Ruby

Un article de Mangue.org, l'encyclopéde libre.

Ruby est un langage de script orienté objet inventé au Japon par Yukihiro Matsumoto (matz). Dans ce langage tout est objet et sa syntaxe se montre particulièrement agréable à utiliser. De plus Ruby est multiplatformes. Ruby a également l'avantage d'être fourni avec des outils très appréciés des développeurs comme le debogueur, le profiler et le tracer. Mais aussi la bibliothèque standard de Ruby est largement frouni: threads, bignum, matrices, réseaux, cgi, xml ...Ruby dispose également de beaucoup d'autres bibliothèques toutes aussi utiles. En effet Qt, Gtk et FOX sont disponibles pour créer des interfaces graphiques, mais aussi des parsers xslt et un mod_ruby afin développer des pages web dynamiques.

Sommaire

Un premier programme

Notre premier programme sera le calcul d'une factorielle. Par convention les fichiers Ruby portent l'extension rb. Pour éxécuter un programme en Ruby il suffit de taper la commande:

```
ruby nom_du_fichier.rb
```

Sous windows l'installeur de Ruby fait en sorte qu'il suffit de cliquer sur un fichier ruby afin de l'éxécuter.

Où trouver Ruby?

Si vous n'avez pas Ruby vous pouvez récupérer les sources ou les binaires à cette adresse http://www.ruby-lang.org. Ruby est également inclut dans des distributions linux.

```
# Une fonction factorielle
def factorielle( n )
  if n == 0
    return 1
  else
    return n * factorielle n - 1
```

```
  end
end

n = 1234
print "La factorielle de " + n.to_s + " est " + factorielle( n ).to_s + "\n"
```

Vous ne l'avez sûrement pas remarqué (quelle élégance de la part de Ruby ;-)) mais Ruby fait une conversion implicite du type entier à Bignum et de Bignum à entier lorsque c'est nécessaire. Ainsi le travail du programmeur est grandement facilité et l'utilisation de la mémoire est optimisée.

Comme on peut le constater les opérateurs pour effectuer des opérations arithmétiques sont identiques à ceux du C. On peut déjà faire quelques remarques à propos de la syntaxe de Ruby. En effet les parenthèses pour les appels de fonctions sont facultatives et les points virgules à la fin d'une ligne pour marquer la fin d'une instruction le sont également. Par contre ils sont obligatoires lorsque l'on veut écrire plusieurs instructions sur une seule ligne.

Les variables

En Ruby il existe cinq catégories de variables: variables locales, variables globales, variables d'insances, variables de classes et les constantes (bien que ce soit un contre sens d'appeller variable une constante).

Les variables locales débutent par un caractère minuscule ou de soulignement (_). Evidemment elles sont accessibles que dans la portée où elles ont été déclarées. Les variables globales ont comme premier caractère le sigle dollar ($). Les variables d'instances sont propres à chaque objet et commencent par le caractère arobas (@). Les variables de classes débutent par un double arobas (@@). En Ruby les constantes ne sont pas tout à fait des constantes. En fait lorsqu'on essaye de leur affecter une nouvelle valeur après leur initialisation un message d'avertissement apparaît et elle est quand même modifiée. Cette sorte de variable protégée débute par une majuscule comme premier caractère. Toutes ces variables (excepté les variables de classes) prennent la valeur nil par défaut.

```
variable_locale
$variable_globale
@variable_d_instance
@@variable_de_classe
Constante
```

En plus des traditionelles variables Ruby dispose également de pseudos variables. Il n'est pas possible d'affecter une valeur à des pseudos variables. La pseudo variable self représente l'objet courant et est ainsi équivalent au this du C++. Les variables true, false et nil sont respectivement vrai, faux et indéfini. Cette valeur nil est équivalente au NULL et a le même effet que le false dans les conditions. Les deux autres pseudos variables **__FILE__** et **__LINE__** représentent respectivement le nom et la ligne du fichier courant.

Les opérateurs

Voici la liste d'ordre de priorité des opérateurs en Ruby

```
::
[]
**
```

```
+ (unaire) - (unaire) ! ~
* / %
+ -
<< >>
&
| ^
> >= < <=
<=> == === != =~!~
&&
||
.. ...
? :
= += -= *= /= %= **= <<= >>= &= |= ^= &&= ||=
not
and or
```

En Ruby les opérateurs sont en réalité des appels de méthodes. En effet, l'écriture de a + b est en réalité interprété comme a.+(b). Ceci permet de surcharger très facilement les opérateurs avec une syntaxe bien plus simple qu'en C++. Par contre certains ne sont pas des appels de fonctions et donc ne peuvent pas être surchargés.

```
...
!
not
&&
and
||
or
::
!=
= += -= *= /= %= **= <<= >>= &= |= ^= &&= ||=
?
```

Il existe un dernier opérateur qui permet d'obtenir des informations à propos d'une expression, il s'agit de defined? . S'il le peut, defined? renvoie une chaîne de caractères pour décrire l'expression. Si il ne peut pas c'est nil qui est renvoyé.

```
defined? var# nil
var = 1
defined? var# "local-variable"
defined? $stdout# "global-variable"
defined? print# "method"
```

Les mots clefs

Les mots réservés en Ruby ne peuvent être utilisés ni par des variables et ni des constantes. Cependant il peuvent être utilisés pour les noms de méthode à condition qu'un récepteur soit précisé. Voici la liste des mots réservés en Ruby:

```
BEGIN
do
next
then
```

```
END
else
nil
TRUE
alias
elseif
not
undef
and
end
or
unless
begin
ensure
redo
until
break
FALSE
rescue
when
case
for
retry
while
class
if
return
yield
def
in
self
__FILE__
defined?
module
super
__LINE__
```

Les structures de contrôles

Ruby dispose des structures de contrôles classiques aux autres langages mais sans pour autant reprendre la syntaxe du C. En effet les parenthèses d'une condition sont facultatives et le blocs sont délimités par end. Le code suivant utilise les structures de contrôles: if, while, unless, until, case et for. Comme le montre le code source on peut utiliser les mots clefs and et or à l'intérieur des conditions. Les opérateurs de comparaisons dont identiques à ceux du C.

```
#! /usr/bin/ruby

str = "Ruby"
var = 5
i = 0
lettre = "b";
tableau = [ 6, 7, 9, 2, 4, 9, 1 ]
j = 0

# La condition if
if var == 5 and str == "Ruby"
  print str + " powered !" + "\n"
else
  print "Python is good too" + "\n"
end

# La boucle while
while i < 10
  if ( i+= 1 ) == 3
    next# Revient juste avant la condition
  end
  print "i = " + i.to_s + "\n"
end

# L'instruction unless equivalent au if inversé
unless i > 100 then
  print "i < 100 \n"
else
  print "i > 100 \n"
end

# La boucle unless equivalent au while inversé
until i > 20
  print "i = " + i.to_s + "\n"
  i += 1
  break if i == 17
end

# Le case equivalent au switch
case lettre
  when "a":
    print "a \n"
  when "b":
    print "b \n"
  when "c":
    print "c \n"
  when "d":
   print "d \n"
  else
    print "other \n"
end

loop = 5

# Un autre style de boucle
0.upto( loop ) { |j| puts j }

# La valeur de loop n'est pas modifiée
loop.downto( 0 ) do |j|
  puts loop
end

# Le for pour la manipulation des tableaux
for t in tableau
```

```
  print "tableau[ " + j.to_s + " ] = " + t.to_s + "\n"
  j += 1
end
```

On peut remarquer qu'il est possible d'effectuer des boucles sans utiliser de structure de contrôle. En effet les méthodes upto et downto de classe Integer répètent un bloc de code jusqu'à ce que la valeur passée en argument soit atteinte. Ce bloc de code doit être délimité par des accolades ({...}). Ces deux méthodes ont l'avantage d'éviter certaines boucles infinies.

```
10.upto( 1 ) { |i| puts i }# Affichera seulement 10
```

La boucle for en Ruby est semblable à celle du foreach en php. Elle permet d'obtenir chaque élément d'un tableau sans se soucier de sa longueur et d'incrémenter un compteur. La variable tableau indique sur quel tableau la boucle va travailler et element représente l'élément du tableau actuellement sélectionné. Cette dernière structure de contrôle nous amène à la manipulation des tableaux en Ruby.

Les tableaux

Les tableaux permettent de stocker des objets de n'importe quel type et sont représentés par la classe Array. Cette classe fournie environ 60 méthodes pour faciliter leur manipulation. Les opérateurs ont été surchargés afin de travailler plus facilement avec les tableaux.

La documentation de Ruby

La liste complète de toutes les méthodes de la classe Array est consultable avec la documentation de référence de Ruby à cette adresse http://www.ruby-doc.org/

```
tableau = [1, 2, 3, "a", "b", "c"]
tableau_imbrique = [ 1, 2, 3, [ "a", "b", "c" ] ]

print "Tableau après initialisation:\n"
print tableau.join( " " ) + "\n"

print "\nPremière occurence de l'élément 2:\n"
print "tableau[ " + tableau.index( 2 ).to_s + " ] = 2\n"

print "\nOpérations pop et push sur le tableau:\n"
tableau.pop
tableau.pop
tableau.pop

tableau.push( 2 )
tableau.push( 9 )
tableau.push( 7 )
tableau.push( 5 )
tableau.push( 6 )
tableau.push( 4 )
tableau.push( nil )
print tableau.join( " " ) + "\n"

print "\nInverse l'ordre des éléments du tableau:\n"
tableau = tableau.reverse
print tableau.join( " " ) + "\n"
```

```
print "\nSuprime tous les éléments nil du tableau:\n"
tableau = tableau.compact
print tableau.join( " " ) + "\n"

print "\nTrie par ordre croissant les éléments d'un tableau:\n"
tableau = tableau.sort
print tableau.join( " " ) + "\n"

print "\nSuprime les doublons du tableau:\n"
tableau = tableau.uniq
print tableau.join( " " ) + "\n"

print "\nSuprime l'élément à la 2eme position et l'élément 3 du tableau:\n"
tableau.delete_at( 1 )
tableau.delete( 3 )
print tableau.join( " " ) + "\n"

print "\nSuprime tous les éléments pairs du tableau:\n"
tableau.delete_if{ |e| e % 2 == 0 }
print tableau.join( " " ) + "\n"

print "\nRempli les éléments d'un tableau sur une plage:\n"
tableau.fill( "fill", 2, 4 )
print tableau.join( " " ) + "\n"

print "\nEfface le contenu du tableau:\n"
tableau.clear

print "\nLongueur du tableau: " + tableau.length.to_s + "\n"
```

La plupart de ces fonctions restent classiques. Par contre il y a une syntaxe particulière pour delete_if.

```
tableau.delete_if{ |e| e % 2 == 0 }
```

En fait cette appel de fonction est semblable au for. En effet la variable e va prendre chaque valeur du tableau, puis entre les deux accolades on peut effectuer n'importe quelles instructions (on n'est pas limité à une seule comme le montre l'exemple). On appel cela un block de code et c'est un particularité régulièrement utilisée en Ruby.

Les Fonctions

Une fonction? Mais il semble y avoir une contradiction! En effet plus haut il a été énoncé que tout est objet en Ruby. Or une fonction n'appartient à aucune classe. En fait si, Ruby l'ajoutera à un objet global et ce sera une méthode bien qu'elle ait l'allure d'une fonction.

Pour définir une méthode il faut utiliser le mot clef def et terminer la définition avec end. En plus de permettre l'utilisation des arguments par défaut, Ruby autorise également les listes variables d'arguments, de passer un tableau à la place de plusieurs variables, de passer des blocs d'instructions et des arguments sous forme d'une table hash. Pour comprendre cela analysons le code source suivant.

```
# Argument par défaut
def arg_defaut( arg1, arg2 = 4 )
  print "Argument par défaut: "+ arg2.to_s + "\n"
end

# Tableau à la place d'arguments
def arg_tableau( a, b, c, d )
  print "Passe un tableau à la place des arguments: "
```

```ruby
  print a.to_s + " "
  print b.to_s + " "
  print c.to_s + " "
  print d.to_s + "\n"
end

# Passage d'arguments de type hash
def arg_hash( nom, param )
  print nom + ": " + param["clef 1" ] + " " + param[ "clef 2" ] + "\n"
end

# Liste variable d'arguments
def liste_variable( a, *b )
  print "Liste variable d'arguments: " + b.join( ", " ) + "\n"
end

# Utilisation d'un block de code
def bloc1( var, &block )
  print "Utilisation d'un bloc: "
  for v in var
    block.call( v )
  end
  print "\n"
end

# Utilisation d'un block de code avec yield
def bloc2( b )
  print "Autre utilisation d'un bloc: "
  if block_given?
    yield( b )
  else
    b
  end
end

arr = [ "a", "b", "c", "d" ]

arg_defaut( "Ruby c'est bien !" )
arg_tableau( *arr )
arg_tableau( "z", *arr[ 1, 3 ] )
arg_tableau( *( 1 .. 4 ) )
arg_hash( "Hash", 'clef 1' => "valeur 1", 'clef 2' => "valeur 2" )
liste_variable( 1, 2, 3, 4, 5 )
bloc1( arr ) { |l| print l + " " }
bloc2( "Les blocs en ruby" ) { |s| print s.length.to_s + "\n" }
bloc2( "c'est super bien, gniark !" ) do |s| print s.length.to_s + "\n" end
```

La définition d'un argument par défaut ce fait à l'aide d'une affectation comme en C++.

Par contre pour définir une liste variable d'arguments ce n'est pas aussi compliqué qu'en C puisque tout en mis dans un objet tableau quand la méthode reçoit les arguments. Ce tableau doit être d'éclaré avec une asterisque devant comme le montre l'exemple.

Pour ce qui est des blocs de code on peut les délimiter soit par des accolades soit par un do ... end. Ensuite la fonction récupère un objet de type Proc qui représente ce bloc de code. Pour l'éxécuter il suffit d'appeller la méthode call. Pour savoir si un bloc de code a été passé à la fonction il faut utiliser la fonction block_given?.

L'instruction return

Si aucun return n'est précisé dans une méthode, alors Ruby retournera par défaut la dernière variable utilisée.

Programmation orientée objet en Ruby

En ruby on définit une classe à l'aide du mot clef class. La fin de la classe est délimitée par le mot clef end. Voici comment définir une classe en Ruby.

```
class Ma_classe < classe_parent
  def initialize
    ...
  end

  def ma_methode
    ...
  end
end
```

L'héritage est possible à l'aide de l'opérateur plus petit que (<). Le nom de la classe doit-être une constante, c'est à dire commencer par une majuscule. Les méthodes de la classe sont définies à l'aide du mot clef def. Les objets sont créés par la méthode new et cette dernière appelle la méthode initialize. Les arguments passés à new sont automatiquement transmis à la méthode initialize. Donc la meilleure solution pour initialiser un objet est d'utiliser cette méthode initialize et non new. De plus la méthode initialize est automatiquement privée. En effet il est possible de définir trois états de visibilité pour les méthodes: public, private et protected.

```
class Ma_classe
  # Défini les accesseurs pour chaque membre
  attr_reader :var1
  attr_writer :var1
  attr_accessor :var2

  # On peut résumer ces 3 lignes par
  attr_accessor :var1, :var2

  # Variable de classe, similaire à un membre statique en C++
  @@count = 0

  # Méthode appelée par new et automatiquement privée
  def initialize( arg )
  @@count += 1
  @var1 = arg
  @var2 = ""
  puts "Initialisation"
 end

  # Méthode publique par défaut
  def ma_methode()
    puts "Ma méthode"
  end

  # Toutes les méthodes suivantes seront privées
  private

  def ma_methode_privee()
    puts "Ma méthode privée"
  end

  # Toutes les méthodes suivantes seront publiques
  public

  def var1=( var1 )
    @var1 = var1
  end
end

# Créer un nouvel objet à l'aide de la méthode new de clui-ci
obj = Ma_classe.new( "Hello" )
obj.ma_methode

# Utilisation des accesseurs
puts obj.var1
puts( obj.var1 = 6 ).to_s
```

```
obj.var2 = "Accessor"
puts "var2 = " + obj.var2

# Obtient l'id et le nom d'une classe
puts obj.id
puts obj.class

# Empêche toute modification de l'objet
obj.freeze

# Sinon on obtient un message d'erreur
begin
  obj.var2 = 4
rescue TypeError => e
  puts "Objet gelé"
end

# Erreur, méthode privée !
begin
  obj.ma_methode_privee
rescue Exception => e
  puts e.class.to_s + " caught"
end
```

Analysons ce code source.

```
attr_reader :var1
attr_writer :var1
```

Les instructions attr_reader et attr_writer déclarent des accesseurs et des mutateurs pour les membres spécifiés. Chaque membres doit-être précédé par un double points (:) et s'il y en a plusieurs ils doivent-être séparés par une virgule. L'instruction attr_writer permet de définir des méthodes comme celle-ci

```
def var1=( var1 )
  @var1 = var1
end
```

Ainsi nous pourrons plus tard utiliser ce mutateur de cette manière en faisant simplement précéder le nom de l'attribut par le nom de l'objet puis d'un point (.).

```
obj.var1 = 6
```

Mais on peut également déclarer des accesseurs et mutateur en même temps à l'aide de l'instruction attr_accessor.

```
attr_accessor :var1, :var2
```

Si aucune méthode n'est définie en tant que mutateur alors Ruby affectera automatiquement la valeur de l'opérande droite à l'attribut.

```
@@count = 0
```

Cette instruction déclare et initialise une variable de classe grâce au préfixe double arobas (@@). Ce genre de variable est similaire à un membre statique en C++. Ainsi tous les objets de la même classe (et les classes dérivées également) la partageront. Cela peut se révéler très utile pour compter le nombre d'instances d'une classe ou pour économiser de la mémoire.

La méthode class retourne un objet de type Class qui décrit le type de l'objet. Ainsi on peut obtenir le nom de la classe ou encore ses classes de base.

Ensuite le programme appelle la méthode freeze de obj. Elle permet de geler un objet pour que celui-ci ne puisse plus être modifé. Toute tentative de modification provoque une erreur qui peut-être intercepté l'aide d'une exception.

```
begin
  obj.var2 = 4
rescue TypeError => e
  puts "Objet gelé"
end
```

Il est également possible d'intercepter une exception lorsque l'on appelle une méthode privée. Ceci nous aùène à ll'étude des exceptions en Ruby.

Les exceptions

Voici la structure d'un code gérant les exceptions.

```
begin
...# Code succeptible de lancer une exception
rescue Type_exception => e then
...# Code éxécuté si une exception est capturée
else
..# Code si aucune exception n'a été capturée
ensure
...# Code éxécuté dans tous les cas
end
```

Le code qui est succeptible de générer une exception doit-être placé dans une instruction begin. Puis le code traîtant les exceptions est contenu dans le bloc rescue. Si aucun type d'exception n'est indiqué alors toutes les exceptions seront interceptées. Le then situé après le type d'exception est obligatoire s'il y a d'autres instructions avant la fin de la ligne. Et enfin les deux derniers bloc else et ensure sont facultatifs. Le code du else est éxécuté s'il n'y a pas eu d'exception. Quant au bloc ensure il sera interprété dans tous les cas. Pour lever une exception il est nécessaire d'utiliser l'instruction raise suivit de l'exception. Maintenant nous somme capable de comprendre le code source suivant.

```ruby
#! /usr/bin/ruby

# Intercepte une division par 0
begin
  puts 5 / 0
rescue ZeroDivisionError => e
  puts e
end

# Intercepte 2 types d'exception et ferme le fichier
# s'il a pu être ouvert
begin
  f = File.new( "fichier", "r" )
  rescue Errno::ENOENT, Errno::EACCES => e
  puts e
else
  # Exécuté s'il n'y a pas eu d'exception
  f.close
  ensure
  puts "Il ne reste plus de fichier ouvert"
end

# Lance une exception si val est inférieur à 10
def fct_inutile( val )
  raise Exception.new( "Inférieur à 10" ) if val < 10
end

i = 0
debut = 5

# Utilisation de l'instruction retry
begin
  debut.upto( 15 ) do |i|
    puts i
    fct_inutile( i )
  end
rescue Exception => e
  puts e
  debut = 10
  retry# Recommence l'éxécution du bloc begin
end

# Intercepte un appel d'exit
begin
  exit( 0 )
rescue SystemExit => e
  puts "L'appel de la méthode exit n'aura pas d'effet car l'exception " +
  "SystemExit a été interceptée"
end

# Une exception personalisée
class MonException < Exception
  attr_reader :time

  def initialize
    @time = Time.new
  end
end

# Démonstration d'un bloc ensure
begin
  raise MonException.new
  rescue MonException => e
  puts "MonException lancée le " + e.time.to_s
  exit( 0 )
ensure
  puts "Le bloc ensure est toujours éxécuté, même après l'appel de exit"
end

puts "Cette instruction ne sera pas éxécutée"
```

L'éxécution de ce script produit cette sortie.

```
divided by 0
No such file or directory - fichier
Il ne reste plus de fichier ouvert
5
Inférieur à 10
10
11
12
13
14
15
L'appel de la méthode exit n'aura pas d'effet car l'exception SystemExit a été interceptée
MonException lancée le Sat Feb 19 11:44:48 CET 2005
Le bloc ensure est toujours éxécuté, même après l'appel de exit
```

On remarquera que l'instruction retry a permis de recommencer l'éxécution du bloc begin. Avant d'appeller retry on aura pris soin d'affecter la valeur 10 à la variable début afin d'éviter une boucle infinie. Cette instruction est à la fois pratique pour recommencer une action afin que le programme se déroule sans erreur, mais elle est en même temps dangereuse. En effet il est important de réinitialiser les paramètres qui ont provoqués cette exception pour ne pas créer de boucle infine.

Pour lever une exception il existe plusieurs syntaxes.

```
raise classe, message
raise objet
raise message
raise
```

Si aucune classe n'est spécifiéé Ruby utilisera RuntimeError par défaut. Le simple appel de raise sans paramètres dans un bloc rescue relève l'exception courante.

Surcharge des opérateurs

Pour surcharger un opérateur il suffit de définir une méthode ayant le même nom que l'opérateur. Ceci est valable pour tous les opérateurs surchargeables (voir le tableau des opérateurs) excepté le + et le & unaires. Dans ce cas on écrira +@ ou -@.

Voici un code source qui surcharge l'opérateur égal.

```
#! /usr/bin/ruby

class Personne
  attr_accessor :nom
  attr_accessor :prenom

  def initialize
    @nom = ""
    @prenom = ""
  end

  def ==( droite )
    return false if self.class != droite.class
    return false if @nom != droite.nom
```

```
    return @prenom == droite.prenom
  end

  def to_s
    @nom + " " + @prenom
  end
end

p1 = Personne.new
p2 = Personne.new
p3 = Personne.new

p1.nom = "Yukihiro"
p1.prenom = "Matsumoto"

p2.nom = "Yukihiro"
p2.prenom = "Matsumoto"

p3.nom = "Stroustrup"
p3.prenom = "Bjarne"

puts p1.to_s + " et " + p2.to_s + " est la même personne." if p1 == p2
puts p1.to_s + " et " + p3.to_s + " sont deux personnes différentes." if p1 != p3
```

Modules

Les modules sont comme des classes mais il n'est pas possible de les instancier. Par contre on peut y définir des méthodes et des variables. La définition d'un module se fait à l'aide de l'instruction module suivit par son nom. Evidemment le nom doit être une constante, donc le premier caractère sera une majuscule. Lorsque que l'on définit un module déjà existant les nouvelles fonctionalités seront ajoutées au module existant. Pour définir une méthode dans un module il faut faire précéder le nom de la méthode par celui du module suivit d'un point (.).

```
Module Nom_module
  def Nom_module.nom_méthode
    ...
  end
end
```

Attention

Lorsqu'une classe inclut plusieurs modules qui comportent les même membres alors le comportement du code ne sera pas forcément celui attendu. Ruby utilisera le membre du premier module inclus.

Les modules offrent une possibilité intéressante qui se nomme les mix-ins. Cela permet d'inclure les méthodes d'un module dans une classe. La classe pourra également utiliser les attributs déclarés dans le module. Cette opération s'effectue à l'aide de l'instruction include suivit par le nom du module à inclure.

```
class Ma_classe
  include Mon_module
    ...
end
```

Ainsi Ma_classe aura accès à toutes les méthodes de Mon_modules et à ses attributs également.

Les mixins sont utiles lorsque plusieurs classes utilisent le même code. Imaginons que nous voulons créer une sorte de STL en Ruby. Donc nous allons écrire des classes Vecteur, Liste, Hachage etc... Ces classes auront certaines choses en communs comme des itérateurs par exemple. Il nous permettent de parcourir un Vecteur de la même manière qu'une Liste. Evidemment nous allons devoir besoin de coder des méthodes pour le tri, la recherche, etc... La première idée qui vient à l'esprit est de coder une version de ces méthodes pour chaque classe que nous avons écrite. Malheureusement ce n'est pas du tout la bonne approche. En effet nous allons dupliquer un code similaire dans toutes ces classes. Cette pratique n'est pas bonne pour la maintenance du code puisque chaque modification sur l'algorithme de tri devra être appliquée plusieurs fois. Et si plus tard si nous décidons de créer un nouveau conteneur on devra réécrire une méthode de tri, de recherche etc... ce qui rajoute du code et donc de refaire des tests. Dans ce cas là les mixins se révèlent particulièrement appréciables puisqu'ils permettent d'utiliser la même méthode sur tous les conteneurs à partir d'un code générique pour chaque algorithme.

```ruby
# tri.rb

module Tri
  def croissant
    # ...
  end

  def decroissant
    # ...
  end
end

# conteneurs.rb
require 'tri'

module Recherche
  def rechercher
    # ...
  end
end

class Liste
  include Tri
    # ...
end

class Vecteur
  include Tri
  # ...
end
```

L'instruction require permet d'indiquer à Ruby d'inclure le fichier spécifié.

Voici un autre code source utilisant les mixins. Nous voulons transformer un objet sous forme xml. Ainsi il devra s'auto documenter (nom des variables, nom de la classe etc...). Il serait trop lourd de faire une classe de base d'un objet sérialisable en xml. La meilleure solution est de créer un module qui, grâce à la reflexion, va pouvoir accéder à tous les attributs de l'objet à traduire sous forme xml. Cette solution est transparente pour les autres programmeur qui voudront utiliser notre module puisque la seule contrainte est d'inclure le module.

```ruby
# Transforme un objet sous forme xml
module Obj2xml
  def to_xml
    buffer = "<ruby-object class-name='" + self.class.to_s + "'>\n"
    # Récupère la liste des variables d'instances
    var_list = instance_variables
    # Met dans buffer le nom et la valeur de chaque variable d'instance
    var_list.each do |var_name|
      buffer += "<attribute name='" + var_name + "' value='" +
```

```
      instance_variable_get( var_name ) + "'/>\n"
    end
    buffer += "</ruby-object>"
  end
end

# Classe de test
class Song
  include Obj2xml
  attr_accessor :title
  attr_accessor :author
  attr_accessor :album
end

s = Song.new
s.author = "Bob Marley"
s.title = "She's gone"
s.album = "Kaya"

puts s.to_xml
```

Et voici la sortie du script.

```
<ruby-object class-name='Song'>
<attribute name='@title' value='She's gone'/>
<attribute name='@album' value='Kaya'/>
<attribute name='@author' value='Bob Marley'/>
</ruby-object>
```

Ensuite il sera très facile de reconstruire cet objet à l'aide de ce document xml.

Les outils de Ruby

Ruby est fournit avec un débuggueur, un profiler et un tracer. Pour les utiliser il suffit simplement de charger le module au moment de l'éxécution du script.

```
ruby -r debug fichier.rb
ruby -r profile fichier.rb
ruby -r tracer fichier.rb
```

Pour afficher la pile des appels d'une certaine partie du code au lieu de tout le programme il est nécessaire d'appeler les méthodes suivantes.

```
Tracer.on {
   ...
}

# Ou bien

Tracer.on
...
Tracer.off
```

Il existe également un interpréteur Ruby interactif nommé irb. Irb éxécute les lignes les une après les autres et il se révèle particulièrement utile pour de tester le comportement des méthodes et des classes.

Liens

- La documentation de référence sur ruby, [1] (*http://www.ruby-doc.org/core/*)
- Ces deux liens regroupent un grand nombre de bibliothèques et projets en Ruby, [2] (*http://raa.ruby-lang.org/*), [3] (*http://rubyforge.org/*)
- Un framework de développement d'applications web, [4] (*http://www.rubyonrails.org/*)
- Un parseur xml en pur Ruby (fournit par défaut à partir de la version 1.8), [5] (*http://www.germane-software.com/software/rexml/*)
- Un parseur xslt (surcouche de libxml), [6] (*http://rubyfr.net/projets/ruby-xslt.asp*)
- Brique de base comme moteur de servlet, [7] (*http://www.webrick.org/*)
- Permet d'embarquer du code Ruby dans du HTML, [8] (*http://www.modruby.net/en/*)