Métaprogrammation en Ruby
OU
Quand Ruby est «vraiment» dynamique

Guy Tremblay
Professeur

Département d'informatique
UQAM
http://www.labunix.uqam.ca/~tremblay_gu

20 novembre 2018

Contenu

1 Qu’est-ce que la «métaprogrammation» ?

2 Typage dynamique en Ruby (*duck typing*)

3 Évaluation dynamique de code

4 Réouverture de classes

5 Envoi et réception dynamique de messages

6 Exemples

7 Autres exemples : Ruby on Rails

8 Conclusion

1. Qu’est-ce que la «métaprogrammation» ?

Quelques *tips* de Hunt & Thomas

Tirés de «*The Pragmatic Programmer—From Journeyman to Master*»

11. DRY—Don't Repeat Yourself
    Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

29. Write Code That Writes Code
    Code generators increase your productivity and help avoid duplication.

Qu’est-ce que la métaprogrammation ?

La métaprogrammation définit un ensemble d’outils et de techniques de programmation pour éliminer la redondance — et autres «code smells» — dans du code.

Source*: «Metaprogramming Ruby», G. Vanderburg, 2005*

Qu'est-ce que la métaprogrammation ?

Metaprogramming is the act of writing code that operates on code rather than on data. This involves inspecting and modifying a program as it runs using constructs exposed by the language.

Source: *https://rubymonk.com/learning/books/2-metaprogramming-ruby/chapters/32-introduction-to-metaprogramming/lessons/75-being-meta*

Qu'est-ce que la métaprogrammation ?

Metaprogramming is the writing of computer programs with the ability to treat programs as their data. It means that a program could be designed to read, generate, analyse or transform other programs, and even modify itself while running.

Source: *https://en.wikipedia.org/wiki/Metaprogramming*

Deux principales approches à la métaprogrammation

Réflexivité

Le langage fournit une API qui permet, durant l'exécution, d'examiner et de modifier le programme — et non pas juste les données du programme.

Génération et exécution dynamique de code

Le langage fournit une API qui permet, à l'exécution, de générer du nouveau code (e.g., définition de méthodes) puis de l'exécuter.

Deux principales approches à la métaprogrammation

Réflexivité

Le langage fournit une API qui permet, durant l'exécution, d'examiner et de modifier le programme — et non pas juste les données du programme.

Par ex., *reflection* en Java

Génération et exécution dynamique de code

Le langage fournit une API qui permet, à l'exécution, de générer du nouveau code (e.g., définition de méthodes) puis de l'exécuter.

Programmation générative

Mais que veut-on dire quand on dit que Ruby est un langage dynamique ?

Ce qui suit traite de métaprogrammation en Ruby, un langage dynamique

Mais que veut-on dire quand on dit que Ruby est un langage dynamique ?

Typage dynamique

- Pas d'analyse des types à la compilation
- Le type est «validé» à l'exécution — *duck typing*

Ce qui suit traite de métaprogrammation en Ruby, un langage dynamique

Mais que veut-on dire quand on dit que Ruby est un langage dynamique ?

Typage dynamique

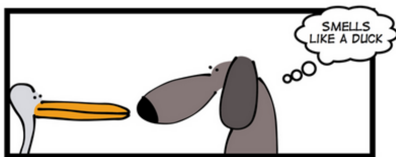
- Pas d’analyse des types à la compilation
- Le type est «validé» à l’exécution — *duck typing*

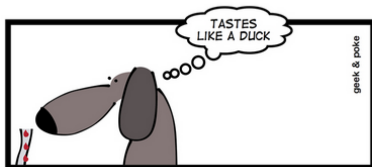
Création, extension et modification des objets et classes à l’exécution

- Évaluation dynamique de code
- Réouverture de classes
- Envoi et réception dynamique de messages

2. Typage dynamique en Ruby
(*duck typing*)

Ruby permet le typage polymorphique à l'aide du «*duck typing*»



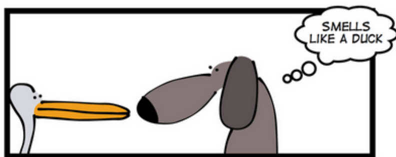


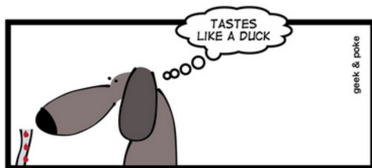
Qu'est-ce que le «*duck typing*» ?

Le type (ou la classe) d'un objet n'a pas d'importance.

Ce qui compte, ce sont les messages auxquels peut répondre un objet.

Ruby permet le typage polymorphique à l'aide du «*duck typing*»





Qu'est-ce que le «*duck typing*» ?

Le type (ou la classe) d'un objet n'a pas d'importance.

Ce qui compte, ce sont les messages auxquels peut répondre un objet.

If it walks like a duck, and swims like a duck and quacks like a duck, then it must be a duck !

Ruby permet le typage polymorphique à l'aide du «*duck typing*»

Définition de méthode

```
def foo( a, b, c )
  a + b * c
end
```

Exemple d'utilisation (a)

```
>> -10.respond_to? :+
=> true
>> +20.respond_to? :*
=> true

>> foo( -10, +20, 5 )
=> 90
```

Ruby permet le typage polymorphique à l'aide du «*duck typing*»

Définition de méthode

```
def foo( a, b, c )
  a + b * c
end
```

Exemple d'utilisation (b)

```
>> "-10".respond_to? :+
=> true
>> "+20".respond_to? :*
=> true

>> foo( "-10", "+20", 5 )
=> "-10+20+20+20+20+20"
```

Ruby permet le typage polymorphique à l'aide du «*duck typing*»

Le *duck typing* permet une forme de polymorphisme

Définition de la méthode `MiniSed.delete`

```
# Effectue des suppressions de lignes du flux d'entree et emet
# lignes non supprimees sur le flux de sortie.
#
# @param [#each] flux_in le flux a traiter
# @param [#<<] flux_out le flux sur lequel emettre les resulta
# @param [String] motif la chaine qui decrit le motif a cherch
#
# @return [void]
#
def self.delete( flux_in, flux_out, motif )
  flux_in.each do |ligne|
    flux_out << ligne unless /#{motif}/ =~ ligne
  end
end
```

Ruby permet le typage polymorphique à l'aide du «*duck typing*»

Le *duck typing* permet une forme de polymorphisme

Utilisation de `MiniSed.delete` — avec des `Array`

```
it "peut traiter des Array" do
  fi = ["abc\n", "def\n", "aB\n", "abab\n"]
  fo = []

  MiniSed.delete( fi, fo, "ab" )

  fo.must_equal ["def\n", "aB\n"]
end
```

Ruby permet le typage polymorphique à l’aide du «*duck typing*»

Le *duck typing* permet une forme de polymorphisme

Utilisation de `MiniSed.delete` — avec des fichiers

```
it "peut traiter des fichiers" do
  fo = File.open('bar.txt', 'w')
  avec_fichier 'foo.txt', ["abc", "def", "aB", "abab"] do
    MiniSed.delete( File.open('foo.txt'), fo, "ab" )
  end
  fo.close

  contenu_fichier('bar.txt').must_equal ["def", "aB"]

  FileUtils.rm_f 'bar.txt'
end
```

Le *duck typing* évite la prolifération des `interface`s comme en Java

Java

```
interface Runnable {
 void run();
}

class Thread {
  Thread(Runnable target)
  // Allocates a new Thread object.

  Thread(Runnable target, String name)
  // Allocates a new Thread object.
}
```

Le *duck typing* évite la prolifération des `interface`s comme en Java

Package `java.util.function` $\approx$ 50 interfaces

```
BiConsumer<T,U>
Represents an operation that accepts two input arguments and returns no result.

BiFunction<T,U,R>
Represents a function that accepts two arguments and produces a result.

BinaryOperator<T>
Represents an operation upon two operands of the same type, producing a result of the same t

BiPredicate<T,U>
Represents a predicate (boolean-valued function) of two arguments.

BooleanSupplier
Represents a supplier of boolean-valued results.
.
.
.

ToLongFunction<T>
Represents a function that produces a long-valued result.

UnaryOperator<T>
Represents an operation on a single operand that produces a result of the same type as its
```

3. Évaluation dynamique de code

Différentes méthodes permettent d'évaluer du code de façon dynamique, par ex., pour définir des méthodes

- `instance_eval`
- `class_eval`
- `define_method`

Ce que fait `instance_eval` : Évalue un segment de code dans le contexte du récepteur du message

Description de `instance_eval`

```
obj.instance_eval { |obj| block } -> obj
```

Evaluates [...] the given `block` within the context of the receiver `(obj)`. In order to set the context, the variable `self` is set to `obj` while the code is executing, giving the code access to `obj`'s instance variables and private methods.

Ce que fait `class_eval` : Évalue un segment de code dans le contexte d'une classe

Description de `class_eval`

```
C.class_eval(string [, filename [, lineno]]) -> obj
```

Evaluates the `string` (or block) in the context of [class `C`], except that when a block is given, constant/class variable lookup is not affected. This can be used to add methods to a class. `class_eval` returns the result of evaluating its argument. The optional `filename` and `lineno` parameters set the text for error messages.

Ce que fait `define_method` : Définit une méthode dans le contexte d'une classe

Description de `define_method`

```
define_method(symbol) { block } -> symbol
```

Defines an instance method in the receiver. The method parameter can be a `Proc`*, a* `Method` *or an* `UnboundMethod` `object`*. If a block is specified, it is used as the method body.*

Ce que fait `instance_eval` : Évalue un segment de code dans le contexte du récepteur du message

Un petit exemple où on évalue un bloc

Définition d'une classe A

```
class A
  def initialize( x )
    @x = x
  end

  def val
    @x
  end
  private :val

  def plus( y )
    @x + y
  end
end
```

Quelques appels

```
>> a = A.new( 10 )
=> #<A:0x000000016c9c00 @x=10>

>> a.instance_eval { @x }
=> 10

>> a.instance_eval { plus 2 }
=> 12

>> a.val
=> NoMethodError: private
    method 'val' called [...]

>> a.instance_eval { val }
=> 10
```

Ce que fait `instance_eval` : Évalue un segment de code dans le contexte du récepteur du message ⋆

Un petit exemple où on évalue une `String` — mais un bloc est préférable

Définition d'une classe `A`

```ruby
class A
  def initialize( x )
    @x = x
  end

  def val
    @x
  end
  private :val

  def plus( y )
    @x + y
  end
end
```

Quelques appels

```ruby
>> a = A.new( 10 )
=> #<A:0x000000016c9c00 @x=10>

>> a.instance_eval "@x"
=> 10

>> a.instance_eval "plus 2"
=> 12

>> x = 1
=> 1

>> a.instance_eval "plus #{2 * x}"
=> 12
```

Ce que fait `class_eval` : Évalue un segment de code dans le contexte d'une classe

Un petit exemple

Définition dynamique de méthodes

```
>> class Foo; end
=> nil

>> Foo.class_eval { def bye; "Bye de l'instance" end }
=> :bye

>> Foo.new.bye
=> "Bye de l'instance"

>> ms = Foo.methods(false)
=> [:bye]

>> Foo.class_eval "def self.#{ms[0]}; 'Bye de la classe' end"
=> :bye

>> Foo.bye
=> "Bye de la classe"
```

4. Réouverture de classes

Une classe existante peut être étendue/modifiée en la «réouvrant»

Réouverture d’une classe pour lui ajouter une méthode

```
>> class Foo
     def foo; "foo" end
   end
=> :foo

>> (f = Foo.new).foo
=> "foo"

>> f.bar
NoMethodError: undefined method 'bar' for #<Foo:0x000000016459f0>...

>> class Foo
     def bar; "bar" end
   end
=> :bar

>> f.bar
=> "bar"
```

Un objet peut aussi être étendu dynamiquement

Ajout d'une méthode spécifique à un objet

```
>> f.buzz
NoMethodError: undefined method 'buzz' for
#<Foo:0x00000016459f0>...

>> def f.buzz; "buzz" end
=> :buzz

>> f.buzz
"buzz"

>> g = Foo.new
=> #<Foo:0x00000002b57b90>

>> g.buzz
NoMethodError: undefined method 'buzz' for
#<Foo:0x00000016459f0>...
```

On peut réouvrir n’importe quelle classe (sic)

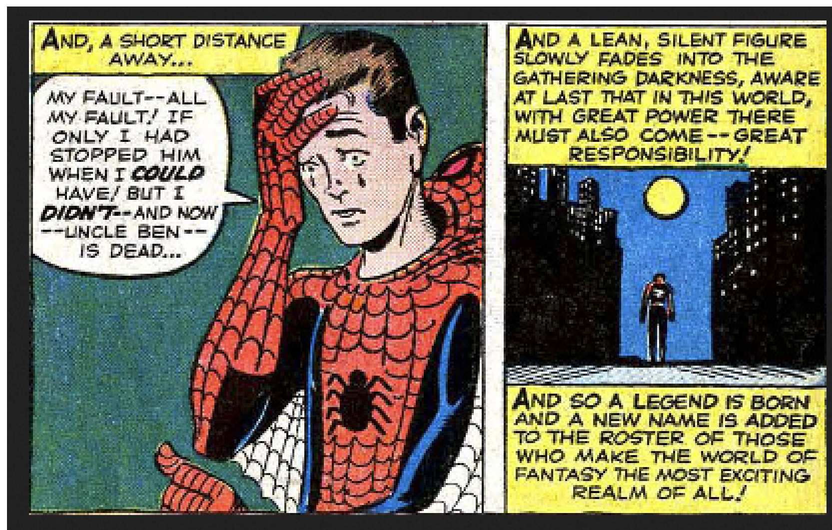
Réouverture d’une classe de la bibliothèque

```
class Fixnum
  def foo
    "foo" * self
  end
end

>> 4.foo
=> "foofoofoofoo"
```

On peut réouvrir n'importe quelle classe (sic[2])

« With great power there must also come great responsibility ! »



Source: `http://quoteinvestigator.com/wp-content/uploads/2015/07/spider400.jpg`

On peut réouvrir n'importe quelle classe (sic[2])

Réouverture d'une classe de la bibliothèque

```
class Fixnum
  alias :vrai_plus :+

  def +( other )
    return 0 if other == 0

    self.send :vrai_plus, other
  end
end

>> 3 + 5
=> 8

>> 28 + 0
=> 0
```

5. Envoi et réception dynamique de messages

Le nom de la méthode appelée peut être spécifié dynamiquement

Le nom de la méthode peut être un `Symbol` ou une `String`

Un appel de méthode est un envoi de message

```
>> 2 + 3
=> 5

>> 2.+( 3 )
=> 5

>> 2.send( :+, 3 )
=> 5

>> a = :*
=> :*

>> 2.send a, 3
=> 6

>> 2.send "X*Z"[1], 3
=> 6
```

L'utilisation de `send` permet d'appeler une méthode privée

```
class Foo
  def foo; "foo" end
  def bar; "bar" end
  private :bar
end

>> (f = Foo.new).foo
=> "foo"

>> f.bar
NoMethodError: private method 'bar' called for #<Foo:0x0000000

f.send :bar
=> "bar"
```

Remarque : Peut être utile lorsqu'on veut tester (dans des tests unitaires) une méthode qui est supposée être privée ⇒ on utilise `send` dans les tests !

La méthode `method_missing` permet «d'attraper» les appels à des méthodes non définies

```
class A
  def m1; puts "In m1: #{self.class}" end
end

class B < A
  def method_missing( sym, *args, &block )
    puts "In method_missing( #{sym} ): #{self.class}"
  end
end

>> A.new.m1
In m1: A
=> nil

>> B.new.m1
In m1: B
=> nil
```

La méthode `method_missing` permet «d'attraper» les appels à des méthodes non définies

```
class A
  def m1; puts "In m1: #{self.class}" end
end

class B < A
  def method_missing( sym, *args, &block )
    puts "In method_missing( #{sym} ): #{self.class}"
  end
end

>> A.new.m2
   missing.rb:15: undefined method 'm2'
   for #<A:0xb7fd2728> (NoMethodError)
```

La méthode `method_missing` permet «d'attraper» les appels à des méthodes non définies

```
class A
  def m1; puts "In m1: #{self.class}" end
end

class B < A
  def method_missing( sym, *args, &block )
    puts "In method_missing( #{sym} ): #{self.class}"
  end
end

>> A.new.m2
   missing.rb:15: undefined method 'm2'
   for #<A:0xb7fd2728> (NoMethodError)

>> B.new.m2
In method_missing( m2 ): B
=> nil
```

6. Exemples

Exemple # 1

Les méthodes `attr_reader` et `attr_writer`

Exemples d'utilisation

```
class Foo
  attr_reader :val
  attr_writer :val

  def initialize( v )
    @val = v
  end
end

>> (f = Foo.new(10)).val
=> 10

>> f.val = "xxxx"
=> "xxxx"

>> f.val
=> "xxxx"
```

Question

Comment peut-on mettre en œuvre les méthodes `attr_reader` et `attr_writer` ?

Les méthodes `attr_reader` et `attr_writer`

Mise en œuvre avec `class_eval` et des `String`s — approche non recommandée

```
class Class
  def attr_reader( attr )
    class_eval "
      def #{attr}
        @#{attr}
      end
    "
  end

  def attr_writer( attr )
    class_eval "
      def #{attr}=( v )
        @#{attr} = v
      end
    "
  end
end
```

Les méthodes `attr_reader` et `attr_writer`

Mise en œuvre avec `define_method` et `instance_eval` — mieux, mais pas idéal

```
class Class
  def attr_reader( attr )
    define_method attr do
      instance_eval "@#{attr}"
    end
  end

  def attr_writer( attr )
    define_method "#{attr}=" do |v|
      instance_eval "@#{attr} = #{v}"
    end
  end
end
```

Les méthodes `attr_reader` et `attr_writer`

Mise en œuvre avec `define_method` et `instance_variable_{get,set}` — **encore mieux**

```
class Class
  def attr_reader( attr )
    define_method attr do
      instance_variable_get "@#{attr}"
    end
  end

  def attr_writer( attr )
    define_method "#{attr}=" do |v|
      instance_variable_set "@#{attr}", v
    end
  end
end
```

Exemple # 2

Les spécifications `MiniTest` avec des noms sous forme de chaines de caractères

Exemple d'utilisation

```
# Style a la RSpec.
it "retourne solde nul si on retire tout" do
  @c.retirer( @c.solde )

  assert_equal 0, @c.solde
end
```

Les spécifications MiniTest avec des noms sous forme de chaines de caractères

Exemple d'utilisation

```
# Style a la RSpec.
it "retourne solde nul si on retire tout" do
  @c.retirer( @c.solde )

  assert_equal 0, @c.solde
end
```

```
# Style a la JUnit.
def test_retourne_solde_nul_si_on_retire_tout
  @c.retirer( @c.solde )

  assert_equal 0, @c.solde
end
```

Question

Comment peut-on mettre en œuvre la méthode `it` ?

Les spécifications MiniTest avec des noms sous forme de chaines de caractères

Mise en œuvre (simple)

Méthode symbole_pour

```
def symbole_pour( prefixe, ident )
  (prefixe + ident.gsub(/\s/, "_")).to_sym
end


>> symbole_pour( "foo_", "1 2 3 4" )
=> :foo_1_2_3_4

>> symbole_pour( "test_", "abc def" )
=> :test_abc_def
```

Les spécifications MiniTest avec des noms sous forme de chaines de caractères

Mise en œuvre (simple)

Méthode it

```
def it( ident, &bloc )
  define_method symbole_pour("test_", ident) do
    instance_eval &bloc
  end
end
```

Les spécifications `MiniTest` avec des noms sous forme de chaines de caractères

Mise en œuvre (simple)

```
it "retourne solde nul si on retire tout" do
  @c.retirer( @c.solde )

  assert_equal 0, @c.solde
end
```

Les spécifications `MiniTest` avec des noms sous forme de chaines de caractères

Mise en œuvre (simple)

```
it "retourne solde nul si on retire tout" do
  @c.retirer( @c.solde )

  assert_equal 0, @c.solde
end
```

⇒

```
def test_retourne_solde_nul_si_on_retire_tout do
  @c.retirer( @c.solde )

  assert_equal 0, @c.solde
end
```

Exemple # 3

Une méthode pour tracer l’exécution de méthodes

Exemple d’utilisation

```
>> Array.extend TracerMethode
=> Array

>> [10, 20, 30].first
=> 10

>> Array.tracer_methode :first
=> :first

>> [10, 20, 30].first
-- Appel a   'first'
++ Retour de 'first' => 10
=> 10

>> [10, 20, 30].last
=> 30
```

Une méthode pour tracer l'exécution de méthodes

Mise en œuvre : Adaptée de «*Effective Ruby*», P.J. Jones, Addison-Wesley, 2015.

```
module TracerMethode
  def tracer_methode( methode )
    # On trouve un nouveau nom pour la vraie methode.
    methode_originale = "#{methode}_sans_logging".to_sym

    # On s'assure que le nom est unique.
    if instance_methods.include?( methode_originale )
      raise NameError, "#{methode_originale} n'est pas distinct
    end

    # On cree un alias pour acceder a la "vraie" methode.
    alias_method methode_originale, methode

    # On remplace la methode par le code de tracage.
    define_method( methode ) do |*args, &block|
      STDOUT.puts "-- Appel a  '#{methode}'"
      resultat = send methode_originale, *args, &block
      STDOUT.puts "++ Retour de '#{methode}' => #{resultat}"
      resultat
    end
  end
```

Ce que fait `alias_method` : Définit un alias — un synonyme — pour une méthode existante

Description de `alias_method`

```
alias_method(new_name, old_name) -> self
```

Makes `new_name` *a new copy of the method* `old_name`*. This can be used to retain access to methods that are overridden.*

Exemple # 4

Une méthode pour trier selon différents champs

Exemple d'utilisation

```
class Personne
  attr_reader :nom, :age

  def initialize(nom, age); @nom = nom; @age = age; end
  def inspect; "#{nom} (#{age})" end
end

>> a = [Personne.new("Abc", 20),
        Personne.new("Xyz", 30),
        Personne.new("Def", 10)]
=> [Abc (20), Xyz (30), Def (10)]

>> a.sort_by :nom
=> [Abc (20), Def (10), Xyz (30)]

>> a.sort_by :age
=> [Def (10), Abc (20), Xyz (30)]
```

Question

Comment peut-on mettre en œuvre la méthode `sort_by` pour des `Enumerable`s ?

Une méthode pour trier selon différents champs

Mise en œuvre

```
module Enumerable
  def sort_by( id )
    sort { |x, y| x.send(id) <=> y.send(id) }
  end
end
```

Exemple # 5

Des méthodes pour trier selon différents champs (bis)

Exemple d'utilisation

```
class Personne
  attr_reader :nom, :age

  def initialize(nom, age); ...; end
  def inspect; ...; end
end

>> a = [Personne.new("Abc", 20),
        Personne.new("Xyz", 30),
        Personne.new("Def", 10)]
=> [Abc (20), Xyz (30), Def (10)]

>> a.sort_by_nom
=> [Abc (20), Def (10), Xyz (30)]

>> a.sort_by_age
=> [Def (10), Abc (20), Xyz (30)]
```

Question

Comment peut-on mettre en œuvre les méthodes `sort_by_nom` et `sort_by_age` pour des `Enumerable`s ?

Des méthodes pour trier selon différents champs (bis)

Mise en œuvre

```
module Enumerable
  def sort_by_age
    sort { |x, y| x.send(:age) <=> y.send(:age) }
  end

  def sort_by_nom
    sort { |x, y| x.send(:nom) <=> y.send(:nom) }
  end
end
```

Des méthodes pour trier selon différents champs (bis)

Mise en œuvre

```
module Enumerable
  def sort_by_age
    sort { |x, y| x.send(:age) <=> y.send(:age) }
  end

  def sort_by_nom
    sort { |x, y| x.send(:nom) <=> y.send(:nom) }
  end
end
```

Mais… quel est le problème avec cette mise en œuvre ?

Des méthodes pour trier selon différents champs (bis)

Mise en œuvre ☹

```ruby
module Enumerable
  def sort_by_age
    sort { |x, y| x.send(:age) <=> y.send(:age) }
  end

  def sort_by_nom
    sort { |x, y| x.send(:nom) <=> y.send(:nom) }
  end
end
```

Mais… quel est le problème avec cette mise en œuvre ?

Pas DRY ☹

Pas extensible ☹

```ruby
class Personne
  attr_accessor :adresse
end
...
a.sort_by_adresse
```

Des méthodes pour trier selon différents champs (bis)

Mise en œuvre DRY ☺

```
module Enumerable
  def method_missing( symb, *args, &block )
    est_sort_by_X = /^sort_by_(.+)$/ =~ symb
    raise NoMethodError, "Methode non-definie '#{symb}'\
                pour #{self}" unless est_sort_by_X

    # Le nom de la methode est "sort_by_<champ_X>"
    champ_X = $1
    sort { |x, y| x.send(champ_X) <=> y.send(champ_X) }
  end
end
```

7. Autres exemples : Ruby on Rails

Qu'est-ce que *Ruby on Rails*?

*Ruby on Rails, or simply **Rails**, is a web application framework written in Ruby [...].*

Rails is a model–view–controller (MVC) framework, providing default structures for a database, a web service, and web pages. [...]

In addition to MVC, Rails emphasizes the use of other well-known software engineering patterns and paradigms, including convention over configuration (CoC), don't repeat yourself (DRY), and the active record pattern.

Source: `https://en.wikipedia.org/wiki/Ruby_on_Rails`

Historique de Ruby on Rails et de Ruby

Ruby	
Version	**Année**
1.0	1996
...	...
Progr. Ruby	2000
...	...
1.8	2003
...	...
1.9	2007
...	...
2.0	2013
...	...
2.3	Déc. 2015
2.4	Déc. 2016
2.5	Déc. 2017
2.6	Déc. 2018
3.0	2020

Rails	
Version	**Année**
1.0	2005
...	...
2.0	2007
...	...
3.0	2010
...	...
4.0	2013
...	...
4.2	Déc. 2014
5.0	Juin 2016
5.1	Mai 2017
5.2	Avril 2018

Les `ActiveRecord`s permettent d'interfacer facilement avec une BD relationnelle

Approche ORM = *Object-Relational Mapping*

```
$ rails new products

$ cd products

$ rails generate scaffold Product title:string\
                              description:text price:decimal

$ rails generate scaffold Order product:references\
                              quantity:integer

$ rake db:migrate
  .
  .
  .
```

Les `ActiveRecord`s définissent un DSL pour spécifier le schéma d'une BD relationnelle

```
$ cat db/schema.rb
ActiveRecord::Schema.define(version: 20151109232424) do
  create_table "orders", force: true do |t|
    t.integer  "product_id"
    t.integer  "quantity"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

  add_index "orders", ["product_id"],
                    name: "index_orders_on_product_id"

  create_table "products", force: true do |t|
    t.string   "title"
    t.text     "description"
    t.decimal  "price"
    t.datetime "created_at"
    t.datetime "updated_at"
  end
end
```

Les `ActiveRecord`s définissent le modèle, qu'on peut étendre avec des validations ou clés externes

```
$ cat app/models/product.rb
class Product < ActiveRecord::Base
end


$ emacs app/models/product.rb
class Product < ActiveRecord::Base
  validates :title, :description, presence: true
  validates :price,
         numericality: {greater_than_or_equal_to: 0.01}
  validates :title, uniqueness: true
  has_many :orders
end
```

Les `ActiveRecord`s définissent le modèle, qu'on peut étendre avec des validations ou clés externes

```
class Product < ActiveRecord::Base
  validates :title, :description, presence: true
  validates :price,
           numericality: {greater_than_or_equal_to: 0.01}
  validates :title, uniqueness: true
  has_many :orders
end
---------------------------------------------
> p = Product.new title: "Crayon rouge",
       description: "Un beau crayon rouge", price:0.99
 => #<Product id: nil, title: "Crayon rouge",
      description: "Un beau crayon rouge",
      price: #<BigDecimal:7fe9912ee6b8,'0.99E0',9(36)>,
      created_at: nil, updated_at: nil>
> p.valid?
  ...
 => true
> p.save
```

Le modèle permet d’accéder aux relations de la BD (créée par les `save/create`) et de faire des requêtes

```
> Product.all
  Product Load (0.4ms) SELECT "products".* FROM "products"
 => #<ActiveRecord::Relation
      [#<Product id: 1, title: "Crayon rouge",
         description: "Un beau crayon rouge",
         price: #<BigDecimal:7fe9912ee6b8,'0.99E0',9(36)>,
         [...]>
    ]>
```

Les ActiveRecords définissent le modèle, qu'on peut étendre avec des validations ou clés externes

```
class Product < ActiveRecord::Base
  validates :title, :description, presence: true
  validates :price,
            numericality: {greater_than_or_equal_to: 0.01}
  validates :title, uniqueness: true
  has_many :orders
end
---------------------------------------------
> p = Product.new title: "Crayon bleu",
         description: "Un beau crayon bleu"
  ...
> p.valid?
  ...
 => false
```

Les `ActiveRecord`s définissent le modèle, qu'on peut étendre avec des validations ou clés externes

```
class Product < ActiveRecord::Base
  validates :title, :description, presence: true
  validates :price,
            numericality: {greater_than_or_equal_to: 0.01}
  validates :title, uniqueness: true
  has_many :orders
end
---------------------------------------------
> p = Product.new title: "Crayon rouge",
      description: "Un autre beau crayon rouge", price: 0.99
  ...
> p.valid?
  ...
 => false
```

Les `ActiveRecord`s définissent le modèle, qu'on peut étendre avec des validations ou clés externes

```
class Product < ActiveRecord::Base
  validates :title, :description, presence: true
  validates :price,
            numericality: {greater_than_or_equal_to: 0.01}
  validates :title, uniqueness: true
  has_many :orders
end
---------------------------------------------
> p = Product.new title: "Crayon bleu",
           description: "Un beau crayon bleu", price:1.10
 => #<Product id: nil, title: "Crayon bleu",
          description: "Un beau crayon bleu",
          price: #<BigDecimal:7fe9912ecd90,'0.11E1',18(36)>,
          created_at: nil, updated_at: nil>
> p.save
...
```

Le modèle permet d’accéder aux relations de la BD (créée par les `save/create`) et de faire des requêtes

```
> Product.all
  Product Load (0.4ms) SELECT "products".* FROM "products"
 => #<ActiveRecord::Relation
       [#<Product id: 1, title: "Crayon rouge",
          description: "Un beau crayon rouge",
          price: #<BigDecimal:7fe9912ee6b8,'0.99E0',9(36)>,
          [...]>,
        #<Product id: 2, title: "Crayon bleu",
          description: "Un beau crayon bleu",
          price: #<BigDecimal:7fe9912ecd90,'0.11E1',18(36)>,
          [...]>
    ]>
```

Le modèle permet d’accéder aux relations de la BD (créée par les `save/create`) et de faire des requêtes

```
> Product.where( "title = 'Crayon rouge'" )
  Product Load (0.2ms) SELECT "products".* FROM "products"
          WHERE (title = 'Crayon rouge')
 => #<ActiveRecord::Relation [
       #<Product id: 1, title: "Crayon rouge",
         description: "Un beau crayon rouge",
         price: #<BigDecimal:7fe9912ee6b8,'0.99E0',9(36)>,
         created_at: "2015-11-09 21:37:56",
         updated_at: "2015-11-09 21:37:56">
    ]>
```

Le modèle permet d’accéder aux relations de la BD (créée par les `save/create`) et de faire des requêtes

```
> Product.find_by title: "Crayon rouge"
  Product Load (0.2ms) SELECT "products".* FROM "products"
          WHERE "products"."title" = 'Crayon rouge' LIMIT 1
 => #<Product id: 1, title: "Crayon rouge",
      description: "Un beau crayon rouge",
      price: #<BigDecimal:7fe9912ee6b8,'0.99E0',9(36)>,
      created_at: "2015-11-09 21:37:56",
      updated_at: "2015-11-09 21:37:56">
```

Le modèle permet d'accéder aux relations de la BD (créée par les `save/create`) et de faire des requêtes

```
> Product.find_by_title "Crayon rouge"
  Product Load (0.2ms) SELECT "products".* FROM "products"
          WHERE "products"."title" = 'Crayon rouge' LIMIT 1
 => #<Product id: 1, title: "Crayon rouge",
      description: "Un beau crayon rouge",
      price: #<BigDecimal:7fe9912ee6b8,'0.99E0',9(36)>,
      created_at: "2015-11-09 21:37:56",
      updated_at: "2015-11-09 21:37:56">
```

Les associations multiples définissent automatiquement des collections

```
> (Order.new product_id: 1, quantity:3).save
> Order.create product_id: 1, quantity:99
> Order.create product_id: 1, quantity:12

> p = Product.find_by(title: "Crayon rouge")
  Product Load (0.3ms) SELECT "products".* FROM "products"
         WHERE "products"."title" = 'Crayon rouge' LIMIT 1
=> #<Product id: 1, title: "Crayon rouge", [...]>

> p.orders
  Order Load (0.1ms)  SELECT "orders".* FROM "orders"
        WHERE "orders"."product_id" = ? [["product_id", 1]]
=> #<ActiveRecord::Associations::CollectionProxy [
     #<Order id: 1, product_id: 1, quantity: 3,  [...]>,
     #<Order id: 2, product_id: 1, quantity: 99, [...]>,
     #<Order id: 3, product_id: 1, quantity: 12, [...]>
   ]>
```

Les durées sont définies par des extensions de la classe `Numeric`

```
1.minute == 60.seconds
10.minutes == 600.seconds
1.minute > 59.seconds

2.minutes == 120.seconds
2.minutes >  59.seconds

#######################

Time.current
=> Tue, 10 Nov 2015 00:18:47 UTC +00:00

8.hours.ago
=> Mon, 09 Nov 2015 16:18:49 UTC +00:00

8.hours.from_now
=> Tue, 10 Nov 2015 08:18:51 UTC +00:00
```

8. Conclusion

La métaprogrammation est un outil puissant. . .

Avantages :

- Permet d'éliminer la redondance dans le code
- Permet d'avoir du code flexible
- Permet de définir des DSL expressifs
- . . .

La métaprogrammation est un outil puissant… mais qui peut être dangereux

Avantages :

- Permet d'éliminer la redondance dans le code
- Permet d'avoir du code flexible
- Permet de définir des DSL expressifs
- …

Désavantages :

- Code (mise en œuvre) parfois difficile à comprendre
- Parfois difficile à déboguer (le code «réel» n'est pas là !)

La métaprogrammation est un outil puissant. . . mais parfois dangereux

Bien que les détails et approches diffèrent, tous les langages dynamiques offrent des constructions de métaprogrammation

Clojure	macros
Elixir	macros
Groovy	`methodMissing`
Python	`__getattr__`
Lisp	macros
Smalltalk	`doesNotUnderstand`

La métaprogrammation est aussi possible… en Java — c'est juste (beaucoup !) plus compliqué ☹

Une classe `Foo`

```
class Foo {
  private int val;

  Foo( int val ) {
    this.val = val;
  }

  private void inc() {
    val += 1;
  }

  int val() {
    return val;
  }
}
```

La métaprogrammation est aussi possible… en Java — c'est juste (beaucoup !) plus compliqué ☹

Quel est l'effet de ce programme ?

```
class CallFoo {
  public static void main( String args[] ) {
    Foo f = new Foo( 99 );
    f.inc();
  }
}
```

La métaprogrammation est aussi possible… en Java — c'est juste (beaucoup !) plus compliqué ☹

Quel est l'effet de ce programme ?

```
class CallFoo {
  public static void main( String args[] ) {
    Foo f = new Foo( 99 );
    f.inc();
  }
}
```

Erreur de compilation ☹

```
$ javac CallFoo.java
CallFoo.java:4: error: inc() has private access in Foo
     f.inc();
      ^
1 error
```

La métaprogrammation est possible… même en Java — c'est juste (beaucoup !) plus compliqué ☹

Quel est l'effet de ce programme (bis) ?

```
class CallFoo
  public static void main( String args[] ) {
    Foo f = new Foo( 99 );
    try {  java.lang.reflect.Method m =
              Foo.class.getDeclaredMethod( "inc" );
          m.setAccessible(true);
          m.invoke( f );
    } catch( Throwable t ) {}
    System.out.println( "val = " + f.val() );
  }
}
```

La métaprogrammation est possible... même en Java — c'est juste (beaucoup !) plus compliqué ☹

Quel est l'effet de ce programme (bis) ?

```
class CallFoo
  public static void main( String args[] ) {
    Foo f = new Foo( 99 );
    try {  java.lang.reflect.Method m =
              Foo.class.getDeclaredMethod( "inc" );
           m.setAccessible(true);
           m.invoke( f );
    } catch( Throwable t ) {}
    System.out.println( "val = " + f.val() );
  }
}
```

Il appelle la méthode `inc()`, privée !, puis `val()`, donc...

```
$ javac CallFoo.java
$ java CallFoo
val = 100
```

Références

G.T. Brown.
Ruby Best Practices.
O'Reilly, 2009.

M. Fowler.
Domain-Specific Languages.
Addison-Wesley, 2011.

P.J. Jones.
Effective Ruby : 48 Specific Ways to Write Better Ruby.
Addison-Wesley Professional, 2014.

P. Perrotta.
Metaprogramming Ruby.
The Pragmatic Bookshelf, 2010.