Un livre de Wikilivres.

Programmation Ruby

Introduction

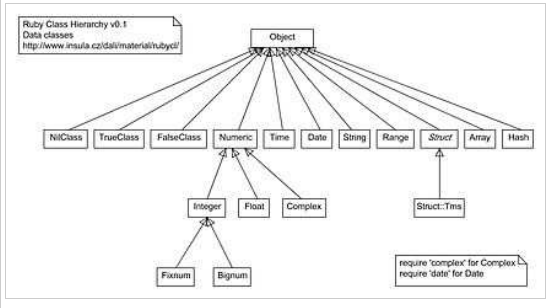
Ruby, ou la langue rouge, est un langage interprété comparable à Perl, Python et Smalltalk pour son approche objet.

Il est capable de gérer des expressions régulières en langue japonaise, cette langue ayant la caractéristique de regrouper trois systèmes graphiques différents, quatre si l'on compte les caractères romains de plus en plus utilisés pour les mots d'origine étrangère.

De plus, Ruby est complètement objet alors que les aspects objets de Python ne sont pas intrinsèques au langage.

Ruby fut développé dans l'idée de pouvoir programmer tout en restant concentré sur le côté créatif du développement, ainsi ruby se veut aussi souple que Perl, tout en restant plus cohérent, notamment via son approche entièrement objet (y compris les types primaires), à l'exception des expressions de contrôle (if-then-else, for,...). Le langage est conçu sur le principe de moindre surprise (PoLS : Principle of Least Surprise), rendant le langage plus "sûr" que de nombreux autres langages de script.

Ruby est interprété, ce qui en fait un outil de développement rapide, en contrepartie d'un certain relâchement des performances. Ces nombreux avantages font qu'il connait un grand succès au Japon (son pays d'origine), et gagne petit à petit du terrain, notamment dans le domaine de la recherche.



Installation

Linux

Sur la plupart des systèmes Linux ou BSD, Ruby est inclus dans la distribution, référez-vous à sa documentation pour savoir comment installer de nouveaux packages.

Paquet logiciel
Ubuntu (? (http://doc.ubunt
Gentoo (? (http://www.gentoo.o

Veuillez à installer tout le nécessaire :

- ruby
- irb (un interpréteur interactif)
- ri qui permet d'accéder à de la documentation
- rubygems qui permet de télécharger et d'installer simplement des bibliothèques à la manière d'un gestionnaire de paquet

... et le superflu :

- rubybook, un livre sur Ruby en anglais

Windows

Le plus simple est de télécharger et lancer (en tant qu'administrateur) l'installeur sur http://rubyinstaller.org/.

Mais par ailleurs les sources sont accessibles ici (http://www.ruby-lang.org/fr/downloads/).

> **Remarque :** il est conseillé de se faire un raccourci de `C:\Program Files (x86)\Ruby200\bin\irb.bat`.

Mac OS X

Mac OS X Leopard intègre Ruby d'origine. Pour les versions précédentes du système, il existe comme pour Windows un One-Click Installer (http://rubyosx.rubyforge.org/)

Autres systèmes

Si Ruby supporte votre système vous pouvez télécharger les sources à partir de cette url :

ftp://ftp.ruby-lang.org/pub/ruby/stable-snapshot.tar.gz

Premiers essais

Pour faire les premiers essais avec **Ruby**, nous allons utiliser l'outil *irb*, un interpréteur **Ruby** très pratique pour tester et déboguer.

Pour le lancer sous Windows un raccourci a été défini au chapitre précédent, et avec Unix à partir d'un terminal via la commande *irb* :

```
# irb
```

Pour ne pas déroger à la règle, nous allons pour premier exemple dire bonjour :

```
irb(main):001:0> puts "Hello World"
```

Ceci renverra :

```
Hello World
=> nil
```

soit l'effet produit par notre fonction (l'affichage de la chaîne "*Hello World*"), et la valeur de retour de celle-ci, dans notre cas *nil*.

Pour un langage objet, cet exemple n'est pas très adapté. Il faut néanmoins savoir que tout code écrit en dehors d'une classe ou d'un module fait partie de la classe *Object*.

On aurait également pu écrire cet exemple de cette manière :

```
irb(main):013:0> STDOUT << "Hello" << " " + "World"
```

L'approche objet est ici plus évidente : la méthode "<<" de l'objet *STDOUT* est appelée avec " *"Hello"* << " " + *"World"*" comme paramètre. De même on appelle la méthode << de l'objet *"Hello"* (qui est une instance de l'objet String).

On découvre ici le modèle tout objet de Ruby. En effet, celui-ci ne comporte pas de type primaire, ainsi, une chaîne de caractères ou un nombre sont des objets, comme on peut le voir avec la méthode Object.class :

```
irb(main):016:0> 3.class
=> Fixnum
irb(main):017:0> "Hello".class
=> String
irb(main):018:0> /^Hello\sWorld$/.class
=> Regexp
```

Bien sûr, chacun de ces objets possède ses propres méthodes, que l'on peut lister ainsi :

```
irb(main):019:0> Fixnum.methods
=> ["method", "send", "name", "class_eval", "object_id", "singleton_methods", "__send__",
"private_method_defined?", "equal?", "taint", "frozen?", "instance_variable_get", "constants",
"kind_of?",  "to_a", "instance_eval", "ancestors", "const_missing", "type", "instance_methods",
"protected_methods", "extend", "protected_method_defined?", "eql?", "public_class_method", "const_get",
"display", "instance_variable_set", "hash", "is_a?", "to_s", "class_variables", "class", "tainted?",
"private_methods", "public_instance_methods", "autoload", "untaint", "included_modules",
"private_class_method", "const_set", "id", "<", "inspect", "<=>", "instance_method", "==",
"induced_from", "method_defined?", ">", "===", "clone", "public_methods", "protected_instance_methods",
">=", "respond_to?", "freeze", "<=", "module_eval", "allocate", "__id__", "=~", "methods",
"public_method_defined?", "superclass", "nil?", "dup", "autoload?", "private_instance_methods",
"instance_variables", "include?", "const_defined?", "instance_of?"]
```

ou encore dans un contexte précis :

```
irb(main):020:0> 3.methods
=> ["%", "between?", "method", "send", "<<", "prec", "modulo", "&", "object_id", ">>", "zero?", "size",
"singleton_methods", "__send__", "equal?", "taint", "id2name", "*", "next", "frozen?",
"instance_variable_get", "+", "kind_of?", "step", "to_a", "instance_eval", "-", "remainder", "prec_i",
"nonzero?", "/", "type", "protected_methods", "extend", "floor", "to_sym", "|", "eql?", "display", "quo",
"instance_variable_set", "~", "hash", "is_a?", "downto", "to_s", "prec_f", "abs",
"singleton_method_added", "class", "tainted?", "coerce", "private_methods", "^", "ceil", "untaint", "+@",
"upto", "-@", "div", "id", "**", "times", "to_i", "<", "inspect", "<=>", "==", ">", "===", "succ",
"clone", "public_methods", "round", ">=", "respond_to?", "<=", "freeze", "divmod", "chr", "to_f",
"__id__", "integer?", "=~", "methods", "nil?", "dup", "to_int", "instance_variables", "[]",
"instance_of?", "truncate"]
```

On constate certaines particularités syntaxiques ; nous les détaillerons plus tard même si elles sont assez claires ; par exemple une méthode dont le nom se termine par *?* indique que celle-ci renvoie un booléen.

Pour plus d'informations sur un objet ou une méthode, vous pouvez, à partir d'un terminal, exécuter la commande *ri* :

```
# ri times
```

Si une méthode est incluse dans plusieurs objets, *ri* vous proposera une liste de choix possibles.

Toute méthode peut être redéfinie :

```
irb(main):029:0> 2+3
=> 5
```

```
irb(main):030:0> class Fixnum
irb(main):031:1> def +(value)
irb(main):032:2> return self-value
irb(main):033:2> end
irb(main):034:1> end

irb(main):035:0> 2+3
=> -1
```

Toutes les méthodes peuvent ainsi être redéfinies, cependant, cette fonctionnalité est à utiliser avec précaution.

Pour finir cette présentation nous allons voir la notion de bloc en **Ruby**, et voir comment ceux-ci peuvent être utilisés. Par exemple :

```
irb(main):024:0> 5.times do |i|
irb(main):025:1* puts "#{i}\n"
irb(main):026:1> end
0
1
2
3
4
```

Ici nous voyons que le bloc délimité par *do* et *end* (Il est à noter que l'on peut également délimiter les blocs par des accolades ouvrantes et fermantes **`{ }`**) est envoyé en paramètre à la méthode *times* de l'objet *Fixnum*. Les valeurs entre pipes (ici la variable *i*) indiquent les paramètres que la méthode passe au bloc. Ici, la méthode times va donc appeler 5 fois le bloc en y passant en paramètre un entier valant au départ 0 et qui sera incrémenté de 1 à chaque appel du bloc.

Tous ces concepts seront vus en détail dans les chapitres suivants.

Syntaxe

Syntaxe du langage

La syntaxe de ruby est à la fois simple, car elle permet de lire simplement le code source, et complexe, car à la manière du perl il y a plusieurs manières d'écrire une même instruction.

Nomenclatures

Notons que par nomenclature les méthodes terminant par un point d'exclamation *!* sont non pures : elles modifient l'objet. Les méthodes terminant par un point d'interrogation *?* renvoient un *booléen* (*vrai* ou *faux*)

Identifiants

Tout nom (que cela soit pour les variables, méthodes, classes...) doit respecter une certaine nomenclature : tout identifiant doit commencer soit par une lettre, soit par un souligné (_), et bien sur ne doit pas être un des mots réservés du langage.

Exemples :

```
MaVariables    => Ok
_maVariable    => Ok
3Variables     => Erreur
```

Commentaires

En Ruby, les commentaires peuvent prendre deux formes. La plus commune insère un commentaire sur une seule ligne et débute par le caractère dièse ("#"). Le reste de la ligne est alors considéré comme un commentaire.

```
#Ceci est un commentaire

# ceci est un
# bloc de
# commentaire

puts "toto"  # ce commentaire suit une instruction
```

Une seconde forme permettant d'insérer des commentaires sur plusieurs lignes est plutôt réservée à l'écriture de documentation. Il s'agit de délimiter les lignes de commentaires par une ligne "=begin" et une ligne "=end".

```
=begin
voici un commentaire
sur plusieurs lignes
utilisant la seconde forme
=end
```

Typage de canard

Si vous avez déjà utilisé des langages typés tels que C ou Java, vous êtes familier de la notion de "type". Ces langages attribuent à chaque variable un "type" c'est à dire un ensemble de choses qu'elle est capable de faire. Lorsqu'une variable a un type donné, le langage considère qu'on ne peut pas demander autre chose à celle-ci que ce que son type l'y autorise. Ainsi les langages typés détectent une erreur de programmation dans un programme avant même de l'utiliser en vérifiant simplement que ce que l'on demande à chaque variable est bien à la portée de son type.

Ruby est fondé sur l'approche inverse, il fait l'hypothèse qu'à priori, on peut demander n'importe quoi à une variable et qu'il détectera une erreur seulement lorsqu'une variable ne sera pas en mesure de faire ce qu'on lui demande au moment où on lui demande. C'est en cela que réside tout l'intérêt de ce langage: il autorise les variables à acquérir ou perdre des fonctionnalités au cours de leur existence. Pour cette raison, ceux habitués au langages typés seront surpris de constater que les paramètres des méthodes n'ont pas de type en Ruby: cette notion n'existe simplement pas.

Ce mécanisme est parfois nommé "typage de canard" (Duck Typing) et résumé ainsi: "si ça a des plumes et que ça fait 'coincoin' alors c'est sûrement un canard". Illustrons cela avec un exemple:

```
# le canard, si on le lui demande
# gentiment, sait faire coincoin
class Canard
 def faire_coincoin
  puts "coincoin"
 end
end

# l'humain parle (trop)
class Humain
 def parle
  puts "bla bla"
 end
end

# un canard:
canard = Canard.new
# un humain
humain = Humain.new
# un imitateur de canard !
imitateur = Humain.new
```

```ruby
# la puissance de Ruby
def imitateur.faire_coincoin
 puts "coin, coin !"
end

# et maintenant voici le typage de canard:
canard.faire_coincoin # => "coincoin"
imitateur.faire_coincoin # => "coin, coin!"
humain.faire_coincoin # provoque une erreur
```

Comme vous pouvez le constater, même si les humains ne font pas coincoin d'après la classe Humain, certains peuvent apprendre et Ruby laissera faire les imitateurs de canards.

Ce mécanisme, permet d'enrichir certains objets comme nous venons de le voir, mais il permet surtout d'écrire du code générique (réutilisable) sans trop effort:

```ruby
def une_fonction(parametre)
 parametre << "toto"
end

# utilisons une_fonction avec différents paramètres
# définissant chacun l'opérateur "<<"
a = [1,2,3] # un tableau
une_fonction(a) # => [1,2,3,"toto"]
s = "une bonne blague de " # une chaîne de caractères
une_fonction(s) # => "une bonne blague de toto"
f = File.new("fichier","w") # un fichier
une_fonction(f) # ajoute "toto" à la fin du fichier
```

Portée et syntaxe des variables

La portée des différentes variables est définie par leur syntaxe :

Variable locale

Une variable locale doit être nommée avec pour premier caractère soit une minuscule, soit un caractère souligné (*underscore*).

exemple :

```ruby
maVariable
_variable
i4
```

La portée d'une variable locale est le bloc courant, sauf si elle est définie en dehors du bloc :

```ruby
variable = "En dehors du bloc"
puts variable                  #     => En dehors du bloc
begin
  variable = "Dans le bloc"
  puts variable                #     => Dans le bloc
end
puts variable                  #     => En dehors du bloc
```

Variable globale

Une variable globale doit être préfixée avec dollar (*$*) comme premier caractère :

exemple:

```ruby
$maVariableGlobale
$VARIABLE
$_VAR
```

Comme son nom l'indique, une variable globale est accessible dans tout le programme :

```ruby
$maGlobale = 3
puts $maGlobale             #        => 3

class A
 $maGlobale = 8

 def initialize
   puts $maGlobale          #       => 8
 end
end

A.new
puts $maGlobale             #       => 8
```

Attribut ou variable d'instance

Une variable d'instance est une variable qui n'est accessible qu'après l'instanciation d'un objet. La variable sera alors accessible en utilisant cette syntaxe : *<Nom de l'instance>.<Nom de la variable>*. Son nom doit être préfixé avec le caractère arobase (*@*)

Exemple :

```ruby
class A
```

```
 attr_reader :variableInstance
 def initialize
   @variableInstance = 42
 end

 def to_s
   return @variableInstance.to_s #Correspond a self.variableInstance.to_s
 end
end

puts variableInstance       #      => Erreur
puts A::variableInstance    #      => Erreur
test=A.new
puts test.variableInstance  #      => 42
puts test.to_s              #      => 42
```

Attributs ou variable de classe

Une variable de classe est commune à toutes les instances d'une même classe. Son nom est préfixé par deux arobases (@@).

Exemple :

```
class A
 @@variabeDeClasse = 0

 def initialize
   @@variableDeClasse += 1
   @variableDeClasse = 5  #Ceci n'est pas une variable de classe, c'est juste pour l'exemple
 end

 def nombreInstance
   return @@variableDeClasse
 end

 def variableDeClasse
   return @variableDeClasse
 end
end

test1 = A.new
puts test1.nombreInstance    #   => 1
puts test1.variableDeClasse  #   => 5
test2 = A.new
puts test1.nombreInstance    #   => 2
puts test1.variableDeClasse  #   => 5
```

Constantes

Une constante définit un élément qui ne pourra jamais changer au cours de l'exécution du programme. Une constante débute toujours par une majuscule, mais vous pouvez n'utiliser que des majuscules pour les identifier des noms de classe dans votre code

```
MACONSTANTE = 42
puts MACONSTANTE      #       => 42
MACONSTANTE = 18      #       => Erreur

POIDS_747 = 1234
POIDS_747 = 0     # => Erreur

Maconstante  = 9786
```

***Note** : En Ruby, les noms de classes sont eux aussi des constantes.*

Les symboles

Les symboles sont des idiomes qui référencent en mémoire, de façon unique, les chaînes de caractères, en s'associant à l'identifiant d'un objet String dans l'espace de nom du contexte d'exécution de programme en cours.

```
"mot".to_sym => :mot
```

***Note** : En Ruby, il n'y a que des références aux objets Ruby*

En Ruby tout est objet, tout appel de méthode est un message envoyé à un objet. Voir la méthode send d'un objet.

donc :

```
mon_objet.ma_methode
```

revient à envoyer le message symbolique :ma_methode à l'objet mon_objet

Les symboles sont couramment utilisés pour la construction d'accesseurs sur les attributs :

```
Class Person

  attr_accessor :nom

  def initialize(un_nom)
    @nom = un_nom
  end
end
```

Ils servent aussi dans les Hash pour servir de clé de référence :

```
config = { :size => 40, :duration => 12 }
puts config[:size] # => 40
```

Expressions

En ruby une expression correspond à tout ce que peut renvoyer un objet, soit à peu près tout :

```
42                          => 42
2 + 2                       => 4
```

Parenthésage

Le parenthésage permet de spécifier des priorités lors de l'interprétation. Comme pour une formule mathématique, les expressions entre parenthèses sont évaluées en premier :

```
3*2+4                       => 10
3*(2+4)                     => 18      # Ruby évalue d'abord l'expression entre parenthèses
3*(2*(2+4))                 => 36      # On peut incrémenter le niveau de parenthésage
```

Assignation

L'assignation d'un objet à une variable se fait avec le caractère égal (=). La variable doit être l'élément de gauche, l'objet ou l'expression doit se trouver à droite :

```
a = 42          # assigne un objet de type Fixnum et ayant pour valeur 42 à a
a = 40 + 2      # idem
```

De même grace à l'objet *Proc* il est possible d'affecter à une variable un bloc de code :

```
a = Proc.new do
    |value|
       2+value
    end
a.call(40)          #      => 42
```

Nous étudierons par la suite plus en détail l'objet *Proc*.

Assignations parallèles

Ruby permet d'assigner plusieurs variables à la fois, en séparant celles-ci par une virgule (,). Par exemple pour intervertir deux variables :

```
a = 8
b = "test"

a, b = b, a
a               #    => "test"
b               #    => 8
```

De même nous pouvons affecter les valeurs d'un tableau à plusieurs variables (nous verrons l'utilisation des tableaux plus tard).

```
a = [1, "test", 42]
a                           #  => [1, "test", 42]

a,b = [1, "test", 42]
a                           #  => 1
b                           #  => "test"

a,b,c,d = [1, "test", 42]
a                           #  => 1
b                           #  => "test"
c                           #  => 42
d                           #  => nil
```

Appels système

Les appels système peuvent se faire de différentes manières

Quotes inversées

Appel d'une commande et récupération de la sortie dans un tableau

```
system_name = `uname`     # => "Linux\n"
system_name = `uname -a`  # => "Linux machine 2.6.22.5 #2 SMP Fri Aug 25 14:31:07 CEST 2006 i686 unknown\n"
```

Pour passer des variables, il faut absolument entourer la variable de `#{}` par exemple avec :

```
arg = "-a"    #         => "-a"
```

si l'on se contente de mettre #a

```
system_name = `uname #arg`  #  => "" l'argument #a est passé et non la valeur de a, ce qui retourne l'erreur :
# Try `uname --help' for more information.
```

Par contre :

```
system_name = `uname #{arg}` => "Linux machine 2.6.22.5 #2 SMP Fri Aug 25 14:31:07 CEST 2006 i686 unknown\n"
```

IO.popen

Il est également possible d'utiliser la commande IO.popen :

```
commande = IO.popen("uname -a")
# => <IO:0x53963888>
```

On peut alors récupérer la sortie de la commande avec :

```
sortie = commande.readlines
# => "Linux machine 2.6.22.5 #2 SMP Fri Aug 25 14:31:07 CEST 2006 i686 unknown\n"
```

Il est bien sûr possible de concaténer en une seule ligne avec :

```
sortie = IO.popen("uname -a").readlines
# => "Linux machine 2.6.22.5 #2 SMP Fri Aug 25 14:31:07 CEST 2006 i686 unknown\n"
```

Et de concaténer la commande et les arguments :

```
arg = "-a"
sortie = IO.popen(["uname",arg].join(" ")).readlines
# => "Linux machine 2.6.22.5 #2 SMP Fri Aug 25 14:31:07 CEST 2006 i686 unknown\n"
```

Cela parait plus compliqué que d'utiliser les simples quotes inversées, mais a l'avantage de ne pas appeler un interpréteur de commande.

Commande system

On peut également utiliser la commande *system*. Elle retourne le code retour de la commande et affiche la sortie de la commande sur la sortie standard.

```
a = system("uname -a")   #   => true
```

et à l'écran :

```
Linux machine 2.6.22.5 #2 SMP Fri Aug 25 14:31:07 CEST 2006 i686 unknown
```

Commande exec

La commande exec, exécute la commande puis quitte définitivement ruby, donc pas de retour

```
exec("uname -a")
```

Quitte le programme en affichant la sortie standard :

```
Linux machine 2.6.22.5 #2 SMP Fri Aug 25 14:31:07 CEST 2006 i686 unknown
bash $
```

Contrôle

Méthodes logiques

Une expression booléenne est toute expression qui renvoie après évaluation vrai (`true`) ou faux (`false`). Une valeur booléenne peut être de type booléen (`true` ou `false`)

Attention:

- 0 est `true` => Objet Fixnum de valeur 0
- "" est `true` => Objet String de valeur ""
- 'False' est `true` => Objet String de valeur 'False'

Il s'agit ici d'une autre des caractéristiques des langages 'tout objet' ou même les nombres sont des objets. Tout objet est true.

Une autre caractéristique de Ruby et de tous les langages "tout objet", est que les "opérateurs", décrit ci-dessous, comme on les appelle dans les langage orientés objet ou procéduraux, sont en réalités des méthodes d'instance. En règles générale, le développeur doit implémenter lui même ses méthodes. Néanmoins certaines d'entre, dont les equivalents d'"opérateurs", sont pre-définies dans la classe `Object`, mère de toutes les autres classes, ou encore dans un module. Nous reviendrons sur ces concepts dans le chapitre traitant de la programmation objet.

Les exemples, ici, utilisent des objets "standards" pour un souci de clarté, ils peuvent être remplacés par des expressions placées entre parenthèses. Ainsi :

```
42 == 42              => true
```

équivaut à

```
(40 + 2) == (21 * 2)   => true
```

defined?

`defined?` permet de vérifier l'existence d'une variable, l'opérateur renvoie `nil` si la variable n'existe pas, sinon renvoi une description :

```
defined? 42           => "expression"
defined? toto         => nil
defined? 42.times     => "method"
defined? $_           => "global-variable"
...
```

eql?

`eql?` teste si 2 variables ou valeurs sont égales et du même type, renvoi `true` si c'est le cas, sinon `false`.

```
42.eql?(42)           => true
42.eql?(18)           => false # Valeurs différentes
42.eql?(42.0)         => false # Types différents, en effet 42 est un entier, 42.0 un flottant
```

equal?

`equal?` renvoie `true` si les deux objets comparés sont en réalité un même objet (même id), sinon renvoie `false`.

```
a = "mon objet"
b = a

a.equal?(a)           => true
a.equal?(b)           => true
```

```
c = "mon objet"
```

```
a.equal?(c)           => false
```

```
42.equal?(42)         => true
"uv".equal?("uv")     => false
```

==

== permet de tester l'égalité entre les valeurs de deux éléments :

```
42 == 42              => true
42 == 23              => false
42 == "42"            => false
```

!=

`!=` permet de tester l'inégalité entre les valeurs de deux éléments :

```
42 != 42               => false
42 != 23               => true
42 != "42"             => true
```

<=>

`<=>` sert à comparer deux variables, cette méthode n'est pas à proprement parlé un opérateur booléen. En effet celle-ci renvoie `-1`, `0` ou `1` si, le premier élément est respectivement inférieur, égal ou supérieur au second, et renvoie `nil` si les types sont différents.

```
2 <=> 2                => 0
2 <=> 3                => -1
42 <=> 18              => 1
42 <=> "42"            => nil
42 <=> 42.0            => 0 # contrairement à eql?, c'est la valeur qui est comparée
```

Pour deux valeurs numériques cet opérateur retourne donc le signe de la différence entre celles-ci.

< et <=

`<` et `<=` correspondent respectivement à "inférieur" et "inférieur ou égal à" :

```
42 < 43                => true
42 < 42                => false
42 <= 42               => true
"abcdef" < "abzd"      => true       #la comparaison entre chaînes de caractères se fait lettre à lettre
"abzd"   < "abcdef"    => false
```

> et >=

`>` et `>=` correspondent respectivement à "supérieur à" et "supérieur ou égal à" :

```
42 > 43                => false
42 > 42                => false
42 >= 42               => true
"abcdef" > "abzd"      => false       #la comparaison entre chaînes de caractères se fait lettre à lettre
"abzd"   > "abcdef"    => true
```

not et !

`not` et `!` exprime la négation, l'opérateur "inverse" l'élément placé juste après, ainsi :

```
not true               => false
(!false)                 => true
```

De même, on peut exprimer la négation sur une expression si celle-ci se trouve entre parenthèses :

```
(!(2 < 10))              => false
not (2 < 10)             => false
```

and et &&

`and` et `&&` correspondent au "et logique" :

```
true and false           => false
true and true            => true
```

Dans un souci d'optimisation, Ruby teste la première expression, et si celle-ci est fausse, il ne teste pas la seconde.

or et ||

`or` et `||` correspondent au "ou logique"

```
true or false            => true
true or true             => true
false or false           => false
```

Dans un souci d'optimisation, Ruby teste la première expression, et si celle-ci est vraie, il ne teste pas la seconde.

Les expressions conditionnelles

Une expression conditionnelle sert à réagir en fonction d'un élément donné.

Expression IF...THEN...ELSE

En algorithmique :

```
SI <EXPRESSION CONDITIONNELLE> ALORS
  EXPRESSION1
[SINON
  EXPRESSION2
]
FIN SI
```

se traduit en Ruby par :

```
if <EXPRESSION CONDITIONNELLE> [then]
  EXPRESSION1
[else
  EXPRESSION2
]
end
```

La condition peut être une expression, à condition que celle-ci soit une expression booléenne (vrai ou faux), ou une valeur numérique (0 ayant valeur de faux).

donc :

```
if (1)
```

équivaut à

```
if (true)
```

qui équivaut à

```
if (2 + 2 == 4)
```

de même :

```
if (0)
```

équivaut à

```
if (false)
```

qui équivaut à

```
if (2 + 2 == 5)
```

Une expression conditionnelle peut être l'élément ouvrant d'un bloc, ou succéder à une expression si celle-ci tient sur une ligne, dans ce denier cas on ne peut utiliser *else* :

```
if true
  puts "true"
end                         => "true"
```

équivaut à

```
puts "true" if true         => "true"
```

Le pendant de *if* est *unless* (sauf si) et suit les mêmes règles, toutefois l'instruction n'est exécuté que si la condition est fausse :

```
puts "false" unless false   => "false"
```

À noter que pour faciliter la lecture, ruby permet de faire suivre l'expression testée du mot clef *then* :

```
if (true) then
```

Expression CASE...WHEN

```
SI <EXPRESSION CONDITIONNELLE>
  VAUT EXPRESSION 1 ALORS EXPRESSION RESULTANTE 1
  VAUT EXPRESSION 2 ALORS EXPRESSION RESULTANTE 2
  ...
  VAUT EXPRESSION N ALORS EXPRESSION RESULTANTE N
SINON EXPRESSION RESULTANTE FIN SI
```

ce qui se traduit en ruby par :

```
case <EXPRESSION CONDITIONNELLE>
  when EXPRESSION 1 then EXPRESSION RESULTANTE 1
[  when EXPRESSION 2 then EXPRESSION RESULTANTE 2
  ...
```

```
  when EXPRESSION N then EXPRESSION RESULTANTE N
else EXPRESSION RESULTANTE] end
```

case va comparer l'expression le suivant avec les expressions passées après les mots-clefs *when*. Si l'une d'elle correspond, case appelera l'expression suivant le *then*. Sinon il évaluera l'expression suivant le *else* :

```
chaine = "ça va ?"

case chaine
  when "bonjour" then "bonjour"
  when "ça va ?" then "oui merci"
  else "au revoir"
end                             => "oui merci"
```

À noter que l'expression suivant le *case* est facultative :

```
age = 40

case
  when ((age > 60) and (age < 100)) then "Vous êtes agé"
  when ((age <= 60) and (age > 15)) then "Vous êtes dans la fleur de l'age"
  when age <= 15 then "Vous êtes jeune"
  else "Mathusalem"
end                             => "Vous êtes dans la fleur de l'age"
```

Si *age* avait été supérieur à 100, c'est la condition *else* qui aurait été prise en compte.

À noter que *then* peut être remplacé par le caractère deux points (:), on peut également mettre la clause *when* sur deux lignes :

```
case
  when ((age > 60) and (age < 100))
    "Vous êtes agé"
...
end
```

Les boucles

Les boucles permettent de parcourir une liste d'éléments, ou d'effectuer une action tant qu'une condition est respectée.

While/Until

```
TANT QUE <EXPRESSION CONDITIONNELLE> FAIRE
  EXPRESSION1
FIN TANT QUE
```

ce qui se traduit en ruby par

```
while <EXPRESSION CONDITIONNELLE>
  EXPRESSION1
end
```

Par exemple :

```
i = 0

while (i < 5)
   puts i
   i = i + 1
end
                                          =>  0
                                              1
                                              2
                                              3
                                              4
```

Dans cet exemple l'expression conditionnelle est évaluée avant l'évaluation du corps de la boucle, donc si i avait été égal à 6, le contenu de la boucle n'aurait jamais été évalué. Néanmoins nous pouvons utiliser une autre construction commençant par un bloc et suivit de *while*, dans ce cas là le corps de la boucle est évalué au moins une fois :

```
i = 12

begin
  puts i
  i = i + 1
end while (i < 5)
                                         =>  12
```

La négation de while est until, que l'on pourrait traduire par "jusqu'à" :

```
i = 0

until (i > 5)
   puts i
   i = i + 1
end
                                          =>  0
                                              1
                                              2
                                              3
                                              4
```

```
                                          5
```

loop

loop ressemble à la structure while, mais ne prend pas d'expression conditionnelle. En fait *loop* correspond à

```
while(true)
```

Le seul moyen de quitter la boucle est d'utiliser l'instruction break :

```
i = 0

loop do
  puts i
  i = i + 1
  break if i >= 5
end
                                       => 0
                                          1
                                          2
                                          3
                                          4
```

For in

La boucle *for* permet d'itérer à travers un ensemble :

```
for i in 5..8
   puts i
end
                                      =>  5
                                          6
                                          7
                                          8
```

Nous verrons les intervalles (de forme x..y) dans le chapitre suivant, pour l'heure il suffit de savoir qu'ils représentent tous les éléments compris entre x et y.

Ceci est une manière d'itérer, néanmoins nous verrons dans les chapitres suivant qu'en ruby il vaut mieux itérer en utilisant les méthodes adaptées. Ainsi le même exemple aurait pu s'écrire :

```
(5..8).each do |i|
  puts i
end
```

Pour les chaînes de caractères, le retour chariot servira de séparateur :

```
for ligne in "première\ndeuxième\ntroisième"
  puts ligne
end
                                      =>  première
                                          deuxième
                                          troisième
```

Itérateurs

un grand nombre d'objet en Ruby implémente des itérateurs, cad; des méthodes de parcours des éléments de l'objet lui-même.

each

La boucle each (chaque en anglais) parcours tous les éléments d'un tableau en assignant à une variable temporaire l'élément actuel.

Soit dans une classe la va

```
noms = ['toto', 'tata', 'titi']
noms.each do |nom|
  puts "Salut #{nom} !"
end
                                       => Salut toto !
                                       => Salut tata !
                                       => Salut titi !
```

each_byte et each_line

Les chaînes de caractères ont une méthode spécifique appelée each_byte (chaque caractère) qui parcours la chaîne caractère par caractère :

```
"abcdef".each_byte{ |caractere|  printf "%c\n", caractere }
                                            a
                                            b
                                            c
                                            d
                                            e
                                            f
```

Et **each_line** (chaque ligne) qui parcoure les lignes séparés par le retour chariot :

```
["première", "suite", "autre"].join("\n").each_line{|ligne| puts ligne}
```

```
                                        =>  première
                                            suite
                                            autre
```

times

Fonction typique à Ruby, la méthode d'itération des classes d'entiers nommée **times** (fois) et qui peut être utilisé avec les constantes numériques (ce langage étant pur objet) :

```
3.times{puts "texte"}
                                        =>   texte
                                             texte
                                             texte
```

ou bien :

```
3.times do
  puts "texte"
done
                                        =>   texte
                                             texte
                                             texte
```

Les mots clés de contrôle d'execution

Parmis les rares mots clés du langage, on trouve un certain nombre de controles d'execution

Break

Nous avons déjà vu *break* qui permet d'interrompre l'exécution d'une boucle :

```
i = 0

while (true)
  break if i > 3
  puts i
  i = i+1
end
                                    => 0
                                       1
                                       2
                                       3
```

Redo

redo va réévaluer le corps de la boucle, mais sans retester la condition, et sans utiliser l'élément suivant (dans un itérateur)

```
for i in 1..4
  print "#{i} "
  if i==2
    i=0
    redo
  end
  puts i
end
                                    => 1 1
                                       2 0 0
                                       3 3
                                       4 4
```

La variable i étant définie localement (dans la boucle), cela ne change pas le déroulement par rapport à la liste globale, mais la variable est bien vue comme ayant une valeur différente de 2 la seconde fois et le redo est évité. Le puts suivant le redo dans le bloc de code de la boucle n'est pas exécuté lorsque celui est exécuté.

Next

next va aller à la fin de la boucle, puis recommencer l'itération avec l'élément suivant dans le cas d'un itérateur :

```
i=0

loop do
  i += 1
  next if i < 3
  puts i
  break if i > 4
end
                                    =>  3
                                        4
                                        5
```

Retry

retry recommence l'itération à son début, dans son état premier :

```
for i in 1..5
  puts i
  retry if i == 2
 end
                                =>  1
                                    2
```

```
                                    1
                                    2
                                     ...
```

On entre ici dans une boucle infinie.

Remarques

Boucles implicites

Le parcours des éléments d'un tableau dans Ruby est implicite lorsque on l'utilise comme variable d'assignation, comme c'est le cas pour les chaînes de caractère dans la majorité des langages :

```
puts [ "élément 1", "élément 2", "élément 3" ]
                                          =>   élément 1
                                               élément 2
                                               élément 3
```

On peut de la même façon extraire une partie d'un tableau simplement sans avoir à créer de boucle qui parcoure tous les éléments.

Exemple tiré du site officiel de Ruby

```
villes  = %w[ Londres
              Oslo
              Paris
              Amsterdam
              Berlin ]
visitees = %w[Berlin Oslo]

puts "J'ai toujours besoin " +
    "de visiter les " +
    "villes suivantes :",
    villes - visitees
                                          =>   Londres
                                               Paris
                                               Amsterdam
```

Cet exemple sort les éléments du tableau qui sont dans les villes mais pas dans visitées

Types standards

Nous allons voir ici tous les types que nous pouvons considérer comme "standards", dans le sens où nous les retrouvons dans la plupart des langages, et que ceux-ci sont directement intégrés à l'interpréteur (*built-in*). Néanmoins il ne faut pas perdre de vue qu'il s'agit en réalité d'objets.

Pour rappel les méthodes dont le nom se termine par *!* sont non pures : elles modifient l'objet.

Chaîne de caractères

En ruby les chaînes de caractères sont représentées par l'objet "*String*". En réalité cet objet contient et permet la manipulation d'un nombre indéfini d'octets, typiquement des caractères, nous pourrons néanmoins y stocker tout type de données binaires ou non.

Créer une chaîne de caractères

En ruby il existe une multitude de manière de créer une chaîne de caractères, par exemple en créant une instance de l'objet *String* :

```
maChaine = String.new("Une chaîne de caractères")
```

Mais le moyen le plus courant de créer une chaîne est de la placer entre simple quote (') ou entre double quote ("). Néanmoins ces deux moyens ne sont pas équivalents : en utilisant les doubles quotes ruby évaluera les expressions contenues entre *#{* et *}*, ainsi que les caractères d'échappements.

```
i = 100

puts "la valeur de i est \n #{i}"
                                      => La valeur de i est
                                         100

puts 'la valeur de i est \n #{i}'
                                      => la valeur de i est \n #{i}
```

Dans une chaîne de caractère \ à une valeur particulière, elle permet de placer un caractères d'échappements. Le caractère \ indique à ruby que le caractère suivant possède une valeur particulière : ainsi "\n" représente une nouvelle ligne. Les caractères d'échappements ne sont pas pris en compte avec une chaîne de caractères débutant par un simple quote, néanmoins nous pouvons l'utiliser pour indiquer à Ruby de considérer le simple quote ou le backslash suivant comme faisant parti de la chaîne :

```
puts 'un simple quote : \' qui ne ferme pas la chaîne, et ici un backslash : \\'
                                     => un simple quote : ' qui ne ferme pas la chaîne, et ici un backslash : \
```

Un autre moyen de créer une chaîne est d'utiliser *%q* ou *%Q*, qui équivalent respectivement au simple et au double quote. Néanmoins l'utilisation de *%q* ou *%Q* permet de définir le caractère d'ouverture et de fermeture de la chaîne :

```
%q!j'utilise le caractère \! pour ouvrir ou fermer la chaîne, ' et " peuvent ainsi être utilisés sans \ !
%q*je peux utiliser n'importe quel caractère non alpha-numérique qui n'est pas dans la chaîne elle même*
```

Si nous avons plusieurs lignes de texte à écrire nous pouvons utiliser les caractères << suivit d'un identifiant pour créer une chaîne ayant les mêmes propriétés qu'une chaîne entre double quote, ou les caractères <<- suivis d'un identifiant entre simple quote pour créer un chaîne de caractères ayant les mêmes propriétés qu'une chaîne entre simple quote :

```
i = 100
maChaine <<DEBUT
   une chaîne de caractères
   i vaut #{i}
DEBUT
                                    => une chaîne de caractères
                                       i vaut 100

i = 100
maChaine <<-'DEBUT'
   une chaîne de caractères
   i vaut #{i}
DEBUT
                                    => une chaîne de caractères
                                       i vaut #{i}
```

Le dernier moyen d'obtenir une chaîne de caractères est d'utiliser la méthode *to_s* que la plupart des objets implémentent, ainsi :

```
42.to_s                              => "42"
```

Comparaisons de chaînes

Les chaînes de caractères peuvent être comparées entre elles comme indiqués dans le chapitre Expressions booléennes grâce aux méthodes <, <=, ==, >=, >

```
"abcdef" <=> "abcde"          =>         1
"abcdef" <=> "abcdef"         =>       0
"abcdef" <=> "abcdefg"        =>      -1
"abcdef" <=> "ABCDEF"         =>       1    # Les caractères minuscules sont supérieurs aux caractères majuscules
"abc" <=> "acc"               =>      -1

"a" < "b"                     =>       true
"a" < "a"                     =>       false
"a" > "b"                     =>       false
"a" > "a"                     =>       false
"a" <= "b"                    =>       true
"a" <= "a"                    =>       true
"a" >= "b"                    =>       false
```

```
"a" >= "a"                    =>       true

"a" == "a"                    =>       true
"a" == "b"                    =>       false

"a" != "a"                    =>       false
"a" != "b"                    =>       true
```

La méthode est équivalente à <=>, à part que la casse n'est pas comparée :

```
"abcdef".casecmp("ABCDEF")     =>       0
```

Manipulation de chaînes

Taille d'une chaîne et index des caractères

length

length permet de connaître la taille d'une chaîne de caractères :

"Bonjour le monde".length => 16

index et rindex

index renvoie l'index de la première occurrence d'une sous chaîne dans la chaîne de caractères, renvoi *nil* si aucune occurrence n'a été trouvée :

```
"hello".index('e')               =>   1
"hello".index('lo')              =>   3
"hello".index('a')               =>   nil
```

Si un entier est donné en second paramètre, celui-ci indique l'index où commencer la recherche.

```
"Bonjour le monde".index('o',7)   =>   12
```

Nous pouvons également utiliser une expression rationnelle.

rindex effectue le même travail à la différence près que la recherche se fait à partir de la fin de la chaîne de caractères :

```
"Bonjour le monde".rindex('o')     =>   12
```

Si le second paramètre est présent, la recherche commencera au caractère indiqué :

```
"Bonjour le monde".rindex('o',9)   =>   4
```

À noter que si le second paramètre est négatif, alors l'index est compté à partir de la fin de la chaîne :

```
"Bonjour le monde".rindex('o',-9)  =>   4
"Bonjour le monde".rindex('o',-2)  =>   12
```

Opérateurs sur les chaînes de caractères

*

L'opérateur de multiplication renvoie une chaîne de caractères contenant *n* fois la chaîne représentée par l'objet :

"Ho! " * 3 => Ho! Ho! Ho!

+

L'opérateur d'addition permet de concaténer deux chaînes de caractères :

```
maChaine = "le Monde"
```

```
"Bonjour " + maChaine     => Bonjour le Monde
```

[] et []=

L'opérateur *[]* permet de récupérer un ou plusieurs caractères faisant parti d'une chaîne. Si un entier est passé en paramètre, l'opérateur renverra le code ASCII du caractère correspondant (le premier caractère est indexé par la valeur 0) :

```
maChaine = "Bonjour le monde"

maChaine[3]                => 106 # Correspond au code ASCII du 4ème caractère (j)
```

Si deux entiers sont passés en paramètres, l'opérateur renverra une sous-chaîne commençant à l'index passé en premier paramètre et de longueur indiquée par le second élément :

```
maChaine[1,5]              =>  onjou
```

On peut également indiquer un intervalle :

```
maChaine[1..6]            =>  onjou
```

Il est à noter que si l'index est négatif, ruby compte à partir de la fin de la chaîne :

```
maChaine[-3,5]            => nde
```

On peut également utiliser une expression rationnelle, nous verrons cela plus en détail dans le chapitre correspondant.

L'opérateur *[]=* permet de changer le contenu d'une chaîne de caractères. L'utilisation est semblable à celle de *[]*, à la différence que plutôt que de renvoyer un ensemble de caractères, l'opérateur modifiera la chaîne :

```
maChaine[3] = "V"
puts maChaine             => BonVour le monde
```

```
maChaine[8] = "tout"
puts maChaine             => Bonjour tout le monde
```

Si deux entiers sont passés en paramètres, ruby remplacera la chaîne commençant à l'index passé en première paramètre et d'une longueur passée en second paramètre :

```
maChaine[8,8] = "Roger"
puts maChaine             => Bonjour Roger

maChaine[8..16] = "Roger"
puts maChaine             => Bonjour Roger
```

On peut directement utiliser une chaîne de caractères ou une expression rationnelle :

```
maChaine["Bonjour"] = "Hello"
puts maChaine             => Hello le monde
```

Formatage de chaîne

split

split permet de séparer les différents éléments d'une chaîne en fonction d'un ou plusieurs délimiteurs, par défaut ceux-ci sont les fins de ligne et les espaces (\s, \t, \r,\n et \r\n). La méthode renvoi un tableau contenant les différents éléments :

```
"Bonjour le monde".split   =>      ["Bonjour", "le", "monde"]
"B.o.n.j.o.u.r".split('.') =>      ["B","o","n","j","o","u","r"]
```

Si un entier est donné en paramètre, le tableau résultant n'aura comme nombre d'élément la valeur de cet entier :

```
"B.o.n.j.o.u.r".split('.',3) =>      ["B","o","n.j.o.u.r"]
```

Nous pouvons également utiliser une expression rationnelle.

chomp, chomp!, chop et chop!

chop et *chop!* permettent de supprimer le dernier caractère d'une chaîne, néanmoins si les deux dernier caractères sont "\r\n", les deux caractères sont supprimés :

```
"Bonjour\r\n".chop         =>      "Bonjour"
"Bonjour\n\r".chop         =>      "Bonjour\n"
"Bonjour\n".chop           =>      "Bonjour"
"Bonjour".chop             =>      "Bonjou"
```

chomp et *chomp!* suppriment le dernier caractère seulement si celui-ci est un caractère de fin de ligne (soit "\r", "\n" et "\r\n") :

```
"Bonjour".chomp             =>      "Bonjour"
"Bonjour\n".chomp           =>      "Bonjour"
"Bonjour \n le monde".chomp =>      "Bonjour \n le monde"
```

Si une chaîne de caractères est donnée en paramètre, celle-ci est supprimée si elle est termine la chaîne :

```
"Bonjour".chomp("jour")     =>      "Bon"
"Bonjour\r\n".chomp("jour") =>      "Bonjour\r\n"
```

Pour rappel, les méthodes finissant par "*!*" sont des méthodes non pures : elles modifient l'objet appelant.

downcase, downcase!, upcase, upcase!, swapcase, swapcase!, capitalize et capitalize!

downcase et *upcase* permettent de passer respectivement tous les caractères en majuscules ou en minuscules.

```
"BonjOUr".downcase          =>       "bonjour"
"BonjOUr".upcase            =>       "BONJOUR"
```

swapcase inverse la casse :

```
"BonjOUr".swapcase          =>      "bONJouR"
```

capitalize ne met que le premier caractère en majuscule :

```
"bonjOUr".capitalize        =>      "BonjOUr"
```

%

L'opérateur *%* permet de formater la chaîne de caractères. Le format utilisé est le même que celui de la fonction *sprintf* de ruby et de bien d'autres langages tel le *C*. Le format est décrit par le caractère *%* suivi d'un indicateur optionnel, un indicateur de taille, de précision et de type.

Les indicateurs de type :

Indicateur	Description
b	données binaire
c	caractère
d ou i	nombre entier
e	convertit un nombre entier sous sa forme exponentielle avec un chiffre avant la virgule, la précision indique le nombre de chiffres après la virgule (par défaut 6)
E	comme *e* mais utilise le caractère majuscule *E* pour indiquer l'exposant
f	nombre flottant, la précision indique le nombre de chiffres après la virgule
g	comme *e* mais converti en nombre flottant
G	comme *E* mais converti en nombre flottant
o	nombre octal
p	symbole
s	chaîne de caractères, si une précision est donnée, alors elle indique le nombre de caractères
u	nombre entier non signé (pas de signe)
x	hexadécimal en utilisant les caractères minuscules (par exemple *f*)
X	hexadécimal en utilisant les caractères majuscules (par exemple *F*)

Format :

Indicateur	Types applicable	Description
espace	bdeEfgGioxXu	Laisse un espace au début des nombres positifs
#	bdeEfgGioxXu	Format alternatif, pour les types *o*, *x*, *X* et *b* préfixe respectivement le résultat par *0*, *0x*, *0X*, et *0b*
+	bdeEfgGioxXu	Ajoute un + au début des nombres positifs
-	bdeEfgGioxXu	Ajoute un - au début des nombres négatifs
0	*tous*	Remplit le format avec des *0* plutôt que des espaces
.	*tous*	Prend l'argument suivant comme taille pour aligner à droite s'il est positif. Si l'argument est négatif, aligne à gauche

```
"%.3s" % 42424242       =>      "424"
```

Si plusieurs éléments doivent être formatés, il faut passer les paramètres dans un tableau :

```
"%d %04x" % [12.5, 42] =>       "12 002a"
```

unpack

La méthode *unpack* permet de décoder des chaînes de caractères (ou contenant des données binaires) en corrélation avec une chaîne de format.

Indicateur	Description	Type renvoyé
A	Chaîne de caractères en supprimant les caractères vides (espace, tabulation...)	String
a	Chaîne de caractères	String
B	Extrait les bits de chaque caractère (bit de poids fort en premier)	String
b	Extrait les bits de chaque caractère (bit de poids faible en premier)	String
C	Extrait un caractère comme un entier non signé	Fixnum
c	Extrait un caractère comme un entier signé	Fixnum
d	Considère *sizeof(double)* caractères comme un double	Float
E	Considère *sizeof(double)* caractères comme un double en *little-endian*	Float
e	Considère *sizeof(float)* caractères comme un flottant en *little-endian*	Float
f	Considère *sizeof(float)* caractères comme un flottant	Float
G	Considère *sizeof(double)* caractères comme un double dans l'ordre réseau	Float
g	Considère *sizeof(float)* caractères comme un flottant dans l'ordre réseau	Float
H	Extrait le code hexadécimal de chaque caractère (bit de poids fort en premier)	String
h	Extrait le code hexadécimal de chaque caractère (bit de poids faible en premier)	String
I	Considère *sizeof(int)* caractères comme un entier non signé	Integer
i	Considère *sizeof(int)* caractères comme un entier signé	Integer
L	Considère 4 caractères consécutifs comme un entier long non signé	Integer
i	Considère 4 caractères consécutifs comme un entier long signé	Integer
M	Décode les chaînes "quoted printable"	String

m	Décode les chaînes en Base64	String
N	Considère 4 caractères consécutifs comme un entier long non signé dans l'ordre réseau	Fixnum
n	Considère 2 caractères consécutifs comme un entier court non signé dans l'ordre du réseau	Fixnum
P	Considère *sizeof(char *)* comme un pointeur et renvoi la taille de la chaîne ainsi référencée	String
P	Considère *sizeof(char *)* comme un pointeur sur une chaîne terminée par le caractère *null* (*\0*)	String
S	Considère 2 caractères consécutifs comme un entier court non signé dans l'ordre natif du système	Fixnum
s	Considère 2 caractères consécutifs comme un entier court signé dans l'ordre natif du système	Fixnum
U	Extrait une chaîne encodée au format UTF8 comme des entiers non signés	Integer
u	Extrait une chaîne encodée en UU	String
V	Considère 4 caractères consécutifs comme un entier long non signé en "little endian"	Fixnum
v	Considère 2 caractères consécutifs comme un entier court non signé en "little endian"	Fixnum
X	Retourne en arrière d'un caractère	N/A
x	Avance d'un caractère	N/A
Z	Supprime les caractères *null* de fin	String
@	Se déplace du nombre donné en argument	N/A

La chaîne de format se compose d'un nombre de directives à piocher dans le tableau précédent, facultativement suivi d'un nombre indiquant le nombre de fois qu'il faut répéter cette directive, un astérisque (*) correspondant a tous les éléments restant. Les directives *s,S,i,I,l* et *L* peuvent être suivis d'un underscore (_) indiquant de choisir le format natif du système.

Exemples :

```
"abc \0\0abc \0\0".unpack('A6Z6')      => ["abc", "abc "]
"abc \0\0".unpack('a3a3')                    => ["abc", " \000\000"]
"aa".unpack('b8B8')                             => ["10000110", "01100001"]
"aaa".unpack('h2H2c')                         => ["16", "61", 97]
"\xfe\xff\xfe\xff".unpack('sS')                => [-2, 65534]
"now=20is".unpack('M*')                      => ["now is"]
"whole".unpack('xax2aX2aX1aX2a') => ["h", "e", "l", "l", "o"]
```

Itérateurs

Les itérateurs sont un mécanisme puissant de ruby, ils permettent de parcourir les éléments d'un objet. Nous verrons plus tard comment étendre facilement sa propre classe avec des itérateurs. Pour chaque type que nous allons voir, nous allons voir ses itérateurs. A l'heure actuelle considérons simplement un itérateur comme une méthode prenant un bloc pour paramètre (et d'ailleurs c'est ce que les itérateurs sont : de simples méthodes).

succ, succ! et upto

succ se contente de renvoyer l'élément succédant à la chaîne, en pratique *succ* incrémente le dernier caractère alphanumérique de la chaîne, si celui-ci a atteint sa limite, *succ* incrémentera l'avant dernier caractère et ainsi de suite:

```
"abcd".succ                    => "abce"
"THX1138".succ                 => "THX1139"
"<<koala>>".succ               => "<<koalb>>"
"1999zzz".succ                 => "2000aaa"
```

upto itère à travers les valeurs successives d'une chaîne, jusqu'à arriver à la chaîne passée en paramètre. La méthode incrémente à partir de la dernière valeur alphanumérique de la chaîne :

```
"<<aa>>".upto("<<bb>>") do
  |i| puts i
end
                               => <<aa>>
                                  <<ab>>
                                  <<ac>>
                                  <<ad>>
                                  <<ae>>
                                  <<af>>
                                  <<ag>>
                                  <<ah>>
                                  <<ai>>
                                  <<aj>>
                                  <<ak>>
                                  <<al>>
                                  <<am>>
                                  <<an>>
                                  <<ao>>
                                  <<ap>>
                                  <<aq>>
                                  <<ar>>
                                  <<as>>
                                  <<at>>
                                  <<au>>
                                  <<av>>
                                  <<aw>>
                                  <<ax>>
                                  <<ay>>
                                  <<az>>
                                  <<ba>>
                                  <<bb>>
```

each_byte

each_byte permet d'itérer à travers la chaîne de caractères, octet par octet. La valeur qui sera envoyée au bloc sera le code ASCII (donc un entier) du caractère :

```
"bonjour".each_byte {|i| print i.to_s+" "} => 98 111 110 106 111 117 114
```

each

each permet d'itérer à travers chaque ligne contenu dans une *String* :

```
"H\nA\r\nL\n".each do
   |i| print i.succ
end                                => IBM
```

Il est à noter que l'itérateur *each* est celui appelé par la construction *for..in' :*

```
for i in "H\nA\r\nL\r"
  print i.succ
end
                                   => IBM
```

Valeur numérique

En ruby les valeurs numériques sont soit flottantes soit entières et sont de taille infinie (jusqu'à la limite de mémoire du système). Plus tôt dans le livre, nous avons indiqué qu'un type entier est de type *Integer*. En fait nous avons menti. Le type Integer permet en réalité de cacher le type réel de la valeur. En effet, les valeurs inférieurs aux valeurs d'un entier sur le système (donc en général 32 ou 64 bits) sont en réalité de type *Fixnum*. Au delà, elles seront de types *Bignum*. En pratique la conversion est transparente pour le développeur :

```
i = 10
5.times do
    print i.class,"\n"
    i = i*i
end
                                                  =>   Fixnum
                                                         Fixnum
                                                         Fixnum
                                                         Fixnum
                                                         Bignum
```

Syntaxe

En ruby une valeur numérique peut s'écrire de différente forme, notamment en fonction de la base utilisée.

Classiquement, une valeur numérique peut s'écrire comme une suite de chiffres séparés éventuellement par des caractères de soulignement (_) qui seront ignorés lors de l'interprétation. Une valeur négative est simplement préfixée par le signe moins (-) :

```
42                                          => 42
4_2                                        => 42
-42                                         => -42
```

On peut également travailler dans une autre base en préfixant la valeur numérique. Ainsi en préfixant avec *0* (zéro) on indique l'utilisation d'un nombre en base octale, 0x pour un nombre en hexadécimal et 0b pour un nombre en binaire :

```
0767                                      => 503
0xaabb                                   => 43707
0b01101101                          => 109
```

Utilisation

Un objet correspondant à une valeur numérique s'instancie simplement en écrivant cette valeur :

```
4                                           => 4
4.class                                  => Fixnum
```

Certains objets, comme l'objet *String*, possèdent une méthode *to_i* qui renvoie si possible une valeur entière, et *to_f* qui renvoi si possible un flottant :

```
"  58_87".to_i                     => 5887
```

Si la chaîne contient des éléments non numériques, à l'exception du caractère de soulignement ou du signe moins, la valeur renvoyée sera celle de la première valeur numérique trouvée et précédant les autres caractères :

```
"toto".to_i                            => 0
"to87to".to_i                         => 0
"87to".to_i                            => 87
```

"Opérateurs" arithmétiques

Attention : En Ruby les opérateurs n'existent pas, ce sont des méthodes inclusent dans l'objet Ruby père Object. On parle donc ici d'"opérateur" par analogie avec les langages orientés objet et procéduraux.

il faut bien comprendre que :

```
2 + 4
```

équivaut à

```
2.+(4)
```

tel que

```
objet1 = 2
objet2 = 4
objet1.+(objet2) => 6
```

Pour simplifier la lecture, on accordera l'usage de cette abus de langage dans le reste du document pour faciliter le passage d'un autre langage vers Ruby

+

L'opérateur + permet d'additionner 2 valeurs numériques :

```
2+2                                 => 4
"3+4".to_i                        => 3
```

-

L'opérateur - permet de soustraire une valeur numérique à une autre :

```
42-10                               => 32
2-10                                  => -8
```

*

L'opérateur * permet de multiplier 2 valeurs numériques :

```
42*2                                => 84
```

/

L'opérateur / permet de diviser une valeur numérique par une autre :

```
9/3                                  => 3
```

ATTENTION: Le type renvoyé est du type des opérandes, ainsi si la division n'est pas entière, seul le dividende est renvoyé:

```
9/4                                  => 2
```

par contre, si nous utilisons au moins un flottant, le type renvoyé sera de type flottant :

```
9.0/4                            => 2.25
```

Une division par 0 lève une exception (nous verrons les exceptions plus tard, il suffit de considérer à l'heure actuelle qu'il s'agit d'une erreur) et interrompt le cours du programme si celle-ci n'est pas interceptée :

```
9/0                                  => ZeroDivisionError: divided by 0
```

**

L'opérateur ** permet d'augmenter une valeur numérique à la puissance indiquée en paramètre :

```
9**2                                => 81
9**0                                => 1
```

%

L'opérateur % (modulo) permet de connaître le reste d'une division :

```
100%30                      => 10
```

Itérateurs

La classe *Integer* propose également certains itérateurs, comme pour tout autre itérateur, ceux-ci prennent un ou plusieurs paramètres ainsi qu'un bloc de code.

succ

succ permet de récupérer la valeur numérique suivante :

```
8.succ                                          => 9
```

times

times permet de créer une boucle allant de 0 à la valeur de l'objet :

```
5.times do |val|
   print val.to_s+'..'
end
                                                => 0..1..2..3..4..
```

upto et downto

upto et *downto* permettent respectivement d'itérer à partir de la valeur de l'objet jusqu'à la valeur passée en paramètre, respectivement en incrémentant ou en décrémentant :

```
5.upto(8) do |i| print i.to_s+'..' end
                                                => 5..6..7..8..

8.downto(5) do |i| print i.to_s+'..' end
                                                => 8..7..6..5..

5.downto(8) do |i| print i.to_s+'..' end
                                                => 5
```

step

step ressemble à *upto* et *downto* à part que l'on peut préciser le pas :

```
5.step(48, 5) do |i| print i.to_s+'..' end  #ici le second paramètre correspond au pas
                                                 => 5..10..15..20..25..30..35..40..45..
```

Expression rationnelle

Les expressions rationnelles (parfois nommées à tort expressions régulières) sont un mécanisme puissant mais qui peut être complexe. Elles permettent des recherches dans une chaîne de caractères selon des critères ou un modèle de recherche précis. Ceci permet la sélection d'une sous chaîne de caractères, ou la manipulation des chaînes ainsi trouvées (typiquement une substitution).

Ruby propose une classe pour l'utilisation des expressions rationnelles, les développeurs Perl seront heureux car son utilisation en Ruby est quasi-identique (voir Catégorie:Expressions rationnelles).

Types conteneurs

Classes

En programmation objet ou orienté objet, une classe est un modèle d'objet constitué d'attributs (ses variables et constantes), de méthodes (ses fonctions), ainsi que des contraintes d'accès. Les méthodes et attributs peuvent être plus ou moins accessibles aux autres classes ou au programme principal.

En Ruby, par défaut, les attributs ne sont pas accessibles.

Déclaration d'une classe

Une classe est déclarée à l'aide du mot-clé `class` et se termine par le mot-clé `end` Les définition de classe étant des constantes, leurs noms doivent commencer par une majuscule en Ruby. Une classe a au minimum une méthode dite constructeur qui sert à initialiser les objets créés par l'opérateur `new`. En Ruby le constructeur est nommé `initialize`. Les méthodes sont définies à l'aide du mot-clé `def` et se terminent par le mot-clé `end`

```
class Point
  def initialize( x = 0, y = 0 )
    @x = x
    @y = y
  end
  def coords()
    puts "x : #{@x}"
    puts "y : #{@y}"
  end
end
```

En Ruby, les variables commençant par `@` sont des attributs d'objet, les autres sont des variables locales. @x est donc différent de x et @y est différent de y.

Dans cette méthode `initialize`, on a donné des valeurs par défaut aux paramètres (x=0, y=0), cela permet de créer des instances de cette classe sans préciser la valeur des arguments.

Instanciation

Une instance de la classe (un objet) est créé par la méthode `new` de la classe :

```
p = Point.new
```

Ici les valeurs par défaut (0,0) seront utilisées

```
p.coords
#  => x : 0
#     y : 0
```

On peut passer des valeurs à la méthode d'initialisation entre parenthèses :

```
p2 = Point.new(2,5)
p2.coords
#   => x : 2
#      y : 5
```

On peut obtenir le type de la classe grâce à la méthode `class` :

```
p.class
#   => Point
```

Méthodes

En programmation objet, les **méthodes** sont les fonctions appartenant à une classe. En réalité, elles sont des messages envoyé à un objet.

En Ruby, elles sont définies de la façon suivante :

```
def <nom de la méthode> (*arguments,&bloc)
  <bloc de code de la méthode>
end
```

Les arguments

Les arguments de la méthode forment une liste de références d'objets.

Ils peuvent être fournis de façon unique ou en liste via :

```
def mamethode(param1, *reste)
  "mon param1 : #{arg1}" << "et le #{reste.join(', ')}" unless reste.empty?
end
mamethode("toto")   »   "mon param1 : toto"
mamethode("toto", "titi")   »   "mon param1 : toto et titi"
mamethode "toto", "titi", "tutu"    »   "mon param1 : toto et titi, tutu"
```

Les blocs liés

Une méthode Ruby peut recevoir en paramètre la référence à un bloc de code tel que :

```
def donnezmoiducode(param)
  if block_given?
    yield(param)
  else
    param
  end
end
donnezmoiducode("j'ai faim !")  »   "j'ai faim !"
donnezmoiducode("J'ai faim !") { |s| s = "miam !"') }  # => "miam !"
```

Si en dernier paramètre de méthode, en prototype, on trouve une déclaration explicite avec une esperluette, alors le bloc est converti en objet Proc et cet objet entre dans la liste de paramètres de la méthode :

```
class BoiteACalculs
  def initialize(nom, &bloc)
    @nom, @bloc = nom, bloc
  end
  def application(valeur)
    "#{valeur} #{@nom}  = #{ @bloc.call(valeur) }"
  end
end
calcul = BoiteACalculs.new("Fois deux ") { |val| val * 2 }
calcul. application(10)  #=>    "10 Fois deux = 20"
calcul. application(20) #=> "20 Fois deux = 40"
```

le constructeur initialize

La méthode `initialize`, par exemple, est la méthode appelée lorsqu'une instance est créée via la méthode `new` :

```
class Point
  def initialize ( x, y )
    @x = x
    @y = y
  end
  def coords
    puts "x : #{@x}"
    puts "y : #{@y}"
  end
end
```

Ici la méthode `initialize` définie les variable d'instances @x et @y en fonction des paramètres passés à la fonction `new`.

Si par exemple on initialise un objet comme suit :

```
p = Point.new(2,3)
# -> @x = 2 et @y = 3
```

Valeurs par défaut

On peut en Ruby donner des valeurs par défaut aux arguments des méthodes.

```
class Parle
  def initialize ( nom = "vous" )
    @qui = nom
  end
  def bonjour
    puts "Bonjour #{@qui}"
  end
end
```

Si l'on appelle la méthode sans argument, la valeur par défaut est assigné à la variable d'instance `@qui`

```
dit = Parle.new
dit.bonjour
#                   => Bonjour vous
```

Si l'on appelle la méthode avec un argument, celui-ci remplacera la valeur par défaut :

```
dit2 = Parle.new("Roger")
dit2.bonjour
#                   => Bonjour Roger
```

Accesseurs

Les accesseurs

Un accesseur est une méthode d'accès à un attribut.

Il existe deux type d'accesseurs :

- accesseurs en lecture (`Get` dans certains langages)
- accesseurs en écriture (`Set` dans certains langages)

En Ruby c'est une méthode qui porte le nom de l'attribut en lecture et le nom de l'attribut suivit de '=' en écriture tel que :

```
Class TestClass(val)
  def initialize (val = "test")
    @val= val
  end

  #Ecriture
  def val=(val)
    @val = val
  end

  #Lecture
  def val
    @val
  end

end

titi = TestClass::new
p titi.val # => "test"
titi.val = "test2"
p titi.val # => "test2"
```

Ruby propose des macro-constructions d'accesseurs pour simplifier l'écriture des classes :

```
attr_reader :v            =>  def v; @v; end
attr_writer :v              =>  def v=(value); @v=value; end
attr_accessor :v        =>  attr_reader :v; attr_writer :v
attr_accessor :v, :w  =>  attr_accessor :v; attr_accessor :w
```

Héritage

L'héritage permet de créer une classe à partir d'une autre pour l'enrichir, ou la spécialiser. Une classe (classe dérivée, ou classe fille) hérite d'une autre classe (classe de base, ou classe mère).

On peut toujours utiliser les méthodes de la (des) classe(s) mère(s) dans une classe dérivée.

L'opérateur < précède le nom de la classe de base.

Exemple

La classe `Point` vue précédemment :

```
class Point
  def initialize( x, y )
    @x = x
    @y = y
  end
  def coords
    puts "x : #{@x}"
    puts "y : #{@y}"
  end
end
```

peut être dérivée en une classe `PointCouleur` représentant un point coloré. Cette classe possède donc un attribut supplémentaire pour la couleur :

```
class PointCouleur < Point
  def initialize( x, y, couleur )
    super(x,y)        # appel au constructeur de la classe Point
    @couleur = couleur
  end
  def couleur
    puts "couleur : #{@couleur}"
  end
end
```

Lorsqu'on crée une instance de la classe dérivée, on bénéficie immédiatement des méthodes de celle-ci :

```
pc = PointCouleur.new(1,2,"rouge")
pc.couleur
#    => couleur : rouge
```

Et les méthodes de la classe de base étant héritées, on pourra la réutiliser dans la classe dérivée :

```
pc.coords
#    => x : 1
#       y : 2
```

Par contre, la classe de base ne possède pas les méthodes définies dans les classes dérivées.

```
  p1 = Point.new
  p1.couleur
     =>  NoMethodError: undefined method `couleur' for #<Point:0xb7c059b0 @x=0, @y=0>
```

exception

mixins

modules

stdlib

Ruby est livré avec une bibliothèque standard. Elle propose plusieurs classes dont, notamment :

- time
- erb
- uri

Ces classes sont utilisées à d'autres points du livre. Pour les autres, vous pouvez vous référer à la documentation de la bibliothèque standard (http://ruby-doc.org/stdlib/).

Ruby et le texte non structuré

Analyser du texte

L'implémentation des réseaux baysiens en Ruby est bn4f (Bayesian Networks for Ruby) (http://bn4r.rubyforge.org/)

On peut pratiquer une analyse sémantique latente grâce à classifier (http://classifier.rubyforge.org/) qui fournit aussi des réseaux bayésiens.

Indexer du texte

Lucene est porté en Ruby sous le nom de Ferret (http://ferret.davebalmain.com/).

Programmation concurrente

Ruby intègre nativement la notion de processus et de processus léger.

Utiliser des processus légers

Les processus légers sont représentés par la classe Thread. À la construction du thread, on passe un bloc qui sera exécuté.

```
# Un entier, un thread qui l'incrémente
# et un thread qui le décrémente
i = 0

Thread.new {
  while true
    i = i + 1
    puts i
  end
}

Thread.new {
  while true
    i = i - 1
    puts i
  end
}
```

Synchroniser des données

Dans l'exemple ci-dessus, la variable i est partagée. Cela peut poser un problème de synchronisation. Pour le résoudre, on utilise un sémaphore. Pour qu'un objet soit synchronisé, il faut lui ajouter un attribut Mutex (http://ruby-doc.org/core/classes/Mutex.html).

Arrêter un thread

Pour arrêter un thread proprement, on affecte son attribut @continue à faux (par exemple dans une méthode #stop).

On peut arrêter un thread de façon brutale avec un appel à Thread#terminate.

Programmation distribuée

Ruby intègre nativement le module DRb (http://ruby-doc.org/stdlib/libdoc/drb/rdoc/classes/DRb.html) qui contient Rinda (http://ruby-doc.org/stdlib/libdoc/rinda/rdoc/index.html).

Persistance des données

Mémoriser des données dans la RAM

Elles seront ainsi disponibles entre deux exécutions d'un programme. Madeleine (http://madeleine.rubyforge.org/) peut donc mémoriser des objets dès lors qu'ils sont *sérialisables* avec Marshal.

Gem Ruby
madeleine

YAML

YAML (YAML Ain't Markup Language) permet de sérialiser des objets et de collection d'objets. Tout comme XML, c'est un format lisible par l'humain. On peut ainsi modifier directement un fichier yaml via un éditeur de texte.

YAML est souvent préféré à XML dans la communauté Ruby qui juge ce dernier trop verbeux.

Ruby permet de manipuler des le format YAML via yaml4r (http://yaml4r.sourceforge.net/doc/).

Manipuler du XML

Cette section est vide, pas assez détaillée ou incomplète.

Bases de données

Web

Ruby permet d'interagir avec le Web en tant que client mais aussi en tant que serveur.

Créer des applications Web


Cette section est vide, pas assez détaillée ou incomplète.

Ruby propose, entre autre :

- RoR :le célèbre framework MVC Ruby on Rails, pour des besoins plutôt applicatifs.
  - RoR s'appuie sur ActiveRecord pour la partie Modèle, les suites de composants Action.* et Active.* pour les controlleurs et les vues.
- Merb : un autre framework MVC voulu plus souple et modulaire, pour une approche plus adaptable.
- Sinatra : un framework minimaliste, léger et terriblement efficace, pour des besoins limités et spécifiques.

Web scraping en Ruby

Il est possible de faire du Web-scraping en Ruby via la bibliothèque net/http. Cela est toutefois rudimentaire et on préfèrera utiliser un framework tel scRUBYt! (http://scrubyt.org/).

Manipuler des fichiers

Services

IHM

Réflexion

Méthodes de l'instance

La méthode générique, présente dans toutes les classes, `instance_methods` renvoie la liste des méthodes existantes.

```
Parle.instance_methods
# => ["bonjour", "method", "instance_variables", "__id__", "to_s", "send", "object_id", "dup", "private_methods",
# "=~", "is_a?", "class", "tainted?", "singleton_methods", "eql?", "untaint", "instance_of?", "id",
# "instance_variable_get", "inspect", "instance_eval", "extend", "nil?", "__send__", "frozen?", "taint",
# "instance_variable_defined?", "public_methods", "hash", "to_a", "clone", "protected_methods", "respond_to?",
# "display", "freeze", "kind_of?", "==", "instance_variable_set", "type", "===", "equal?", "methods"]
```

Cette liste contient l'ensemble des méthodes existant par défaut, plus celle qui ont été créées.

On peut obtenir une liste restreinte aux fonctions que l'on a crée en remplaçant l'argument `true` (vrai, valeur par défaut) par `false` (faux) :

```
Parle.instance_methods(false)
# => ["bonjour"]
```

Au contraire, on peut obtenir les méthodes qui ne sont pas que des méthodes d'instances :

```
Parle.methods
# => ["inspect", "private_class_method", "const_missing", "clone", "method", "public_methods",
# "public_instance_methods", "instance_variable_defined?", "method_defined?", "superclass", "equal?", "freeze",
# "included_modules", "const_get", "methods", "respond_to?", "module_eval", "class_variables", "dup",
# "protected_instance_methods", "instance_variables", "public_method_defined?", "__id__", "eql?", "object_id",
# "const_set", "id", "singleton_methods", "send", "class_eval", "taint", "frozen?", "instance_variable_get",
# "include?", "private_instance_methods", "__send__", "instance_of?", "private_method_defined?", "to_a", "name",
# "autoload", "type", "new", "<", "protected_methods", "instance_eval", "<=>", "display", "==", ">", "===",
# "instance_method", "instance_variable_set", "kind_of?", "extend", "protected_method_defined?", "const_defined?",
# ">=", "ancestors", "to_s", "<=", "public_class_method", "allocate", "hash", "class", "instance_methods",
# "tainted?", "=~", "private_methods", "class_variable_defined?", "nil?", "untaint", "constants", "is_a?",
# "autoload?"]
```

Classe de l'instance

La méthode `type` permet de connaître le type (donc la classe) utilisé par l'instance :

```
dit = Parle.new
dit.type
#      => Parle
```

Existence d'une méthode

On peut déterminer si une méthode existe ou pas à l'aide des méthodes par défaut :

```
Parle.method_defined?("bonjour")
# => true
Parle.method_defined?("aurevoir")
# => false
```

On peut également déterminer si elle est publique (accessible en dehors de la classe) :

```
Parle.public_method_defined?("bonjour")
# => true
```

Manipuler des contextes

Cette section est vide, pas assez détaillée ou incomplète.

L'objet Binding (http://ruby-doc.org/core/classes/Binding.html) que l'on peut obtenir grâce à Kernel#binding (http://ruby-doc.org/core/classes/Kernel.html#M005946).

Pour les programmeurs Java

Cette page s'adresse aux développeurs qui sont habitués à Java, elle liste les subtilités et donne des conseils pour trouver rapidement ces marques.

Subtilité dans la visibilité

Les notions de *private*, *protected* et *public* différent en Java et en Ruby.

En Java, deux objets peuvent faire appels aux méthodes privées de l'un et de l'autre tant que ces deux objets sont les instances de la même classe. Ce n'est pas la cas en Ruby où une méthode déclarée *private* ne peut être appelée sur une instance que par l'instance elle-même.

En revanche, le comportant décrit précédemment en Java peut être obtenu en Ruby avec *protected*. Autrement dit, un objet peut appeler les méthodes protégées d'un autre objets seulement si ces deux objets sont instances de la même classe.

Remarquons qu'en Ruby, contrairement à Java, il n'est pas possible de rendre une méthode *invisible* pour ses sous-classes (ce que permet *protected* en Java). La notion de visibilité *public* est similaire dans les deux langages.

Divers

- Le mot clé *synchronised* revient, en Ruby, à créer un instance de Mutex (http://ruby-doc.org/core/classes/Mutex.html) dans l'objet synchronisé.
- Prevayler en Java se nomme Madeleine

Pour les programmeurs Python

Cette page s'adresse aux développeurs qui sont habitués à Python, elle liste les subtilités et donne des conseils pour trouver rapidement ces marques.

- La bibliothèque Reverend[1] de Python est portée sous Ruby sous le nom de « Bishop » (http://bishop.rubyforge.org/) (*Archive (https://web.archive.org/web/*/http://bishop.rubyforge.org/)* • *Wikiwix (https://archive.wikiwix.com/cache/?url=http://bishop.rubyforge.org/)* • *Que faire ?*), 2014-07-24

Références

1. https://pypi.python.org/pypi/Reverend

Pour les programmeurs PHP

IRB

IRB (Interactive Ruby Shell) est un interpréteur Ruby en ligne de commande.

Installer IRB

Cette section est vide, pas assez détaillée ou incomplète.

Sous linux, utilisez votre gestionnaire de paquet.

Paquet logiciel

Ubuntu (? (http://doc.ubuntu-fr

Configurer IRB

À chaque lancement de irb, le fichier *~/.irbrc* est interprété. Voici un exemple de fichier qui vous rendra l'utilisation d'IRB plus agréable :

```
# Activation de l'auto-complétion
require 'irb/completion'

# Support de l'Unicode
$KCODE='u'
require 'jcode'
```

Debuggueur

Le mode DEBUG

Une première façon simple de faire est d'utiliser le mode DEBUG natif à Ruby :

- Dans votre programme, placez vos instructions (vos traces) dans un bloc conditionnel avec un test sur la constante globale $DEBUG
- Pour lancer le script en mode debug vous pouvez
  - utiliser `ruby -d mon_script.rb`
  - lancer irb avec `irb -d`
  - dans irb, entrer la commande $DEBUG=true

Utiliser le débogueur

Le débogueur est très simple à utiliser :

Gem Ruby
ruby-breackpoint

```ruby
require 'breakpoint'

# du code...

breakpoint

# du code...
```

lancez votre script avec `ruby`. Dès que l'interpréteur rencontrera l'instruction breakpoint, il vous donnera la main via une session irb. Vous pourrez alors directement via l'interpréteur afficher le contenu des variables, faire des tests, injecter des données de test, utiliser la réflexion pour vérifier qu'un objet vérifie bien une propriété attendue, etc. Quittez la session irb et l'interpréteur reprendra là où il s'est arrêté.

Breakpoints intelligents

Plutôt que de faire appel à `breakpoint` dans votre code, utilisez assert. Passez-lui un bloc de code à évaluer et assert ne vous donnera la main que si le résultat est faux.

Ainsi, assert ne vous propose une session irb que s'il semble qu'il y ait besoin d'inspecter l'état du programme.

Corriger un code extérieur

Vous découvrirez parfois des problèmes dans du code dont vous dépendez (bibliothèques, gems diverses). Il peut s'agir de bogues ou de problèmes de sécurité ou de performances. Quoiqu'il en soit, ce problème vous ennui et vous devez le corriger.

Une première possibilité serait d'obtenir le code-source du code erroné, de le corriger et de construire l'application avec cette version. Cela pose évidemment des problèmes des mises à jour (il faudra fusionner vos corrections avec les modifications apportées par les nouvelles versions de la bibliothèque).

Ruby permet de résoudre ce problème simplement. Il vous suffit de redéfinir le code problématique. Supposons une classe problématique :

```ruby
class Bibliotheque
  def une_methode()
    # du code erroné
  end
end
```

Il suffit de rouvrir la classe et de redéfinir la méthode. Au préalable, on crée un alias vers l'ancienne version pour des raisons de compatibilité.

```ruby
class Bibliotheque

  alias :une_methode_BUG :une_methode

  def une_methode()
    # du code corrigé
  end
end
```

Cette méthode très efficace permet de contourner temporairement un problème. Bien sûr, proposez, dans la mesure du possible, votre correction aux auteurs de la bibliothèque concernée.

Profileur

Le profiling permet de déterminer quelles sont les parties de l'application qui prennent le plus de temps à s'exécuter.

Pour obtenir un profil de votre programme, ajoutez simplement

```
require 'profile'
```

et lancez le script avec `ruby`.

Tests unitaires

La bibliothèque standard intègre Test::Unit (http://ruby-doc.org/stdlib/libdoc/test/unit/rdoc/classes/Test/Unit.html) qui permet de réaliser des tests unitaires.

Rdoc

La plate-forme Ruby propose l'outil Rdoc (http://rdoc.sourceforge.net/).

Rubygems

rake

Étendre Ruby

Créer un wrapper Ruby pour une bibliothèque C MRI

Faire appel à des bibliothèques Java depuis JRuby

Cela est rendu possible par l'implémentation JRuby de Ruby.

Implémentations

Mots Réservés

```
=begin   break    elsif    module   retry    unless
=end     case     end      next     return   until
BEGIN    class    ensure   nil      self     when
END      def      false    not      super    while
alias    defined? for      or       then     yield
and      do       if       redo     true
begin    else     in       rescue   undef
```

variables globales prédéfinies

Variables globales

Nom	Type	Description
__FILE__	*String*	Représente le fichier courant
__LINE__	*Fixnum*	Représente la ligne courante
$!	*String*	*Dernier message d'erreur*
$'	*Object*	*description*
$"	*Object*	*description*
$$	*Object*	*Retourne le numéro de process du programme*
$&	*Object*	*description*
$*	*Object*	*Liste des arguments passés au script*
$+	*Object*	*description*
$,	*Object*	*description*
$-0	*Object*	*Nom du script en cours d'exécution*
$-F	*Object*	*description*
$-I	*Object*	*description*
$-K	*Object*	*description*
$-a	*Object*	*description*
$-d	*Object*	*description*
$-i	*Object*	*description*
$-l	*Object*	*description*
$-p	*Object*	*description*
$-v	*Object*	*description*
$-w	*Object*	*description*
$.	*Object*	*Numéro de la dernière ligne lue par l'interpréteur*
$/	*Object*	*description*
$:	*Array*	Alias de $LOAD_PATH
$;	*Object*	*description*
$<	*Object*	*description*
$=	*Object*	*description*
$>	*IO*	Alias de STDOUT / $stdout
$?	*Object*	*Valeur de la sortie du dernier sous-processus exécuté*
$@	*String*	*Contexte de la dernière erreur*
$\	*Object*	*description*
$_	*Object*	*Dernière chaîne de caractère donnée par **gets***
$`	*Object*	*description*
$~	*MatchData*	Dernière Regexp évaluée
$1 à $9	*Object*	*description*
$0	*String*	Chemin du script en cours d'exécution
$configure_args	*Object*	*description*
$DEBUG	*Object*	*description*
$deferr	*Object*	*description*
$defout	*Object*	*description*
$expect_verbose	*Object*	*description*
$F	*Object*	*description*
$FILENAME	*Object*	*description*
$KCODE	*Object*	*description*
$LOAD_PATH	*Array*	Emplacements de recherche des libs pour require
$SAFE	*Fixnum*	Niveau de sécurité
$stderr	*IO*	Représente la sortie d'erreur standard
$stdin	*IO*	Représente l'entrée standard
$stdout	*IO*	Représente la sortie standard
$VERBOSE	*Object*	*description*

Constantes globales

Nom	Type	Description
DATA	*File*	Données placées en fin de script avec le mot clé __END__
FALSE	*FalseClass*	faux
NIL	*NilClass*	correspond à null (rien)
RUBY_PLATFORM	*String*	Indique la plateforme courante
RUBY_RELEASE_DATE	*String*	Date de la distribution de la version courante
RUBY_VERSION	*String*	Le numéro de version
STDERR	*IO*	La sortie d'erreur standard
STDIN	*IO*	L'entrée standard
STDOUT	*IO*	La sortie standard
SCRIPT_LINES__	*Object*	*description*
TOPLEVEL_BINDING	*Object*	*description*
TRUE	*TrueClass*	vrai

Caractères d'échappements

Le tableau suivant liste des caractères d'échappement utilisables dans les chaînes de caractères.

Nom	Description
\a	Sonnerie
\b	Efface le caractère précédent
\e	Efface le caractère suivant
\n	Nouvelle ligne
\r	Retour en début de ligne
\s	Espace
\t	Tabulation horizontale
\v	Tabulation verticale
nnn	Caractère dont le code ASCII est donné en octal *nnn*
\x*nn*	Caractère dont le code ASCII est donné en hexadécimal *nn*
\C-*x*	Control-*x*
\M-*x*	Meta-*x*
\M-\C-*x*	Meta-control-*x*

Webographie

- Le site officiel du langage Ruby :
  http://www.ruby-lang.org
- Le site de la documentation Ruby :
  http://www.ruby-doc.org
- Le site regroupant la plupart des modules Ruby (équivalent du cpan de perl) :
  http://raa.ruby-lang.org/
- Le site regroupant la plupart des projets écrits en Ruby :
  http://rubyforge.org/
- Demandes de changement à Ruby :
  http://rcrchive.net/
- News, discussions... autour de Ruby :
  http://www.rubygarden.org/
- Rubygems : le gestionnaire de package pour ruby :
  http://rubyforge.org/projects/rubygems/
- Pleac : Exemples de code en Ruby :
  http://pleac.sourceforge.net/pleac_ruby/index.html
- Apprendre à programmer : un manuel d'apprentissage de la programmation basé sur Ruby :
  http://www.ruby-doc.org/docs/ApprendreProgrammer/Apprendre_%E0_Programmer.pdf
- Tutoriel complet Ruby on Rails :
  http://www.meshplex.org/wiki/Ruby/Ruby_on_Rails_programming_tutorials