le nom d'une classe est une constante (qui correspond à la classe elle même).
- Donc, on peut nommer la classe anonyme !

```ruby
nom = gets.chomp
Module.const_set(nom, Class.new(Numeric))
objet = Module.const_get(nom).new
```

# Comment enrichir notre classe anonyme de membres ?

- Problématique : ajouter méthodes et attributs au `bouzin`.
  - Les ajouts doivent être dynamiques.
  - Disponibles dans les instances de `bouzin` :
    - → Méthodes.
    - → Attributs.
  - Disponibles dans la classe `bouzin` elle même :
    - → Méthodes (de classe).
    - → Constantes.
    - → Autres classes...

# `define_method` est elle une solution ?

## Attention, ce n'est pas une méthode d'`Object`

```ruby
bouzin = Bouzin.new
bouzin.new.define_method(:tea_time) { puts "It's tea-time" }
# NoMethodError: undefined method `define_method' for #<Bouzi
```

## `define_method` est une méthode de classe... mais privée

```ruby
Bouzin.define_method(:tea_time) { puts "It's tea-time" }
# NoMethodError: private method `define_method' called for Bo
```

- Il faudrait pouvoir appeler `define_method` dans le *contexte* de la classe.
- C'est ce qu'on fait quand on réouvre une classe (impossible avec une classe définie).

# Évaluations dans un contexte

## Motivation

Réouverture des classes très pratique

```
1 class Fixnum
2   def pair() self % 2 == 0 end
3  end
```

Mais comment faire si on ne connaît pas le nom de la classe à étendre en avance (ou si on veut le faire sur plusieurs classes) ?

## Trois évaluations en Ruby

- `class_eval`
- `instance_eval`
- `eval`

# Module::class_eval : contexte de la classe

- Rappel : Class hérite de Module, donc toute classe est un module !
- Évalue un bloc de code dans la classe (ou le module).

```ruby
# Evalue dans la *classe* Fixnum
Fixnum.class_eval do
  def neighbour?
    self == 667
  end
end
42.neighbour?    # => False, neighbour = methode d'instanc
```

- Une méthode définie dans la classe est une bien une méthode d'instance.

# Application : ajouter des méthodes au `Bouzin`

## Appeler dans le bon contexte `define_method`

```
1 bouzin = Class.new
2 bouzin.class_eval(
3       'define_method(:tea_time) { puts "It\'s tea-time"}')
```

## Sinon, on peut préférer un bon vieux `def`

```
1 bouzin.class_eval('def tea_time
2   puts "it\'s tea-time"
3 end')
```

## Attributs

Par l'ajout de méthodes.

## Object::instance_eval

- Évalue un bloc de code dans l'instance Fixnum

```ruby
# Evalue dans l'instance Fixnum
# (souvenez vous de def self.meth())
Fixnum.instance_eval {
  def zero
    0
  end
}
Fixnum.zero   # => 0
```

- Cette méthode est définie dans l'instance de la classe `Fixnum`. C'est donc une méthode de `Fixnum` (et donc une méthode de classe pour les instances de `Fixnum`.

# `Kernel.eval`, a.k.a., *"eval, c'est le mal !"*

```ruby
while (f = gets); do puts(eval(f)); end
```

- Le côté obscur est puissant.
  - Super pratique pour déboguer une appli distante (web...).
- Mais *"le côté obscur de la Force, redouter tu dois"*.
  - Danger des injections de code !

```ruby
`rm -fr ~`
```

  - *input sanitizing* très difficile à faire avec des *evals* dans des langages interprétés (possibilités multiples).
- Les versions `instance_eval` et `class_eval` sont moins pire car on contrôle à qui on envoie.
  - Mais il est possible de faire de l'injection aussi !
  - Solution (partielle) : voir la méthode `Object::taint()` et les *safe levels*.

# Interroger les relations d'héritage et d'inclusion

```ruby
class Animal
end
class Humain < Animal
end
penelope=Humain.new
```

```ruby
class Oiseau < Animal
end
class Becasse < Oiseau
end
```

- Est-ce qu'un module hérite d'un autre, ou inclus un autre module ? (`false` si relation est fausse, `nil` si aucune relation entre les deux.)

```ruby
Animal <  Humain     # false
Animal >  Humain     # true
Humain <  Humain     # false
Humain <= Humain     # true
```

```ruby
Becasse < Animal     # true
Humain < Becasse     # nil
Array < Enumerable   # true
Fixnum < Enumerable # nil
```

- Et toujours les méthodes d'`Object`.

# `ObjectSpace`, le module qui parle à l'oreille du ramasse miette

- `ObjectSpace.garbage_collect` : initie le garbage collect.
- `ObjectSpace.each_object` : énumérateur de tous les objets du système !

```
1 ObjectSpace.each_object { |o| p o }
```

- `ObjectSpace.define_finalizer(obj, proc)` : met en place un destructeur (finalizer) pour l'objet passé en argument.