A. Les classes

1. Introduction

Avant d'utiliser le potentiel de Ruby, il faut tout d'abord avoir quelques notions de programmation objet. Dans un langage objet, le développeur réalise des classes, c'est une étape préparatoire. Une classe décrit les possibilités d'un ensemble d'objets, ces objets donneront vie à l'application.

C'est un peu comme un plan de fabrication pour de futurs objets. Ce système en deux temps, préparation et fabrication d'objets est en réalité assez naturel. En effet, tout autour de vous, il y a des objets, par exemple une chaise, une table... Toutes les chaises, les tables... ont des caractéristiques communes attachées à leur définition. La chaise est un siège à dossier avec ou sans accoudoir par exemple. C'est parce que certains objets respectent la définition d'une chaise qu'ils vont être considérés comme tel.

Cette définition de la chaise en programmation objet deviendra une classe Chaise. On y trouvera donc ce qui caractérise notre chaise. Dans cette classe, nous aurons les actions autorisées (déplacer, orienter la chaise...), mais également l'état de l'objet au fur et à mesure de son utilisation (inclinée, usagée, détériorée...).

Autre exemple pour la notion d'état, imaginez une classe Stylo, nous devons pouvoir associer une cartouche d'encre à nos objets stylos. Un objet stylo peut au cours de son utilisation avoir différentes cartouches, d'où la nécessité de conserver quelque part cette information sinon notre objet pourrait presque devenir inutilisable (on ne saurait pas s'il y aurait ou non une cartouche, ni quelle serait sa couleur).

Les actions sont traduites en programmation objet par des méthodes de classe. L'état est maintenu au travers de variables de classe que l'on nomme attributs.

Pour résumer, les classes donnent un cadre d'exécution à des objets. L'usage des objets donne naissance à une application.

2. La classe dans Ruby

a. Structure

Une classe en Ruby est définie dans un bloc de forme :

```
class MaClasse
  # Attributs & Méthodes
end
```

À noter qu'un nom de classe commence par une majuscule par convention.

Commençons par modéliser une personne.

```
class Personne
  def bonjour
    puts "Bonjour"
  end
end
```

Pour construire des objets `Personne`, nous avons la fonction `new` à disposition. Cette fonction est un peu particulière car elle sert à construire et initialiser en même temps chaque objet.

```
alex = Personne.new
alex.bonjour
```

Nous obtenons bien en sortie :

```
Bonjour
```

Vous remarquerez que la classe `Personne` ne décrit pas toutes les formes de personne, mais seulement une forme suffisante à notre besoin. C'est le principe de la modélisation : traduire une réalité dans une forme abstraite et adaptée à un service que l'on souhaite obtenir.

Lorsque vous créez un objet, l'objet courant est toujours accessible par le mot-clé `self`.

Exemple de code avec `self`, en supposant que nous ayons une méthode prenant en argument un objet de type `Personne` :

```
class Personne
  def bonjour()
    salut(self)
  end
  def salut(unePersonne)
    "salut @#{unPersonne}"
  end
end
```

À l'opposé de self, le mot-clé `nil` désigne une absence d'objet.

b. Construction d'une classe en plusieurs étapes

Avec Ruby, une classe n'est jamais définitivement construite, c'est-à-dire qu'il est toujours possible de compléter la définition d'un classe. Tous les objets qui auront été construits précédemment disposeront alors de ces changements.

Exemple :

```
# Nous avons écrit une deuxième classe Personne à la suite de
# l'exemple précédent
class Personne
  def auRevoir
   puts 'Au revoir'
  end
end
alex.auRevoir
```

L'objet `alex` peut maintenant vous dire « bonjour » et « au revoir », nous n'avons pas eu besoin de créer un nouvel objet. Tous les prochains objets pourront également dire « bonjour » et « au revoir » :

```
phileas = Personne.new
phileas.bonjour
```

Il est également possible de réécrire une méthode, par contre l'effet ne sera pas rétroactif, autrement dit, ce changement n'aura d'effet que pour les prochains usages.

```
phileas.auRevoir
class Personne
  def auRevoir
    puts 'Adieu'
  end
end
phileas.auRevoir
```

Nous obtenons en sortie :

```
Au revoir
Adieu
```

On peut résumer ce système par : toute modification d'une classe n'aura d'effet que sur les prochains usages touchant à ces modifications, c'est lié à l'ordre d'exécution.

Ce système à définitions répétées a des effets intéressants, notamment il vous est permis de changer le comportement des classes prédéfinies (entiers, décimaux...).

Exemple :

```
class Float
  def quoi
    "C'est un décimal"
  end
end

puts 2.0.quoi
```

La classe `Float` est prédéfinie et concerne les décimaux. Nous lui ajoutons une méthode `quoi`. Nous obtenons bien en sortie : `C'est un décimal`.

3. Le message

Lorsqu'une action est invoquée, on parle de message envoyé à l'objet. La méthode `send` est prévue à cet effet, vous n'avez pas d'intérêt à l'invoquer directement mais cela peut aider à comprendre ce qui se passe lors de l'appel d'une méthode.

```
class Personne
  def adieu( id )
    puts "Adieu #{id}"
  end
end
phileas.send :adieu, "M."
```

Nous obtenons en sortie : `Adieu M.`

Pour assouplir ce système et pouvoir invoquer une méthode uniquement avec son nom, vous disposez de la fonction `method`. Cela peut être intéressant pour exécuter un code dynamiquement.

```
m = phileas.method( 'adieu' )
m.call "Mme"
```

Nous obtenons en sortie :`Adieu Mme`

Il vous est également possible d'intercepter les messages incorrects par la méthode `method_missing`.

```
class Personne
  def method_missing( m )
    puts "Je ne sais pas faire #{m.id2name}"
  end
end
phileas.leTourDuMonde
```

Nous obtenons alors en sortie :`Je ne sais pas faire leTourDuMonde`

Pour connaître toutes les méthodes (accessibles) d'un objet, nous avons `methods` à disposition. Cette dernière nous retourne un tableau avec toutes les méthodes disponibles (y compris celles obtenues par héritage, concept que nous expliquerons ultérieurement).

```
for m in Personne.methods
  puts "On peut demander #{m}"
end
```

4. Définition d'opérateurs

Lorsque vous réalisez une opération de type `2 + 1` en réalité, il s'agit de l'appel d'une méthode `+` sur un objet de valeur `2` et prenant en argument la valeur `1`.

Exemple :

```
class Nombre
  attr_reader :val
  def initialize( val )
    @val = val
  end
  def +( autreNombre )
    @val = @val + autreNombre.val
    self
  end
  def to_s()
    @val
  end
end

n = Nombre.new( 10 )
m = Nombre.new( 20 )
o = n + m
puts o.val
```

Nous obtenons bien en sortie `30`.

Nous pouvons également redéfinir le rôle des crochets, à la fois en écriture et en lecture, ce qui revient à utiliser notre objet un peu comme un tableau ou une table de hachage.

Exemple :

```
class Salle
  def initialize()
    @places = {}
  end
  def[]=( place, eleve )
    @places[ place ] = eleve
  end
  def[]( place )
    @places[ place ]
```