

# Apprenez Ruby

par Daniel Carrera

Date de publication : 05 Décembre 2006

Dernière mise à jour : 05 Décembre 2006

Ce cours présente le langage Ruby.

## Introduction

- 0-1 - Remerciements
- 1 - Les bases
  - 1-1 - Premiers pas
    - 1-1-A - Utiliser Ruby comme une calculatrice
    - 1-1-B - Les nombres et Ruby
      - 1-1-B-1 - Entiers
      - 1-1-B-2 - Décimaux
    - 1-1-C - Autres opérateurs arithmétiques
    - 1-1-D - Nombres énormes, et nombres minuscules
    - 1-1-E - Quelques exercices
  - 1-2 - Chaînes de caractères
    - 1-2-A - Opérateurs
    - 1-2-B - Méthodes
    - 1-2-C - Quelques exercices
  - 1-3 - De nouveaux noms pour de vieilles choses
    - 1-3-A - Classes et objets
    - 1-3-B - La notation Classe#méthode
    - 1-3-C - Convertir une classe
    - 1-3-D - Convertir une classe
  - 1-4 - Variables
    - 1-4-A - Travailler avec des variables
    - 1-4-B - Raccourcis
    - 1-4-C - Constantes
    - 1-4-D - Quelques exercices
  - 1-5 - Votre premier programme
    - 1-5-A - Bonjour monde
    - 1-5-B - Un autre exemple
    - 1-5-C - Créer des programmes exécutables
    - 1-5-D - Quelques exercices
  - 1-6 - Ecrire de bons programmes
    - 1-6-A - Noms de variables
      - 1-6-A-1 - Noms significatifs
      - 1-6-A-2 - Noms composés
    - 1-6-B - Travailler avec IRB
    - 1-6-C - Utiliser des constantes
- 2 - Contrôle de l'exécution
  - 2-1 - Boucles
    - 2-1-A - Compter
    - 2-1-B - Somme de nombres
    - 2-1-C - Étaler une expression sur plusieurs lignes
    - 2-1-D - Plus d'exemples
      - 2-1-D-1 - Compter à l'envers
      - 2-1-D-2 - Travailler sur une variable un certain nombre de fois
      - 2-1-D-3 - Quelques exercices
  - 2-2 - Saisir des données provenant de l'utilisateur
    - 2-2-A - Ce fichu «\n»!
    - 2-2-B - Retour à notre programme
    - 2-2-C - Quelques exercices
  - 2-3 - Conditions
    - 2-3-A - Vrai ou faux
    - 2-3-B - Opérateurs conditionnels
    - 2-3-C - Comparaison de chaînes de caractères
    - 2-3-D - L'instruction elsif

- 2-3-E - Un exemple: prix.rb
- 2-3-F - Quelques exercices
- 2-4 - Les boucles while
  - 2-4-A - Compteur
  - 2-4-B - Puissance de 2
  - 2-4-C - Quelques exercices
- 2-5 - Ecrire de bons programmes
  - 2-5-A - Commentaires
    - 2-5-A-1 - Comment utiliser des commentaires
  - 2-5-B - Indentation
- 3 - Structures de données
  - 3-1 - Tableaux : introduction
  - 3-2 - Que peuvent faire les tableaux?
    - 3-2-A - Array#sort
    - 3-2-B - Array#reverse
    - 3-2-C - Array#length
    - 3-2-D - Opérations arithmétiques sur des tableaux
    - 3-2-E - Afficher le contenu d'un tableau
    - 3-2-F - Quelques exercices
  - 3-3 - Itérateurs
    - 3-3-A - Quelques exercices
  - 3-4 - Hachages
    - 3-4-A - Quelques définitions : clef et valeur
    - 3-4-B - Hachages et itérateurs
      - 3-4-B-1 - Hash#each
      - 3-4-B-2 - Hash#each\_key
      - 3-4-B-3 - Hash#each\_value
  - 3-5 - Exemple pratique : un carnet d'adresses
    - 3-5-A - Première étape : analyse
    - 3-5-B - Deuxième étape : adresses
    - 3-5-C - Troisième étape : personnes
    - 3-5-D - Quatrième étape : carnet d'adresses
    - 3-5-E - Quelques exercices
  - 3-6 - Afficher le contenu du carnet
    - 3-6-A - Afficher des structures de données complexes
    - 3-6-B - Noms complets
    - 3-6-C - Numéros de téléphone
    - 3-6-D - Adresses
  - 3-7 - Trier les éléments du carnet
    - 3-7-A - Retour sur Array#sort
    - 3-7-B - Retourner des valeurs
    - 3-7-C - L'opérateur <=>
    - 3-7-D - Trier sur le nom de famille
    - 3-7-E - Trier le carnet d'adresses
    - 3-7-F - Quelques exercices
  - 3-8 - Ecrire de bons programmes
    - 3-8-A - Tableau ou hachage?
    - 3-8-B - Noms de variables
      - 3-8-B-1 - Tableaux
      - 3-8-B-2 - Hachages
    - 3-8-C - Commentaires
    - 3-8-D - Sous-structures
      - 3-8-D-1 - Bien
      - 3-8-D-2 - Peut mieux faire

- 3-8-D-3 - Pas bien
- 4 - Classes et méthodes
  - 4-1 - Fonctions
    - 4-1-A - Qu'est-ce qu'une fonction ?
    - 4-1-B - "Bonjour monde" avec une fonction
    - 4-1-C - Paramètres de fonction
    - 4-1-D - Afficher une adresse
  - 4-2 - Classes et méthodes
    - 4-2-A - Lire les données d'un objet
    - 4-2-B - Modifier les données d'un objet
    - 4-2-C - Accéder à des données
  - 4-3 - Plus de classes
    - 4-3-A - Une classe Personne
    - 4-3-B - Afficher le contenu d'une classe
    - 4-3-C - Quelques exercices
  - 4-4 - Implémentation du carnet d'adresses
    - 4-4-A - Première étape
    - 4-4-B - Tri automatique
      - 4-4-B-1 - Comment trier un tableau?
      - 4-4-B-2 - Simplification
      - 4-4-B-3 - Finalement
  - 4-5 - Ecrire des itérateurs
    - 4-5-A - Exécuter un bloc de code
    - 4-5-B - Passage de paramètres
    - 4-5-C - Implémentation de `Carnet#chaque_personne`
    - 4-5-D - Implémentation de `Carnet#chaque_adresse`
    - 4-5-E - Code complet de la classe `Carnet`
  - 4-6 - Autres fonctionnalités
    - 4-6-A - Méthodes publiques et privées
    - 4-6-B - Ré-utilisation du code avec `require`
  - 4-7 - Ecrire de bons programmes
    - 4-7-A - Fonctions et méthodes
    - 4-7-B - Commentaires
- A - Obtenir de l'aide
  - A-1 - Support technique
  - A-2 - Documentation
    - A-2-A - Autres tutoriels
    - A-2-B - Livres
- B - Installer Ruby sur votre machine
  - B-1 - GNU/Linux
    - B-1-A - Installation par RPM (Redhat, Mandrake...)
    - B-1-B - Gentoo
    - B-1-C - Debian
  - B-2 - Mac OS X
  - B-3 - FreeBSD
  - B-4 - Microsoft Windows

## Introduction

Ce tutoriel va vous permettre de vous familiariser avec le langage Ruby. Il est avant tout destiné aux débutants. Ces derniers pourront, sans posséder aucune connaissance au préalable, se servir de ce tutoriel comme tremplin pour leur long apprentissage du meilleur langage de programmation du monde!

Ce tutoriel est protégé par la **License de documentation libre GNU**. Ce qui signifie entre autre qu'il vous est permis de le redistribuer gratuitement, ainsi que de le modifier à votre guise (tout en respectant les conditions de la license).

N'hésitez pas à contacter notre liste de diffusion si vous avez des questions supplémentaires, des suggestions, des critiques... à propos de ce document. Déposez simplement un e-mail à l'adresse [ruby-tut-developer@nongnu.org](mailto:ruby-tut-developer@nongnu.org).

## 0-1 - Remerciements

Le traducteur tient à remercier les personnes suivantes, sans lesquelles l'élaboration de ce document n'aurait été possible :

- Eric Landuyt
- Laurent Mouillart
- Philippe Lesage
- Stéphanie Dijoux

## 1 - Les bases

### À propos de ce chapitre

Ce chapitre se focalise sur les bases fondamentales nécessaires afin que vous puissiez écrire votre premier programme en Ruby. Il est donc indispensable pour toute personne qui veut programmer en Ruby de lire ce chapitre.

### 1-1 - Premiers pas

Nous allons commencer notre exploration du langage Ruby en utilisant l'invite de commandes interactif IRB (en anglais, «interactive ruby shell»). Pour se faire, ouvrez un terminal et entrez :

```
irb --simple-prompt
```

 Vérifiez que le programme IRB démarre correctement avant de continuer.

### 1-1-A - Utiliser Ruby comme une calculatrice

L'interpréteur interactif IRB peut s'utiliser facilement comme calculatrice :

Ruby est capable d'évaluer toutes les opérations arithmétiques de base :

Symbole	Effet
+	addition
-	soustraction
*	multiplication
/	division

Pour quitter le programme IRB, il suffit d'entrer exit :

Nous allons maintenant jouer un peu plus avec les opérations arithmétiques de base. Essayez ceci :

Remarquez bien le comportement de Ruby lorsque nous essayons de diviser 3 par 2 :

Que s'est-il passé ? Rien d'anormal, je vous rassure! Ruby nous montre simplement qu'il travaille avec deux classes différentes de nombres :

- Entiers
- Décimaux (appelés aussi « flottants »)

### 1-1-B - Les nombres et Ruby

## 1-1-B-1 - Entiers

Un entier est un nombre comme 1, 2, -5, etc... Lorsque nous travaillerons avec des entiers, Ruby nous donnera toujours une réponse mathématiquement exprimée sous forme entière.

3 / 2 devrait nous donner 1.5, mais comme cette valeur n'est pas entière et que les deux nombres utilisés dans l'opération sont entiers, Ruby renverra donc à la place une réponse formulée sous forme entière, c'est à dire 1.

## 1-1-B-2 - Décimaux

Un nombre décimal est souvent appelé «flottant», par analogie avec sa partie décimale. 3.14, 1.5 et 3.0 sont des flottants. Ruby renverra toujours une valeur flottante comme résultat pour des opérations arithmétiques agissant sur des flottants. Par exemple :

## 1-1-C - Autres opérateurs arithmétiques

Avant d'en finir avec ce chapitre, jetons un petit coup d'oeil à deux autres opérateurs arithmétiques :

Symbole	Effet
**	exposant
%	modulo (le reste d'une division)

Notez bien le comportement de l'opérateur % avec les nombres décimaux. Dans cet exemple, 2 rentre deux fois dans 5.1, et il reste 1.1 de trop.

## 1-1-D - Nombres énormes, et nombres minuscules

Ruby convient très bien lorsqu'il s'agit de travailler avec de très grands et de très petits nombres. Supposez par exemple que vous souhaitez enregistrer quelque part le nombre 192349562563447.

Bon, c'est loin d'être facile à lire. Dans la vie courante, nous sommes habitués à séparer les chiffres par des caractères spéciaux, en fonction du pays d'origine (il s'agit souvent d'une virgule, d'un point ou d'un espace blanc). Par exemple, nos amis anglo-saxons auraient représenté ce fameux nombre de cette façon : 192,349,562,563,447. Ruby fonctionne exactement de la même façon, en utilisant des «blancs soulignés» (en anglais, «underscores») :

Comment faire si vous désirez travailler avec des nombres énormes comme 17\_000\_000\_000\_000\_000\_000, ou des nombres minuscules comme 0.000\_000\_000\_000\_321? En temps normal, la méthode la plus simple (et la plus paresseuse) consiste à utiliser une notation scientifique en écrivant 1.7 x 10<sup>19</sup>, et 3.21 x 10<sup>-13</sup>. Et comme vous vous en seriez douté, Ruby permet évidemment d'utiliser cette notation :

## 1-1-E - Quelques exercices

- i Calculez le nombre d'heures contenues dans année.
- ii Calculez le nombre de minutes dans une décennie.
- iii Quel est votre age en secondes?

- iv Que vaut l'expression  $3.24 * ((34/2) - 54)/33.4 * 3.4$  ? (notez qu'il est possible d'utiliser des parenthèses)
- v Quel sera le résultat d'une opération qui combine à la fois des flottants et des entiers? Par exemple, calculez ceci :  $3.0 / 2$  ;  $3 / 2.0$  ;  $4 ** 2.0$  ;  $4.1 \% 2$

La réponse sera-t-elle donnée sous forme entière ou décimale?

## 1-2 - Chaînes de caractères

Les nombres ne sont pas les seuls types de données que vous utiliserez pour programmer avec Ruby. En effet, vous allez probablement devoir manipuler des *lettres*, des *mots* ou encore des blocs de *texte*.

Ruby identifie ce genre de données par le terme chaîne de caractères (en anglais, «string»). Voici quelques exemples de chaînes

- "a"
- "Salut."
- "Longue vie à Ruby!!!"
- "5 est mon nombre préféré. Quel est le vôtre?"
- "Snoopy a dit: #%^?\*@!"

### 1-2-A - Opérateurs

Voici quelques trucs amusants que Ruby vous permet de faire avec des chaînes de caractères :

Voici un moyen de s'en souvenir : "salut " \* 3 fait la même chose que "salut " + "salut " + "salut " , ce qui nous donnera donc "salut salut salut " .

Remarquez ce qu'il se passe lorsque nous retirons l'espace blanc de "salut " :

Maintenant, regardez ceci :

Ruby considère "1" comme étant une chaîne de caractères, pas un nombre.

### 1-2-B - Méthodes

Voici quelques méthodes utiles qui travaillent avec des chaînes de caractères :

### 1-2-C - Quelques exercices

Quel sera le résultat de l'expression suivante : 'Bonjour '.length + 'monde!'.length ?

## 1-3 - De nouveaux noms pour de vieilles choses

Ruby identifie par des noms spéciaux certaines choses que nous connaissons déjà. Par exemple, il utilise le mot Float («flottant») pour représenter un nombre décimal. Voici quelques autres définitions :

- Object («objet») : Ce n'est rien d'autre qu'un bout de données. Par exemple, le nombre 3 ou la chaîne de caractères "Salut!" sont des objets.
- Class («classe») : Ruby sépare toujours les types de données en classes. Par exemple, les entiers, les flottants et les chaînes de caractères sont représentés par des classes différentes.
- Method («méthode») : Les méthodes sont des bouts de code qui peuvent travailler sur des objets. Par exemple, l'opérateur + (qui permet d'additionner deux objets, souvent des entiers ou des chaînes) est bel et bien une méthode.

Voici la liste des classes Ruby associées aux types de données que vous venez de découvrir :

Type de donnée	Classe associée
nombre entier	Integer
nombre décimal	Float
texte	String

Souvenez-vous, vous avez également utilisé plusieurs méthodes associées à ces classes :

Classe	Quelques méthodes
Integer	+, -, /, *, %, **
Float	+, -, /, *, %, **
String	capitalize, reverse, length, upcase, +, *

### 1-3-A - Classes et objets

Soyez certain de bien comprendre la différence entre une classe et un objet. Un objet, c'est une donnée quelconque. Une classe, c'est la structure de l'objet, ce à quoi il ressemble, ce qui le différencie des autres objets.

Par exemple, 3 et 5 sont des nombres différents. Ce sont donc des objets différents. Mais ce sont tous deux des nombres entiers, donc ils appartiennent à la même classe («Integer»). Voici quelques exemples :

Objet	Classe
2, -5	Integer
7.2, 3.14	Float
"Vive", "Ruby"	String

### 1-3-B - La notation Classe#méthode

Souvenez-vous, différentes classes peuvent avoir différentes méthodes. Voici quelques différences que nous avons déjà remarqué dans ce chapitre :

- L'opérateur de division (/) ne fonctionne pas de la même façon avec des entiers et des flottants.
- L'opérateur d'addition (+) se comporte différemment avec des nombres et des chaînes de caractères. Il peut servir à calculer le résultat de l'addition de deux nombres, et il peut également concaténer deux chaînes de caractères.
- Les chaînes de caractères possèdent plusieurs méthodes que les entiers et les flottants n'ont pas. Par exemple : capitalize, length, upcase, etc..

Pour cette raison, nous utiliserons la notation Classe#méthode pour identifier exactement de quelle méthode nous parlerons. Par exemple, nous dirons Integer#+ afin de différencier cette méthode de Float#+ et de String#+. Nous pouvons également affirmer que String#upcase existe, mais que Integer#upcase n'existe pas.

### 1-3-C - Convertir une classe

Voici quelques méthodes permettant d'effectuer des conversions pour les trois classes que nous venons d'étudier :

Méthode	Classe de départ	Classe d'arrivée
String#to_i	Chaîne de caractères	Entier
String#to_f	Chaîne de caractères	Flottant
Float#to_i	Flottant	Entier
Float#to_s	Flottant	Chaîne de caractères
Integer#to_f	Entier	Flottant
Integer#to_s	Entier	Chaîne de caractères

Un petit exemple pratique avec notre ami IRB :

### 1-3-D - Convertir une classe

Vous pouvez demander à Ruby si un objet est issu d'une classe particulière. Essayez ceci dans l'interpréteur :

```
12.is_a?(Integer)
12.is_a?(Float)
12.is_a?(String)

'12'.is_a?(Integer)
'12'.is_a?(Float)
'12'.is_a?(String)

12.0.is_a?(Integer)
12.0.is_a?(Float)
12.0.is_a?(String)
```

Essayez également ceci :

```
12 + 12
'12' + '12'

'12'.to_i + 12
'12' + 12.to_s

12 * 12
'12' * 12
```

Avez-vous obtenu les résultats que vous espériez?

Comment expliqueriez-vous la différence entre 12, '12' et 12.0 à un enfant?

## 1-4 - Variables

Ruby vous permet d'utiliser des variables pour associer des noms à des objets particuliers. Exemple

```
ville = "Toronto"
```

Ici, Ruby associe la chaîne de caractères "Toronto" à la variable ville.

Si vous avez des difficultés, imaginez-vous Ruby fabriquant deux tableaux. Un contenant les objets, et un autre contenant les noms qui leurs sont associés. Ensuite, imaginez-vous Ruby dessinant une flèche de ville à "Toronto".

Lorsque Ruby rencontrera la variable ville, il suivra logiquement la flèche et arrivera sur la chaîne de caractères "Toronto".

 *Les noms de variables doivent toujours commencer par une minuscule!*

### 1-4-A - Travailler avec des variables

Vous pouvez manipuler des variables exactement de la même façon que vous auriez manipulé les objets qu'elles représentent. Voici un exemple:

L'avantage des variables, c'est que vous pouvez garder quelque part une trace de vos données facilement. Imaginez par exemple que l'on vous donne les instructions suivantes :

- 1 Additionnez ensemble 2, 4, 6 et 8
- 2 Prenez le résultat, et divisez-le par 5
- 3 Calculez le produit simultané de 2, 3 et 4
- 4 Prenez le résultat obtenu à la ligne 2, et soustrayez-y ce que vous venez d'obtenir à la ligne 3

Bien entendu, vous pouvez écrire une longue expression pour calculer ceci. Mais il est évidemment plus facile d'écrire :

### 1-4-B - Raccourcis

Dans l'exemple ci-dessus, nous avons vu les expressions :

```
x = x / 5  
y = y - x
```

On rencontre assez souvent ce genre d'expressions, donc Ruby (qui pense à nous) nous permet d'utiliser les raccourcis suivants :

Exemple	Raccourci	Effet
$x = x + 2$	$x += 2$	Additionner 2 à x
$x = x - 3$	$x -= 3$	


		Soustraire 3 de x
<code>x = x * 6</code>	<code>x *= 6</code>	Multiplier x par 6
<code>x = x / 2</code>	<code>x /= 2</code>	Diviser x par 2
<code>x = x**3</code>	<code>x **= 3</code>	Exposer x au cube
<code>x = x % 4</code>	<code>x %= 4</code>	Calculer le reste de la division de x par 4, et y enregistrer le résultat

Donc, notre exemple du dessus aurait pu être écrit de cette façon :

### 1-4-C - Constantes

Les constantes ressemblent à des variables, à la différence près que vous informez Ruby que sa valeur est supposée ne pas changer. Si vous essayez de modifier la valeur d'une constante, Ruby vous enverra un avertissement.

Vous pouvez définir des constantes de la même façon que des variables, sauf que leur nom doit commencer par une majuscule.

 *Même si `VILLE` est une constante, sa valeur change quand même. Définir une constante signifie seulement que Ruby vous avertira si vous modifiez sa valeur. Regardez :*

### 1-4-D - Quelques exercices

Pensez-vous que les raccourcis fonctionnent également avec les chaînes de caractères? Essayez ceci :

```
var = "Bonjour "
var += "monde"
```

À votre avis, que va donner ce code :

```
var = "salut"
var *= 3
```

Essayez-le. Comment auriez-vous expliqué le résultat à un enfant?

## 1-5 - Votre premier programme

Félicitations! Vous êtes maintenant prêt à commencer à écrire des programmes en Ruby.

### 1-5-A - Bonjour monde

Ouvrez votre éditeur favori, et entrez-y la ligne suivante :

```
puts "Bonjour monde"
```

Sauvez le fichier sous le nom de `bonjour.rb` et démarrez-le en invoquant

```
ruby bonjour.rb
```

`puts` est une méthode qui affiche une chaîne de caractères sur le terminal.

Voici un nouvel exemple :

Regardez bien ce que nous avons fait. `nom` est une chaîne de caractères. Donc, nous pouvons la concaténer à d'autres chaînes de caractères, comme nous avons vu précédemment.

 Dans un programme Ruby, seules les lignes passées à `puts` seront affichées à l'écran.

## 1-5-B - Un autre exemple

Souvenez-vous, dans la section précédente, nous avons entré ceci dans IRB :

Nous allons créer un nouveau programme Ruby à partir de ce code. Copiez simplement les lignes dans un nouveau fichier, et rajoutez-y à la fin la ligne suivante :

```
puts y
```

Ensuite, sauvez le fichier et invoquez Ruby :

Ça fonctionne! Cependant, nous aimerions modifier un peu le message de sortie. L'idéal serait que le programme affiche «La réponse est 20» à l'écran. On peut toujours essayer ceci :

```
puts "La réponse est " + y # provoquera une erreur
```

Ça ne marchera évidemment pas. Pourquoi? Souvenez-vous, seules des chaînes peuvent-être concaténées à d'autres chaînes. Nous devons donc convertir notre entier `y` en chaîne de caractères. Nous savons qu'il faut utiliser la méthode `Integer#to_s`.

## 1-5-C - Créer des programmes exécutables

Si vous travaillez sur un système compatible UNIX (Linux, \*BSD, Solaris, Cygwin + Win32...), vous pouvez rendre vos programmes Ruby exécutables. Ils pourront donc être démarrés de la même façon que les autres programmes.

Premièrement, nous devons savoir où se cache Ruby sur votre machine. Pour ce faire, entrez la commande `which ruby` dans un terminal :

Recopiez le chemin sur la toute première ligne de votre programme. N'oubliez pas de préfixer la ligne par `#!`.

Il ne reste plus qu'à spécifier l'attribut du programme, ce que fera la commande `chmod +x prog.rb` (+x signifie «exécutable») :

## 1-5-D - Quelques exercices

Transformez en programme les exercices vus dans la section précédente.

Complétez le code suivant :

```
nom = "Laurent"  
age = 22
```

Le programme doit afficher "Laurent a 22 ans".

## 1-6 - Ecrire de bons programmes

Beaucoup trop de tutoriels vous enseignent simplement comment programmer, sans vous apprendre comment faire de bons programmes.

### 1-6-A - Noms de variables

Voici certaines choses dont vous devriez tenir compte lors du choix des noms de variables.

#### 1-6-A-1 - Noms significatifs

Le nom d'une variable doit permettre de deviner le type de l'information contenue dans l'objet. Voici quelques exemples :

Bien	Pas bien
age	a
etudiant	foo
nom	xwy
somme	truc


#### 1-6-A-2 - Noms composés

N'ayez surtout pas peur de choisir des noms de variables composés de plus d'un mot. Soyez simplement sûr que le nom de la variable est lisible.

Imaginez que vous désirez utiliser une variable pour enregistrer l'âge d'un étudiant. Il existe deux conventions :

- ageEtudiant
- age\_etudiant

Nous préférons la dernière possibilité. Mais c'est à vous de choisir.

 *N'écrivez pas ageetudiant. Vous n'auriez probablement pas apprécié si nous avions écrit ce tutoriel sans espacer les mots.*

## 1-6-B - Travailler avec IRB

Vous ne devez pas arrêter d'utiliser IRB maintenant que vous savez comment utiliser votre éditeur pour écrire des programmes Ruby. Nous vous avons montré IRB tout au début pour une bonne raison. Lorsque vous programmerez avec Ruby, vous devriez toujours garder un terminal ouvert avec IRB, et y retourner pour expérimenter vos idées.

IRB a été conçu dans cette optique d'utilisation. Si vous l'utilisez judicieusement, vous deviendrez un bon programmeur Ruby.

## 1-6-C - Utiliser des constantes


Utilisez toujours des constantes pour représenter des valeurs fixes dans vos programmes. Il s'agit d'une bonne précaution, car Ruby vous aidera à identifier des erreurs potentielles dans votre code.

Quelques exemples :

```
Pi = 3.14159265
Masse_electron = 9.109e-31
Vitesse_lumiere = 3e8
Distance_terre_soleil = 5.79e10
```

Essayez également d'éviter d'écrire directement des valeurs numériques. Utilisez des constantes pour rendre votre code plus lisible. Par exemple, la formule pour calculer l'aire d'un cercle est la suivante :

aire = pi . r<sup>2</sup>

 *r = rayon*

Votre code doit se rapprocher au maximum de la formule. Voici ce qu'il faut faire, et ce qu'il ne faut pas faire :

Bien	Pas bien
PI = 3.14159265 aire = PI * (rayon ** 2)	aire = 3.14159265 * (rayon ** 2)

## 2 - Contrôle de l'exécution

### À propos de ce chapitre

Ce chapitre aborde les concepts fondamentaux requis pour effectuer n'importe quelle tâche avec Ruby. Vous devrez probablement traiter des données provenant de l'utilisateur, ou exécuter une tâche plusieurs fois. Tout ceci sera introduit dans ce chapitre.

### 2-1 - Boucles

Dans cette section, nous allons vous initier à un des aspects les plus puissants de la programmation : les *boucles*.

Démarrez votre éditeur favori, et entrez-y les lignes suivantes :

Pouvez-vous deviner ce que fera ce bout de code? Sauvegardez le fichier sous le nom `boucles.rb` et exécutez-le :

Comme vous avez pu le constater, le contenu de la boucle a été exécuté 4 fois. Il s'agit probablement du moyen le plus direct que Ruby nous offre pour reproduire des tâches.

#### 2-1-A - Compter

Nous allons ici nous servir de ce que nous venons d'apprendre sur Ruby et les variables pour afficher les nombres compris entre 1 et 5 :

Souvenez vous, la méthode `Integer#to_s` se charge de convertir la valeur numérique représentée par l'objet en une chaîne de caractères. Nous avons besoin de cette méthode pour pouvoir concaténer le résultat à la chaîne `"count = "`.

Lors de l'exécution, vous devriez avoir ceci :

#### 2-1-B - Somme de nombres


Supposons que nous devons calculer la somme de tous les nombres compris entre 1 et 11. Nous venons juste de découvrir comment obtenir les nombres compris entre 1 et 11. Il nous reste simplement à les additionner :

Et voici ce qui s'affichera à l'écran :

#### 2-1-C - Étaler une expression sur plusieurs lignes


Dans le dernier exemple, l'expression puts devenait quand même longue. Que faire si nous devons afficher une ligne deux, trois, ou quatre fois plus longue?

Vous pouvez couper l'expression en plusieurs lignes, en terminant chaque ligne par le caractère «backslash» ('\').

 *Le caractère «backslash» doit absolument être le dernier caractère de la ligne. Si vous rajoutez un espace après, vous obtiendrez une erreur.*

Essayez ceci dans IRB :

La ligne "=>nil" signifie simplement que la méthode puts ne renvoie rien.

 *En fait, ce n'est pas tout à fait vrai. puts renvoie quand même une valeur qui représente un état non initialisé. Ruby représente cet état particulier par l'expression nil.*

Si nous avons écrit ceci :

```
etat = puts "Salut"
```

La variable etat aurait contenu nil.

Nous venons donc de voir comment étaler notre commande puts sur deux lignes. Mettons donc en pratique (dans notre programme) ce que nous venons d'apprendre :

En effet, nous pouvons utiliser cette technique pour afficher autant de lignes que nous désirons :

Vous n'êtes évidemment pas obligé de présenter la sorte du programme de cette façon. Cependant, nous pensons que l'utilisateur comprendra mieux le fonctionnement du programme. Voici ce que vous pouvez obtenir lors de son exécution :

## 2-1-D - Plus d'exemples

Nous allons ici vous montrer deux autres exemples à propos des boucles dans Ruby.

### 2-1-D-1 - Compter à l'envers


Et si nous essayions de compter à l'envers? Ce concept peut se révéler fort utile, imaginez que la Nasa vous demande d'écrire un programme de décomptage pour leurs fusées! Essayez de copier ceci dans un nouveau programme :

Et voici le résultat fort attendu (notez que nous ne sommes pas responsables de l'explosion de votre ordinateur) :

### 2-1-D-2 - Travailler sur une variable un certain nombre de fois

Évidemment, nous pouvons utiliser des boucles pour effectuer plusieurs fois une même opération sur une même variable.

Voici un exemple pratiquement valable. Il s'agit de calculer la *factorielle* d'un nombre.

 La factorielle d'un nombre est le produit de tous les nombres compris entre 1 et ce nombre. Vous me suivez? Par exemple, voici l'expression pour calculer la factorielle du nombre 6 :

```
6! = 6 * 5 * 4 * 3 * 2 * 1 = 720
```

Les mathématiciens utilisent le symbole  $n!$  pour représenter la factorielle du nombre  $n$

Tapez ceci dans un nouveau programme Ruby, et démarrez-le :

### 2-1-D-3 - Quelques exercices

- Que vaut la somme des entiers compris entre 1 et 1000?
- Que vaut la somme des entiers compris entre 10 et 100?
- Écrivez un programme qui affiche les paroles de la chanson :

```
1 kilomètre à pied, ça use, ça use  
1 kilomètre à pied, ça use, ça use les souliers  
  
La peinture à l'huile, c'est bien difficile  
Mais c'est bien plus beau  
Que la peinture à l'eau  
  
2 kilomètre à pied...
```

(arrêtez-vous à 100 kilomètres)

### 2-2 - Saisir des données provenant de l'utilisateur

Dans cette section, nous allons écrire un programme qui salue l'utilisateur. Le programme lui demandera son nom avant de le saluer. La méthode pour lire une chaîne de caractère venant de l'utilisateur s'appelle `gets`.

Sauvez le fichier et démarrez-le :


Que s'est-il passé? Pourquoi ce fichu programme va-t'il à la ligne?

La réponse est évidente : tout simplement parce que vous avez *entré* une nouvelle ligne au clavier. Oui, souvenez-vous, quand vous avez appuyé sur la touche **Enter**...

Nous allons regarder ceci de plus près dans IRB. Démarrez IRB et tapez l'expression `nom = gets`. L'ordinateur va attendre une saisie, entrez une chaîne quelconque. Regardez attentivement ce qu'il va se passer :

Que signifie donc ce «`\n`» à la fin de la chaîne de caractères?

Ce «\n» représente un *retour à la ligne*. Il s'agit du caractère envoyé par le clavier lorsque l'utilisateur appuie sur la touche Enter.

 *Avez-vous remarqué comment nous venons d'utiliser IRB pour comprendre ce qu'il s'était passé? Il arrivera tôt ou tard que votre code se comporte d'une façon anormale. A ce moment là, essayez-le dans IRB, vous comprendrez plus facilement la source de vos ennuis!*

## 2-2-A - Ce fichu «\n»!

OK, nous savons maintenant ce qui ne va pas. Mais comment résoudre ce problème?

Il existe précisément une méthode qui s'occupera de retirer le caractère «\n» d'une chaîne de caractères : `String#chomp`. Retournons dans IRB pour tester directement cette méthode :

Magnifique! La méthode `String#chomp` nous renvoie la chaîne de caractères sans le retour à la ligne. Nous pouvons donc maintenant écrire :

Notez bien que la variable `nom` contient toujours le saut à la ligne. La bonne variable (sans le «\n») est `nouveau_nom`. Mais pourquoi utiliser une autre variable? Pourquoi ne pas simplement écrire ceci :

```
>>nom = nom.chomp
```

De cette façon, nous retirons directement le saut à la ligne de la variable `nom`.

## 2-2-B - Retour à notre programme

Il est maintenant temps de corriger le problème dans notre code :

Une dernière petite réflexion :

- 1 `chomp` est une méthode de la classe `String` (souvenez-vous de la notation : `String#chomp`)
- 2 `gets` nous renvoie une chaîne de caractères (précisément, un objet de la classe `String!`)

Pourquoi ne pas écrire `gets.chomp`? De cette façon, `String#chomp` sera appelé sur ce que `gets` retournera. En d'autres termes, nous pouvons écrire :

N'est-ce pas joli? Mettez à jour votre code et invoquez Ruby :

## 2-2-C - Quelques exercices

- Écrivez un programme qui capture deux mots du clavier, et qui les imprime sur l'écran dans l'ordre opposé de leur saisie.

- Écrivez un programme dont le comportement devra ressembler à ceci :

Pour calculer l'année de naissance de l'utilisateur, effectuez une simple soustraction entre l'année courante et son âge.

- Écrivez un programme qui demandera à l'utilisateur un nombre ainsi qu'une chaîne de caractères. Ensuite, le programme doit afficher autant de fois la chaîne inversée. Exemple :

Il existe plusieurs solutions possibles à cet exercice. Choisissez celle que vous préférez.

- Devinez ce que produira Ruby avec cette ligne :

```
nombre = gets.chomp.to_i
```

Essayez-la dans IRB.

- Écrivez un programme qui calculera et affichera la factorielle d'un nombre entré au clavier par l'utilisateur. Il devra fonctionner comme ceci :

Vous pouvez réutiliser le code que nous avons écrit au chapitre précédent.

## 2-3 - Conditions

Dans Ruby, il est possible d'effectuer plusieurs actions en fonction d'une série de conditions, en utilisant le mot magique `if`. Voici un exemple :

```
#1
if ville == "Toronto"
  age_legal_alcool = 19
#2
else
  age_legal_alcool = 21
end
```

Ce qui veut dire :

- #1 Si (`if`) le contenu de la variable `ville` est égale à (`==`) la chaîne "Toronto", alors nous affectons à la variable `age_legal_alcool` le nombre 19.
- #2 Sinon (`else`), nous affectons à la variable `age_legal_alcool` le nombre 21.


### 2-3-A - Vrai ou faux

Ruby permet de distinguer deux états différents d'une expression :

- vrai (`true`)
- faux (`false`)

Le meilleur moyen d'illustrer cette théorie est d'expérimenter quelques exemples. Comme toujours, démarrez IRB et entrez les commandes suivantes :

L'instruction `if` évalue si une expression quelconque (par exemple, `ville == "Toronto"`) est soit vraie (`true`), soit fausse (`false`) et ensuite agit en conséquence.

 Notez bien la différence entre les opérateurs `=` et `==` :

- `=` est un opérateur d'affectation (il permet d'assigner une valeur à une variable)
- `==` est un opérateur de comparaison (il permet de comparer l'état de deux expressions)


## 2-3-B - Opérateurs conditionnels

Voici une liste regroupant la plupart des opérateurs conditionnels :


Opérateur	Effet
<code>==</code>	égal à
<code>!=</code>	n'est pas égal à
<code>&gt;</code>	est plus grand que
<code>&lt;</code>	est plus petit que
<code>&gt;=</code>	est plus grand ou égal à
<code>&lt;=</code>	est plus petit ou égal à

## 2-3-C - Comparaison de chaînes de caractères

Comment ces opérateurs se comportent-ils avec des chaînes de caractères? L'opérateur `==` vérifie si deux chaînes sont exactement identiques. Les autres opérateurs se basent sur la table ASCII pour produire et renvoyer un état de comparaison.


 *Qu'est-ce que cette table ASCII? Il s'agit d'une table contenant tous les caractères pouvant être générés par le clavier. Ces caractères sont triés dans cet ordre :*

- 1 d'abord les chiffres (012...789)
- 2 ensuite les lettres majuscules (ABC...XYZ)
- 3 et finalement les lettres minuscules (abc...xyz)

 *ASCII: « American Standard Code for Information Interchange ». Ce code permet d'échanger des informations entre des ordinateurs construits par différentes entreprises. Il a été conçu pour la langue anglaise à la base. Néanmoins, il ne convient pas pour les autres langues utilisant des caractères spéciaux (comme le japonais, par exemple). Pour remédier à ce problème, Ruby utilise un système d'encodage nommé UTF8, qui permet*

de combiner des caractères ASCII (codés sur un octet) et des symboles spéciaux (codés sur deux octets).

Maintenant, démarrez IRB et entrez ceci :

 Avez-vous remarqué comment IRB peut clarifier les choses? Vous devriez vous habituer à essayer d'abord certaines idées dans IRB. C'est un outil génial, utilisez-le !

## 2-3-D - L'instruction elsif

L'instruction elsif vous permet de rajouter plus d'une condition dans votre expression. Prenez ceci par exemple :

```
#1
if age >= 60
  puts "Prix senior"
#2
elsif age >= 14
  puts "Prix adulte"
#3
elsif age >= 2
  puts "Prix enfant"
#4
else
  puts "Gratuit!"
end
```

Essayons de comprendre ceci :


- #1 Si la variable age contient une valeur plus grande ou égale à 60, nous donnons un prix senior.
- #2 Si ce n'est pas vrai, mais que age est plus grand ou égal à 14, nous donnons un prix adulte.
- #3 Si ce n'est pas vrai, mais que age est plus grand que 2, nous donnons un prix enfant.
- #4 Sinon, c'est gratuit!

Ruby exécutera cette séquence de conditions une par une. Uniquement la première condition vraie sera exécutée. Vous pouvez rajouter autant de elsif que vous désirez.

## 2-3-E - Un exemple: prix.rb

Pour rendre les choses plus simple à comprendre, nous allons rajouter ces conditions dans un programme. Il devra demander l'age de l'utilisateur, et ensuite afficher le prix correspondant à cet age :

Ce programme devrait se comporter comme ceci :

 *Soyez bien conscient de l'ordre dans lequel vous rajoutez vos conditions avec elsif. Seulement le code associé à la première condition évaluée comme étant «vraie» sera exécuté. Cet exemple illustre bien ce danger :*

```
age = 21
```

```
if age >5
  puts "Plus vieux que 5 ans"
elsif age >10
  puts "Plus vieux que 10 ans"
elsif age >20
  puts "Plus vieux que 20 ans"
elsif age >30
  puts "Plus vieux que 30 ans"
end
```

En lisant ce code rapidement, il semblerait logique que le programme affiche "Plus vieux que 20 ans" à l'écran. Il n'en est rien! Comme dit plus haut, seule la première condition valide remporte la mise. Le programme affichera "Plus vieux que 5 ans".

Voici la bonne méthode à utiliser :

```
age = 21

if age >30
  puts "Plus vieux que 30 ans"
elsif age >20
  puts "Plus vieux que 20 ans"
elsif age >10
  puts "Plus vieux que 10 ans"
elsif age >5
  puts "Plus vieux que 5 ans"
end
```

## 2-3-F - Quelques exercices

- i Triez ces caractères en utilisant l'ordre ASCII : 2, u, A, 4, w, f, R, y
- ii Souvenez-vous, la table ASCII contient tous les caractères pouvant être générés par le clavier. Utilisez IRB pour déterminer où se trouve le caractère «?» (point d'interrogation) ?
  - Avant 0 ?
  - Après 9 mais avant A ?
  - Après Z mais avant a ?
  - Après z ?
- iii En se basant sur la question précédente, écrivez un programme Ruby qui accepte un caractère quelconque au clavier. Il devra afficher un des messages suivants :
  - x se situe avant 0
  - x se situe entre 9 et A
  - x se situe entre Z et a
  - x se situe après z

## 2-4 - Les boucles while

Maintenant que vous êtes familiers avec les conditions dans Ruby, il est temps de voir un autre type de boucle: la boucle while («tant que»).

Cette boucle est un peu plus intéressante que celle que vous venez de voir. Elle se base sur une condition :

```
while condition
  ...
end
```

```
end
```

Ou *conditionn* est rien d'autre qu'une expression conditionnelle, comme celles que nous venons d'appréhender précédemment.

## 2-4-A - Compteur

Voici un exemple tout simple, illustrant l'utilisation d'une boucle while :

Analysons le programme :

- 1 Initialisation de compteur à 0.
- 2 Comme compteur est bien inférieur à 10, nous rentrons dans la boucle.
- 3 Dans la boucle, nous affichons un message, et nous incrémentons compteur de 1. Maintenant, compteur vaut 1.
- 4 Comme compteur est toujours inférieur à 10, nous sautons à nouveau dans la boucle.
- 5 ...

Le scénario se répétera jusqu'à ce que compteur soit égal à 10. Voici la sortie du programme :

En d'autres mots, une boucle while répétera la boucle tant que la condition sera évaluée comme étant vraie.

## 2-4-B - Puissance de 2

Certaines choses sont faciles à réaliser avec une boucle while, mais moins évidentes avec *n.times*.

Supposez que nous voulons calculer la plus grande puissance de 2 plus petite que 10000. C'est très facile à programmer avec une boucle while :

Voici ce que ça donne :

Imaginez combien ça aurait été difficile d'arriver au même résultat, en utilisant *n.times*.

## 2-4-C - Quelques exercices

- Re-implémentez le dernier programme, pour qu'il demande à l'utilisateur une valeur maximale (à la place de 10000). Le programme devra se comporter de cette façon :
- Démarrez le même programme, en y entrant 1e10 comme entrée. Que se passe-t'il ?

Si vous avez utilisé la méthode `String#to_i`, il y a des chances pour que 1e10 soit converti par le nombre 1. Essayons de comprendre pourquoi dans IRB :

Comme vous pouvez le remarquer, `1e10` est en Float (nombre flottant). Donc, vous devez utiliser la méthode `String#to_s`.

## 2-5 - Ecrire de bons programmes

### 2-5-A - Commentaires

Un *commentaire* est une zone de texte dont chaque ligne est préfixée par le symbole `#`. Ces lignes seront ignorées par Ruby, donc vous pouvez les utiliser pour écrire des commentaires pour votre usage personnel. Par exemple :

```
# Calcule la plus grande puissance de 2
# inférieure à 10000.

nombre = 1
while nombre < 10_000
  nombre *= 2
end
```

Les commentaires sont un peu comme des notes personnelles. Simplement pour vous aider à vous souvenir de ce que vous avez écrit.

#### 2-5-A-1 - Comment utiliser des commentaires

- Écrivez des commentaires pour chaque étape de votre programme. Essayez de les rendre clairs et précis.
- Expliquez toujours ce que votre code fait, pas comment il le fait. Si le comment n'est pas évident en lisant le commentaire, alors vous devez réorganiser votre code.
- Dans le doute, il vaut mieux écrire plus de commentaires, que pas assez.

### 2-5-B - Indentation

L'*indentation* se résume à une mise en page réalisée à l'aide de tabulations ou d'espaces, afin de rendre un code source plus lisible.

Indentez *toujours* votre code. Il s'agit d'un des principes les plus fondamentaux, que vous devez assimiler si vous désirez produire de bons programmes.

#### Bien

```
# Additionne des nombres impairs
nombre, somme = 1, 0
while nombre < 100
  # % donne le reste
  if nombre % 2 == 1
    somme += nombre
  end
  puts somme
end
```


#### Pas bien

```
# Additionne des nombres impairs
```

```
nombre, somme = 1, 0
while nombre < 100
  # % donne le reste
  if nombre % 2 == 1
    somme += nombre
  end
  puts somme
end
```

Il existe plusieurs standards en matière d'indentation. En général, les programmeurs utilisent soit une tabulation, soit 2, 4 ou 8 espaces. Choisissez le style qui vous convient le mieux!

## 3 - Structures de données

 *La plus grande force de Ruby réside dans ses structures de données. Ce chapitre vous enseignera ces concepts pour écrire des programmes bien plus intéressants. Vous allez apprendre comment exprimer facilement des informations complexes, comme un carnet d'adresses complet. Ce chapitre est indispensable si vous désirez créer des programmes plus ou moins complexes.*


### 3-1 - Tableaux : introduction

Vous vous êtes déjà familiarisé avec deux classes Ruby, Integer et String. La classe Array représente une collection d'objets. Voici un exemple :

La méthode class nous indique que la variable nombres est un Array (plus précisément, un objet provenant de la classe Array). Vous pouvez accéder aux éléments du tableau de cette façon :

Vous pouvez rajouter des éléments dans le tableau tout simplement en tapant :

Remarquez que les éléments d'un tableau sont enregistrés séquentiellement. Un tableau peut contenir autant d'objets que vous le désirez. Les objets contenus dans le tableau peuvent être manipulés comme nous venons de voir :

 *Un tableau commence à compter à 0, pas 1! Ainsi, nombres[1] nous donnera le deuxième élément du vecteur, et non pas le premier.*

Quels types de choses pouvons-nous insérer dans un tableau? N'importe quel objet, en fait. Par exemple, des entiers et des chaînes de caractères :

Et pourquoi pas un autre tableau?

Vous comprenez pourquoi les tableaux sont vraiment intéressants?

### 3-2 - Que peuvent faire les tableaux?

Nous allons découvrir dans cette section quelques méthodes utiles concernant les tableaux en Ruby.

#### 3-2-A - Array#sort

Vous pouvez trier les éléments d'un tableau en utilisant la méthode Array#sort :

#### 3-2-B - Array#reverse


Inverser les éléments d'un tableau se fait tout aussi facilement :

### 3-2-C - Array#length

La méthode Array#length vous donne le nombre d'éléments d'un tableau :

### 3-2-D - Opérations arithmétiques sur des tableaux


Les méthodes Array#+, Array#- et Array#\* se comportent logiquement :

 *Il n'existe évidemment pas de méthode Array#/ , vu qu'il est impossible de diviser un tableau.*

Souvenez-vous des raccourcis +=, -= et \*=. Ils fonctionnent également avec des tableaux.

### 3-2-E - Afficher le contenu d'un tableau

Finalement, nous pouvons bien sûr afficher le contenu d'un tableau à l'écran, en utilisant l'instruction puts :

 *Souvenez-vous, nil signifie que puts ne renvoie rien.*

Notez également qu'il est possible de convertir un tableau en une chaîne de caractères, en invoquant la méthode Array#to\_s :


### 3-2-F - Quelques exercices

Selon vous, que fera ce bout de code?

```
>>adresses = [
  [ 17, "Boulevard de la Sauvenière" ],
  [ 2, "Place de la République Française"],
  [ 19, "Rue de la Renaissance" ]
]
>>adresses.sort
```

Et ceci?

```
>>adresses = [
  [ 20, "Rue de la Renaissance" ],
  [ 20, "Place de la République Française" ]
]
>>adresses.sort
```

 *N'oubliez pas votre ami IRB!*

### 3-3 - Itérateurs

Cette section illustre une des fonctionnalités les plus intéressantes de Ruby : les *itérateurs*.

Un itérateur est une méthode un peu particulière. Il s'agit d'une méthode vous permettant d'accéder *unpar unà* des éléments.

Les tableaux font partie des objets capables de supporter des itérateurs. Voici un exemple tout simple, utilisant la méthode `Array#each` :

```
amis = [ "Benjamin", "David", "Stéfanie", "Laura" ]
amis.each do |ami|
  puts "J'ai un ami qui s'appelle " + ami
end
```

Ce qui produira à l'écran :

```
J'ai un ami qui s'appelle Benjamin
J'ai un ami qui s'appelle David
J'ai un ami qui s'appelle Stéfanie
J'ai un ami qui s'appelle Laura
```

Le lecteur attentif aura peut-être remarqué que cette méthode d'itération ressemble fort à la toute première boucle illustrée dans ce tutoriel (celle avec la forme `n.times do ... end`). Il s'agit effectivement d'un itérateur! Cette méthode vous permet d'itérer sur la séquence d'entiers compris entre 0 et n-1. Regardez :

```
4.times do |nombre|
  puts nombre
end
```

Ce qui produira à l'écran :

```
0
1
2
3
```

Vous remarquerez que l'itérateur `times` commence à compter à 0. Un peu comme les indices des tableaux (0 représentant le premier élément). Donc, le code suivant :

```
amis = [ "Benjamin", "David", "Stéfanie", "Laura" ]
amis.each do |ami|
  puts "J'ai un ami qui s'appelle " + ami
end
```

Peut également être ré-écrit comme ceci :

```
# 'i' est une notation standard pour représenter un indice.
4.times do |i|
  puts "J'ai un ami qui s'appelle " + amis[i]
end
```

Maintenant, voici quelque chose d'amusant. Vous souvenez-vous de la méthode `Array#length`? Nous pouvons améliorer notre code en l'utilisant :

```
amis.length.times do |i| # 'i' pour indice
  puts "J'ai un ami qui s'appelle " + amis[i]
end
```

Essayons maintenant d'afficher uniquement les éléments du tableau dont l'indice est pair. Le moyen le plus facile de déterminer si un entier est pair est de vérifier si le reste de sa division avec 2 est bien égal à 0. Souvenez-vous, l'opérateur `Integer#%` donne le reste d'une division. Voici le code :

```
amis.length.times do |i| # 'i' pour indice
  # Nous n'affichons que les indices pairs
  if i % 2 == 0
    puts "J'ai un ami qui s'appelle " + amis[i]
  end
end
```

Ce qui produira :

```
J'ai un ami qui s'appelle Benjamin
J'ai un ami qui s'appelle Stéphanie
```

Comment faire pour afficher les amis dans l'ordre alphabétique? Rien de plus simple, il suffit d'utiliser la méthode `Array#sort` :

```
amis.sort.each do |ami|
  puts "J'ai un ami qui s'appelle " + ami
end
```

Ce qui produira :

```
J'ai un ami qui s'appelle Benjamin
J'ai un ami qui s'appelle David
J'ai un ami qui s'appelle Laura
J'ai un ami qui s'appelle Stéphanie
```

### 3-3-A - Quelques exercices

- Affichez le contenu du tableau `amis` dans l'ordre inverse alphabétique.
- Vous souvenez-vous de l'exercice du chapitre 2, où vous deviez afficher à l'écran les paroles de la chanson «Un kilomètre à pied, ça use...»? Ré-implémentez votre solution, en utilisant cette fois-ci un itérateur!
- Considérez le tableau suivant :

```
noms = [ "laurent", "david", "stéphanie", "laura" ]
```

Écrivez à l'écran la liste de ces noms, avec une majuscule comme première lettre. La méthode à utiliser est `String#capitalize`.

### 3-4 - Hachages

Nous allons étudier dans cette section la classe `Hash`.

Les hachages ne sont rien d'autre que des tableaux spécialisés. Au lieu de n'accepter que des indices représentés par des nombres entiers (comme dans `mon_tableau[3]`), les hachages acceptent n'importe quel objet comme «index». Par exemple, vous pouvez écrire `mon_hachage["une_chaine_de_caractères"]`.


Supposons que nous voulons enregistrer des informations concernant un ami. Nous pourrions utiliser un tableau, comme ceci :

```
ami = [ "Jean-Paul", "Goret", "Rue de l'église, 26", "Houtsiplou", "Liège" ]
```

Évidemment, ceci fonctionnera. Mais nous allons devoir nous souvenir que `ami[0]` pointe vers le prénom, `ami[1]` vers le nom de famille, et ainsi de suite. Ceci peut se révéler fort compliqué par la suite.

C'est exactement le type de problème qu'un hachage peut résoudre. Voici la définition d'un hachage en Ruby :

```
ami = {  
  "prénom"      => "Jean-Paul",  
  "nom de famille" => "Goret",  
  "adresse"     => "Rue de l'église, 26",  
  "ville"       => "Houtsiplou",  
  "province"    => "Liège"  
}
```

 *Remarquez bien qu'il faut utiliser des accolades pour créer un hachage. Les tableaux, quant à eux, se définissent par l'intermédiaire de crochets.*

Comme les tableaux, il est tout à fait possible de rajouter par la suite des champs à un hachage :

```
ami["pays"] = "Belgique"
```

### 3-4-A - Quelques définitions : clef et valeur

Comme les hachages n'utilisent pas que des indices numériques, on utilise le mot clef (en anglais : «key») pour les identifier. Un objet représenté à travers une clef est appelé valeur (en anglais : «value»).

Donc, dans l'exemple précédent, «prénom», «nom de famille» et «pays» sont des clefs. Leurs valeurs respectives sont «Jean-Paul», «Goret» et «Belgique».

### 3-4-B - Hachages et itérateurs

Évidemment, les hachages possèdent des itérateurs. En voici quelques un :

#### 3-4-B-1 - Hash#each

A l'instar des tableaux, les hachages ont également une méthode `each`. Cependant, elle nous donne à la fois la clef et la valeur de l'entrée. En voici un exemple :


```
amis.each do |clef, valeur|  
  puts clef + " => " + valeur  
end
```

Et voici son résultat à l'écran :

```
ville =>Houtsiplou
nom de famille =>Goret
pays =>Belgique
adresse =>Rue de l'église, 26
province =>Liège
prénom =>Jean-Paul
```

Ceci illustre un des plus gros défauts des hachages. Les données contenues dans un hachage ne sont pas classées dans un ordre particulier. Comment Ruby pourrait-il savoir que «prénom» doit venir avant «adresse»?

Le meilleur moyen de résoudre ce type de problème est de créer sa propre classe. Ruby est excellent pour ça. Mais avant d'étudier cette technique, il y a encore quelques notions que nous devons voir. Le prochain chapitre vous expliquera comment créer vos propres classes et méthodes.

 *Hash#each\_pair est un synonyme de Hash#each*

### 3-4-B-2 - Hash#each\_key

Le nom de cette méthode nous dit tout. Elle permet de parcourir toutes les clefs d'un hachage, un peu de la même façon que Hash#each :

```
>>ami.each_key do |clef|
?> puts clef
>>end
ville
nom de famille
pays
adresse
province
prénom
```

### 3-4-B-3 - Hash#each\_value

Cette méthode parcourt toutes les valeurs d'un hachage :

```
>>ami.each_value do |valeur|
?> puts valeur
>>end
Houtsiplou
Goret
Belgique
Rue de l'église, 26
Liège
Jean-Paul
```

## 3-5 - Exemple pratique : un carnet d'adresses

Dans cette section, nous allons construire un petit carnet d'adresses, contenant des informations sur trois amis : Nicolas, François et Marina.

La structure d'un carnet d'adresses est relativement complexe. Elle doit contenir plusieurs personnes, chacun ayant un nom, une adresse, et ainsi de suite.

Pour implémenter ce carnet d'adresses, notre stratégie sera de découper le problème en plusieurs petits problèmes.

### 3-5-A - Première étape : analyse

Il nous faut commencer par déterminer le type d'informations que notre carnet d'adresses devrait référencer :

- 1 Le carnet d'adresse doit contenir une liste de personnes. Nous devrions être capable de trier ces personnes par ordre alphabétique.
- 2 Chaque personne possède évidemment un prénom, un nom de famille, un numéro de téléphone ainsi qu'une adresse.
- 3 Chaque adresse est définie par le nom d'une rue, d'un code postal ainsi que d'une ville et le pays dans lequel elle se situe.

Nous allons commencer par définir une structure représentant une adresse, et nous finirons par assembler le carnet tout entier.

### 3-5-B - Deuxième étape : adresses

Nous avons ici deux possibilités pour implémenter la structure d'une adresse :

- 1 Tableau : nous pourrions ajouter successivement l'adresse, le code postal, la ville et le pays dans un tableau. Ça fonctionnerait.
- 2 Hachage : évidemment, c'est plus facile de se souvenir d'une expression comme `adresse["ville"]` plutôt que `adresse[2]`.

Nous allons évidemment choisir d'utiliser un hachage. Voici la représentation de nos adresses:

```
# Adresse de Nicolas
adresse_de_nicolas = {
  "rue"      => "Rue du port, 32",
  "code postal" => "56000",
  "ville"    => "Vannes",
  "pays"     => "France"
}

# Adresse de François
adresse_de_francois = {
  "rue"      => "Avenue de la tranchée, 14",
  "code postal" => "1000",
  "ville"    => "Bruxelles",
  "pays"     => "Belgique"
}

# Adresse de Marina
adresse_de_marina = {
  "rue"      => "Strada di l'amore, 61",
  "code postal" => "50100",
  "ville"    => "Firenze",
  "pays"     => "Italia"
}
```

### 3-5-C - Troisième étape : personnes

Chaque personne est identifiée par un prénom, un nom de famille, un numéro de téléphone ainsi qu'une adresse. Comme pour la structure précédente, un hachage convient parfaitement. Voici comment nous pouvons représenter nos amis avec Ruby :

```
# Nicolas
nicolas = {
  "prénom"      =>"Nicolas",
  "nom de famille" =>"Rocher",
  "téléphone"   =>"(+33) 02 93 45 49 19",
  "adresse"     =>adresse_de_nicolas
}

# François
francois = {
  "prénom"      =>"François",
  "nom de famille" =>"Willemart",
  "téléphone"   =>"(+32) 02 679 24 81",
  "adresse"     =>adresse_de_francois
}


# Marina
marina = {
  "prénom"      =>"Marina",
  "nom de famille" =>"Nantini",
  "téléphone"   =>"(+39) 055 681 32 11",
  "adresse"     =>adresse_de_marina
}
```

### 3-5-D - Quatrième étape : carnet d'adresses

Maintenant que nous avons défini toutes les autres structures, il ne nous reste plus qu'à créer le carnet d'adresses. Comme nous désirons pouvoir par la suite trier les éléments, nous devons donc utiliser un tableau pour réunir nos trois amis :

```
carnet = [ nicolas, francois, marina ]
```

Et voilà! Nous venons d'implémenter complètement la structure de notre carnet d'adresses. Dans la section suivante, nous allons apprendre comment trier les éléments du carnet, ainsi que d'autres petites choses amusantes.

 *Sauvegardez les structures dans un fichier. Nous allons les ré-utiliser dans les sections qui vont suivre.*

### 3-5-E - Quelques exercices

- i Ajoutez un de vos amis dans le carnet d'adresses.
- ii Modifiez la structure du carnet d'adresses afin qu'il puisse contenir des adresses e-mail.

### 3-6 - Afficher le contenu du carnet

Nous allons apprendre dans cette section comment afficher à l'écran le contenu de notre carnet d'adresses.

#### 3-6-A - Afficher des structures de données complexes

Nous pouvons toujours taper puts carnet, mais le résultat n'est pas très lisible pour un être humain (essayez dans IRB par vous-même). Nous voudrions mieux définir notre propre moyen pour afficher son contenu.

Comme carnet est un tableau, nous pouvons utiliser l'itérateur Array#each. Commençons par simplement afficher le prénom de nos amis :

```
carnet.each do |personne|  
  puts personne["prénom"]  
end
```

Ce qui affichera :

```
Nicolas  
François  
Marina
```

### 3-6-B - Noms complets

La prochaine étape est d'afficher les noms complets :

```
carnet.each do |personne|  
  prenom = personne["prénom"]  
  nom    = personne["nom de famille"]  
  puts prenom + " " + nom  
end
```

```
Nicolas Rocher  
François Willemart  
Marina Nantini
```

### 3-6-C - Numéros de téléphone

Rajouter le numéro de téléphone n'est pas plus difficile :

```
carnet.each do |personne|  
  prenom = personne["prénom"]  
  nom    = personne["nom de famille"]  
  tel    = personne["téléphone"]  
  puts prenom + " " + nom  
  puts "      " + tel  
end
```

Le résultat :

```
Nicolas Rocher  
  (+33) 02 93 45 49 19  
François Willemart  
  (+32) 02 679 24 81  
Marina Nantini  
  (+39) 055 681 32 11
```

### 3-6-D - Adresses

Finalement, il ne nous reste plus qu'à afficher les adresses. Voici le code qui devrait effectuer cette requête, en prenant soin d'espacer chaque personne par une nouvelle ligne :

```
carnet.each do |personne|  
  # Nom et téléphone  
  prenom = personne["prénom"]  
  nom    = personne["nom de famille"]  
  tel    = personne["téléphone"]
```

```
puts prenom + " " + nom
puts "      " + tel

# Adresse
rue   = personne["adresse"]["rue"]
cp    = personne["adresse"]["code postal"]
ville = personne["adresse"]["ville"]
pays  = personne["adresse"]["pays"]

puts "      " + rue
puts "      " + cp + ", " + ville
puts "      " + pays

# Une ligne vide pour séparer les entrées
puts ""

end
```

Ce qui produira :

```
Nicolas Rocher
(+33) 02 93 45 49 19
Rue du port, 32
56000, Vannes
France

François Willemart
(+32) 02 679 24 81
Avenue de la tranchée, 14
1000, Bruxelles
Belgique

Marina Nantini
(+39) 055 681 32 11
Strada di l'amore, 61
50100, Firenze
Italia
```

Et voilà le travail!

### 3-7 - Trier les éléments du carnet

Souvenez-vous, nous avons choisi d'utiliser un tableau pour contenir les différentes personnes de notre carnet d'adresses, parce que nous avons décidé au début qu'il devrait être possible de trier ces personnes, selon des critères quelconques.

Il est maintenant temps de trier les éléments de notre carnet.

#### 3-7-A - Retour sur Array#sort

Nous avons déjà utilisé Array#sort pour trier des tableaux contenant des nombres entiers, et des chaînes de caractères. Mais comment pouvons-nous trier des tableaux contenant des structures de données plus complexes (comme nos hachages)?

Lorsque le comportement par défaut n'est pas ce que vous désirez, Array#sort vous permet de spécifier comment trier le tableau. Commençons avec un exemple beaucoup plus facile que notre carnet d'adresses actuel :

```
amis = [
  [ "Nicolas", "Rocher" ],
```

```
[ "François", "Willemart" ],  
  [ "Marina", "Nantini" ]  
]
```

Nous savons que par défaut la méthode `Array#sort` appelée sur l'objet `amis` effectuera le tri en fonction du premier élément de chaque sous-tableau - dans ce cas, les prénoms. Et si nous voulions trier ces éléments par leur noms de famille?

À l'instar des itérateurs, la méthode `Array#sort` peut accepter un bloc de code. Comme ceci :

```
amis.sort do |a, b|  
  # ...  
end
```

La méthode `Array#sort` attend 3 comportements de ce bloc de code :

- Retourner la valeur -1 si la variable `a` est jugée plus petite que la variable `b`.
- Retourner la valeur 0 si la variable `a` est jugée égale à la variable `b`.
- Retourner la valeur 1 si la variable `a` est jugée plus grande que la variable `b`.

Grâce a cette information, la méthode `Array#sort` peut savoir exactement comment il faut trier le tableau.

### 3-7-B - Retourner des valeurs

Comment faire pour retourner une valeur? La valeur de retour est tout simplement la valeur de la dernière expression interprétée. Jetons un petit coup d'oeil dans IRB :

Les lignes => sont les valeurs de retour des expressions.

### 3-7-C - L'opérateur <=>

Trier étant une opération fort courante, il existe donc un opérateur spécialement dédié à ça. L'opérateur `<=>` retourne pour nous les fameuses valeurs -1, 0 et 1. Essayez ceci dans IRB :

### 3-7-D - Trier sur le nom de famille

Retournons à notre petit tableau :

```
amis = [  
  [ "Nicolas", "Rocher" ],  
  [ "François", "Willemart" ],  
  [ "Marina", "Nantini" ]  
]
```

Et à notre instruction de triage :

```
amis.sort do |a, b|  
  # ...  
end
```

a et b sont des éléments du tableau amis. Le comportement par défaut de `Array#sort` est équivalent à :

```
amis.sort do |a, b|
  a[0] <=>b[0] # Tri sur le premier élément
end
```

Mais comme nous voulons trier sur le *second* élément (c'est à dire le nom de famille), nous devons écrire :

```
amis = amis.sort do |a, b|
  a[1] <=>b[1] # Tri sur le second élément
end
# Maintenant, les éléments du tableau 'amis' sont triés sur le nom de famille.
```

### 3-7-E - Trier le carnet d'adresses

Maintenant que nous connaissons `Array#sort` un peu mieux, nous sommes prêts à personnaliser le tri du carnet d'adresses. Nous avons :

```
carnet.sort do |personne_a, personne_b|
  # ...
end
```

`personne_a` et `personne_b` sont tous deux des objets provenant du même hachage. Nous pouvons donc, par exemple, les trier par ordre alphabétique sur leur prénom, en écrivant tout simplement ceci :

```
carnet.sort do |personne_a, personne_b|
  personne_a["prénom"] <=>personne_b["prénom"]
end
```

Essayez, ça fonctionne!

### 3-7-F - Quelques exercices

Écrivez le code Ruby permettant de trier chaque élément du tableau carnet en fonction du nom complet (c'est à dire, le prénom et le nom de famille).

## 3-8 - Ecrire de bons programmes

### 3-8-A - Tableau ou hachage?

Quand faut-il utiliser un tableau? Quand faut-il utiliser un hachage?

La structure du carnet d'adresses est un bon exemple pour répondre à ces deux questions.

- Si vos données doivent-être triées, il vous faut un tableau.
- Si vos données représentent une collection de données différentes, sans aucune corrélation entre elles (par exemple, un prénom, un nom et un numéro de téléphone), il vous faut un hachage.
- Par contre, si les éléments appartiennent à la même catégorie de données (par exemple, une liste de noms), il vous faut un tableau.

Dans l'exemple du carnet d'adresses, les structures pour les adresses et les personnes contiennent des données différentes. Donc, nous les avons représentés par des hachages.

Mais le carnet lui-même contient des données du même type (des personnes), et nous voulions également être capable de trier ses éléments. Donc, représenter le carnet par un tableau était le meilleur choix.

## 3-8-B - Noms de variables

Comme vous êtes maintenant capable d'écrire des structures de données complexes, il devient plus important que vous choisissiez de bons noms pour vos variables.

### 3-8-B-1 - Tableaux

Un tableau devrait toujours représenter une collection d'éléments du *même type*. Comme une groupe de voitures, une liste de noms, etc.. Vous pouvez représenter ceci en utilisant en mettant au *plurielle* nom de votre tableau. Par exemple :

- Si chaque élément du tableau est une voiture, le tableau devrait s'appeler voitures.
- Si chaque élément du tableau est le nom d'une personne, le tableau devrait s'appeler noms.

De cette façon, le nom de la variable vous rappelle qu'il s'agit d'un tableau, et en même temps, elle se lit comme du français.

### 3-8-B-2 - Hachages

Lorsque vous travaillerez avec des hachages, il est important que vous choisissiez de bons noms pour vos clefs de hachage. Un bon nom doit être à la fois clair, descriptif et facile à se souvenir.

- Bien : "deuxième prénom"
- Pas bien : "2eprenom"

N'oubliez pas que Ruby accepte des espaces et des accents comme noms pour vos clefs de hachage. Vous n'avez donc aucune excuse d'utiliser "2eprenom" pour une clef.

## 3-8-C - Commentaires

N'oubliez surtout pas de rajouter des commentaires clairs dans vos programmes. Souvenez-vous, on écrit le code une fois, et on le lit une infinité de fois.

## 3-8-D - Sous-structures

En général, il est recommandé d'éviter de déclarer des sous-structures. Néanmoins, si vous utilisez une indentation propre et des noms significatifs pour vos variables, ça peut être bon.

Jetez un coup d'oeil à ces exemples :

### 3-8-D-1 - Bien

```
# Adresse de Nicolas
adresse_de_nicolas = {
  "rue"      =>"Rue du port, 32",
  "code postal" =>"56000",
  "ville"    =>"Vannes",
  "pays"     =>"France"
}

# Nicolas
nicolas = {
  "prénom"      =>"Nicolas",
  "nom de famille" =>"Rocher",
  "téléphone"   =>"(+33) 02 93 45 49 19",
  "adresse"     =>adresse_de_nicolas
}
```

### 3-8-D-2 - Peut mieux faire

```
# Nicolas
nicolas = {
  "prénom"      =>"Nicolas",
  "nom de famille" =>"Rocher",
  "téléphone"   =>"(+33) 02 93 45 49 19",
  "adresse"     =>{
    "rue"      =>"Rue du port, 32",
    "code postal" =>"56000",
    "ville"    =>"Vannes",
    "pays"     =>"France"
  }
}
```

### 3-8-D-3 - Pas bien

```
# Nicolas
nicolas = {
  "prénom" =>"Nicolas",
  "nom de famille" =>"Rocher",
  "téléphone" =>"(+33) 02 93 45 49 19",
  "adresse" =>{
    "rue" =>"Rue du port, 32",
    "code postal" =>"56000",
    "ville" =>"Vannes",
    "pays" =>"France"
  }}
}}
```

## 4 - Classes et méthodes

### A propos de ce chapitre

Vous venez d'apprendre comment créer vos propres structures de données. Dans ce dernier chapitre, nous passerons à la vitesse suivante, en transformant ces structures en classes (Adresse, Personne et Carnet), et en leur ajoutant des méthodes.

Ce seront de véritables classes Ruby, au même titre que Integer, String, Array ou Hash.

Créer ses propres classes procure quelques avantages intéressants :

- *Réutilisation du code* : Vous souvenez-vous du nombre de lignes qu'il a fallu écrire pour afficher le contenu du carnet d'adresses? N'aurait-il pas été plus joli d'écrire ce code quelque part, et d'ensuite appeler puts carnet chaque fois que nous désirions l'afficher ? C'est exactement ce que nous allons apprendre dans ce chapitre.
- *Abstraction des données* : Vous utilisez une classe en appelant ses méthodes. Le contenu de la classe en elle-même (l'implémentation des méthodes) vous est caché. Savez-vous comment la classe String est fabriquée? Probablement pas, mais vous êtes néanmoins capable de l'utiliser.

### 4-1 - Fonctions

#### 4-1-A - Qu'est-ce qu'une fonction ?

Une fonction est une méthode qui n'est pas associée à un objet en particulier. Vous avez déjà utilisé une fonction auparavant : puts. Remarquez bien la syntaxe :

```
puts "Salut!"      # à la place de : un_objet.puts "Salut!"
```

#### 4-1-B - "Bonjour monde" avec une fonction

Voici une fonction toute simple :

```
def bonjour
  puts "Bonjour monde!"
end
```

Maintenant, nous avons défini la fonction bonjour. Le code qu'elle contient sera exécuté à chaque appel de la fonction. Un exemple :

```
def bonjour
  puts "Bonjour monde!"
end

bonjour
bonjour
```

Qui produira :

```
Bonjour monde!
Bonjour monde!
```

Comme vous pouvez dès à présent le constater, les fonctions peuvent servir à réutiliser du code facilement.

## 4-1-C - Paramètres de fonction

Vous savez déjà qu'il est possible de passer des paramètres aux méthodes et fonctions. Mais vous ne savez pas encore comment faire! Voici la fonction `bonjour` légèrement améliorée :

```
def bonjour(nom)
  puts "Bonjour " + nom + ", comment vas-tu"
end

bonjour("Laurent")
bonjour("Stéphanie")
```

Ce qui produira :

```
Bonjour Laurent, comment vas-tu?
Bonjour Stéphanie, comment vas-tu ?
```

## 4-1-D - Afficher une adresse

Écrivons maintenant une fonction un peu plus utile. Souvenez-vous des structures d'adresses du chapitre précédent :

```
# Adresse de Nicolas
adresse_de_nicolas = {
  "rue"      => "Rue du port, 32",
  "code postal" => "56000",
  "ville"    => "Vannes",
  "pays"    => "France"
}

# Adresse de François
adresse_de_francois = {
  "rue"      => "Avenue de la tranchée, 14",
  "code postal" => "1000",
  "ville"    => "Bruxelles",
  "pays"    => "Belgique"
}
```

Voici le code d'une fonction qui permet de les afficher à l'écran :

```
def affiche_adresse(adresse)
  code_postal = adresse["code postal"]
  ville      = adresse["ville"]

  puts "    " + adresse["rue"]
  puts "    " + code_postal + ", " + ville
  puts "    " + adresse["pays"]
end
```

Maintenant, nous pouvons facilement afficher des adresses :

```
puts "Nicolas:"
affiche_adresse(adresse_de_nicolas)

puts "François:"
affiche_adresse(adresse_de_nicolas)
```

Ce qui produira à l'écran :

```
Nicolas:
  Rue du port, 32
  56000, Vannes
  France
François:
  Rue du port, 32
  56000, Vannes
  France
```

## 4-2 - Classes et méthodes

Nous sommes maintenant prêts pour créer notre propre classe Adresse. Commençons simplement par une classe qui ne contient que le champ "rue". Voici un exemple :

```
1
class Adresse
2
  def initialize(rue)
3
    @rue = rue
  end
end
```

Explication :

- 1 Le mot clef class définit une classe. On associe une méthode à une classe simplement en définissant la méthode à l'intérieur de la classe.
- 2 La méthode initialize permet de construire la classe. Il s'agit en fait du constructeur de la classe. Chaque classe doit contenir une méthode initialize!
- 3 @rue est une variable de classe. Ça ressemble un peu à une clef de hachage. Le symbole @ permet de distinguer les variables d'un objet. Chaque fois que vous allez instancier un objet de la classe Adresse, cet objet contiendra une variable @rue.

Utilisons maintenant cette classe pour créer un objet :

```
adresse = Adresse.new("Rue de la Renaissance, 49")
```

Et voilà le travail. adresse est dès à présent un objet provenant de la classe Adresse.

### 4-2-A - Lire les données d'un objet

Comment s'y prendre pour obtenir la rue de notre objet adresse? Nous pouvons par exemple écrire une méthode qui renvoie les données :

```
class Adresse
  def initialize(rue)
    @rue = rue
  end

  # Renvoie simplement @rue
  def rue
    @rue
  end
end
```

Maintenant, la méthode `Adresse#rue` vous permet d'obtenir la rue d'une adresse. Dans IRB :

```
>>adresse.rue
=>"Rue de la Renaissance, 49"
```

Une propriété d'un objet, visible de l'extérieur, est appelée attribut. Dans ce cas, `rue` est un attribut. Plus spécialement, un attribut en lecture, car il permet de lire la valeur d'une des propriétés de la classe.

Comme ce genre d'attribut se retrouve assez souvent, Ruby nous offre un raccourci : le mot clef `attr_reader` :

```
class Adresse
  attr_reader :rue
  def initialize(rue)
    @rue = rue
  end
end
```


## 4-2-B - Modifier les données d'un objet

Évidemment, il est également possible de modifier une propriété d'un objet. Par exemple, en rajoutant une méthode dans la classe :

```
class Adresse
  attr_reader :rue
  def initialize(rue)
    @rue = rue
  end
  def rue=(une_rue)
    @rue = une_rue
  end
end
```

Et voici comment utiliser cette nouvelle méthode :

```
adresse.rue = "Une autre adresse"
```

 *Remarquez que Ruby accepte des espaces entre `rue` et `=` (dans l'affectation). Par contre, il ne faut pas mettre d'espaces dans la définition de la méthode.*

Maintenant que nous savons comment modifier les données de notre classe, nous n'avons plus besoin d'initialiser la rue dans le constructeur. Nous pouvons donc simplifier la méthode `initialize` :

```
class Adresse
  attr_reader :rue
  def initialize
    @rue = ""
  end
  def rue=(une_rue)
    @rue = une_rue
  end
end

adresse = Adresse.new
adresse.rue = "Rue de la Renaissance, 49"
```

Cette petite modification rendra le code plus simple par la suite, lorsque nous rajouterons les autres données (le code postal, la ville, le pays).

Maintenant, `rue` est également un attribut en écriture, car via la méthode `Adresse#rue=`, il nous est possible de modifier sa valeur. Comme pour la lecture, il s'agit d'une opération courante, et Ruby nous permet d'utiliser le mot clef `attr_writer` :

```
class Adresse
  attr_reader :rue
  attr_writer :rue
  def initialize
    @rue = ""
  end
end
```


## 4-2-C - Accéder à des données

Fort souvent, vous aurez besoin d'attributs qui fonctionneront à la fois en lecture et en écriture. Comme vous vous en doutez peut-être, Ruby possède un mot clef qui regroupe ces deux états à la fois : `attr_accessor` :

```
class Adresse
  attr_accessor :rue
  def initialize
    @rue = ""
  end
end
```

Il est maintenant temps de définir entièrement la structure adresse de notre carnet sous la forme d'une classe. Le code ci-dessous ne devrait pas vous poser de problème :

```
class Adresse
  attr_accessor :rue, :code_postal, :ville, :pays
  def initialize
    @rue = @code_postal = @ville = @pays = ""
  end
end
```

 Notez que `attr_accessor` accepte plusieurs arguments.

## 4-3 - Plus de classes

### 4-3-A - Une classe Personne

Créons maintenant une classe `Personne`. Une personne doit avoir un prénom, un nom de famille, une adresse e-mail, un numéro de téléphone, et une adresse physique (celle que nous venons d'implémenter à la section précédente) :


```
class Personne
  attr_accessor :prenom, :nom, :email, :telephone, :adresse
  def initialize
    @prenom = @nom = @email = @telephone = ""
    @adresse = Adresse.new
  end
end
```

La seule chose qui pourrait vous surprendre dans cet exemple, c'est la ligne `@adresse = Adresse.new`. La variable `@adresse` n'est pas une chaîne de caractères comme les autres variables, mais un objet issu de la classe `Adresse`.

Il est maintenant possible de créer une personne :

```
# Adresse:
adresse_de_nicolas      = Adresse.new
adresse_de_nicolas.rue  = "Rue du port, 32"
adresse_de_nicolas.code_postal = "56000"
adresse_de_nicolas.ville = "Vannes"

# Personne:
nicolas                 = Personne.new
nicolas.prenom          = "Nicolas"
nicolas.nom              = "Rocher"
nicolas.email           = "nicolas.rocher@free.fr"
nicolas.adresse         = adresse_de_nicolas
```

 Notez que nous n'avons ni affecté de pays, ni de numéro de téléphone à Nicolas. Comme par défaut, toutes les valeurs sont vides (c'est à dire ""), nous pouvons créer des objets Adresse et Personne avec des informations incomplètes. Et nous pouvons toujours appeler `nicolas.telephone` sans aucun problème, Ruby nous renverra "" en retour. Essayez dans IRB!

Et si nous ajoutons une méthode dans la classe Personne, qui nous renverrait le nom complet de la personne représentée? Très facile :

```
class Personne
  attr_accessor :prenom, :nom, :email, :telephone, :adresse
  def initialize
    @prenom = @nom = @email = @telephone = ""
    @adresse = Adresse.new
  end
  def nom_complet
    @prenom + " " + @nom
  end
end

# ...

puts nicolas.nom_complet
```

Ce qui affichera "Nicolas Rocher".

## 4-3-B - Afficher le contenu d'une classe


Ne serait-il pas magnifique si nous pouvions simplement taper `puts adresse_de_nicolas`? Ruby est capable d'exaucer ce rêve.

La fonction `puts` travaille de la façon suivante : elle essaye d'appeler la méthode `to_s` sur l'objet qu'on lui donne, et elle affiche le résultat sur l'écran. Vous vous souvenez de `Integer#to_s` et de ses amis?

La seule chose à faire, c'est de définir une méthode `Adresse#to_s` :

```
class Adresse
  attr_accessor :rue, :code_postal, :ville, :pays
  def initialize
    @rue = @code_postal = @ville = @pays = ""
  end
  def to_s
    " " + @rue + "\n" + \
```

```
        "    " + @code_postal + ", " + @ville + "\n" \
        "    " + @pays
    end
end
```

 *Le caractère \n représente une nouvelle ligne. Il s'agit d'un fait du caractère que votre clavier envoie lorsque vous pressez la touche Enter.*

Nous pouvons maintenant taper ceci :

```
adresse           = Adresse.new
adresse.rue       = "Rue du port, 32"
adresse.code_postal = "56000"
adresse.ville     = "Vannes"
adresse.pays      = "France"

puts adresse
```

Ce qui affichera :

```
Rue du port, 32
56000, Vannes
France
```

### 4-3-C - Quelques exercices

Ecrivez une méthode `Personne#to_s`, qui devra afficher le nom complet de la personne, son adresse e-mail, son numéro de téléphone, et son adresse physique.

### 4-4 - Implémentation du carnet d'adresses

Maintenant que nous avons les classes `Adresse` et `Personne`, il ne nous reste plus qu'à créer la classe `Carnet`.

#### 4-4-A - Première étape

Notre carnet d'adresse doit contenir un tableau, qui contient tous nos contacts. Nous n'utiliserons pas `attr_accessor` parce que nous ne voulons pas qu'on puisse accéder directement au tableau. Nous allons donc écrire nos propres méthodes pour travailler sur le tableau interne.

Voici à quoi peut ressembler le code de notre classe `Carnet` :

```
class Carnet
  def initialize
    # Initialise le tableau. Meme chose que ``Array.new``.
    @personnes = []
  end
end
```

C'était plutôt facile. Rajoutons maintenant deux méthodes d'accès : `Carnet#ajoute` et `Carnet#retire` :

```
class Carnet
  def initialize
    @personnes = []
  end
end
```

```
def ajoute(personne)
  #1
  @personnes.push(personne)
end
def retire(personne)
  #2
  @personnes.delete(personne)
end
end
```

Explication du code :

- 1 : La méthode Array#push permet d'ajouter un objet dans un tableau. L'objet sera empilé sur les objets existants, un peu comme si vous rajoutiez une assiette sur une pile d'assiettes.
- 2 : La méthode Array#delete permet de supprimer un objet d'un tableau. Si le tableau contient plusieurs objets identiques, ils seront tous supprimés.

Pour mieux comprendre le fonctionnement de Array#delete, essayez ceci dans IRB :

```
>>a = [ 1, 3, 3, 3, 3, 5]
=>[1, 3, 3, 3, 3, 5]
>>a.delete(3)
=>3
>>a
=>[1, 5]
```

## 4-4-B - Tri automatique

Nous allons maintenant rajouter une super fonctionnalité dans notre Carnet : un tri automatique. Par exemple, imaginez le code suivant :

```
carnet = Carnet.new
carnet.ajoute nicolas
carnet.ajoute francois
carnet.ajoute marina
```

Le carnet classera automatiquement les personnes à chaque ajout. Cette fonctionnalité va rendre notre classe Carnet bien plus intéressante qu'un simple tableau.

### 4-4-B-1 - Comment trier un tableau?

Nous voulons que le tableau contenant les contacts soit trié par ordre alphabétique, en se basant sur le nom complet de la personne (prénom et nom de famille). Dans la section précédente, nous avons écrit ceci :

```
# ``carnet`` est un tableau ici
carnet.sort do |personne_a, personne_b|
  personne_a["prénom"] <=>personne_b["prénom"]
end
```

Adaptons ce code pour notre classe Carnet :

```
@personnes.sort do |a, b|
  a.prenom <=>b.prenom
end
```

Si vous avez effectué les exercices du chapitre précédent, vous devriez déjà savoir comment trier les personnes en se basant sur leur nom complet. Voici une façon de le faire :

```
@personnes.sort do |a, b|
  if a.prenom == b.prenom
    a.nom <=>b.nom
  else
    a.prenom <=>b.prenom
  end
end
```

Si les prénoms sont les mêmes, nous comparons les noms de famille. Sinon, on compare les prénoms.

#### 4-4-B-2 - Simplification

Le principe fondamental de la simplification est de diviser le problème en petites parties. Nous pouvons déplacer le bloc de code dans une méthode :

```
def par_nom(a, b)
  if a.prenom == b.prenom
    a.nom <=>b.nom
  else
    a.prenom <=>b.prenom
  end
end
```

Maintenant, nous pouvons écrire :

```
@personnes.sort do |a, b| par_nom(a, b) end
```

Ce qui est beaucoup plus simple à lire.

Il est possible de définir avec Ruby des blocs de code en utilisant deux syntaxes différentes :

```
@personnes.sort do |a, b|
  # ...
end
```

```
@personnes.sort { |a, b|
  # ...
}
```

Ces deux notations veulent dire exactement la même chose. La différence est que `do ... end` est plus lisible, et que `{ ... }` est plus court.

Nous pouvons écrire le tri de notre tableau de cette façon :

```
@personnes.sort { |a, b| par_nom(a, b) }
```

Vous pouvez littéralement lire "tri de personnes par le nom". C'est du code très lisible. Voici une suggestion :

- Utilisez la notation `{ ... }` quand il est possible de faire tenir l'expression sur une seule ligne.
- Sinon, utilisez `do ... end`.

### 4-4-B-3 - Finalement

Il est maintenant temps de déplacer ce code dans notre classe `Carnet`, et d'implémenter notre tri automatique :

```
class Carnet
  def initialize
    @personnes = []
  end
  def ajoute(personne)
    @personnes.push(personne)
    @personnes = @personnes.sort { |a, b| par_nom(a, b) }
  end
  def retire(personne)
    @personnes.delete(personne)
  end
  def par_nom(a, b)
    if a.prenom == b.prenom
      a.nom <=>b.nom
    else
      a.prenom <=>b.prenom
    end
  end
end
```

Maintenant, à chaque fois que vous ajouterez une personne dans le carnet, ce dernier sera trié automatiquement!

### 4-5 - Ecrire des itérateurs

Dans cette section, nous allons ajouter deux itérateurs dans la classe `Carnet` : `Carnet#chaque_personne` et `Carnet#chaque_adresse`. Au résultat, nous pourrions écrire ceci :

```
carnet.chaque_personne do |personne|
  # ...
end

carnet.chaque_adresse do |adresse|
  # ...
end
```

### 4-5-A - Exécuter un bloc de code

Le mot clef `yield` permet d'appeler un bloc de code. Voici un exemple :

```
def deux_fois
  yield
  yield
end

deux_fois { puts "Vive Ruby!" }
```

Ce qui produira :

```
Vive Ruby!
Vive Ruby!
```

### 4-5-B - Passage de paramètres

Vous pouvez utiliser `yield` exactement comme n'importe quelle autre méthode. Pour passer des arguments à un bloc de code, passez-les simplement à `yield`. Prenez cet exemple :

```
def noms
  yield("Nicolas")
  yield("François")
  yield("Marina")
end

noms do |nom|
  puts "Salut " + nom + ", comment vas-tu?"
end
```

Ce qui affichera à l'écran :

```
Salut Nicolas, comment vas-tu?
Salut François, comment vas-tu?
Salut Marina, comment vas-tu?
```

Vous pouvez passer autant de paramètres que vous voulez au bloc de code. Par exemple :

```
def noms
  yield("Nicolas", "Rocher")
end

noms do |prenom, nom|
  puts prenom + " " + nom
end
```

Ce qui donnera :

```
Nicolas Rocher
```

#### 4-5-C - Implémentation de `Carnet#chaque_personne`

Ce premier itérateur est le plus facile des deux à écrire : il suffit simplement de parcourir chaque personne dans le tableau `@personnes` et d'appeler `yield` sur chaque élément :

```
class Carnet
  # ...
  def chaque_personne
    @personnes.each { |p| yield(p) }
  end
end
```

Et voilà!

#### 4-5-D - Implémentation de `Carnet#chaque_adresse`

Cet itérateur est quasi aussi simple à écrire que le premier. Au lieu de passer chaque personne au bloc de code, nous allons passer l'adresse de cette personne :

```
class Carnet
  # ...
  def chaque_adresse
    @personnes.each { |p| yield(p.adresse) }
  end
end
```

```
end  
end
```

## 4-5-E - Code complet de la classe Carnet

Juste pour tout mettre au clair, voici le code complet commenté de la classe Carnet. Il s'agit d'un morceau de code assez complexe, mais en découpant les tâches au fur et à mesure, il est beaucoup plus facile à maintenir :

```
class Carnet  
  #  
  # Méthodes fondamentales:  
  #   initialize  
  #   ajoute  
  #   retire  
  #  
  def initialize  
    @personnes = []  
  end  
  def ajoute(personne)  
    @personnes.push(personne)  
    @personnes = @personnes.sort { |a, b| par_nom(a, b) }  
  end  
  def remove(personne)  
    @personnes.delete(personne)  
  end  
  
  #  
  # Iterateurs:  
  #   chaque_personne  
  #   chaque_adresse  
  #  
  def chaque_personne  
    @personnes.each { |p| yield p }  
  end  
  def chaque_adresse  
    @personnes.each { |p| yield p.adresse }  
  end  
  
  #  
  # Fonction de tri.  
  #  
  def par_nom(a, b)  
    if a.prenom == b.prenom  
      a.nom <=>b.nom  
    else  
      a.prenom <=>b.prenom  
    end  
  end  
end  
end
```

## 4-6 - Autres fonctionnalités

Avant de terminer, nous pouvons encore voir deux petites choses intéressantes concernant notre classe Carnet.

### 4-6-A - Méthodes publiques et privées

Prenez `Carnet#par_nom`. Cette méthode est différente des autres sur un point très important : elle est utilisée à l'intérieur même de la classe. C'est ce qu'on appelle une méthode interne, ou privée.

Lorsque vous programmerez une méthode comme celle-ci, il est possible de la déclarer comme étant privée. Une méthode privée ne peut être appelée que par l'objet lui-même, et jamais par l'utilisateur. Une méthode normale (comme toutes celles que nous avons écrites), est appelée méthode publique.

Vous pouvez déclarer des méthodes en utilisant les mots clefs `public` et `private`. Quand vous ajoutez le mot clef `private`, toutes les méthodes définies à partir de là seront privées, jusqu'à ce que vous rajoutiez le mot clef `public`.

Par exemple :

```
class UneClasse
  # Par défaut, les méthodes sont publiques
  def methode1
    # ...
  end

  private # Maintenant, les méthodes sont privées

  def methode2
    # ...
  end
  def methode3
    # ...
  end

  public # Sauf celle-ci, qui sera publique

  def methode4
    # ...
  end
end
```

Donc, dans notre carnet d'adresses, nous devons rajouter le mot clef `private` juste avant la définition de `Carnet#par_nom` :

```
class Carnet
  #
  # Méthodes fondamentales :
  #   initialize
  #   ajoute
  #   retire
  #
  def initialize
    @personnes = []
  end

  # ...

  private # Début des méthodes privées

  #
  # Fonction de tri.
  #
  def par_nom(a, b)
    if a.prenom == b.prenom
      a.nom <=>b.nom
    else
      a.prenom <=>b.prenom
    end
  end
end
```

## 4-6-B - Ré-utilisation du code avec `require`

Nous avons passé beaucoup de temps à écrire les trois classes pour notre carnet d'adresses. Nous ne voulons pas copier et coller éternellement le code chaque fois que nous allons l'utiliser dans un programme. Fort heureusement, nous n'avons pas besoin de le faire.

Copiez le code des trois classes dans un fichier, et sauvez-le sous le nom carnet.rb. Maintenant, créez un nouveau fichier (dans le même répertoire) et tapez :

```
require 'carnet'

# Nicolas

adresse = Adresse.new

adresse.rue          = "Rue du port, 32"
adresse.code_postal = "56000"
adresse.ville       = "Vannes"
adresse.pays        = "France"

puts "Nicolas:"
puts adresse
```

Sauvez-le sous le nom nicolas.rb par exemple. Maintenant, invoquez Ruby :

Vous l'aurez compris, le mot clef require permet de charger du code existant.

## 4-7 - Ecrire de bons programmes

L'utilisation propice de fonctions, méthodes et classes permet de distinguer les programmeurs expérimentés. Vous trouverez dans cette dernière section quelques conseils qui vous aideront par la suite.

### 4-7-A - Fonctions et méthodes

Les fonctions et les méthodes doivent toujours se charger d'une seule chose :

- Si vous ne pouvez pas résumer en une ligne ce que fait votre fonction, elle est probablement trop compliquée;
- Si le code de votre fonction ne tient pas sur un seul écran, elle est probablement trop longue.
- Des études ont démontré qu'un être humain se souvient au maximum de 7 choses à la fois. Si votre fonction comporte plus de 6 variables, elle est probablement trop compliquée, et trop longue.

### 4-7-B - Commentaires

Plus vous écrirez de programmes complexes, plus l'usage de commentaires se révélera important :

- Les commentaires ne doivent pas nécessairement être longs. Ils doivent juste expliquer clairement l'objectif du code.
- Chaque fonction devrait avoir un commentaire qui stipule ce que fait la fonction. Exceptionnellement, vous pouvez omettre le commentaire si la fonction est si simple que son usage est évident.
- Dans le cas du carnet d'adresses :
  - Les méthodes initialize, ajoute et retire n'ont pas besoin d'être commentées.

- Dans le code de la classe, les méthodes `chaque_personne` et `chaque_adresse` ont un seul commentaire, qui stipule que ce sont des itérateurs. Une fois que le programmeur sait que ces méthodes sont des itérateurs, il devine plus facilement l'usage de ces méthodes.
- Même chose pour la méthode `par_nom`, dont le commentaire explique au lecteur qu'il s'agit d'une fonction de tri.
- Les commentaires peuvent également être utilisés pour grouper ensemble une série de fonctions communes. Par exemple, dans le carnet d'adresses, nous avons utilisé des commentaires pour rassembler les "méthodes fondamentales", "itérateurs" et "fonctions de tri".




*Essayez toujours de diviser le code en petites parties, plus facilement maintenables.*


## A - Obtenir de l'aide

### A-1 - Support technique

Nombreux sont les développeurs qui ont choisi Ruby comme langage de prédilection! Et ce nombre tend à croître intensivement avec le temps. Ce qui forme une communauté fort soudée, toujours prête à vous aider, quelque soit le problème.

- La liste de diffusion officielle de Ruby est probablement le meilleur moyen pour trouver de l'aide. Les abonnés répondent rapidement, et ce d'une façon très amicale. Pour vous y inscrire, il suffit d'envoyer un e-mail à [ruby-talk-ctl@ruby-lang.org](mailto:ruby-talk-ctl@ruby-lang.org), en spécifiant le message suivant dans le corps de l'e-mail : "subscribe Votre-Prénom Votre-Nom".

 *Cette liste est très active. Vous recevrez une quantité non négligeable d'e-mails par jour.*

 *Les discussions se font uniquement en anglais. Néanmoins, il existe une liste de diffusion francophone. Pour s'inscrire, il faut envoyer à l'adresse [ruby-fr-ctl@ruby-lang.org](mailto:ruby-fr-ctl@ruby-lang.org) même style de message que pour la liste anglophone. Malheureusement, cette liste n'est pas très active...*

- IRC : Si vous avez un client IRC, vous pouvez obtenir de l'aide directement via le canal de discussion Ruby officiel. Connectez-vous simplement sur le réseau [freenode](#), et joignez le canal #ruby-lang.

 *Si vous ne savez pas ce qu'est IRC, alors [ce site](#) est fait pour vous.*

- Groupes d'utilisateurs Ruby : Il existe peut-être un groupe d'utilisateurs Ruby près de chez vous. Dans ce cas, il s'agit souvent d'un moyen intéressant pour avoir de l'aide sur Ruby, ainsi que de rencontrer par la même occasion d'autres passionnés. Vous pouvez obtenir une liste des groupes d'utilisateurs recensés à [l'adresse suivante](#).

### A-2 - Documentation

En plus de ce tutoriel, vous pouvez trouver d'autres documents utiles sur le langage Ruby :

#### A-2-A - Autres tutoriels

Chris Pine a également écrit un tutoriel d'introduction à Ruby : [Learn to Program](#)(disponible en anglais uniquement).

L'auteur a adopté une approche différente de la mienne, pour couvrir pratiquement le même contenu. Je vous suggère d'y jeter un coup d'oeil.

#### A-2-B - Livres

Voici quelques livres se focalisant sur Ruby. Remarquez, je n'ai pas lu les deux premiers bouquins. Mes commentaires sont donc basés sur ce que j'ai pu entendre via la communauté.

- *Sams Teach Yourself Ruby in 21 Days*(de Mark Slagell) : Il s'agit du livre le plus accessible dans cette liste. Le lecteur n'a besoin d'aucune connaissance particulière pour aborder cet ouvrage. L'auteur adopte un style "tutoriel" pour enseigner toutes les fonctionnalités importantes du langage, que ce soit les bases ou mêmes certains points avancés. Ce livre a été très bien accepté par la communauté. C'est probablement un bon bouquin.
- *The Ruby Way*(de Hal Fulton) : Ce livre est basé sur une approche "livre de cuisine". Vous y trouverez un bon nombre de solutions pour un large panel de problèmes fréquemment rencontrés. Il s'agit d'un livre d'une difficulté moyenne, et il est très souvent recommandé par la communauté.
- *Programming Ruby*(de David Thomas) : Ce livre est une référence complète du langage Ruby. Il est souvent représenté sous le nom de "PickAxe" (qui peut se traduire en français par "piolet", une sorte de pic à glace), simplement parce qu'il s'agit de l'illustration principale sur la couverture du livre. C'est de loin l'ouvrage le plus souvent mentionné dans la liste de diffusion Ruby. Il s'agit d'une excellente référence.

Ce livre est un peu plus avancé que ce tutorial. Le lecteur doit avoir quelques connaissances préalables en matière de programmation, et doit comprendre l'architecture orientée objet. Il est disponible gratuitement en ligne sur le site [rubycentral](http://rubycentral.com).

 Ces livres ne sont disponibles qu'en anglais.

## B - Installer Ruby sur votre machine

Voici quelques brèves instructions concernant l'installation de Ruby sur plusieurs systèmes différents. Si vous rencontrez des problèmes, n'hésitez pas à demander de l'aide sur la liste de diffusion (Annexe A).

### B-1 - GNU/Linux

La plupart des distributions GNU/Linux fournissent Ruby dans l'installation par défaut. Néanmoins, si Ruby n'est pas présent sur votre machine, voici ce qu'il faut faire pour l'installer en fonction de votre système :

#### B-1-A - Installation par RPM (Redhat, Mandrake...)

Vous pouvez télécharger un paquetage RPM sur [RPM Findet](#) et l'installer via la commande **rpm -Uhv ruby-\*.rpm**.

#### B-1-B - Gentoo

Ruby fait partie de portage. Un simple **emerge ruby** fera le boulot à votre place.

#### B-1-C - Debian

Ruby s'installera sur votre Debian comme n'importe quel autre logiciel, en invoquant **apt-get install ruby**.

### B-2 - Mac OS X

Ruby est distribué par défaut avec Mac OS X. Si vous le désirez, vous pouvez obtenir un binaire sur [le site de téléchargement d'Apple](#).

### B-3 - FreeBSD

Ruby fait partie de la collection des ports de FreeBSD. Vous pouvez donc l'installer en utilisant les instructions suivantes :

```
# cd /usr/ports/lang/ruby
# make
# make install
# make clean
```

### B-4 - Microsoft Windows

A l'heure où nous écrivons ces lignes, la dernière version disponible de Ruby pour Microsoft Windows est 1.6.8. Il existe un installateur Microsoft Windows qui se chargera de copier Ruby sur votre système, vous pouvez le télécharger [ici](#). Vous pouvez toujours trouver la dernière version de cet installateur sur [cette page](#).

