Atelier du libre
Programmer avec Ruby

Paul Rivier paul·r·ml∂gmail·com
Pierre Paysant-Le Roux pplr∂free·fr
GFDL

2009

Introduction à la programmation et à Ruby

Plan

Informatique

Étymologie

Informatique

De la contraction des mots ***« information »*** *et* ***« automatique »****. Inventé par Philippe Dreyfus en 1962, il fut officiellement consacré par Charles de Gaulle qui trancha lors d'un Conseil des ministres entre « informatique » et « ordinatique ».*

Définition

Informatique, nom commun, féminin

Domaine des concepts et autres techniques employées pour le traitement automatique de l'information.

Motivations

calcul	répéter rapidement un traitement de données
réseaux	communiquer à distance
données	expression relationnelle, duplication, recherche
...	*beaucoup d'autres choses*

Histoire de l'informatique

Histoire ancienne : systèmes mécaniques

Avant XVI^{e}	bouliers, régles de calculs
XVI^{e}- XVII^{e}	premières calculettes mécaniques addition/soustration (type « pascaline »)
XVIII^{e}- XX^{e}	calculettes élaborées, constructions de tables (logarithme, trigonométrie)
XVIII^{e}	première machine programmable, un métier à tisser qui lit un ruban perforé

Histoire moderne : systèmes électroniques

1930 - 1956 calculateur électroniques à lampes

1956 - 1971 calculateurs électroniques programmables à transistors

1970 - aujourd'hui micro-informatique

Histoire moderne : les langages

1950 - 1970 Fortran, LISP, ALGOL, COBOL, Simula, BASIC

1970 - 1980 C, Smalltalk, Prolog, ML

1980 - 2000 C++, Eiffel, Haskell, Ruby, ANSI CL ...

Depuis la micro-informatique, le support materiel a peu évolué, et le travail sur le techniques de programmation logicielle a acceléré.

Les langages

Les usages des langages

système [noyaux, pilotes] Priorité au contrôle du materiel et à la vitesse (C ou C++)

calcul [algorithmes, simulations] Priorité à la vitesse de calcul et à la gestion de la mémoire

applications [navigateur, mail, dessin, ...] Priorité à l'expressivité et à l'abstraction (Ruby)

Ruby est un langage open-source dynamique qui met l'accent sur la simplicité et la productivité.

Concepts élémentaires

Le fichier source

- un « langage » est un programme dont l'entrée est un fichier source
- on peut utiliser plusieurs fichiers sources en les référençant

Mon premier programme

- ouvrir un éditeur de texte
- taper `puts "J'aime les ateliers du libre"`
- enregistrer dans `chemin/vers/fichier.rb`
- ouvrir une console et taper `ruby chemin/vers/fichier.rb`
- insérer en première ligne `#!/usr/bin/ruby -w`, puis faites `chmod +x fichier.rb` pour pouvoir faire `./fichier.rb`

Avant d'aller plus loin

Le développement se fait avec un gestionnaire de sources. Ce dernier va vous permettre de :

- programmer de façon incrémentale
- revenir en arrière en cas de problème
- maintenir plusieurs axes de développement concurrents
- assembler ces axes si besoin
- publier votre code
- et bien plus ...

Mercurial

Allez dans votre dossier `chemin/vers/` puis tapez

- `hg init`
- `hg add fichier.rb`
- `hg ci -m "mon premier programme"`
- `hg view`

L'instruction

- une instruction est un mot reconnu par le langage
- en ruby, les instructions principales sont `alias`, `and`, `begin`, `break`, `case`, `class`, `def`, `do`, `else`, `elsif`, `end`, `false`, `for`, `if`, `in`, `module`, `next`, `nil`, `not`, `or`, `return`, `self`, `super`, `then`, `true`, `unless`, `until`, `when`, `while`, `yield`

Entrées et sorties

entrée le programme acquiert de l'information depuis l'extérieur

sortie l'extérieur acquiert de l'information depuis le programme

Entrées et sorties (garder #!...)

```
puts "ecris ton prenom et tape entree"
prenom = gets.chomp
puts "ton prenom est #{prenom}"
```

Unités logiques et nommage

variable conteneur de données

fonction conteneur d'instructions paramétrable

Variables et fonctions (tout virer)

```
def bonjour(texte)
  return "Bonjour, " + texte
end

ma_var = "J'aime les ateliers du libre."
puts bonjour(ma_var)
```

Types

- chaque variable a un type
- chaque type a sa collection de méthodes
- les comparaisons doivent utiliser des types compatibles
- types de base : entier, nombre décimal (approché), chaîne de caractères, booléen (vrai ou faux), tableau ...

Types (à essayer dans irb)

```
5.class
"bonjour".class
5>4
5>"bonjour"
[2, 4, 6]
```

Algorithmique

- controle d'exécution
- branchements conditionnels
- boucles

Boucles et branchements (tout garder)

```
puts "nombre de repetitions ?"
n = gets.chomp.to_i
if n>5 then
  puts "c'est trop"
  n=5
end
n.times do
  puts bonjour(ma_var)
end
```

Les tableaux

- un tableau est une collection ordonnée
- en ruby, un tableau peut contenir tous types d'éléments, mélangés
- y compris d'autres tableaux

Les tableaux

Les tableaux (tout virer sauf def bonjour)

```
continuer=true
tableau=Array.new
while continuer
  t=gets.chomp
  if t!=""
    tableau.push t
  else
    continuer=false
  end
end
tableau.each { |t| puts bonjour(t) }
```

Chaînes de caractères

- utiles pour récupérer une entrée ou préparer une sortie

Chaînes de caractères (dans irb)

```
a="Bonjour"
a.size
a.upcase
b="salut les amis"
b.split(" ")
b.gsub("amis","libristes")
```

Structures associatives

- structures conçues pour associer une valeur à une clef
- le tableau associatif est le plus simple
- la table de hashage est optimisée pour les grandes collections

Structures associatives (dans irb)

```
h=Hash.new
h["paul"]="Paul parle des structures associatives"
h["christophe"]="Christophe regarde rigolant"
h
puts h["paul"]
puts h["christophe"]
```

La généricité avec Ruby

Plan

Motivations

À partir de la base « programmation impérative » :

- diminuer le nombre de lignes de code
- regrouper les unités logiques
- fiabiliser le programme
- abstraire les traitements
- rendre le programme modulaire
- simplifier la maintenance et l'évolution

Mécanismes/paradigmes actuels et origines

- la fonction
- la programmation orientée objet (inspiration SmallTalk)
- les aspects fonctionnels (inspiration LISP)
- la métaprogrammation (inspiration LISP)

Programmation orientée objet

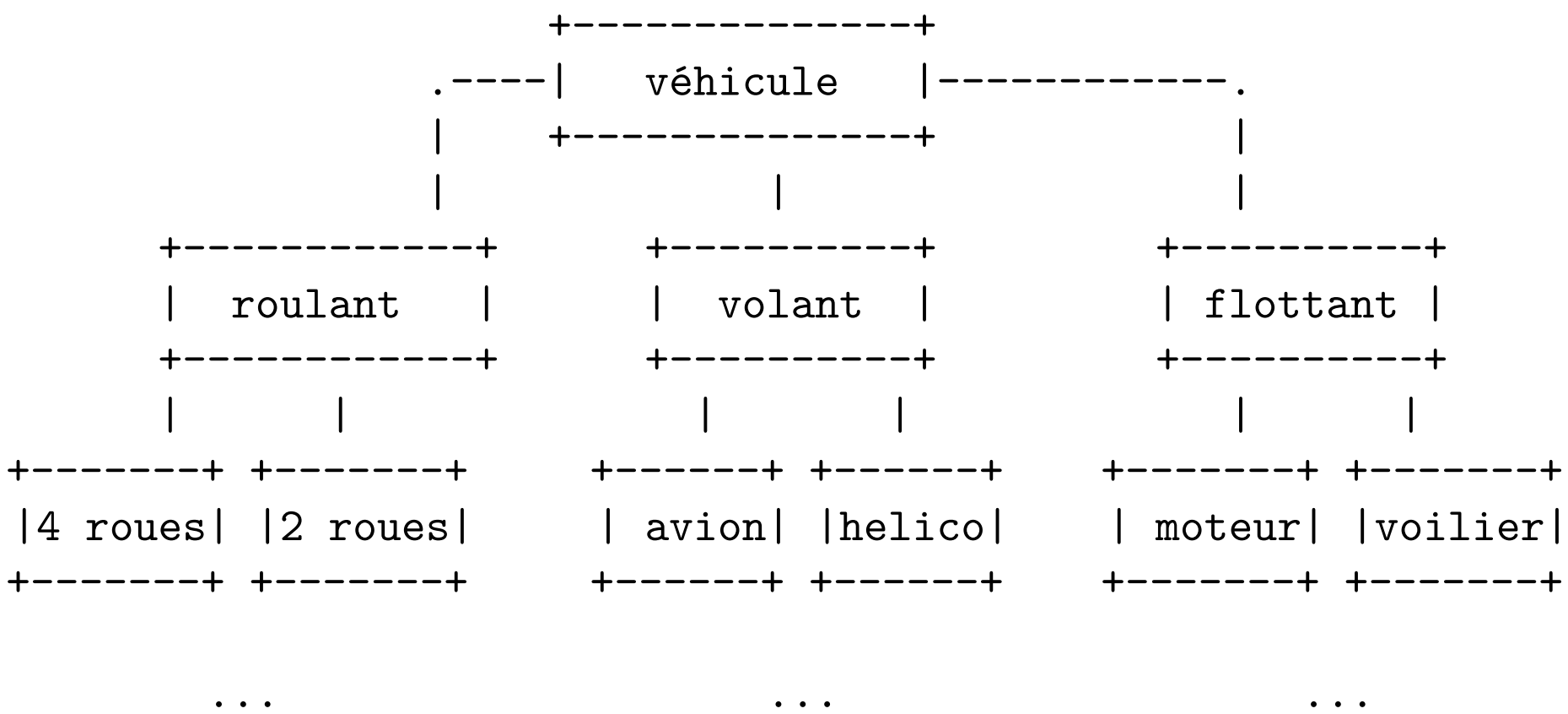
Motivations de la P.O.O

- confiner les effets de bords dans des conteneurs d'état
- en profiter pour y caser aussi les mécanismes de transitions
- proposer d'abstraire la structure de données interne (état) et ses mécanismes de transitions derrière une *interface* plus jolie
- étendre le typage à ces conteneurs
- avoir une convention permettant de réutiliser un type de conteneur générique pour en créer un spécifique, par spécialisation

Vocabulaire de la P.O.O

- une *classe* définit la structure de données (état), les fonctions de transitions d'états et les fonctions d'interface
- les *attributs* sont les variables propres à un conteneur, ils en forment donc la structure de donnée (état)
- une *instance* est un conteneur, une forme de variable créée en *instanciant* une classe, modelé d'aprés la définition de cette dernière
- l'*interface* est l'ensemble des méthodes accessibles pour que l'extérieur manipule l'instance
- une classe A *hérite* d'une classe B lorsque A est une spécialisation de B, qui se contente d'exprimer sa différence
- une classe A *agrège* une classe B lorsque A utilise un attribut de type B pour sa structure de données

Un exemple, un exemple !



Du code, du code !

définition de l'arbre d'héritage

```
class Vehicule
  attr_accessors :vitesse_maxi, :poids, :milieu
  ...
end

class VehiculeRoulant < Vehicule
  attr_accessors :nombre_de_roues
  def initialize
    super
    @milieu = "terre"
  end
end
```

Du code, du code !

définition de l'arbre d'héritage (suite)

```
class DeuxRoues < VehiculeRoulant
  def initialize
    super
    @nombre_de_roues = 2
  end
end

class Velo < DeuxRoues
  attr_accessor :categorie
  ...
end
```

Du code, du code !

surcharge des méthodes

```
class Vehicule
  def avancer
    raise "Not Implemented"
  end
end
class Voiture < QuatreRoues
  def avancer
    accelerer
  end
end
class Velo < DeuxRoues
  def avancer
    pedaler
  end
end
```

Du code, du code !

créer une instance, régler les attributs

```
peugeot_404 = Voiture.new
peugeot_404.marque = "Peugeot"
peugeot_404.modele = "404"
peugeot_404.vitesse_maxi = 200
peugeot_404.position = 0
peugeot_404.avancer_a_fond(30*60)  # 30 minutes
puts peugeot_404.position     #=> 100
```

Les types

Si le type B hérite du type A, alors B est aussi de type A

- class Parking ...
- def ajouter_vehicule(véhicule_roulant) ...
- refuser si véhicule_roulant.poids $>$ LIMITE ...
- véhicule_roulant.rouler(100 metres) ...

Ruby et l'objet

Ruby, comme SmallTalk, pousse l'objet très loin :

- tout en ruby est un objet (exceptions, classes et fonctions comprises), et tous les types héritent de la même classe "Object"
- toute la librairie standard est orientée objet
- les classes et les objets sont manipulables à la volée (ex. : on peut ajouter ou modifier une méthode dans Array en faisant class Array ; def méthode ... ; end)

Aspects fonctionnels

Ruby et les fonctions

en ruby, les méthodes sont des objets

- que l'on peut appeler (directement ou via .call)
- que l'on peut passer en paramètre
- qui préservent le contexte lexical

Ruby et le type « Proc »

on peut créer un objet de type « bloc de code » à tout moment

- en utilisant lambda (recommandé)
- en utilisant Proc.new
- via une fonction qui demande un bloc de code en paramètre

Ruby et les lambdas

fonctions pour renvoyer le carré des éléments d'un *tableau*

```
# beuuuuuurk !!
tableau2=Array.new
for i in 0..(tableau.size-1) do
  tableau2[i] = tableau[i]**2
end
tableau2

# mieux ...
tableau2=Array.new
tableau.each do |elem|
  tableau2.push elem**2
end
tableau2
```

Ruby et les lambdas

les effets de bords nuisent à l'expressivité, il vaut mieux utiliser :

```
# bien :
tableau.map{ |elem| elem**2}
```

- pas d'accumulateur temporaire nommé (tableau2)
- pas de variable d'itération nommée (i)
- court et lisible

Array#map (la fonction map d'une instance Array) prend en paramètre un lambda d'arité 1 et renvoit le tableau image de ses éléments par ce lambda.

```
l = lambda { |p| p + 5 }
[1,2,3].map(&l)      #=> [6,7,8]
```

Ruby et les lambdas

Implémentation de map : my_map

```
class Array
  def my_map(&block)
    temp = Array.new
    self.each {|elem| temp.push yield(elem) }
    temp
  end
end
[1,2,3].my_map{|e| e**3}    #=> [1, 8, 27]
```

Ruby et les lambdas

Implémenter une fonction Array#all_match qui retourne true si tous les éléments respectent une condition, false sinon

```
class Array
  def all_match(&block)
    self.map{|e| yield e}.fold_left{|ac,n|
                                    ac and n}
  end
  def fold_left(&block)
    a = self.first
    self[1..-1].each{|i| a = yield(a, i) }
    a
  end
end

[5,6,7].all_match{|e| e>5}   #=> false
[5,6,7].all_match{|e| e>=5}  #=> true
```

Un mot sur Ruby et les fermetures

attribution en fonction du contexte lexical

```
def get_a_accessors
  a = 0
  return [lambda {a}, lambda {|new_a| a=new_a }]
end
reader, writer = get_a_accessors

a = 5
reader.call       #=> 0
writer.call(100)
reader.call       #=> 100
```

Métaprogrammation

Qu'est-ce que la métaprogrammation ?

- Des programmes qui manipulent des programmes
- Permet de modifier la structure du programme pendant l'exécution du programme
- Ruby est son propre métalangage

Ruby est un langage réflexif

Introspection

Capacité à connaître son propre état

Intersession

Capacité à modifier son état

L'introspection

Connaître son état ?

- Les instances qui composent le programme
- Les classes et leur hiérarchie
- Le contenu des instances
- Les méthodes des instances

Exemple : recherche des instances du programme

```
s = Time.now
=> Tue Sep 15 13:37:40 +0200 2009
ObjectSpace.each_object(Time){|s| puts s}
Tue Sep 15 13:37:40 +0200 2009
=> 1
```

Exemple : introspection d'une classe

```
Float.superclass
=> Numeric
Float.ancestors
=> [Float, Precision, Numeric, Comparable, Object,
    Kernel]
```

Autres méthodes

instance_methods, singleton_methods, constants

Exemple : introspection d'une instance

```
Object.new.methods
=> ["inspect", "tap", "clone", "public_methods",
    "__send__", "object_id",
    "instance_variable_defined?", "equal?",
    "freeze", "extend", "send", "methods", "hash",
    "dup", "to_enum", "instance_variables", "eql?",
     "instance_eval", "id", "singleton_methods",
    "taint", "enum_for", "frozen?",
    "instance_variable_get", "instance_of?",
    "display", "to_a", "method", "type", ...]
```

Autres méthodes

respond_to?, class, kind_of?, instance_of?, instance_variables, send

Rappel

Modification du comportement du programme en fonction de son état.

Les mécanismes d'intersession

- Modification locale du comportement
- Macros
- Callbacks

Modification locale

Modification du comportement d'une instance seulement : les métaclasses

```
str = "Ateliers du libre"
=> "Ateliers du libre"
str.to_s
=> "Ateliers du libre"
def str.to_s
  format("### %s ###", self)
end
=> nil
str.to_s
=> "### Ateliers du libre ###"
"Ateliers du libre".to_s
=> "Ateliers du libre"
```

Les macros

- Méthode de classe qui créé des méthodes
- Automatise la création de code redondant
- Rend le code plus simple à comprendre

Accéder au contexte lexical courant depuis la définition

- `eval` : beurk
- Accès transparent au contexte lexical de définition (contrairement à `def`)

Exemple de macro : attr_reader

Définie une méthode pour lire la variable d'instance

```
class Module
    def attr_reader(variable)
        class_eval do
          define_method variable do
            instance_variable_get("@#{variable}")
          end
        end
     end
   end
end
```

Les callbacks

Méthodes appelées automatiquement lors d'évènements clefs

- nouvelle méthode définie (Module#method_added)
- classe héritée (Class#inherited)
- module inclus (Module#included)
- appel d'une méthode inexistance (Object#method_missing)

Exemple : génération de code HTML

```
class FastHtml

  def method_missing(meth, &block)
    print_element(meth, &block)
  end

  def print_element(element, &block)
    puts format("<%s>", element)
    puts block.call
    puts format("</%s>", element)
  end

end
```

```
html = FastHtml.new
html.body{
  html.h1{"Ateliers du libre"}
  html.p{
    "Ruby aime la metaprogrammation"
  }
  html.ul{
     html.li{ "el1"}
     html.li{ "el2"}
  }
}
```

En allant plus loin

La libraire Markaby

```
mab = Markaby::Builder.new
  mab.html do
    head { title "Boats.com" }
    body do
      h1 "Boats.com has great deals"
      ul do
        li "$49 for a canoe"
        li "$39 for a raft"
        li "$29 for a huge boot that floats and can
            fit 5 people"
      end
    end
  end
  puts mab.to_s
```