Une introduction à Ruby

v 0.90

G. Burri

21 février 2003

Table des matières

1 Introduction

1.1 À qui s'adresse ce document ?

Ce document s'adresse aux personnes souhaitant commencer le Ruby, découvrir ses atouts et ses possibilités. Des connaissances en programmation procédurale et notamment en programmation orienté objet sont conseillées au préalable. La notion d'objet ne sera que brièvement expliquée au chapitre 3.

Il est conseillé de lire ce document d'une manière séquentielle car la plupart des chapitres s'appuient sur des notions se trouvant dans de précédents chapitres. Il en va de même avec les exemples de codes donnés dont leur «complexité» va en augmentant.

1.2 Bref historique

Ruby a été créé le 24 février 1993 par un japonais nommé Yukihiro Matsumoto dépité de ne pas trouver de langage de script strictement orienté objet. Il connaissait *perl (perl4)* mais le trouvait trop apparenté à un langage «joujou» et non à un vrai langage clairement défini. Il existait bien *Python* à l'époque mais ce n'était pas un langage totalement orienté objet.

La version publique de Ruby dont les caractéristiques s'inspiraient passablement de perl est sortie en décembre 1995. Depuis lors la communauté de Ruby n'a cessé de croître surtout au Japon dans un premier temps puis ensuite dans le reste du monde.

Il faut noter qu'actuellement au Japon le nombre d'utilisateur de Ruby a dépassé celui de *Python*.

1.3 Qu'est ce que Ruby ?

Ruby est un langage de programmation interprété et totalement orienté objet. Le fait qu'il soit interprété signifie que le code est entièrement portable d'un système d'exploitation à un autre pour autant que l'interpréteur existe et soit installé.

La bibliothèque standard (voir chapitre 7) ainsi que la bibliothèque intégrée (voir chapitre 6) livrées avec Ruby vous apportent tous les outils nécessaires à la programmation de haut niveau tel que :

- Tableaux dynamiques (Piles et Queues).
- Processus et processus légers (thread).
- Fonctions mathématiques et outils tel que la gestion de matrices ou de nombre complexes.
- Gestion de fichiers et de répertoires.
- Gestion du temps et des dates.
- Réseau : Sockets TCP et UDP, protocole HTTP, FTP, IMAP, POP3, SMTP, Telnet.
- CGI (gestion des cookies et des sessions, production d'entête HTTP et des contenus HTML).
- Programmation concurrente : moniteurs, mutex (zone d'exclusion mutuel), queue de communication inter-thread (rendez-vous).
- Persistance de données.
- Motifs de conception (design pattern).

De plus il est très facile avec Ruby de faire des interfaces-utilisateur (GUI) grâce à FXRuby (FOX pour Ruby) ou encore avec Tk. Il est aussi possible de faire de l'OpenGL avec GLUT ou en l'intégrant dans un `'canvas'` FXRuby. Toutes ces bibliothèques supplémentaires sont fournies par défaut sous Windows.

Ruby est totalement orienté objet, tout ce que l'on manipule ne sont que des objets. Par exemple le nombre '-23' est une instance de la classe 'fixnum' qui est la classe pour les petits nombres entiers. Si l'on désire connaître les méthodes que l'on peut y appliquer, il suffit d'aller voir dans la classe 'fixnum' et ses parents. Par exemple on peut y trouver la méthode 'abs' qui renvoie la valeur absolue.

Voici une illustration, la valeur renvoyée est notée après '>>' ceci sera valable pour toutes les illustrations de ce document.

```
-23.abs          >> 23
```

Ruby à été créé pour simplifier au maximum la programmation, Sa syntaxe est claire et facile à comprendre. De plus Ruby fournit une quantité impressionnante de bibliothèques, nous en parleront plus en détail aux chapitres 6 et 7.

Voici un petit exemple de code :

```
def factorielle(n)
    if n == 0
        1
    else
        n * factorielle(n - 1)
    end
end

puts factorielle(10)
```

Il est possible d'inclure directement ce code dans un fichier et de l'utiliser tel quel, sans que la méthode 'factorielle' n'est besoin d'être dans une classe, en réalité cette méthode fait implicitement partie d'un classe, la section 3.8 en dit plus sur ce sujet.

Les méthodes privées, comme la méthode `'factorielle'` du code ci dessus, s'utilisent sous forme fonctionnelles (sans récepteur), on peut donc les appeler des 'fonctions'.

Je pense que vous avez deviné que 'puts' est une fonction d'affichage faisant partie de 'Kernel'. Notez également qu'il n'est pas nécessaire d'utiliser le mot-clef 'return' pour renvoyer une valeur.

Lorsque l'on exécute un fichier Ruby le code est interprété, ce qui signifie qu'il n'y a pas de phase de compilation contrairement à des langages comme C/C++ ou Ada. Ce type de langage est communément appelé (à tort) un langage de script.

Un autre avantage est le fait qu'il soit entièrement dynamique, les types des variables et des expressions sont déterminés à l'exécution. Il est possible de générer un programme dans un programme puis de l'exécuter ou encore d'ajouter à la volée des méthodes à un objet.

La notion de bloc qui est décrite au chapitre 5 est un point très important du langage Ruby. En voici une petite illustration :

```
tab = [1, 2, 3, 4, 5]
tab.each{|element|
    puts element
}
```

Ce code affiche les éléments du tableau 'tab'.

2 Installation

2.1 Téléchargement

Vous pouvez télécharger différentes versions de l'interpréteur Ruby à cette adresse : `http://www.ruby-lang.org/en/20020102.html`. En particulier sur cette page pour la version MS Windows : `http://rubyinstaller.sourceforge.net/`

2.2 Linux

La version sous Linux doit être compilée avant d'être utilisable. De plus elle est livrée sous le format Tar + Gzip se qui signifie que c'est une archive compressée. Avant la compilation/installation il faut dans un premier temps décompresser le fichier.

Contrôlez que vous possédez bien l'application Gzip en tapant à la console 'gzip –help' et que vous possédez également l'application 'Tar' en tapant 'Tar –help'.

Voici les sites de 'Gzip' respectivement 'Tar' sur lesquels il vous est possible de télécharger l'une ou l'autre de ces deux applications. Vous trouverez toutes les informations nécessaires à leur installation sur leur site web respectif :

- `http://www.gnu.org/software/gzip/gzip.html`
- `http://www.gnu.org/software/tar/tar.html`

Voici la commande à appliquer pour décompresser le fichier en admettant que son nom soit 'ruby-1.6.8.tar.gz' et qu'il se trouve dans le répertoire courant (l'invite de commande est ici représentée par un '$') :

```
$ gunzip -d ruby-1.6.8.tar.gz
```

Puis défaites l'archive ainsi obtenue qui doit s'appeler 'ruby-1.6.8.tar' comme ceci :

```
$ tar -xf ruby-1.6.8.tar
```

'Tar' vous à créé un répertoire du nom de l'archive contenant tous les fichiers de Ruby. Il ne vous reste plus qu'a le compiler et à l'installer. Ceci n'est pas expliqué ici mais se trouve dans le fichier 'README' contenu dans le repertoire créé.

Pour afficher le contenu vous pouvez utiliser l'application 'cat' :

```
$ cat README | less
```

ou

```
$ cat README | more
```

dans le cas où 'less' n'est pas installé.

Entête des fichiers Ruby

Il est conseillé de placer au début de chaque fichier Ruby susceptible d'être exécuté l'entête suivante :

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

Cette entête va permettre au 'shell' de trouver le chemin de l'interpréteur sans que l'on ait besoin de lui indiquer explicitement, dans notre cas il se trouve dans le répertoire '/usr/local/bin/ruby'.

On va pouvoir ensuite lancer un 'script' Ruby comme ceci :

```
$ ./mon_script.rb
```

2.3 Windows

L'installation sous Windows est très facile, après avoir téléchargé le programme d'installation il vous suffit de l'exécuter et de suivre les instructions.

Il est nécessaire après l'installation de redémarrer l'ordinateur pour que la variable d'environnement 'PATH' soit mise à jour.

Pour exécuter un 'script' Ruby il suffit de taper à la console son nom :

```
> mon_script.rb
```

2.4 Mon premier programme Ruby

Nous allons maintenant tester si l'installation de Ruby à correctement été effectuée en réalisant un petit «Hello, World !».

En premier lieu tapez ceci dans la console (valable pour Windows et Linux) :

```
$ ruby -v
```

Vous devriez voir la version de l'interpréteur installée. Si ce n'est pas le cas alors veuillez vous reportez aux sections précédentes et vérifier si Ruby est correctement installé, il se peut que sous windows la variable d'environnement 'PATH' n'est pas été mise à jour, dans ce cas l'interpréteur de commandes ne peut pas trouver l'interpréteur Ruby. Ceci peut se produire si l'on a pas redémarré l'ordinateur après l'installation.

Si l'interpréteur Ruby est correctement installé, alors créez un fichier texte se nommant 'hello.rb', puis à l'aide d'un éditeur quelconque tel que Vi sous Un*x ou Notepad sous Windows tapez les deux lignes suivantes :

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

puts 'Hello, World!'
```

Notez que '#!/usr/local/bin/ruby' n'est utilisé que sous les systèmes Un*x mais qu'il est préférable de le mettre systématiquement pour des raisons de compatibilités. Il sera ignoré par l'interpréteur Ruby puisque c'est un commentaire, les commentaires en Ruby commencent par un dièse : '#'.

Enregistrez-le puis exécutez-le comme ceci :

Sous Linux :

```
$ ./hello.rb
```

Sous Windows :

```
> hello.rb
```

Notez qu'un retour à la ligne après le message à automatiquement été ajouté, ceci est fait par la fonction 'puts'. Pour ne pas avoir de retour à la ligne il faut utiliser la fonction 'print'.

En cas de problèmes

Dans le cas ou le fichier n'a pas pu être exécuté veuillez vérifier les points suivants :

Sous Linux :

- Le dossier où se trouve l'interpréteur Ruby doit se trouver dans la variable d'environnement 'PATH', tapez ceci pour l'afficher et vérifier qu'il s'y trouve bien :
  ```
  $ echo $PATH
  ```
  Dans le cas où le répertoire n'est pas présent vous pouvez l'ajouter comme ceci :
  ```
  $ set PATH=$PATH:<nouveau répertoire>
  ```
  Où '<nouveau répertoire>' est le nouveau répertoire.
- Vous n'avez pas les droits d'exécuter le fichier Ruby. Pour ajouter le droit correspondant à l'exécution procédez comme ceci (en admettant que vous êtes le propriétaire du fichier) :
  ```
  $ chmod u+x <nom fichier source>
  ```
  Où '<nom fichier source>' est le nom du fichier Ruby.
- L'entête '#!/usr/local/bin/ruby' n'est pas correct et le shell vous dit qu'il ne trouve pas l'interpréteur. Dans ce cas essayez de localiser l'interpréteur grace au programme 'locate' :

```
$ locate ruby
```

Il est possible qu'il soit dans '/usr/bin' ('#!/usr/bin/ruby').

Sous Windows :

Comme sous Linux, Le dossier où se trouve l'interpréteur Ruby doit se trouver dans la variable d'environnement 'PATH', tapez ceci pour l'afficher et vérifier qu'il s'y trouve bien :

Sous Windows :

```
> PATH
```

Également comme sous Linux : dans le cas où le répertoire n'est pas présent vous pouvez l'ajouter comme ceci :

```
> set PATH=%PATH%;<nouveau répertoire>
```

Où '<nouveau répertoire>' est le nouveau répertoire.

2.5 Un éditeur Ruby

Sous Windows 'RubyWin' est fournit d'office avec l'installation de Ruby. Sous Linux il n'y a pas d'éditeur fournit. Je recommande personnellement l'utilisation de l'éditeur *Jext* qui est écrit en Java ce qui veut dire qu'il consomme une grande quantité de mémoire et de temps CPU mais ce qui signifie aussi qu'il existe sous Windows comme sous Linux ce qui peut être un avantage lorsque l'on passe d'une plateforme à l'autre. *Jext* inclut par défaut la coloration syntaxique pour Ruby et pour également un grand nombre d'autres langages.

Vous pouvez télécharger et en savoir plus sur *Jext* en vous rendant sur le site officiel : `http://www.jext.org`.

3 Un langage totalement orienté objet

3.1 Introduction à l'objet

Au fil du temps la programmation dite procédurale tel que le langage *c* a commencée à montrer ses faiblesses quant à l'organisation de programmes, surtout lorsque la taille de celui-ci devenait importante et qu'un grand nombre de personnes travaillaient ensemble dessus.

De ce problème, est née la programmation orientée objet qui permet de concevoir des applications d'une façon structurée et bien définies grâce à certains nombre de concepts que nous survoleront en peu plus loin dans cette section.

Pour parler simplement nous pouvons dire qu'un objet contient à la fois des données et le code permettant de les traiter. On pourrait dresser un parallèle avec le type `'struct'` du langage *c* qui regroupe un certains nombre de données hétérogènes auxquelles on y ajouterait des fonctions pour traiter ces données. Ces fonctions serait encapsulées à l'intérieur du `'struct'`. On ne pourrait donc y accéder que depuis une variable du type du `'struct'`, il faudrait alors post-fixer le nom de cette variable de la fonction à laquelle on souhaite accéder.

En objet on appelle une fonction une méthode.

Il existe, sans entrer dans les détails, trois concepts fondamentaux définissant la programmation orientée objet :

- **L'encapsulation :** Ce concept consiste à encapsuler les données et le code, c'est à dire à «cacher» les données et le code à l'utilisateur et à ne lui montrer qu'une interface qui va lui servir à communiquer avec l'objet.
- **L'héritage :** Voici une notion fondamentale de la programmation objet. Un objet peut hériter des caractéristiques d'un autre objet en redéfinissant certaines méthodes et/ou en ajoutant des méthodes ou des données supplémentaires.
- **Le polymorphisme :** Cette notion découle de l'héritage. Prenons l'exemple d'un objet `'A'` contenant la méthode `'afficher'` et que l'on dérive de cet objet un objet `'B'` il se pourrait alors que l'objet `'B'` ait besoin de s'afficher d'une autre manière, dans ce cas il faut «redéfinir» la méthode `'afficher'`.
  L'objet `'A'` et l'objet `'B'` auront alors chacun une méthode `'afficher'`, si on utilise un objet de type `'A'` et que l'on appelle la méthode `'afficher'` alors c'est le `'afficher'` de `'A'` qui sera appelé, si l'on utilise un objet `'B'` héritant des méthodes de `'A'` mais redéfinissant la méthode `'afficher'` alors ce sera la méthode redéfinie qui sera appelée, c'est à dire le `'afficher'` de `'B'`.

Prenons un exemple concret : le but est de gérer une vidéothèque. On peut identifier deux objets : **la vidéothèque** en elle même qui va gérer une liste de films et **le film**.

Commençons par décrire un film :

```
class Film
    def initialize(titre, duree)
        @titre = titre
        @duree = duree
    end

    def dureeHM
        return (@duree / 60).to_s + 'h ' +
        (@duree % 60).to_s + 'min '
    end

    def afficher
        puts @titre + ' ( ' + self.dureeHM + ')'
    end
end
```

Un film contient deux données : son titre et sa durée (en minutes) et trois méthodes : `'initialize'`, `'dureeHM'` et `'afficher'`. Il est important de noter que le nom du type d'un objet, ici `'Film'`, doit toujours commencer par une majuscule et que les noms des méthodes commencent toujours par une minuscule.

La méthode `'initialize'` est spéciale et va être appelée lors de la création de l'objet, elle n'est pas obligatoire mais dans notre cas elle va permettre de directement définir le titre du film ainsi que sa durée lors de la création d'un objet de type `'Film'`.

La méthode `'dureeHM'` renvoie simplement la durée du film en heures + minutes sous la forme d'une chaîne de caractères. Le `'return'` précédant la valeur à renvoyer n'est pas obligatoire.

La méthode `'afficher'` affiche le titre du film et sa durée. l'appel à une méthode de l'objet depuis une autre méthode de ce même objet est désigné par `'self'`. Le `'self'` désigne en fait l'objet dans lequel on se trouve.

Les données d'un objet débute toujours par un `'@'`, il est recommandé de les définir dans la méthode `'initialize'` mais ce n'est pas obligatoire. Les données de l'objet ne sont pas visibles depuis l'extérieur (voir les sections 3.6.1 et 3.6.2) , seules les méthodes peuvent être utilisées.

Décrivons maintenant notre vidéothèque qui contiendra les films :

```
class Videotheque
    def initialize
        @lesFilms = Array::new
    end

    def ajouterUnFilm(nouveauFilms)
        @lesFilms.push(nouveauFilms)
    end

    def afficherTitres
        @lesFilms.each{|film|
            film.afficher
        }
    end
end
```

Chaque classe que l'on définit dont on ne spécifie aucun parent (elle n'hérite de personne) hérite automatiquement de la classe `'Objet'`. Cette classe `'Objet'` contient certaines méthodes comme `'new'` qui est employé pour créer de nouveau objet. Cette méthode est une méthode de classe (voir section 3.4.2).

Dans la méthode d'initialisation on définit une variable `'@lesFilms'` de type tableau (ou liste, cela dépends du point de vue). `'Array : :new'` crée un tableau vide.

Dans la méthode `'ajouterUnFilm'` on utilise la méthode `'push'` sur la liste de films pour insérer un nouvel objet, dans notre cas un objet de type Film.

La méthode `'afficherTitres'` on utilise la méthode `'each'` qui va exécuter le bloc (voir chapitre 5) pour chaque objet de la liste en lui passant comme paramètre l'objet en question.

Après avoir défini nos deux type nous pouvons les utiliser en instanciant[1] notre classe `'Videotheque'`.

```
maVideotheque = Videotheque::new
maVideotheque.ajouterUnFilm(Film::new('LasVegas Parano', 111))
maVideotheque.ajouterUnFilm(Film::new('Requiem for a Dream', 97))
maVideotheque.ajouterUnFilm(Film::new('Fight Club', 132))
maVideotheque.afficherTitres
```

[1]Instancier est synonyme, en langage objet, de «créer un nouvel objet à partir d'une classe»

On crée une variable `'maVideotheque'` de type `'Videotheque'` puis on lui ajoute trois films à l'aide de la méthode `'ajouterUnFilm'`. Pour chaque appel à la méthode d'ajout de film on crée directement un film en utilisant la méthode de class 'new' et en lui passant son titre et sa durée en minute.

Pour finir on affiche les films contenus dans notre vidéothèque en appellant la méthode `'afficherTitre'`.

Voici le résultat de cet exemple après exécution :

```
Las Vegas Parano ( 1h 51min )
Requiem for a Dream ( 1h 37min )
Fight Club ( 2h12min )
```

3.2 Données membres

Les données membres en Ruby sont obligatoirement précédées d'un `'@'` ce qui les rends facilement identifiable. En Ruby les données membres sont automatiquement «privées» à l'objet, elles ne peuvent être utilisés qu'à l'intérieur de l'objet et non à l'extérieur ce qui diffère de beaucoup de langage tel que Python.

Il est fortement conseillé de déclarer toutes les données membres dans la méthode d'initialisation (le constructeur).

Voici un petit exemple :

```
class Choses
    def initialize(a)
        @a = a + 1
        @b = @a * 2
        @t = Array::new
    end
end
```

3.3 Données à l'échelle de classe

Les données à l'échelle de classe commencent par `'@@'`. Elles peuvent bien sur être déclarées à l'extérieur des méthodes comme à l'intérieur.

Par exemple voici une classe `'Truc'` qui compte le nombre de fois qu'elle à été instancier :

```
class Truc
    @@nombreInstance = 0
    def initialize
        @@nombreInstance += 1
    end

    def Truc::nombreInstance
        puts @@nombreInstance
    end
end

Truc::nombreInstance
t1, t2, t3 = Truc::new, Truc::new, Truc::new
Truc::nombreInstance        >> 3
```

On peut constater que la variable `'@@nombreInstance'` est une donnée à l'échelle de la classe `'Truc'`. Elle est commune pour tous les objets créés et existe même lorsque aucun objet n'est créé.

Les variables `'t1'`, `'t2'`, `'t3'` sont affectées en parallèles, ceci est juste une forme raccourcie.

3.4 Les méthodes

Les méthodes sont toujours déclarées dans une classe, elles commencent pas le mot réservé `'def'` et peuvent être ensuite suivies d'un ou plusieurs paramètres. Un paramètre spécial commençant par un `'*'` peut être ajouté comme dernier paramètre, il va alors réunir dans un tableau tous les paramètres donnés en trop.

Il est bien sur possible de donner une valeur par défaut à un paramètre.

Une méthode peut, comme en C++, soit être publique, protégée ou privée.

Une méthode publique est accessible par tout le monde, que se soit depuis l'intérieur de la classe dont elle fait partie, depuis les enfants de la classe ou depuis l'extérieur (utilisation de la classe en l'instanciant). Par défaut les méthodes sont publiques.

Une méthode protégée est accessible uniquement depuis l'intérieur de la classe dont elle fait partie et depuis les enfants de la classe. Mais elle n'est pas visible depuis l'extérieur.

Vous l'aurez sans doutes devinez, une méthode privée ne peut être utilisée qu'a l'intérieur de sa propre classe.

La définition du type d'accès d'une méthode se fait à l'aides des mots réservés `'public'`, `'protected'` et `'private'` qui définissent des zones, il existe une autre manière de définir les type d'accès mais cela n'est pas vu ici. Voici un petit exemple :

```
class Bidon
    #par défaut les méthodes suivantes sont publiques
    def a; end
    def b; end

    #les méthodes suivantes sont privées
    private

    def c; end
    def d; end

    #on repasse en publique
    public

    def e; end
    def f; end

    #on fini en protégé
    protected

    def g; end
    def h; end
end
```

3.4.1 Méthodes membres

Déclaration

Une méthode membre fait partie de l'instance d'une classe, elle peut accéder au données membres de celle-ci. C'est à dire aux données représentant l'état d'un objet.

Exemple :

Utilisation

Pour appeler une méthode membre il faut utiliser l'opérateur `'.'` sur un objet suivi du nom de la méthode et des éventuels paramètres. Ces paramètres peuvent être entre parenthèses ou non, nous vous conseillons tout de même de systématiquement les employer pour une lecture plus aisée.

```
class A
    def x
        puts 'x'
    end

    def y(s, u = 'u'); puts s, u; end

    def z(v, *w)
        puts v; puts w
    end
end
```

```
a = A::new

a.x

a.y 'hello'

a.z('hello', 'salut', 'ciao')
```

Exemple utilisant la classe `'A'` précédemment déclarée :

L'appel à `'y'` n'utilise pas de parenthèse (déconseillé).

L'appel à `'z'` donne plus de paramètres que prévu, les paramètres excédents vont être placés dans la variable `'w'` sous la forme d'un tableau que l'on affiche dans notre cas.

La fonction `'puts'` ainsi que la fonction `'print'` vont faire appels à la méthode `'to_s'` de l'objet à afficher pour le convertir en chaîne de caractère (to string). Une telle méthode est définit pour le type Array (qui est ici le type de `'w'` dans `'y'`).

Une méthode peut se terminer par un `'!'` ou un `'?'`, ceci aide à la compréhension de la sémantique de la méthode, nous vous conseillons d'utiliser le `'!'` lorsque la méthode modifie un ou plusieurs paramètre et le `'?'` lorsque la méthode renvoie une valeur de type Booléenne : `'true'` ou `'false'`.

3.4.2 Méthodes à l'échelle de classe

Les méthodes à l'échelle de classe ne concerne qu'une classe et ne peuvent accéder qu'aux données à l'échelle de classe. Elles ne peuvent en aucun cas accéder à des méthodes ou données membres d'une classe. Par contre il est tout à fait possible d'accéder à des méthode ou des données à l'échelle de classe depuis des méthodes membres.

La définition d'une méthode à l'échelle de classe ressemble fortement à celle d'une méthode membre à la différence que son nom est précédé du nom de la classe et d'un point (`'.'`) ou de deux doubles points (`' : :'`).

Lors de l'appel à une tel méthode il suffit de la nommer comme elle a été déclarée, c'est à dire le nom de la classe un point ou deux double points et le nom de la méthode et bien entendu les éventuelles paramètres. Naturellement aucune instance de classe n'entre en jeu dans cet appel. un exemple impliquant ces méthodes est fournit à la section 3.3.

3.5 Héritage

L'héritage est une notion très importante de la programmation orienté objet, il permet d'étendre des classes avec de nouvelles fonctionnalités ou encore de mieux structurer un programme. Ruby ne supporte pas l'héritage multiple contrairement à C++ ou Python.

Voici un exemple d'héritage :

`'B'` dérive ou hérite (ces deux mots sont synonymes) de `'A'`.

```
class B < A
...
end
```

Pour un exemple plus concret reprenons notre vidéothèque de la section 3. Nous allons y ajouter le type de support des films : DVD et VHS. Un films DVD se caractérise par ses langues disponibles et un films VHS sa qualité.

Afin d'ajouter ces fonctionnalitées on pourrait ajouter une donnée membre `'type'` à notre type `'Film'` qui va définir si c'est un DVD ou une VHS ce qui engendrerait quelques problèmes comme le fait qu'une donnée sur la qualité soit toujours présente alors que l'on en a pas besoin dans le cas d'un DVD. De plus le code deviendrai plus lourd et moins compréhensible.

Pour faire cela plus élégamment nous allons avoir besoin du concept d'héritage. On va dériver de la classe `'Film'` deux sous-classes appelées DVD et VHS qui ajouteront les instructions supplémentaires nécessaires.

Voici le code qui a été ajouté (les classes `'Film'` et `'Videotheque'` n'ont pas changées) :

Les méthodes `'initialize'` de ces deux classes vont prendre les données passées en paramètres destinées au parent `'*donnees'` et appeler la méthode `'initialize'` du parent en lui donnant les paramètres `'*donnees'`. L'utilisation du `'super'` dans une méthode va appeler la méthode du parent ayant le même nom que la méthode dans laquelle on se trouve.

Chacune des nouvelles classes possèdent toutes les méthodes et données membres de la classe parente, dans notre cas il est possible d'utiliser la méthode `'dureeHM'` pour les trois type : `'Film'`, `'DVD'` et `'VHS'`.

Voici le résultat de cet exemple après exécution :

3.6 Accès aux données membres

En ruby les données membres sont par défaut inaccessibles depuis l'extérieur, il faut pour les rendre visibles utiliser une méthode pour la modification et/ou une méthode pour la lecture (qui va se contenter de renvoyer la valeur de la donnée membre).

Pour simplifier Ruby met à disposition un mécanisme de définition d'accès pour les données membres.

3.6.1 Lecture

Si l'on veut autoriser à lire une donnée membre directement depuis l'extérieur il faut le spécifier quelque part dans la classe (en général à la fin) en utilisant l'instruction `'attr_reader'`. Voir la section 'Écriture' pour un exemple.

3.6.2 Écriture

L'autorisation en écriture se fait de la même manière que la lecture mais en utilisant l'instruction `'attr_writer'`. Attention, ce mode ne donne l'accès qu'en écriture, pour pouvoir lire et écrire une donnée membre il faut utiliser les deux instructions : `'attr_reader'` et `'attr_writer'`. Voici un exemple de données en lecture/écriture :

3.7 Données virtuelles

Il est possible de faire croire à l'utilisateur qu'il manipule une données existant réellement dans l'objet alors qu'on lui montre une représentation différente. Admettons que l'on souhaite pouvoir affecter et lire la durée d'un film en secondes (en non en minutes), le création d'un attribut virtuel est, dans ce cas, adéquat :

Dans cet exemple, la méthode 'dureeEnSeconde' tient le rôle d'accesseur et la méthode 'dureeEnSeconde=' tient le rôle de modificateur.

```
class DVD < Film
    def initialize(*donnees)
        super(*donnees)
        @listeLangue = Array::new
    end

    def ajouterLangue(langue)
        @listeLangue.push(langue)
    end

    def afficher
        print '[DVD] '
        super
        puts 'Langues : ' + @listeLangue.join(', ')
    end
end

class VHS < Film
    def initialize(qualite, *donnees)
        super(*donnees)
        @qualite = qualite
    end

    def afficher
        print '[VHS] '
        super
        puts 'Qualité : ' + @qualite
    end
end

maVideotheque = Videotheque::new
film = DVD::new('Las Vegas Parano', 111)
film.ajouterLangue('Francais')
film.ajouterLangue('Anglais')
film.ajouterLangue('Italien')

maVideotheque.ajouterUnFilm(film)
maVideotheque.ajouterUnFilm(VHS::new('moyenne', 'Requiem for a Dream', 97))
maVideotheque.ajouterUnFilm(VHS::new('bonne', 'Fight Club', 132))
maVideotheque.afficherTitres
```

```
[DVD] Las Vegas Parano ( 1h 51min )
Langues : Francais, Anglais, Italien
[VHS] Requiem for a Dream ( 1h 37min )
Qualité : moyenne
[VHS] Fight Club ( 2h 12min )
Qualité : bonne
```

3.8 L'objet 'main' et la classe 'Object'

Lorsque l'on exécute du code Ruby il ne s'exécute pas réellement hors contexte mais à l'intérieur d'une méthode d'un objet (tout est objet en Ruby). Cet objet s'appel 'main', il est une instance de la class 'Objet' de la bibliothèque standard. Il est possible de voir ces informations comme ceci :

Lorsque l'on définit une méthode dans une méthode de classe, celle-ci va s'ajouter comme

```
class Film
    def initialize(titre, duree)
        @titre = titre
        @duree = duree
    end

    attr_writer :titre
    attr_reader :titre, :duree
end

f = Film::new('Las Vegas Parano', 111)
f.titre = 'Les autres'
puts f.titre
```

```
class Film
    def initialize(titre, duree)
        @titre = titre
        @duree = duree #stocké en minute
    end

    def dureeEnSeconde
        @duree * 60
    end

    def dureeEnSeconde=(valeur)
        @duree = valeur / 60
    end
end

f = Film::new('Las Vegas Parano', 111)
f.dureeEnSeconde = 6660
puts f.dureeEnSeconde
```

```
puts self.class  #affiche le type de l'objet dans lequel on se trouve

self.display     #affiche le nom de l'objet dans lequel on se trouve
```

méthode privée de classe de la classe `'Object'` :

```
class A
    def x
        def y
            puts 'y'
        end
        y
    end
end

A::new.x
```

Dans cet exemple on a ajouter la méthode `'y'` à la classe `'A'`.

Ceci explique pourquoi il est possible de définir des méthodes sans qu'elles fassent parties d'une classe puisque elles font partie en réalité de la classe `'Objet'` qu'implemente `'main'` :

```
def fonction
    puts 'je suis une fonction privée à l\'échelle
de classe de la classe \'Object\'' end

fonction
```

3.9 Étendre une classe existante

En Ruby il est possible de définir une classe en plusieurs parties, cela peut paraître bizarre à première vue mais va être très utile pour étendre une classe sans avoir à dériver de celle-ci. Voir la section 4.6 traitant des modules pour plus de détails.

```
class A
    def x
        puts 'x'
    end
end

# un peu plus loin...

class A
    def y
        puts 'y'
    end
end

a = A::new
a.x
a.y
```

Voici un autre exemple qui étend la classe `'Array'` incluse dans Ruby en lui ajoutant une méthode qui renvois le premier et le dernier élément du tableau. Cet exemple est purement démonstratif, son utilité reste encore à être prouvée.

```
class Array
    def debutFin
        [self[0], self[-1]]
    end
end

tab = [1, 2, 3, 4, 5]

puts tab.debutFin       >> [1, 5]
```

Le `'self'` représente l'objet lui même, dans notre cas un tableau. Toutes les méthodes propres aux tableaux peuvent donc lui être appliquées y compris la méthode `'[]'`.

4 Les bases du langage

4.1 Généralités

Fins de lignes

Les fins de lignes ainsi que les points-virgules représentent la fin d'une instruction. Il n'est donc pas nécessaire de terminer chaque instruction par un point-virgule comme en *c*. Cependant, si Ruby rencontre des opérateurs comme +, -, ou un anti slash (\) à la fin de la ligne, il considère qu'ils indiquent que l'instruction se continue sur la ligne suivante.

Importation de fichier

Il est bien sur très utile de pouvoir répartir les classes et modules que l'on utilise dans plusieurs fichiers afin d'avoir une meilleure gestion du code ainsi qu'une plus grande modularité (si l'on veut réutiliser une classe ou un module, il suffit de reprendre les fichiers la ou le concernant).

L'importation se réalise à l'aide de la fonction `'require(<nom fichier>)'`. ¡nom fichier¿ est le nom du fichier à importer, il peut se trouver dans un repertoire, dans ce cas il suffit d'ajouter le chemin devant le nom du fichier

Le fichier peut être un fichier de source Ruby (`'.rb'`) ou une bibliothèque d'extension (`'.so'`).

`'Require'` ne chargera pas plusieurs fois la même bibliothèque, par exemple, dans le cas de plusieurs inclusions du même fichier.

Chaîne de caractère

Une chaîne est une instance de la classe `'String'`. Il y a deux manière d'écrire une chaîne, soit en utilisant des apostrophes comme délimiteurs soit des guillemets : 'hello' ou "hello".

Dans le cas de l'utilisation des guillemets il est possible de placer des notations utilisant l'anti-slash et des expressions.

Voici une partie des notations anti-slash possibles :

Séquence	*Caractère représenté*
\n	Retour à la ligne
\r	Retour de chariot
\b	Backspace
\e	Échappement

Il est possible d'inclure des expressions directement dans une chaîne en la plaçant entre `'# { '` et `'} '`. Voici un exemple :

```
...
    def afficherHeure(minutes)
        puts "il est #{minutes / 60}h #{minutes % 60}"
    end
...
```

Variables globales

Ruby autorise l'utilisation de variables globales qui sont visibles dans n'importe quelle partie du programme. Une variable globale commence par un `'$'` :

```
$vGlob = 10
```

Il est important de minimiser l'utilisation de variables globales car cela va à l'encontre de la programmation objet.

Constantes

Une constante commence par une majuscule, une constante ne peut pas être déclarée dans le corps d'une méthode.

4.2 Fonctions intégrées

Les fonctions intégrées sont des méthodes provenant du module 'Kernel' ces méthodes peuvent être appelées sous la forme fonctionnelle (sans récepteur). On appelle ces méthodes des fonctions. Voici la liste des plus utiles qui pourront vous servir par la suite.

`block_given?`
Renvois true si un bloc à été passé (voir chapitre 5).

`exit([resultat = 0])`
Met fin au programme en renvoyant `'resultat'` comme code de retour.

`format(fmt[, param...])`
Renvois une chaîne formatée en applicant les paramètres `'param...'` sur la chaîne `'fmt'`. Fonctionne comme `'printf()'` en *c*.

```
v1 = 4.928301984

format("v1 = %.2f", v1)     >> 'v1 = 4.93'
```

`gets([rs = $/])`
Lit une ligne passé en ligne de commande ou sur l'entrée standard. `'rs'` définit le séparateur d'enregistrement. La valeur lue est automatiquement stocké dans la variable `'$_'`.

`puts([chaîne])`
Affiche `'chaîne'` sur la sortie standard. Un retour à la ligne est automatiquement ajouté à la fin.

`print([chaîne])`
Même chose que `'puts'` mais ne crée pas de retour à la ligne.

`srand([germe])`
Initialise le générateur de nombre aléatoire, si `'germe'` n'est pas donnée alors l'heure système est utilisée à la place.

`rand([max=1])`
Génère un nombre aléatoire entre 0 (inclus) et max (exclus).

`sleep([sec])`
Suspend l'exécution du programme pendant `'sec'` secondes. Si `'sec'` n'est pas fournit alors le programme est définitivement suspendu.

4.3 Opérateurs

Il n'y a pas grand chose à dire sur les opérateurs en Ruby car ils ressemblent à ceux que l'on trouvent dans les autres langages à quelques différences près. Soulignons qu'il existe comme dans le langage *c* des opérateurs raccourcis : `+= -=. *=` etc..

Pour le personne ne connaissant pas ces opérateurs, sachez que `'a += b'` correspond à `'a = a + b'`.

L'opérateur ternaire du *c* existe également en Ruby mais il est préférable d'utiliser la condition `'if'` :

```
condition = true

#ces deux expressions sont identiques
r = condition ? 'Vrai' : 'Faux'
puts r

r = if condition then 'Vrai' else 'Faux' end
puts r
```

4.4 Instructions conditionnelles

4.4.1 If

Pour exécuter une certaine partie de code ou une autre en fonction d'une condition il existe l'instruction `'if'`, voici sa syntaxe :

```
if <condition> [then]
    code
[elsif <condition> [then]
    code]...
[else
    code]
end
```

Si <condition> est vrai alors exécute le premier `'code'` sinon test les éventuelles conditions intermédiaire (`'elsif'`) jusqu'a en trouver une qui est vrai, dans ce cas exécute le `'code'` associé. Si aucune des conditions n'est vrai alors exécute le `'code'` du `'else'` si il existe.

4.4.2 Case

La structure de contrôle `'case'` permet d'exécuter un certain code en fonction de la valeur d'une expression.

```
case <expression> [then]
    code
[when <expression>[,<expression>] [then]
    code]...
[else
    code]
end
```

Compare l'expression spécifiée par `'case'` avec les différentes branches `'when'`, dès qu'une valeur correspond alors le `'code'` associé est exécuté. Si aucune des branches ne correspond alors exécute la branche `'else'` si elle existe. Voici un petit exemple pour illustrer :

```
v = 2
case v
    when 1
        puts '1'
    when 2
        puts '2'
    when 3
        puts '3'
end
```

Si v prends la valeur 1 alors affiche `'1'`, si il prends la valeur 2 alors affiche `'2'` si il prends la valeur 3 alors affiche `'3'`. Si v est égale à autre chose que ces valeurs alors rien ne se passe.

4.5 Boucles

Comme dans la plupart des langages, Ruby possède des structures itératives. Elles sont très simples à comprendre en ne seront ici qu'énumérées.

4.5.1 While

Exécute le code tant que <condition>est vrai.

```
while <condition> [do]
    code
end

begin
    code
end while condition
```

4.5.2 Until

Exécute le code tant que <condition>est faux.

```
until <condition> [do]
    code
end

begin
    code
end until condition
```

4.5.3 For

Itère dans <expression >.

```
for variable[, variable...] in <expression> [do]
    code
end
```

Voici deux exemples typique d'utilisation de boucles `'for'` : La première boucle itère de 1 à

```
for i in 1..5
    puts i
end

t = [3, 6, 2] for i in t
    puts i
end
```

5, `'1..5'` crée en fait un objet intervalle. On peut également utiliser 1...5 pour ne pas inclure le dernier élément, le 5 dans notre cas.

La deuxième boucle itère sur un tableau. Nous verrons au chapitre 5 que l'on peut quasiment faire la même chose à l'aide de blocs.

4.5.4 break

Termine une boucle while/until. Termine une méthode si `'break'` est appelé depuis le bloc associé.

4.5.5 next

Va au point situé juste avant l'évaluation de la condition de boucle. Si `'next'` est appelé dans un bloc alors celui-ci se termine. Correspond à l'instruction `'continue'` dans le langage *c*.

4.5.6 redo

Va au point situé juste après l'évaluation de la boucle (recommence l'itération en cours). Si `'redo'` est exécuté dans un bloc alors recommence celui-ci.

4.6 Modules

Afin d'obtenir une meilleure gestion des ensembles de classes, on peut définir des modules en Ruby. Un module crée une espace de noms dédié aux classes, aux méthodes ou aux constantes. Si lors de la définition d'un module un autre module de même nom existe déjà alors les composants des deux modules sont fusionné pour n'en former qu'un seul, cela permet de répartir un module sur plusieurs fichiers.

4.6.1 Modules de classes

Comme un module crée un espace de nom il est nécessaire de pre-fixer les classes que l'on utilise provenant de modules. Il est également possible d'importer toutes les classes du modules dans l'espace de nom actuel pour ne pas avoir besoin de pre-fixer les classes. Mais attention au conflit de noms !

Voici un exemple comprenant trois fichiers, un fichier contenant le module 'animaux : `'animaux.rb'` et deux fichiers de test du module.

```
module Animaux
    class Animal
        @nombreDePatte = 4
    end

    class AnimalDomestique < Animal
        def initialize(nom = '')
            self.DonnerNom(nom)
        end

        def DonnerNom(nom)
            @nom = nom
        end
    end

    class Chien < AnimalDomestique
        def aboyer
            puts 'ouah! ouah!'
        end
    end
end
```

le module : Animaux.rb

le test 1 : animaux_test1.rb

```
require 'Animaux'

monChien = Animaux::Chien::new('medor')

monChien.aboyer
```

```
require 'Animaux' include Animaux

monChien = Chien::new('medor')

monChien.aboyer
```

le test 2 : animaux_test2.rb Le test 1 préfixe la classe avec le nom du module alors que le test 2 «charge» les classes de `'Animaux'` dans l'espace de noms courant.

Comme une classe, le nom d'un module doit commencer par une majuscule.

4.6.2 Modules de méthodes

Une autre possibilité d'utilisation des modules est d'y mettre des méthodes. Ces méthodes vont pouvoir s'utiliser de deux manières. Soit comme des méthodes ne faisant pas parties d'un objet et s'utilisant directement en les pre-fixant avec le nom du module (ou en utilisant un include, mais attention aux collisions de noms), voici un exemple de la première solution :

```
module Maths
    def Maths::carre(n)
        n * n
    end
end

puts Maths::carre(4)
```

Ce n'est pas vraiment très utile et n'a pas grand chose avoir avec l'orienté objet

4.7 Gestion des erreurs

4.7.1 Rescue

La gestion des erreurs en Ruby se fait de manière très simple, voici la structure permettant de trapper une erreur :

```
begin
    code_douteux
[rescue [classe_exception[, classe_exception]...] [=> variable] [then]
    code]...
[else
    code]
[ensure
    code]
end
```

La ou les branches `'rescue'` permette de capturer une ou plusieurs exception. Si une erreur est levé alors c'est le code de la branche traitant cette exception qui est exécuté, si `'=> variable'` est présent alors l'objet exception y est stocké.

Si aucune exception n'est levée, le code de la branche `'else'` est exécutée. Le code de la branche `'ensure'` est dans tous les cas exécuté avant de sortir du begin/end.

4.7.2 Raise

l'instruction `'raise'` permet de propager volontairement une erreur. Elles peut s'utiliser de quatre manières différentes :

```
raise class_exception, message

raise objet_exception

raise message

raise
```

Si aucune classe n'est fournit alors `'RuntimError'` est pris par défaut.

4.8 Expressions rationnelles

Ruby, au même titre que Perl, est un très bon outil pour gérer les chaînes de caractères et notamment les expressions rationnelle. Il existe une classe en Ruby dédiée au expression rationnelle se nommant `'Regexp'` que nous ne voyons pas ici.

Une expression rationnelle ressemble à une chaîne mais au lieu d'être délimité par des guillemets ou des apostrophes elle se trouve entre deux slashs. L'opérateur de correspondance pour les expressions rationnelle est `'=~ '`.

Des variables implicites sont utilisées pour récupérer certaines informations après que l'on ait appliqué une correspondance entre une expressions rationnelle est une chaîne, en voici deux :

`$n ($1, $2, $3...)`
Contient la chaîne capturé par le nième groupe de la dernière recherche par motifs.

`$&`
Contient la chaîne capturée par la dernière recherche par motifs.

Exemple d'utilisation d'expressions rationnelles Ce code test la validité d'une adresse E-mail et capture le nom de domaine.

```
eMail = 'alfred.newman@madmagazine.com'

if /^(?:\w|\.|_|-)+@(\w+\.\w+)$/ =~ eMail
    puts "Ok, le domaine est : #{$1}";
end
```

Petites explications :

- Le `'^ '` correspond au début de la chaîne.
- La paire de parenthèses `'(? : )'` sert à grouper une expression sans la capturer pour lui appliquer ensuite le `'+'` qui signifie une repetition de une ou plusieurs fois le groupement.
- Le premier groupement est soit une lettre (représenté par l'ensemble `'\w'`), soit un point : `'\.'`, soit un underscore : `'_'`, soit un tiret : `'-'`.
- Au milieu de l'E-mail se trouve le `'@'`.
- Puis un groupement capture le nom de domaine.
- Le nom de domaine est composé d'une partie de une ou plusieurs lettre `'\w+'` puis d'un point `'\.'` puis encore de une ou plusieurs lettre `'\w+'` qui représente le `'.com'`, `'.fr'` etc.
- Finalement si l'E-mail est valide alors on affiche le nom de domaine capturé dans le premier groupe qui se trouve dans la variable `'$1'`.

5 Concept de bloc

Dans un langage orienté objet classique lorsque l'on appelle une méthode on ne peut que lui passer des objets (puisque tout est objet), la méthode qui réceptionne les paramètres doit, en général, connaître leur type pour pouvoir appeler les bonnes méthodes correspondantes au type. Ceci est vrai dans beaucoup de langage tel que Java, Python, etc.

Ruby Introduit une nouvelle notion, qui est celle de bloc. En plus de pouvoir passer toutes sortes de paramètres à une méthode, on peut également, pour autant qu'elle l'accepte, lui passer un bloc. Ce bloc n'est rien d'autre qu'un bout de code définissant un nouvel espace de nom comme dans une méthode.

Un bloc peut accepter des paramètres qui peuvent lui être passés depuis l'intérieur de la méthode qui l'utilise. Ces paramètres sont ensuite utilisables à à l'intérieur du bloc.

La grosse différence entre un bloc et un objet passé à une méthode est que le bloc n'impose pas une interface contrairement à l'objet. mais c'est la méthode que l'on appelle qui va dire comment elle va utiliser le bloc qu'elle va recevoir, c'est elle qui va lui imposer les paramètres qu'elle va lui transmettre.

5.1 Passage d'un bloc

Un bloc commence pas un `'do'` et fini par un `'end'`. Il existe aussi une version abrégée commençant par `'{'` et finissant par `'}'`, comme vous avez pu vous en apercevoir, ce document ce base uniquement sur la notation abrégée qui est plus simple et plus lisible.

Pour passer un bloc à une méthode il suffit de faire suivre l'appel de la méthode par le bloc. Voici la syntaxe :

```
objet.methode(paramètres...){
    <code du bloque>
}
```

5.2 Paramètre à un bloc

Un petit exemple vaut mieux qu'un grand discours :

```
objet.methode(paramètres...){|param1, param2...|
    <code du bloque>
}
```

Les paramètres d'un bloc se mettent entre deux pipes (`'|'`) et sont séparés par des virgules dans le cas ou il y a plusieurs paramètres.

5.3 L'instruction yield

L'instruction `'yield'` permet à la méthode réceptrice d'un bloc de l'appeler. Lors de l'appel d'un bloc il est possible de lui passer des paramètres comme ceci : `'yield(paramètres...)'`. Si trop de paramètres sont fournis alors ils seront simplement ignorés.

Un bloc peut, comme une méthode, renvoyer une valeur à la différence qu'il n'est pas permis d'utiliser le mot-clef `'return'` : Ce code affiche simplement `'hello'`, remarquez que l'instance de `'A'` n'est pas référencée.

Il est possible à l'intérieur d'une méthode susceptible de recevoir un bloc de tester si il en a effectivement reçut un à l'aide de la fonction `'bloc_given?'`.

```
class A
    def m
        puts yield
    end
end

A::new.m{ 'hello' }
```

5.4 Quelques exemples

5.4.1 Avec une classe propre

Exemple 1

Cet exemple reprends la classe `'Film'` de la section 3.1. Le but est de permettre d'accéder au titre de chaque film. Un accesseur en lecture à été ajouté sur le titre à la classe `'Film'` : `'attr_reader :titre'` (voir section 3.6).

Méthode ajoutée à la classe `'Videotheque'` :

```
def chaqueTitre
    @lesFilms.each{|film|
        yield(film.titre)
    }
end
```

Utilisation :

```
maVideotheque = Videotheque::new
maVideotheque.ajouterUnFilm(Film::new('LasVegas Parano', 111))
maVideotheque.ajouterUnFilm(Film::new('Requiem for a Dream', 97))
maVideotheque.ajouterUnFilm(Film::new('Fight Club', 132))

maVideotheque.chaqueTitre{|titre|
    puts ">> #{titre}"
}
```

Cet exemple d'illustration un peu simpliste pourrait, par exemple, être utiliser pour afficher les titres en les formatant avec des balises HTML en vue de les afficher dans un tableau sur une page web.

```
puts '<table>'
 maVideotheque.chaqueTitre{|titre|
    puts "<tr><td>#{titre}</td></tr>"
}
puts '</table>'
```

Exemple 2

Cet exemple illustre la sauvegarde de bloc. Le but est de créer un bouton personnalisé dérivant d'une hypothétique classe `'Button'` faisant partie d'une bibliothèque de fenêtrage.

le nom du dernier paramètre d'une méthode peut être pre-fixé par un `'&'` se qui signifie que le bloc deviendra un objet de type `'Proc'` et sera référencer par une variable.

On appel le bloc «contenu» dans un objet `'Proc'` à l'aide de la méthode `'call'`, il est possible de lui fournir un ou plusieurs paramètres si le bloc en demande.

```
class MonBouton < Button
  def initialize(label, &action)
    super(label)
    @action = action #référence le bloc
  end
  def presserBouton
    @action.call(self)
  end
end

Bouton1 = MonBouton::new('Activez moi !'){
    puts 'Hello, je suis le bouton 1'
}
```

Dans cet exemple le bloc donné au bouton n'est pas prévu pour recevoir de paramètre, l'instance de notre bouton (`'self'`) qui lui est passé lors de son appel est donc ignoré.

5.4.2 Avec la bibliothèque intégrée

Énormément de méthodes des classes des bibliothèques fournies en standard s'utilisent avec le passage d'un bloc, voici quelques exemples d'illustrations :

```
#affiche chaque pays
listePays = ['France', 'Suisse', 'Belgique']
listePays.each{|pays|
    Puts pays
}

#multiplie pas 2 chaque nombre du tableau
a = [4, 2, 4, 5, 6]
a.map{|n|
    2 * n
}

#efface les nombres paires de la liste
a = [6, 1, 3, 8, 4, 9]
a.delete_if{|n|
    n % 2 == 0
}

#ouvre un fichier et affiche chacune de ses lignes
IO::foreach('mon_fichier.txt'){|ligne|
    print ligne
}
```

6 Bibliothèque intégrée

Voici la présentation de quelques classes et modules intégrées à Ruby, pour en savoir plus je vous invite à vous rendre sur le site officiel : `http://www.rubycentral.com/ref/`.

6.1 La classe `Objet` et le module `Kernel`

La classe `Objet` est la mère de toutes les classes de Ruby, ses méthodes sont donc accessibles par n'importe quelle classe. Elle ne définit que la méthode `'initialize'` que n'importe classe peut redéfinir mais inclus le module `'Kernel'` et donc possède toutes ses méthodes.

Toutes les méthodes définies par `'Kernel'` sont accessible par toutes les classes, en voici un échantillon des plus utilisées :

o == autre
Test si deux valeurs sont égales.

o === autre
Compare l'égalité ou confirme l'appartenance de la classe.

o =~ autre
Test la correspondance d'une expression rationnelle avec une chaîne (voir section 4.8).

o.clone
Renvois une copie récursive de l'objet 'o'.

o.dup
Crée une copie de l'objet, en copiant ses valeurs.

o.inspect
Renvois une représentation de l'objet sous la forme d'une chaîne. La fonction `'p(o)'` affiche un objet en utilisant sa méthode `'inspect'`.

o.to_a
Renvois une représentation de l'objet sous la forme d'un tableau. Si l'objet ne peut pas être convertit alors renvois un tableau possédant comme unique élément l'objet.

o.to_s
Renvois une représentation de l'objet 'o' sous la forme d'une chaîne.

6.2 Manipulation de chaînes de caractères

Une chaîne de caractère est une instance de la classe `'String'`. Voici quelques méthodes de cette classe (il en existe beaucoup d'autre) :

Méthodes à l'échelle de classe

String : :new(chaîne)
Crée une nouvelle chaîne.

Méthodes membres

s + chaîne
Renvois une chaîne formée de `'s'` concaténé à `'chaîne'`.

s << chaîne
Concatène `'chaîne'` à `'s'`.

s[n]
Renvois le code du caractère situé à la position `'n'`. Si `'n'` est négatif, il est considéré comme un déplacement depuis la fin de la chaîne. Par exemple `s[-1]` représente le code du dernier caractère de `s`.

s[n..m]
s[n, longueur]
Renvois une sous chaîne de `'n'`.

```
'hello'[0..2]   # >> 'hel'
'hello'[2..-1]  # >> 'llo'
'hello'[-1..-1] # >> 'o'
'hello'[-3..-2] # >> 'll'
'hello'[1, 3]   # >> 'hel'
'hello'[-2, 2]  # >> 'lo'
```

s[n] = valeur
Remplace le caractère situé à la position `'n'` par `'valeur'`. `'n'` est soit un code caractère soit un chaîne.

s[n..m] = valeur
s[n, longueur] = valeur
Remplace une partie de la chaîne `'s'` par `'valeur'`.

s.length
Renvois le longueur de `'s'`.

s.to_f
Renvois la conversion de `'s'` en flottant.

s.to_i
Renvois la conversion de `'s'` en entier.

6.3 Les tableaux

Les tableaux sont très utiles, ils peuvent être utilisés comme tableau classique où chaque élément est référencé par un entier allant de 0 pour le premier à n - 1 pour le dernier où n est le nombre d'élément du tableau. Un tableau peut s'utiliser comme une pile ou encore comme une queue. Toutes ces «types» d'utilisations n'utilise qu'un seul type : `Array`.

Méthodes à l'échelle de classe

Array : :new([taille = 0[, remplissage = nil]])
Crée un nouveau tableau, on peut spécifier sa taille et ses valeurs initiales.

Méthodes membres

arr + tableau
Renvois un tableau résultant de la concatenation de `'arr'` et de `'tableau'`.

arr << élément
Ajoute `['élément']` à la fin de `['arr']`.

arr[n]
Désigne l'élément à la position `'n'`. Si `'n'` est négatif, il est considéré comme un déplacement depuis la fin de la chaîne.

arr[n..m]
arr[n, longueur]
Renvois une tranche de tableau.

arr[n] = élément
arr[n..m] = tableau
arr[n, longueur] = tableau
Affecte `'élément'` ou `'tableau'` aux éléments indiqués.

arr.clear
Supprime tous les éléments de `'arr'`.

arr.map{|x| ...}
arr.map !{|x| ...}
Invoque le bloc pour chaque élément de `'arr'` et renvois un tableau contenant les résultats. `'map !'` modifie directement `'arr'`.

arr.delete_at(n)
Supprime l'élément à la position `'n'` de `'arr'`.

arr.empty ?
Renvois `'true'` si `'arr'` est vide.

arr.join([s = $,])
Renvois une chaîne obtenue en joignant tous les éléments de `'arr'` et en les séparant par `'s'`.

```
['pourquoi', 'pas'].join         # >> 'pourquoipas'
['pourquoi', 'pas'].join(' ')    # >> 'pourquoi pas'
```

arr.length
Renvois le nombre d'élément de `'arr'`.

arr.pop
Supprime et renvois le dernier élément de `'arr'`.

arr.push(obj...)
Ajoute `'obj'` à la fin de `'arr'`.

arr.reverse
Inverse l'ordre des éléments de `'arr'`.

arr.shift
Supprime et renvois le premier élément de `'arr'`.

arr.sort
Trie un tableau.

arr.sort{|a, b| ...}
Trie un tableau en indiquant le condition de la comparaison à l'aide d'un bloc.

arr.unshift
Ajoute `'obj'` au début de `'arr'`.

6.4 Les dictionnaires

Les dictionnaire sont des structures contenant un nombre quelconque d'objet au même titre que les tableaux. Il diffère des tableaux dans le sens ou les éléments peuvent être référencés par n'importe quel objet possédant les deux méthodes : `'hash'` et `'eql?'`.

Les méthodes de la classe `'Hash'` ne sont pas listées ici, vous pouvez par contre les trouver à cette adresse : `http://www.rubycentral.com/ref/ref_c_hash.html`.

6.5 Manipulation de fichiers et de répertoires

En ruby il existe de nombreuses méthodes concernant l'accès aux fichiers et plus généralement concernant les entrées/sorties. Les classes concernées sont `'IO'`, `'File'`, `'File : :stat'`, `'Filetest'` et `'Dir'`.

Leurs méthodes sont décrites ici : `http://www.rubycentral.com/ref/`.

6.6 Processus léger (thread)

Il est très facile de créer des threads ou des groupes de thread en Ruby, ceci ce fait à l'aide des classes `'Thread'` et `'ThreadGroup'`.

Attention ! Des problèmes on été rencontrés lors de l'utilisation de threads sous Windows avec la version 1.6.8 de Ruby, nous conseillons l'utilisation de la version 1.7.3.

Voici un petit exemple d'utilisation :

```
# 1er façon
Thread::start{
    loop{
        puts 'hello'
        sleep(1)
    }
}

# 2ieme façon
class MonThread < Thread
    def initialize
        super{
            loop{
                puts 'salut'
                sleep(1.5)
            }
        }
    end
end

MonThread::new

sleep #bloque l'exécution de la partie principale
```

7 Bibliothèque standard

En plus de la bibliothèque intégrée vue au chapitre précédent la distribution de Ruby offre en standard un multitude de classes et de modules supplémentaires.

7.1 Exemple d'utilisation de classes réseau

Voici un petit exemple d'application client-serveur en Ruby, ceci est vraiment le minimum que l'on puisse faire.

Le serveur :
Le serveur va écouter sur le port '2000'. La méthode `'accept'` est bloquante jusqu'a l'arrivé d'une demande de nouvelle connexion. Lorsqu'une nouvelle connexion arrive `'accept'` renvois un objet de type `'TCPsocket'` et un thread est crée. Ce qui fait qu'un thread par client est crée.

```
require 'socket'

gs = TCPserver.open(2000)
puts "Le serveur est sur #{gs.addr.join(':')}"

while true
    Thread.start(gs.accept){|c|
        puts "#{c.addr.join(':')} est acceptée"
        while c.gets
            puts $_
        end
        puts "#{c.addr.join(':')} se termine"
    }
end
```

Le client :
Le client va lire les arguments envoyés au programme, si il y en a qu'un il le considère comme le port est définit l'hôte comme étant `'localhost'`. Si deux paramètres sont fournit au programme alors le premier est l'hôte et le deuxième le port.

Après que la connexion est été établit il va attendre qu'une ligne soit tapée suivit d'un 'return' (fin de ligne), lorsque cela se produit il va simplement écrire la ligne sur le socket pour la transmettre au serveur.

```
require 'socket'

hote = if ARGV.length == 2 then ARGV.shift else 'localhost' end
puts('connexion au serveur ...')
STDOUT.flush
s = TCPsocket.open(hote, ARGV.shift)
puts('connexion établie')
puts("adresse source   : #{s.addr.join(':')}")
puts("adresse distante : #{s.peeraddr.join(':')}")

while gets()
    s.write($_)
end

s.close
```

7.2 Motifs de conception (design patterns)

Il existe en Ruby des modules spécialement conçus pour réaliser certains motifs de conception. Ces motifs ne sont ici que listés est non étudiés.

- **Forwardable** : Ce module fournit une délégation de méthode plus explicite.
- **SingleForwardable** : Ce module fournit une délégation de méthode plus explicite pour un objet donné.
- **Singleton** : Ce module permet de vérifier qu'une seule instance d'une classe est crée au file d'un programme.
- **Observable** : Une classe incluant ce module permet d'être observer par un ou plusieurs objets.

8 Conclusion

J'espère que ce survole vous à permis de prendre conscience de la puissance et de l'élégance de Ruby et d'apprendre les principaux concepts de ce langage.

Il vous reste encore beaucoup à découvrir et je vous invite à rendre visite au site dédié à ce langage : `http://www.ruby-lang.org/` qui regroupe évidemment énormément de choses comme la documentation, les versions de Ruby téléchargable, un Wiki, etc. Vous pouvez également y trouver une section (la RAA : Ruby application archive) rassemblant une multitude de programmes et sources Ruby classés par type d'application.

9 Références bibliographiques

[1] Yukihiro Mastumoto. *Ruby in a nutshell*. O'Reilly, 2002.

[2] `http://www.ruby-lang.org/en/`.

[3] `http://www.rubycentral.com/book/`.