Licence 3 d'Informatique — Langages de *script* — TP 6

Début : jeudi 9 novembre 2023

Durant cette seconde série d'exercices de travaux pratiques consacrée au langage Ruby, vous allez d'abord faire connaissance avec les *blocs*, qu'on peut définir sommairement comme des *fragments* de progammes. Vous aurez besoin des fichiers :

for-lc-6.rb animals-plus.rb

respectivement utilisés aux §§ 1 & 2 et que vous pourrez trouver sur le serveur moodle, dans le répertoire « Travaux pratiques Ruby 2023 > for-lc-6 », qui est une partie du cours :

Programmation fonctionnelle, scripts et XML

la plupart des définitions de ces deux fichiers sources ayant déjà été discutées en cours. Ensuite, vous développerez une hiérarchie de classes en Ruby. Bien sûr, vous pourriez en faire autant dans des langages à objets plus « classiques » tels que Java ou C#, mais n'oubliez pas que Ruby se prête particulièrement à du développement *rapide*. Aussi, n'hésitez pas à utiliser des constructions telles que `attr_reader`, plutôt que de développer des méthodes de lecture *ad hoc*. Ensuite, nous verrons comment la notion de *module* de Ruby nous permet de simuler *grosso modo* un héritage multiple [1].

1 Blocs

1.1 Introduction aux blocs

Considérez les définitions et exemples d'évaluations donnés dans le fichier for-lc-6.rb et reproduits dans la figure 1. Comme vous pouvez le voir, quelques méthodes applicables aux *tableaux* — de classe `Array` — admettent des blocs à un ou deux arguments et remarquez comment ces blocs sont notés lors de l'appel de l'une de ces méthodes [2]. Il est important de s'apercevoir que :

— de tels blocs ne sont *pas* des objets de première classe, dont on peut retenir la valeur dans une variable ;

— appeler les méthodes citées dans la figure 1 sans fournir de bloc provoque des erreurs.

1. Rappelons qu'en Ruby, l'héritage est *simple*, comme en Java ou en C#. Dans ces deux derniers langages, il est possible d'approcher un héritage multiple, par exemple, par l'emploi simultané des constructions `extends` et `implements` en Java. Des exemples de langages à objets offrant directement une construction d'héritage multiple sont C++ et Python.

2. Il est possible de remplacer les mots réservés `do` et `end` par des accolades ouvrante et fermante, mais les priorités sont différentes.

```
$a1 = [21,10,2023,2024]

$a1.each do |x| p x end                                  # Écriture des éléments successifs du
                                                         # tableau $a1.

$a1.each_index do |index| p index end                    # Écriture des positions positives ou nulles
                                                         # du tableau $a1.

$a1.each_with_index do |x,index| p [x,index] end         # Écriture de chaque élément, suivi de sa
                                                         # position.

p($a1.select do |x| x.odd? end)          ⟹ [21,2023]
p([].inject(0) do |x,y| x + y end)       ⟹ 0
p($a1.inject(0) do |x,y| x + y end)      ⟹ 4074
p((1..10).inject(1) do |x,y| x * y end)  ⟹ 3628800

def twice(x)
  yield(yield(x))
end

p(twice(2023) do |y| y + 1 end) ⟹ 2025

def commutativefor(x,y)
  yield(x,y) == yield(y,x)
end

p(commutativefor(2023,2022) do |x,y| x + y end) ⟹ true
p(commutativefor(2023,2022) do |x,y| x - y end) ⟹ false
```

Figure 1 : Définitions et exemples du fichier source for-lc-6.rb.

Bien constater que la méthode `inject` s'emploie avec des tableaux, mais aussi avec des *intervalles*, de classe `Range` [3]. Suivent deux fonctions globales :

— `twice`, qui admet un argument quelconque, s'emploie avec un bloc à un argument et compose deux fois l'application du bloc à partir de l'argument fourni ;

— `commutativefor`, qui admet deux arguments, s'emploie avec un bloc à deux arguments et retourne `true` si l'application du bloc donne le même résultat quelque soit l'ordre d'apparition suivi pour les deux arguments, `false` sinon.

Faites quelques essais pour vous convaincre que l'instruction `yield`, qui déclenche l'évaluation d'un bloc, doit présenter autant de valeurs — ni plus, ni moins — qu'il existe de variables locales dans le bloc fourni.

1.2 À vous de jouer

⟹ Comment pouvez-vous utiliser la méthode `inject` pour définir une méthode `fact`, liée à la classe des entiers relatifs [4] — rappelons qu'en Ruby, il est toujours possible de *ré-ouvrir* une

3. De tels intervalles sont toujours *fermés à gauche*, ils sont soit *fermés à droite* — par exemple, « `1..10` » — soit *ouverts à droite* — par exemple, « `1...10` ». Utiliser la méthode `to_a`, qui retourne le tableau composé de tous les éléments de l'intervalle, pour bien observer la différence.

4. Cette classe est `Integer` sur la version de Ruby installée dans les salles de travaux pratiques, mais si vous travaillez sur votre propre matériel et employez une version ancienne, il se peut que cette classe soit `Fixnum`.

classe (même prédéfinie) pour lui ajouter de nouvelles définitions — et permettant de calculer la factorielle d'un nombre entier naturel ? Votre méthode `fact` devra retourner `nil` si elle est appliquée à un nombre strictement négatif.

$\Longrightarrow$ Ajouter à la classe `Array` des tableaux une méthode `halfapply` qui, appelée à partir d'un tableau $\mathtt{a}_0$, applique un bloc aux éléments de rang pair de $\mathtt{a}_0$ et récupère dans un tableau les résultats dans l'ordre d'apparition. Exemple :

```
$a1.halfapply do |x| x + 1 end ⟹ [11,2024]
```

en sachant que :

— vous pouvez utiliser les méthodes `each_index` ou `each_with_index` de la classe `Array`, vues précédemment — les indices commençant à zéro, comme en Java — ;

— vous pouvez créer un tableau vide, puis y ajouter un élément à sa droite par la méthode `push` de la classe `Array` :

```
a0 = []          a0.push('L3')
```

et accéder à un élément dont on connaît le rang en appliquant la méthode `at` de la classe `Array`, ou en donnant ce rang entre crochets droits [5] :

```
a0.at(0) ⟹ "L3"          a0[0] ⟹ "L3"
```

— un autre procédé consiste à créer d'abord un tableau en spécifiant sa dimension, puis à le « remplir » progressivement ; dans ce cas, la méthode `at` n'est pas permise en partie gauche d'une affectation, mais la notation entre crochets droits y est utilisable [6] :

```
a1 = Array.new(10)          a1[2] = 2025
```

— enfin, les méthodes `even?` et `odd?` de la classe des entiers relatifs [7] sont à votre disposition pour tester si un nombre entier relatif est pair ou impair.

$\Longrightarrow$ Donner maintenant une fonction globale `powerfultwice`, dont l'argument *optionnel* est par défaut lié à la valeur `2023`. Cette fonction `powerfultwice` se comporte comme la fonction `twice` de la figure 1 si un bloc est fourni — utiliser la construction `block_given?` pour ce test — ; sinon cette fonction équivaut à la fonction identité. Exemples :

```
powerfultwice                          ⟹ 2023
powerfultwice(2022)                    ⟹ 2022
powerfultwice do |x| x + 1 end         ⟹ 2025
powerfultwice(2024) do |x| x + 1 end   ⟹ 2026
```

Évaluez l'expression `2022.class` — ou quelque chose d'approchant — pour savoir à quoi vous en tenir.

5. Une convention qui est inutile pour l'exercice, mais qui peut être malgré tout intéressante à connaître : si vous utilisez des nombres strictement négatifs comme indices, les positions sont comptées à reculons à partir de l'élément le plus à droite dans le tableau.

6. L'emploi d'indices strictement négatifs selon la convention de la note précédente est également possible.

7. Voir à ce sujet la note 4 en bas de la page 2.

```ruby
class Animal
  #
  attr_reader :name
  #
  def initialize(name_0)
    @name = name_0
    @@animalnb += 1
  end
  #
  def eat
    puts 'I\'m eating.'
  end
  #
  @@animalnb = 0
  #
  def self.nb
    @@animalnb
  end
  #
  CrDate = '2016/12/13'
  #
end

class Dog < Animal
  #
  attr_reader :ownername
  #
  def initialize(name_0,ownername_0)
    super(name_0)
    @ownername = ownername_0
  end
  #
  def bark
    puts 'I\'m barking.'
  end
  #
  class << self
    undef_method :nb
  end
  #
end

class Monkey < Animal
  #
  def climb
    puts 'I\'m climbing.'
  end
  #
  class << self
    undef_method :nb
  end
  #
end

$a = Animal.new('Baloo')
$a.eat
p $a.name

$d = Dog.new('Rintintin','Bonanza')
$d.eat
$d.bark
p $d.name
p $d.ownername

$m = Monkey.new('Snow Flake')
$m.eat
$m.climb
p $m.name

p Animal.nb

$kingkong = Monkey.new('King-Kong')

def $kingkong.skullisland
  puts 'I\'m at home.'
end

class << $kingkong
  def empirestatebuilding
    puts 'That\'s the end, my friend.'
  end
end

class Dog
  #
  def bite
    puts 'Biting U!'
  end
  #
  def do_with_names_and(x)
    yield(@name,@ownername,x)
  end
  #
end

$d.bite

puts Animal::CrDate
```

Figure 2 : Fichier animals-plus.rb (début).

```
class Animal
  #
  def gimme
    puts 'Gimme, gimme ' + yield
  end
  #
  def do_with_name
    puts "I'm #{@name}"
    yield(@name) if block_given?
  end
  #
end
```

Figure 3 : Fichier animals-plus.rb (fin).

1.3 Expressions régulières et blocs

$\Longrightarrow$ Donner une méthode globale `filtermap`, telle que l'évaluation de l'expression :

`filtermap(`a_0`,`*regexp*`)`

applique un bloc à tous les éléments du tableau a_0 qui sont filtrés par l'expression régulière *regexp* et retourne le tableau des résultats, dans l'ordre d'apparition. Exemple :

```
filtermap(['2023-11-09','12',11,2023,'11-09-2023'],/2023/) do |x| [x] end ⟹
        [["2023-11-09"],["11-09-2023"]]
```

On pourra utiliser un algorithme assez proche de celui qui est suggéré au § 1.2 pour la fonction `halfapply`, ou — au prix de deux parcours de tableaux — les méthodes `select` et `map` de la classe `Array`.

2 Retour aux classes d'animaux

2.1 Quelques rappels

Les figures 2 & 3 reproduisent l'exemple du fichier animals-plus.rb distribué : la base — déjà vue en cours — en est une hiérarchie de classes Ruby représentant des animaux. Rappelons encore une fois qu'en Ruby, il est toujours possible de ré-ouvrir une classe pour lui ajouter de nouvelles définitions et remarquons que les méthodes `gimme` et `do_with_name` ont été ajoutées à la classe `Animal`, tandis que la méthode `do_with_name_and` a été ajoutée à la classe `Dog`. Essayer les évaluations suivantes :

```
$kingkong.gimme do 'A man after midnight' end ⟹ ??
$kingkong.do_with_name                        ⟹ ??
$kingkong.do_with_name do |name| puts "#{name} and Jane" end ⟹ ??
```

2.2 À vous

$\Longrightarrow$ En considérant la méthode `do_with_name_and` de la classe `Dog`, quel bloc fournir à cette méthode pour réaliser la session suivante ?

```
$d = Dog.new('Rintintin','Bonanza')
$d.do_with_names_and('John Wayne') do ... end ⟹
        "Rintintin, Bonanza, played by John Wayne"
```

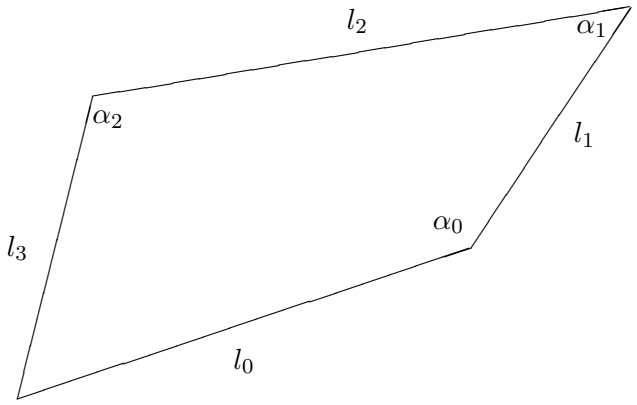

Figure 4 : Informations associées à un quadrilatère.

$\Longrightarrow$ Considérer à nouveau la classe `Animal` et y ajouter une méthode `with` prenant un argument un objet `a` de classe `Animal` et utilisant un bloc sans argument retournant une chaîne de caractères. De deux choses l'une — nous notons $\texttt{self}_0$ l'objet auquel est appliquée la méthode `with` — :

— si $\texttt{self}_0$ et `a` sont de même classe — en tenant compte des sous-classes qui existent pour la classe `Animal` —, cette méthode affiche :

$$\texttt{self}_0\texttt{'s name with a's name for ...}$$

— où « ... » est le résultat de l'application du bloc, quant aux parties composées en caractères inclinés, elles doivent bien sûr être remplacées par les noms des deux animaux — après quoi cette méthode retourne la valeur `true` ;

— sinon, cette méthode n'affiche rien et retourne la valeur `false`.

Exemples :

```
$kingkong.with($m) do 'getting Jane' end ⟹
        King-Kong with Snow Flake for getting Jane   # Affichage.
        true                                         # Résultat.
$a.with($d) do 'anything' end ⟹ false
```

3 Vos premières classes

3.1 Hiérarchie d'objets géométriques

Vous allez à présent développer une première version d'une hiérarchie d'objets géométriques. La classe la plus englobante est `Quadrilateral`, elle représente un *quadrilatère* quelconque au moyen des informations données dans la figure 4, les longueurs l_0, l_1, l_2, l_3 — de classe `Float`

	Périmètre	Surface
Quadrilatère	$l_0 + l_1 + l_2 + l_3$	
Parallélogramme	$2(l_0 + l_1)$	$l_0\, l_1\, \cos(\alpha_0 - \frac{\pi}{2})$
Rectangle	⟵ parallélogramme.	$l_0\, l_1$
Carré	$4l_0$	⟵ rectangle.

Table 1 : Calculs de périmètres et de surfaces, version préliminaire.

— étant toutes exprimées dans la même unité, et les angles $\alpha_0, \alpha_1, \alpha_2$ — également de classe `Float` — donnés en radians [8]. La grandeur α_0 (resp. α_1, α_2) est la mesure de l'angle entre les côtés de longueurs l_0 et l_1 (resp. l_1 et l_2, l_2 et l_3). Peu importe le côté par lequel on commence, l'important étant de « faire le tour » du quadrilatère dans le sens inverse des aiguilles d'une montre. Rappelons les cas particuliers de quadrilatères :

— un **parallélogramme** est un quadrilatère dont les côtés sont parallèles deux à deux ; par rapport au schéma précédent, $l_2 = l_0$, $l_3 = l_1$, $\alpha_1 = \pi - \alpha_0$, $\alpha_2 = \alpha_0$;

— un **rectangle** est un parallélogramme dont les angles sont droits ;

— un **carré** est un rectangle dont les quatre côtés sont de même longueur.

Nous considérons que les calculs de périmètre et de surface doivent s'effectuer comme nous l'avons indiqué dans la table 1 — le calcul de la surface n'est pas envisagé pour un quadrilatère quelconque —, le signe « ⟵ » signifiant « utiliser la méthode pour un … » Pour obtenir la valeur du nombre π en Ruby, utiliser la constante `Math::PI`, la classe `Math` fournissant en outre la fonction trigonométrique `Math.cos`. Attention aux points suivants :

— l'emploi de l'opérateur « `::` » de portée dans « `Math::PI` », car il ne s'agit pas d'une méthode, mais de la définition d'un nombre constant [9] ;

— il n'est pas permis en Ruby de terminer la notation d'un nombre par le point décimal, aussi, écrivez bien au moins un zéro derrière le point s'il n'y a pas de partie décimale ; par exemple, écrivez « `0` » pour le nombre zéro entier et « `0.0` » pour le nombre zéro flottant ; évaluez les expressions :

```
0.class          0.0.class
```

pour vous assurer que cette convention est non seulement correcte, mais ne produit pas d'ambiguïtés.

Dans cette première version, nous conviendrons que la méthode de calcul d'une surface pour un quadrilatère quelconque va retourner `nil`.

8. Rappelons que l'angle plat vaut π radians.

9. La définition et l'utilisation de `Math::PI` sont comparables à ce qui est fait pour la chaîne de caractères `CrDate`, définie dans la classe `Animal` du fichier animals-plus.rb.

	Périmètre	Surface
Quadrilatère	$l_0 + l_1 + l_2 + l_3$	
Parallélogramme	$2(l_0 + l_1)$	$l_0\, l_1\, \cos(\alpha_0 - \frac{\pi}{2})$
Rectangle	⟵ parallélogramme.	$l_0\, l_1$
Losange	$4l_0$	⟵ parallélogramme.
Carré	⟵ losange.	⟵ rectangle.

Table 2 : Calculs de périmètres et de surfaces, version enrichie.

$\Longrightarrow$ Définissez en Ruby toute cette hiérarchie de classes — `Quadrilateral`, `Parallelogram`, `Rectangle`, `Square` —, accompagnée des calculs de périmètres et de surfaces, avec les contraintes suivantes :

(C1) les constructeurs doivent être appelés avec le moins d'informations possible ; par exemple, *trois* informations suffisent pour un parallélogramme, *deux* pour un rectangle ;

(C2) une fois qu'un quadrilatère est créé, il est impossible de modifier ses informations ; même chose pour un parallélogramme, un rectangle, ou un carré ;

(C3) nous devons pouvoir *lire* les informations qui caractérisent un quadrilatère, mais il ne doit pas exister plus de méthodes d'accès que d'informations pertinentes : par exemple, il doit exister *sept* méthodes d'accès pour un quadrilatère quelconque, correspondant aux sept grandeurs qui le caractérisent, mais il ne doit exister qu'*une seule* méthode d'accès pour un carré, retournant la longueur d'un de ses côtés [10].

3.2 Introduire une dose d'héritage multiple

Nous souhaitons maintenant introduire les *losanges* dans notre hiérarchie d'objets géométriques : rappelons qu'un **losange** est un paralélogramme dont les deux côtés ont même longueur. D'un point de vue « informatique », le calcul du périmètre d'un losange est simplifié par rapport à celui d'un parallélogramme « plus général ». Dès lors, deux classes sont à présent des sous-classes de `Parallelogram` : la classe `Rectangle` et la nouvelle classe `Lozenge`. Conceptuellement, nous pourrions considérer que les carrés étant à la fois des rectangles et des losanges, la classe `Square` hérite à la fois des classes `Rectangle` et `Lozenge`, mais rappelons que Ruby ne permet pas l'héritage multiple. Un compromis intéressant consiste à ce que la classe `Square` hérite uniquement de la classe `Rectangle` mais utilise, pour le calcul du périmètre d'un carré, la méthode éponyme de la classe `Lozenge`. C'est-à-dire que le tableau 1 est revu et complété comme nous l'indiquons dans la table 2. Cet effet peut se réaliser au moyen des *modules* de Ruby.

Ne pas perdre de vue que la classe `Lozenge` doit être assujettie aux contraintes (C1), (C2) et (C3), déjà énoncées en ce qui concerne les classes `Quadrilateral`, `Parallelogram`, `Rectangle` et `Square` (*cf.* § 3.1).

10. Nous rappelons l'existence de la construction `undef_method` en Ruby.