Langages de script

Licence professionnelle ASRALL

Auteur : Philippe Dosch

Date : 2013/2014

UNIVERSITÉ DE LORRAINE
INSTITUT UNIVERSITAIRE DE TECHNOLOGIE
2 ter boulevard Charlemagne
CS 5227
54052 • NANCY cedex
————————
Tél : 03.83.91.31.31
Fax : 03.83.28.13.33
http://iut-charlemagne.univ-nancy2.fr/

Sommaire

Document réalisé sous Linux avec (X)Emacs et compilé grâce à XeLaTeX. Style de présentation librement inspiré de la classe `joinreport` (©Jonathan Aceituno).

CHAPITRE 1

Introduction

SECTION 1

Les langages de script

Introduction

— Langages de programmation généralement destinés à être interprétés
— Permettent de raccourcir sensiblement les développements (évitent « édition/compilation/édition de liens/exécution »)
— Bien adaptés à l'écriture de « petits » programmes, notamment ceux destinés à l'administration système (même si certains permettent aussi l'écriture d'applications complètes)

Particularités des langages de script

— Ne nécessitent pas d'étape de compilation (même si certains le permettent cependant)
— Instructions interprétées au fur et à mesure de chaque exécution d'un programme
— En contrepartie, utilisent généralement plus de mémoire et s'exécutent plus lentement que des programmes compilés
— Généralement assez flexibles : typage faible, types et opérations de haut-niveau, permettant de simplifier et d'accélérer l'écriture de programmes
— Développements plus rapides, favorisant en particulier le prototypage
— Bien adaptés à l'interfaçage de plusieurs composantes logicielles hétérogènes (autres langages, frameworks)
— Facilement interfaçables avec des bibliothèques graphiques (réalisation d'interfaces homme-machine)
— Parfois conçus pour un domaine spécifique, ce qui n'empêche généralement pas de les utiliser pour écrire des programmes de portées plus générales

Usage des langages de script

— Largement utilisés pour l'administration système, les *front-end* graphiques, le prototypage, les programmes *batch*…
— Également utilisés lorsqu'il y a beaucoup d'interactivité à gérer ainsi que de nombreuses commandes externes à exécuter : c'est typiquement le cas des *shell* UNIX
— Langages généralement portables, hormis ceux qui sont des *shell* UNIX
— Les langages de scripts les plus évolués (PHP, Python, Ruby) tendent à avoir des caractéristiques

à mi-chemin entre les langages de script traditionnels et les langages compilés typiques

Taxonomie des langages de script

- Shell UNIX : sh/bash, csh, ksh, zsh…
- Système, fichiers textes : Tcl, Perl
- Fonctionnels : LISP, Scheme
- Généralistes, orientés objet : Python, Ruby
- Orientés Web
  - historiquement : Perl
  - spécifiquement : JavaScript, PHP, ASP
  - plus récemment, avec des bibliothèques évoluées : Python (zope/plone, django…), Ruby (rails, nitro…)

SECTION 2

Quels langages pour l'administration système ?

Les shell UNIX (sh/bash, csh,ksh)

- L'interface en mode console proposé sur tout système UNIX (la ligne de commande)
- Disponibles en standard sur toute distribution UNIX
- Sont à la base de la majeure partie des scripts systèmes
- Supportent les variables et quelques structures de contrôles
- Ont une syntaxe stricte et pauvre
- Peu d'opérateurs et de fonctions disponibles (reposent surtout sur les commandes externes)
- Doivent impérativement être maîtrisés par tout administrateur

Perl

Langage de programmation créé en 1987 par Larry Wall, inspirées de C, sh, AWK, LISP…

- Langage plus riche et plus flexible que les shells (structures de contrôle, gestion des fichiers…)
- Support des tableaux, des tables associatives, des expressions régulières…
- Langage puissant à la syntaxe parfois obscure (hackers)
- Disposant de nombreuses bibliothèques (environ 500 !) permettant la création de scripts systèmes (sockets, réseau, utilisateurs, mémoire partagée, SGBD, CISCO…)
- Communauté d'utilisateurs très active, notamment chez les administrateurs

Python

Langage de programmation orienté objet créé en 1991 par Guido van Rossum, inspiré de Perl, Scheme, Smalltalk, Tcl…

- Langage haut-niveau : types de données évolués, structures de contrôle sophistiquées, programmation fonctionnelle…
- Syntaxe plus lisible que Perl, indentation obligatoire pour la définition des blocs
- Expressions régulières, tests unitaires, ramasse-miettes
- Nombreuses bibliothèques, supportant des formats (MIME), des protocoles (HTTP, FTP), les interfaces graphiques, les bases de données…
- Utilisé pour de nombreuses applications, systèmes et autres

CHAPITRE 2

Le langage Ruby

SECTION 1

Introduction

Historique

- Langage lancé au Japon en 1993 par Yukihiro Matsumoto dont la première version a été distribuée en 1995
- Ruby est un langage de script de haut niveau orienté objet, dont la syntaxe est inspirée de Perl, Smalltalk, Lisp
- La syntaxe résultante est simple et lisible
- La philosophie du langage met en avant les besoins humains plus que ceux des ordinateurs
- L'objectif est ainsi de créer des programmes extrêmement clairs, dont l'exécution soit la plus fidèle possible à celle que l'on s'en fait intuitivement à la lecture d'un code source
- Licence GPL, disponible sur toute plate-forme

Généralités

- Ruby est un langage orienté objet dans lequel **tout** est objet !
- Il n'y a aucune fonction, il n'existe que des méthodes !
- Toutefois, des méthodes sont parfois invoquées sans objet receveur. Un objet est alors implicitement utilisé : l'objet courant `self`
- Les commentaires
  - commencent au caractère # et finissent en fin de ligne
  - peuvent être placés entre des lignes *commençant* respectivement par `=begin` et `=end` s'ils tiennent sur plusieurs lignes

Les fins de ligne...

- Les instructions se terminent « naturellement » en fin de ligne (pas besoin de ;)
- Si l'interpréteur détermine que l'instruction courante est incomplète (p. ex. elle se finit par une , ou un opérateur), il cherche automatiquement à la compléter avec la ligne suivante
- Un *backslash* (\) doit être utilisé si on souhaite continuer une instruction sur la ligne suivante alors qu'elle se termine « bien » sur la première…

```ruby
a = 1 + 2 +       # '\' inutile
    3             #
```

```ruby
a = 1 + 2 \       # '\' indispensable
    + 3           #
```

Conventions

— Les conventions de nommage suivantes sont **obligatoires**, le langage détermine ainsi comment utiliser un identificateur donné

Variable locale	`nom`, `nom_compose`
Méthode	`nom`, `nom_compose`
Paramètre	`nom`, `nom_compose`
Classe	`Nom`, `NomCompose`
Variable d'instance	`@nom`, `@nom_compose`
Variable de classe	`@@nom`, `@@nom_compose`
Variable globale	`$nom`, `$NOM`, `$nom_comp`, `$NOMCOMP`
Constante	`Nom`, `NOM`, `NomCompose`, `NOMCOMPOSE`
Symbole	`:nom`, `:nom_compose`

— Conventions de nommage moins critiques, mais néanmoins systématiquement observées
  — Méthode qui questionne : `nom?`, `nom_compose?`
  — Méthode dangereuse : `nom!`, `nom_compose!`
— Autre convention : les parenthèses marquant les appels de méthodes peuvent être omises si le contexte n'est pas ambigu

Entrées-sorties de base

— Pour **afficher** : `puts`

```ruby
puts("Bonjour")
puts(2+3)
```

ou encore :

```ruby
puts "Bonjour"
puts 2+3
```

— Pour **lire** : `gets`

```ruby
a = gets
puts a
```

ou, pour supprimer au passage le retour à la ligne et les espaces inutiles de la saisie :

```ruby
a = gets.strip
puts a
```

Exécution de commandes externes

Ruby permet d'exécuter des commandes externes et de récupérer les sorties standards de ces commandes

— la méthode `system` permet d'exécuter une commande dans un sous-processus (la sortie standard de la commande se confond avec celle de Ruby)

— il est aussi possible d'exécuter une commande entre une paire de ‘ (la sortie standard est alors renvoyée et peut être affectée à une variable)

Dans tous les cas, la variable globale `$?` contient le code de retour de la commande exécutée

```
1 proc = ‘ps aux‘
2 puts "Commande exécutée avec code de retour #{$?}"
3 puts proc
```

Un premier exemple

```
1 #!/usr/bin/ruby -w
2
3 puts "Quel est ton nom :"
4 nom = gets.strip
5 puts "Quelle est ton année de naissance :"
6 date = gets.strip.to_i
7 age = Time.now.year - date
8 puts "Salut #{nom}, c'est donc l'année de tes #{age} ans ?"
```

Exemple à placer dans un fichier à extension `.rb` et à rendre exécutable (`chmod +x`). Il est également possible d'utiliser `irb`, qui permet d'utiliser Ruby interactivement

Documentation

— Le site de référence
`http://www.ruby-lang.org/`
— L'API en ligne
`http://www.ruby-doc.org/core/`
— La forge dédiée
`http://rubyforge.org/`
— Site francophone sur Ruby
`http://rubyfr.org/`
— Site tutorial
`http://tryruby.hobix.com/`
— Programming Ruby (pragmatic)
`http://www.rubycentral.com/book/`

SECTION 2

Gestion des données

Variables

— Pas de déclaration !
— Tout est objet en Ruby, **toute** variable est donc… un objet !
— Les conventions de nommage vues précédemment doivent être respectées
— Les variables ne sont pas typées en Ruby, elles peuvent donc contenir des valeurs de *types* différents au cours d'une même exécution

Types standards : les nombres

- Entiers
  - longueur arbitraire (la RAM est la seule limite)
  - 2 représentations en interne (`Fixnum` et `Bignum`) **transparente** pour les utilisateurs
  - exemples : `6`, `1254`, `14_147` (_ ignoré), `-16`, `0x45A` (hexa), `055` (octal), `0b100100` (binaire)
- Réels
  - représentés comme des objets de la classe `Float` (i.e. le type C `double` de l'architecture native)
  - un nombre est automatiquement réel s'il comporte un point décimal ou une notation scientifique
  - exemples : `12.`, `1.5e5`, `125.e03`

Types standards : les chaînes de caractères

- Les chaînes de caractères sont des objets de type `String`
- Les chaînes sont souvent créées de manière littérale, de plusieurs manières différentes
  - avec le délimiteur `'` : la chaîne n'est pas interprétée et il existe 2 caractères spéciaux (`\\` → `\` et `\'` → `'`)

```
'Hello'
'C\'est aujourd\'hui'
```

  - avec le délimiteur `"` : une vingtaine de caractères spéciaux sont supportés (`\n`, `\"`) et les séquences `#{expr}` sont remplacées par leur évaluation

```
"Bonjour à tous"
"C'est la \"meilleure\" !"
"Il y a #{60*60*24} secondes par jour"
```

Types standards : booléens et `nil`

- Les valeurs booléennes prédéfinies sont `true` (instance singleton de la classe `TrueClass`) et `false` (instance singleton de la classe `FalseClass`)
- `nil` est un objet (instance singleton de la classe `NilClass`) qui représente le concept de vide, d'inexistence
- En Ruby, tout ce qui n'est pas `nil` ou la valeur constante `false` est vrai
- **Attention** : contrairement à d'autres langages, la constante entière 0 est évaluée à vrai et non pas à faux…

Intervalles

- Ruby offre des possibilités de représentation des intervalles de valeurs littérales à partir des séquences suivantes
  - `..` (2 points) : définit un intervalle où les bornes sont inclues
  - `...` (3 points) : définit un intervalle où la borne inférieure est inclue et la borne supérieure est exclue
- Exemples

```
1..10
'a'..'z'
0...tableau.length
```

Tableaux

— Les tableaux sont des collections d'éléments accessibles au moyen d'un *indice*, toujours entier en Ruby
— Sous Ruby, ils sont accessibles *via* la classe `Array`
— Ils sont définis littéralement au moyen des `[]`

```
1 tab = [0, 2, -5, 15]
```

et

```
1 tab[5] = 18
```

— Les tableaux Ruby peuvent contenir des éléments de types hétérogènes

```
1 tab = [5, 'hello', 5.23]
```

— L'accès à un élément donné se fait au moyen de son indice (les indices commencent à 0) et de l'opérateur `[]`

```
1 a = tab[0]
```

— L'opérateur `<<` ajoute un élément en fin de tableau

```
1 tab << a
```

— Il est possible d'extraire facilement un sous-tableau en utilisant la syntaxe `tab[début, nombre]`

```
1 tab = [5, 4, 8, 11, 7]
2 a   = tab[2, 2]           # retourne [8, 11]
```

— On peut utiliser les intervalles pour stipuler les indices de début et de fin lors de l'extraction d'un sous-tableau

```
1 a = tab[1..3]     # retourne [4, 8, 11]
2 a = tab[1...3]    # retourne [4, 8]
```

— Si un indice négatif est fourni lors d'un accès, l'élément est recherché depuis la fin (qui commence à -1)

```
1 a = tab[-2]       # retourne 11
```

Tables associatives

— Les Hash, ou tables associatives, sont des collections d'éléments accessibles au moyen d'une *clé*
— Sous Ruby, ils sont accessibles *via* la classe `Hash`
— Ils sont définis littéralement au moyen des `{}` et de `=>`

```
1 tab = {'phil' => 'dosch', 'mickey' => 'mouse'}
```

— **Mais** au moyen de `[]` si une seule valeur est définie

```
1 tab['donald'] = 'duck'
```

— En Ruby, la clé peut être de n'importe quel type (pas seulement une chaîne)
— L'accès à un élément donné se fait en fournissant une clé et **grâce à l'opérateur `[]`**

```
1 a = tab['phil']
```

Symboles

— Un symbole Ruby est un identificateur correspondant à une chaîne de caractères précédée d'un « : »

```
:toto
:une_chaine
:'bonjour'
:"variable"
```

- Les symboles sont des instances de la classe `Symbol`
- Ils peuvent généralement être utilisés partout où une chaîne de caractères est attendue
- **Particularité** : si un même symbole est présent plusieurs fois dans un fichier source, il n'occupe qu'une seule case mémoire !
- *A contrario*, s'il y a 5 chaînes `"Bonjour"` dans un source, chacune d'elles est un objet différent, dans une case mémoire différente…
- Un des usages qui en est fait : les symboles sont souvent utilisés à la place de chaînes lorsque ces chaînes font référence à une valeur convenue particulière, ce qui renforce la lisibilité
- Exemples

```
puts livre[:auteur]                  plutôt que :  puts livre['auteur']
trie(tableau, :nom, :asc)            plutôt que :  trie(tableau, 'nom', 'asc')
recherche(tableau, :prenom, 'Michel')  plutôt que :  recherche(tableau, 'prenom', 'Michel')
```

SECTION 3

Instructions de base

Affectations

- L'affectation est réalisée grâce à l'opérateur `=`
- L'affectation renvoie une valeur, il est donc possible de cascader les affectations

```
a = b = c = 0
```

- Comme dans d'autres langages, il existe des opérateurs d'affectation : `+=`, `-=`, `*=`…
- Il n'existe par contre **pas** d'opérateur d'incrémentation ou de décrémentation (les `++` et `--` du C par exemple)

Affectations parallèles

- Il est possible de réaliser des affectations en parallèle

```
a, b = 1, 3
```

- Cette fonctionnalité est particulièrement intéressante pour intervertir deux valeurs

```
a, b = b, a
```

- Quand une affectation comporte plusieurs valeurs à droite et **une seule** à gauche, l'affectation renvoie alors globalement toutes ces valeurs dans un tableau
- Règles en cas d'affectation non équilibrée
  - s'il y a plus d'opérandes à gauche qu'à droite, les dernières variables de gauche sont affectées à `nil`
  - s'il y a plus d'opérandes à droite qu'à gauche et qu'il y a au moins 2 opérandes à gauche, les dernières opérandes à droite sont ignorées

Affectations parallèles : cas particuliers

— S'il y a plusieurs opérandes à gauche et qu'il n'y a qu'un tableau à droite, ses éléments sont extraits

```
a, b, c = [2, 4, 6, 10]      # a = 2, b = 4, c = 6
```

— S'il y a plus d'opérandes à droite qu'à gauche, le dernier opérande de gauche se transforme en tableau s'il est préfixé par `*`

```
a, b = 2, 7, 8, 1            # a = 2, b = 7
a, *b = 2, 7, 8, 1           # a = 2, b = [7, 8, 1]
```

Opérateurs, méthodes

— Sur les nombres
  — opérateurs arithmétiques : `+`, `-`, `*`, `/`, `%` (modulo), `**` (puissance)
  — méthodes pouvant être appelées même sur des valeurs littérales

```
-3.abs
4.25.round
nb.infinite?
```

— Sur les chaînes de caractères
  — opérateurs : `+` (concaténation), `*` (copie)
  — méthodes très nombreuses (voir API)

Autres opérateurs et méthodes

— Opérateurs booléens : `&&` (et), `||` (ou), `!` (non)
— Opérateurs de comparaison : `==` (égalité), `<`, `>`, `<=`, `>=`
— `<=>` : opérateur de comparaison général, renvoie -1, 0 ou 1 (pour inférieur, égal, supérieur)
— `eql?` : même type et même valeur ? (ex : `1.eql?(1.0)` retourne `false`)
— `equal?` : même objet ?
— L'opérateur `defined?` renvoie `nil` si son argument n'est pas défini ; il en renvoie une description sinon
— Les méthodes `instance_of?` et `is_a?` (définies dans la classe `Object`) permettent de tester dynamiquement le type d'une variable (resp. de façon stricte ou en considérant l'arbre d'héritage)

Structures de contrôle : les conditionnelles

— Syntaxe

```
if condition then
    instructions
elsif condition then
    instructions
...
else
    instructions
end
```

— Le mot clé `then` n'est obligatoire que si la conditionelle tient sur une seule ligne et peut être remplacé par un `:` s'il est utilisé
— En Ruby, `if` est une expression, pas une instruction, et il renvoie donc une valeur
— Il est donc possible d'écrire

```
valeur = if ph < 7 then
           'acide'
         elsif ph == 7 then
           'neutre'
         else
           'basique'
         end
```

— Il existe aussi une version modificateur, i.e. suffixée, utilisable de cette façon

```
a = 23 if b > 5
```

— Enfin, il existe une version contraire du `if`, le `unless` dont la syntaxe est similaire à celle du `if`

```
unless a < 25
  b = 0
else
 b = 1
end
```

— Et il existe une version modificateur du `unless`

```
puts total unless total.zero?
```

Structures de contrôle : les branchements

— Comme dans d'autres langages, Ruby possède son expression de branchement qui est, cependant, nettement plus puissante

```
case option
when 1..500 then "Petit entier"
when 501...5000: "Grand entier"
when 5000..10000
  "Hors limite !"
when "erreur": "Erreur au départ..."
when /p\s+(\w+)/
  ...
else
  "Gros problème..."
end
```

— Le `case` supporte donc les intervalles, les expressions régulières, les chaînes…
— Il n'y a pas besoin de `break` pour quitter un choix
— Il est également possible, au moyen de cette expression, de faire des branchements relatifs à la classe d'origine de la variable examinée par le `case`…

Structures de contrôle : les boucles `while`

— Les boucles `while` itèrent sur leur contenu tant que la condition est vérifiée (éventuellement aucune itération)

```
while a > 5
  puts a
  a -= 1
end
```

— Il existe, comme pour `if` et `unless`, des versions modificateur

```ruby
a *= 2 while a < 1000
```

Structures de contrôle : les boucles `while` et `until`

— Il est également possible de faire des boucles s'exécutant toujours au moins une fois, toujours avec `while`

```ruby
begin
  puts 1
end while false
```

— De la même manière que `unless` est le contraire du `if`, `until` est le contraire du `while`

```ruby
until a <= 5
  puts a
  a -= 1
end
```

— Et `until` existe naturellement en version modificateur…

Structures de contrôle : les boucles `for ... in`

— Cette structure peut être utilisée pour itérer sur toute structure dotée d'une méthode each (voir après), comme les intervalles, les tableaux…

```ruby
for i in ['hello', 'bonjour', 'guten tag']
  puts i, " "
end

for i in 1..500
  puts i, ", "
end
```

— Cette structure est souvent remplacée par l'utilisation d'itérateurs (voir après)

CHAPITRE 3

Ruby plus en détail...

SECTION 1

Blocs et itérateurs

Introduction

— Les blocs sont des ensembles d'instructions formant un tout
— Ces blocs sont définis par les instructions présentes entre les délimiteurs `{ ... }` ou `do ... end`
— Conventionnellement, les blocs tenant sur une ligne le sont avec `{ ... }`, les autres le sont avec `do ... end`
— Les blocs peuvent être associés à des appels de méthodes

```
1 toto { puts 'Bonjour' }
```

— Les méthodes recevant des blocs peuvent aussi recevoir des paramètres

```
1 toto(2, 3) { puts 'Bonjour' }
```

L'instruction `yield`

— Une méthode peut invoquer le bloc qui lui a été associé, une ou plusieurs fois, au moyen de l'instruction `yield`
— Exemple

```
1 def deux_fois
2   puts 'Début'
3   yield
4   yield
5   puts 'Fin'
6 end
7
8 deux_fois { puts 'Hello' }
```

— Il est possible de passer des paramètres à l'instruction `yield`
— Ces paramètres seront alors passés au bloc associé à la fonction

```
1 def deux_fois
2   yield('appel', 1)
3   yield('lancement', 2)
4 end
```

```
5
6 deux_fois { |ch,i| puts "C'est mon #{ch} numéro #{i}" }
```

— Les blocs déclarent les variables qu'ils comportent en début de bloc entre une paire de `|`

Retour sur les blocs

— Les blocs sont des objets de la classe `Proc`
— Il est possible de sauvegarder et d'exécuter par la suite des blocs en manipulant des objets `Proc`

```
1 p1 = Proc.new { puts 'Hello' }
2 p1.call
3
4 p2 = Proc.new { |x, y| x + y }
5 p2.call(45, 87)
```

Les itérateurs

— Les *itérateurs* Ruby sont des méthodes qui invoquent les blocs qui y sont associés de façon répétée
— Les itérateurs remplacent avantageusement certaines des structures de contrôles vues en améliorant de plus la lisibilité du code
— Suivant les types de données considérés, il existe de nombreux itérateurs comme `each`, `find`...
— Deux syntaxes possibles : une seule ligne ou plusieurs

```
1 5.times { puts 'Hello' }
2
3 5.times do
4   puts 'Hello'
5 end
```

— L'itérateur `upto` (*concerne les entiers*)

```
1 1.upto(15) { |x| print x, ' ' }
```

produit : 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

— L'itérateur `step` (*concerne les nombres*)

```
1 1.step(15, 2) { |x| print x, ' ' }
```

produit : 1 3 5 7 9 11 13 15

— L'itérateur `each` (*concerne les tableaux, les hashs, les intervalles*)

```
1 [1, 2, 7, 10, 15].each { |val| print 2 * val, ' ' }
```

produit : 2 4 14 20 30

— Les itérateurs `collect!` et `join` (*concernent les tableaux*)

```
1 puts [1, 2, 7, 10, 15].collect! { |i| i * 2 }.join(', ')
```

produit : 2, 4, 14, 20, 30

— L'itérateur `inject` (*concerne les tableaux, les hashs, les intervalles*)

```
1 puts (1..5).inject(1) { |prod, n| prod * n }
```

produit : 120

SECTION 2

Fichiers

Les fichiers

— En Ruby, les fichiers sont manipulés par l'intermédiaire de la classe `File`
— Comme dans la majeure partie des langages, les fichiers doivent être ouverts avant d'être manipulés et refermés après usage
— Exemple

```
1 file = File.new('test.txt', 'r')
2 # ...
3 file.close
```

— Les modes d'ouverture (deuxième paramètre) sont les mêmes qu'en C : `'r'` pour une ouverture en lecture, `'w'` pour une ouverture en écriture

Lectures dans un fichier

— Il est possible de simplifier le squelette précédent en utilisant les blocs
— La méthode open, qui ouvre et ferme un fichier à la fois, peut être associée à un bloc

```
1 File.open('test.txt', 'r') do |file|
2   while line = file.gets
3     puts line
4   end
5 end
```

— On peut également imbriquer les itérateurs...

```
1 File.open('test.txt', 'r') do |file|
2   file.each_line { |line| puts line }
3 end
```

— Enfin, il est également possible de lire globalement un fichier et de le placer dans une variable grâce à `IO.read`

```
1 puts IO.read('test.txt')
```

— ... ou de récupérer toutes les lignes d'un fichier dans un tableau de chaînes de caractères

```
1 IO.readlines('test.txt').each do |line|
2   puts line
3 end
```

Écritures dans un fichier

— Comme pour les lecture, il est nécessaire que le fichier soit préalablement ouvert (avec un mode d'ouverture positionné à `'w'` ou `'w+'`)
— L'opérateur << peut être utilisé pour ajouter des données dans un fichier ouvert

```
1 file = File.new('test.txt', 'w')
2   15.times { |i| file << i }
3 file.close
```

SECTION 3

Ligne de commande et paramètres

— Un script Ruby peut récupérer les arguments qui lui sont fournis grâce au tableau `ARGV`
— Il est donc possible de parcourir ces arguments de cette façon

```
1 ARGV.each { |arg| puts arg }
```

— Ruby dispose de deux classes, `OptionParser` et `GetoptLong`, permettant de faciliter le traitement des options passées en ligne de commande (voir doc)
— Pour utiliser ces classes, il suffit d'inclure, respectivement

```
1 require 'optparse'
2 require 'getoptlong'
```

SECTION 4

Environnement

Certains outils gravitent autour de Ruby
— `gem` : outil proposant un système standard d'installation et de maintenance de packages Ruby. Attention, cet outil est en concurrence avec les outils proposés par les distributions Linux, il faut donc faire un choix…
— `irb` : outil permettant d'exécuter des instructions Ruby de façon interactive et de visualiser les résultats immédiatement
— `rake` : outil de construction similaire à `make` sur le plan des fonctionnalités, mais dédié à Ruby

SECTION 5

Modèle objet

Les classes

— Exemple de classe avec constructeur

```
1 class Livre
2   def initialize(titre, auteur, nbpages)
3     @titre = titre
4     @auteur = auteur
5     @nbpages = nbpages
6   end
7 end
```

— Une classe se définit entre `class` et `end`
— Les constructeurs de classe s'appellent toujours `initialize`
— En Ruby, les classes ne sont jamais fermées, il est toujours possible d'y ajouter quelque chose

Les classes : autre exemple

```ruby
class Toto
  def instance_methode
    ...
  end
  def Toto.class_methode
    ...
  end
  def self.class_methode2
    ...
  end
end

t = Toto.new
t.instance_methode
Toto.class_methode
Toto.class_methode2
```

Le typage « à la canard » (*Ducktyping*)

- Ruby juge un objet sur ce qu'il est capable de faire et pas sur un type défini (statiquement ou dynamiquement)
  - si un objet marche comme un canard et fait « coin-coin », alors Ruby le voit comme un canard
  - un objet se définit par ses attributs et ses méthodes, donc sur ce qu'il est capable de faire, pas sur un nom de type
- D'ailleurs, les classes Ruby ne correspondent pas vraiment à des types…

Les méthodes

- Exemple de méthode

```ruby
def prix_ttc(prix_ht, tva)
  return prix_ht * tva
end
```

- Une méthode se définie entre `def` et `end`
- Son nom doit suivre les conventions présentées précédemment
- Lors de la définition, les parenthèses peuvent être supprimées si la méthode n'accepte pas de paramètre
- Il est possible d'ajouter des méthodes à des classes, **comme à des objets**…
- Si une méthode est définie en dehors d'une classe, elle est ajoutée à l'objet courant (`self`)
- L'instruction `return` permet de renvoyer un (ou plusieurs) résultat(s) et provoque la sortie immédiate de la méthode
- Si aucune instruction `return` n'est présente, la valeur de retour d'une méthode est la valeur de sa dernière évaluation d'expression

```ruby
def prix_ttc(prix_ht, tva)
  prix_ht * tva
end
```

- En général, on oublie donc `return` s'il n'est pas nécessaire…
- La création d'un nouvel objet se fait au moyen de la méthode (de classe) `new`

```ruby
t = Time.new
```

- En Ruby, les paramètres ne sont pas typés

  - en pratique, il n'y a donc jamais d'erreur de typage
  - exception : si la valeur `nil` est fournie (ce qui est fait pour)
- La surcharge (*overloading*) n'existe pas en Ruby, mais elle peut être émulée par certaines techniques
- Il est possible d'attribuer des valeurs par défaut à certains paramètres

```
def yop(arg1, arg2 = 50)
  # ...
end
```

- Si le dernier paramètre d'une méthode est préfixé par `*`, il permet de stocker dans un tableau tous les paramètres supplémentaires fournis
  - définition

```
def yop(arg, *tab)
  # ...
end
```

  - exemples d'utilisation
    - l'appel `yop(1)` entraîne `arg = 1`, `tab = []`
    - l'appel `yop(1, 2)` entraîne `arg = 1`, `tab = [2]`
    - l'appel `yop(1, 2, 3)` entraîne `arg = 1`, `tab = [2, 3]`
- Cette syntaxe n'est d'ailleurs pas nouvelle : c'est celle relative aux affectations parallèles, qui s'appliquent dans ce contexte

Accessibilité des membres

- En Ruby, il est possible d'associer une accessibilité à chaque **méthode** de classe parmi
  - `public` : membre pouvant être utilisé par quiconque (visibilité par défaut des méthodes si rien n'est spécifié)
  - `protected` : membre pouvant être utilisé par la classe elle-même et ses classes dérivées
  - `private` : membre ne pouvant être utilisé que par l'**objet courant** (et non pas la classe courante comme c'est généralement le cas dans les autres langages orientés objet)
- Les attributs sont **toujours privés**
- Usage : les méthodes sont généralement publiques, sauf celles spécifiquement destinées à être utilisées en interne (c'est le cas par exemple du constructeur `initialize` qui est toujours privé)
- Utilisation : il existe deux manières de spécifier l'accessibilité des membres
  - en définissant des sections au moyen de `public`, `protected` et `private` dans lesquelles sont définis les membres
  - en énumérant en regard de chaque spécificateur d'accessibilité la liste des membres concernés

Accessibilité des membres : exemples de spécifications d'accessibilité

```
class Toto
    def m1 .. end

  protected

    def m2 .. end

    def m3 .. end

  public

```

```
12      def m4 .. end
13  end
```

```
1   class Toto
2     def m1 .. end
3
4     def m2 .. end
5
6     def m3 .. end
7
8     def m4 .. end
9
10    public :m1, :m4
11    protected :m2, :m3
12  end
```

Les *getters* et les *setters*

— Ces méthodes sont généralement définies lorsque l'encapsulation des données est pratiquée
— En Ruby, l'écriture d'un *getter* peut se faire de la manière suivante

```
1   class Livre
2     def titre
3       @titre
4     end
5   end
```

— Il existe un moyen de faire plus simple, en écrivant (3 attributs traités à la fois dans l'exemple)

```
1   class Livre
2     attr_reader :titre, :auteur, :resume
3   end
```

— De la même manière, l'écriture d'un *setter* en Ruby peut s'écrire

```
1   class Livre
2     def titre=(new_titre)
3       @titre = new_titre
4     end
5   end
```

— Là encore, il existe un moyen de faire plus simple, en écrivant (3 attributs traités à la fois dans l'exemple)

```
1   class Livre
2     attr_writer :titre, :auteur, :resume
3   end
```

permettant d'écrire (en dehors de la classe) :

```
1   l = Livre.new(...)
2   l.titre = 'Agile Web Development'
```

— La méthode `attr_accessor` permet de combiner l'effet des deux méthodes `attr_reader` et `attr_writer`
— Ainsi

```
1   attr_accessor :toto
```

est équivalent à :

```ruby
def toto
  @toto
end

def toto=(new_toto)
  @toto = new_toto
end
```

Héritage

- L'héritage permet de créer des classes (appelées *classes dérivées*) correspondant à la spécialisation d'autres classes (appelées *classes de base*)
- Syntaxe
  `class Derivee < Base`
- À l'intérieur du constructeur de la classe `Derivee`, un appel au constructeur de la classe de base peut être réalisé au moyen du mot-clé `super`
- `super` peut également être utilisé pour invoquer n'importe quelle méthode de la classe de base ayant même nom que la méthode de la classe dérivée où est invoqué `super`

Les classes intégrées

- Types de base : `Object`, `String`, `Array`, `Hash`, `Range`, `Regexp`, `Numeric`, `Float`, `Integer`, `Fixnum`, `Bignum`, `TrueClass`, `FalseClass`, `NilClass`
- Fichiers : `IO`, `File`, `FileTest`, `Dir`, `Errno`
- Kernel : `Kernel`, `Exception`, `Time`, `GC`, `Marshal`
- Système : `Process`, `Signal`, `Thread`, `ThreadGroup`
- `Math`
- Ainsi que de nombreuses classes standards couvrant les bases de données, les protocoles réseau, le Web/XML, la concurrence d'accès, la distribution, les mathématiques… (voir doc)

SECTION 6

Expressions régulières

Introduction

- Les expressions régulières ont pour objectif de filtrer des données (chaînes de caractères)
- Elles font partie des outils puissants et très utiles
- Ruby dispose nativement d'un support des expressions régulières très sophistiqué
- Les expressions régulières sont des objets de type `Regexp`
- Elles peuvent être créées en appelant explicitement un des constructeurs de cette classe ou en utilisant une des formes littérales suivantes
  - `/pattern/`
  - `%r{pattern}`

Utilisation

- Si une expression régulière est créée sous forme d'un objet de la classe `Regexp`, il faut invoquer la méthode `match` de cette classe qui renvoie un objet de type `MatchData` (voir documentation en ligne)
- Si une expression régulière est définie de façon littérale, les opérateurs suivants sont disponibles
  - `=~` : test de la concordance
  - `!~` : test de la non concordance
- Exemples

```
1 if name =~ /^\s*[a-z]/}
```

```
1 if title !~ /[aeiou]{2,99}/
```

```
1 er = Regexp.new('^\w+\s$')
2 m = er.match('Bonjour')
```

Les *patterns*

- Toutes les expressions régulières contiennent un *pattern*, utilisé pour filtrer une chaîne de caractères selon cette expression régulière
- À l'intérieur d'un *pattern*, les caractères suivants sont des méta-caractères et ont donc une signification particulière

  `. | ( ) [ ] { } + \ ^ $ * + /`

  (plus le `-` suivant sa place dans l'expression)
- Ces caractères ne peuvent pas être utilisés directement pour une recherche les concernant, puisque leur signification est différente de celle qu'ils ont habituellement
- Ils doivent donc être précédés d'un `\` si on souhaite les rechercher dans les chaînes traitées
- Les autres caractères sont des caractères « normaux » et se représentent eux-mêmes
- En plus des méta-caractères, d'autres séquences de caractères ont des significations particulières dans le contexte des expressions régulières
  - des séquences du type `\`*x*, où *x* est un caractère alphabétique
  - des séquences du type `[ :`*xxx*`:]`, où *xxx* est une chaîne de caractères
- Le « `.` » représente n'importe quel caractère, sauf le retour à la ligne

Les ancres

- Par défaut, une expression régulière recherche la **première** occurrence d'un *pattern* à l'intérieur d'une chaîne de caractères
- Exemple : `'Bonjour à tous' =~ /ou/`
- Les ancres permettent d'introduire des emplacements particuliers dans les expressions régulières
  - `^` : début d'une ligne
  - `$` : fin d'une ligne
  - `\A` : début d'une chaîne (qui contient plusieurs lignes)
  - `\Z` : fin d'une chaîne
  - `\b` : séparateur de mots (suite de lettres, de chiffres et de _)
  - `\B` : composante d'un mot

Les ancres : exemples

— `'this is\nthe time' =~ /^the/`
— `'this is\nthe time' =~ /is$/`
— `'this is\nthe time' =~ /\Athis/`
— `'this is\nthe time' =~ /\Athe/` (pas de concordance)
— `'this is\nthe time' =~ /\bis/`
— `'this is\nthe time' =~ /\Bis/`
— `'this is\nthe time' =~ /s\b/`
— `'this is\nthe time' =~ /s$/`
— `'this is\nthe time' =~ /e\b/`
— `'this is\nthe time' =~ /e\Z/`

Les classes de caractères

— Une classe de caractères est un ensemble de caractères entre `[]`
— Une classe de caractères représente un **choix**
— N'importe quel **unique** caractère concordera avec une classe de caractères si cette classe le contient
— Exemples
  — choix d'**une** voyelle : `[aeiouy]`
  — choix d'**un** symbole de ponctuation : `[,. :;!?]`
— Remarque : `.` est un méta-caractère et devrait donc être représenté par `\.`, cependant les méta-caractères perdent leur signification particulière à l'intérieur de `[]`
— À l'intérieur de `[]`, le caractère `-` permet de représenter des intervalles
  — `[A-Z]` : les majuscules
  — `[0-9]` : les chiffres
  — `[a-zA-Z]` : les caractères alphabétiques
— Si le premier caractère d'une classe de caractère est `^`, les caractères de la classe sont interdits au lieu d'être autorisés
  — `[^a-z]` : tout sauf les minuscules
  — `[^a-fA-F]` : tout sauf les lettres des nombres hexadécimaux

Les classes de caractères : abréviations

Séquence	Remplace	Signification
\d	`[0-9]`	Chiffres
\D	`[^0-9]`	Tout sauf un chiffre
\s	`[ \t\r\n\f]`	Caractère séparateur
\S	`[^ \t\r\n\f]`	Tout sauf un caractère séparateur
\w	`[A-Za-z0-9_]`	Caractères constituant les mots
\W	`[^A-Za-z0-9_]`	Tout sauf les caractères des mots

Les classes de caractères : abréviations POSIX

Séquence	Signification
`[ :alnum :]`	Caractères alphanumériques
`[ :alpha :]`	Lettres minuscules et majuscules
`[ :blank :]`	Espace et tabulation
`[ :cntrl :]`	Caractères de contrôle
`[ :digit :]`	Chiffres
`[ :graph :]`	Caractères affichables (*CA*) sauf l'espace
`[ :lower :]`	Minuscules
`[ :print :]`	*CA* y compris l'espace
`[ :punct :]`	*CA* sauf espace et alphanumériques
`[ :space :]`	Espace (similaire à `\s`)
`[ :upper :]`	Majuscules
`[ :xdigit :]`	Caractères des nombres hexadécimaux

Répétitions

— Il est possible d'associer des possibilités de répétition à des parties du *pattern* d'une expression régulière
— En considérant que `p` est une partie d'un *pattern*, une chaîne de caractères concordera avec ce pattern
  — `p*` : s'il y a 0 ou plus occurrences de `p`
  — `p+` : s'il y a 1 ou plus occurrences de `p`
  — `p?` : s'il y a 0 ou 1 occurrence de `p`
  — `p{m,n}` : s'il y a entre `m` et `n` occurrences de `p`
  — `p{m,}` : s'il y a au moins `m` occurrences de `p`
  — `p{m}` : s'il y a exactement `m` occurrences de `p`

  dans la chaîne candidate à l'endroit où est localisé `p`

Exemples

— Pour une heure au format anglo-saxon

```
/\d{1,2}:\d{2}[aApP][mM]/
```

— Pour une adresse mail

```
/\w+@\w+\.\w{2,4}/
```

— Pour un prénom et un nom

```
/[:upper:][:lower:]*[:blank:][:upper:][:lower:]*/
```

— Pour un numéro de sécurité sociale

```
/\d{13}\s\d{2}/
```

ou

```
/[:digit:]{13}[:blank:][:digit:]{2}/
```

Répétitions gloutonnes et non gloutonnes

— Par défaut, les répétitions sont *gloutonnes* (ou *avides*) : elles cherchent à avancer le plus loin possible dans la chaîne testée

— Il est également possible d'utiliser des versions non gloutonnes des répétitions, qui s'arrêtent dès que possible dans la chaîne testée
  — `p*?` : s'il y a 0 ou plus occurrences de `p`
  — `p+?` : s'il y a 1 ou plus occurrences de `p`
  — `p??` : s'il y a 0 ou 1 occurrence de `p`
  — `p{m,n}?` : s'il y a entre `m` et `n` occurrences de `p`
  — `p{m,}?` : s'il y a au moins `m` occurrences de `p`
  — `p{m}?` : s'il y a exactement `m` occurrences de `p`

Exemples

— Reconnaissance gloutonne

```
'aaaabbbb' =~ /a+/       # Reconnaît 'aaaa'
```

— Reconnaissances non gloutonnes

```
'aaaabbbb' =~ /a+?/      # Reconnaît 'a'
```

```
'aaaabbbb' =~ /a+?b/       # Reconnaît 'aaaab'
```

Alternatives

— Le caractère `|` permet de proposer des alternatives
— Il peut être utilisé plusieurs fois à la suite pour proposer de nombreuses alternatives différentes
— Exemple : `couleur =~ /rouge|vert|bleu/`
— Attention, `|` a une priorité très basse…
`/lunettes noires|nuits blanches/`
se lit :
`/lunettes noires|nuits blanches/`
et non pas :
`/lunettes noires|nuits blanches/`

— Les parenthèses peuvent être utilisées pour permettre d'obtenir le deuxième comportement

Regroupement

— Les parenthèses peuvent être utilisées pour regrouper des termes à l'intérieur d'expressions régulières
— Exemples

```
/(rouge|vert) \w+/
```

— Les parenthèses permettent aussi de définir des sous-expressions régulières au sein d'expressions régulières
— Lors d'une mise en correspondance réussie, les variables `$1`, `$2`, … contiendront respectivement la première, la seconde, … sous-expression régulière

```
if '12:50am' =~ /(\d\d):(\d\d)(..)/
  puts "Heures : #{$1}, Minutes : #{$2}"
```

— Il est aussi possible de définir des regroupements qui ne constituent pas des sous-expressions régulières
— Il faut alors utiliser la syntaxe `(?: … )` au lieu de la syntaxe `( … )`
— Il est également possible d'utiliser une sous-expression régulière **directement** au sein de l'expression qui l'intègre, en utilisant `\1, \2...` pour référencer les sous-expressions correspondantes
— Exemple typique : le mot qui se répète

```ruby
'aaa aaa' =~ /(\w+)\s\1/    # Reconnaît 'aaa aaa'
```

Substitution

- Si on ne souhaite pas vérifier la présence d'un *pattern* au sein d'une chaîne de caractères, mais effectuer une substitution basée sur les expressions régulières, on peut utiliser les méthodes `sub` et `gsub` de la classe `String`
- `sub` n'effectue qu'un seul remplacement et `gsub` effectue tous les remplacements possibles
- Ces méthodes renvoient la nouvelle chaîne de caractères, mais existent aussi dans une version permettant de changer directement la chaîne originale (`sub!` et `gsub!`)
- Exemple

```ruby
s = 'Bonjour'
s1 = s.sub(/[aeiouy]/, '*')    # 'B*njour'
s2 = s.gsub(/[aeiouy]/, '*')   # 'B*nj**r'
```

- Il est également possible de passer un bloc au lieu d'un deuxième paramètre, auquel cas la valeur du bloc est substituée dans la chaîne originale
- Exemple

```ruby
s = 'Bonjour'
s1 = s.gsub(/[aeiouy]/) { |voy| voy.upcase }
puts s1                                    # BOnjOUr
```

Substitution et sous-expressions régulières

- Si des sous-expressions régulières sont définies, il est possible d'y faire référence dans la chaîne de remplacement au moyen de `\1`, `\2`... qui ont même rôle que `$1`, `$2`
- Exemple

```ruby
s = 'Phil Dosch'.gsub(/(\w+)\s(\w+)/, '\2, \1')
puts s                                      # 'Dosch, Phil'
```

- Attention

```ruby
s = 'Phil Dosch'.gsub(/(\w+)\s(\w+)/, "\\2, \\1")
puts s                                      # 'Dosch, Phil'
```

- Il existe aussi d'autres séquences permettant d'exploiter la chaîne originale (liste non exhaustive)
  - `\`` : chaîne avant la concordance
  - `\'` : chaîne après la concordance

SECTION 7

Gestion des modules

Introduction

- Les *modules* Ruby sont une manière de regrouper des méthodes, des classes et des constantes
- La définition de module présente deux intérêts principaux en Ruby
  - les modules définissent un espace de nommage, ce qui permet d'éviter des conflits liés aux noms
  - les modules implémentent les *mixin*
- Les modules ne sont pas des classes, ils ne peuvent pas avoir d'instance

Modules et espaces de nommage

- Un module définit un espace de nommage
- Toutes les composantes (constantes, méthodes, classes) définies à l'intérieur d'un module voient leur nom complet préfixé du nom du module
- Syntaxe de définition

```
module MonModule
   ...
end
```

- Un script Ruby doit charger un module, au moyen du mot-clé `require`, pour pouvoir utiliser les composantes de ce module

Modules et espaces de nommage : exemple

```ruby
module Trigo
  PI = 3.1415926
  def sin(x)
    # ...
  end
  def cos(x)
    # ...
  end
end
```

```ruby
require 'Trigo'

y = Trigo.sin(Trigo::PI/2)
```

- L'opérateur de résolution de portée `::` permet de désigner des constantes définies dans des classes ou des modules
- `require` accepte des noms de fichier absolus ou relatifs et cherche aussi les fichiers dans les chemins définis dans la variable `$:`

Modules et *mixins*

- Les modules, par l'intermédiaire des *mixins*, sont également une réponse aux besoins d'héritage multiple
- Il est en effet possible d'inclure un module *à l'intérieur* d'une classe, au moyen du mot-clé `include`
- Toutes les composantes définies dans le module inclus sont alors définies dans la classe ayant inclus ce module
- Ce mécanisme est appelé *mixin*

Modules et *mixins* : exemple

```ruby
module Identite
  def coucou
    puts "Coucou #{self.class.name}"
  end
end

```

```ruby
class Pc
  include Identite
end

class Mac
  include Identite
end

p = Pc.new
m = Mac.New
p.coucou      # 'Coucou Pc'
m.coucou      # 'Coucou Mac'
```

Exemple concret : la classe `Object` (ancêtre de toutes les classes Ruby) inclut le module `Kernel`, contenant `puts`, `gets`…

Modules et *mixins* : remarques

- `require` a pour effet d'*importer* un fichier externe
  - son paramètre est une **chaîne de caractères**
  - l'opération sous-jacente est une manipulation de fichiers
- `include` crée une *référence* au module cité dans la classe comportant ce mot-clé
  - son paramètre est un **identificateur**, correspondant au nom du module
  - si une ou plusieurs classes incluent un (même) module, celui-ci n'est donc pas dupliqué (il n'est que référencé)
  - si la définition d'un module est modifiée dynamiquement, ces modifications influent donc immédiatement sur toutes les classes qui le référencent

SECTION 8

Exceptions

Introduction

- Ruby intègre un mécanisme d'exceptions pour la gestion des erreurs
- Comme dans les autres langages, ce mécanisme présente certains avantages
  - permet une gestion centralisée des erreurs
  - permet une gestion différée des erreurs, jusqu'à ce que le programme soit « capable » de les traiter
- Les informations relatives à une erreur sont stockées dans un objet, instance de la classe `Exception` ou d'une de ses classes dérivées, propagé dans la pile des appels jusqu'à ce qu'une instruction déclare être capable de traiter ce type d'erreur

Gestion des exceptions

- Les exceptions sont « levées » lorsque des instructions n'ont pas pu être exécutées normalement
- Ce genre de problème survient typiquement avec des instructions concernant les entrées-sorties (mémoire, fichiers, réseau…)
- Lorsqu'une instruction est susceptible de lever une exception, on peut ajouter une section permettant de traiter les éventuelles exceptions levées

Syntaxe générale

— Les instructions pouvant lever une exception doivent être placées dans un bloc délimité par `begin` ... `end`
— Les différentes exceptions peuvent être interceptées avec des clauses `rescue`
— La syntaxe générale est

```
begin
    instructions pouvant lever une exception
rescue TypeException1
    instructions traitant ce type d'exception
rescue TypeException2
    instructions traitant ce type d'exception
...
end
```

— Il est possible de récupérer l'objet correspondant à l'exception dans une clause `rescue` (objet `e` dans l'exemple)

```
rescue NoMemoryError => e
```

— Il est également possible dans ce genre de sections d'exécuter des instructions particulières si aucune exception n'a été levée, grâce à la clause `else`
— Enfin, il est possible de spécifier des instructions qui doivent toujours être exécutées, qu'une exception ait été levée ou non, grâce à la clause `ensure`
— À l'intérieur d'une clause `rescue`, il est possible d'utiliser
  — `retry` : répète le bloc entier (de `begin` à `end`)
  — `raise` : relance l'exception (ce qui permet de la transmettre en dehors du bloc)
— Un bloc peut donc, au final, ressembler à

```ruby
begin
  # ...
rescue xxx
  # ...
rescue yyy
  # ...
else
  # ...
ensure
  # ...
end
```

— Exemple

```ruby
begin
  fd = File.open(nomfic, FILE::RDONLY) do |fic|
    # ...
  end

rescue Errno::ENOENT
  STDERR.puts 'Fichier inexistant'
  exit!(1)
rescue Exception => e       # Autres exceptions possibles
  STDERR.puts(e)
  exit!(2)
ensure
  fd.close
end
```

Création d'exception

- Les utilisateurs peuvent naturellement créer leurs propres exceptions au moyen de la clause `raise`
- Plusieurs syntaxes sont possibles
  - `raise` : utilisé seul, `raise` relance l'exception en cours ou crée une exception de type `RuntimeError` s'il n'est pas utilisé dans le cadre du traitement d'une exception
  - `raise "Problème..."` : crée une exception de type `RuntimeError` et lui associe le message passé en paramètre
  - `raise InterfaceException, "Clavier..."` : crée une exception du type passé en paramètre, associé au message correspondant au deuxième paramètre

CHAPITRE 4

Aspects systèmes

SECTION 1

Les fichiers

Informations sur les fichiers

- Les méthodes `stat` et `lstat` de la classe `File` renvoient un objet de la classe `File::Stat`, regroupant des informations sur un fichier donné
  - `stat` : suit les liens symboliques et donne des informations sur le fichier cible
  - `lstat` : ne suit par les liens symboliques et donne des informations sur le lien si c'en est un qui est fourni en paramètre
- Les objets de la classe `File::Stat` proposent de nombreuses méthodes pour tester les dates, les propriétaires, les types, les droits…

Les répertoires

- La classe `Dir` regroupe les différents appels système liés aux répertoires
- `Dir.new` et `Dir.open` fonctionnent globalement comme leur homologue de la classe `File`
- Là encore, de nombreuses méthodes disponibles (voir doc)
- Exemple : parcours de répertoire (ls)

```
1  Dir.open('/etc') do |rep|
2    fichiers = rep.find_all {|nom| nom =~ /[^.]\w+\.txt/}
3
4    fichiers.each do |nom|
5      puts File.stat("/etc/#{nom}").size
6    end
7  end
```

- Ou plus simplement avec `Dir.glob`, qui combine l'ouverture et la recherche…

SECTION 2

Les verrous

Généralités

- Les verrous permettent de résoudre certains des problèmes liés à la concurrence d'accès aux fichiers
- Sous Unix, à partir d'un shell, c'est la commande `lockfile` qui permet d'implanter ce mécanisme
- Sous Ruby, il existe plusieurs possibilités implantant ce mécanisme, dont l'utilisation de la méthode `flock` de la classe `File`
- Les différents paramètres possibles de cette méthode sont
  - `LCK_EX` : verrou exclusif, empêchant la définition de tout verrou
  - `LCK_SH` : verrou partagé, permettant la définition d'autres verrous du même type
  - `LCK_UN` : libération d'un verrou
- Par défaut, les verrous sont bloquants : l'exécution s'arrête jusqu'à ce que le verrou soit accessible
- Il est également possible de faire des appels non bloquants à `flock`, en combinant le paramètre choisi avec `LCK_NB` au moyen d'un ou logique (`|`)
- Attention : ces verrous sont des verrous *consultatifs*
- Cela suppose que tous les processus en cause « jouent le jeu » (*i.e.* utilisent ce mécanisme Ruby), le système (l'OS) n'empêchera pas de modifier (par un autre biais) un fichier pourtant verrouillé…

Les verrous : attention...

- Le descripteur de fichier utilisé pour positionner un verrou doit être le même que celui utilisé pour supprimer ce même verrou
- Cette solution ne marche pas…

```
1 lf = File.new(lockfile)
2 lf.flock(File::LOCK_EX)
3 lf.close
4
5 # ...
6
7 lf = File.new(lockfile)
8 lf.flock(File::LOCK_UN)
9 lf.close
```

- Mais cette solution marche…

```
1 lf = File.new(lockfile)
2 lf.flock(File::LOCK_EX)
3
4 # ...
5
6 lf.flock(File::LOCK_UN)
7 lf.close
```

- En revanche, un verrou donné peut naturellement être utilisé (demandé / positionné / supprimé) par des processus différents… (heureusement !)

SECTION 3

Les signaux

- Il est possible d'émettre des signaux en Ruby selon les mêmes modalités que la commande `kill` sous Unix
- Sous Ruby, la classe `Signal` propose 2 méthodes
  - `list` : accès à la liste des signaux supportés
    `Signal.list.invert.sort.each {|p| puts p(' : ')}`
  - `trap` : permet de changer le comportement d'un programme lors de la réception d'un signal

```
1 Signal.trap(3) { puts "J'ai reçu SIGQUIT" }
```

- Pour envoyer un signal, il faut utiliser la méthode `kill` de la classe `Process`

SECTION 4

Les processus

Création

- Pour créer un nouveau processus, il faut utiliser la méthode `fork`
- Il y a 2 types d'appels possibles
  - sans bloc associé, `fork` crée un processus fils, renvoie son PID au processus père et renvoie `nil` au processus fils
  - avec un bloc, l'appel à `fork` renvoie les mêmes valeurs et fait exécuter les instructions du bloc dans le processus fils, tandis que le père continue son exécution après le bloc
- La création de nouveaux processus est particulièrement intéressante lors de l'exécution d'un script sur les machines multi-processeurs…

Exemples

```
1  if (fork == nil)
2    puts("Je suis le fils de PID #{Process.pid}")
3  else
4    puts("Je suis le père de PID #{Process.pid}")
5  end
6
7  # ----------------------------------------------
8
9  fork do
10   puts("Je suis le fils de PID #{Process.pid}")
11 end
12 puts("Je suis le père de PID #{Process.pid}")
```

Terminaison de processus

- Le fils se termine généralement avec un appel à `exit`, en passant un code de sortie
- Le père doit attendre la fin de ses fils grâce à la méthode `wait` de la classe `Process`, qui renvoie le PID du fils ou -1 en cas d'erreur

— Le code renvoyé par le fils est récupérable grâce à la méthode `exitstatus`

```ruby
# -*- coding: utf-8 -*-
# ...
pid = Process.wait
if $?.exitstatus == 0
  puts ("Mon fils #{pid} s'est bien terminé")
else
  puts ("Mon fils #{pid} s'est mal terminé")
end
```

Exécution de processus externes

— Rappel : il est possible de lancer des processus externes au moyen d'une paire de `‘`
— Après exécution d'un `fork`, il est également possible de faire appel à la méthode `exec` pour remplacer le processus fils par un autre (un processus externe)
— Cela est notamment pratique lorsqu'on souhaite utiliser des outils comme IPC (*Inter Processus Communication*) : pipes, mémoire partagée, sémaphores…
— Exemple

```ruby
exec("gcc", "-v")
```

Exemple : ce qu'il ne faut pas faire...

```ruby
#!/usr/bin/ruby -w

1.upto(4) do |i|
  if fork != nil
    puts "pid = #{Process.pid}, ppid = #{Process.ppid}, i = #{i}"
  end
end
```

Questions : combien de processus sont lancés au final (nombre de lignes affichées) ? comment corriger le problème ?

CHAPITRE 5

Interfaces graphiques

SECTION 1

Introduction

Introduction

— Ruby peut s'interfacer avec des toolkits graphiques afin de pouvoir créer des interfaces graphiques
— De nombreux toolkits sont disponibles, dont les 2 ténors sous Linux, GTK et Qt
— Autre toolkit intéressant : shoes (`http://shoesrb.com`)
— Il est alors facile et rapide de créer des *front-ends*, facilitant l'exécution de commandes sophistiquées

Ruby et Qt

— Une version de Qt est utilisable par Ruby
— La documentation est disponible sur `http://techbase.kde.org/Development/Languages/Ruby`
— Un bon tutorial est également disponible à `http://www.darshancomputing.com/qt4-qtruby-tutorial/`

SECTION 2

Interfaçage Ruby / Qt

1 Application de base

Hello World !

```
1 #!/usr/bin/ruby -w
2
3 require 'Qt4'
4
5 app = Qt::Application.new(ARGV)
6
7 hello = Qt::PushButton.new("Bonjour !")
8 hello.resize(100, 30)
```

```
hello.show

app.exec
```

Remarques

— Nécessite la création d'une application Qt

```
a = Qt::Application.new(ARGV)
```

— Construction du programme en lui-même, basé sur la programmation événementielle
— Lancement de la boucle de gestion d'événements

```
app.exec
```

— Communication entre composantes basée sur les signaux et slots

❷ Communication entre composants : signaux et slots

```
#!/usr/bin/ruby -w

require 'Qt4'

app = Qt::Application.new(ARGV)

hello = Qt::PushButton.new("Ciao !")
hello.resize(100, 30)
Qt::Object.connect(hello, SIGNAL('clicked()'),
                   app, SLOT('quit()'))

hello.show()

app.exec()
```

— Les *signaux* sont émis par des composantes Qt
— Les *slots*, disponibles eux aussi sur les composantes Qt, permettent de traiter les signaux émis
— Il est possible de raccorder un signal à un slot au moyen de la méthode `Qt::Object.connect` : les profils des méthodes doivent concorder
— La liste des signaux et slots disponibles figure sur la documentation des classes

❸ Intégration de plusieurs composants

Les `widgets`

```
#!/usr/bin/ruby -w

require 'Qt4'

app = Qt::Application.new(ARGV)

window = Qt::Widget.new
window.resize(200, 150)

```

```
hello = Qt::PushButton.new(”Ciao !”, window)
hello.setGeometry(50, 50, 100, 30)
Qt::Object.connect(hello, SIGNAL(’clicked()’),
                   app, SLOT(’quit()’))

window.show()

app.exec()
```

— Les *widgets* sont à la base de toutes les composantes graphiques
— Utilisés sous leur forme de base, il servent de containers
— Les composantes ajoutées dans un *widget* doivent incorporer une référence vers ce *widget* lors de leur création

D’autres composantes...

```
...
window = Qt::Widget.new
window.resize(200, 150)

lcd = Qt::LCDNumber.new(2, window)
lcd.move(10, 50)

slider = Qt::Slider.new(Qt::Horizontal, window)
slider.setRange(0,99)
slider.setValue(0)
slider.move(10, 80)

Qt::Object.connect(slider, SIGNAL(’valueChanged(int)’),
                   lcd, SLOT(’display(int)’))
...
```

Et avec une syntaxe plus... Ruby ?

```
Qt::Application.new(ARGV) do
    Qt::Widget.new do
        self.window_title = ’Ruby way!’
        resize(200, 100)
        button = Qt::PushButton.new(’Quit’) do
            connect(SIGNAL :clicked) { Qt::Application.instance.quit }
        end

        label = Qt::Label.new(’<big>Ruby way!</big>’)

        self.layout = Qt::VBoxLayout.new do
            add_widget(label, 0, Qt::AlignCenter)
            add_widget(button, 0, Qt::AlignRight)
        end
        show
    end
    exec
```

```
18 end
```

SECTION 3

Conclusion

- Pas forcément simple d'accès, nécessite idéalement une connaissance des classes Qt
- Pas forcément compliqué, une IHM se construit en quelques lignes de script
- Un outil en ligne de commande, `rbqtapi`, documente le toolkit
- Un accès à l'API de Qt est naturellement nécessaire `http://qt-project.org/doc/qt-4.7`