Introduction au langage Ruby par des exemples

MGL7460

Automne 2016

Table des matières

1 Introduction : Pourquoi Ruby?

Le langage Ruby a été conçu, au milieu des années 90, par Yukihiro Matsumoto, un programmeur Japonais. Son objectif était d'avoir un langage qui soit « plaisant» à utiliser :

> *Ruby is* **"made for developer happiness"***!*
>
> Y. Matsumoto

Y. Matsumoto s'est inspiré de plusieurs langages de programmation : Perl [WCS96] (pour le traitement de texte et les expressions régulières) Smalltalk [Gol89] (pour ses blocs et pour son approche orientée objet «pure» où tout est objet), CLU [LG86] (pour ses itérateurs), Lisp [Ste84] (pour ses fonctions d'ordre supérieur). La figure 1 présente un arbre généalogique de Ruby. Quant au tableau 1, il présente les «ancêtres» de Ruby, avec les principales caractéristiques héritées de ces ancêtres.

Langage	**Année**	**Caractéristiques**
Lisp	1958	approche fonctionnelle métaprogrammation
CLU	1974	itérateurs
Smalltalk	1980	langage objet pur, blocs de code GUI, sUnit
Eiffel	1986	*Uniform Access Principle*
Perl	1987	expressions régulières et *pattern matching*
Ruby	1993	

Tableau 1: Les ancêtres de Ruby.

Ruby a été popularisé notamment par le développement de Ruby on Rails [Dix11, Har13, Lew15, RTH13], un *framework* pour la mise en oeuvre et le déploiement d'applications Web.

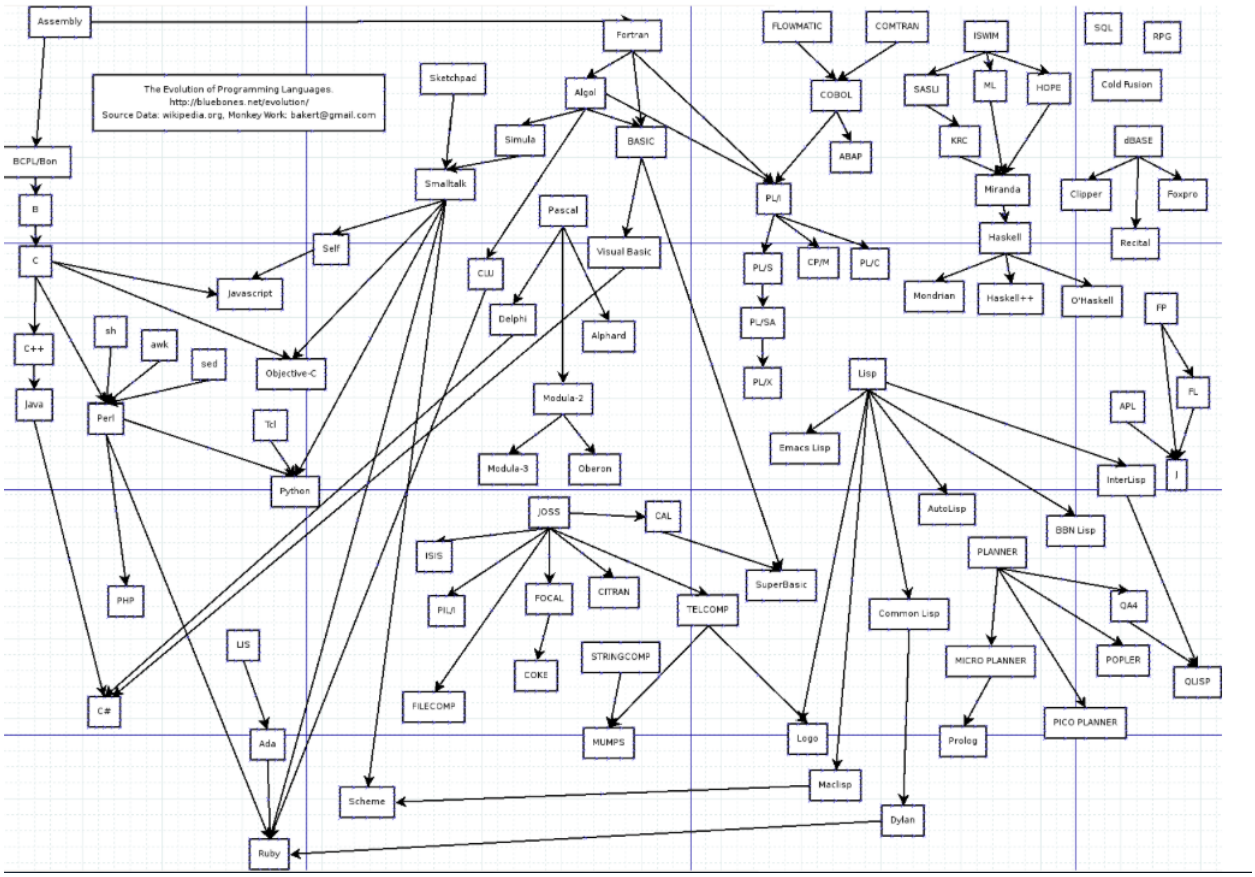


Figure 1: Arbre généalogique de divers langages de programmation, incluant Ruby.
Source: `https://www.madetech.com/blog/pros-and-cons-of-ruby-on-rails`.

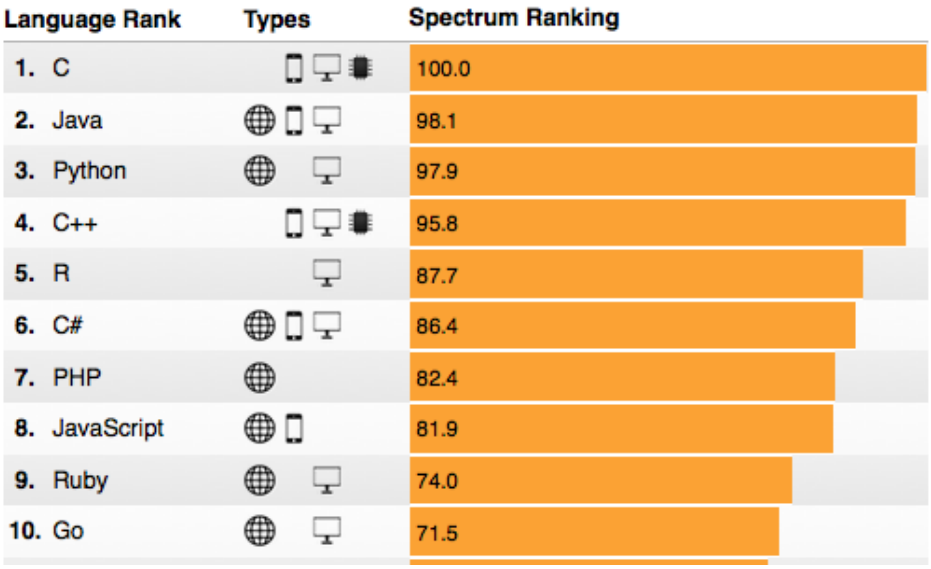


Figure 2: Les 10 langages de programation les plus utilisés (2016) selon IEEE Spectrum.

Selon une enquête faite par la revue IEEE Spectrum en 2016,[1] Ruby est parmi les 10 langages de programmation les plus utilisés : voir Figure 2.

Philosophie de Ruby

Ruby, comme Perl et Python [Bla04], est un langage à typage dynamique — un langage dit de «script» — avec une syntaxe *flexible* (on verra comment plus loin) :

> *Ruby... est un langage open-source* **dynamique** *qui met l'accent sur la simplicité et la productivité. Sa syntaxe élégante en facilite la lecture et l'écriture.* `https://www.ruby-lang.org/fr/`

> Dixit Yukihiro Matsumoto, le concepteur de Ruby : *Ruby inherited the Perl philosophy* **of having more than one way to do the same thing**. *[...] I want to make Ruby users free. I want to give them the freedom to choose. People are different. People choose different criteria. But if there is a better way among many alternatives, I want to encourage that way by making it comfortable. So that's what I've tried to do.*
>
> Source : `http://www.artima.com/intv/rubyP.html`

```
$ rvm list known
# MRI Rubies
[ruby-]1.8.6[-p420]
[ruby-]1.8.7[-head] # security released on head
[ruby-]1.9.1[-p431]
[ruby-]1.9.2[-p330]
[ruby-]1.9.3[-p551]
[ruby-]2.0.0[-p643]
[ruby-]2.1.4
[ruby-]2.1[.5]
[ruby-]2.2[.1]
[ruby-]2.2-head
ruby-head

# JRuby
jruby-1.6.8
jruby[-1.7.19]
jruby-head
jruby-9.0.0.0.pre1

# Rubinius
rbx-1.4.3
rbx-2.4.1
rbx[-2.5.2]
rbx-head

# Opal
opal

# Minimalistic ruby implementation - ISO 30170:2012
mruby[-head]

# Ruby Enterprise Edition
ree-1.8.6
ree[-1.8.7][-2012.02]

# GoRuby
goruby

# Topaz
topaz

# MagLev
maglev[-head]
maglev-1.0.0

# Mac OS X Snow Leopard Or Newer
macruby-0.10
macruby-0.11
macruby[-0.12]
macruby-nightly
macruby-head

# IronRuby
ironruby[-1.1.3]
ironruby-head
```

Figure 3: Les mises en oeuvre de Ruby disponibles par l'intermédiaire de `rvm` (février 2016) — voir annexe A .

Figure 4: Quelques organisations qui utilisent `JRuby`. Source : «*JRuby 9000 Is Out; Now What?*, T. Enebo and C. Nutter, RubyConf 2015, `https://www.youtube.com/watch?v=KifjmbSHHs0`

Mises en oeuvre de Ruby

Il existe plusieurs mises en oeuvre de Ruby, par exemple, MRI (Matz's Ruby Interpreter, parfois appelé CRuby car sa mise en oeuvre est en langage C), Rubinius (mise en oeuvre... *en Ruby*), JRuby (mise en oeuvre en Java sur la JVM = *Java Virtual Machine*).

Dans ce qui suit, nous allons introduire le langage Ruby par l'intermédiaire de divers exemples. Dans certains exemples, `jruby` est utilisé pour évaluer certaines expression, mais ces expressions auraient tout aussi bien pu être évaluées avec `ruby/MRI`.

[1] `http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016`

2 Compilation et exécution de programmes Ruby

Exemple Ruby 1 Deux versions d'un programme «Hello world!».

```
$ cat hello0.rb
puts 'Bonjour le monde!'

$ ruby hello0.rb
Bonjour le monde!

-------------------------------------

$ cat hello1.rb
#!/usr/bin/env ruby

puts 'Bonjour le monde!'

$ ls -l hello1.rb
-rwxr-xr-x. 1 tremblay tremblay 46 26 jun 09:52 hello1.rb*

$ ./hello1.rb
Bonjour le monde!
```

Remarques et explications pour l'exemple Ruby 1 :

- L'instruction `puts` (`puts`*string*) affiche la chaîne indiquée en argument sur la sortie standard (`STDOUT`) et ajoute un saut de ligne — on peut utiliser «`print`» pour ne pas ajouter automatiquement un saut de ligne.
- Ruby étant un langage dynamique — on dit aussi un «langage de script» — la compilation et l'exécution se font à l'aide d'une seule et même commande :
  - À l'aide d'un appel *explicite* à la commande `ruby` : «`ruby hello0.rb`».
  - À l'aide d'un appel *implicite* à la commande `ruby` : «`./hello1.rb`», la première ligne (débutant par «`#!`») de `hello1.rb` indiquant que le script doit être compilé et exécuté avec `ruby`.
- Bien qu'on parle souvent d'un processus «d'interprétation», les versions plus récentes de Ruby utilisent un *compilateur* : une première passe analyse et compile le programme pour générer du code-octet, puis ce code-octet est exécuté par la machine virtuelle de Ruby (ou par la JVM dans le cas de JRuby).

3 `irb` : Le *shell* interactif Ruby

Exemple Ruby 2 `irb`, le shell interactif de Ruby.

```
$ irb --prompt=simple
>> 10
=> 10

>> 2 + 4
=> 6

>> puts 'Bonjour le monde!'
Bonjour le monde!
=> nil

>> r = puts 'Bonjour le monde!'
Bonjour le monde!
=> nil
>> r
=> nil

>> puts( 'Bonjour le monde!' )
Bonjour le monde!
=> nil

>> STDOUT.puts( 'Bonjour le monde!' )
Bonjour le monde!
=> nil

>> STDERR.puts( 'Bonjour le monde!' )
Bonjour le monde!
=> nil

>> STDIN.puts( 'Bonjour le monde!' )
IOError: not opened for writing
     from org/jruby/RubyIO.java:1407:in 'write‘
     [...]
     from /home/tremblay/.rvm/rubies/jruby-1.7.16.1/bin/irb:13:in
     '(root)'
```

```
>> # _ denote la valeur de la derniere expression evaluee.

>> 8 * 100 / 2
=> 400

>> _ + _
=> 800

>> _ / 3
=> 266

>> _ / 3.0
=> 88.66666666666667
```

```
# On peut creer une nouvelle "session" (interne) qui modifie self,
# l'objet courant.
>> irb [10, 20]

>> self.class
=> Array

>> self
=> [10, 20]

>> size
=> 2

>> irb "abcde"
>> self
=> "abcde"

>> ^D
=> #<IRB::Irb: @context=#<IRB::Context:0x0000000170a660>,
   @signal_status=:IN_EVAL, @scanner=#<RubyLex:0x0000000191a7c0>>

>> self
=> [10, 20]
```

Remarques et explications pour l'exemple Ruby 2 :

- Une autre façon d'exécuter du code Ruby est d'utiliser `irb` — *interactive Ruby*.
- Le processus d'évaluation d'`irb` — le *REPL* = *Read-Eval-Print-Loop* — procède comme suit :
  - À l'invite de commande (*prompt*) indiquée par «`>>` », on entre une expression — par ex., «`>> 2 + 4`».[2]
  - L'expression est évaluée.
  - Le résultat retourné par l'expression est affiché après le symbole «`=>`» — par ex., «`=> 6`».
- Le premier exemple de `puts` montre que si l'expression évaluée affiche quelque chose sur `STDOUT`, alors cela s'affiche *avant* que le résultat retourné par l'expression ne soit affiché. On note aussi que le résultat retourné par un appel à `puts` est `nil`.
- Les autres exemples illustrent ce qui suit :
  - Dans un appel de méthode, les parenthèses — à moins qu'il n'y ait ambiguité à cause de la précédence des opérateurs : voir plus bas — peuvent être omises. Donc, un appel tel que «`puts 'xxx'`» est équivalent à l'appel «`puts( 'xxx' )`».
  - Un appel direct à `puts` est équivalent à un appel de `puts` sur `STDOUT`, la constante qui dénote le flux standard de sortie. Lors d'une session interactive `irb`, tant `STDOUT` que `STDERR` sont associés à l'écran. Quant au flux d'entrée `STDIN`, il est aussi disponible et est associé par défaut au clavier, donc il ne peut évidemment pas être utilisé pour effectuer des écritures.
- L'identificateur «`_`» est toujours associé à la valeur produite par la dernière expression évaluée.
- On peut créer une nouvelle session, qui modifie l'objet courant, i.e., `self`. Ceci permet donc d'examiner plus facilement un objet et ses propriétés.

  On termine une session avec «Ctrl-D».

Dans plusieurs des exemples qui suivent, ce sont les résultats produits avec `irb` qui seront affichés, bien que dans certains cas les détails affichés seront quelque peu simplifiés pour faciliter la lecture des exemples.

[2]Plus précisément, ce *prompt* est obtenu en utilisant la commande «`irb --prompt=simple`». Si cette option est omise, alors le *prompt* affiche des informations additionnelles, par exemple, «`jruby-1.7.16.1 :001 >`» — la version de Ruby utilisée et un numéro pour l'expression.

4 Tableaux

Exemple Ruby 3 Les tableaux et leurs opérations de base.

```
>> # Valeurs litterales, indexation et taille.
?> a = [10, 20, 30]
=> [10, 20, 30]

>> a[0]
=> 10

>> a[2]
=> 30

>> a[2] = 55
=> 55

>> a
=> [10, 20, 55]

>> a.size
=> 3
```

```
?> # Valeur nil par defaut et extension de la taille.
?> a[6]
=> nil

>> a.size
=> 3

>> a[5] = 88
=> 88

>> a
=> [10, 20, 55, nil, nil, 88]

>> a.size
=> 6
```

```
?> # Acces au 'dernier' element.
?> a[a.size-1]
=> 88

>> a[-1]
=> 88
```

Remarques et explications pour l'exemple Ruby 3 :

- Un commentaire débute par «`#`» et se termine à la fin de la ligne. Une ligne ne contenant qu'un commentaire n'a donc pas d'expression à évaluer et, dans `irb`, l'entrée se continue à la ligne suivante avec le *prompt* de continuation («`?>` »).

  Il est possible d'indiquer un commentaire formé par *un bloc de lignes* — bien que ce soit utilisé assez peu fréquemment sauf pour certaines formes de documentation (e.g., RDoc) :

  ```
  =begin
  Blah blah
  ...
  =end
  ```

- L'indice du premier élément d'un tableau, comme en C et en Java, est 0.

- Un tableau `a` est automatiquement étendu à la taille appropriée si on affecte à un indice plus grand ou égal à `a.size`. Les valeurs non explicitement définies sont alors égales à `nil`.

- L'indice du dernier élément d'un tableau `a` est `a.size-1`. On peut aussi accèder au dernier élément avec l'indice `-1`, à l'avant-dernier avec l'indice `-2`, etc.

Exemple Ruby 4 Les tableaux et leurs opérations de base (suite 1).

```
?> # Tableaux heterogenes.
?> a
=> [10, 20, 55, nil, nil, 88]

>> a[8] = 'abc'
=> "abc"

>> a
=> [10, 20, 55, nil, nil, 88, nil, nil, "abc"]
```

```
?> # Ajout d'elements.
?> a = []
=> []

>> a << 12
=> [12]

>> a << 'abc' << [2.7, 2.8]
=> [12, "abc", [2.7, 2.8]]
```

```
?> # Creation de tableaux avec valeurs initiales.
?> b = Array.new(3) { 10 }
=> [10, 10, 10]

>> d = Array.new(4)
=> [nil, nil, nil, nil]
```

Remarques et explications pour l'exemple Ruby 4 :

- Les tableaux sont *hétérogènes*, i.e., ils peuvent contenir des éléments de types variés.

- Il existe de nombreuses opérations pour ajouter ou retirer des éléments d'un tableau, par exemple, `push`, `pop`, `shift`, `unshift`, etc.[3]
- Une opération fréquemment utilisée est `push`, qui a comme alias «`<<`».
- L'opération `new` permet de créer un tableau d'une certaine taille de départ et permet aussi de spécifier, de façon optionnelle, une valeur initiale pour les différents éléments du tableau.

Exemple Ruby 5 Les tableaux et leurs opérations de base (suite 2).

```
?> # Tranches de tableaux.
?> a = [10, 20, 30, 40, 50]
=> [10, 20, 30, 40, 50]

>> a[0..2]
=> [10, 20, 30]

>> a[3..3]
=> [40]

>> a[1..-1]
=> [20, 30, 40, 50]

>> a[7..7]
=> nil
```

Remarques et explications pour l'exemple Ruby 5 :

- On peut obtenir une *tranche* d'un tableau en spécifiant comme indice un `Range`. Un `Range` avec «..» est inclusif — par ex., `b_inf..b_sup` — donc inclut tant la borne inférieure que la borne supérieure. Par contre, un `Range` avec «...» est exclusif, i.e., inclut la borne inférieure mais *exclut* la borne supérieure.

[3] Voir `http://ruby-doc.org/core-2.2.0/Array.html` pour la liste complète.

```
?> # Intervalles inclusifs vs. exclusifs
>> a
=> [10, 20, 30, 40, 50]

>> a[1..3]
=> [20, 30, 40]

>> a[1...3]
=> [20, 30]

>> a[1..a.size-1]
=> [20, 30, 40, 50]

>> a[1...a.size]
=> [20, 30, 40, 50]
```

5 Chaînes de caractères

Exemple Ruby 6 Les chaînes de caractères et leurs opérations de base.

```
>> # String semblable a Array.
?> s1 = 'abc'
=> "abc"

>> s1.size
=> 3

>> s1[0..1]  # Retourne String.
=> "ab"

>> s1[2]     # Retourne String aussi!
=> "c"
```

```
?> # Concatenation vs. ajout.
?> s1 + 'def'
=> "abcdef"

>> s1
=> "abc"

>> s1 << 'def'
=> "abcdef"

>> s1
=> "abcdef"
```

Remarques et explications pour l'exemple Ruby 6 :

- Un `String` est, par certaines opérations, semblable à un `Array`, notamment, on peut l'indexer pour obtenir un élément ou une sous-chaîne.
- L'indexation d'un `String`... retourne un `String`,[4] que l'indice soit un `Range` ou un `Fixnum`.

[4] Depuis Ruby 1.9.

- L'opération «+» concatène deux chaînes pour produire une nouvelle chaîne (opération fonctionnelle, immuable) alors que l'opérateur «<<» ajoute (`append`) des caractères à la fin d'une chaîne existante (opération impérative, mutable).

Exemple Ruby 7 Les chaînes de caractères et leurs opérations de base (suite).

```
>> # Egalite de valeur *sans* partage de reference.
?> a, b = 'abc', 'abc'
=> ["abc", "abc"]

>> a == b
=> true

>> a.equal? b
=> false

>> a[0] = 'X'
=> "X"

>> a
=> "Xbc"

>> b
=> "abc"
```

Remarques et explications pour l'exemple Ruby 7 :

- Ruby possède plusieurs opérations de comparaison d'égalité. Les deux premières sont les suivantes :
  - «==» : Comparaison de *valeur*. C'est généralement cette opération que l'on spécifie dans les classes que l'on définit nous-mêmes.
  - «`equal?`» : Comparaison d'*identité*, i.e., est-ce que les deux éléments comparés dénotent le même objet (le même pointeur, la même référence)?

  Note : La signification de ces opérateurs est *l'inverse* de celle de Java, où «==» est la comparaison d'identité alors que `equals` est la comparaison de valeur.
- En Ruby, il est considéré de bon style qu'une méthode qui retourne un booléen se termine par «?», par exemple, `equal?`, `empty?`, `nil?`, `block_given?`, etc.

```
?> # Egalite de valeur *avec* partage de reference.
?> a = b = 'abc'
=> "abc"

>> a == b
=> true

>> a.equal? b
=> true

>> a[0] = 'X'
=> "X"

>> a
=> "Xbc"

>> b
=> "Xbc"
```

Exemple Ruby 8 Interpolation d'une expression dans une chaîne.

```
>> # Interpolation d'une expression dans une chaine.
?> x = 123
=> 123

?> "abc \"#{x}\" def"
=> "abc \"123\" def"

?> "abc '#{10 * x + 1}' def"
=> "abc '1231' def"

?> "abc #{x > 0 ? '++' : 0} def"
=> "abc ++ def"

?> # Une String peut etre definie avec '...' mais sans interpolation.
?> 'abc "#{x}" def'
=> "abc \"\#{x}\" def"
```

Remarques et explications pour l'exemple Ruby 8 :

- Une chaîne produite avec les doubles guillemets permet d'*interpoler* une ou des expressions. Dans une telle chaîne, «`#{...}`» indique alors une expression qui doit être évaluée — on dit aussi *interpolée*. Si cette expression produit un résultat qui n'est pas une chaîne, alors la méthode `to_s` sera implicitement appelée — voir plus bas.

- On peut aussi définir une chaîne avec de simples guillemets, mais dans ce cas *aucune interpolation n'est effectuée* (chaine textuelle).

  Donc, les guillemets doubles vs. apostrophes simples ont la même interprétation qu'en `bash`!

Exemple Ruby 9 Opérations `split` et `join`.

```
# Split decompose une chaine en sous-chaines
# en fonction du <<motif>> specifie en argument.

>> s = "abc\ndef\nghi\n"
=> "abc\ndef\nghi\n"

>> s.split("\n")            # Un cas typique!
=> ["abc", "def", "ghi"]

>> s.split("def")
=> ["abc\n", "\nghi\n"]

>> s.split("a")
=> ["", "bc\ndef\nghi\n"]

>> s.split(/\w{3}/)         # \w = [a-zA-Z0-9_]
=> ["", "\n", "\n", "\n"]
```

```
# Join combine un tableau de sous-chaines
# en une chaine unique.

>> s
=> "abc\ndef\nghi\n"

>> r = s.split("\n")
=> ["abc", "def", "ghi" ]

>> r.join("+")
=> "abc+def+ghi"

>> r.join("\n")  # Donc: s.split("\n").join("\n") != s
=> "abc\ndef\nghi"


>> [].join(";")
=> ""

>> ['abc'].join(";")
=> "abc"

>> ['abc', 'def'].join(";")
=> "abc;def"
```

Remarques et explications pour l'exemple Ruby 9 :

- La méthode `split` décompose une chaine pour produire un tableau de sous-chaines, et ce en utilisant le séparateur indiqué.
- Le séparateur peut être un caractère, une chaine ou une expression régulière. Les séparateurs *ne font pas partie* des sous-chaines retournées par `split`.
- Une sous-chaine peut être la chaine vide (`""`) si la chaine continent deux séparateurs consécutifs ou si la chaine débute ou se termine par un séparateur.
- La méthode `join` reçoit un tableau de chaines et les combine en une seule chaine en utilisant la chaine indiquée.

6 Symboles

Exemple Ruby 10 Les symboles.

```
>> # Symbole = "sorte" de chaine *unique et immuable*.
>> :abc
=> :abc

>> :abc.class
=> Symbol

>> :abc.to_s
=> "abc"

>> puts :abc
abc
=> nil

>> :abc[2]
=> "c"

>> :abc[2] = "x"
NoMethodError: undefined method '[]=' for :abc:Symbol
    from (irb):4
    from /home/tremblay/.rvm/rubies/ruby-2.1.4/bin/irb:11:in '<main>'

>> "abc".to_sym
=> :abc

>> "abc def .!#%".to_sym
=> :"abc def .!#%"
```

Remarques et explications pour l'exemple Ruby 10 :

- Un *symbole* — classe `Symbol` — est une représentation unique d'un identificateur.
- Un identificateur débutant par «:» est un *symbole*.
- Un symbole est une *sorte* de chaîne mais *unique et immuable*.

```
?> # Possede un numero d'identification unique.
>> :a
=> :a

>> :a.object_id
=> 365128

>> "a".object_id
=> 11000000

>> "a".object_id
=> 10996280

>> :a.object_id
=> 365128

>> "a".to_sym.object_id
=> 365128
```

Ainsi, bien que deux chaînes peuvent avoir la même valeur tout en étant distinctes l'une de l'autre — `ch1 == ch2` mais `!(ch1.equal? ch2)` — ce n'est pas le cas avec les symboles — `sym1 == sym2` implique `sym1.equal? sym2`.

- On peut obtenir le symbole associé à une chaîne arbitraire à l'aide de la méthode `to_sym`.
- On peut observer l'unicité d'un symbole en obtenant son `object_id` — un entier qui identifie de façon unique n'importe quel objet (≈ l'adresse de l'objet en mémoire!).

  Deux chaines peuvent être égales mais ne pas être le même objet, donc avoir deux `object_id` distincts. Par contre, chaque référence à un symbole retourne toujours le même `object_id`.
- Les symboles sont souvent utilisés comme clés pour les *hashes*, de même que pour les *keyword arguments* — voir plus bas.

```
?> # Egalite de valeur vs. de reference.
?>
>> :abc == :abc
=> true

>> :abc.equal? :abc
=> true



>> "abc" == "abc"
=> true

>> "abc".equal? "abc"
=> false



>> "abc".to_sym == "abc".to_sym
=> true

>> "abc".to_sym.equal? "abc".to_sym
=> true
```

7 Hashes

Exemple Ruby 11 Les *hashes* et leurs opérations de base.

```
>> # Definition d'un hash.
?> hash = { :abc => 3, :de => 2, :ghijk => 5 }
=> {:abc=>3, :de=>2, :ghijk=>5}

?> # Principales proprietes.
?> hash.size
=> 3

>> hash.keys
=> [:abc, :de, :ghijk]

>> hash.values
=> [3, 2, 5]

>> # Indexation.
?> hash[:abc]
=> 3

>> hash[:de]
=> 2

>> hash["de"]
=> nil
```

Exemple Ruby 12 Les *hashes* et leurs opérations de base (suite).

```
?> # Definition d'une nouvelle cle.
?> hash.include? "de"
=> false

>> hash["de"] = 55
=> 55

>> hash.include? "de"
=> true

?> # Redefinition d'une cle existante.
?> hash[:abc] = 2300
=> 2300

>> hash
=> {:abc=>2300, :de=>2, :ghijk=>5, "de"=>55}
```

```
?> # Creation d'un hash avec valeur par defaut.
?> h2 = Hash.new( 0 )
=> {}

>> h2[:xyz]
=> 0
```

Remarques et explications pour les exemples Ruby 11–12 :

- Les *hashs* — classe `Hash` — sont aussi appelés *dictionnaires*, *maps*.
- Un *hash* peut être vu comme une forme généralisée de tableau, au sens où l'index — la clé — est arbitraire et non pas un simple entier plus grand ou égal à 0.

  On peut aussi interpréter un *hash* comme une *fonction* associant des valeurs (`values` = codomaine) à des clés (`keys` = domaine). Donc, étant donné une *clé*, le *hash* retourne la valeur associée.

- La valeur associée à une clé peut être modifiée, et ce simplement en affectant une valeur nouvelle, par exemple, «`hash["de"] = 55`». Un objet de classe `Hash` est donc un objet *mutable*.
- Si aucune valeur par défaut n'a été spécifiée pour un *hash*, la valeur retournée pour une clé non définie est `nil`. Par contre, au moment de la création du *hash*, il est possible de spécifier la valeur qui doit être retournée par défaut, i.e., la définition à retourner si la clé n'est pas explicitement définie.

8 Expressions booléennes

Le point important à retenir pour comprendre les expressions booléennes :

- `false` et `nil` sont des valeurs «fausses»
- *Toute autre valeur* est «vraie».

Quelques exemples avec l'opérateur ternaire ?: — voir aussi plus bas :

```
>>  false ? 'oui' : 'non'
=> "non"

>> nil ? 'oui' : 'non'
=> 'non'

>>  0 ? 'oui' : 'non'
=> "oui"

>>  '' ? 'oui' : 'non'
(irb):5: warning: string literal in condition
=> "oui"

>> nil.nil? ? 'nil' : 'pas nil'
=> "nil"
```

Exemple Ruby 13 Les expressions booléennes.

```
>> # Toute valeur differente de false ou nil est vraie.
?> true ? 'oui' : 'non'
=> "oui"

>> 0 ? 'oui' : 'non'
=> "oui"

>> [] ? 'oui' : 'non'
=> "oui"

?> # Seuls false et nil ne sont pas vraies.
?> false ? 'oui' : 'non'
=> "non"

>> nil ? 'oui' : 'non'
=> "non"

>> !false ? 'oui' : 'non'
=> "oui"

>> !nil ? 'oui' : 'non'
=> "oui"

?> # Seul nil est nil
?> 2.nil? ? 'nil' : 'pas nil'
=> "pas nil"

>> [].nil? ? 'nil' : 'pas nil'
=> "pas nil"

>> nil.nil? ? 'nil' : 'pas nil'
=> "nil"
```

Remarques et explications pour l'exemple Ruby 13 :

- En Ruby, toute valeur *différente de* `false` *ou* `nil` peut être interprétée comme une valeur «vraie».
- Bien que `nil` et `false` soient tous deux faux, seul `nil` est `nil?`.

Exemple Ruby 14 Les expressions booléennes (suite 1).

```
?> # Les expressions && et || sont court-circuitees.
?> true || (3 / 0) ? true : false
=> true

>> false && (3 / 0) ? true : false
=> false

?> false || (3 / 0) ? true : false
ZeroDivisionError: divided by 0
    [...]

>> true && (3 / 0) ? true : false
ZeroDivisionError: divided by 0
    [...]
```

Remarques et explications pour l'exemple Ruby 14 :

- Bien qu'il existe aussi des opérateurs `and` et `or`, on utilise plutôt les opérateurs `&&` et `||`.

  Ces derniers opérateurs sont évalués en mode «court-circuité». En d'autres mots, on évalue uniquement la portion d'expression nécessaire pour s'assurer que le résultat soit vrai ou faux, selon l'opérateur utilisé — puisqu'on a que «`false && x == x`» et que «`true || x == x`».

Exemple Ruby 15 Les expressions booléennes (suite 2).

```
>? # L'operateur || retourne la premiere expression 'non fausse'.
?> 2 || 3
=> 2

>> nil || false || 2 || false
=> 2

>> nil || false
=> false

>> false || nil
=> nil
```

Exemple Ruby 16 Les expressions booléennes (suite 3).

```
# On peut utiliser ||= pour initialiser une variable,
# sauf si elle est deja initialisee.

>> x
NameError: undefined local variable or method 'x' for main:Object
    [...]

>> x ||= 3
=> 3
>> x
=> 3

>> x ||= 8
=> 3
>> x
=> 3
```

Remarques et explications pour les exemples Ruby 15–16 :

- On a dit que l'opérateur «`||`» était évalué de façon court-circuitée.

  Une autre façon d'interpréter son comportement est de dire qu'il retourne la première valeur non fausse (différente de `false` ou de `nil`), ou sinon la dernière valeur.

- L'opérateur «`||=`» est semblable, mais pas complètement, à l'opérateur «`+=`» — et à de nombreux autres opérateurs — en ce qu'il dénote une forme abréviée d'une expression binaire :[5]

  ```
  x += 1      # x = x + 1
  x /= 2      # x = x / 2

  x ||= 1     # x || x = 1  ET non pas x = x || 1
  x &&= 1     # x && x = 1  Et non pas x = x && 1
  ```

- On utilise souvent l'opérateur «`||=`» pour donner une valeur initiale à une variable *à la condition qu'elle ne soit pas déjà initialisée*.

[5] `http://www.rubyinside.com/what-rubys-double-pipe-or-equals-really-does-5488.html`

9 Définitions et appels de méthodes

Exemple Ruby 17 Définitions et appels de méthodes.

```
>> # Definition et appels de methode.
   def add( x, y )
     x + y
   end

>> add( 2, 3 )
=> 5
>> add 20, 30      # Les parentheses sont optionnelles.
=> 50

>> # Resultat = derniere expression evaluee.
   def abs( x )
     if x < 0 then -1 * x else x end
   end

>> abs( 3 )
=> 3
>> abs( -3 )
=> 3
```

```
>> # On utilise return pour sortir 'avant la fin'.
   def abs2( x )
     return x if x >= 0
     -x
   end

>> abs2( 23 )
=> 23
>> abs2( -23 )
=> 23
```

Remarques et explications pour l'exemple Ruby 17 :

- Une définition de méthode est introduite par le mot-clé `def`, suivi du nom de la méthode, suivi du nom des paramètres, suivi d'une ou plusieurs instructions, suivi du mot-clé `end`.
- Ruby étant un langage à typage dynamique, il n'est pas nécessaire — ni possible — de spécifier le type des paramètres ou du résultat d'une méthode.
- Une méthode peut être définie de façon plus «compacte» — bien que cela ne soit pas recommandée — en séparant les éléments par «`;`». L'exemple qui suit illustre donc qu'en Ruby, le «`;`» sert de *séparateur* (et non de *terminateur*, comme en C ou Java) :

```
def add( x, y ); x + y; end
```

- Dans un `if`, le `then` n'est requis que si on utilise le `if` comme expression sur une seule ligne — les règles de style de Ruby veulent qu'on omette le `then` si le `if` est écrit sur plusieurs lignes.
- Le résultat retourné par une méthode est la dernière expression évaluée par la méthode. Les méthodes `add` suivantes sont donc équivalentes — mais la première est considérée d'un «meilleur style» :

```
def add( x, y )
  x + y
end

def add( x, y )
  return x + y
end
```

  Le style `Ruby` veut que `return` soit utilisée seulement pour retourner un résultat «au milieu» d'une série d'instructions, i.e., avant la dernière instruction ou expression d'une méthode. La méthode `abs2` illustre une telle utilisation.

- **Remarque importante :** Comme l'illustre l'un des exemples, les parenthèses pour un appel de méthodes peuvent être omises. Par contre, si ces parenthèses sont présentes, alors *il ne doit pas y a avoir d'espaces entre le nom de la méthode et les parenthèses*, car dans ce dernier cas, les parenthèses pourraient indiquer le début d'une expression complexe :

```
add( 2, 3 )       # OK ☺
add 2, 3          # OK ☺
add ( 2, 3 )      # Pas OK ☹
add (1+1), (2+1)  # OK ☺
```

Exemple Ruby 18 Appels de méthodes et envois de messages.

```
>? # Un operateur est une methode.
?> 2 + 3
=> 5

>> 2.+( 3 )
=> 5

>> 2.+ 3
=> 5

>? # Un appel de methode est un envoi de message.
>? 2.+( 3 )
=> 5

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

Remarques et explications pour l'exemple Ruby 18 :

- Les opérateurs (dits «infixes») pour les expressions sont des méthodes comme les autres. Une expression telle que «`2 + 3`» est donc un appel à la méthode «`+`» de l'objet «`2`» avec l'objet «`3`» comme argument.
- Un appel de méthode avec des arguments correspond *à l'envoi d'un message* à un objet avec ces arguments.

10 Structures de contrôle

Exemple Ruby 19 Structures de contrôles: `if`.

```
>> # Instruction conditionnelle classique.
   def div( x, y )
     if y == 0
       raise "Oops! Division par zero :("
     else
       x / y
     end
   end

>> div( 12, 3 )
=> 4
>> div( 12, 0 )
RuntimeError: Oops! Division par zero :(
     from (irb):4:in 'div'
     [...]
     from /home/tremblay/.rvm/rubies/jruby-1.7.16.1/bin/irb:13:in '(root)'
```

```
>> # Garde (condition) if associee a une instruction.
   def div( x, y )
     raise "Oops! Division par zero :(" if y == 0

     x / y
   end

>> div( 12, 3 )
=> 4
```

Remarques et explications pour l'exemple Ruby 19 :

- Dans une instruction `if`, bien que le mot-clé `then` puisse être indiqué, les règles de style de Ruby veulent qu'on l'omette.
- Une instruction `raise` lance une exception, de type `RuntimeError`.

- Tel qu'indiqué précédemment, le caractère «;» est utilisé seulement pour **séparer** des instructions apparaissant sur une même ligne. Contrairement aux langage C et Java, le «;» ne sert donc pas à terminer une instruction — un saut de ligne suffit.
- Lorsqu'une instruction simple doit être exécutée de façon conditionnelle, il est considéré de bon style d'utiliser une garde, donc :

```
instruction_simple if condition

  ... plutôt que ...

if condition
  instruction_simple
end
```

  Dans un tel cas, il est aussi suggéré de toujours utiliser une condition positive, si nécessaire en utilisant `unless`. Exemple : on veut retourner le premier élément d'un tableau `a` si un tel élément existe :

```
return a[0] unless a.empty?
  ... plutôt que ...
return a[0] if !a.empty?
```

Exemple Ruby 20 Structures de contrôles: `while`.

```
?> # Instruction while.
   def pgcd( a, b )
     # On doit avoir a <= b.
     return pgcd( b, a ) if a > b

     while b > 0
       a, b = b, a % b
     end

     a
   end

>> pgcd( 12, 8 )
=> 4
>> pgcd( 80, 120 )
=> 40
```

Remarques et explications pour l'exemple Ruby 20 :

- Une instruction `while` s'exécute tant que la condition indiquée reste vraie (i.e., non `false`, non `nil`).
- On parle d'une *affectation parallèle* lorsque, du coté gauche de l'opérateur d'affectation, on retrouve plusieurs variables séparées par des «,».

  Voici comment, en Ruby, on peut interchanger le contenu de `x` et `y` sans utiliser de variable temporaire :

  ```
  x, y = y, x
  ```

  On peut aussi utiliser de telles affectations pour «déconstruire» un tableau :

  ```
  x, y, z = [10, 20, 30]
  # x == 10 && y == 20 && z == 30

  x, y = [10, 20, 30]
  # x == 10 && y == 20

  x, *y = [10, 20, 30]
  # x == 10 && y == [20, 30]
  ```

Exemple Ruby 21 Structures de contrôles : Itération sur les index avec `for` et `each_index`.

```
?> # Instruction for
   def somme( a )
     total = 0
     for i in 0...a.size
       total += a[i]
     end

     total
   end

>> somme( [10, 20, 30] )
=> 60



?> # Iterateur each_index.
   def somme( a )
     total = 0
     a.each_index do |i|
       total += a[i]
     end

     total
   end

>> somme( [10, 20, 30] )
=> 60
```

Exemple Ruby 22 Structures de contrôles : Itération sur les éléments avec `for` et `each`.

```
?> # Instruction for (bis)
   def somme( a )
     total = 0
     for x in a
       total += x
     end

     total
   end

>> somme( [10, 20, 30] )
=> 60



?> # Iterateur each.
   def somme( a )
     total = 0
     a.each do |x|
       total += x
     end

     total
   end

>> somme( [10, 20, 30] )
=> 60
```

Remarques et explications pour les exemples Ruby 21–22 :

- Une expression telle que `0..n` est un `Range` (intervalle) qui génère les valeurs 0, 1, . . . , `n`. Par contre, un `Range` tel que `0...n` génère les valeurs 0, 1, . . . , `n-1`. On parle donc de `Range` *inclusif* (..) vs. *exclusif* (...) — voir plus haut.

  Une boucle telle que «`for i in 0...a.size`» permet donc traiter tous les indices valides de `a`, donc 0, 1, . . . , `a.size-1`.

- En Ruby, l'utilisation de la boucle `for` *est fortement déconseillée* — son utilisation est considérée comme «de mauvais style» — et ce pour deux raisons :

  - La variable d'itération n'est pas strictement locale à la boucle, donc sa valeur est modifiée par la boucle si elle existait déjà.
  - Une boucle `for` est mise en oeuvre par un `each`, donc est moins efficace — son utilisation ajoute un niveau additionnel d'indirection.

- La méthode `each` est la méthode générale et universelle pour l'itération : tous les objets composites — les collections — définissent (ou à tout le moins devraient définir) une méthode `each` qui permet de parcourir les éléments de la collection. On verra des exemples ultérieurement.

  Dans l'exemple 21, on utilise tout d'abord `each_index`, qui génére les différents indices de `a`. Dans ce cas, cela produit une solution semblable à celle du `for`.

  Par contre, dans l'exemple 22, on utilise plutôt `each`, qui génère directement les différents éléments de `a`. C'est cette dernière solution qui est la solution typique — plus «idiomatique» — en Ruby.

11 Paramètres des méthodes

Exemple Ruby 23 Paramètres des méthodes : valeur par défaut et nombre variable d'arguments.

```
?> # Argument optionnel et valeur par defaut.
   def foo( x, y = 40 )
     x + y
   end

>> foo( 3, 8 )
=> 11

>> foo( 3 )
=> 43
```

```
>> # Nombre variable d'arguments.
   def bar( x, *args, y )
     "bar( #{x}, #{args}, #{y} )"
   end

>> bar( 1, 2, 3, 4, 5 )
=> "bar( 1, [2, 3, 4], 5 )"

>> bar( 1, 2 )
=> "bar( 1, [], 2 )"

>> bar( 23 )
ArgumentError: wrong number of arguments (1 for 2+)
          from (irb):24:in 'bar'
          from (irb):27
          from /home/tremblay/.rvm/rubies/ruby-2.1.4/bin/irb:11:in '<main>'
```

Remarques et explications pour l'exemple Ruby 23 :

- Un ou des argument d'une méthode peuvent être omis si on spécifie une valeur par défaut. Par contre, ces arguments optionnels doivent apparaître *après* les arguments obligatoires.

- Il est possible de transmettre un nombre variable d'arguments, ce qu'on indique en préfixant le nom du paramètre avec «*». Dans la méthode, le nom du paramètre, sans le «*», est alors un *tableau* formé des arguments reçus.

Exemple Ruby 24 Paramètres des méthodes : arguments par mots-clés (*keyword arguments*).

```
>> # Arguments par mot-cles (keyword arguments).
   def diviser( numerateur:, denominateur: 1 )
     numerateur / denominateur
   end

>> diviser numerateur: 12, denominateur: 3
=> 4

>> diviser denominateur: 3, numerateur: 12
=> 4

>> diviser numerateur: 12
=> 12

>> diviser 10
ArgumentError: missing keyword: numerateur
        from (irb):31
        from /home/tremblay/.rvm/rubies/ruby-2.1.4/bin/irb:11:in '<main>'
```

Remarques et explications pour l'exemple Ruby 24 :

- Depuis Ruby 2.0,[6] on peut définir des méthodes où les paramètres et arguments sont définis par des mots-clés — *keyword arguments* — donc semblables à ce qu'on retrouve en Smalltalk [Gol89].
- Les paramètres par mot-clés permettent de spécifier les arguments lors d'un appel de méthode *sans avoir à respecter un ordre strict quant à la position des arguments* — voir les deux premiers appels à `diviser`.
- On peut, ou non, spécifier une valeur par défaut pour les arguments par mots-clés.

[6] On peut aussi définir de telles méthodes en Ruby 1.9, mais pour ce faire il faut manipuler explicitement un `hash`.

```
?> # Argument par mot-cle.
   def premier_index( a, x, res_si_absent: nil )
     a.each_index do |i|
       return i if a[i] == x
     end
     res_si_absent
   end

>> premier_index( [10, 20, 10, 20], 10 )
=> 0
>> premier_index( [10, 20, 10, 20], 88 )
=> nil
>> premier_index( [10, 20, 10, 20], 88, res_si_absent: -1 )
=> -1
```

- De tels paramètres sont souvent utiles pour les arguments optionnels, par exemple, `res_si_absent`. La présence du mot-clé rend alors l'appel plus clair quant au rôle de l'argument additionnel.

  Ainsi, la méthode `premier_index` aurait pu être définie et utilisée comme suit, mais le rôle de l'argument additionel aurait été moins clair :

  ```
  def premier_index( a, x, res_si_absent = nil )
    ...
  end

  premier_index( [10, 20, 10, 20], 88, -1 )
  ```

Soit la méthode suivante :

```ruby
def foo( x, y, z = nil )
  return x + y * z if z

  x * y
end
```

Indiquez ce qui sera affiché par chacun des appels suivants :

```ruby
# a.
puts foo( 2, 3 )

# b.
puts foo 2, 3, 5

# c.
puts foo( "ab", "cd", 3 )

# d.
puts foo( "ab", "cd" )
```

Exercice 1: Définition et utilisation de méthodes diverses avec plusieurs sortes d'arguments.

Soit la méthode suivante :

```
def bar( v = 0, *xs )
  m = v
  xs.each do |x|
    m = [m, x].max
  end
  m
end
```

Indiquez ce qui sera affiché par chacun des appels suivants :

```
# a.
puts bar

# b.
puts bar( 123 )

# c.
puts bar( 0, 10, 20, 99, 12 )
```

Exercice 2: Définition et utilisation de méthodes diverses avec plusieurs sortes d'arguments.

Soit le segment de code suivant :

```
def foo( x, *y, z = 10 )
  x + y.size + z
end

puts foo( 10, 20, 30 )
```

Qu'est-ce qui sera affiché?

Exercice 3: Définition d'une méthode avec plusieurs sortes d'arguments.

12 Définitions de classes

Exemple Ruby 25 Un script avec une classe (simple) pour des cours.

```
$ cat cours.rb
# Definition d'une classe (simple!) pour des cours.
class Cours
  attr_reader :sigle

  def initialize( sigle, titre, *prealables )
    @sigle, @titre, @prealables = sigle, titre, prealables
  end

  def to_s
    sigles_prealables = " "
    @prealables.each do |c|
      sigles_prealables << "#{c.sigle} "
    end

    "< #{@sigle} '#{@titre}' (#{sigles_prealables}) >"
  end
end
```

Exemple Ruby 25 Un script avec une classe (simple) pour des cours (suite).

```
if $0 == __FILE__
  # Definition de quelques cours.
  inf1120 = Cours.new( :INF1120, 'Programmation I' )
  inf1130 = Cours.new( :INF1130, 'Maths pour informaticien' )
  inf2120 = Cours.new( :INF2120, 'Programmation II',
                                 inf1120 )
  inf3105 = Cours.new( :INF3105, 'Str. de don.',
                                 inf1130, inf2120 )

  puts inf1120
  puts inf3105
  puts inf1120.sigle
  puts inf1120.titre
end
```

Exemple Ruby 26 Appel du script avec une classe pour des cours.

```
$ ruby cours.rb
< INF1120 'Programmation I' ( ) >
< INF3105 'Str. de don.' ( INF1130 INF2120 ) >
INF1120
NoMethodError: undefined method `titre' for #<Cours:0x13969fbe>
  (root) at cours.rb:34
```

Remarques et explications pour les exemples Ruby 25–26 :

- Une définition de classe est introduite par le mot-clé `class`, suivi du nom de la classe, suivi des attributs et méthodes, suivi de `end`.
- En Ruby, la convention pour nommer les identificateurs est la suivante :
  - On utilise le `snake_case` pour les variables locales et les paramètres, ainsi que les noms de méthodes — e.g., `initialize`, `to_s`, `prealables`, `sigles_prealables`, etc.
  - On utilise le `CamelCase`, avec une majuscule comme première lettre, pour les noms de classe — e.g., `Cours`, `Array`.
  - On utilise uniquement des majuscules avec tirets — *Screaming snake case* — pour les constantes, e.g., `STDOUT`, `STDERR`, etc.
- Un nom de variable débutant par «`@`» dénote une *variable d'instance* — un attribut de l'objet. Dans l'exemple, un objet de la classe `Cours` possède donc trois variables d'instance — trois attributs, toujours *privés* : `@sigle`, `@titre` et `@prealables`.
- On crée un nouvel objet à l'aide de la méthode `new`, méthode que, généralement, on ne définit pas. C'est plutôt la méthode `new` par défaut (`Object`) qui appelle la méthode `initialize` pour définir l'état initial de l'objet — pour initialiser les variables d'instance.
- Par défaut, toutes les *méthodes* sont publiques.
- Il n'est possible d'accèder à un attribut d'un objet que si une méthode appropriée, *publique*, a été définie.

- Une «*déclaration*» telle que «`attr_reader :sigle`» définit un attribut accessible en lecture [7], en définissant une méthode d'accès appropriée. Une telle déclaration est donc équivalente à la méthode suivante (définition implicite) :

```
def sigle
  @sigle
end
```

  Note : En fait, «`attr_reader :sigle`» représente un appel à la méthode `attr_reader` avec l'argument `:sigle`. Voir plus loin (section D).

- On peut indiquer qu'un segment de code ne doit être exécuté que si le script est appelé directement comme programme, donc ne doit pas être exécuté si le fichier est utilisé/chargé par un autre fichier. Pour ce faire, il s'agit d'utiliser la condition «`$0 == __FILE__`» :
  - `$0` = Nom du programme principal, i.e., nom du fichier appelé comme argument direct de `ruby`.
  - `__FILE__` = Nom du fichier courant, donc contenant le code où apparait la variable `__FILE__`.
  - `$0 == __FILE__` : Pour notre exemple, cette condition sera vraie seulement pour un appel «`$ ruby cours.rb`». Si on exécute plutôt un autre programme/script qui utilise `cours.rb`, seule la classe `Cours` sera définie.

- L'appel «`puts inf1120.sigle`» produit un résultat correct parce que `sigle` est bien une méthode publique. Par contre, l'appel «`puts 1120.titre`» n'est pas permis puisqu'aucune méthode nommée `titre` n'a été définie — ni explicitement (avec `def`), ni implicitement (avec `attr_reader`).

- La méthode `to_s` est utilisée pour obtenir une chaîne de caractères représentant l'objet. Cette méthode est donc équivalente au `toString` de Java.

- Étant donnée une collection — ici, `@prealables` — on peut obtenir et traiter les différents éléments de cette collection à l'aide de l'itérateur `each` — voir plus loin.

Pour la classe `Cours`, définissez une méthode qui permet d'obtenir le titre d'un cours et une autre méthode qui permet de **modifier** le titre d'un cours.

Utilisez ensuite cette dernière méthode pour changer le titre du cours `inf1120` en `"Programmation Java I"`.

Exercice 4: Méthodes pour lire et modifier le titre d'un cours.

[7] Un *getter* dans la terminologie Java.

13 Lambda-expressions

Exemple Ruby 27 Les lambda-expressions : type et méthodes de base.

```
>> # Une lambda-expression represente un objet,
   # de classe Proc, qu'on peut 'appeler'.
   # Un Proc est donc une  "fonction anonyme".
?> lambda { 0 }.call
=> 0


>> zero = lambda { 0 }
=> #<Proc:0x5c5eefef@(irb):2 (lambda)>

>> zero.class
=> Proc

>> zero.arity    # Lambda avec 0 argument!
=> 0
>> zero.parameters
=> []

>> zero.call
=> 0
```

```
?> # Une lambda-expression peut avoir des arguments.
?> inc = lambda { |x| x + 1 }
=> #<Proc:0x16293aa2@(irb):8 (lambda)>

>> inc.arity
=> 1

>> inc.parameters
=> [[:req, :x]]

>> inc.call( 3 )
=> 4

# Et le nombre d'arguments est verifie!
>> inc.call
ArgumentError: wrong number of arguments (0 for 1)
                from [...]
>> inc.call( 10, 20 )
ArgumentError: wrong number of arguments (2 for 1)
                from [...]
```

```
?> double = lambda do |y|
?>   y + y
>> end
=> #<Proc:0x5158b42f@(irb):11 (lambda)>

>> double.arity
=> 1

>> double.call( 3 )
=> 6
```

Remarques et explications pour l'exemple Ruby 27 :

- Une lambda-expression peut être vue comme une «*fonction anonyme*» — une

fonction ou méthode sans nom. On peut affecter une telle expression à une variable, qui réfère alors à cette fonction.

- Une lambda-expression peut aussi être vue comme une expression dont on retarde l'exécution, expression qui ne sera évaluée qu'au moment où on fera un appel explicite à `call`.

- Une lambda-expression est dénotée par le mot clé `lambda` suivi d'un *bloc*. Le style Ruby veut qu'on utilise l'une de deux notations possibles pour les blocs :
  - Si le bloc est court (une seule ligne), on utilise «`{...}`».
  - Si le bloc comporte plusieurs lignes, on utilise «`do...end`», sur des lignes distinctes.

  Règle générale, dans ce qui suit, nous respecterons ce style, **sauf parfois pour rendre plus compacte la mise en page du texte ou des diapositives**.

- Étant donné un objet de classe `Proc` qui dénote une lambda-expression, on peut déterminer :
  - Son `arity` = le nombre d'arguments que doit recevoir cette lambda-expression — le nombre d'arguments à fournir lors d'un appel à la méthode `call`.
  - Ses `parameters` = la liste des paramètres que doit recevoir cette lambda-expression — donc la liste des arguments à fournir lors d'un appel à la méthode `call`, avec le nom du paramètre et son mode (obligatoire, optionnel, etc.).

Exemple Ruby 28 Les lambda-expressions, comme n'importe quel autre objet, peuvent être transmises en argument.

```
>> # Une methode pour executer deux fois du code (sans arg.).
   def deux_fois( f )
     f.call
     f.call
   end


>> deux_fois( lambda { print 'Bonne '; print 'journee!\n' } )
Bonne journee!
Bonne journee!
=> nil


>> deux_fois lambda { print 'Bonne '; print 'journee!\n' }
Bonne journee!
Bonne journee!
=> nil


?> # Ici, les () sont obligatoires, sinon erreur de syntaxe...
?>   deux_fois( lambda do
                  print 'Bonne '
                  print 'journee!\n'
                end )
Bonne journee!
Bonne journee!
=> nil
```

Remarques et explications pour l'exemple Ruby 28 :

- Une lambda-expression, comme n'importe quel objet, peut être transmise en argument.
- Le méthode `deux_fois` permet d'exécuter deux fois un bout de code, code représenté par une lambda-expression.
- Si le bout de code est complexe — composé de plusieurs lignes — il est préférable d'utiliser `do... end` plutôt que des accolades. Toutefois, dans notre exemple, les parenthèses deviennent obligatoires (sinon erreur de syntaxe), ce qui rend plus difficile la lecture du code.

- Dans plusieurs cas, on retrouve une structure semblable à celle de cet exemple, à savoir, une méthode qui exécute *un unique bloc de code reçu en dernier argument*. Bien que cela puisse faire avec des `lambda`s, comme l'illustre l'exemple, Ruby rend cela encore plus facile avec *les blocs*, qu'on verra à la prochaine section.

Exemple Ruby 29 Les lambda-expressions, comme n'importe quel objet, peuvent être retournées comme résultat d'une fonction.

```
?> # Une lambda-expression peut etre retournee comme resultat.
?> def plus_x( x )
     lambda { |y| x + y }
   end

>> plus_x(3).call(12)
=> 15

?> plus_bis = lambda { |a| lambda { |b| a + b } }
=> #<Proc:0x2d7275fc@(irb):44 (lambda)>

>> plus_bis.call(3).call(12)
=> 15
```

Remarques et explications pour l'exemple Ruby 29 :

- Une lambda-expression étant un objet comme n'importe quel autre, elle peut être retournée comme résultat d'une méthode, ou même d'une autre lambda-expression.

Exemple Ruby 30 Les lambda-expressions capture les variables non-locales.

```
?> # Une lambda-expression 'capture' les variables
   # non-locales requises dans le bloc.
?> x = 23
=> 23

>> plus_x = lambda { |y| x + y }
=> #<Proc:0x72d1ad2e@(irb):31 (lambda)>

>> plus_x.call(7)
=> 30

>> x = 999
=> 999

>> plus_x.call 2
=> 1001
```

Remarques et explications pour l'exemple Ruby 30 :

- Une lambda-expression peut utiliser des variables *non-locales* — ici, `x`. On dit alors que la lambda-expression *capture* ces variables de façon à définir son environnement d'exécution.

Pour la classe `Cours` :

a. Définissez une méthode `prealables` qui reçoit en argument un `predicat` — une lambda-expression — et qui retourne la liste des prélables du cours qui satisfont ce `predicat`.

b. Utilisez la méthode `prealables` pour obtenir les prélables du cours `inf3105` dont le sigle contient la chaine `"INF"`.

   Remarque : Pour ce dernier point, vous devez utiliser une expression de *pattern-matching*. En Ruby, l'expression suivante retourne un résultat *non nil* si `x`, une chaine, matche le motif `INF` :

   ```
   /INF/ =~ x
   ```

   Plus précisément, l'expression retourne `nil` si le motif n'apparait pas dans la chaine, sinon elle retourne la position du premier *match*.

Exercice 5: Une méthode pour identifier un sous-ensemble de préalables d'un cours.

Exemple Ruby 31 Les appels à une lambda-expression peuvent aussi être faits avec «`.()`» plutôt qu'avec `call` — mais c'est rarement utilisé!

```
>> lambda { 0 }.()
=> 0


>> zero = lambda { 0 }
=> #<Proc:0x5c5eefef@(irb):2 (lambda)>

>> zero.()
=> 0

>> inc = lambda { |x| x + 1 }
=> #<Proc:0x16293aa2@(irb):8 (lambda)>

>> inc.( 3 )
=> 4
```

14 Blocs

Un **bloc** est un **segment de code** entre accolades {...} ou entre `do`...`end` :

```
a.each { |x| total += x }

a.each_index do |i|
  total += a[i]
end


inc = lambda { |x| x + 1 }

double = lambda do |y|
  y + y
end
```

L'utilisation des blocs en Ruby est étroitement liée à l'instruction `yield`. Voici tout d'abord quelques définitions du verbe anglais «*to yield*» :

- *to produce (something) as a result of time, effort, or work*
- *to surrender or relinquish to the physical control of another : hand over possession of*
- *to surrender or submit (oneself) to another*

Quelques traductions françaises possibles du verbe «*to yield*» sont «céder» ou «*produire*».

L'instruction `yield`, lorsqu'exécutée dans une méthode, a l'effet suivant :

- elle évalue (exécute) **le** bloc passé en argument à la méthode

  Note : ce bloc peut ne pas apparaitre dans la liste des arguments — argument implicite

Exemple Ruby 32 Une méthode pour exécuter deux fois un bout de code — avec un *bloc*.

```
>> # Une autre methode pour executer deux_fois du code, avec bloc!
   def deux_fois
     yield
     yield
   end


>> deux_fois { print 'Bonne '; print 'journee!\n' }
Bonne journee!
Bonne journee!
=> nil


>> deux_fois do
     print 'Bonne '
     print 'journee!\n'
   end
Bonne journee!
Bonne journee!
=> nil


>> deux_fois
LocalJumpError: no block given (yield)
         from (irb):1:in 'deux_fois'
         from (irb):3
         from /home/tremblay/.rvm/rubies/ruby-2.1.4/bin/irb:11:in '<main>'
```

```
>> # Methode pour executer k fois du code.
   def k_fois( k )
     k.times do
       yield
     end
   end

>> k_fois( 3 ) do
     print 'Bonne '
     print 'journee!\n'
   end
Bonne journee!
Bonne journee!
Bonne journee!
```

Remarques et explications pour l'exemple Ruby 32 :

- Un bloc est indiqué par un bout de code entre `{...}` (lorsque sur la même ligne) ou `do... end` (lorque sur plusieurs lignes).
- Toute méthode, en plus des arguments explicites, peut recevoir un *bloc* comme *argument implicite*. Ce bloc–argument étant implicite, il n'a pas besoin d'être indiqué dans la liste des arguments — bien qu'il puisse l'être : voir plus loin.
- On peut exécuter le bloc passé en argument implicite en appelant la méthode `yield`. Donc : `yield` $\approx$ *le_bloc_passé_en_argument*`.call`.

Exemple Ruby 33 Une méthode pour évaluer une expression — avec `lambda`, avec *bloc implicite* et avec *bloc explicite*.

```
>> # Methode pour evaluer une expression: avec lambda.
   def evaluer( x, y, expr )
     expr.call( x, y )
   end

>> evaluer( 10, 20, lambda { |v1, v2| v1 + v2 } )
=> 30


>> # Methode pour evaluer une expression: avec bloc implicite.
   def evaluer( x, y )
     yield( x, y )
   end

>> evaluer( 10, 20 ) { |a, b| a * b }
=> 200

>> # Methode pour evaluer une expression: avec bloc explicite.
   def evaluer( x, y, &expr )
     expr.call( x, y )
   end

>> evaluer( 10, 20 ) { |x, y| y / x }
=> 2
```

```
>> # On peut verifier si un bloc a ete passe ou non.
   def evaluer( x, y )
     return 0 unless block_given?
     yield( x, y )
   end

>> evaluer( 10, 20 ) { |x, y| y / x }
=> 2
>> evaluer( 10, 20 )
=> 0
```

```
>> def foo( &b )
     [b.class, b.arity, b.parameters] if block_given?
   end
=> :foo

>> foo
=> nil

>> foo { 2 }
=> [Proc, 0, []]

>> foo { |x| x + 1 }
=> [Proc, 1, [[:opt, :x]]]
```

Remarques et explications pour l'exemple Ruby 33 :

- De la même façon qu'un `lambda` peut recevoir des arguments, un bloc peut aussi en recevoir.
- Si le dernier paramètre de l'en-tête d'une méthode est préfixé par «`&`», alors le bloc transmis à l'appel de la méthode sera associé à cet argument, donc le bloc devient *explicite*. Dans ce cas, c'est comme si le bloc avait été transformé en un `lambda` et associé au paramètre ; on peut donc l'appeler explicitement avec `call`, mais aussi l'exécuter avec `yield`.
- La méthode `block_given?` permet de déterminer si un bloc a été transmis (en dernier argument implicite). Si c'est le cas, on peut alors évaluer le bloc avec `yield`.

Même question que la précédente, mais cette fois en utilisant un bloc implicite pour le prédicat plutôt qu'une lambda-expression.

Exercice 6: Une méthode pour identifier un sous-ensemble de préalables d'un cours.

15 Portée des variables

> *sigil* (Ésotérisme) Symbole graphique ou sceau représentant une intention ou un être magique.
>
> Source : `https://fr.wiktionary.org/wiki/sigil`

Ruby utilise un certain nombre de *sigils* pour indiquer la portée des variables — pour indiquer dans quelles parties du programme une variable est connue et accessible :

`foo`	variable locale (à une méthode ou un bloc)
`@foo`	variable d'instance (attribut d'un objet)
`@@foo`	variable de classe (attribut d'une classe)
`$foo`	variable globale (accessible partout)

> *Matz [the designer of Ruby] stated more than one time that sigils for globals and for instance variables are there to remind you that you should not use them directly. You should encode the global information in a class or a constant, and access the instance variables via accessor methods. When you're writing quick & dirty code you can use them, but globals are evil and the sigils are there to reify a code smell.*
>
> *Source : http://c2.com/cgi/wiki?TheProblemWithSigils*

Exemple Ruby 34 Illustration de la vie et portée des variables.

```
>> # Une definition de methode ne voit pas
   # les variables non-locales.
?> x = 22
=> 22

>> def set_x
     x = 88
   end
=> :set_x

>> set_x
=> 88

>> x      # Inchangee!
=> 22
```

```
?> # Un bloc capture les variables non-locales
   # si elles existent.
?> def executer_bloc
     yield
   end
=> :executer_bloc

>> x = 44
=> 44

>> executer_bloc { x = 55 }
=> 55

>> x     # Modifiee!
=> 55
```

```
?> # Si la variable n'existe pas deja,
   # alors est strictement locale au bloc.
?> z
NameError: undefined local variable or method 'z' for main:Object
[...]

?> executer_bloc { z = 88 }
=> 88

>> z
NameError: undefined local variable or method 'z' for main:Object
[...]
```

```
>> # Une variable globale est accessible partout!
?> $x_glob = 99
=> 99

>> def set_x_glob
     $x_glob = "abc"
   end
=> :set_x_glob

>> set_x_glob
=> "abc"

>> $x_glob
=> "abc"


>> lambda { $x_glob = [10, 20] }.call
=> [10, 20]

>> $x_glob
=> [10, 20]
```

```
>> # Une variable locale est accessible dans l'ensemble
   # de la methode.
?> def foo( x )
     if x <= 0 then a = 1 else b = "BAR" end
     [a, b]
   end
=> :foo

>> foo( 0 )
=> [1, nil]

>> foo( 99 )
=> [nil, "BAR"]
```

```
>> # Mais un bloc definit une nouvelle portee, avec des variables
   # strictement locales!
?> def bar( *args )
     args.each do |x|
       r = 10
       puts x * r
     end
     r
   end
=> :bar

>> bar( 10, 20 )
100
200
NameError: undefined local variable or method 'r' for main:Object
           [...]
```

Remarques et explications pour l'exemple Ruby 34 :

- Les variables non-locales sont pas visibles à l'intérieur d'une définition de méthode — évidemment, les variables d'instance (avec préfixe «`@`») le sont. Dans la méthode `set_x`, le `x` auquel on affecte 88 est strictement local à la méthode et donc le `x` initial est inchangé.
- Par contre, les variables non-locales sont visibles dans un bloc, et donc un bloc *capture* les variables non-locales. L'exécution d'un bloc se fait donc *dans le contexte dans lequel le bloc a été créé*. Dans notre exemple, le bloc modifie la variable `x` et cette variable existait avant l'appel : c'est donc ce `x` qui est utilisé, et la variable `x` non-locale est modifiée.
- Si un bloc introduit une variable locale, i.e., cette variable n'a pas été capturée par le bloc, alors cette variable est strictement locale au bloc — comme pour une méthode. Donc, dans le dernier exemple, puisque `z` n'existait pas avant l'appel, le `z` affecté est local au bloc, d'où l'erreur lors de l'utilisation de `z` après l'exécution du bloc.
- Les branches d'un `if` ne définissent pas une nouvelle portée. Une variable est considérée comme étant existante même si le code la définissant n'est pas exécuté — avec une valeur par défaut de `nil` si non initialisé, ce qui n'est pas la même chose qu'une variable non existante qui soulève une exception si on tente de l'utiliser.

16 Modules

Modules are a way of grouping together methods, classes, and constants. Modules give you two major benefits:

1. *Modules provide a namespace and prevent name clashes.*
2. *Modules implement the mixin facility.*

Source : http://ruby-doc.com/docs/ProgrammingRuby/html/tut_modules.html

Exemple Ruby 35 Les modules comme *espaces de noms*.

```
module M1
  C1 = 0
end

module M2
  C1 = 'abc'
end

M1::C1 != M2::C1    # => true
```

Remarques et explications pour l'exemple Ruby 35 :

- Un `module` Ruby, tout comme un `package` en Java, permet une forme d'encapsulation en permettant de définir des espaces de noms indépendants — des *namespaces* distincts. Deux modules peuvent donc définir des constantes ou des méthodes avec les mêmes noms sans qu'il n'y ait de conflit.
- Pour accéder à une constante définie dans un module, on utilise la notation «`NomModule::NOM_CONSTANTE`».

```ruby
module Module1
  def self.zero
    0
  end

  def un
    1
  end

  def val_x
    @x
  end

  def inc_inc( y )
    inc( y )
    inc( y )
  end
end

class C1
  include Module1

  def initialize( x )
    @x = x
  end

  def inc( y )
    @x += y
  end
end

class C2
  include Module1
end
```

Exemple Ruby 36 Un module *mixin* `Module1` et son utilisation.

```
>> # Appel sur le module de la methode de classe.
?> Module1.zero
=> 0

>> # Appel sur le module de la methode d'instance.
?> Module1.un
NoMethodError: undefined method 'un' for Module1:Module
    ...

>> # Appel sur un objet C1 des methodes
?> # de classe et d'instance du module.
?> c1 = C1.new( 99 )
=> #<C1:0x12cf7ab @x=99>

>> c1.zero
NoMethodError: undefined method 'zero' for #<C1:0x12cf7ab @x=99>
    ...
>> c1.un
=> 1
>> c1.val_x
=> 99
>> c1.inc_inc( 100 )
=> 299

>> # Appel sur un objet C2 des methodes
?> # de classe et d'instance du module.
?> c2 = C2.new
=> #<C2:0x1a8622>
>> c2.un
=> 1
>> c2.val_x
=> nil
>> c2.inc_inc( 100 )
NoMethodError: undefined method 'inc' for #<C2:0x1a8622>
    ...
```

Remarques et explications pour l'exemple Ruby 36 :

- Un `module` — comme une classe — peut définir des méthodes de classe et des méthodes d'instance.

- Une méthode de classe d'un module — comme pour une classe — est indiquée par le préfixe «`self.`» devant le nom de la méthode. On peut aussi indiquer le nom du module ou de la classe — par ex., «`def Module1.zero`» — mais le style Ruby suggère d'utiliser plutôt «`def self.zero`».

- Une méthode de classe peut être appelée avec un appel de la forme suivante :

  ```
  NomDuModule.nom_methode
  ```

  On peut aussi utiliser la forme suivante d'appel :

  ```
  NomDuModule::nom_methode
  ```

- Une méthode d'instance d'un module **ne peut être appelée que par l'intermédiaire d'un objet dont la définition de classe a inclus, avec `include`, le module.**

  La méthode d'instance est alors exécutée comme si elle avait été définie directement comme méthode d'instance de la classe ayant effectuée l'inclusion.

- Une méthode d'instance définie dans un module peut faire référence à des attributs ou méthodes **qui ne sont pas définies dans le module**. Évidemment, pour que ces références soient valides à l'exécution, les attributs ou méthodes en question doivent être définis dans la classe ayant effectué l'inclusion.

- Quelques définitions d'un *mixin* :

  > *A* **mixin** *provides an easy way to dynamically add the methods of an existing class to a custom class without using inheritance. You mix in the class, and then add that class's members to the prototype object of the custom class.*
  > *[...]*
  > *A* **mixin** *is an atomic unit in an object-oriented language that adds functionality to another class. Generally, mixins are not meant to be used on their own, just as you would not order a bowl of nuts at an ice cream stand. Instead, they provide a small piece of specialized functionality that you can easily add to a base class.*
  > *https://www.adobe.com/support/documentation/en/flex/1/mixin/mixin2.html*

In object-oriented programming languages, a **mixin** *is a class* **that contains methods for use by other classes without having to be the parent class of those other classes**. *How those other classes gain access to the mixin's methods depends on the language. Mixins are sometimes described as being "included" rather than " inherited".*

[. . .]

A mixin can also be viewed as an interface with implemented methods.

`https://en.wikipedia.org/wiki/Mixin`

17 Modules `Enumerable` et `Comparable`

17.1 Module `Enumerable`

La figure à la page 71 présente la liste des méthodes du module `Enumerable` — donc les diverses méthodes disponibles **lorsque la méthode `each` est définie par une classe et que le module `Enumerable` est inclus (avec `include`)!**

Des exemples illustrant ces diverses méthodes sont ensuite présentés.

Methods

::[]
::new
::try_convert
#&
#*
#+
#-
#<<
#<=>
#==
#[]
#[]=
#any?
#assoc
#at
#bsearch
#clear
#collect
#collect!
#combination
#compact
#compact!
#concat
#count
#cycle
#delete
#delete_at
#delete_if
#drop
#drop_while
#each
#each_index
#empty?
#eql?
#fetch
#fill
#find_index
#first
#flatten
#flatten!
#frozen?
#hash
#include?
#index
#initialize_copy
#insert
#inspect
#join
#keep_if
#last
#length
#map
#map!
#pack
#permutation
#pop
#product
#push
#rassoc
#reject
#reject!
#repeated_combination
#repeated_permutation
#replace
#reverse
#reverse!
#reverse_each
#rindex
#rotate
#rotate!
#sample
#select
#select!
#shift
#shuffle
#shuffle!
#size
#slice
#slice!
#sort
#sort!
#sort_by!
#take
#take_while
#to_a
#to_ary
#to_h
#to_s
#transpose
#uniq
#uniq!
#unshift
#values_at
#zip
#|

Exemple Ruby 37 Exemples d'utilisation du module `Enumerable`.

```
?>> # La classe Array definit la methode each et
    # inclut le module Enumerable.

>> a = [10, 20, 30, 40]
=> [10, 20, 30, 40]

?> # Appartenance d'un element.
?> a.include? 20
=> true

>> a.include? 999
=> false
```

```
?> # Application fonctionnelle.
?> a.map { |x| x + 2 }   # Synonyme = collect.
=> [12, 22, 32, 42]

>> a                     # a n'est pas modifie.
=> [10, 20, 30, 40]


?> # Application imperative (mutable)!
>> a.map! { |x| 10 * x }
=> [100, 200, 300, 400]

>> a                     # a est modifie!
=> [100, 200, 300, 400]
```

Remarques et explications pour l'exemple Ruby 37 :

- Toute classe qui, comme `Array`, définit une méthode `each` et *inclut* le module `Enumerable`, hérite automatiquement d'un grand nombre de méthodes :
  - `include?` : Permet de déterminer si un élément appartient à la collection.
  - `map` (appelé aussi `collect`) : Permet d'appliquer une fonction (un bloc) aux éléments de la collection pour produire une nouvelle collection.
    Note : Il existe aussi une variante `map!` qui *modifie* la collection.

```
?> # Selection/rejet d'elements selon un critere.
>> a.select { |x| x >= 300 }
=> [300, 400]

>> a.reject { |x| x >= 300 }
=> [100, 200]

>> a
=> [100, 200, 300, 400]

# Il existe aussi des variantes imperatives/mutables:
#   select!
#   reject!
```

```
?> # Obtention du premier element qui satisfait un critere.
>> a
=> [100, 200, 300, 400]

>> a.find { |x| x > 200 } # Synonyme = detect.
=> 300

>> a.find { |x| x < 0 }
=> nil
```

```
?> # Quantificateurs.
?> a.all? { |x| x > 0 }
=> true

>> a.any? { |x| x > 500 }
=> false
```

```
?> # Reduction avec un operateur binaire.
?> a.reduce { |x, y| x + y } # Synonyme = inject.
=> 1000

>> a.reduce( :+ )
=> 1000

>> a.reduce( &:+ )
=> 1000


>> a.reduce( :* )
=> 2400000000

>> a.reduce( 999, :+ )
=> 1999
```

```
?> # Autres exemples de reduction, avec operateurs divers.
>> a.reduce(0) { |max, x| x > max ? x : max }
=> 400

>> a.map { |x| x / 10 }
=> [10, 20, 30, 40]

>> a.reduce([]) { |a, x| a << x / 10 }
=> [10, 20, 30, 40]

>> a.reduce([]) { |ar, x| [x] + ar + [x] }
=> [400, 300, 200, 100, 100, 200, 300, 400]
```

```
?> # Autres exemples de reduction, avec operateurs divers.
>> a.reduce(0) { |max, x| x > max ? x : max }
=> 400

>> a.map { |x| x / 10 }
=> [10, 20, 30, 40]

>> a.reduce([]) { |a, x| a << x / 10 }
=> [10, 20, 30, 40]

>> a.reduce([]) { |ar, x| [x] + ar + [x] }
=> [400, 300, 200, 100, 100, 200, 300, 400]
```

```
?> # Regroupement, dans un Hash, des elements
   # avec une meme valeur specifiee par le bloc.
>> a.group_by { |x| x }
=> {100=>[100], 200=>[200], 300=>[300], 400=>[400]}


>> a.group_by { |x| x >= 222 }
=> {false=>[100, 200], true=>[300, 400]}


>> a.group_by { |x| x / 100 }
=> {1=>[100], 2=>[200], 3=>[300], 4=>[400]}


>> a.group_by { |x| x % 2 }
=> {0=>[100, 200, 300, 400]}


>> a.group_by { |x| (x / 100) % 2 }
=> {1=>[100, 300], 0=>[200, 400]}
```

– `select` et `reject` : Permettent de sélectionner ou rejeter les éléments de la collection qui répondent à un critère spécifié par un bloc pour produire une nouvelle collection.

  Note : Il existe aussi des variantes `select!` et `reject!` qui *modifient* directement la collection.
– `find` (appelé aussi `detect`) qui retourne le premier élément de la collection qui satisfait un certain critère (spécifié par un bloc).
– `all?` et `any?` : Quantificateurs universel (pour tout élément) et existentiel (il existe un élément) — le prédicat évalué pour chaque élément est défini par le bloc passé en argument.
– `reduce` (appelé aussi `inject`) : Combine les éléments de la collection en utilisant un opérateur binaire ou, dans sa forme la plus générale, un bloc qui reçoit deux (2) arguments :
  * Le premier argument est «l'accumulateur», et représente la valeur obtenue jusqu'à présent. Sa valeur initiale est (possiblement) définie par la valeur passée en argument à la méthode `reduce`.
  * Le deuxième argument est un élément de la collection ;

  L'exemple Ruby 38 présente une mise en oeuvre possible de `reduce`, qui aide à comprendre le rôle des deux arguments **du bloc** ainsi que de l'argument de `reduce` reçu comme valeur initiale.

  La réduction à l'aide d'un opérateur binaire simple (commutatif et associatif) — par exemple «`+`», «`*`» — étant un cas souvent rencontré, on peut aussi passer en argument (explicite) à `reduce` un symbole dénotant un tel opérateur, et ce avec ou sans valeur initiale.
– `group_by` : Produit un `Hash` où chaque clé est une valeur spécifiée par le critère associé au bloc et où la définition associée est l'ensemble des valeurs de la collection qui génèrent cette clé.

Voir p. 71 pour une liste plus détaillée des opérations du module `Enumerable`.

Donnez une mise en oeuvre, dans un style fonctionnel, de la méthode `to_s` de la classe `Cours` vue précédemment.

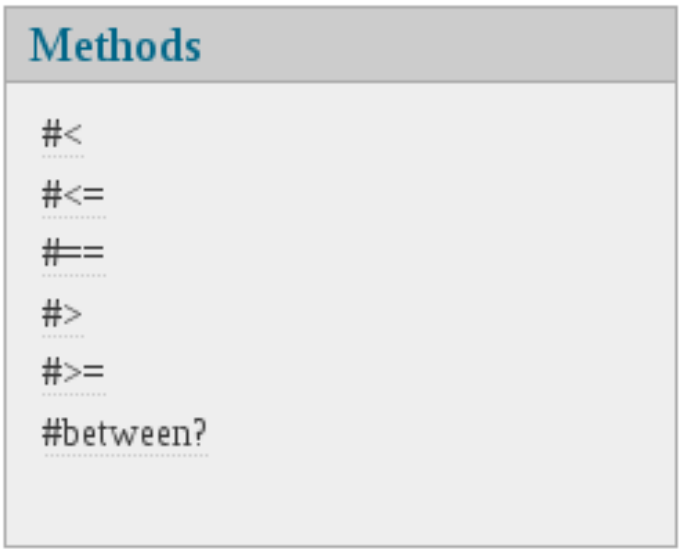
Exercice 7: Mise en oeuvre fonctionnelle de `Cours#to_s`.

Exemple Ruby 38 Une mise en oeuvre, en Ruby, de quelques méthodes du module `Enumerable`, méthodes qui utilisent la méthode `each` de la classe ayant exécuté l'appel «`include Enumerable`».

```ruby
# Mise en oeuvre possible, en Ruby, de quelques methodes
# du module Enumerable: on utilise *uniquement* each!

module Enumerable
  def include?( elem )
    each do |x|
      return true if x == elem
    end

    false
  end

  def find
    each do |x|
      return x if yield(x)
    end

    nil
  end

  def reduce( val_initiale )
    # Autre argument implicite = bloc recevant deux arguments.
    accum = val_initiale
    each do |x|
      accum = yield( accum, x )
    end

    accum
  end
end
```

17.2 Module `Comparable`

La figure ci-bas présente la liste des méthodes du module `Comparable`, c'est-à-dire, les diverses méthodes disponibles **lorsque la méthode `<=>` est définie par une classe et que le module `Comparable` est inclus (avec `include`)!**



Les exemples Ruby qui suivent présentent la méthode «`<=>`» et les méthodes définies par le module `Comparable`.

Exemple Ruby 39 Tris avec `Enumerable` et `<=>`.

```
>> # Comparaison avec l'operateur 'spaceship'.
?> 29 <=> 33
=> -1
>> 29 <=> 29
=> 0
>> 29 <=> 10
=> 1
```

```
>> # Tris.
?> a.sort
=> [10, 29, 33, 44]

>> a.sort { |x, y| x <=> y }
=> [10, 29, 33, 44]

>> a.sort { |x, y| -1 * (x <=> y) }
=> [44, 33, 29, 10]

>> a.sort { |x, y| (x % 10) <=> (y % 10) }
=> [10, 33, 44, 29]
```

Remarques et explications pour l'exemple Ruby 39 :

- L'opérateur de comparaison, dit opérateur «*spaceship*», doit retourner les valeurs suivantes pour un appel «`x <=> y`» :

-1	si `x` est *inférieur* à `y`
0	si `x` est *égal* à `y`
1	si `x` est *supérieur* à `y`

  Lorsque cet opérateur est correctement défini (-1, 0 et 1) et que le module `Comparable` est inclus, alors les autres opérateurs de comparaison sont alors **automatiquement définis** : `<`, `<=`, `>`, `>=`, `==`, `!=`.

- Une collection qui définit une méthode `each`, qui inclut le module `Enumerable` et dont les éléments peuvent être comparés avec l'opérateur «*spaceship*» hérite automatiquement d'une méthode `sort`, ainsi que des méthodes `min` et `max`.

  Par défaut, i.e., sans bloc après `sort` (idem pour `max` et `min`), la comparaison se fait avec l'opérateur «`<=>`». Par contre, on peut aussi spécifier une méthode de comparaison en transmettant un bloc, qui doit retourner -1, 0, 1 comme l'opérateur «`<=>`».

Remarques et explications pour l'exemple Ruby 40 :

- Lorsque la méthode «`<=>`» est définie par une classe et que cette classe inclut le module `Comparable`, diverses méthodes deviennent alors automatiquement disponibles.

Dans cet exemple, l'ordre entre deux `Cours` est déterminé par l'ordre entre leurs sigles. Donc, étant donné deux cours `c1` et `c2`, `c1 <=> c2 == c1.sigle <=> c2.sigle`.

- Puisque la classe `Cours` inclut le module `Comparable`, les méthodes de comparaison telles que «`<`», «`>=`», etc., deviennent automatiquement disponibles.

Exemple Ruby 40 Comparaison et tri de `Cours` via les sigles.

```
$ cat cours-bis.rb
require_relative 'cours'

class Cours
  include Comparable

  def <=>( autre )
    sigle <=> autre.sigle
  end
end

if $0 == __FILE__
  # Definition de quelques cours.
  inf1120 = Cours.new( :INF1120, 'Programmation I' )
  inf1130 = Cours.new( :INF1130, 'Maths pour informaticien' )
  inf2120 = Cours.new( :INF2120, 'Programmation II', inf1120 )
  inf3105 = Cours.new( :INF3105, 'Str. de don.', inf1130, inf2120 )

  cours = [ inf3105, inf1120, inf2120, inf1130 ]

  # Quelques expressions
  puts inf3105 < inf1120
  puts inf2120 >= inf1130
  cours.sort.each { |c| puts c }
end
-------------------------------------

$ ruby cours-bis.rb
false
true
< INF1120 'Programmation I' ( ) >
< INF1130 'Maths pour informaticien' ( ) >
< INF2120 'Programmation II' ( INF1120 ) >
< INF3105 'Str. de don.' ( INF1130 INF2120 ) >
```

Que fait la méthode suivante? Quel nom plus significatif pourrait-on lui donner?

```
class Array
  def mystere( p )
    reduce( [[], [], []] ) do |res, x|
      res[1 + (x <=> p)] << x
      res
    end
  end
end
```

Exercice 8: Méthode `mystere` sur un `Array`.

18 Itérateurs définis par le programmeur

Exemple Ruby 41 Une classe (simplifiée) pour des `Ensembles`.

```
class Ensemble
  include Enumerable

  # Ensemble initialement vide (sans element).
  def initialize
    @elements = []
  end

  # Ajout d'un element, sauf si deja present!
  def <<( x )
    @elements << x unless contient? x
    self
  end

  def each
    @elements.each do |x|
      yield( x )
    end
  end
```

```ruby
  def cardinalite
    count
  end

  def contient?( x )
    include? x
  end

  def somme( val_initiale = 0 )
    reduce(val_initiale) { |s, x| s + x }
  end

  def produit( val_initiale = 1 )
    reduce(val_initiale) { |s, x| s * x }
  end

  def to_s
    "{ " << map { |x| x.to_s }.join(", ") << " }"
  end
end
```

Remarques et explications pour l'exemple Ruby 41 :

- La classe `Ensemble` définit une classe pour des objets représentant des **ensembles** d'éléments, i.e., des collections où chaque élément n'est présent au plus qu'une seule fois.
- Un ensemble nouvellement créé — avec `Classe.new`, qui appelle `initialize` — est vide, i.e., son tableau d'`@elements` est vide.
- Un opérateur (infixe) tel que «`<<`» est défini comme n'importe quelle autre méthode.

  Puisqu'on doit modéliser un ensemble, où chaque élément n'est présent au plus qu'une seule fois, si l'élément est déjà présent dans le tableau des `@elements` (`include?`), alors on ne l'ajoute pas aux `@elements`.

  Dans cet exemple, le résultat retourné par «`<<`» est `self`, i.e., l'objet lui-même — équivalent au `this` de Java. Ceci permet de chaîner plusieurs appels les uns à la suite des autres : voir l'exemple Ruby 42, qui présente quelques expressions utilisant cette classe.

- La méthode `cardinalite` retourne le nombre d'éléments d'un ensemble — tous distincts, puisque l'ajout avec «`<<`» assure leur unicité.

- La classe `Ensemble` définit une méthode `each`, qui permet d'itérer sur les éléments d'un ensemble.

- La classe `Ensemble` inclut le module `Enumerable`. Les nombreuses méthodes de ce module sont donc disponibles : `map`, `select`, `reject`, `reduce`, `all?`, `any?`, etc.

  Les méthodes `contient?`, `somme`, `produit` et `to_s` peuvent donc être définies en utilisant les méthodes du module `Enumerable`, ici, `include?`, `reduce` et `map`.

- Soulignons que pour que les méthodes `somme` et `produit` fonctionnent correctement, un élément de l'ensemble doit pouvoir répondre aux messages «`+`» et «`*`».

- Les méthodes `cardinalite` et `contient?`, puisqu'elles en font qu'appeler une autre méthode avec un nom différent, aurait pu être définies plus simplement comme suit — donc les deux définitions de méthodes :

```
alias :cardinalite :count
alias :contient? :include?
```

Exemple Ruby 42 Quelques expressions utilisant un objet `Ensemble`.

```
?> # Cree un ensemble avec divers elements.
?> ens = Ensemble.new << 1 << 5 << 3

=> #<Ensemble:0x000000023c9298 @elements=[1, 5, 3]>
>> ens.to_s
=> "{ 1, 5, 3 }"

?> # L'operation << modifie l'objet.
?> ens << 2

=> #<Ensemble:0x000000023c9298 @elements=[1, 5, 3, 2]>
>> ens.to_s
=> "{ 1, 5, 3, 2 }"


?> # Appels a diverses methodes directement definies par Ensemble.
?> ens.contient? 10
=> false
>> ens.contient? 2
=> true


>> ens.somme
=> 11
>> ens.somme(33)
=> 44


>> ens.produit
=> 30
```

```
>?> # Appels a des methodes definies par Enumerable.
>> ens.to_s
=> "{ 1, 5, 3, 2 }"

?> ens.map { |x| x * 10 }
=> [10, 50, 30, 20]

>> ens.reject { |x| x.even? }
=> [1, 5, 3]

>> ens.find { |x| x >= 2 }
=> 5
```

Supposons que dans la classe `Array`, on veuille définir les méthodes `map` et `select`, et ce utilisant `each` ou `each_index`. Quel code faudrait-il écrire?

```
class Array
  def map
    ...
  end

  def select
    ...
  end
end
```

Remarque : *Conceptuellement*, dans la vraie classe `Array`, ces méthodes sont disponibles simplement parce que la classe `Array` **inclut** le module `Enumerable`. **En pratique**, la mise en oeuvre de ces méthodes pour la classe `Array` est faite de façon spécifique à cette classe, pour des raisons d'efficacité — notamment, méthodes écrites en C dans Ruby/MRI.

Exercice 9: Mises en oeuvre de `map` et `select`.

19 Expressions régulières et *pattern matching*

19.1 Les caractères spéciaux

Le tableau 2 présente les principaux caractères spéciaux utilisés dans les expressions régulières Ruby.

En gros, les expressions régulières Ruby combinent les éléments des expressions régulières *simples* d'Unix — création de groupes et de *back references* — et les éléments des expressions régulières *étendues* — caractères `?`, `+`, etc. — en plus d'ajouter d'autres éléments. (Seuls quelques-uns de ces éléments additionnels sont mentionnés dans ce qui suit!)

`\`	Supprime la signification spéciale du caractère qui suit
`.`	Un caractère arbitraire
Répétitions	
`*`	0, 1 ou plusieurs occurrences du motif qui précède
`?`	0 ou 1 occurrence du motif qui précède
`+`	1 ou plusieurs occurrences du motif qui précède
`{n}`	Exactement `n` occurrences du motif qui précède
`{n,}`	Au moins `n` occurrences du motif qui précède
`{,n}`	Au plus `n` occurrences du motif qui précède
`{n,m}`	De `n` à `m` occurrences du motif qui précède
Ancrages	
`^`	Début de la ligne
`$`	Fin de la ligne
Classes de caractères	
`[...]`	Un caractère qui fait partie de la classe
`[^...]`	Un caractère qui ne fait pas partie de la classe
`\d`	Un nombre décimal
`\D`	Tout sauf un nombre décimal
`\s`	Un espace blanc (espace, tabulation, saut de ligne, etc.)
`\S`	Tout sauf un espace blanc
`\w`	Un caractère alphanumérique = `a-zA-Z0-9_`
`\W`	Tout sauf un caractère alphanumérique
Autres caractères spéciaux	
m_1`|`m_2	Choix entre motif m_1 ou motif m_2
`(...)`	Création d'un groupe et d'une référence au groupe matché
`\b`	Une *frontière* de mot
`\A`	Le début de la chaine
`\z`	La toute fin de la chaine
`\Z`	La fin de la chaine (en ignorant possiblement le saut de ligne qui suit)

Tableau 2: Les principaux caractères spéciaux utilisés dans les expressions régulières.

19.2 Les expressions régulières et la méthode «=~»

Exemple Ruby 43 Une expression régulière est un objet de classe `Regexp`.

```
>> # Exemples de base.

>> /ab.*zz$/.class
=> Regexp

>> re = /ab.*zz$/
=> /ab.*zz$/
>> re.class
=> Regexp

>> re = Regexp.new( "ab.*zz$" )
=> /ab.*zz$/
>> re.class
=> Regexp
```

Remarques et explications pour l'exemple Ruby 43 :

- Une expression régulière est un objet de classe `Regexp`.
- Une expression régulière peut être créée à l'aide d'une expression utilisant les barres obliques — `/.../` — ou en créant explicitement avec `new` un objet de classe `Regexp`.

Exemple Ruby 44 Une expression régulière peut être utilisée dans une opération de *pattern-matching* avec «=~».

```
>> # Exemples de base (suite).

>> re = Regexp.new( "ab.*zz$" )
=> /ab.*zz$/

>> re =~ "abcdzz00"
=> nil

>> re =~ "abcdzz"
=> 0

>> re.=~ "abcdzz"
=> 0

>> re =~ "....abcdzz"
=> 4

>> "....abcdzz" =~ re
=> 4
```

```
>> puts "Ca matche" if re =~ "....abcdzz"
Ca matche
=> nil


>> re !~ "abcdzz00"
=> true

>> re !~ "abcdzz"
=> false
```

Remarques et explications pour l'exemple Ruby 44 :

- L'opération de base pour *matcher* une chaine et une expression régulière est l'opérateur «=~».

Cet opérateur est en fait une méthode de la classe `Regexp`, mais une version symétrique est aussi définie dans la classe `String`. Les appels suivants sont donc équivalents (les parenthèses sont optionnels!) :

  - `re =~ ch`
  - `re.=~( ch )`
  - `re.=~ ch`
  - `ch =~ re`

- La méthode «`=~`» retourne la position dans la chaine où débute le match si un tel match a pu être trouvé. Elle retourne `nil` si aucun match n'a pu être trouvé.
- La méthode «`!~`» retourne `true` si *aucun* match n'a pu être trouvé, sinon elle retourne `false`.

19.3 Quelques caractères spéciaux additionnels et quelques options

Exemple Ruby 45 Autres caractères spéciaux des motifs et options.

```
>> # L'option i permet d'ignorer la casse.

>> /bc/ =~ "ABCD"
=> nil

>> /bc/i =~ "ABCD"
=> 1
```

```
>> # Un "." *ne matche pas* un saut de ligne... sauf avec l'option m.
   # Un \s matche un saut de ligne.

>> /z.abc/ =~ "xyz\nabc"
=> nil

>> /z\sabc/ =~ "xyz\nabc"
=> 2

>> /z.abc/m =~ "xyz\nabc"
=> 2
```

Exemple Ruby 46 L'option «x» permet de mieux formater des expressions régulières complexes.

```
>> motif = /(#{CODE_REG}) # Le code regional
            -              # Un tiret
            (#{TEL})       # Le numero de tel.
           /x
=> /((?-mix:\d{3})) # Le code regional
         -              # Un tiret
         ((?-mix:\d{3}-\d{4})) # Le numero de tel.
        /x

>> motif.match "Tel.: 514-987-3000 ext. 8213"
=> #<MatchData "514-987-3000" 1:"514" 2:"987-3000">
```

Remarques et explications pour les exemples Ruby 45–46 :

- L'option «i» permet d'ignorer la casse.
- Par défaut, le *pattern matching* se fait ligne par ligne, et le «.» ne matche pas un saut de ligne — alors qu'un \s va le matcher.

  Par contre, si on indique l'option «m» — pour *multi-lignes* — alors le «.» pourra matcher un saut de ligne.
- L'option «x» permet de mettre en forme des expressions régulières complexes en utilisant des blancs, sauts de lignes et commentaires, qui sont ensuite ignorés dans l'opération de matchage.

Exemple Ruby 47 Début/fin de chaine vs. début/fin de ligne.

```
>> # Debut de ligne vs. debut de chaine.

>> /^abc/ =~ "xxx\nabc\n"
=> 4

>> /\Aabc/ =~ "xxx\nabc\n"
=> nil



>> # Fin de ligne vs. fin de chaine.

>> /abc$/ =~ "xxx\nabc\n"
=> 4

>> /abc\z/ =~ "xxx\nabc\n"
=> nil

>> /abc\n\z/ =~ "xxx\nabc\n"
=> 4

>> /abc\Z/ =~ "xxx\nabc\n"
=> 4
```

Remarques et explications pour l'exemple Ruby 47 :

- En mode ligne par ligne, le mode par défaut, les ancres «^» et «$» dénotent le début et la fin de la ligne.
- Si on veut matcher le début ou la fin de la chaine, on utilise «\A» et «\z» ou «\Z».

  La différence entre «\z» et «\Z» : le premier dénote la «vraie» fin de la chaine, donc ne matchera pas si un saut de ligne suit ; le deuxième, par contre, va matcher si ce qui suit est uniquement un saut de ligne.

19.4 La classe `MatchData`

Exemple Ruby 48 Les méthodes d'un objet `MatchData`, objet retourné par l'opération `Regexp#match`.

```
>> # Les objets MatchData.

>> CODE_REG = /\d{3}/
=> /\d{3}/
>> TEL = /\d{3}-\d{4}/
=> /\d{3}-\d{4}/

>> m = /(#{CODE_REG})-(#{TEL})/.match "FOO"
=> nil

>> m = /(#{CODE_REG})-(#{TEL})/.
       match "Tel.: 514-987-3000 ext. 8213"
=> #<MatchData "514-987-3000" 1:"514" 2:"987-3000">

>> m[0..-1]
=> ["514-987-3000", "514", "987-3000"]

>> m.begin(0)..m.end(0)
=> 6..18
>> m.begin(1)..m.end(1)
=> 6..9
>> m.begin(2)..m.end(2)
=> 10..18


>> m.pre_match
=> "Tel.: "

>> m.post_match
=> " ext. 8213"
```

Remarques et explications pour l'exemple Ruby 48 :

- Au lieu d'utiliser la méthode «`=~`» pour effectuer du *pattern-matching*, on peut utiliser à la place la méthode «`match`».

- Si la chaine ne matche pas le motif, l'appel à la méthode retourne `nil`, donc comme pour «`=~`».
- Si la chaine matche le motif, alors un objet `MatchData` est retourné.
- Un objet `MatchData` possède diverses méthodes, qui permettent de déterminer ce qui a été matché, les groupes capturés, la partie avant ou après le match, etc. :
  - Les éléments matchés, notamment les groupes, le groupe 0 indiquant la partie complète ayant été matchée :
    * `m[0]` : La partie complète matchée.
    * `m[1]` : Le premier groupe capturé par des `(...)`.
    * `m[2]` : Le deuxième groupe capturé par des `(...)`.
    * Etc.
  - Les positions de début et fin des groupes matchés (y compris le groupe 0) :
    * `m.begin(i)` : La position où débute le match du groupe `i`.
    * `m.end(i)` : La position *qui suit* la fin du match du groupe `i`.
  - `m.pre_match` : La partie de la chaine qui *précède* la partie matchée.
  - `m.post_match` : La partie de la chaine qui *suit* partie matchée.

Exemple Ruby 49 Les groupes *avec noms* et les variables spéciales «`$i`» définies par la méthode «`=~`».

```
# Des groupes avec noms explicites.

>> m = /(?<code_reg>#{CODE_REG})-(?<tel>#{TEL})/.
       match "Tel.: 514-987-3000 ext. 8213"
=> #<MatchData "514-987-3000" code_reg:"514"
                              tel:"987-3000">

>> m[:code_reg]
=> "514"

>> m.begin(:code_reg)
=> 6


>> m[:tel]
=> "987-3000"

>> m.end(:tel)
=> 18
```

```
# Les variables speciales $1, $2, etc. pour les groupes.

>> if /(#{CODE_REG})-(#{TEL})/ =~
         "Tel.: 514-987-3000 ext. 8213"
     "code reg. = #{$1}; tel. = #{$2}"
   end
=> "code reg. = 514; tel. = 987-3000"
```

Remarques et explications pour l'exemple Ruby 49 :

- Un groupe capturé par des `(...)` peut être explicitement nommé, et ce en définissant le groupe capturé avec `(?<ident>...)`.
- Lorsqu'on utilise la méthode «`=~`», il est possible d'accéder aux groupes capturés dans le match en utilisant les variables spéciales — `$1` (1^{er} groupe), `$2` (2^{e} groupe), etc.

- Il existe aussi des variables spéciales pour le match dans son ensemble, la partie avant le match, la partie après, etc., mais les règles de style Ruby veulent qu'on évite de les utiliser : si on a besoin de ces élément, on utilise plutôt un objet `MatchData` explicite créé avec la méthode `match` et on utilise les méthodes associées.

Qu'est-ce qui sera imprimé par les instructions `p` suivantes :

```
code_permanent = /(\w{4})     # NOMP
                  (\d{2})     # Annee
                  (\d{2})     # Mois
                  (\d{2})     # Jour
                  ([^\D]{2})
                 /x

m = code_permanent.
          match "CP: DEFG11229988."

p m[1]
p m[5]
p m.pre_match
p m.post_match
```

Exercice 10: Objet `MatchData`.

20 Interactions avec l'environnement

Ctte section traite des arguments du programme, des entrées/sorties, des manipulations de fichiers, et de l'exécution de commandes externes.

20.1 Arguments du programme

Exemple Ruby 50 Les arguments d'un programme Ruby et les variables d'environnement.

```
$ cat argv.rb
#!/usr/bin/env ruby

i = 0
while arg = ARGV.shift do
  puts "ARGV[#{i++}] = '#{arg}' (#{arg.class})"
end

puts "ENV['FOO'] = '#{ENV['FOO']}'"
ENV['FOO'] = 'FOO argv.rb'
puts "-----"
```

Remarques et explications pour l'exemple Ruby 50 :

- Les arguments transmis lors de l'appel du programme sont accessibles par l'intermédiaire du tableau `ARGV`.
- Contrairement aux scripts Unix, `ARGV[0]` est *le premier argument*, et non le nom du programme — c'est la variable spéciale $0 qui contient le nom du programme.
- Le tableau `ARGV` peut être manipulé et modifié... sans générer d'avertissement — puisqu'étant en majuscules, `ARGV` devrait être traité comme une constante.
- On peut accéder aux variables de l'environnement — au sens Unix — à l'aide du `Hash ENV`. La clé à utiliser est la chaine dénotant le nom de la variable.

  La valeur d'une variable d'environnement peut aussi être modifiée en utilisant `ENV`. Toutefois, cette modification ne sera visible que dans le processus courant ou dans les enfants de ce processus.

```
$ echo $FOO


$ ./argv.rb
ENV['FOO'] = ''
-----

$ ./argv.rb 1234 'abc "" def' abc def "'"
ARGV[0] = '1234' (String)
ARGV[1] = 'abc "" def' (String)
ARGV[2] = 'abc' (String)
ARGV[3] = 'def' (String)
ARGV[4] = ''' (String)
ENV['FOO'] = ''
-----

$ export FOO=xyz; ./argv.rb def; echo $FOO
ARGV[0] = 'def' (String)
ENV['FOO'] = 'xyz'
-----
xyz

$ FOO=123 ./argv.rb def; echo $FOO
ARGV[0] = 'def' (String)
ENV['FOO'] = '123'
-----
xyz
```

Soit le script suivant :

```
$ cat argv2.rb
#!/usr/bin/env ruby

ENV['NB'].to_i.times do
  puts ARGV[0] + ARGV[1]
end
```

Qu'est-ce qui sera imprimé par les appels suivants :

```
# a.
NB=3 argv2.rb 3 8

# b.
NB=2 argv2.rb [1, 2] [3]

# c.
unset NB; argv2.rb [1009, 229342] [334]
```

Exercice 11: Utilisation de `ARGV` et `ENV`.

20.2 Écriture sur le flux de sortie standard : `printf`, `puts`, `print` et `p`

Exemple Ruby 51 Exemples de conversions implicites lorsqu'on utilise `printf`.

```
>> printf "%d\n", "123"
123
=> nil

>> printf "%s\n", "123"
123
=> nil

>> printf "%d\n", "abc"
ArgumentError: invalid value for Integer(): "abc"
        [...]

>> printf "%s\n", "abc"
abc
=> nil

>> printf( "%d\n", [10, 20] )
TypeError: can't convert Array into Integer
        [...]

>> printf( "%s\n", [10, 20] )
[10, 20]
=> nil
```

```
# On peut aussi utiliser un format pour
# generer une chaine, sans effet
# sur le flux de sortie.

>> res = sprintf "%d\n", 123
=> "123\n"

>> res
=> "123\n"
```

Remarques et explications pour l'exemple Ruby 51 :

- La méthode `printf` utilise par défaut le flux standard de sortie, `STDOUT`.
- Le format utilise les mêmes règles qu'en C.

  La différence toutefois est qu'en Ruby, certaines *conversions*, lorsqu'elles sont possibles (?!), sont faites de façon implicite.
- La méthode `sprintf`, comme en C, n'émet aucune information sur le flux de sortie. Elle génère plutôt une chaine (*string* `printf`).

Exemple Ruby 52 Écriture d'un entier ou d'une chaine simple.

```
$ cat print-et-al.rb
#!/usr/bin/env ruby

def imprimer( methode, *valeurs )
  puts "*** Avec #{methode}:"
  valeurs.each do |x|
    send methode, x
    puts "..."
  end
end

imprimer( :puts, 123, "123" )
imprimer( :print, 123, "123" )
imprimer( :p, 123, "123" )


$ ./print-et-al.rb
*** Avec puts:
123
...
123
...
*** Avec print:
123...
123...
*** Avec p:
123
...
"123"
...
```

Exemple Ruby 53 Écriture d'un tableau d'entiers ou un tableau de chaines.

```
$ cat print-et-al.rb
#!/usr/bin/env ruby

def imprimer( methode, *valeurs )
  puts "*** Avec #{methode}:"
  valeurs.each do |x|
    send methode, x
    puts "..."
  end
end

imprimer( :puts, [123, 456], ["123", "456"] )
imprimer( :print, [123, 456], ["123", "456"] )
imprimer( :p, [123, 456], ["123", "456"] )


$ ./print-et-al.rb
*** Avec puts:
123
456
...
123
456
...
*** Avec print:
[123, 456]...
["123", "456"]...
*** Avec p:
[123, 456]
...
["123", "456"]
...
```

Exemple Ruby 54 Écriture d'un objet **qui n'a pas** de méthodes `to_s` et `inspect`.

```
$ cat print-et-al.rb
#!/usr/bin/env ruby

def imprimer( methode, *valeurs )
  puts "*** Avec #{methode}:"
  valeurs.each do |x|
    send methode, x
    puts "..."
  end
end

class Bar
  def initialize( val ); @val = val; end
end




imprimer( :puts, Bar.new(10) )
imprimer( :print, Bar.new(10) )
imprimer( :p, Bar.new(10) )


$ ./print-et-al.rb
*** Avec puts:
#<Bar:0x000000015022a0>
...
*** Avec print:
#<Bar:0x00000001502110>...
*** Avec p:
#<Bar:0x00000001501f80 @val=10>
...
```

Exemple Ruby 55 Écriture d'un objet **qui a** des méthodes `to_s` et `inspect`.

```
$ cat print-et-al.rb
#!/usr/bin/env ruby

def imprimer( methode, *valeurs )
  puts "*** Avec #{methode}:"
  valeurs.each do |x|
    send methode, x
    puts "..."
  end
end

class Foo
  def initialize( val ); @val = val; end

  def to_s; "#{@val}"; end

  def inspect; "#<Foo: val=#{@val}>"; end
end

imprimer( :puts, Foo.new(10) )
imprimer( :print, Foo.new(10) )
imprimer( :p, Foo.new(10) )


$ ./print-et-al.rb
*** Avec puts:
10
...
*** Avec print:
10...
*** Avec p:
#<Foo: val=10>
...
```

Remarques et explications pour les exemples Ruby 52–55 :

- Outre `printf`, plusieurs autres méthodes sont disponibles pour émettre sur le flux de sortie standard (ou tout autre flux, si on utilise un objet approprié comme récepteur du message), notamment `puts`, `print` et `p`.

- En gros, voici les principales différences entre ces trois méthodes :
  - `puts` et `p` ajoutent un saut de ligne à la fin de la chaine émise, alors que `print` n'ajoute pas de saut de ligne.
  - `puts` utilise la méthode `to_s` pour convertir l'objet en une chaine, alors que `p` utilise `inspect`. Quant à `print`, il utilise aussi `to_s`, sauf pour les tableaux.
- Donc, en gros (mais pas tout à fait dans le cas des tableaux), on a les équivalences suivantes :

`puts x`	`print x.to_s; print "\n"`
`p x`	`print x.inspect; print "\n"`

- Remarque : `p` est particulièrement utile pour déboguer, car on voit plus explicitement le type d'un objet — e.g., dans le cas d'une chaine, les guillemets sont indiqués explicitement, dans le cas d'un tableau, on a les crochets et les virgules, etc.
- Lorsqu'un objet ne possède pas de méthode `to_s` ou `inspect`, c'est la méthode de même nom de la classe `Object` qui est utilisée :
  - `Object#to_s` : Retourne le nom de la classe et l'adresse de l'objet.
  - `Object#inspect` : Retourne le nom de la classe, l'adresse de l'objet et les valeurs des différentes variables d'instance.

20.3 Manipulation de fichiers

Exemple Ruby 56 Différentes façon de lire et d'afficher sur `stdout` le contenu d'un fichier texte.

```
$ cat cat.rb
#!/usr/bin/env ruby

nom_fichier = ARGV[0]

File.open( nom_fichier, "r" ) do |fich|
  fich.each_line do |ligne|
    puts ligne
  end
end
```

```
$ cat foo.txt
abc def
123 456

xxx
...

$ ./cat.rb foo.txt
abc def
123 456

xxx
...
```

Exemple Ruby 56 Différentes façon de lire et d'afficher sur `stdout` le contenu d'un fichier texte.

```
$ cat cat.rb
#!/usr/bin/env ruby

nom_fichier = ARGV[0]

fich = File.open( nom_fichier, "r" )

fich.each_line do |ligne|
  puts ligne
end

fich.close
```

```
$ cat foo.txt
abc def
123 456

xxx
...

$ ./cat.rb foo.txt
abc def
123 456

xxx
...
```

Exemple Ruby 56 Différentes façon de lire et d'afficher sur `stdout` le contenu d'un fichier texte.

```
$ cat cat.rb
#!/usr/bin/env ruby

nom_fichier = ARGV[0]

IO.readlines( nom_fichier ).each do |ligne|
  puts ligne
end
```

```
$ cat foo.txt
abc def
123 456

xxx
...

$ ./cat.rb foo.txt
abc def
123 456

xxx
...
```

Exemple Ruby 56 Différentes façon de lire et d'afficher sur `stdout` le contenu d'un fichier texte.

```
$ cat cat.rb
#!/usr/bin/env ruby

nom_fichier = ARGV[0]

puts IO.readlines( nom_fichier )
```

```
$ cat foo.txt
abc def
123 456

xxx
...

$ ./cat.rb foo.txt
abc def
123 456

xxx
...
```

Exemple Ruby 57 Différentes façon de lire et d'afficher sur `stdout` le contenu d'un fichier texte, dont une façon qui permet de recevoir les données *par l'intermédiaire du flux standard d'entrée.*

```
$ cat cat.rb
#!/usr/bin/env ruby

nom_fichier = ARGV[0]

puts (nom_fichier ? IO : STDIN).readlines nom_fichier
```

```
$ cat foo.txt
abc def
123 456

xxx
...

$ ./cat.rb foo.txt
abc def
123 456

xxx
...

$ cat foo.txt | ./cat.rb
abc def
123 456

xxx
...
```

Remarques et explications pour les exemples Ruby 56–57 :

- Il est possible d'appeler `File.open` avec un bloc, qui reçoit en argument le descripteur du fichier ouvert. Dans ce cas, le fichier est automatiquement fermé lorsque le bloc se termine!

- L'ouverture d'un fichier avec un bloc correspond en fait à la deuxième approche illustrée : le fichier est ouvert explicitement avec la méthode (de classe) `File.open`, puis le fichier est fermé explicitement avec `close`. Parce qu'il est dangeureux d'oublier de fermer le fichier avec cette approche, c'est la première qui est recommandée en Ruby.

  De plus, comme on le verra plus loin, le fichier sera fermé même en cas d'exception.

- Pour lire un fichier de texte, on peut aussi utiliser directement `IO.readlines`, qui retourne un tableau des lignes lues.

Un fichier peut évidemment être ouvert dans d'autres modes que la lecture. La figure 5 donne les différents modes pouvant être utilisés.

```
Mode |  Meaning
-----+-----------------------------------------------------------
"r"  |  Read-only, starts at beginning of file  (default mode).
-----+-----------------------------------------------------------
"r+" |  Read-write, starts at beginning of file.
-----+-----------------------------------------------------------
"w"  |  Write-only, truncates existing file
     |  to zero length or creates a new file for writing.
-----+-----------------------------------------------------------
"w+" |  Read-write, truncates existing file to zero length
     |  or creates a new file for reading and writing.
-----+-----------------------------------------------------------
"a"  |  Write-only, starts at end of file if file exists,
     |  otherwise creates a new file for writing.
-----+-----------------------------------------------------------
"a+" |  Read-write, starts at end of file if file exists,
     |  otherwise creates a new file for reading and
     |  writing.
-----+-----------------------------------------------------------
 "b" |  Binary file mode (may appear with
     |  any of the key letters listed above).
     |  Suppresses EOL <-> CRLF conversion on Windows. And
     |  sets external encoding to ASCII-8BIT unless explicitly
     |  specified.
-----+-----------------------------------------------------------
 "t" |  Text file mode (may appear with
     |  any of the key letters listed above except "b").
```

Figure 5: Modes d'ouverture des fichiers (source : `http://ruby-doc.org/core-2.0.0/IO.html`).

20.4 Exécution de commandes

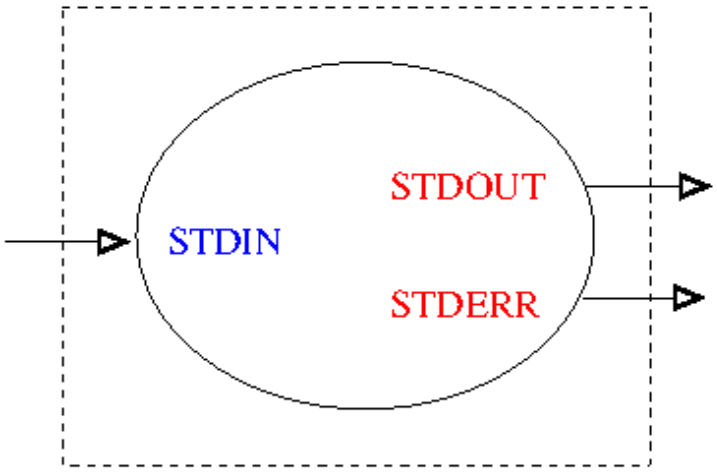
Exemple Ruby 58 Exécution de commandes externes avec *backticks* ou `%x{...}`

```
>> # Execution avec backticks.
>> ext = 'rb'
=> "rb"

>> puts `ls [e]*.#{ext}`
ensemble.rb
ensemble_spec.rb
entrelacement.rb
=> nil

>> "#{$?}"
=> "pid 29829 exit 0"
```

```
>> # Execution avec %x{...}.
>> puts %x{ ls [e]*.#{ext} }
ensemble.rb
ensemble_spec.rb
entrelacement.rb
=> nil

>> $?
=> #<Process::Status: pid 30019 exit 0>
```

```
>> # Emission sur stderr vs. stdout
>> %x{ ls www_xx_z }
ls: impossible d'accéder à www_xx_z:
                Aucun fichier ou dossier de ce type
=> ""

>> "#$?"
=> "pid 29831 exit 2"
```

Vue de l'intérieur



Vue de l'extérieur

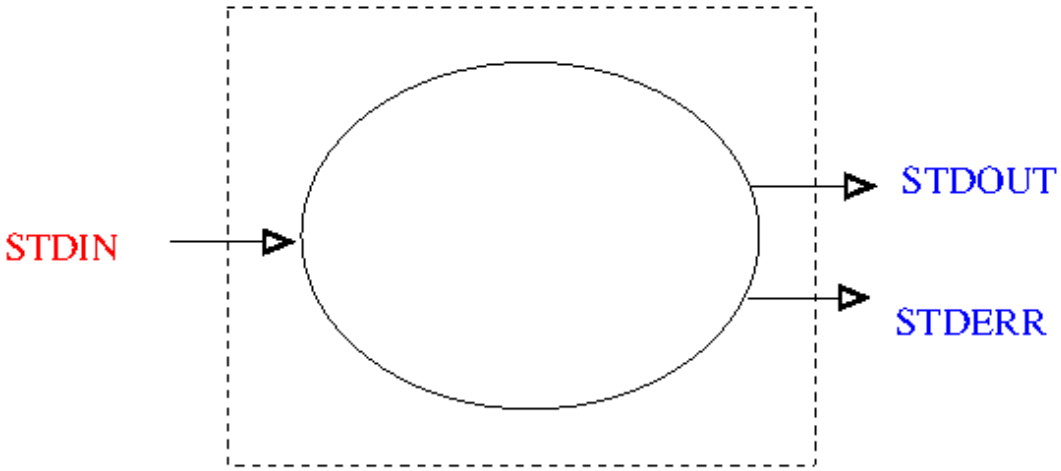


Figure 6: Deux points de vue sur les flux associés à un processus : a) Vue de l'intérieur du processus — on **lit** sur `STDIN` et on **écrit** sur `STDOUT` et `STDERR`; b) Vue de l'extérieur du processus — on **écrit** sur `STDIN` et on **lit** de `STDOUT` et `STDERR`.

Exemple Ruby 59 Exécution de commandes externes avec `Open3.popen3`.

```
$ cat commandes2.rb
require 'open3'
Open3.popen3( "wc -lw" ) do |stdin, stdout, stderr|
  stdin.puts ["abc def", "", "1 2 3"]
  stdin.close

  puts "--stdout--"
  puts stdout.readlines
  puts "--stderr--"
  puts stderr.readlines
  puts
end

$ ./commandes2.rb
--stdout--
      3       5
--stderr--
```

Exemple Ruby 60 Exécution de commandes externes avec `Open3.popen3`.

```
$ cat commandes3.rb
require 'open3'
Open3.popen3( "wc -lw xsfdf.txt" ) do |_, out, err|
  puts "--out--"
  puts out.readlines
  puts "--err--"
  puts err.readlines
  puts
end



$ ./commandes3.rb
--out--
--err--
wc: xsfdf.txt: Aucun fichier ou dossier de ce type
```

Remarques et explications pour les exemples Ruby 58–60 :

- Comme dans un script *shell*, on peut utiliser les *backticks* pour lancer l'exécution d'un programme externe et obtenir son résultat.

  Toutefois, bien qu'on puisse utiliser les *backticks*, règle générale on utilise plutôt la forme avec `%x{...}`.

- Ces deux formes permettant l'interpolation de variables et expressions.

- Le statut retourné par la commande est dans la variable «`$?`», commme dans un script *shell*.

- Pour interpoler les variables spéciales, il n'est pas toujours nécessaire de les mettre entre accolades.

- Ces deux forme ne donnent pas accès explicite aux différents flux. Plus spécifiquement, le résultat retourné est ce qui est produit sur le flux de sortie standard (`stdout`).

- On peut utiliser le module `Open3` pour un contrôle plus fin des flux, notamment pour avoir accès au flux d'erreur (`stderr`).

- L'exemple utilise la méthode `open3`, qui permet de manipuler explicitement tant le flux d'entrée que les flux de sortie.

  On remarque une chose intéressante dans ce contexte, où le bloc fournit les données et obtient les résultats : le flux `stdin` est utilisé en écriture, alors que les flux `stdout` et `stderr` sont utilisés en lecture!

- Signalons que les noms des flux utilisés à l'intérieur du bloc sont arbitraires, comme l'illustre le dernier exemple.

21 Traitement des exceptions

21.1 Classe `Exception` et sous-classes standards

En Ruby (comme en Java d'ailleurs), les exceptions sont aussi des objets. La figure 7 présente la hiérarchie de classe des exceptions standards prédéfinies.

```
NoMemoryError
ScriptError
    LoadError
    NotImplementedError
    SyntaxError
SignalException
    Interrupt
StandardError -- default for rescue
    ArgumentError
    IndexError
        StopIteration
    IOError
        EOFError
    LocalJumpError
    NameError
        NoMethodError
    RangeError
        FloatDomainError
    RegexpError
    RuntimeError -- default for raise
    SecurityError
    SystemCallError
        Errno::*
    SystemStackError
    ThreadError
    TypeError
    ZeroDivisionError
SystemExit
fatal -- impossible to rescue
```

Figure 7: Hiérarchie des classes/sous-classes standards pour les exceptions (source : `http://ruby-doc.org/core-2.1.1/Exception.html`).

21.2 Attraper et traiter une exception

Exemple Ruby 61 Une méthode `div` qui attrape et traite diverses exceptions.

```
>> def div( x, y )
     begin
       z = x / y
     rescue ZeroDivisionError => e
       puts "*** Division par 0 (#{e})"
       p e.backtrace
       nil
     rescue Exception => e
       puts "*** Erreur = '#{e.inspect}'"
     end
   end
=> :div


>> div 3, 0
*** Division par 0 (divided by 0)
["(irb):4:in '/'",
 "(irb):4:in 'div'", "(irb):14:in 'irb_binding'",
 "/home/tremblay/.rvm/rubies/ruby-2.1.4/lib/ruby/2.1.0/irb/workspace.rb:86:in 'eval
 ...,
 "/home/tremblay/.rvm/rubies/ruby-2.1.4/bin/irb:11:in '<main>'"]
=> nil

>> div 3, nil
*** Erreur = '#<TypeError: nil can't be coerced into Fixnum>'
=> nil

>> div nil, 3
*** Erreur = '#<NoMethodError: undefined method '/' for nil:NilClass>'
=> nil
```

Exemple Ruby 62 Une méthode `traiter_fichier` qui attrape et traite des exceptions et qui s'assure de restaurer le système dans un bon état, qu'une exception soit signalée ou non — dans ce cas-ci, en s'assurant de fermer le descripteur du fichier ayant été ouvert.

```
>> def traiter_fichier( fich )
     f = File.open( fich )
     begin
       traiter_contenu_fichier( f.readlines )
       puts "+++ Traitement termine"
     rescue Exception => e
       puts "*** Erreur = '#{e.inspect}'"
     ensure
       f.close
     end

     f.inspect  # Pour voir l'etat final de f.
   end
=> :traiter_fichier


>> traiter_fichier( "foo.txt" )
+++ Traitement termine
=> "#<File:foo.txt (closed)>"

>> traiter_fichier( "bar.txt" )
*** Erreur = '#<RuntimeError: Erreur dans traiter_contenu_fichier>'
=> "#<File:bar.txt (closed)>"
```

Exemple Ruby 63 La méthode `File.open`, lorsqu'appelée avec un bloc, assure que le fichier sera fermé, qu'une exception survienne ou pas.

```
>> def traiter_fichier( fich )
     le_f = nil
     File.open( fich ) do |f|
       le_f = f
       begin
         traiter_contenu_fichier( f.readlines )
         puts "+++ Traitement termine"
       rescue Exception => e
         puts "*** Erreur = '#{e.inspect}'"
       end
     end
     le_f.inspect
   end
=> :traiter_fichier


>> traiter_fichier( "bar.txt" )
*** Erreur = '#<RuntimeError: Erreur dans traiter_contenu_fichier>'
=> "#<File:bar.txt (closed)>"
```

Remarques et explications pour les exemples Ruby 61–63 :

- Une séquence d'instructions avec traitement d'exceptions est introduite par `begin`/`end`.
- Une clause `rescue` permet d'indiquer quelle/quelles exceptions est/sont traitée/s par cette clause. L'identificateur qui suit «`=>`» donne un nom à l'exception attrapée et donc en cours de traitement.
- Il est possible d'attraper **n'importe quelle exception** en indiquant «`rescue Exception`». Par contre, si on indique simplement «`rescue`», ceci est équivalent à «`rescue StandardError`».
- Un bloc de traitement d'exception peut (devrait) aussi inclure une clause `ensure`. Les instructions associées à cette clause seront **toujours** exécutées — qu'une exception survienne ou pas.
- La méthode `File.open` utilisée avec un bloc **assure de toujours fermer le fichier ayant été ouvert, qu'une exception survienne ou pas**. En général, il est donc préférable d'utiliser cette forme de `File.open`.

21.3 Signaler une exception

Exemple Ruby 64 Exemples illustrant l'instruction `fail`, appelée avec 0, 1 ou 2 arguments.

```
>> class MonException < RuntimeError
     def initialize( msg = nil )
       super
     end
   end
=> :initialize

>> def executer
     begin
       yield
     rescue Exception => e
       "classe = #{e.class}; message = '#{e.message}'"
     end
   end
=> :executer

>> executer { fail }
=> "classe = RuntimeError; message = ''"

>> executer { fail "Une erreur!" }
=> "classe = RuntimeError; message = 'Une erreur!'"

>> executer { fail MonException }
=> "classe = MonException; message = 'MonException'"

>> executer { fail MonException, "Probleme!!" }
=> "classe = MonException; message = 'Probleme!!'"
```

Exemple Ruby 65 Exemples illustrant l'instruction `raise` utilisée pour **resignaler** une exception.

```
>> def executer
     begin
       yield
     rescue Exception => e
       puts "classe = #{e.class}; message = '#{e.message}'"
       raise
     end
   end
=> :executer

>> executer { fail MonException, "Probleme!!" }
classe = MonException; message = 'Probleme!!'
MonException: Probleme!!
              from (irb):16:in 'block in irb_binding'
              from (irb):9:in 'executer'
              from (irb):16
              from /home/tremblay/.rvm/rubies/ruby-2.1.4/bin/irb:11:in
              '<main>'
```

Remarques et explications pour les exemples Ruby 64–65 :

- L'instruction `fail` permet de signaler une exception. Cette instruction peut recevoir divers arguments :
  - Aucun argument : soulève une exception `RuntimeError`, sans message associé.
  - Un unique argument `String` : soulève une exception `RuntimeError`, avec la chaine utilisée comme message.
  - Un unique argument qui est une sous-classe d'`Exception` : soulève une exception de la classe indiquée, avec le nom de la classe utilisée comme message.
  - Un premier argument qui est une sous-classe d'`Exception` et une deuxième argument qui est un `String` : soulève une exception de la classe indiquée, avec la chaine utilisée comme message.
- Il est aussi possible d'utiliser l'instruction `raise` pour signaler une exception. Les deux sont en fait **des synonymes**.

Certains auteurs suggèrent d'utiliser `fail` et `raise` comme suit :

- On utilise `fail` lorsqu'on veut signaler une nouvelle exception, donc suite à un problème qu'on vient tout juste de détecter — premier appel/signal.
- On utilise `raise` lorsqu'on désire **resignaler** une exception, qui a déjà signalée. Par exemple, on exécute une clause `rescue`, on fait certains traitements, puis on resignale la même exception pour que les méthodes appelantes puissent elles aussi traiter l'exception. Dans ce cas, il n'est pas nécessaire d'indiquer explicitement le nom de l'exception.

22 Autres éléments de Ruby

22.1 L'opérateur préfixe «*»

Exemple Ruby 66 Utilisation de l'opérateur «*» (*splat*) **devant** un objet — `Range`, scalaire ou `Range` — dans une expression.

```
>> # L'operateur "splat" (*) devant un tableau "enleve" un niveau de
   # tableau, i.e., integre directement les elements du tableau plutot
   # que le tableau lui-meme.

>> a = [98, 99]
=> [98, 99]

>> [1, [10, 20], a, 1000]
=> [1, [10, 20], [98, 99], 1000]

>> [1, *[10, 20], *a, 1000]
=> [1, 10, 20, 98, 99, 1000]
```

```
>> # L'operateur splat (*) devant un scalaire ou un Range genere un
   # tableau avec l'element ou les elements indiques... mais pas
   # n'importe ou.

>> a = *10
=> [10]

>> a = *(1..10)
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

>> (1..10).to_a
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]



# Mais...

>> *(1..10)
SyntaxError: (irb):41: syntax error, unexpected '\n',
                       expecting :: or '[' or '.'
        ...

>> *10
SyntaxError: (irb):33: syntax error, unexpected '\n',
                       expecting:: or '[' or '.'
  ...
```

Exemple Ruby 67 Utilisation de l'opérateur «*» du coté gauche d'une affectation parallèle (multiple).

```
>> # Dans la partie gauche d'une affectation parallele, un * permet de
   # <<deconstruire>> un tableau. Dans ce cas, la variable
   # prefixee avec * doit etre unique et va denoter un sous-tableau
   # d'elements.

>> a, b, c = [10, 20, 30, 40]
=> [10, 20, 30, 40]

>> puts "a = #{a}; b = #{b}; c = #{c}"
a = 10; b = 20; c = 30
=> nil


>> a, *b, c = [10, 20, 30, 40]
=> [10, 20, 30, 40]

>> puts "a = #{a}; b = #{b}; c = #{c}"
a = 10; b = [20, 30]; c = 40
=> nil
```

```
>> premier, *derniers = [10, 20, 30]
=> [10, 20, 30]

>> puts "premier = #{premier}; derniers = #{derniers}"
premier = 10; derniers = [20, 30]
=> nil


>> *premiers, dernier = [10, 20, 30]
=> [10, 20, 30]

>> puts "premiers = #{premiers}; dernier = #{dernier}"
premiers = [10, 20]; dernier = 30
=> nil
```

Exemple Ruby 68 Utilisation de «*» dans la spécification de paramètres de méthodes : l'effet est semblable à des affectations parallèles.

```
>> # L'utilisation de * s'applique aussi aux parametres
   # formels d'une methode, ainsi qu'aux arguments effectifs
   # (expressions passees en argument).
>> def foo( x, *args )
     puts "x = #{x}"
     args.each_index { |k| puts "args[#{k}] = #{args[k]}" }
   end
=> :foo

>> foo( 10 )
x = 10
=> []

>> foo( 10, 20 )
x = 10
args[0] = 20
=> [20]

>> foo( 10, 20, 30 )
x = 10
args[0] = 20
args[1] = 30
=> [20, 30]

>> foo( [10, 20, 30] )
x = [10, 20, 30]
=> []

>> foo( *[10, 20, 30] )
x = 10
args[0] = 20
args[1] = 30
=> [20, 30]
```

22.2 L'opérateur préfixe «&» pour la manipulation de blocs

Exemple Ruby 69 Utilisation de l'opérateur «&» pour rendre explicite un bloc comme paramètre d'une méthode.

```
>> # L'operateur prefixe & utilise devant le dernier parametre
   # rend explicite le bloc transmis a l'appel de la methode.
   # Ce parametre est alors un objet Proc pouvant
   # etre execute avec call.

>> def call_yield( x, &bloc )
     return x unless block_given?

     [ bloc.class, bloc.call(x), yield(x) ]
   end
=> :call_yield

>> call_yield( 99 )
=> 99

>> call_yield( 99 ) { |x| x + 10 }
=> [Proc, 109, 109]
```

Exemple Ruby 70 Utilisation de l'opérateur «&» pour transformer un objet `lambda` ou `Symbole` en bloc.

```
>> # L'operateur prefixe & devant une lambda expression
   # transforme l'objet Proc en un bloc.
   # Ce bloc peut alors transmis explicitement comme
   # dernier argument (argument additionnel en plus
   # des arguments non blocs explicites).

>> double = lambda { |x| 2 * x }
=> #<Proc:0x000000028b0950@(irb):24 (lambda)>


>> call_yield( 2 ) { |x| 2 * x }
=> [Proc, 4, 4]

>> call_yield( 2 ) double
SyntaxError: (irb):26: syntax error, unexpected tIDENTIFIER, expecting end-of-
        ...

>> call_yield( 2 ) &double
TypeError: Proc can't be coerced into Fixnum
        ...


>> call_yield( 2, &double )
=> [Proc, 4, 4]
```

```
>> # Cette transformation s'applique meme lorsque le bloc
   # est implicite.
   # Et elle s'applique aussi aux symboles,
   # via un appel implicite a to_proc.

>> def yield_un_arg( x )
     yield( x )
   end
=> :yield_un_arg

>> yield_un_arg( 24, &double )
=> 48

>> yield_un_arg( 24, &:even? )
=> true

>> # :s.to_proc == Proc.new { |o| o.s } (...ou presque)
>> yield_un_arg( 24, &:even?.to_proc )
=> true


>> yield_un_arg( 24, &:- )
ArgumentError: wrong number of arguments (0 for 1)
    ...

>> yield_un_arg( 24, &:-@ ) # Voir section suivante.
=> -24
```

22.3 Les opérateurs (préfixes) unaires

Exemple Ruby 71 Opérateurs (préfixes) unaires définis par le programmeur.

```
>> class Foo
     def +( autre )
       puts "self = #{self}; autre = #{autre}"
     end

     def +@
       puts "self = #{self}"
     end
   end
=> :+@

>> foo = Foo.new
=> #<Foo:0x000000019910c8>

>> foo + 10
self = #<Foo:0x000000019910c8>; autre = 10
=> nil

>> + foo
self = #<Foo:0x000000019910c8>
=> nil
```

Remarques et explications pour l'exemple Ruby 71 :

- Les symboles tels que «:+» et «:-» dénotent par défaut les opérateurs **binaires**.
- Pour référer aux opérateurs (préfixes) **unaires**, on utilise les symboles «:+@» ou «:+@».
- Comme n'importe quelles autres méthodes, les méthodes pour les opétateurs, tant binaires qu'unaires, peuvent être définies par le programmeur.

A Installation de Ruby sur votre machine

Voici comment procéder pour installer `Ruby` ou `JRuby` sur une machine Linux avec CentOS — donc un Linux comme sur les machines des laboratoires `labunix`.

Les étapes devraient être les mêmes pour d'autres versions de Linux (pas testé ☹) ou pour Mac OS (testé ☺). Quant à Windows,... désolé, mais aucune idée ☹

L'installation décrite utilise `rvm` (*Ruby Version Manager*), un outil qui permet d'installer et utiliser plusieurs versions différentes de Ruby (JRuby, Ruby/MRI 1.9, 2.1, 2.2, etc.). Comme le suggère le site Web de `rvm`[8] , à cause de la façon dont sont gérées les bibliothèques Ruby (les `gems`), il est préférable que `rvm` soit installé dans votre compte personnel — donc sans installation `sudo`.

Voici donc les étapes à suivre :

1. Obtenir la clé pour `rvm` et obtenir `rvm` (dernière version stable) :

```
$ gpg --keyserver\
        hkp://keys.gnupg.net\
      --recv-keys\
        409B6B1796C275462A1703113804BB82D39DC0E3
$ \curl -sSL https://get.rvm.io | bash -s stable
```

2. Activer les fonctions associées à `rvm` :

```
$ source ~/.rvm/scripts/rvm
```

3. Pour la programmation séquentielle (cours MGL7460), je vous suggère d'installer `ruby` — `rvm list` permet de vérifier qu'il est bien installé :

```
$ rvm install ruby
$ rvm list
```

4. Installer le `gem bundler`, requis ultérieuement pour la gestion des `gems` :

```
$ gem install bundler
```

[8] `https://rvm.io/`

B Le cadre de tests unitaires `MiniTest`

Nous expliquons tout d'abord ce qu'est un cadre de tests. Nous présentons ensuite le cadre de tests «standard» pour Ruby : `MiniTest`.

B.1 Tests unitaires et cadres de tests

Niveaux de tests

Il existe différents niveaux de tests [RK03] :

- Tests unitaires : Un test unitaire vérifie le bon fonctionnement *d'un module*, d'une *classe* ou d'un *composant* indépendant. Les tests unitaires forment la *fondation* sur laquelle repose l'ensemble des activités de tests : il est inutile de tester l'ensemble du système si *chacun* des modules n'a pas été testé à fond.

- Tests d'intégration : Les tests d'intégration vérifient que les principaux *sous-systèmes* fonctionnent correctement, c'est-à-dire que les différents modules qui composent un sous-système donné sont correctement *intégrés* ensemble. Ces tests peuvent être vus comme une forme de tests unitaires, l'unité étant alors un *groupe (cohésif) de modules* plutôt qu'un unique module.

- Tests de système : Les tests de système vérifient le fonctionnement du système dans son ensemble, en termes des fonctionnalités attendues du système.

- Tests d'acceptation : Les tests d'acceptation sont des tests, de niveau système, effectués lorsque le système est prêt à être déployé, donc juste avant qu'il soit livré et officiellement installé.

Dans ce qui suit, où l'on s'intéresse à la mise en oeuvre de petites unités de programmes, on s'intéresse plus particulièrement aux *tests unitaires*.

B.1.1 Pratique professionnelle et tests

De nos jours, on considère que dans une pratique *professionnelle* de développement de logiciels, l'écriture de tests unitaires fait partie intégrante du processus d'écriture de code — en d'autres mots, «**code source = programme + tests**».

En fait, certains auteurs suggèrent même d'utiliser une approche dite de «développement piloté par les tests» («*Test-Driven Development*» [Bec03]). Une telle approche, proposée initialement par les promoteurs de la *Programmation eXtrême* (XP = *eXtreme Programming* [Bec00, AMN02]), repose principalement sur la pratique *d'écrire les tests avant le programme* (*Test first*) :

- Les cas de tests devraient être développés et écrits (codés) **avant** le code lui-même!
- Du nouveau code ne devrait jamais être écrit s'il n'y a pas déjà un cas de test qui montre que ce code est nécessaire.

> *Never write a line of functional code without a broken test case. (K. Beck [Bec01])*

Cadres de tests

Qu'on utilise ou non une telle approche de développement piloté par les tests, il est malgré tout fondamental, lorsqu'on écrit un programme, de développer des tests appropriés qui vérifient son bon fonctionnement. De plus, il est aussi important que ces tests puissent être exécutés **fréquemment** et de façon **automatique** — ne serait-ce que pour assurer que tout fonctionne bien lorsque des modifications sont effectuées (tests de *non régression* du code).

De nos jours, il existe de nombreux outils qui permettent d'automatiser l'exécution des tests unitaires, qui facilitent le développement de tels tests et leur association au code testé, et ce peu importe le langage de programmation utilisé. Ces outils sont appelés des **cadres de tests** (*tests frameworks*).

Le cadre de tests le plus connu est `JUnit` [BG98, Bec01, HT03], popularisé par les promoteurs de l'approche XP (*eXtreme Programming*). Des cadres équivalents existent pour divers autres langages.[9] Dans ce qui suit, nous allons présenter un cadre de tests développé pour Ruby et maintenant intégré au noyau du langage : `MiniTest`.

Mais auparavant, il est utile de comprendre comment fonctionnent de tels cadres de tests. Tout d'abord, la caractéristique la plus importante de tous les cadres de tests est qu'on utilise des **assertions** pour décrire et spécifier ce qui doit être vérifié. En JUnit, de telles assertions ont la forme suivante et ont toujours la propriété que *rien n'est signalé si l'assertion est vérifiée* :

```
assertEquals( expectedResult, value )
assertEquals( expectedResult, value, precision )
assertTrue( booleanExpression )
assertNotNull( reference )
etc.
```

En d'autres mots, ceci implique qu'aucun résultat n'est produit ou généré si le test ne détecte pas d'erreur.

[9] Par exemple, voir `http://www.xprogramming.com/software.htm`.

D'autres caractéristiques des cadres de tests sont les suivantes :

- Ils permettent l'exécution automatique des tests — support pour les tests de (non-)régression, pour l'intégration continue, etc..
- Ils permettent l'organisation *structurée* des jeux d'essais (cas de tests, suites de tests, classes de tests).

  Plus spécifiquement :

  - Un *cas de tests* porte sur une fonctionnalité limitée, une instance particulière d'une méthode ou procédure/fonction.
  - Une *suite de tests* regroupe un certain nombre de cas de tests, qui représentent diverses instances liées à une même procédure ou groupe de procédures liées entre elles.
  - Une *classe de tests*, dans un langage objet, regroupe l'ensemble des suites de tests permettant de tester l'ensemble des fonctionnalités du module, c'est-à-dire de la classe. Un *programme de tests* joue le même rôle dans un contexte impératif et procédural (non orienté objets).
- Ils fournissent des mécanismes pour la construction *d'échafaudages* de tests — par exemple, `setUp`, `tearDown` en `JUnit` —, lesquels permettent de définir le contexte d'exécution d'une suite de tests.
- Ils fournissent des mécanismes permettant d'analyser les résultats de l'exécution des tests, ainsi que signaler clairement les cas problématiques.

B.2 Le cadre de tests `MiniTest`

Dans ce qui suit, nous présentons, toujours à l'aide d'exemples, les principaux éléments de `MiniTest`, le cadre de tests qui fait partie intégrante du noyau de Ruby (depuis la version 1.9). Pour plus de détails, voir le site Web suivant :

`http://ruby-doc.org/stdlib-2.0.0/libdoc/minitest/rdoc/MiniTest.html`

Tout d'abord, il faut souligner que `MiniTest` permet deux formes de tests :

- Des tests unitaires «classiques», avec des assertions, qui conduisent à des tests semblables à ce qu'on obtient en Java avec JUnit.

Dans cette forme, un programme de tests pour une méthode `bar` d'une classe `Foo` aurait l'allure suivante :

```
class TestFoo < MiniTest::Unit::TestCase
  def setup
    @foo = Foo.new
  end

  def test_bar_est_initialement_0
    assert_equal 0, @foo.bar
  end
  ...
end
```

- Des tests unitaires dits de «spécifications», avec des «*exceptations*», qui conduisent à des tests semblables à ce qu'on obtient avec RSpec [CAD+10], un `gem` Ruby qui définit un langage de tests — un langage-spécifique au domaine[10] pour les tests.

  Dans cette forme, un programme de tests pour une méthode `bar` d'une classe `Foo` aurait l'allure suivante :

```
describe Foo do
  describe "#bar" do
    before do
      @foo = Foo.new
    end

    it "retourne une taille nulle lorsque cree" do
      @foo.bar.must_equal 0
    end
   ...
  end
  ...
end
```

C'est cette dernière forme que nous allons présenter dans les exemples qui suivent.

[10] DSL = *Domain Specific Language* [Fow11].

B.3 Des spécifications `MiniTest` pour la classe `Ensemble`

Exemple Ruby 72 Une suite de tests pour la classe `Ensemble` (partie 1)

```
require 'minitest/autorun'
require 'minitest/spec'

require_relative 'ensemble'

describe Ensemble do
  before do
    @ens = Ensemble.new
  end

  describe '#contient?' do
    it "retourne faux quand un element n'est pas present" do
      refute @ens.contient? 10
    end

    it "retourne vrai apres qu'un element ait ete ajoute" do
      refute @ens.contient? 10
      @ens << 10
      assert @ens.contient? 10
    end
  end

  # ...
```

Exemple Ruby 73 Une suite de tests pour la classe `Ensemble` (partie 2)

```ruby
# ...

describe '#<<' do
  it "ajoute un element lorsque pas deja present" do
    @ens << 10
    assert @ens.contient? 10
  end

  it "laisse l'element ajoute lorsque deja present" do
    @ens << 10
    assert @ens.contient? 10

    @ens << 10
    assert @ens.contient? 10
  end

  it "retourne self ce qui permet de chainer des operations" do
    res = @ens << 10
    res.must_be_same_as @ens
  end
end

# ...
```

Exemple Ruby 74 Une suite de tests pour la classe `Ensemble` (partie 3)

```
  # ...

  describe '#cardinalite' do
    it "retourne 0 lorsque vide" do
      @ens.cardinalite.must_equal 0
    end

    it "retourne 1 lorsqu'un seul et meme element est ajoute, 1 ou plusieurs
      @ens << 1
      @ens.cardinalite.must_equal 1

      @ens << 1 << 1 << 1
      @ens.cardinalite.must_equal 1
    end

    it "retourne le nombre d'elements distincts peu importe le nombre de fois
      @ens << 1 << 1 << 1 << 2 << 2 << 1 << 2
      @ens.cardinalite.must_equal 2
    end
  end
end
```

Les exemples Ruby 72–74 présentent une suite de tests pour la classe `Ensemble` — voir l'exemple Ruby 41 (p.-84) pour la mise en oeuvre de cette classe.

Remarques et explications pour les exemples Ruby 72–74 :

- Le `describe` de niveau supérieur indique généralement le nom de la classe testée — c'est une règle de style, pas une règle syntaxique.

  Les `describe`s internes indiquent quant à eux les noms des méthodes. Un nom de méthode tel que «`#foo`» indique une série de cas de tests pour la méthode d'instance `foo`, alors qu'un nom tel que «`.foo`» indique une série de cas de tests pour la méthode de classe `foo`.

- Le bloc de code indiqué par `before` sera exécuté *avant chaque cas de test*. Il sert à définir le contexte d'exécution de chacun des tests. Ici, pour éviter la duplication de code, on alloue un nouvel objet `Ensemble`, initialement vide, qu'on affecte à la variable `@ens`. Puisque cette variable est une variable d'instance du test, elle sera accessible dans chacun des cas de test.

- Chaque appel à `it` décrit un *cas de test* spécifique, qui ne devrait tester qu'une et une seule chose. Règle générale (règle de style, pas syntaxe), il ne devrait y avoir qu'une seule assertion (*expectation*) par tests.

- Une assertion telle que «`assert` *expr*» affirme que l'expression *expr* est vraie. Si c'est le cas, le test réussit — donc, en mode non verbeux, un «.» sera affiché. Par contre, si *expr* est fausse, alors le test échoue et un message d'erreur approprié sera affiché — voir plus bas.

  Une assertion telle que «`refute` *expr*» affirme que l'expression *expr* est fausse — dont «`refute` *expr*» est équivalent à «`assert !`*expr*».

- La clause «`res.must_be_same_as @ens`» affirme que `res` et `@ens` dénotent en fait le même objet. Cette clause est donc équivalente à la suivante :

  ```
  assert res.equal? @ens
  ```

- La clause «`@ens.cardinalite.must_equal 0`» est équivalente à l'une ou l'autre des clauses suivantes :

  ```
  assert @ens.cardinalite == 0
  assert_equal 0, @ens.cardinalite
  ```

  La deuxième clause serait celle utilisée dans la forme classique de descriptions des cas de tests (à la JUnit), qui distingue clairement entre le résultat attendu (premier argument) et le résultat obtenu (deuxième argument). L'avantage d'utiliser `must_equal` ou `assert_equal` plutôt qu'un simple `assert` est que le message d'erreur résultant est plus clair et explicite :

  ```
  ============================
  Avec simple assert
  ============================
  1) Failure:
  Ensemble::#cardinalite#test_0003_retourne [...] [ensemble_spec.rb:61]:
  Failed assertion, no message given.

  ============================
  Avec must_equal/assert_equal
  ============================
  1) Failure:
  Ensemble::#cardinalite#test_0003_retourne [...] [ensemble_spec.rb:61]:
  Expected: 2
    Actual: 0
  ```

Exemple Ruby 75 Des exemples d'exécution de la suite de tests pour la classe `Ensemble`.

```
======================
Execution ordinaire
======================

$ ruby ensemble_spec.rb
Run options: --seed 43434

# Running:

........

Finished in 0.001556s, 5140.4367 runs/s, 7068.1005 assertions/s.

8 runs, 11 assertions, 0 failures, 0 errors, 0 skips

----------------------------------------------------------------

======================
Execution 'verbeuse'
======================

$ ruby ensemble_spec.rb -v
Run options: -v --seed 18033

# Running:

Ensemble::#<<#test_0003_retourne self ce qui permet de chainer des operations = 0.00 s = .
Ensemble::#<<#test_0001_ajoute un element lorsque pas deja present = 0.00 s = .
Ensemble::#<<#test_0002_laisse l'element ajoute lorsque deja present = 0.00 s = .
Ensemble::#contient?#test_0001_retourne faux quand un element n'est pas present = 0.00 s = .
Ensemble::#contient?#test_0002_retourne vrai apres qu'un element ait ete ajoute = 0.00 s = .
Ensemble::#cardinalite#test_0001_retourne 0 lorsque vide = 0.00 s = .
Ensemble::#cardinalite#test_0002_retourne 1 lorsqu'un seul et meme element est ajoute,\
          1 ou plusieurs fois = 0.00 s = .
Ensemble::#cardinalite#test_0003_retourne le nombre d'elements distincts peu importe\
          le nombre de fois ajoutes = 0.00 s = .

Finished in 0.001686s, 4745.7382 runs/s, 6525.3900 assertions/s.

8 runs, 11 assertions, 0 failures, 0 errors, 0 skips
```

Remarques et explications pour l'exemple Ruby 75 :

- L'exécution en mode ordinaire affiche simplement, comme en JUnit, un «.» pour chaque cas de test exécuté — chaque utilisation de la méthode de test «`it`» — suivi d'un sommaire d'exécution indiquant le nombre de tests exécutés (`8 runs`), le nombre d'assertions évaluées (`11 assertions`), le nombre d'échecs (i.e., de tests pour lesquels certaines assertions n'étaient pas valides) et d'erreurs (erreurs d'exécution), etc.
- L'exécution en mode «verbeux» (option d'exécution «`-v`») affiche, pour chaque cas de test, le nom du test et le temps d'exécution, suivi du même sommaire.

  On remarque que le nom complet d'un test est formé de la concaténation des identificateurs et chaînes des `describe` englobant, suivi de «`#test_`», suivi d'un numéro unique au `describe` courant, suivi de la chaîne utilisée comme argument à `it`.

Exemple Ruby 76 Un exemple d'exécution de la suite de tests pour la classe `Ensemble` avec des échecs — la méthode `cardinalite` retourne toujours `0`.

```
======================
Execution avec echecs
======================
$ ruby ensemble_spec.rb
Run options: --seed 7910

# Running:

...FF...

Finished in 0.001950s, 4101.7438 runs/s, 5127.1797 assertions/s.

  1) Failure:
Ensemble::#cardinalite#test_0002_retourne 1 lorsqu'un seul et meme element est ajoute,\
          1 ou plusieurs fois [ensemble_spec.rb:54]:
Expected: 1
  Actual: 0


  2) Failure:
Ensemble::#cardinalite#test_0003_retourne le nombre d'elements distincts peu importe\
          le nombre de fois ajoutes [ensemble_spec.rb:62]:
Expected: 2
  Actual: 0

8 runs, 10 assertions, 2 failures, 0 errors, 0 skips
```

Remarques et explications pour l'exemple Ruby 76 :

- Dans cet exemple d'exécution, la méthode `cardinalite` a été modifiée pour toujours retourner `0`. On voit alors, en cours d'exécution, que certains cas de tests *échouent* — un «`F`» est affiché plutôt qu'un «.». Les détails des tests échoués sont ensuite affichés.

La figure 8 (p. 151) présente la liste détaillée des *expectations* de `MiniTest`.

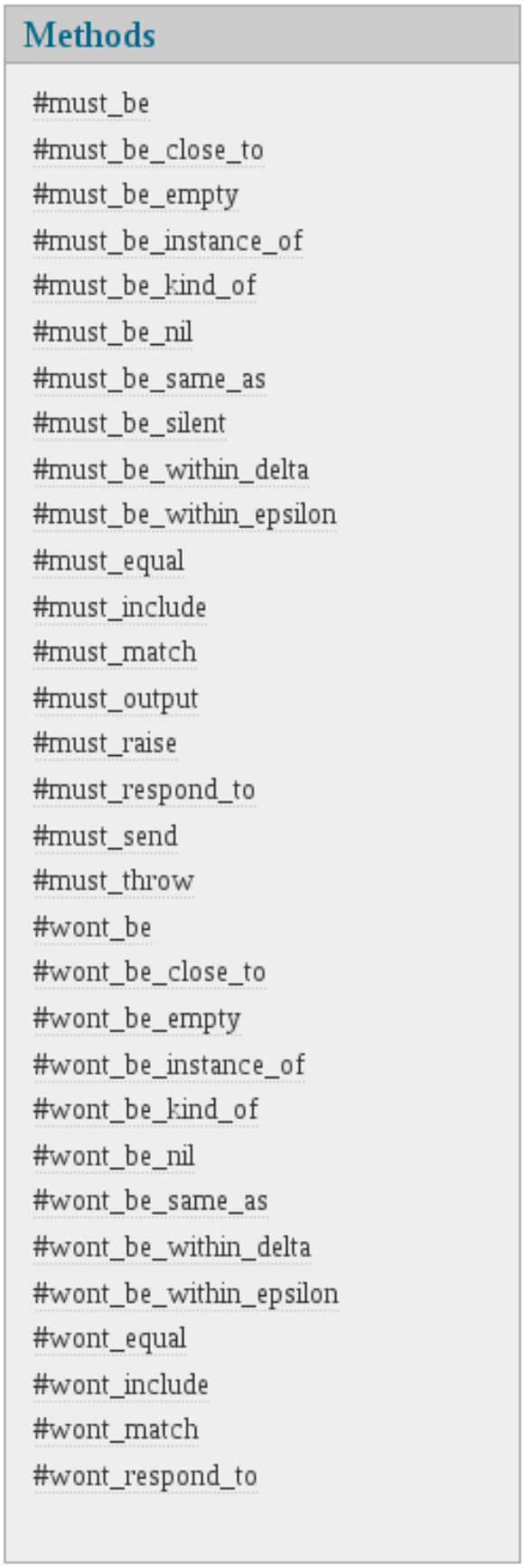


Figure 8: La liste des *expectations* disponibles dans `MiniTest`. Source : `http://ruby-doc.org/stdlib-2.1.0/libdoc/minitest/rdoc/MiniTest/Expectations.html`.

Exemple Ruby 77 Quelques autres méthodes de `MiniTest` — dans le style avec *expectations*.

```
gem 'minitest'
require 'minitest/autorun'
require 'minitest/spec'

describe Array do
  let (:vide) { Array.new }

  before do
    @singleton_10 = Array.new << 10
  end

  describe ".new" do
    it "cree un tableau vide lorsque sans argument" do
      vide.must_be :empty?
    end
  end

  describe "#push" do
    it "ajoute un element, lequel devient inclu" do
      @singleton_10.must_include 10
    end
  end
```

```
  describe "#size" do
    it "retourne 0 lorsque vide" do
      vide.size.must_equal 0
    end

    it "retourne 0 lorsque vide (bis)" do
      vide.size.must_be :==, 0
    end

    it "retourne > 0 lorsque non vide" do
      @singleton_10.size.must_be :>, 0
    end
  end
```

```
  describe "#to_s" do
    it "retourne '[]' lorsque vide" do
      vide.to_s
        .must_equal "[]"
    end

    it "retourne les elements separes par des virgules" do
      (vide << 10 << 20 << 30).to_s
          .must_equal "[10, 20, 30]"
    end

    it "retourne les elements separes par des virgules (bis)" do
      a = vide << 10 << 20 << 30
      virgule = /\s*,\s*/

      a.to_s
        .must_match
          /^\[\s*10#{virgule}20#{virgule}30\s*\]$/
    end
  end
end
```

Remarques et explications pour l'exemple Ruby 77 :

- Dans l'étape de *setup* des tests, lorsqu'on veut définir des objets pouvant être utilisés dans plusieurs cas de tests, on peut utiliser deux approches :

  - Objet défini avec une expression simple : on peut utiliser `let`, qui reçoit comme argument un `Symbol` (arg. explicite) et un bloc (arg. implicite). On peut ensuite utiliser directement l'identificateur dans les tests.
  - Objet plus complexe (ou grand nombre d'objets) : on utilise la méthode `before`. Dans ce cas, les objets sont des variables d'instance du test, donc doivent leurs noms doivent être précédés du *sigil* «`@`».

  Dans les deux formes, les objets ainsi définis sont évalués/recréés **pour chaque cas de test**, et ce dans le but d'assurer l'indépendance de chacun des tests.

- La méthode `must_be` prend comme premier argument un symbole dénotant un nom de méthode, laquelle sera appelée sur l'objet testé. Un deuxième argument peut aussi être fourni si la méthode associée au symbole prend un argument.

- La méthode `must_match` est utile pour vérifier la représentation textuelle d'objets, en vérifiant certains élements essentiels et en ignorant des détails secondaires — dans l'exemple, on ignore le fait qu'il pourrait y avoir 0, 1 ou plusieurs blancs avant/après les virgules, mais on veut que les virgules soient présentes **entre** les éléments du tableau.

C Règles de style Ruby

Plusieurs auteurs présentent des «règles de style» pour Ruby — donc décrivent comment écrire du «beau code» Ruby. Une de ces présentations, assez complète, est disponible à l'adresse suivante :

`https://github.com/styleguide/ruby`

Les principales règles de style Ruby, inspirées de cette référence, qui ont généralement été utilisées dans les exemples et **que vous devrez respecter** sont les suivantes :

- Indentation de seulement deux (2) espaces, *avec des blancs*, jamais des caractères de tabulation.
- Jamais de blancs à la fin d'une ligne.
- Des blancs autour des opérateurs binaires (y compris =), après les virgules, les deux points et les points-virgules, autour des { et avant les }.
- Pas de blanc avant ou après [et], ou après !.
- Jamais de `then` pour une **instruction** `if`/`unless` et pas de parenthèses autour des conditions (on est en Ruby, pas en Java ou C!) :

```ruby
# NON
if ( condition ) then
  ...
end
```

```ruby
# OK
if condition
  ...
end
```

- Pas de parenthèses pour une définition de méthode si aucun argument — idem pour l'appel :

```ruby
def une_methode_sans_arg
   ...
end

def une_methode_avec_args( arg1, ..., argk )
   ...
end

# NON
une_methode_sans_arg()

# OK
une_methode_sans_arg
```

- Opérateur ternaire `?:` seulement pour une **expression** conditionnelle simple, non imbriquée, tenant *sur une seule ligne*.
- On utilise une garde `if/unless` quand il y a une seule instruction :

```
# NON
if condition
  une_instruction...
end
```

```
# OK
une_instruction if condition
```

- On utilise `unless` si la condition est négative (idem pour les gardes)... mais on n'utilise pas `unless/else` :

```
# NON
if !expr
  ...
  res
end
```

```
# OK
unless expr
  ...
  res
end
```

```
# NON
unless expr
  ... si faux ...
else
  ... si vrai ...
end
```

```
# OK
if expr
  ... si vrai ...
else
  ... si faux ...
end
```

- Pour les blocs, on utilise {...} lorsque le corps peut s'écrire sur une seule ligne. Autrement, on utilise `do ... end`.
- On utilise `return` seulement pour retourner un résultat au milieu d'une méthode, pas lorsque l'expression est la dernière évaluée dans la méthode :

```
# NON
if expr
  ...
  return res
else
  ...
  return autre_res
end
```

```
# OK
if expr
  ...
  res
else
  ...
  autre_res
end
```

```
# NON
def m_rec( ... )
  if expr
    return res_base
  else
    ...
    return res_rec
  end
end
```

```
# OK
def m_rec( ... )
  return res_base if expr

  ...
  res_rec
end
```

- Dans la définition d'une classe `C`, on utilise `def self.m` pour définir une méthode de classe `m`, plutôt que `def C.m`.

- Pour les objets de classe `Hash`, on utilise généralement des `Symbol`s comme clés. Et on utilise la forme avec «=>» :

```
hash = {
  :cle1 => defn1,
  :cle2 => defn2,
  ...
  :clek => defnk
}
```

Quelques remarques additionnelles concernant le style utilisé dans les exemples :

- Des espaces sont mis autour des parenthèses des définitions de méthodes, contrairement à ce qui est suggéré dans ce guide :

```
# Style suggere dans le guide.
def methode(a, b, c)
  ...
end

# Style dans le materiel de cours
def methode( a, b, c )
  ...
end
```

Quelques règles additionnelles

Les règles de style qui suivent sont basées sur des erreurs typiques rencontrées lors de la correction de devoirs.

- Les méthodes `map` (`collect`), `select` (`find_all`), `reject` doivent être utilisées *pour produire une nouvelle collection*, et non pour leurs effets de bord.

  Notamment, voici un exemple *à ne pas faire* s'il est demandé d'utiliser *le style fonctionnel* :

  ```
  res = []
  a.map { |x| res << foo(x) }  # Faux map!
  ```

  Dans cet exemple, le résultat produit par le `map` n'est pas utilisé. Le `map` est en fait utilisé comme «un faux `each`», donc n'est pas dans un style fonctionnel.

- On utilise une instruction avec garde — `if` ou `unless` après l'instruction — seulement si l'instruction «*fitte*» facilement sur une seule ligne :

  ```
  instruction if condition  # OK si instruction est courte.
  ```

  Si l'instruction est trop longue pour la ligne, alors on utilise la forme avec une *instruction* `if` :

  ```
  if condition
    instruction
  end
  ```

- Il faut éviter les effets de bord dans les gardes — i.e., la condition d'une garde ne devrait rien modifier :

  ```
  puts x if x = ARGV.shift    # NON!
  ```

  Dans certains cas simples, on peut accepter une affectation en début d'une instruction `if` si l'effet de bord est bien *visible* au tout début du code :

  ```
  if x = ARGV.shift
    puts x
  end
  ```

- On utilise une instruction avec garde *seulement si le cas complémentaire n'a pas besoin d'être traité*, par exemple, si on peut retourner un résultat (ou signaler une erreur) de façon *immédiate*. Donc, le segment de code qui suit n'est pas approprié (NON!) :

  ```
  instruction1 if condition
  instruction2 unless condition  # NON!
  ```

Dans un tel cas, on utilise plutôt une instruction `if` :

```
if condition
  instruction1
else
  instruction2
end
```

- Il est correct d'*enchainer* plusieurs appels de méthodes. Toutefois, si l'instruction résultante est longue, alors on met les appels sur plusieurs lignes :

```
# OK seulement si *tres* court
res = a.select { |x| ... }.map { |x| ... }.sort.join

# Preferable si relativement long ---
# plus facile a lire, modifier, ajouter un autre appel, etc.
res = a.select { |x| ... }
       .map { |x| ... }
       .sort
       .join
```

D Méthodes `attr_reader` et `attr_writer`

Exemple Ruby 78 Une définition des méthodes `attr_reader` et `attr_writer`.

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

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

class Foo
  attr_reader :bar
  attr_writer :bar

  def initialize
    self.bar = 0
  end
end

foo = Foo.new
foo.bar += 3
```

Exemple Ruby 79 Une autre définition des méthodes `attr_reader` et `attr_writer`.

```ruby
class Class
  def attr_reader_( attr )
    self.class_eval do
      define_method attr do
        instance_variable_get "@#{attr}"
      end
    end
  end

  def attr_writer_( attr )
    self.class_eval do
      define_method "#{attr}=" do |v|
        instance_variable_set( "@#{attr}", v )
      end
    end
  end
end

class Foo
  attr_reader :bar
  attr_writer :bar

  def initialize
    self.bar = 0
  end
end
```

L'exemple Ruby 78 montre une définition possible des méthodes `attr_reader` et `attr_writer`. Comme c'est souvent le cas en Ruby, il y a plusieurs façons différentes d'obtenir le même résultat. L'exemple Ruby 79 montre donc une autre définition possible de ces mêmes méthodes.

E Interprétation vs. compilation

Soit l'affirmation suivante : «Ruby est un langage interprété».

Cette affirmation est-elle vraie ou fausse?

Exercice 12: Ruby, un langage interprété?

Pourquoi les performances d'un programme Ruby sont-elles généralement moins bonnes (programme plus lent ☹) que celles d'un programme Java?

Exercice 13: Performances de Ruby.

Références

[AMN02] D. Astels, G. Miller, and M. Novak. *A Practical Guide to eXtreme programming.* The Coad Series. Prentice-Hall PTR, 2002.

[Bec00] K. Beck. *Extreme Programming Explained—Embrace Change.* Addison-Wesley, 2000.

[Bec01] K. Beck. Aim, fire. *IEEE Software*, 18(6):87–89, 2001.

[Bec03] K. Beck. *Test-Driven Development—By Example.* Addison-Wesley, 2003.

[BG98] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[Bla04] C. Blaess. *Scripts sous Linux—Shell Bash, Sed, Awk, Perl, Tcl, Tk, Python, Ruby.* Eyrolles, 2004.

[CAD+10] D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North. *The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends.* The Pragmatic Bookshelf, 2010.

[Dix11] P. Dix. *Service-Oriented Design with Ruby and Rails.* Addison-Wesley Professional Ruby Series, 2011.

[Fow11] M. Fowler. *Domain-Specific Languages.* Addison-Wesley, 2011.

[Gol89] A. Goldberg. *Smalltalk-80: The Language.* Addison-Wesley, 1989.

[Har13] M. Hartl. *Ruby on Rails Tutorial (Second Edition).* Addison-Wesley Professional Ruby Series, 2013.

[HT03] A. Hunt and D. Thomas. *Pragmatic Unit Testing In Java with JUnit.* The Pragmatic Bookshelf, 2003.

[Lew15] A. Lewis. *Rails Crash Course—A No-Nonsense Guide to Rails Development.* No Starch Press, 2015.

[LG86] B. Liskov and J. Guttag. *Abstraction and specification in program development.* MIT Press, 1986.

[RK03] P.N. Robillard and P. Kruchten. *Software Engineering Process with the UP-EDU.* Addison-Wesley, 2003.

[RTH13] S. Ruby, D. Thomas, and D.H. Hansson. *Agile Web Development with Rails 4.* THe Pragmatic Bookshelf, 2013.

[Ste84] G.L. Steele Jr. *Common LISP: The Language.* Digital Press, 1984.

[WCS96] L. Wall, T. Christiansen, and R.L. Schwartz. *Programming Perl (2nd Edition).* O'Reilly, 1996.