Christopher David PINE

APPRENDRE à PROGRAMMER

Traduction de Jean-Pierre ANGHEL

APPRENDRE à PROGRAMMER

Un tutoriel pour les futurs programmeurs

Table des matières

Avant propos

Si vous voulez programmer un ordinateur, vous devez "parler" dans une langue que votre machine comprendra : un langage de programmation. Il existe de nombreux langages de ce type, et certains sont même excellents. Dans ce tutoriel j'ai choisi d'utiliser mon langage de programmation favori, *Ruby*.

Avant d' être mon langage préféré, Ruby est aussi le langage le plus facile que j'aie jamais vu (et j'en ai vu quelques-uns croyez-moi). En fait, la raison principale pour laquelle j'ai écrit ce tutoriel est la suivante : je n'ai pas décidé d'écrire un tutoriel et choisi Ruby parce que c'est mon langage favori; j'ai trouvé Ruby si facile que j'ai pensé qu'il irait très bien comme tutoriel pour débutant. C'est la simplicité de Ruby qui m'a incité à écrire ce tutoriel et non le fait qu'il soit mon favori. (Ecrire un tutoriel identique pour un autre langage, C++ ou Java, nécessiterait des centaines et des centaines de pages). Mais n'allez pas penser que Ruby n'est qu' un langage pour débutant parce qu'il est facile ! C'est un puissant et robuste langage de programmation professionnel s'il en fut.

Quand vous écrivez quelque chose dans une langue humaine, ce que vous écrivez est appelé 'texte'. Quand vous écrivez quelque chose dans un langage de programmation, ce que vous écrivez est appelé 'code'. J'ai inclus plusieurs exemples tout au long de ce tutoriel, la plupart d'entre eux étant des programmes complets que vous pourrez exécuter sur votre ordinateur. Pour rendre le code plus facile à lire, j'ai coloré le code de couleurs différentes. (Par exemple, les nombres sont toujours en vert). Ce que vous êtes censés taper est dans une

```
boîte grise
```

et ce que le programme renvoie dans une

```
boîte bleue.
```

S'il y a quelque chose que vous ne comprenez pas, ou si vous avez une question qui demeure sans réponse, gardez-la sous le coude et attendez d'avoir fini votre lecture ! Il est fort possible que la réponse se trouve dans le chapitre suivant. Toutefois si à la fin du dernier chapitre vous n'avez toujours rien vu venir, je vous dirai qui questionner. Il y a de nombreuses personnes capables de vous aider, il suffit juste de savoir où les trouver.

Mais nous devons tout d'abord télécharger Ruby et l'installer sur votre ordinateur.

Installation sous Windows

L'installation de Ruby sous Windows est enfantine. Vous devez premièrement télécharger Ruby (http://rubyinstaller.sourceforge.net/) . Vous aurez sans doute le choix entre deux versions ; ce tutoriel a été fait avec la version 1.8.0, aussi assurez-vous de télécharger la version la plus récente. (Je voulais juste indiquer la dernière version disponible). Ensuite laissez-vous guider par le programme d'installation. Il vous sera demandé dans quel répertoire vous voulez installer Ruby. Choisissez de préférence le répertoire par défaut, à moins que vous n'ayez d'excellentes raisons de ne pas le faire.

Pour programmer, vous devez être capable d'écrire et d' exécuter un programme. Pour ce faire vous devez avoir un éditeur de texte et une ligne de commande.

L'installation de Ruby fournit un traitement de texte appelé 'SciTE' (pour Scintilla Text Editor) . Vous pouvez exécuter SciTE à partir de la fenêtre de commande de Ruby dans le menu 'Démarrer'. SciTE vous permettra d'une part d'avoir du code coloré comme dans ce tutoriel, et d'autre part vous évitera de taper à chaque essai `ruby + le nom du programme`, un appui sur la touche F5 (ou 'Outils/Exécuter', dans le menu) fera tout cela à votre place, le résultat s'affichant dans la partie droite de l'éditeur.

Ce sera aussi une bonne idée que de créer un répertoire pour sauvegarder tous vos programmes. Assurez-vous toujours quand vous sauvegardez un programme que vous êtes bien dans le bon répertoire.

Pour obtenir votre ligne de commande (dans une fenêtre noire), sélectionnez :

- Pour Windows 98 : 'Démarrer' puis 'Programmes' puis 'Accessoires' et enfin 'Commandes MSDos'.
- Pour Windows XP : 'démarrer' puis 'Programmes' puis 'Accessoires' et enfin 'Invite de commandes'.

Vous devrez ensuite naviguer jusqu'au répertoire où sont entreposés vos programmes. Taper

```
cd..
```

vous fait remonter d'un niveau dans la hiérarchie des répertoires.

Et taper

```
cd nom_répertoire
```

vous conduit directement où vous voulez.

Pour voir tous les répertoires inclus dans le vôtre tapez :

```
dir  /ad
```

Installation sur Macintosh

Si vous possédez Mac OS X 10.2 (Jaguar) vous avez déjà Ruby sur votre système ! Que demander de plus ? Si vous avez la version 10.1, vous devez télécharger Ruby. Et si malheureusement pour vous, vous êtes encore sous Mac OS 9 ou une version encore plus ancienne, je ne pense pas que vous puissiez utiliser Ruby.

Pour programmer, vous devez être capable d'écrire et d' exécuter un programme. Pour ce faire vous devez avoir un éditeur de texte et une ligne de commande.

Votre ligne de commande est accessible par l'application Terminal (dans Applications/Utilitaires)

Comme éditeur de texte utilisez celui avec lequel vous vous sentez le plus à l' aise. Si toutefois vous utilisez TextEdit, assurez-vous lors des sauvegardes que vous êtes en texte seulement ! Sinon vos programmes ne fonctionneront pas. D'autres options accessibles de la ligne de commande sont emacs, vi et pico.

Installation sous Linux

Premièrement, vérifiez que Ruby n'est pas déjà installé sur votre système ! Tapez :

```
which ruby
```

Si vous obtenez une réponse du style

```
/usr/bin/which: no ruby in (…)
```

vous devez alors télécharger Ruby, sinon vérifiez la version de Ruby en tapant

```
ruby -v
```

Si votre version est inférieure à celle proposée sur le site de téléchargement, chargez donc la nouvelle version.

Si vous êtes l'administrateur (root), vous n'aurez certainement besoin d' aucune instruction pour installer Ruby. Dans le cas contraire, vous devez demander à votre administrateur système de faire l'installation pour vous (ceci afin que quiconque sur ce système puisse utiliser Ruby).
Sinon, vous pouvez l' installer comme seul utilisateur. Déplacez le fichier que vous avez téléchargé dans un répertoire temporaire, par exemple $HOME/tmp. Si le nom du fichier est ruby-1.6.7.tar.gz, vous pouvez l'ouvrir par

```
tar zxvf ruby-1.6.7.tar.gz
```

Changez de répertoire pour celui que vous venez de créer. Dans cet exemple :

```
cd ruby-1.6.7
```

Configurez votre installation en tapant :

```
./configure --prefix=$HOME
```

puis tapez :

```
make
```

ce qui construira votre interpréteur. Ceci peut prendre quelques minutes. Une fois terminé tapez :

```
make install
```

pour l'installer.

Ensuite vous devez ajouter $HOME/bin à votre commande de recherche de chemin en éditant le fichier $HOME/.bashrc (vous devez quitter la session et re-entrer pour que ceci soit pris en compte). Après ceci testez votre installation :

```
ruby -v
```

Si vous obtenez une réponse vous indiquant le numéro de version, vous pouvez alors effacer les fichiers dans le répertoire $HOME/tmp (où là où vous les aviez mis).

1. Les Nombres

Maintenant que nous avons tout installé, nous allons écrire un programme. Ouvrez votre éditeur de texte favori et tapez ce qui suit:

```
puts 1+2
```

Sauvez votre programme (si, si c'est un programme !) sous le nom p7.rb ('.rb' c'est ce que nous mettons habituellement à la fin du nom d'un programme en Ruby), “p7” c’est le numéro de la page, comme cela vous pourrez facilement retrouver vos exemples. Maintenant exécutez votre programme en tapant:

`ruby p7.rb  (ou bien F5 dans SciTE)`

dans votre ligne de commande. Vous devez obtenir un 3 sur votre écran. Vous voyez, programmer n'est pas si difficile, n'est ce pas ?

Introduction à `puts`

Que va donc faire ce programme ? Je suis sûr que vous avez deviné ce que donnera 1+2; votre programme est en fait équivalent à

```
puts 3
```

`puts` (de put : mettre, poser) écrit simplement tout ce qui le suit.

Entiers et réels

Dans la plupart des langages (et Ruby ne fait pas exception) les nombres sans virgule sont appelés des *entiers*, et ceux avec une virgule des *réels*. Attention, comme sur les calculettes, la virgule française est remplacée par un point !

Voici quelques entiers :

```
5
-205
999999999999999999999999
0
```

et voici quelques réels :

```
54.321
0.001
-205.3884
0.0
```

En pratique les programmes n'utilisent pas de réels; seulement des entiers. (Après tout, personne ne va regarder 7,4 courriels, ou faire défiler 1,8 pages du Web, ou écouter 5,24

morceaux de musique...). Les réels sont utilisés plutôt pour des applications académiques (expériences de physique etc...) et pour les graphiques 3D. Même les programmes financiers utilisent des entiers; ils gardent seulement trace du nombre de centimes !

Arithmétique simple

Bon, nous avons tout ce qu'il faut pour simuler une calculatrice. Les calculatrices utilisent les réels, donc si nous voulons que notre ordinateur réagisse comme elles, nous devons utiliser les réels. Pour les additions et les soustractions, nous utiliserons les signes '+' et '-'. Pour les multiplications et les divisions les signes '*' et '/'. La plupart des claviers possèdent ces touches sur le clavier numérique . Sur les mini-claviers et les portables il faut les rechercher sur le clavier normal en utilisant la touche 'Shift' (la flèche vers le haut) simultanément. Modifions maintenant un peu notre programme en p8.rb. Tapez ce qui suit et faites exécuter à chaque ligne :

```
puts 1.0 + 2.0
puts 2.0 * 3.0
puts 5.0 - 8.0
puts 9.0 / 2.0
```

ce qui vous retournera :

```
3.0
6.0
-3.0
4.5
```

(les espaces dans le programme ne sont pas importants; ils rendent tout simplement le code plus lisible.) La virgule française est remplacée par un point ! Bien, à part ça, il n'y a pas de surprises. Essayons maintenant avec des entiers :

```
puts 1+2
puts 2*3
puts 5-8
puts 9/2
```

A première vue c'est pareil, non ?

```
3
6
-3
4
```

Zut, le dernier ne donne pas le même résultat ! Et bien quand vous calculez avec des entiers vous obtenez des réponses en entiers. Quand l'ordinateur ne peut pas donner la "bonne" réponse, il arrondit toujours le résultat. (Bien sûr, 4 est la bonne réponse en arithmétique entière pour 9/2; il se peut cependant que ce ne soit pas la réponse à laquelle vous vous attendiez.)

Peut-être vous demandez-vous à quoi servent les entiers. Imaginez que vous ayez 9 euros et que vous désiriez voir des films dans un cinéma à 2 euros la séance (NDT: l'exemple est américain, cela va de soit !) Combien pourrez-vous voir de films ? 4,5 n'est décidément pas la bonne réponse dans ce cas; on ne vous laissera pas voir la moitié d'un film ni payer la moitié du prix pour voir un film entier... certaines choses ne sont pas divisibles.

Essayez donc maintenant de faire des exemples tout seul. Si vous voulez vous pouvez utiliser des expressions complexes et utiliser des parenthèses. Par exemple :

```
puts 5 * (12-8) + -15
puts 98 +  (59872 / (13*8)) * -52
```

```
5
-29802
```

Quelques petits trucs à essayer

Ecrivez un programme qui vous dise :
- combien d'heures y a-t-il dans une année.
- combien de minutes dans une décennie.
- de combien de secondes êtes-vous âgé(e).
- combien de chocolat espérez-vous manger durant votre vie.

Attention : cette partie de programme risque de prendre un certain temps de calcul !

Ici une question plus difficile :
- Si je suis âgé de 870 millions de secondes, quel âge ai-je ?

2. Les lettres

Bien, nous avons tout appris des nombres, mais qu'en est-il des lettres, des mots, du texte?

Nous nous réfèrerons à un groupe de lettres dans un programme, comme à une *chaîne de caractères.* (Vous pouvez penser à des lettres que l'on rajoute les unes après les autres sur une bannière). Pour rendre plus visibles les parties de code qui sont des chaînes de caractères, je vais les mettre en rouge. Voici quelques exemples :

```
'Coucou'
'Ruby marche fort.'
'5 est mon nombre favori...quel est le vôtre ?'
'Snoopy dit #%^&*@ quand il se cogne le pied.'
'       '
''
```

Comme vous pouvez le voir, les chaînes de caractères peuvent contenir de la ponctuation, des chiffres, des symboles et des espaces en plus des lettres. La dernière chaîne n'a rien à l'intérieur, nous dirons que c'est une *chaîne vide*.

Nous avons utilisé puts pour écrire des nombres; essayons avec quelques chaînes de caractères :

```
puts 'Coucou'
puts ''
puts 'au revoir.'
```

```
Coucou

au revoir.
```

Cela "marche" bien. Maintenant essayez des chaînes de votre cru.

Chaînes arithmétiques

De la même façon que vous faites de l'arithmétique avec des nombres, vous pouvez en faire avec des chaînes ! Enfin, quelque chose qui y ressemble. Essayons d'additionner deux chaînes et voyons ce que fait puts dans ce cas.

```
puts 'Mon fromage préféré est' + 'le roquefort.'
```

```
Mon fromage préféré estle roquefort.
```

Oups! J'ai oublié un espace entre 'est' et 'le'. Les espaces sont habituellement sans importance, mais dans le cas des chaînes si. C'est vrai ce que l'on dit : les ordinateurs ne font pas ce que vous voudriez qu'ils fassent, mais seulement ce que vous leur dites de faire.)

Essayons encore :

```
puts 'Mon fromage préféré est ' + 'le roquefort.'
puts 'Mon fromage préféré est' + ' le roquefort.'
```

```
Mon fromage préféré est le roquefort.
Mon fromage préféré est le roquefort.
```

(Comme vous pouvez le voir, l'endroit où j'ai rajouté l'espace n'a pas d'importance.)

Si vous pouvez ajouter des chaînes vous pouvez aussi les multiplier (par un nombre):

```
puts 'pim ' * 3
```

```
pim pam poum
```

Non, je rigole... en réalité on obtient :

```
pim pim pim
```

En fait si vous réfléchissez, cela est logique. Après tout, 7*3 signifie 7+7+7, ainsi 'miaou' * 3 signifie simplement 'miaou' + 'miaou' + 'miaou'.

12 versus '12'

Avant d'aller plus loin, il vaut mieux être sûr que vous faites bien la différence entre *nombres* et *chiffres*. 12 est un nombre, mais '12' est une chaîne de caractères de deux chiffres.
Insistons encore un peu :

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

```
24
1212
12  +  12
```

et que pensez-vous de cela :

```
puts 2 * 5
puts '2' * 5
puts '2  *  5'
```

```
10
22222
2  *  5
```

Ces exemples sont bien sûr très simples. Cependant, si vous ne faites pas très attention lorsque vous mélangez des chaînes de caractères et des nombres, vous pouvez vous tromper...

Problèmes

A ce stade vous avez peut-être découvert des choses qui ne fonctionnaient pas. Dans le cas contraire, voici tout de même quelques exemples de ce qui peut arriver :

```
puts '12' + 12
puts '2' * '5'
```

```
p12.rb:1:in `+': failed to convert Fixnum into String (TypeError)
    from p12.rb:1
```

Aïe... un message d'erreur. Le problème vient de ce que vous ne pouvez pas ajouter un nombre à une chaîne, ou multiplier une chaîne par une autre chaîne. Cela n'a pas plus de sens que :

```
puts 'Betty' + 12
puts 'Fred' * 'Jean'
```

Encore un autre cas pour montrer l'importance de l'ordre d'écriture : vous pouvez écrire 'pim' * 5 dans un programme, puisque cela signifie seulement que vous allez ajouter 5 fois la chaîne 'pim'. Par contre, vous ne pouvez pas écrire 5 * 'pim', ce qui voudrait dire 'pim' fois le chiffre 5, ce qui ne veut rien dire.

Finalement, si je veux écrire un programme qui affiche 'J'ai faim !', essayons comme cela :

```
puts 'J'ai faim !'
```

Bon, ça ne "marche" pas. Je ne peux même pas l'exécuter. L'ordinateur pense que nous en avons terminé avec la chaîne lorsqu'il rencontre la deuxième apostrophe. (D'où l'avantage d'utiliser un éditeur de texte qui colorie la syntaxe pour vous !). Comment faire pour indiquer à l'ordinateur que la chaîne continue ? Nous devons supprimer l'action de l'apostrophe comme ceci :

```
puts 'J\'ai faim !'
```

```
J'ai faim !
```

Le caractère "\" est le caractère d'*échappement*. Ce qui signifie que, si vous avez un "\" et un autre caractère, ils sont parfois transformés en un autre caractère. Les seules choses que "\" modifie sont l'apostrophe et "\" lui-même.

Quelques exemples pour mieux comprendre :

```
puts 'J\'ai faim !'
puts 'à la fin d\'une chaîne : \\'
puts 'oui\\non'
puts 'oui\non'
```

```
J'ai faim !
à la fin d'une chaîne : \
oui\non
oui\non
```

Puisque "\" n'a pas d'influence sur un "n", mais sur lui-même oui, les deux derniers exemples sont équivalents. Ils ne paraissent pas les mêmes dans votre code, mais pour l'ordinateur ils sont identiques.

3. Variables et affectations

Jusqu'à présent, tout ce que nous avons "putsé" c'est une chaîne ou un nombre, mais ce que nous avons "putsé" a disparu. Ce que je veux dire c'est que si nous voulons afficher quelque chose deux fois à l'écran, nous devons le taper deux fois :

```
puts '... vous pouvez répéter...'
puts '... vous pouvez répéter...'
```

```
... vous pouvez répéter...
... vous pouvez répéter...
```

Ce qui serait bien, c'est si l'on pouvait juste taper une chose une fois, la prendre et ... la stocker quelque part. Bien sûr, nous le pouvons, sinon je n'aurais pas posé la question !

Pour stocker la chaîne dans la mémoire de l'ordinateur, nous devons donner un nom à la chaîne. Les programmeurs disent qu'ils *affectent* une valeur, et appellent les noms des *variables*. Ces variables peuvent être des séquences de lettres ou bien de chiffres, mais le premier caractère est obligatoirement une minuscule. Essayons ceci dans le programme précédent, mais à ce stade il faut donner un nom à la chaîne. Pourquoi pas 'maChaine' (que l'on pourrait tout aussi bien appeler chat ou maToutePetiteChaine ou encore louisQuatorze)

```
maChaine = '...pouvez-vous répéter...'
puts maChaine
puts maChaine
```

```
...pouvez-vous répéter...
...pouvez-vous répéter...
```

Chaque fois que vous essayez de faire quelque chose avec maChaine, le programme vous retourne '`...pouvez-vous répéter...`'. On peut imaginer que la variable maChaine "pointe" sur la chaîne '`...pouvez-vous répéter...`'. Voici un exemple plus intéressant :

```
nom = 'Hubert de La PâteFeuilletée'
puts 'Mon nom est ' + nom + '.'
puts 'Ouf! ' + nom + ' est vraiment un nom long!'
```

```
Mon nom est Hubert de La PâteFeuilletée.
Ouf ! Hubert de La PâteFeuilletée est vraiment un nom long!
```

Bon, de la même façon que nous pouvons *affecter* un objet à une *variable,* nous pouvons *ré-affecter* un objet différent à cette *variable*. (C'est pour cela que nous les appelons des variables : parce que ce qu'elles "pointent" (désignent) peut varier.)

```
compositeur = 'Mozart'
puts compositeur + ' était "une vedette", en son temps.'
compositeur = 'Beethoven'
```

```
puts 'Mais je préfère ' + compositeur +', personnellement.'
```

```
Mozart était "une vedette", en son temps.
Mais je préfère Beethoven, personnellement.
```

Bien entendu les variables peuvent pointer vers d'autres choses que des chaînes de caractères :

```
var = 'juste une autre ' + 'chaîne'
puts var

var = 5 * (1 + 2)
puts var
```

```
juste une autre chaîne
15
```

En fait les variables peuvent pointer sur n'importe quoi ... excepté sur d'autres variables. Mais que se passe-t-il si nous essayons ceci :

```
var1 = 8
var2 = var1
puts var1
puts var2

puts ''

var1 = 'Huit'
puts var1
puts var2
```

```
8
8

Huit
8
```

Dans le premier cas, quand nous essayons de faire pointer `var2` vers `var1`, en réalité nous pointons sur 8 (juste parce ce que `var1` pointait sur 8). Ensuite nous avons `var1` qui pointe sur '`Huit`', mais puisque `var2` n'a jamais réellement pointé vers `var1`, elle reste sur 8.

4. Mélangeons-les

Nous avons vu différentes sortes d'objets (nombres et lettres), et nous avons fabriqué des variables pour pointer vers eux; la prochaine étape vise à jouer, le plus agréablement possible, avec tout ces objets ensemble.

Nous avons vu que si nous voulons écrire un programme pour écrire 25, ce qui suit *ne marche pas*, parce que nous ne pouvons pas additionner des nombres et des chaînes de caractères :

```
var1 = 2
var2 = '5'

puts var1 + var2
```

Une partie du problème provient du fait que votre ordinateur ne comprend pas si vous voulez obtenir 7 (2 + 5), ou bien 25 ('2' + '5').

Avant de pouvoir les additionner ensemble, nous devons trouver un moyen d'obtenir la version chaîne de var1, ou bien la version nombre de var2.

Conversions

Pour obtenir la version chaîne de caractères d'un objet, il suffit d'écrire `.to_s` à sa suite (s pour *string;* `.to_s`= transforme en chaîne).

```
var1 = 2
var2 = '5'

puts var1.to_s + var2
```

De la même façon, `to_i` donne la version nombre (entière) d'un objet (i pour *integer*), et `to_f` la version nombre réel (f pour *float*). Voyons d'un peu plus près ce que donnent ces trois méthodes:

```
var1 = 2
var2 = '5'

puts var1.to_s + var2
puts var1 + var2.to_i
```

```
25
7
```

Notons que même après avoir obtenu la version chaîne de `var1` en appelant to_s, var1 pointait toujours sur 2, et jamais sur '2'. A moins que nous réassignions `var1` (ce qui nécessite le signe =), la variable pointera sur 2 durant toute la durée du programme.

Essayons maintenant quelques conversions intéressantes (pour ne pas dire étranges) :

```
puts '15'.to_f
puts '99.999'.to_f
puts '99.999'.to_i
puts ''
puts '5 est mon chiffre favori !'.to_i
puts 'Qui vous parle de 5 ?'.to_i
puts 'votre maman le faisait'.to_f
puts ''
puts 'bizarre'.to_s
puts 3.to_i
```

```
15.0
99.999
99

5
0
0.0

bizarre
3
```

Bon, il y a quelques surprises. La première réponse était prévisible et donne 15.0. Après ça, nous convertissons la chaîne '99.999' en un nombre réel puis en un entier.
`to_f` a fait ce que nous attendions; `to_i` a arrondi à la valeur entière immédiatement inférieure.

Ensuite, nous avons quelques exemples de chaînes inhabituelles converties en nombres. `to_i` ignore la première chose qu'il ne comprend pas, et le reste de la chaîne à partir de ce point. Ainsi la première chaîne a été convertie en 5, mais les autres, puisqu'elles débutent par une lettre, sont totalement ignorées... l'ordinateur choisit simplement de répondre par un 0.

Finalement, nous nous rendons compte que les deux dernières conversions ne font rien du tout, comme on pouvait s'y attendre.

Un autre regard sur `puts`

Il y a tout de même quelque chose d'étrange dans notre méthode favorite... jetez un coup d' oeil sur ce qui suit :

```
puts 20
puts 20.to_s
puts '20'
```

```
20
20
20
```

Pourquoi diable ces trois-là impriment-ils la même chose ? Bon, les deux derniers ça se comprend, puisque `20.to_s` est bien '20'. Mais qu'en est-il du premier, l'entier 20? Dans ce cas, que signifie écrire l'*entier* 20? Quand vous écrivez un 2 et ensuite un 0 sur une feuille de papier, vous écrivez une chaîne de caractères et non un entier. L'*entier* 20 est le nombre de doigts et d'orteils que j'ai; ce n'est pas un 2 suivi d'un 0.

Bien, il est temps de dévoiler le grand secret de notre ami `puts` : avant que `puts` tente d'écrire un objet, il utilise `to_s` pour obtenir la version chaîne de cet objet. En fait, le s de `puts` est mis pour *string* (chaîne en anglais); `puts` en réalité signifie *put string (exprimer, mettre en chaîne)*

Ceci peut vous paraître peu intéressant, mais il y a une multitude d'objets dans Ruby (et vous allez même apprendre à créer les vôtres !), et il est intéressant de savoir ce qui se passe lorsque vous tentez d'appliquer la méthode puts à un objet étrange, tel qu'une photo de votre grand-mère, ou un fichier musical etc... Mais cela viendra plus tard...

En attendant, nous avons quelques méthodes pour vous, et elles vont nous permettrent d'écrire toutes sortes de programmes sympathiques.

Les méthodes `gets` et `chomp`

Si puts signifie *put string*, je suis sûr que vous devinerez ce que veut dire gets (get = obtenir). Et de la même façon que puts affiche toujours des chaînes, gets renverra seulement des chaînes. Et d'où les obtiendra-t-il ?

De vous ! Bon, de votre clavier, bien sûr. Et puisque votre clavier fabrique seulement des chaînes, c'est parfait. Ce qu'il se passe actuellement c'est que gets lit tout ce que vous tapez jusqu'à ce que vous appuyiez sur la touche Enter. Essayons ceci :

```
puts gets
```

```
Y a-t-il un écho ici ?
Y a-t-il un écho ici ?
```

Bien sûr, tout ce que vous tapez est répété après vous. Exécutez-le quelques fois et essayez différentes choses.

Maintenant nous pouvons faire des programmes interactifs ! Dans celui-ci, entrez votre nom et il vous répondra :

```
puts 'Bonjour, quel est donc votre nom ?'
monnom = gets
puts 'Votre nom est ' + monnom + '? Quel joli nom !'
puts 'Ravi de faire votre connaissance ' + monnom + '. :)'
```

Aïe! J'exécute -j'entre juste mon nom - et voici ce qui s'affiche :

```
Bonjour, quel est donc votre nom ?
Christian
Votre nom est Christian? Quel joli nom !
Ravi de faire votre connaissance Christian
. :)
```

Hmmm...il semble que lorsque je tape les lettres `C,h,r,i,s,t,i,a,n` et presse ensuite la touche `Enter`, `gets` renvoie toutes les lettres de mon nom mais aussi la touche `Enter`! Heureusement il existe une méthode pour ce genre de choses : chomp. Elle enlève toutes les frappes de la touche Enter qui peuvent suivre votre chaîne. Essayons le programme de nouveau, mais avec `chomp` (mastiquer) pour nous aider :

```
puts 'Bonjour, quel est donc votre nom ?'
monnom = gets.chomp
puts 'Votre nom est ' + monnom + '? Quel joli nom !'
puts 'Ravi de faire votre connaissance ' + monnom + '. :)'
```

```
Bonjour, quel est donc votre nom ?
Christian
Votre nom est Christian? Quel joli nom !
Ravi de faire votre connaissance Christian. :)
```

Voilà qui est mieux ! Notons que puisque `monnom` pointe vers `gets.chomp`, nous n'avons pas eu à dire `monnom.chomp`; `monnom` était déjà “chompé”.

Quelques petites choses à essayer

- Ecrire un programme qui demande à une personne son premier prénom, son deuxième, et enfin son nom de famille. En sortie le programme devra féliciter la personne en utilisant son identité complète.
- Ecrire un programme qui demande à une personne son nombre favori. Votre programme devra ajouter un à ce nombre, et proposer le résultat comme étant plus grand et donc meilleur pour un favori (Dites-le tout de même avec tact !).

Lorsque vous aurez achevé ces deux programmes (et plus si vous désirez en essayer d'autres), nous apprendrons d'autres méthodes (et un peu plus sur les méthodes en général).

5. Où on en apprend plus sur les méthodes

Nous avons vu un certain nombre de méthodes, `puts` et `gets` et d'autres (**Test surprise** : donnez la liste des méthodes que nous avons déjà vues ! Il y en a dix; la réponse est ci-dessous), mais nous ne nous sommes pas posé la question de savoir ce qu'était exactement une méthode. Nous savons ce qu'elles font mais pas ce qu'elles sont.

En réalité, c'*est* ce qu'elles sont : des choses qui “font” . Si les objets (tels que chaînes, entiers, réels) sont les *noms* dans le langage Ruby, alors les méthodes ressemblent à des *verbes*. Et, comme en français, vous ne pouvez avoir un verbe sans un nom ("sujet") pour l'exécuter. Par exemple, sonner n'est pas le fait du hasard; le carillon (ou un autre engin) fait cela. En français nous dirons, "Le carillon sonne". En Ruby nous dirons `carillon.sonne` (en supposant que carillon soit un objet Ruby, bien sûr). Les programmeurs disent alors qu'ils "appellent la méthode `sonne` de `carillon`".

Bien, avez-vous fait le test surprise ? Je suis sûr que vous vous souveniez des méthodes `puts`, `gets` et `chomp`, puisque nous venions juste de les voir. Vous n'avez probablement pas oublié nos méthodes de conversions, `to_i`, `to_f` et `to_s`. Cependant, avez-vous trouvé les quatre autres ? pourtant ce ne sont que nos vieilles connaissances arithmétiques +, -, * et / !

Comme je l'ai déjà dit, de la même façon qu'un verbe nécessite un nom, une méthode nécessite un objet. Il est habituellement facile de dire quel objet exécute telle méthode : c'est celui qui précède le point, comme dans notre exemple `carillon.sonne`, ou dans `101.to_s`. Ce n'est pas toujours aussi simple; comme avec les méthodes arithmétiques. Quand nous écrivons `5 + 5`, il s'agit en réalité du raccourci de `5.+ 5`. Par exemple :

```
puts 'Salut ' .+ 'les gars'
puts (10.* 9).+ 9
```

```
Salut les gars
99
```

Ce n'est pas très joli, aussi ne l'écrivons-nous jamais ainsi; cependant, il est important de comprendre ce qui se passe réellement. (Sur ma machine, j'ai droit à un avertissement : `warning: parenthesize argument(s) for future version` (attention : argument(s) entre parenthèses pour version future). Le code s'est bien exécuté mais on me prévient qu'il peut y avoir des difficultés à interpréter ma pensée , donc d'utiliser plus de parenthèses dans le futur). Cela nous donne aussi une explication plus profonde sur le fait qu'on peut écrire `'pim'*5` mais pas `5*'pim'` : `'pim'*5` appelle `'pim'` pour faire une multiplication par 5, tandis que `5*'pim'` appelle 5 pour faire une multiplication par `'pim'` !. `'pim'` sait comment faire 5 copies de lui-même et les ajouter ensemble, mais 5 éprouve de sacrées difficultés pour faire `'pim'` copies de lui-même et les additionner.

Et bien sûr nous avons `puts` et `gets` à expliquer. Où sont donc leurs objets? En français, vous pouvez parfois omettre le nom; par exemple si on interpelle quelqu'un par "Viens !", le nom de celui qui est interpellé est implicite pour lui. En Ruby, si je dis `puts 'Etre ou ne pas être'`, ce que je suis en train de dire en réalité c'est `self.puts 'Etre ou ne pas être'`. Mais qu'est-ce donc que ce `self` (moi-même) ? C'est une variable spéciale qui pointe sur un objet quelconque dans lequel vous

êtes. Nous ne savons pas encore comment être dans un objet, mais jusqu'à ce que nous le trouvions, sachez que nous sommes toujours dans un grand objet qui est... le programme lui-même en entier ! Et heureusement pour nous, le programme possède quelques méthodes en propre, telles que puts et gets. Observons ceci :

```
jepeuxdonnerunnomdevariabletreslongpourpointersurun3 = 3
puts jepeuxdonnerunnomdevariabletreslongpourpointersurun3
sel.puts jepeuxdonnerunnomdevariabletreslongpourpointersurun3
```

```
3
3
```

Si vous n'avez pas entièrement suivi tout ce qui précède, c'est parfait. La chose importante à retenir de tout ça est que toute méthode appartient à un objet même s'il n'y a pas de point qui la précède. Si vous avez compris cela, vous êtes sur la bonne voie.

Méthodes originales de chaînes

Voici quelques méthodes amusantes des chaînes de caractères. Vous n'avez pas à les retenir toutes; vous pourrez toujours revenir à cette page si vous les oubliez. Je veux juste vous montrer une petite partie de ce que peuvent faire les chaînes. En fait, je ne suis pas capable moi-même d'en retenir ne serait-ce que la moitié, mais cela n'a aucune importance car on trouve sur Internet de nombreuses références sur les méthodes de chaînes avec toutes les explications nécessaires. (Je vous indiquerai à la fin de ce tutoriel où les trouver.)
Vraiment, je n'ai pas besoin de connaître toutes les méthodes des chaînes; ce serait comme connaître tous les mots du dictionnaire. Je suis capable de m'exprimer correctement en français sans pour cela en connaître tous les mots... et puis n'est-ce pas tout l'intérêt du dictionnaire ? Ainsi vous n'avez pas à savoir ce qu'il y a dedans.

Bon, notre première méthode s'appelle reverse, et donne une version inversée d'une chaîne :

```
var1 = 'stop'
var2 = 'stressed'
var3 = 'pouvez-vous prononcer ceci inversé ?'

puts var1.reverse
puts var2.reverse
puts var3.reverse
puts var1
puts var2
puts var3
```

```
pots
desserts
? ésrevni icec recnonorp suov-zevuop'
stop
stressed
pouvez-vous prononcer ceci inversé ?
```

Comme vous le voyez, `reverse` n'inverse pas la chaîne d'origine; elle en fait seulement une nouvelle version inversée. C'est bien pour cela que `var1` est bien égale à `'stop'` après l'appel de reverse sur `var1`.

Une autre méthode est `length`, qui nous renseigne sur le nombre de caractères composant la chaîne (espaces inclus) :

```
puts 'Quel est votre nom complet ?'
nom = gets.chomp
puts 'Savez-vous qu\' il y a ' + nom.length + ' caractères dans
votre nom, ' + nom + ' ?'
```

```
Quel est votre nom complet ?
Christopher David Pine
P22.rb:3:in '+' : cannot convert Fixnum into String (TypeError)
                            from p22.rb:3
```

Aïe, aïe, aïe. Quelque chose ne marche pas, et cela semble se produire après la ligne `nom = gets.chomp`... Voyez-vous le problème ? essayez de vous le représenter.

Le problème est avec `length` : la méthode nous donne un nombre, mais il nous faut une chaîne. Facile, il nous suffit de rajouter un `to_s` (et de croiser les doigts) :

```
puts 'Quel est votre nom complet ?'
nom = gets.chomp
puts 'Savez-vous qu\' il y a ' + nom.length.to_s + ' caractères
dans votre nom, ' + nom + ' ?'
```

```
Quel est votre nom complet ?
Christopher David Pine
Savez-vous qu' il y a 22 caractères dans votre nom, Christopher
David Pine ?
```

Non, je ne le savais pas. Note : c'est le nombre de caractères dans mon nom, pas le nombre de lettres (comptez-les). Je propose que vous écriviez un programme qui vous demande vos nom et prénoms, individuellement, et les ajoute ensuite l'un à la suite de l'autre... Hé, pourquoi ne le faites-vous pas? Un peu de nerf, je vous attends.

Alors, c'est fait ? Bien! N'est-ce pas que c'est agréable de programmer ? Encore quelques chapitres et vous serez étonné(e) de ce que vous pouvez faire.

Bien, il existe aussi des méthodes qui changent la casse de votre chaîne (majuscule et minuscule). `upcase` change chaque minuscule en majuscule, et `downcase` change chaque majuscule en minuscule. `swapcase` intervertit chaque lettre de la chaîne et, enfin, `capitalize` est comme `downcase`, sauf qu' elle met la première lettre de la chaîne en majuscule (si c'est une lettre).

```
lettres = 'aAbBcCdDeE'
puts lettres.upcase
puts lettres.downcase
puts lettres.swapcase
```

```
puts lettres.capitalize
puts ' a'.capitalize
puts lettres
```

```
AABBCCDDEE
aabbccddee
AaBbCcDdEe
Aabbccddee
a
aAbBcCdDeE
```

Joli petit truc. Comme vous pouvez le voir dans la ligne `puts ' a'.capitalize`, la méthode `capitalize` ne met en majuscule que le premier caractère, non la première lettre. Et le dernier exemple montre, comme nous l'avons déjà vu, que l'appel de toutes ces méthodes ne change pas la variable `lettres`. Je sais que je rabâche, mais il est important de comprendre cela.

Il existe des méthodes qui modifient l'objet associé, mais nous ne les verrons pas encore, et cela pour un bout de temps.

Les dernières méthodes originales que nous allons voir concernent le formatage visuel des chaînes. La première, `center`, ajoute des espaces au début et à la fin pour centrer la chaîne. Cependant, de la même façon que vous appelez `puts` pour ce que vous voulez afficher, et + pour ce que vous voulez lui ajouter, vous appellerez `center` pour centrer une chaîne suivant vos désirs. Ainsi si je souhaite centrer les lignes d'un poème, je dois faire quelque chose dans le style :

```
largeurLigne = 50

puts (          'Une souris verte'.center(largeurLigne))
puts ('Qui courrait dans l\'herbe'.center(largeurLigne))
puts ('Je l\'attrape par la queue'.center(largeurligne))
puts ('Je la montre à ce monsieur'.center(largeurLigne))
puts (        'Ce monsieur me dit'.center(largeurligne))
puts (  'Trempez-la dans l\'huile'.center(largeurLigne))
puts (    'Trempez-la dans l\'eau'.center(largeurligne))
puts (       'Ca fera un escargot'.center(largeurLigne))
puts (                 'Tout chaud'.center(largeurligne))
```

```
          Une souris verte
       Qui courrait dans l'herbe
       Je l'attrape par la queue
       Je la montre à ce monsieur
          Ce monsieur me dit
        Trempez-la dans l'huile
         Trempez-la dans l'eau
          Ca fera un escargot
             Tout chaud
```

Hmmm...Je ne suis pas vraiment sûr que la comptine sois juste, mais j'ai la flemme de vérifier. (Je voulais juste vérifier la partie `.center(largeurLigne)`, c'est pourquoi j'ai laissé des espaces avant les chaines. Je pense que cela fait plus joli ainsi. Les programmeurs ont souvent des impressions mauvaises quant à ce qui peut être joli dans un programme et le dénigrent. Plus vous programmerez, plus vous vous créerez un style). En parlant de paresseux, la paresse n'est pas toujours une mauvaise chose en programmation. Par exemple, savez-vous pourquoi j'ai stocké la largeur du poème dans la variable `largeurLigne` ? C'est afin que dans le cas où je reprenne plus tard le poème et le fasse plus large, je n'ai plus que la première ligne du programme à changer et non chaque ligne à centrer. Avec un poème très long, cela me fait gagner du temps. On peut dire que la paresse est une vertu en programmation.

Bien, au sujet du centrage... vous avez certainement observé que ce n'est pas aussi joli que ce que vous donne un traitement de texte. Si vous voulez réellement perfectionner le centrage (et peut-être la fonte), alors vous devez tout simplement utiliser un traitement de texte ! Ruby est un outil magnifique, mais aucun outil n'est le bon outil pour tous les travaux.

Deux autres méthodes de formatage de chaîne sont `ljust` et `rjust`, qui signifient respectivement left justify (justification gauche) et right justify (justification droite). Elles sont similaires à `center`, sauf qu'elles remplissent la chaîne d'espaces sur la droite pour la première et sur la gauche pour la deuxième. Voyons un peu les trois en action :

```
largeurLigne = 40
str = '--> texte <--'
puts str.ljust largeurLigne
puts str.center largeurLigne
puts str.rjust largeurLigne
puts str.ljust (largeurLigne/2) + str.rjust (largeurLigne/2)
```

```
p24.rb:3 warning: parenthesize argument(s) for future version
...
p24.rb:6 syntax error
puts str.ljust (largeurLigne/2) + str.rjust (largeurLigne/2)
```

Vlan. Une erreur de syntaxe. Les erreurs normales signifient que nous avons fait une chose qui n'est pas permise. Les erreurs de syntaxe signifient quand à elles, que l'ordinateur ne peut même pas interpréter ce que nous essayons de lui dire. Souvent il va indiquer l'endroit à partir duquel il ne comprend plus (ici la sixième ligne), ce qui n'est pas toujours une information d'un grand secours. Habituellement le problème se résout en ajoutant quelques parenthèses et en supprimant les espaces avant les dites parenthèses. Le problème tient au fait qu'un ordinateur manque totalement de sens commun... mais je ne vais tout de même pas tenter de vous expliquer pourquoi l'ordinateur n'arrive pas à deviner vos intentions. Corrigeons simplement :

```
largeurLigne = 40
str = '--> texte <--'
puts(str.ljust( largeurLigne))
puts(str.center(largeurLigne))
puts(str.rjust( largeurLigne))
puts(str.ljust(largeurLigne/2) + str.rjust(largeurLigne/2))
```

```
--> texte <--
                --> texte <--
                                  --> texte <--
--> texte <--                     --> texte <--
```

On ne peut pas dire que ce code soit joli (du moins c'est ce que je pense), mais il fonctionne. (J'ai ajouté les parenthèses aux autres lignes pour supprimer les avertissements qui s'affichaient.)

Quelques petites choses à essayer

- Ecrivez un programme à la Louis de Funès patron. Il doit vous demander rudement ce que vous voulez. Quoique ce soit que vous puissiez répondre, il répètera votre question et vous éconduira. Par exemple, si vous tapez '`je voudrais une augmentation`', il vous répondra :

```
Mais qu'est ce que ça veux dire "je voudrai une augmentation"?!?
Viré, vous êtes viré!!
```

- Essayez maintenant d'écrire - pour vous perfectionner dans l'utilisation de `center`, `ljust` et `rjust` - un programme qui affiche une table des matières qui ressemble à ceci :

```
Table des matières

Chapitre 1 : Nombres                    page 1
Chapitre 2 : Lettres                   page 72
Chapitre 3 : Variables                page 118
```

Math Sup

(Cette section est totalement optionnelle. Elle suppose une bonne connaissance en mathématiques. Si vous n'êtes pas intéressé(e) vous pouvez passer directement au chapitre suivant sans problème. Cependant, un bref coup d'oeil à la partie traitant des nombres aléatoires ne vous fera aucun mal.)

Il n'y a pas autant de méthodes pour les nombres que pour les chaînes de caractères (ce qui ne veut pas dire que je les ai toutes en tête). Ici, nous allons voir le reste des méthodes arithmétiques, un générateur de nombres aléatoires, et l'objet Math, avec ses méthodes trigonométriques et transcendentales.

Plus d'arithmétique

Les deux autres méthodes arithmétiques sont **(exponentiation) et %(modulo). Ainsi si vous voulez dire "cinq au carré" en Ruby, vous devez l'écrire 5**2. Vous pouvez aussi utiliser des réels pour l'exposant, donc si vous souhaitez la racine carrée de 5, vous pouvez écrire 5**0.5. La méthode modulo donne le reste de la division par un nombre. Par exemple, si je divise 7 par 3, j'obtiens 2 et un reste de 1. Essayons cela dans un programme :

```
puts 5**2
puts 5**0.5
puts 7/3
puts 7%3
puts 365%7
```

```
25
2.23606797749979
2
1
1
```

La dernière ligne nous confirme qu'une année non bissextile a un certain nombre de semaines plus un jour. Ainsi si votre anniversaire tombe un mardi cette année, on vous le souhaitera un mercredi l'année prochaine. Vous pouvez aussi utiliser des réels avec la méthode modulo. La fonction modulo est généralisée de la seule façon sensée...mais je vous laisse vous amuser avec ça.

Il reste encore une méthode avant d'attaquer les nombres aléatoires : `abs`. Elle renvoie tout simplement la valeur absolue d'un nombre :

```
puts (5-2).abs
puts (2-5).abs
```

```
3
3
```

Nombres aléatoires

Ruby nous arrive avec un très joli générateur de nombre aléatoire. La méthode pour obtenir un nombre au hasard est `rand` (de random : hasard). Si vous appelez `rand` tel quel, vous obtiendrez un réel supérieur ou égal à 0.0 et inférieur à 1.0. Si vous passez à `rand` un entier (5 par exemple) en paramètre, il vous renverra un entier supérieur ou égal à 0 et strictement inférieur à 5. (soit cinq nombres possibles de 0 à 4).

Voyons `rand` en action. (Si vous recopiez cet exemple et l'exécutez, le résultat obtenu changera chaque fois. Vous ne doutiez pas que j'essayais moi-même ces programmes, n'est-ce pas?)

```
puts rand
puts rand
puts rand
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(1))
puts(rand(1))
puts(rand(1))
puts(rand
(99999999999999999999999999999999999999999999999999999999999999999999
99999999999999999999))
```

```
puts('La météo nous dit qu\'il y a '+rand(101).to_s+'% de chance
qu\'il pleuve,')
puts('mais il ne faut jamais écouter la météo.')
```

```
0.319318517809734
0.448052901308984
0.44638533401303
2
24
63
0
0
0
6033906975456789045532345676543122335776889888
La météo nous dit qu'il y a 14% de chance qu'il pleuve,
mais il ne faut jamais écouter la météo.
```

Notez que j'ai utilisé `rand(101)` pour obtenir des nombres de 0 à 100, et que `rand(1)` retourne toujours 0. L'incompréhension de l'intervalle de valeurs possibles en retour avec `rand`, est un des grands mystères que je connaisse; même chez les programmeurs professionnels; même dans les produits que vous pouvez acheter dans le commerce. J'ai même eu un lecteur de CD qui en mode aléatoire jouait toutes les plages sauf la dernière...(je me demande ce que ça aurait donné si j'avais introduit un CD avec une seule plage ?)

Parfois vous pouvez avoir besoin que `rand` retourne les mêmes nombres aléatoires dans la même séquence lors de deux exécutions différentes de votre programme. (Par exemple, lorsque j'utilise des nombres aléatoires pour créer un monde aléatoire dans un jeu informatique. Si je trouve un monde qui me plait beaucoup, peut-être souhaiterai-je y retourner, ou l'envoyer à un ami.) Pour obtenir ceci, vous devrez semer une graine ("seed"en anglais), ce que vous ferez avec `srand`. Comme ceci:

```
srand 1776
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts ''
srand 1776
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
puts(rand(100))
```

```
14
9
50
84
23

14
```

```
9
50
84
23
```

La même chose se produira chaque fois que vous sèmerez avec le même nombre. Si vous voulez obtenir de nouveau des nombres différents (comme si vous n'aviez jamais utilisé `srand`), il suffira d'appeler `srand 0`. Cela sèmera avec un nombre étrange, utilisant (parmi d'autres choses) l'horloge de votre ordinateur, et ceci jusqu'à la milliseconde.

L'objet Math

Regardons finalement l'objet `Math`. Nous pouvons tout aussi bien y plonger d'un coup :

```
puts (Math::PI)
puts (Math::E)
puts (Math.cos(Math::PI/3))
puts (Math.tan(Math::PI/4))
puts (Math.log(Math::E**2))
puts ((1 + Math.sqrt(5))/2)
```

```
3.14159265358979
2.71828182845905
0.5
1
2.0
1.61803398874989
```

La première chose que vous avez remarquée est certainement la notation **::**. Expliquer l'opérateur de champ (scope operator) (qui est ce qu'il est) est vraiment en dehors du, euh... champ de ce tutoriel. Bon, plus de jeux de mots. Je le jure. Il suffit de le dire, vous pouvez utiliser `Math::PI` pour ce que vous pensez qu'il est : la valeur de PI.

Comme vous pouvez le voir, Math a tout ce que vous attendez d'un calculateur scientifique décent. Et comme toujours, les réels sont réellement proches des bonnes réponses.

6.Contrôle de flux

Ahhh, le contrôle de flux. C'est là que les Athéniens s'atteignirent. Un titre mystérieux pour dire simplement que lorsque l'on programme il arrive un moment où il va falloir faire une chose ou une autre, aller à un endroit ou à un autre, bref bifurquer. Quoique ce chapitre soit plus court et plus simple que celui des méthodes, il nous ouvrira un monde plein de possibilités pour la programmation. Après ce chapitre, nous serons vraiment capables d'écrire des programmes interactifs; dans le passé nous avons construit des programmes qui disaient différentes choses en fonction de vos entrées au clavier, mais après ce chapitre ils feront vraiment d'autres choses. Mais avant d'en arriver là, nous devons être capables de comparer des objets dans nos programmes. Nous devons...

Méthodes de comparaison

Fonçons tête baissée dans cette partie afin d'en arriver à la prochaine section, les branchements, où tout le meilleur va arriver.
Bien, pour voir si un objet est plus grand ou plus petit qu'un autre, nous utilisons les méthodes > et < comme ceci :

```
puts 1 > 2
puts 1 < 2
```

```
false
true
```

`false` signifiant "faux" et `true` "vrai".

Pas de problème. De la même façon nous pouvons dire si un objet est plus grand que ou égal à un autre (ou inférieur ou égal) avec les méthodes >= et <=

```
puts 5 >= 5
puts 5 <= 4
```

```
true
false
```

Et pour finir, nous pouvons voir si deux objets sont égaux ou non en utilisant == (qui signifie "sont-ils égaux ?") et != (qui signifie "sont-ils différents ?"). Il est important de ne pas confondre = avec ==. = est pour dire à une variable de pointer vers un objet (assignement), et == pose la question : "est-ce que ces deux objets sont égaux ?"

```
puts 1 == 1
puts 2 != 1
```

```
true
true
```

Nous pouvons aussi bien comparer des chaines. Lorsque des chaines sont comparées, c'est leur ordre lexicographique qui entre en jeu, ce qui signifie l'ordre dans le dictionnaire. chat vient avant chien dans le dictionnaire, ainsi :

```
puts 'chat' < 'chien'
```

```
true
```

Il y a cependant un petit problème : les ordinateurs considèrent que les majuscules précèdent les minuscules dans les fontes qu'ils utilisent. (Ceci parce que dans les fontes, l'alphabet est en double et les majuscules sont placées avant). Cela signifie que 'Zoo' vient avant 'abat', et que si vous voulez qu'un mot figure en premier dans un dictionnaire, il ne faut pas oublier d'appliquer `downcase` (ou `upcase` ou `capitalize`) sur les deux mots avant de les comparer.

Une dernière remarque avant les branchements : les méthodes de comparaison ne nous donnent pas les chaines 'true' et 'false'; elles nous renvoient les objets spéciaux `true` et `false`. (Bien entendu, `true.to_s` nous donne 'true', c'est pour ça que `puts` affiche 'true'.) `true` et `false` sont le plus souvent utilisés dans...

les branchements

Les branchements sont un concept simple, mais néanmoins magnifique. En fait, il est si simple que je parie que je n'ai jamais eu à l'expliquer à quiconque; je vais juste vous le montrer :

```
puts 'Bonjour, quel est votre prénom ?'
nom = gets.chomp
puts 'Bonjour, ' + nom + '.'
if nom == 'Christian'
  puts 'Quel joli prénom vous avez !'
end
```

```
Bonjour, quel est votre prénom ?
Christian
Bonjour, Christian.
Quel joli prénom vous avez !
```

Mais si nous répondons par un prénom différent...

```
Bonjour, quel est votre prénom ?
Gaston
Bonjour, Gaston.
```

Le branchement c'est ça. Si ce qui suit le si (`if`) est vrai (`true`), le code situé entre le si (`if`) et la fin (`end`) est exécuté. Si ce qui suit le si (`if`) est faux (`false`), rien n'est exécuté. Clair et net !

J'ai indenté (mis en retrait du bord) le code entre le `if` et le `end` uniquement parce que je pense qu'il est plus facile ainsi de repérer le branchement. La plupart des programmeurs font comme cela, quelque soit leur langage de programmation. Cela ne paraît pas être d'une aide formidable dans cet exemple, mais quand les choses se compliquent, ça fait une grande différence.

Souvent, nous aimerions un programme qui fasse quelque chose si une expression est vraie, et une autre si elle est fausse. C'est pour cela que `else` (sinon) existe:

```
puts 'Je suis une voyante. Donnez-moi votre prénom :'
nom = gets.chomp
if nom == 'Christian'
  puts 'Je vois de grandes choses dans votre avenir.'
else
  puts 'Votre futur est... Oh, bon sang ! Regardez l'heure qu\'il
est !'
  puts 'je n\'ai plus le temps... désolée !'
end
```

```
Je suis une voyante. Donnez-moi votre prénom :
Christian
Je vois de grandes choses dans votre avenir.
```

Maintenant essayons un autre prénom...

```
Je suis une voyante. Donnez-moi votre prénom :
Gaston
Votre futur est... Oh, bon sang ! Regardez l'heure qu'il est !
je n'ai plus le temps... désolée !
```

Le branchement est comme une bifurcation dans le code : nous prenons un chemin pour les gens qui se prénomment Christian, et un autre dans le cas contraire.
Et tout comme les branches d'un arbre, vous pouvez avoir des branches qui ont elles-mêmes d'autres branches.(C'est ce que l'on appelle, dans le jargon des programmeurs, des "if" imbriqués.)

```
puts 'Bonjour, et bienvenue au club.'
puts 'Mon nom est Mademoiselle Longbec. Et votre nom est...'
nom = gets.chomp

if nom ==nom.capitalize
  puts 'Je vous en prie, asseyez-vous, ' + nom + '.'
else
  puts nom + '? Vous voulez dire ' + nom.capitalize + ', non ?'
  puts 'On ne vous a donc jamais appris à écrire correctement
votre nom ?'
  bis = gets.chomp
  if bis.downcase == 'oui'
    puts 'Hmmm... bon asseyez-vous !'
  else
    puts 'Dehors !!'
  end
end
```

```
Bonjour, et bienvenue au club.
Mon nom est Mademoiselle Longbec. Et votre nom est...
christian
christian? Vous voulez dire Christian, non ?
On ne vous a donc jamais appris à écrire correctement votre nom?
oui
Hmmm... bon asseyez-vous !
```

Mieux, je "capitalize" mon prénom...

```
Bonjour, et bienvenue au club.
Mon nom est Mademoiselle Longbec. Et votre nom est...
Christian
Je vous en prie, asseyez-vous, Christian.
```

Parfois il peut y avoir confusion et on ne sait plus où conduisent les `if`, `else` et `end`. Ce que je fais, c'est que je commence par les écrire et je comble ensuite avec le code. Ainsi le programme ci-dessus ressemble à ceci au début de sa conception :

```
puts 'Bonjour, et bienvenue au club.'
puts 'Mon nom est Mademoiselle Longbec. Et votre nom est...'
nom = gets.chomp

if nom ==nom.capitalize
else
end
```

Ensuite je remplis avec des commentaires, que la machine ignorera :

```
puts 'Bonjour, et bienvenue au club.'
puts 'Mon nom est Mademoiselle Longbec. Et votre nom est...'
nom = gets.chomp

if nom ==nom.capitalize
  # là elle est civile.
else
  # là elle est enragée.
end
```

Tout ce qui suit "#" est considéré comme un commentaire (sauf bien sûr si vous êtes dans une chaîne de caractères). Après ceci, je remplace les commentaires par du code actif. Certains laissent ces commentaires; personnellement je pense qu'une bonne écriture du code parle d'elle-même. Autrefois je faisais plus de commentaires, mais plus j'utilise Ruby moins j'en fais. Je trouve cela plus troublant qu'autre chose. Mais c'est une affaire de goût; vous trouverez petit à petit votre style. Ma prochaine étape ressemblait à celà :

```
puts 'Bonjour, et bienvenue au club.'
puts 'Mon nom est Mademoiselle Longbec. Et votre nom est...'
nom = gets.chomp
```

```
if nom ==nom.capitalize
  puts 'Je vous en prie, asseyez-vous, ' + nom + '.'
else
  puts nom + '? Vous voulez dire ' + nom.capitalize + ', non ?'
  puts 'On ne vous a donc jamais appris à écrire correctement
votre nom ?'
  bis = gets.chomp

  if bis.downcase == 'oui'

  else

  end
end
```

De nouveau j'ai écris les if, else, end en même temps. C'est vraiment très utile pour savoir où "je suis" dans le code. Cela facilite le travail parce que je peux me focaliser sur une petite partie, comme remplir le code entre le if et le else. Un autre avantage de procéder ainsi est que l'ordinateur comprend le programme à chaque étape. Chaque version inachevée du programme ci-dessus fonctionne sans problème. Ce ne sont pas des programmes terminés mais ce sont des programmes qui cependant fonctionnent. Ainsi je peux les tester au fur et à mesure de leur conception et visualiser plus facilement le travail achevé et celui qui reste à accomplir. Une fois tous les tests accomplis, je sais que j'ai terminé !

Ces trucs vous aideront à écrire des programmes possédant des branchements, mais ils vous aideront aussi avec l' autre principal type de contrôle de flux :

Les boucles

Souvent vous désirerez que votre ordinateur fasse la même chose encore et encore. Après tout ne dit-on pas que c'est dans les tâches répétitives que ces machines excellent?

Lorsque vous dites à votre ordinateur de répéter quelque chose, vous devez aussi penser à lui dire quand s'arrêter.Les ordinateurs ne se lassent jamais, aussi si vous omettez de leur dire de s'arrêter, ils ne le feront pas. Nous nous assurerons que cela n'arrivera pas, en disant à l'ordinateur de répéter une partie du programme tant (`while`) qu'une condition est vraie (true). Cela ressemble beaucoup au fonctionnement de if :

```
commande = ''

while commande != 'Ciao'
  puts commande
  commande = gets.chomp
end

puts 'Au revoir, à bientôt !'
```

```
Coucou ?
Coucou ?
Ohé !
```

```
Ohé !
Ciao
Au revoir, à bientôt !
```

Et voilà pour une boucle. (Vous avez noté la ligne blanche au début des réponses; c'est à cause du premier `puts`, avant le premier `gets`. Comment modifieriez-vous le programme pour supprimer cette ligne blanche ? Essayez! le programme doit fonctionner exactement comme avant, excepté la première ligne.)

Les boucles vous permettent de faire plein de choses intéressantes, comme je suis persuadé que vous allez en imaginer. Cependant elles peuvent créer des problèmes si vous vous trompez. Que faire si votre ordinateur entre dans une boucle infinie ? Si vous pensez que vous êtes dans ce cas, il suffit d'appuyer sur les touches "`Ctrl`" et "`C`".

Avant que de s'amuser avec les boucles, apprenons donc quelques petits trucs pour nous faciliter le travail.

Un petit peu de logique

Regardons un peu notre premier programme sur les branchements. Supposons que ma femme rentre à la maison, voit le programme, l'essaye, et qu'il ne lui dise pas qu'elle a un joli prénom ? Je ne voudrais pas heurter sa sensibilité (ni dormir sur le canapé !), aussi je le reécris comme ceci :

```
puts 'Bonjour, quel est votre prénom ?'
nom = gets.chomp
puts 'Bonjour, ' + nom + '.'
if nom == 'Christian'
  puts 'Quel joli prénom vous avez !'
else
  if nom =='Cathy'
    puts 'Quel joli prénom vous avez !'
  end
end
```

```
Bonjour, quel est votre prénom ?
Cathy
Bonjour, Cathy.
Quel joli prénom vous avez !
```

Bon, ça marche... mais ce n'est pas encore un joli programme. Pourquoi non ? parce que la meilleure règle que j'ai apprise en programmant a été la règle PDR : Pas De Répétitions. Je devrais certainement écrire un petit livre pour expliquer pourquoi c'est une bonne règle. Dans notre cas, nous répétons la ligne `puts 'Quel joli prénom vous avez !'`. Pourquoi est-ce si important ? Premièrement, que se passe-t-il si je me trompe en recopiant la ligne ? deuxièmement si je veux changer "joli" par "magnifique" dans les deux lignes ? Je suis paresseux si vous vous souvenez ? De façon simple, si je veux que mon programme réponde la même chose quand on lui dit 'Christian' ou 'Cathy', alors il doit réellement faire la même chose :

```
puts 'Bonjour, quel est votre prénom ?'
nom = gets.chomp
puts 'Bonjour, ' + nom + '.'
if (nom == 'Christian' or nom == 'Cathy')
  puts 'Quel joli prénom vous avez !'
end
```

```
Bonjour, quel est votre prénom ?
Cathy
Bonjour, Cathy.
Quel joli prénom vous avez !
```

C'est mieux. En fait pour obtenir ce résultat j'ai utilisé `or` (ou). Les autres opérateurs logiques sont `and` (et) et `not` (pas). Il vaut mieux utiliser des parenthèses lorqu'on travaille avec eux. Regardons un peu leur fonctionnement :

```
jeSuisChristian = true
jeSuisViolet = false
jaimeManger = true
jeSuisEnPierre = false

puts (jeSuisChristian and jaimeManger)
puts (jaimeManger and jeSuisEnPierre)
puts (jeSuisViolet and jaimeManger)
puts (jeSuisViolet and jeSuisEnPierre)
puts
puts (jeSuisChristian or jaimeManger)
puts (jaimeManger or jeSuisEnPierre)
puts (jeSuisViolet or jaimeManger)
puts (jeSuisViolet or jeSuisEnPierre)
puts
puts (not jeSuisViolet)
puts (not jeSuisChristian)
```

```
true
false
false
false

true
true
true
false

true
false
```

Le seul qui puisse vous jouer des tours est "or". En français, nous utilisons souvent "ou" en pensant "un ou l'autre, mais pas les deux". Par exemple, votre maman peut dire "Au dessert, il y a du gateau ou du flan." Elle n'envisage pas une seconde que vous puissiez vouloir des deux ! Un ordinateur au contraire, utilise "ou" en pensant "un ou l'autre, ou les deux." (Une autre façon de dire serait : "au moins un des deux est vrai") C'est pour cela que les ordinateurs sont plus amusants que les mamans !

Quelques petites choses à essayer

- Ecrivez un programme Mamie est sourde. Quoique vous disiez à Mamie (quoique vous tapiez plutôt), elle vous répond par "Hein ?! Parle plus fort !", jusqu'à ce que vous criiez (tapez en majuscule). Si vous criez, elle peut vous entendre (ou du moins c'est ce qu'elle pense) et vous répond, "non, pas depuis 1938 !" Pour rendre votre programme plus réaliste, Mamie doit répondre par une date différente à chaque fois; cela peut être aléatoirement entre 1930 et 1950. (Cette partie est optionnelle, et vous serez plus à l'aise après avoir lu la section de Ruby traitant du générateur de nombres aléatoires à la fin du chapitre sur les méthodes.) Vous ne pouvez pas cesser de répondre à Mamie jusqu'à ce que vous entriez Ciao.

  Remarque : n'oubliez pas chomp ! 'Ciao' avec un "Enter" n'est pas le même que sans!

  Remarque 2: Essayez d'imaginer quelle partie de votre programme doit se répéter encore et encore. Tout cela doit être dans votre boucle `while`.

- Améliorez votre programme sur Mamie qui est sourde : Que faire si Mamie ne veut pas que vous la quittiez ? Quand vous criez "Ciao", elle prétend ne pas vous entendre. Changez votre précédent programme de telle sorte que vous devez crier "Ciao" trois fois sur la même ligne pour qu'elle vous entende. Testez votre programme : si vous criez trois fois "Ciao", mais pas sur la même ligne, vous devrez continuer à répondre à Mamie.

- Les années bissextiles. Ecrivez un programme qui vous demande une année de départ et une année de fin, puis affichez toutes les années bissextiles entre ces deux limites (en les incluant si tel est le cas). Les années bissextiles sont des années divisibles par quatre (comme 1984 et 2004). Cependant, les années divisibles par 100 ne sont pas bissextiles (comme 1800 et 1900) sauf si elles sont divisibles par 400 (comme 1600 et 2000).(Oui cela prête à une jolie confusion, mais pas autant que d'avoir juillet au milieu de l'hiver, ce qui peut éventuellement arriver.)

Quand vous aurez terminé tout cela, prenez donc une pause ! Vous avez déjà appris pas mal de choses. Félicitations ! Réalisez-vous le nombre de choses que vous pouvez demander à un ordinateur maintenant ? Encore quelques chapitres et vous serez capable de programmer n'importe quoi. Sérieusement ! Regardez tout ce que vous pouvez faire depuis que vous connaissez les boucles et branchements.

Nous allons maintenant nous attaquer à une nouvelle sorte d'objet qui garde une trace, sous forme de liste, des autres objets : les tableaux.

7. Tableaux et itérateurs

Ecrivons un programme qui nous demande d'entrer autant de mots que nous voulons (un mot par ligne, jusqu'à ce que nous tapions la touche "Enter" sur une ligne vide), et qui nous renvoie ces mots par ordre alphabétique. D'accord ?

Bon... en premier -heu... Hummm... Bon, nous devons... Hummm...

Vous savez, je ne pense pas que nous puissions faire cela. Nous avons besoin d'un moyen de stocker un certain nombre de mots, et d'un autre pour en garder la trace, de telle sorte qu'il n'y ait pas de mélange avec les autres variables. Nous avons besoin de les garder dans une sorte de liste. Nous avons besoin de tableaux.

Un tableau est tout simplement une liste dans votre ordinateur. Chaque élément de la liste agit comme une variable : vous pouvez voir quelle sorte d'objet pointe sur un élément de tableau, et nous pouvons le faire pointer vers un objet différent. Jetons un coup d'oeil sur quelques tableaux :

```
[]
[5]
['Bonjour', 'Adieu']

parfum = 'Vanille'      # ceci n'est pas un tableau, bien sûr...
[89.9, parfum, [true, false]]  # mais ceci, oui.
```

Pour commencer nous avons un tableau vide, puis un tableau contenant un simple entier, puis un tableau avec deux chaines de caractères. Ensuite vient un assignement; puis un tableau contenant trois objets, le dernier étant lui-même un tableau `[true, false]`. Souvenez-vous, les variables ne sont pas des objets, notre dernier tableau pointe donc réellement sur un réel, une chaîne, et un tableau. Même si nous faisons pointer `parfum` sur quelque chose d'autre, cela ne change pas le tableau.

Pour nous aider à trouver un objet particulier dans un tableau, chaque élément correspond à un index. Les programmeurs (et, accidentellement la plupart des mathématiciens) commencent à compter à partir de zéro, ainsi le premier élément dans le tableau est l'élément zéro. Ce qui suit nous montre comment faire référence à des objets dans un tableau :

```
noms = ['Ada', 'Belle', 'Christiane']

puts noms
puts noms[0]
puts noms[1]
puts noms[2]
puts noms[3]    # ceci est hors indice.
```

```
Ada
Belle
Christiane
Ada
Belle
```

```
Christiane
nil
```

Bon, nous voyons que puts noms affiche chaque nom du tableau `noms`. Puis nous utilisons `puts noms[0]` pour afficher le "premier" nom du tableau, et `puts noms[1]` pour le second... Je suis sûr que cela vous parait confus, mais vous devez l'utiliser ainsi. Vous n'avez qu'à réellement commencer à compter en pensant que pour compter on débute à zéro, et arrêter d'utiliser des mots tels que "premier" et "second". Si vous jouez aux courses, ne parlez plus de la "première"; parlez de la course zéro (et dans votre tête pensez course[0]). Vous avez cinq doigts à votre main droite et ils sont numérotés 0, 1, 2, 3 et 4. Vous saurez que vous y êtes arrivé quand vous direz "le quatrième élément du tableau", et que vous penserez réellement monTableau[4]. Cela apparaîtra vraisemblablement lorsque vous parlerez aussi du "zéroième". Si, si, le mot existe; demandez donc à un programmeur ou à un mathématicien.

Pour finir, nous avons essayé `puts nom[3]`, juste pour voir ce que cela donnait. Vous attendiez une erreur, n'est-ce pas ? Parfois vous posez une question, votre question n'a pas de sens (du moins pour votre ordinateur); c'est là que vous obtenez une erreur. Parfois, cependant, vous pouvez poser une question dont la réponse est “rien”. Quel est le troisième élément du tableau ? rien. Qu'est-ce que `noms[3]` ? `nil` : c'est la façon de Ruby de dire "rien". `nil` est un objet spécial qui signifie "plus d'autre objet".

Si toute cette jolie numérotation d’éléments de tableaux vous rend perplexe , ne vous effrayez pas ! Souvent nous pouvons l’éviter complètement en utilisant diverses méthodes des tableaux, comme celle-ci :

La méthode `each`

`each` nous permet de faire quelque chose (tout ce que nous voulons) à each (chaque) objet du tableau. Ainsi si nous souhaitons dire quelque chose d'agréable pour chaque langage du tableau ci-dessous, nous ferons comme ceci :

```
langages = ['Anglais', 'Allemand', 'Ruby']

langages.each do |langue|
  puts 'J\' aime parler en ' + langue + ' !'
  puts 'Pas vous ?'
end

puts 'Et tant pis pour le C++ !'
puts '...'
```

```
J' aime parler en Anglais !
Pas vous ?
J' aime parler en Allemand !
Pas vous ?
J' aime parler en Ruby !
Pas vous ?
Et tant pis pour le C++ !
...
```

Que se passe-t-il au juste ? Nous sommes capables d'aller d'objet en objet dans le tableau sans utiliser aucun nombre, ce qui est parfait.
Traduit en français, le programme ci-dessus équivaut à : pour chaque (`each`) objet dans `langages`, pointez la variable `langue` vers l'objet et puis faire (`do`) tout ce que je vous demande, jusqu'à ce que vous arriviez à la fin (`end`). (Juste une remarque; comme vous le savez C++ est un autre langage de programmation. Nettement plus difficile à apprendre que Ruby. En général un programme en C++ est plus long qu'un programme en Ruby pour faire la même chose.)
En votre for intérieur vous devez vous dire, "C'est la même chose que les boucles que nous avons apprises il y a peu de temps." Oui, ça y ressemble. Une différence importante est que la méthode `each` est uniquement cela : une méthode. `while` et `end` (ou mieux `do`, `if`, `else` et tous les autres mots en bleu) ne sont pas des méthodes. Ce sont une partie fondamentale du langage Ruby, tout comme = et les parenthèses.

Mais pas `each`; `each` est seulement une méthode de tableaux. Les méthodes qui agissent comme les boucles sont souvent appellées des itérateurs.

Une chose à noter au sujet des itérateurs, c'est qu'ils sont toujours suivis de `do`...`end`. `while` et `if` n'ont jamais un `do` à côté d'eux; nous n'utilisons `do` qu'avec les itérateurs.

Voici un ingénieux petit itérateur, mais il ne s'agit pas d'une méthode de tableaux... c'est une méthode d'entiers !

```
3.times do
  puts 'Hip, Hip, Hip, Hourra !'
end
```

```
Hip, Hip, Hip, Hourra !
Hip, Hip, Hip, Hourra !
Hip, Hip, Hip, Hourra !
```

Quelques petites choses à essayer

- Ecrivez le programme dont nous parlions au tout début de ce chapitre.

Remarque : il existe une très appréciable méthode de tableaux qui vous donnera une version triée d'un tableau : `sort`. Utilisez-la !

- Essayez d'écrire le programme ci-dessus sans utiliser la méthode `sort`. La plupart du temps la programmation consiste à résoudre des problèmes, aussi mettez en pratique tout ce que vous pouvez !

D'autres méthodes de tableaux

Nous venons d'apprendre `each` et `sort`,mais il existe de nombreuses autres méthodes de tableaux... la plupart d'entre elles sont des méthodes de chaînes! En fait, certaines (comme `length`, `reverse`, + et *) fonctionnent comme pour des chaînes, à part qu'elles agissent sur les éléments du tableau plutôt que sur les lettres d'une chaîne. D'autre, comme `last` et `join`, sont spécifiques aux tableaux. D'autres encore, comme `push` et

pop, changent même le tableau. Et, exactement comme pour les méthodes de chaînes, vous n'avez pas à vous souvenir de toutes ces méthodes, il vous suffit de savoir où retrouver leur description (ici par exemple).

```
aliments = ['artichaut', 'brioche', 'caramel']

puts aliments
puts
puts aliments.to_s
puts
puts aliments.join(', ')
puts
puts aliments.join(' :) ') + ' 8)'

200.times do
  puts []
end
```

```
artichaut
brioche
caramel

artichautbriochecaramel

artichaut, brioche, caramel

artichaut :) brioche :) caramel 8)
```

Comme vous pouvez le voir, puts traite les tableaux différemment des autres objets : puts est appelé pour chaque élément du tableau. C'est pour cette raison que “putser” un tableau vide 200 fois ne fait rien; le tableau pointant sur rien, il n’y a donc rien pour puts. (Ne rien faire 200 fois est comme ne rien faire du tout). Essayez de "putser" un tableau contenant d'autres tableaux; est-ce que le résultat est conforme à votre attente ?

Avez-vous remarqué que je laisse des chaînes vides lorsque je veux afficher une ligne blanche ? c'est la même chose.

Jetons maintenant un coup d'oeil à push ,pop et last. Les méthodes push et pop sont en quelque sorte opposées comme + et -, push ajoute un objet à la fin de votre tableau, et pop enlève le dernier objet du tableau en affichant ce qu'il était. last est similaire à pop en ce sens qu'il indique ce qu'il y a à la fin du tableau, mais sans rien supprimer. Répétons-le, push et pop modifient réellement le tableau:

```
favoris =[]
favoris.push '2001 l\'Odyssée de l\'espace'
favoris.push 'Docteur Folamour'

puts favoris[0]
puts favoris.last
puts favoris.length
puts favoris.pop
puts favoris
```

```
puts favoris.length
```

```
2001 l'Odyssée de l'espace
Docteur Folamour
2
Docteur Folamour
2001 l'Odyssée de l'espace
1
```

Quelques petites choses à essayer

- Réécrivez votre table des matières (du chapitre sur les méthodes). Débutez le programme par un tableau contenant toutes les informations de la table des matières (noms des chapitres, numéros de pages, etc...). Puis extrayez les informations dans une magnifique Table des Matières formatée.

Puisque nous avons appris un certain nombre de méthodes, il est temps à présent d'apprendre à fabriquer les nôtres.

8. Ecrire vos propres méthodes

Comme nous l'avons vu, les boucles et les itérateurs nous permettent de faire la même chose (exécuter le même code) encore et encore. Cependant, parfois nous voudrions faire une chose un certain nombre de fois mais à partir d'endroits différents du programme. Par exemple, imaginons que nous écrivons un questionnaire pour un étudiant en psychologie. Ce sont d'ailleurs des étudiants en psychologie que je connais qui m'ont donné le questionnaire qui ressemblait à peu près à ceci :

```
puts 'Bonjour, et merci de prendre le temps'
puts 'de m\'aider dans cette expérience. Mon expérience'
puts 'a trait à la perception par les gens de la '
puts 'nourriture Mexicaine. Pensez uniquement à la'
puts 'nourriture Mexicaine et essayez de répondre'
puts 'honnêtement à toutes les questions, soit par'
puts 'un "oui" soit par un "non". Mon expérience n\'a'
puts 'absolument rien à voir avec le pipi au lit.'
puts

#  Nous posons ces questions, mais nous en ignorons les réponses.

bonneReponse = false
while (not bonneReponse)
  puts 'Aimez-vous manger des tacos?'
  reponse = gets.chomp.downcase
  if (reponse == 'oui' or reponse == 'non')
    bonneReponse = true
  else
    puts 'S\'il vous plait, répondez par "oui" ou par "non".'
  end
end

bonneReponse = false
while (not bonneReponse)
  puts 'Aimez-vous les burritos?'
  reponse = gets.chomp.downcase
  if (reponse == 'oui' or reponse == 'non')
    bonneReponse = true
  else
    puts 'S\'il vous plait, répondez par "oui" ou par "non".'
  end
end

#  C'est à cette réponse que nous devons prêter attention.

bonneReponse = false
while (not bonneReponse)
  puts 'Faites-vous pipi au lit?'
  reponse = gets.chomp.downcase
  if (reponse == 'oui' or reponse == 'non')
    bonneReponse = true
```

```
    if reponse == 'oui'
      pipi = true
    else
      pipi = false
    end
  else
    puts 'S\'il vous plait, répondez par "oui" ou par "non".'
  end
end

bonneReponse = false
while (not bonneReponse)
  puts 'Aimez-vous les chimichangas?'
  reponse = gets.chomp.downcase
  if (reponse == 'oui' or reponse == 'non')
    bonneReponse = true
  else
    puts 'S\'il vous plait, répondez par "oui" ou par "non".'
  end
end

puts 'Plus que quelques questions...'

bonneReponse = false
while (not bonneReponse)
  puts 'Aimez-vous les sopapillas?'
  reponse = gets.chomp.downcase
  if (reponse == 'oui' or reponse == 'non')
    bonneReponse = true
  else
    puts 'S\'il vous plait, répondez par "oui" ou par "non".'
  end
end

#  Posez d'autres questions sur les plats mexicains si vous
voulez.

puts
puts 'Compte-rendu:'
puts 'Merci d\'avoir pris le temps de m\'aider'
puts 'dans cette expérience. En fait, cette expérience'
puts 'n\'a rien à voir avec la nourriture Mexicaine.'
puts 'C\'est une étude sur le pipi au lit. La nourriture'
puts 'Mexicaine n\'était là que pour détourner l\'attention'
puts 'en espérant que vous répondiez honnêtement. Merci encore.'
puts
puts pipi
```

```
Bonjour, et merci de prendre le temps
de m'aider dans cette expérience. Mon expérience
a trait à la perception par les gens de la
nourriture Mexicaine. Pensez uniquement à la
nourriture Mexicaine et essayez de répondre
honnêtement à toutes les questions, soit par
```

```
un "oui" soit par un "non". Mon expérience n'a
absolument rien à voir avec le pipi au lit.

Aimez-vous manger des tacos?
oui
Aimez-vous les burritos?
oui
Faites-vous pipi au lit?
Plaît-il?
S'il vous plait, répondez par "oui" ou par "non".
Faites-vous pipi au lit?
NON
Aimez-vous les chimichangas?
oui
Plus que quelques questions...
Aimez-vous les sopapillas?
oui

Compte-rendu:
Merci d'avoir pris le temps de m'aider
dans cette expérience. En fait, cette expérience
n'a rien à voir avec la nourriture Mexicaine.
C'est une étude sur le pipi au lit. La nourriture
Mexicaine n'était là que pour détourner l'attention
en espérant que vous répondiez honnêtement. Merci encore.

false
```

Ce fut un long programme, avec son lot de répétitions. (Toutes les sections de code concernant les questions sur la nourriture Mexicaine étaient identiques, et seule la question sur le pipi au lit différait sensiblement.) Les répétitions sont une mauvaise chose. Seulement nous ne pouvons tout mettre dans une grande boucle ou des itérateurs, parce que parfois nous avons des choses à faire entre deux questions. Dans de telles situations, il vaut mieux écrire une méthode. Et voici comment :

```
def meugler
  puts 'Meuhhh...'
end
```

Euh... notre programme ne meugle pas. Pourquoi ? Parce que nous ne lui avons pas dit de le faire. Nous lui avons dit comment meugler, mais pas de le faire. Essayons encore une fois :

```
def meugler
  puts 'Meuhhh...'
end

meugler
meugler
puts 'quack ! quack !'
meugler
```

```
meugler
```

```
Meuhhh...
Meuhhh...
quack ! quack !
Meuhhh...
Meuhhh...
```

Ah ! C'est mieux. (Juste au cas où vous ne parleriez pas anglais, c'est un canard anglais au milieu du programme. Dans les pays anglo-saxons les canards font " quack ! quack !".)

Nous définissons la méthode `meugler`. (Les noms de méthodes, comme ceux des variables, débutent par une minuscule. Il y a quelques exceptions, comme + ou ==). Mais les méthodes ne sont-elles pas toujours associées à des objets ? La réponse est oui, et dans ce cas (comme avec `puts` et `gets`), la méthode est simplement liée à l'objet représentant le programme tout entier. Dans le prochain chapitre nous verrons comment ajouter des méthodes aux autres objets. Mais pour le moment :

Paramètres de méthodes

Vous avez certainement noté que certaines méthodes (comme `gets, to_s, reverse`), s'appellent à partir d'un objet. D'autres méthodes cependant, (comme +, -, `puts`...) nécessitent un paramètre pour indiquer à l'objet comment exécuter la méthode. Par exemple vous ne dites pas simplement 5+, n'est-ce pas ? Cela voudrait dire que vous appellez 5 pour ajouter, mais ajouter quoi ?

Pour ajouter un paramètre à la méthode `meugler` (pour indiquer le nombre de meuglements par exemple), nous ferons comme ceci :

```
def meugler nombreMeuh
  puts 'Meuhhh...'* nombreMeuh
end

meugler 3
puts 'coin-coin'
meugler  # Ici l'oubli du paramètre va déclencher une erreur.
```

```
Meuhhh...Meuhhh...Meuhhh...
coin-coin
p45.rb:7 :in 'meugler':wrong # of arguments (0 for 1)(Argument
Error)
                                from p45.rb:7
```

`nombreMeuh` est une variable qui pointe sur le paramètre passé à la méthode. Je le répète, parce que c'est un peu confus : `nombreMeuh` est une variable qui pointe sur le paramètre. Ainsi si je tape `meugler 3`, le paramètre est 3, et la variable `nombreMeuh` pointe sur 3.

Comme vous le voyez un paramètre est maintenant requis. Après tout, par quoi 'Meuh-hh...' est-il supposé être multiplié si on ne donne pas un paramètre? Votre "pôvre" ordinateur ne le sait pas.

Si les objets sont en Ruby comme des noms en français, et les méthodes comme des verbes, alors vous pouvez imaginer les paramètres comme des adverbes (comme avec meugler, où le paramètre nous disait comment faire meugler) ou parfois comme des compléments d'objets directs (comme avec `puts`, où le paramètre est ce qu'il faut "putser").

Variables locales

Dans le programme suivant il y a deux variables :

```
def doubler nombre
  deuxFois = nombre * 2
  puts nombre.to_s + ' doublé fait ' + deuxFois.to_s
end

doubler 44
```

```
44 doublé fait 88
```

Les variables sont `nombre` et `deuxFois`. Elles sont toutes deux à l'intérieur de la méthode `doubler`. Ce sont (comme celles que nous avons déjà vues) des variables locales. Cela signifie qu'elles vivent à l'intérieur de la méthode, et ne peuvent en sortir. Si vous essayez, vous obtiendrez une erreur :

```
def doubler nombre
  deuxFois = nombre * 2
  puts nombre.to_s + ' doublé fait ' + deuxFois.to_s
end

doubler 44
puts deuxFois.to_s
```

```
44 doublé fait 88
p46.rb:7: undefined local variable or method `deuxFois' for #<Ob-
ject:0x258d810> (NameError)
```

Variable locale indéfinie...En fait nous avons défini cette variable, mais elle n'est pas locale à l'endroit où nous voulons l'utiliser. Elle est locale à la méthode.

Cela peut paraître comme un inconvénient, mais pas du tout. Puisque cela signifie que vous n'avez pas accés aux variables internes des méthodes, cela veut dire aussi que ces dernières n'ont pas accés aux vôtres, et ne peuvent donc les "bousiller" :

```
def petitePeste var
  var = nil
  puts 'Ha, Ha ! : votre variable ne vaut plus rien !'
end
```

```
var = 'Personne ne touche à ma variable !'
petitePeste var
puts var
```

```
Ha, Ha ! : votre variable ne vaut plus rien !
Personne ne touche à ma variable !
```

Il y a en fait , dans ce petit programme, deux variables appellées `var` : une à l'intérieur de la méthode `petitePeste` et l'autre externe à elle. Quand nous appelons `petitePeste var`, nous ne faisons que passer une chaîne d'une `var` à l'autre, de telle sorte que les deux pointent sur la même chaîne. La petitePeste peut donc faire pointer sa variable locale sur `nil`, cela ne concerne pas la variable du même nom à l'extérieur. Il est tout de même déconseillé de se prêter à ce petit jeu de noms identiques...

Valeurs de retour

Vous avez peut-être noté que certaines méthodes vous retournaient quelque chose lorsque vous les appeliez. Par exemple `gets` vous renvoie une chaîne (celle que vous avez entrée), et la méthode + dans 5 + 3 (qui est en réalité 5.+(3)) renvoie 8. Les méthodes arithmétiques appliquées aux nombres retournent des nombres,et les méthodes arithmétiques pour les chaînes renvoient des chaînes.

Il est important de comprendre la différence qu'il y a entre les méthodes retournant une valeur à l'endroit d'où elles ont été appelées, et votre programme qui affiche des informations sur l'écran, comme le fait `puts`. Notez que 5+3 retourne 8; il n'affiche pas 8.

Mais que retourne donc `puts` ? Nous n'avons jamais fait attention à cela, mais il n'est pas trop tard pour regarder :

```
valeurRetour = puts 'Ce puts retourne : '
puts valeurRetour
```

```
Ce puts retourne :
nil
```

Si la variable `valeurRetour` pointe sur `nil` (2ème ligne de réponse) c'est que `puts` a retourné `nil` dans la première ligne. `puts` renvoie toujours une valeur `nil`. Toute méthode se doit de retourner quelque chose, même s'il ne s'agit que de `nil`.

Faites une petite pause et écrivez un programme pour voir ce que retourne la méthode `meugler`.

Etes-vous surpris(e) ? Bon, voici comment ça marche : la valeur retournée par une méthode est tout simplement la dernière ligne de la méthode. Pour vous en convaincre, remplacez `puts` par l'affectation à une variable interne (`var1 = 'Meuhhh...'* nombreMeuh`) et réexécutez le programme. Si nous voulons que nos méthodes retournent toutes la même chaîne '`caramel`', il nous suffit de mettre cette chaîne en dernière ligne :

```
def meugler nombreMeuh
  puts 'Meuhhh...'* nombreMeuh
```

```
'Caramel'
end

x = meugler 2
puts x
```

```
Meuhhh...Meuhhh...
Caramel
```

Bien, essayons de nouveau l'expérience de psychologie, mais laissons cette fois une méthode poser les questions pour nous. Il sera nécessaire de considérer la question comme un paramètre de méthode, et d'avoir comme valeur de retour `true` (vrai) si la réponse "oui" et `false` (faux) si la réponse est "non". (Même si la plupart du temps la réponse nous est indifférente, c'est seulement une bonne idée que de faire retourner une valeur par nos méthodes. Et nous pourrons aussi l'utiliser pour la question sur le pipi au lit).J'ai raccourci l'introduction et le compte-rendu pour faciliter la lecture :

```
def interroge question
  bonneReponse = false
  while (not bonneReponse)
    puts question
    reponse = gets.chomp.downcase
    if (reponse == 'oui' or reponse == 'non')
      bonneReponse = true
      if reponse == 'oui'
        retour = true
      else
        retour =false
      end
    else
      puts 'S\'il vous plait, répondez par "oui" ou par "non".'
    end
  end
  retour  # C'est ce que la méthode retourne. (true ou false)
end


puts 'Bonjour, et merci de prendre le temps...'
puts

interroge 'Aimez-vous manger des tacos?'
interroge 'Aimez-vous les burritos?'
pipi = interroge 'Faites-vous pipi au lit?'
interroge 'Aimez-vous les chimichangas?'
interroge 'Aimez-vous les sopapillas?'
interroge 'Aimez-vous les tamales?'
puts 'Plus que quelques questions...'
interroge 'Aimez-vous boire de l\'horchata?'
interroge 'Aimez-vous les flotas?'

puts
```

```
puts 'Compte-rendu:'
puts 'Merci d\'avoir pris le temps de m\'aider...'
puts
puts pipi
```

```
Bonjour, et merci de prendre le temps...'

Aimez-vous manger des tacos?
oui
Aimez-vous les burritos?
oui
Faites-vous pipi au lit?
Plait-il?
S'il vous plait, répondez par "oui" ou par "non".
NON
Aimez-vous les chimichangas?
oui
Aimez-vous les sopapillas?
oui
Aimez-vous les tamales?
oui
Plus que quelques questions...
Aimez-vous boire de l'horchata?
oui
Aimez-vous les flotas?
oui

Compte-rendu:
Merci d'avoir pris le temps de m'aider...

false
```

Pas mal, non? Nous avons la possibilité d'ajouter des questions (et c'est maintenant facile) et notre programme est nettement plus court! C'est une nette amélioration, un rêve de programmeur paresseux.

Encore un long exemple

Je pense à un autre exemple de méthode susceptible de nous aider. Nous l'appellerons `nombreBelge`. Elle prend un nombre en paramètre, par exemple 22, et retourne la version en toutes lettres (soit "vingt-deux"). Pour le moment cantonnons-nous aux nombres de 0 à 100.

(Remarque : cette méthode utilise une nouvelle astuce en forçant prématurément une valeur de retour à l'aide du mot-clé `return`, et en introduisant un nouveau test de branchement : `elsif` (sinon si). Cela est suffisamment clair dans l'exemple qui suit)

```
def nombreBelge nombre
  #  Nous voulons seulement des nombres de 0 à 100.
  if nombre < 0
    return 'SVP entrez un nombre = à zéro ou supérieur.'
  end
```

```
if nombre > 100
  return 'SVP entrez un nombre = à 100 ou inférieur.'
end

chaineNombre = ''  #  C'est la chaîne que nous retournerons.
                   #  "gauche" indique combien de milliers, de
                   #  centaines ou de dizaines il y a.
#  "ecrire" est la partie que nous écrivons à droite.
#  ecrire et gauche... ça va ?  :)
gauche  = nombre
ecrire = gauche/100    # Combien de centaines doit-on écrire à
                       # gauche?
gauche  = gauche - ecrire*100  #  On enlève ces centaines.

if ecrire > 0
  return 'cent'
end

ecrire = gauche/10     #  Combien de dizaines doit-on écrire à
                       #  gauche?
gauche  = gauche - ecrire*10  #  On soustrait ces dizaines.

if ecrire > 0
  if ecrire == 1  #  Ehh...
    #  Puisque nous ne pouvons écrire "dix-deux" à la place
    #  de douze nous devons faire une exception pour ces
    #  nombres.
    if    gauche == 0
      chaineNombre = chaineNombre + 'dix'
    elsif gauche == 1
      chaineNombre = chaineNombre + 'onze'
    elsif gauche == 2
      chaineNombre = chaineNombre + 'douze'
    elsif gauche == 3
      chaineNombre = chaineNombre + 'treize'
    elsif gauche == 4
      chaineNombre = chaineNombre + 'quatorze'
    elsif gauche == 5
      chaineNombre = chaineNombre + 'quinze'
    elsif gauche == 6
      chaineNombre = chaineNombre + 'seize'
    elsif gauche == 7
      chaineNombre = chaineNombre + 'dix-sept'
    elsif gauche == 8
      chaineNombre = chaineNombre + 'dix-huit'
    elsif gauche == 9
      chaineNombre = chaineNombre + 'dix-neuf'
    end
    #  Puisque nous avons déjà tenu compte du premier chiffre,
    #  nous n'avons rien à écrire à gauche.
    gauche = 0
  elsif ecrire == 2
    chaineNombre = chaineNombre + 'vingt'
```

```
  elsif ecrire == 3
    chaineNombre = chaineNombre + 'trente'
  elsif ecrire == 4
    chaineNombre = chaineNombre + 'quarante'
  elsif ecrire == 5
    chaineNombre = chaineNombre + 'cinquante'
  elsif ecrire == 6
    chaineNombre = chaineNombre + 'soixante'
  elsif ecrire == 7
    chaineNombre = chaineNombre + 'septante'
  elsif ecrire == 8
    chaineNombre = chaineNombre + 'huitante'
  elsif ecrire == 9
    chaineNombre = chaineNombre + 'nonante'
  end

  if gauche > 0
    chaineNombre = chaineNombre + '-'
  end
end

ecrire = gauche  #  Combien d'unités à écrire à gauche?
gauche  = 0      #  On les enlève et il ne reste rien.

if ecrire > 0
  if    ecrire == 1
    chaineNombre = chaineNombre + 'un'
  elsif ecrire == 2
    chaineNombre = chaineNombre + 'deux'
  elsif ecrire == 3
    chaineNombre = chaineNombre + 'trois'
  elsif ecrire == 4
    chaineNombre = chaineNombre + 'quatre'
  elsif ecrire == 5
    chaineNombre = chaineNombre + 'cinq'
  elsif ecrire == 6
    chaineNombre = chaineNombre + 'six'
  elsif ecrire == 7
    chaineNombre = chaineNombre + 'sept'
  elsif ecrire == 8
    chaineNombre = chaineNombre + 'huit'
  elsif ecrire == 9
    chaineNombre = chaineNombre + 'neuf'
  end
end

if chaineNombre == ''
  #  Le seul cas où "chaineNombre" puisse être vide
  #  est quand "nombre" est à 0.
  return 'zéro'
end

#  Si nous sommes arrivés ici, c'est que nous avons un nombre
```

```
  #  compris entre 0 et 100, nous devons donc retourner
  #  "chaineNombre".
  chaineNombre
end

puts nombreBelge(  0)
puts nombreBelge(  9)
puts nombreBelge( 10)
puts nombreBelge( 11)
puts nombreBelge( 17)
puts nombreBelge( 32)
puts nombreBelge( 88)
puts nombreBelge( 99)
puts nombreBelge(100)
```

```
zéro
neuf
dix
onze
dix-sept
trente-deux
huitante-huit
nonante-neuf
cent
```

Bon, il y a quelques toutes petites choses que je n'aime pas dans ce programme. Premièrement il y a trop de répétitions. Deuxièmement il n'accepte pas les nombres supérieurs à 100. Troisièmement il y a trop de cas spéciaux, trop de `return`s. Utilisons quelques tableaux et essayons de nettoyer un peu tout ça :

```
def nombreBelge nombre
  if nombre < 0  #  Pas de nombres négatifs.
    return 'SVP entrez un nombre positif.'
  end
  if nombre > 1999
    return 'SVP entrez un nombre inférieur à 2000.'
  end
  if nombre == 0
    return 'zéro'
  end

  #  Plus de cas spéciaux (ou presque...), plus de return.

  chaineNombre= ''  #  C'est la chaîne de retour.

  unite= ['un','deux', 'trois', 'quatre', 'cinq',
          'six', 'sept', 'huit', 'neuf']
  dizaine = ['dix', 'vingt', 'trente', 'quarante', 'cinquante',
             'soixante', 'septante', 'huitante', 'nonante']
  onzadixneuf = ['onze', 'douze', 'treize', 'quatorze',
                 'quinze', 'seize', 'dix-sept', 'dix-huit',
```

```
                'dix-neuf']

#  "gauche" indique combien de dizaines, centaines etc...
#  "ecrire" est la partie que nous écrivons à droite.
#  ecrire et gauche... ça va ?  :)
gauche = nombre
ecrire = gauche/100          # Combien de centaines à écrire?
gauche =gauche- ecrire*100   #  On soustrait ces centaines.

if ecrire > 0
  #  Ici se trouve la subtilité :
  centaines  = nombreBelge ecrire
  if centaines == 'un'    # On ne dit pas "un cent" mais
                          # "cent" tout court.
    centaines = ''
    chaineNombre = 'cent'
  elsif
    chaineNombre = chaineNombre + centaines + ' cent'
  end
  #  C'est ce qu'on appelle la "récursivité".
  #  De quoi s'agit-il au juste ?
  #  Je dis à cette méthode de s'appeler elle-même,
  #  mais avec "ecrire" au lieu de "nombre".
  #  Souvenez-vous que "ecrire" est (en ce moment)
  #  le nombre de centaines que nous avons à écrire.
  #  Ensuite nous ajoutons "centaines" à "chaineNombre",
  #  nous ajoutons la chaîne 'cent' après.
  #  Ainsi, par exemple, si nous avons appelé nombreBelge
  #  avec 1999 (tel que "nombre" = 1999),
  #  à ce point "ecrire" vaudra 19, et"gauche" vaudra 99.
  #  La chose la plus simple à présent est de laisser
  #  nombreBelge ecrire le 'dix-neuf' pour nous, puis
  #  d'écrire 'cent', et nombreBelge ecrira 'nonante-neuf'.

  if gauche> 0
    #  Nous ne voulons pas écrire 'deux centcinquante-un'...
    chaineNombre = chaineNombre + ' '
  end
end

ecrire = gauche/10           #  Combien de dizaines ?
gauche =gauche- ecrire*10    #  On enlève les dizaines.

if ecrire > 0
  if ((ecrire == 1) and (gauche > 0))
      #  Puisque nous ne pouvons écrire "dix-deux" au lieu de
      # "douze", nous devons faire encore une exception.

    chaineNombre = chaineNombre + onzadixneuf[gauche-1]

      #  "-1" parce que onzadixneuf[3] est 'quatorze', et non
      # 'treize'. Puisque nous avons déjà tenu compte du
      # premier chiffre, nous n'avons rien à écrire à gauche.
```

```
     gauche= 0
    else
      chaineNombre = chaineNombre + dizaine[ecrire-1]
        # "-1" parce que dizaine[3] est 'quarante', et non
        # 'trente'.
    end

    if gauche> 0
      #  Nous ne pouvons écrire 'soixantequatre'...
      chaineNombre = chaineNombre + '-'
    end
  end

  ecrire =gauche #  Combien d'unités à écrire ?
  gauche = 0     #  Il ne reste plus rien.

  if ecrire > 0
    chaineNombre = chaineNombre + unite[ecrire-1]
    #  Le "-1" est là parce que unite[3] représente 'quatre'
    #  et non 'trois'.
  end

  #  Maintenant nous retournons "chaineNombre"...
  chaineNombre
end

puts nombreBelge(0)
puts nombreBelge(9)
puts nombreBelge(10)
puts nombreBelge(11)
puts nombreBelge(17)
puts nombreBelge(32)
puts nombreBelge(88)
puts nombreBelge(99)
puts nombreBelge(100)
puts nombreBelge(101)
puts nombreBelge(234)
puts nombreBelge(899)
puts nombreBelge(1999)
```

```
zéro
neuf
dix
onze
dix-sept
trente-deux
huitante-huit
nonante-neuf
cent
cent un
deux cent trente-quatre
huit cent nonante-neuf
dix-neuf cent nonante-neuf
```

Ahhh! Voilà qui est mieux, beaucoup mieux. Le programme est plus dense, ce qui m'a permis de mettre des commentaires. Il fonctionne pour des nombres plus grands...mais pas autant que vous l'espériez n'est-ce pas? Essayez `nombreBelge(1000)`... En fait vous êtes maintenant capable d'arranger le programme tout(e) seul(e).

Quelques petites choses à essayer

- Améliorez nombreBelge en rajoutant tout d'abord les milliers. Il devra donc dire "mille neuf cent nonante-neuf".
- Améliorez ensuite en rajoutant les millions, les billions etc...
- Pour les courageux (vraiment très très courageux) transformez `nombreBelge` en `nombreFrançais`. A la place de septante, huitante et nonante il faudra donc afficher soixante-dix, quatre-vingts, quatre-vingt-dix sans oublier qu'en France on dit soixante et onze, mais quatre-vingt-onze et soixante-douze(plus de "et", et pas soixante-dix et deux). Ne pas oublier non plus que dans quatre-vingt-deux il n'y a plus d'"s" à vingt...(pareil pour "cent" d'ailleurs). Pfff!...

Félicitations ! A ce point, vous êtes un vrai programmeur! Vous avez suffisamment appris de choses pour construire entièrement de grands programmes. Si vous avez des idées de programmes à écrire, n'hésitez pas, foncez!

Bien sûr, construire quelque chose et partir de zéro est peut-être un procédé un peu lent. Mais pourquoi perdre du temps à élaborer un code alors que quelqu'un l'a peut-être déjà écrit ? Voulez-vous que votre programme envoie des courriels? Voulez-vous sauvegarder des fichiers sur votre ordinateur ? Comment générer des pages Web pour un tutoriel où les exemples de code s'exécutent lors du chargement de la page ? Pas de problèmes, Ruby est là et vous propose tout un choix d'objets différents que vous pouvez utiliser pour écrire mieux et plus vite vos programmes.

9. Les Classes

Jusqu'ici nous avons vu différents genres, ou *classes*, d'objets : les chaines, les entiers, les réels, les tableaux et quelques objets spéciaux (`true`, `false` et `nil`) dont nous parlerons plus tard. En Ruby, les noms de ces classes débutent toujours par une majuscule (capitalized) : `String, Integer, Float, Array...`etc. En général si nous voulons créer un nouvel objet d'une certaine classe, nous utilisons le mot "`new`" (nouveau) :

```
a = Array.new + [12345]
b = String.new('') + 'coucou'
c = Time.new

puts 'a= ' + a.to_s
puts 'b= ' + b.to_s
puts 'c= ' + c.to_s
```

```
a =12345
b =coucou
c == Thu Apr 08 08:10:05 Paris, Madrid (heure d'été) 2004
```

Au passage remarquez le paramètre passé obligatoirement à `String.new`; pour `Array.new` il est facultatif.

Parce que nous pouvons créer des tableaux et des chaines en utilisant [...] et '...' respectivement, nous utiliserons rarement `new` dans leur cas.(Bien que cela ne soit pas très évident dans l'exemple ci-dessus, `Array.new` crée un ensemble vide et `String.new('')` une chaîne vide). Les entiers eux, font exception. Nous ne pouvons créer un entier avec `Integer.new`. Il nous suffit d'écrire l'entier.

La classe Time

Qu'est-ce donc que cette histoire de `classe Time` ? L'objet `Time` représente des moments dans le temps. Vous pouvez ajouter (ou soustraire) des nombres à (ou d') un temps pour obtenir un nouveau temps : ajouter 1.5 à un temps crée un nouveau temps situé une seconde et demie plus tard :

```
time = Time.new
time2 = time + 60

puts time
puts time2
```

```
Thu Apr 08 08:29:53 Paris, Madrid (heure d'été) 2004
Thu Apr 08 08:30:53 Paris, Madrid (heure d'été) 2004
```

Vous pouvez aussi fabriquer un temps spécifique avec `Time.mktime` :

```
puts Time.mktime(2004,1,1)            # Année, mois, jour.
puts Time.mktime(1976, 8, 3,10, 11)   # Ma date de naissance
```

```
Thu Jan 01 00:00:00 Paris, Madrid 2004
Tue Aug 03 10:11:00 Paris, Madrid (heure d'été) 1976
```

Les parenthèses regroupent les paramètres de `mktime`. Plus vous mettez de paramètres, plus votre date est précise.
Vous pouvez comparer des temps en utilisant les méthodes de comparaison (un ancien temps est inférieur à un nouveau temps) et si vous soustrayez un temps à un autre, vous obtiendrez le nombre de secondes entre les deux. Amusez-vous avec ça...

Quelques petits trucs à essayer

- Un billion de secondes... trouvez donc depuis combien de secondes vous êtes né (si vous pouvez). Trouvez à quel âge vous avez passé le billion (ou quand vous le passerez !). Inscrivez vite ça sur votre calendrier...

- Bon anniversaire ! Demandez à une personne de votre entourage, ses jour, mois et année de naissance. Affichez l'âge précis qu'elle a alors, et affichez ensuite autant de 'Bing !' que d'anniversaires que cette personne aura eu.

La classe `Hash`

Une autre classe pratique est la classe `Hash` (hachage).Rien à voir avec votre boucher !...Les tables de hachages sont un peu comme les tableaux : elles ont un groupe d'éléments qui peuvent pointer vers différents objets. Cependant, dans un tableau, les éléments sont en lignes et numérotés (à partir de zéro). Dans une table de hachage, les éléments ne sont pas comme alignés, (ils sont juste mélangés ensemble), et vous pouvez utiliser n'importe quel objet pour vous référer à un élément, pas uniquement un nombre. C'est une bonne chose que d'utiliser les tables de hachage quand vous avez un groupe de choses dont vous voulez garder trace, mais pas forcément dans une liste ordonnée. Par exemple les couleurs que j'utilise dans différentes parties de ce tutoriel :

```
tableauCouleurs = [] # identique à Array.new
tableHCouleur = {}   # identique à Hash.new

tableauCouleurs[0] = 'rouge'
tableauCouleurs[1] = 'vert'
tableauCouleurs[2] = 'bleu'
tableHCouleur['chaîne'] = 'rouge'
tableHCouleur['nombre'] = 'vert'
tableHCouleur['mot_clef'] = 'bleu'

tableauCouleurs.each do |couleur|
  puts couleur
end
tableHCouleur.each do |codeType, couleur|
  puts codeType + ': ' + couleur
end
```

```
rouge
vert
bleu
```

```
nombre: vert
chaine: rouge
mot_clef: bleu
```

Si j'utilise un tableau j'ai à me souvenir que l'élément 0 est pour les chaines, le 1 pour les nombres, etc... Mais si j'utilise une table de hachage, c'est facile! L'élément 'chaine' contient la couleur de la chaîne bien sûr. Rien d'autre à retenir. Vous avez sans doute noté que lorsqu'on utilise each, les objets n'arrivent pas dans l'ordre dans lequel nous les avons entrés (peut-être même pas comme ils sont écrits ici, on ne sait jamais avec les tables de hachage). Les tableaux sont faits pour garder les choses en ordre, pas les tables de hachage.

Bien qu'on utilise des chaines pour nommer des éléments d'une table de hachage, vous pouvez utiliser n'importe quelle sorte d'objet, même des tableaux et d'autres tables de hachage (encore que je ne vois pas dans quel but vous feriez cela).

```
hashBizarre = Hash.new
hashBizarre[12] = 'Décembre'
hashBizarre[] = 'Vide'
hashBizarre[Time.new] = 'Rien ne vaut le temps présent'
```

Les tables de hachage et les tableaux sont adaptés à des choses différentes; c'est à vous de décider en fonction du problème.

Les classes étendues

A la fin du chapitre précédent , nous avons écrit une méthode qui, à un entier fait correspondre une chaîne. Ce n'est pas ce que l'on peut appeler une méthode d'entier; juste une méthode de "programme" générique. Ne serait-il pas plus agréable d'avoir à écrire quelque chose comme 22.to_belge au lieu de nombreBelge 22 ? Ce qui suit vous montre comment faire :

```
class Integer

  def to_belge
    if self == 5
      belgium = 'cinq'
    else
      belgium = 'cinquante-huit'
    end
    belgium
  end
end


# Test sur un couple de nombres
puts 5.to_belge
puts 58.to_belge
```

```
cinq
cinquante-huit
```

Bon, ça a l'air de marcher. ;)

Nous sommes donc entrés dans la classe `Integer` pour y déclarer une méthode d'entier, y avons défini cette méthode, et sommes sortis de la classe. Maintenant tous les entiers possèdent cette méthode (quelque peu incomplète).
En fait, si vous n'aimez pas la manière dont fonctionne une méthode comme `to_s`, il vous suffit de la redéfinir de la même façon...quoique je ne le recommande pas ! Il vaut mieux abandonner l'ancienne méthode à son triste sort, et en construire une nouvelle quand vous désirez faire quelque chose de nouveau.

Bon, il y a quelque chose qui vous chagrine ? Laissez-moi revenir un petit peu sur le dernier programme. Jusqu'à présent, chaque fois que nous exécutions du code ou définissions une méthode, nous le faisions dans l'objet "programme" par défaut. Dans notre dernier programme, nous avons pour la première fois abandonné cet objet pour entrer dans la classe `Integer`. Nous y avons défini une méthode,qui est devenue une méthode pour les entiers, et donc que tous les entiers peuvent utiliser. A l'intérieur de la méthode nous avons utilisé le mot `self` comme référence à l'objet (l'entier) utilisant la méthode.

Création de classes

Nous avons vu un certain nombre de classes d'objets. Il serait cependant facile de montrer tous les genres d'objets faisant défaut à Ruby. Mais heureusement il est aussi facile de construire une nouvelle classe que d'en étendre une ancienne. Essayons de jouer aux dés avec Ruby. Voici comment créer la classe Dé:

```
# Ruby ne veut pas d'accents (en dehors des commentaires), donc dans ce qui suit
# De=Dé et de=dé

class De

  def roule
    1 + rand(6)
  end

end

# Prenons deux dés.

jeu = [De.new, De.new]

# ...Et lançons les.
jeu.each do |de|
  puts de.roule
end
```

```
3
1
```

(Au cas où vous auriez sauté la partie sur les nombres aléatoires, `rand(6)` renvoie un nombre compris entre 0 et 5, ce qui explique l'ajout de 1 dans la méthode `roule`.)

Et voilà! Un objet bien à nous. Lancez encore quelques fois les dés, jusqu'à ce que vous gagniez...

Nous pouvons définir toutes sortes de méthodes pour nos objets...mais il manque quelque chose. Travailler avec ces objets ressemble un peu à ce que nous faisions avant de connaitre les variables. Regardez nos dés par exemple. Nous pouvons les lancer et chaque fois que nous le faisons nous obtenons des nombres différents. Mais si nous voulons garder ce nombre, nous serons obligés de créer une variable pointant sur ce nombre. Il est tout de même raisonnable d'envisager qu'un dé puisse posséder un nombre bien à lui, et que le lancement du dé change ce nombre. Si nous gardons trace du dé, nous n'aurons plus besoin de nous soucier du nombre.

Variables d'instance

Normalement lorsque nous voulons parler d'une chaîne de caractères, nous l'appelons simplement *string*. Nous pouvons aussi l'appeler un objet string. Les programmeurs parlent parfois d'une instance de la classe string, mais c'est juste une manière pompeuse (plutot que bavarde) de dire string. Une instance de classe est tout simplement un objet de cette classe.

Les variables d'instance sont simplement les variables d'un objet. Les variables locales d'une méthode durent jusqu'à ce que la méthode ne soit plus utilisée. Les variables d'instance d'un objet, d'autre part, dureront toute la vie de l'objet. Pour les différencier des variables locales, elles ont un '@' au début de leur nom :

```
class De

  def roule
    @nombre = 1 + rand(6)
  end

  def afficheNombre
    @nombre
  end

end

jeu = De.new
jeu.roule
puts jeu.afficheNombre
puts jeu.afficheNombre
jeu.roule
puts jeu.afficheNombre
puts jeu.afficheNombre
```

```
1
1
5
5
```

Très bien ! Ainsi `roule` fait rouler les dés, et `afficheNombre` nous indique quel nombre est sorti. Cependant, que peut bien nous indiquer cette dernière méthode avant qu'on ait lancé le dé (avant l'affectation d'une valeur à @nombre)?

```
class De

  def roule
    @nombre = 1 + rand(6)
  end

  def afficheNombre
    @nombre
  end

end

# Puisque je n'ai pas l'intention d'utiliser cette valeur de
#  nouveau, il n'est pas nécessaire de la sauvegarder dans une
# variable.

puts De.new.afficheNombre
```

```
nil
```

Hmmm...Bon, au moins nous n'avons pas une erreur en retour. Tout de même, c'est un manque de bon sens pour un dé d'être "non lancé", ou du moins c'est ce que semble signifier le `nil`. Ce qui serait bien c'est que nous puissions lui donner une valeur lors de sa création. Pas de problèmes, la méthode `initialize` est faite pour ça :

```
class De

  def initialize
    # Je lance seulement le dé,
    # si nous voulions obtenir une valeur fixe, comme 6,
    # il faudrait faire autre chose.
    roule
  end

  def roule
    @nombre = 1 + rand(6)
  end

  def afficheNombre
    @nombre
  end

end

puts De.new.afficheNombre
```

```
4
```

Lors de la création d'un objet, sa méthode `initialize` (si elle existe) est toujours appelée.

Notre dé est presque parfait. La seule chose qui puisse lui manquer est une façon de lui imposer une face à montrer... Pourquoi n'écririez-vous pas une méthode `triche` qui fasse justement cela ! Revenez quand ce sera fini (et que vous l'aurez testée bien sur !). Assurez-vous qu'on ne puisse pas afficher 7 !

C'est un joli petit truc que nous venons d'apprendre. Il est difficile, aussi laissez-moi vous donnez un autre exemple plus intéressant. Essayons de fabriquer un petit animal de compagnie, un bébé dragon. Comme tous les bébés, il doit être capable de manger, dormir et faire caca, ce qui signifie que nous devrons être capable de l'alimenter, de le mettre au lit et de le promener. De façon interne , notre dragon devra garder trace de son état d'inanition, de fatigue ou d'entrain, mais nous ne devrons pas voir tout ça lors d'une interaction avec notre dragon, par exemple nous ne pourrons pas lui demander, comme à un bébé humain, "As-tu faim ?". Nous avons ajouté quelques petits trucs de communication avec notre dragon, et lors de sa naissance, nous pourrons lui donner un nom.(Quoique vous passiez dans la méthode `new`, cela est passé pour vous dans la méthode `initialize`). Et bien allons-y :

```
class Dragon

  def initialize nom
    @nom = nom
    @endormi = false
    @estomac   = 10  #  Il est "plein"!.
    @intestins =  0  #  Pas besoin de pot.

    puts @nom + ' est né.'
  end

  def nourrir
    puts 'Vous nourrissez ' + @nom + '.'
    @estomac = 10
    lapsDeTemps
  end

  def promener
    puts 'Vous promenez ' + @nom + '.'
    @intestins = 0
    lapsDeTemps
  end

  def coucher
    puts 'Vous mettez ' + @nom + ' au lit.'
    @endormi = true
    3.times do
      if @endormi
        lapsDeTemps
      end
      if @endormi
        puts @nom + ' ronfle, remplit la pièce de fumée.'
```

```
    end
  end
  if @endormi
    @endormi = false
    puts @nom + ' se réveille lentement.'
  end
end

def lancer
  puts 'Vous lancez ' + @nom + ' en l\'air.'
  puts 'Il rigole, ce qui vous brûle les sourcils.'
  lapsDeTemps
end

def bercer
  puts 'Vous bercez ' + @nom + ' tendrement.'
  @endormi = true
  puts 'Il s\'assoupit brièvement...'
  lapsDeTemps
  if @endormi
    @endormi = false
    puts '...mais s\'éveille quand vous cessez.'
  end
end

private

#  "private" signifie que les méthodes définies ici sont
#  des méthodes internes à l'objet.  (Vous pouvez nourrir
#  votre dragon, mais vous ne pouvez pas lui demander s'il a
#  faim.)

def faim?
  #  Les noms de méthodes peuvent se terminer par un "?".
  #  Normalement nous faisons cela si elles
  #  retournent true ou false, comme ceci:
  @estomac <= 2
end

def caca?
  @intestins >= 8
end

def lapsDeTemps
  if @estomac > 0
    #  Déplace les aliments de l'estomac vers l'intestin.
    @estomac = @estomac - 1
    @intestins = @intestins + 1
  else  #  Votre dragon est affamé!
    if @endormi
      @endormi = false
      puts 'Il s\'éveille soudain!'
    end
```

```
      puts @nom + ' est affamé!  De désespoir il VOUS a mangé!'
      exit  #  Ceci fait quitter le programme.
    end

    if @intestins >= 10
      @intestins = 0
      puts 'Hop-là!  ' + @nom + ' a un accident...'
    end

    if faim?
      if @endormi
        @endormi = false
        puts 'Il s\'éveille soudain!'
      end
      puts @nom + ' a l\'estomac qui gronde...'
    end

    if caca?
      if @endormi
        @endormi = false
        puts 'Il s\'éveille soudain!'
      end
      puts ' ça URGE !!!...'
    end
  end

end

petit = Dragon.new 'Norbert'
petit.nourrir
petit.lancer
petit.promener
petit.coucher
petit.bercer
petit.coucher
petit.coucher
petit.coucher
petit.coucher
```

```
Norbert est né.
Vous nourrissez Norbert.
Vous lancez Norbert en l'air.
Il rigole, ce qui vous brûle les sourcils.
Vous promenez Norbert.
Vous mettez Norbert au lit.
Norbert ronfle, remplit la pièce de fumée.
Norbert ronfle, remplit la pièce de fumée.
Norbert ronfle, remplit la pièce de fumée.
Norbert se réveille lentement.
Vous bercez Norbert tendrement.
Il s'assoupit brièvement...
...mais s'éveille quand vous cessez.
Vous mettez Norbert au lit.
Il s'éveille soudain!
```

```
Norbert a l'estomac qui gronde...
Vous mettez Norbert au lit.
Il s'éveille soudain!
Norbert a l'estomac qui gronde...
Vous mettez Norbert au lit.
Il s'éveille soudain!
Norbert a l'estomac qui gronde...
ça URGE !!!...
Vous mettez Norbert au lit.
Il s'éveille soudain!
Norbert est affamé!  De désespoir il VOUS a mangé!
```

Ouf! Bien sûr, ce serait mieux d'avoir un programme interactif, mais vous pourrez faire cela plus tard. Je voulais simplement essayer de montrer les différentes parties relatives à la création d'une nouvelle classe dragon.

Nous avons vu quelques petites nouveautés dans cet exemple. La première est toute simple : `exit` termine le programme. La seconde est le mot `private` que nous avons placé juste au milieu de notre définition de classe. J'aurais pu l'omettre, mais je souhaitais renforcer l'idée de certaines méthodes que vous pourriez faire exécuter à un dragon et d'autres qui se produisent sans le dragon. Vous pouvez les assimiler à ce qui se passe "sous le capot" : à moins que vous ne soyez une mécanique automobile, tout ce que vous avez à connaître pour conduire est la pédale d'accélérateur, la pédale de freins, l'embrayage et le volant. Un programmeur appellerait ceci l'interface publique de votre voiture. “Quand l'airbag doit-il se déclencher” est une action interne à la voiture; l'utilisateur typique (le chauffeur) n'a pas besoin d'en savoir plus.

En fait, pour un exemple un peu plus concret, essayons de voir comment vous pourriez vous représenter une voiture dans un jeu vidéo (ce qui est mon boulot). Premièrement, vous déciderez de ce à quoi doit ressembler votre interface publique; en d'autres termes, quelles méthodes l'utilisateur pourra appeler d'un de vos objets voiture? Bon, il devra pouvoir accélérer et freiner, mais il aura aussi besoin de savoir avec quelle force appuyer sur les pédales. (Il y a une grande différence entre effleurer et enfoncer les pédales) Il devra aussi pouvoir tourner le volant, et de nouveau, savoir avec quelle force il tourne les roues. Je suppose que vous irez encore plus loin et ajouterez des clignotants, un lance-fusées, la postcombustion, etc... cela dépend du type de jeu que vous voulez faire.

Il ya bien d'autres choses internes nécessaires dans un objet voiture; par exemple la vitesse, la direction et la position (au minimum). Ces attributs seront modifiés par l'appui sur la pédale d'accélérateur ou celle des freins ou en tournant le volant, mais l'utilisateur ne pourra pas décider exactement de la position (qui pourra être faussée). Vous pourriez aussi envisager de garder trace des dérapages et des dégats, ainsi de suite. Tout cela sera interne à votre objet voiture.

Quelques petits trucs à essayer

- Construisez une classe `Oranger`. Elle devra avoir une méthode hauteur qui renverra la hauteur de l'arbre, et une méthode `unAnDePlus` qui, lors de son appel, vieilira l'arbre d'une année. Chaque année l'arbre grandit (il vous faudra imaginer de combien un oranger peut grandir en un an...), et après un certain

nombre d'années (idem) l'arbre mourra. Pour les toutes premières années, il ne produira aucun fruit, mais après quelque temps si, et je pense que les vieux arbres produisent plus que les jeunes...enfin faites ça avec le plus de bon sens possible. Et, bien sur, vous devrez avoir la variable `nombreOranges` (qui mesurera votre récolte), et une autre `gouteOrange` (qui réduira `@nombreOrange` d'une unité et renverra une chaîne vous annonçant combien l'orange était bonne, ou bien vous préviendra qu'il n'y a plus d'oranges à goûter pour cette année). N'oubliez pas que les oranges non ramassées d'une année, tombent de l'arbre avant l'année suivante.

- Ecrivez un programme qui soit interactif avec votre bébé dragon. Vous devrez pouvoir entrer des commandes telles que `nourrir` et `promener`, et ces méthodes devront pouvoir être appelées par le dragon. Bien sur, ce que vous allez entrer sera des chaines de caractères, vous devrez donc avoir une sorte de répartiteur de méthodes, qui permettra au programme de se diriger vers telle ou telle méthode suivant la chaîne entrée.

Et voilà, c'est pratiquement fini. Mais attendez une seconde... je ne vous ai parlé d'aucune classe qui permette d'envoyer un e-mail, ou bien charger ou sauver un fichier sur le disque, ou comment créer des fenêtres et des boutons, ou un monde en 3D, ou bien d'autres choses ! Eh bien , il existe tellement de classes que vous pouvez utiliser que je ne peux vous les montrer toutes; d'ailleurs je suis bien incapable de vous dire combien il y en a ! Ce que je puis vous dire tout de même est où les trouver, avoir des renseignements à leur sujet de telle sorte que vous puissiez les utiliser dans vos programmes. Mais avant de vous éclairer sur ce sujet, laissez-moi toutefois vous parler d'une ou deux particularités de Ruby, que d'autres langages ne possèdent pas, mais sans lesquelles je ne puis tout simplement plus vivre : les blocs et les procs.

10. Blocs et Procs

C'est sans aucun doute une des plus formidables particularités de Ruby. D'autres langages la possèdent aussi, parfois sous un autre nom (comme *closures*), mais pas la grande majorité des plus populaires, ce qui est bien dommage.

Mais qu'est-ce donc que cette nouvelle chose? Il s'agit de la possibilité de prendre un bloc de code (entre un `do` et un `end`), de l'emballer dans un objet (appelé une *proc*), le stocker dans une variable ou bien le passer à une méthode, et exécuter le code contenu dans le bloc quand bon vous semble (et plusieurs fois, si vous le voulez). C'est en fait comme une méthode en soi, excepté que le bloc n'est pas relié à un objet (c'est un objet), et vous pouvez le stocker ou l'associer à un objet quelconque selon vos désirs. Je pense qu'il est temps de donner un exemple :

```
toast = Proc.new do
  puts 'Bravo !!!'
end
toast.call
toast.call
toast.call
```

```
Bravo !!!
Bravo !!!
Bravo !!!
```

J'ai donc créé une `proc` (je suppose que c'est l'abréviation du mot "procédure", mais le plus important est qu'il rime avec "bloc") qui contient le bloc de code, puis j'ai appelé (`call`) la `proc` 3 fois. Comme vous pouvez le voir cela ressemble bien à une méthode.

Effectivement, les procs vont ressembler encore plus à des méthodes quand je vous aurai dit que les blocs peuvent avoir des paramètres.

```
aimezVous = Proc.new do |uneBonneChose|
  puts 'J\'aime *bien* ' +uneBonneChose+'!' # Attention, pas
                                            # d'espaces entre +
end                                         # et uneBonneChose.

aimezVous.call 'le chocolat'
aimezVous.call 'Ruby'
```

```
J'aime *bien* le chocolat !
J'aime *bien* Ruby !
```

Bon, nous avons vu ce qu'étaitent les blocs et les procs, et comment les utiliser, mais où est l'intérêt? Pourquoi ne pas simplement utiliser des méthodes? Parce qu'il y a des choses que l'on ne peut pas faire avec des méthodes. En particulier, vous ne pouvez pas passer une méthode dans une autre méthode (mais vous pouvez passer des procs dans des méthodes), et une méthode ne peut retourner une autre méthode (mais peut retourner des procs). Tout simplement parce que les procs sont des objets; les méthodes non.

(Au fait, est-ce que ceci ne vous est pas familier ? Mais c'est bien sûr, vous avez déjà vu des blocs... quand vous avez appris les itérateurs. Mais on va essayer d'en dire un peu plus).

Les méthodes qui acceptent des procs

Quand nous passons une proc à une méthode, nous pouvons contrôler, comment, si ou combien de fois nous appelons la proc. Par exemple, disons que nous voulons faire quelque chose avant et après l'exécution du code :

```
def important uneProc
  puts 'Tout le monde dit ATTENDEZ ! Je suis occupé...'
  uneProc.call
  puts 'Bon, bon, j\'attends. Faites ce que vous avez à faire.'
end

direBonjour = Proc.new do
  puts 'Bonjour'
end

direAuRevoir = Proc.new do
  puts 'Au revoir'
end

important direBonjour
important direAuRevoir
```

```
Tout le monde dit ATTENDEZ ! Je suis occupé...
Bonjour
Bon, bon, j'attends. Faites ce que vous avez à faire.
Tout le monde dit ATTENDEZ ! Je suis occupé...
Au revoir
Bon, bon, j'attends. Faites ce que vous avez à faire.
```

Il se pourrait bien que cela ne vous paraisse pas fabuleux...mais ça l'est.:) Il est très courant en programmation d'avoir des contraintes qui doivent être accomplies. Si, par exemple, vous voulez sauvegarder un fichier, vous devez ouvrir ce fichier, y écrire ce que vous désirez qu'il conserve, et enfin le fermer. Si vous oubliez de fermer le fichier, il va se passer de très mauvaises choses. Mais chaque fois que vous voudrez sauvegarder ou charger un fichier, vous devrez faire la même chose : ouvrir le fichier, faire ce que vous voulez vraiment faire, et puis fermer le fichier. Il est assommant et facile d'oublier. En Ruby, sauvegarder (ou charger) un fichier, fonctionne d'une manière similaire au code ci-dessus, vous n'avez donc pas d'autres soucis que de savoir réellement ce que vous désirez sauver (ou charger). (Dans le prochain chapitre je vous dirai enfin où trouver comment faire pour sauver ou charger des fichiers).

Vous pouvez aussi écrire des méthodes qui détermineront combien de fois, ou même dans quelle condition (`if`), appeler une proc. Voici une méthode qui appellera la proc passée une fois sur deux, et l'autre qui l'appellera deux fois:

```ruby
def aFaire uneProc
  if rand(2) == 0
    uneProc.call
  end
end

def deuxFois uneProc
  uneProc.call
  uneProc.call
end

clignote = Proc.new do
  puts '<Clignote>'
end

fixe = Proc.new do
  puts '<Fixe>'
end

aFaire clignote
aFaire fixe
puts ''
deuxFois clignote
deuxFois fixe
```

```
<Clignote>

<Clignote>
<Clignote>
<Fixe>
<Fixe>
```

(Si vous relancez ce code quelques fois, vous verrez que la sortie change). Ce sont quelques-unes des utilisations habituelles que nous permettent les procs, et que nous ne pourrions pas faire à l'aide des seules méthodes. Bien sûr, vous pourriez écire une méthode qui clignote deux fois, mais pas une qui fasse précisément quelque chose deux fois!

Avant de terminer, un dernier exemple. Jusqu'à présent les procs que nous avons passées se ressemblaient fort. Ce coup-ci cela va être un peu différent, et vous verrez combien la méthode est dépendante de ce que vous lui passez. Notre méthode va prendre un objet et une proc et appellera la proc de cet objet. Si la proc retourne faux (`false`) nous quittons; sinon nous appelons la `proc` avec l'objet en retour. Nous ferons cela jusqu'au renvoi de false par la proc. (ce qu'elle a de mieux à faire , sinon le programme plantera). La méthode retournera la dernière valeur non fausse renvoyée par la proc.

```ruby
def faireJusquaFaux premierement, uneProc
  entree = premierement
  sortie = premierement

  while sortie
    entree = sortie
    sortie = uneProc.call entree
```

```
  end

  entree
end

tableauDeCarres = Proc.new do |tableau|
  dernierNombre = tableau.last
  if dernierNombre <=0
    false
  else
    tableau.pop  # Enlève le dernier nombre.
    tableau.push dernierNombre*dernierNombre
                 # ...et le remplace par son carré.
    tableau.push dernierNombre-1
                 # ..suivi par le nombre suivant inférieur.
  end
end

toujoursFaux = Proc.new do |aIgnorer|
  false
end

puts faireJusquaFaux([5], tableauDeCarres).inspect
puts faireJusquaFaux('J\'écris ceci à 15 heures; quelqu\'un
m\'appelle !', toujoursFaux)
```

```
[25, 16, 9, 4, 1, 0]
J'écris ceci à 15 heures; quelqu'un m'appelle !
```

Bon, c'est un exemple sacrément bizarre, je l'admets. Mais il montre combien différemment réagissent nos méthodes lorsqu'on leur passe des procs très différentes.

La méthode `inspect` est semblable à `to_s`, sauf que la chaîne retournée tâche de vous montrer de façon lisible le code Ruby de construction de l'objet que vous lui avez passé. Ici elle montre le tableau entier retourné après le premier appel à `faireJusquaFaux`. Vous pouvez noter aussi que nous n'appelons jamais 0, à la fin du tableau, pour l'élever au carré puisque 0 au carré reste 0, ce qui nous évite un appel. Et puisque `toujoursFaux` était, souvenez-vous, toujours faux,
`faireJusquaFaux` ne fait rien du tout lors du second appel; la méthode retourne seulement ce qu'on lui a passé.

Les méthodes qui retournent des procs

Une des autres choses que l'on peut faire avec les procs, c'est de les créer dans les méthodes et de les retourner ensuite. Ceci permet une puissance de programmation folle (des choses avec des noms impressionnants comme *évaluation de paresse*, *structures de données infinies*, et *étrillage*), mais le fait est que je n'ai jamais mis cela en pratique, ni jamais vu quelqu'un l'employer dans son code. Je pense que ce n'est pas quelque chose que vous serez amenés à utiliser fréquemment en Ruby, et peut-être bien que Ruby vous encouragera à faire différemment; je ne sais pas. En tout cas, je veux juste vous en parler un peu.

Dans cet exemple, `compose` a deux procs en paramètres et retourne une nouvelle procédure qui, lorsqu'on l'appelle, appelle la première proc et passe son résultat à la seconde proc.

```
def compose proc1, proc2
  Proc.new do |x|
    proc2.call(proc1.call(x))
  end
end

carre = Proc.new do |x|
  x * x
end

double = Proc.new do |x|
  x + x
end

doublePuisCarre = compose double, carre
carrePuisDouble = compose carre, double

puts doublePuisCarre.call(5)
puts carrePuisDouble .call(5)
```

```
100
50
```

Notez que l'appel à `proc1` doit être entre parenthèses pour `proc2` afin d'être exécutée en premier.

Passer des blocs (pas des procs) dans les méthodes

D'accord, ceci a été une sorte de divertissement *académique*, mais aussi une petite pagaïe. Une partie du problème vient de ce que vous avez trois étapes à parcourir (définir la méthode, construire la proc, et appeler la méthode à l'aide de la proc), et qu'on sent bien que deux pourraient suffire (définir la méthode, et simplement passer le bloc dans la méthode, sans utiliser une proc), puisque la plupart du temps vous n'utiliserez plus le proc/bloc après son passage à la méthode. Et bien, vous vous en doutiez, Ruby a tout résolu pour nous! En fait, vous avez déjà accompli tout cela avec les itérateurs.

Avant d'aller plus loin, je vais vous montrer rapidement un premier exemple.

```
class Array

  def chaquePair(&etaitUnBloc_estUneProc)
    estPair = true  #  Nous débutons par "true" parce que les
                    # tableaux débutent à l'index 0, qui est
                    # pair.

    self.each do |objet|
      if estPair
```

```
        etaitUnBloc_estUneProc.call objet
      end

      estPair = (not estPair)   #  Bascule de pair à impair, ou
                                # de impair à pair.
    end
  end

end

['pommes', 'mauvaises pommes', 'cerises', 'poires'].chaquePair do
|fruit|
  puts 'Hmmm! J\'aime les tartes aux '+fruit+' , pas vous?'
end

#  Souvenez-vous, nous démarrons avec les éléments pairs
#  du tableau, en fait tous ceux qui sont représentés par des
#  nombres impairs, simplement parce que j'aime bien poser ce
# genre de problème...
[1, 2, 3, 4, 5].chaquePair do |balleImpaire|
  puts balleImpaire.to_s+' n\'est PAS un nombre pair!'
end
```

```
Hmmm! J'aime les tartes aux pommes , pas vous?
Hmmm! J'aime les tartes aux cerises , pas vous?
1 n'est PAS un nombre pair!
3 n'est PAS un nombre pair!
5 n'est PAS un nombre pair!
```

Ainsi pour passer un bloc à `pourChacun`, tout ce que nous avons à faire est de placer le bloc après la méthode. Vous pouvez passer un bloc dans toute méthode de cette façon. Afin que votre méthode n'ignore pas le bloc, mais le prenne et le transforme en proc, faites précéder le nom de la proc passée en paramètre à la méthode d'une esperluète (&). Cette partie est un peu délicate, mais pas trop, et vous n'avez à le faire qu'une fois (lorsque vous définissez la méthode). Vous pouvez ensuite utiliser plusieurs fois la méthode, de la même manière que les méthodes préfabriquées qui acceptent les blocs comme `each` et `times`. (Souvenez-vous : `5.times do...`?)

Si vous perdez les pédales, essayez de vous représenter ce que `chaquePair` est sensé faire : appeler le bloc avec chaque élément du tableau. Une fois que vous l'avez écrit et qu'il fonctionne correctement, vous n'avez plus à vous préoccuper de ce qui se passe sous le capot ("Quel bloc est appelé quand ???") ; en fait, c'est exactement pourquoi nous écrivons de telles méthodes; nous n'avons plus jamais à nous poser la question de savoir comment elles marchent. Nous les utilisons simplement.

Je me souviens d'une fois où je désirais connaître le temps que mettaient à s'exécuter différentes parties d'un programme. (On parle alors de *profiler* le code). J'écrivis alors une méthode qui notait l'heure avant l'exécution du code, celle à la fin du code, et qui me donnait par différence le temps écoulé. Je ne saurais retrouver ce code, mais ce n'est pas nécessaire; il devait ressembler à quelque chose comme ça :

```ruby
def profile descriptionDuBloc, &bloc
  debut = Time.now

  bloc.call

  duree = Time.now - debut

  puts descriptionDuBloc+':  '+duree.to_s+' secondes'
end

profile 'doubler 25000' do
  nombre = 1

  25000.times do
    nombre = nombre + nombre
  end

  puts nombre.to_s.length.to_s+' chiffres'
     #  C'est le nombre de chiffres de cet ENORME nombre.
end

profile 'compte jusqu\'à un million' do
  nombre = 0

  1000000.times do
    nombre = nombre + 1
  end
end
```

```
7526 chiffres
doubler 25000:  7.69 secondes
compte jusqu'à un million:  11.09 secondes
```

Comme c'est simple! Et élégant! Avec cette toute petite méthode, je suis maintenant capable de chronométrer une partie quelconque d'un programme quelconque; il me suffit de prendre le code et de le passer à `profile`. Quoi de plus simple? Dans la plupart des langages, je serais obligé d'ajouter explicitement ce code de chronométrage (le contenu de `profile`) à chaque partie à tester. En Ruby, pas du tout, chaque chose reste à sa place, et (le plus important) pas sur mon chemin!

Quelques petites choses à essayer

- L'horloge de grand-père. Ecrivez une méthode qui prend un bloc et l'appelle pour chaque heure passée de la journée. Ainsi, si je passe un bloc du style : `do puts 'DONG!' end`, il devra "carillonner" comme une horloge. Tester votre méthode avec des blocs différents (y compris celui que je vous ai indiqué). **Remarque :** Vous pouvez utiliser `time.now.hour` pour obtenir l'heure actuelle. Cependant cette méthode vous retournera un nombre compris entre 0 et 23, vous aurez donc à corriger ceci afin de vous rapprocher de la réalité d'une horloge qui affiche, elle, les heures de 1h à 12h.

- Un pisteur de programme. Ecrivez une méthode appelée `log`, qui prend la chaîne de description d'un bloc et, bien sûr, un bloc. Identique à la méthode `important`,elle doit indiquer par un `puts` que le bloc a débuté, par une autre chaîne qu'il s'est achevé, et indiquer aussi ce que retourne le bloc. Testez votre méthode en lui envoyant un bloc de code. A l'intérieur du bloc, faites un autre appel à `log`, et passez-lui un autre bloc. (on parle alors de blocs imbriqués.) En d'autres termes, votre sortie doit ressembler à quelque chose du style :

```
Début de "bloc externe"
Début de "un petit bloc"
..."un petit bloc" terminé, retourne : 5
Début de "encore un autre bloc"...
..."encore un autre bloc" terminé, retourne : J'aime la cuisine
ThaÏ!
..."bloc externe" terminé, retourne : false
```

- Un meilleur pisteur. La sortie de ce dernier pisteur est plutôt difficile à lire, et cela ira en empirant en l'utilisant fréquemment. Il serait plus aisé à lire en indentant les lignes dans les blocs internes. Pour faire ceci, vous devez garder trace de la profondeur d'imbrication où se trouve le pisteur lorsqu'il doit écrire quelque chose. Pour faire ceci utilisez une variable globale, c'est à dire une variable visible de tous les endroits de votre code. Pour rendre une variable globale il suffit de faire précéder son nom du signe "$", comme ceci : `$globale`, `$profondeur`, et `$sommet`. A la fin votre pisteur vous retournera une sortie comme :

```
Début de "bloc externe"
  Début de "un petit bloc"
    Début de "tout petit bloc"
    ..."tout petit bloc" terminé, retourne : bisous!
  ..."un petit bloc" terminé, retourne : 42
  Début de "encore un bloc"
  ..."encore un bloc" terminé, retourne : j'aime la cuisine indi-
enne!
"bloc externe" terminé, retourne : true
```

Bien nous voici à la fin de ce tutoriel. Félicitations! Vous avez appris pas mal de choses! Il se peut que vous ne vous souveniez déjà plus de tout, ou bien que vous ayez sauté quelques parties...ce n'est pas vraiment grave. Programmer n'est pas tout connaître, mais plutôt tout ce que vous pouvez imaginer. Tant que vous saurez où trouver les choses que vous avez oubliées, tout ira bien. J'espère que vous ne pensez pas que j'ai écrit tout ce qui précède sans lever les yeux une seule minute! Parce que si. J'ai obtenu pas mal d'aide du code des exemples qui s'exécutaient tout au long de ce tutoriel. Mais que dois-je regarder et à qui puis-je demander de l'aide? Suivez-moi...

11. Au delà de ce tutoriel

Où aller à présent. Si vous avez une question, à qui la poser? Que faire si vous voulez que votre programme ouvre une page Web, envoie un courriel, ou redimensionne une image graphique? Et bien, il y a beaucoup, beaucoup d'endroits où trouver de l'aide sur Ruby. Malheureusement, c'est plutôt un handicap, n'est ce pas? :)

En ce qui me concerne, il y a en fait seulement trois endroits pour obtenir de l'aide sur Ruby. S'il s'agit d'une petite question, et que je pense que je peux essayer de m'en sortir seul, j'utilise **irb**. S'il s'agit d'une question plus importante, je regarde dans la bible de Ruby. Et si je ne m'en tire pas par moi-même, je fais alors appel à ruby-talk.

IRB: Interactive Ruby

Si vous avez installé Ruby, alors vous avez installé **irb**. Pour l'utiliser, il vous suffit d'ouvrir votre fenêtre de commande et de taper `irb`. Une fois que vous êtes dedans, vous pouvez taper l'expression que vous souhaitez, et irb vous en retournera la valeur. Tapez `1+2`, et il vous répondra 3. (Notez que vous n'avez pas à utiliser `puts`.) C'est une sorte de calculateur Ruby géant. Lorsque vous avez terminé, tapez seulement "`exit`" (sortir) pour quitter le mode de commande. Pour en apprendre plus sur **irb**, voyez la bible (ou bien "Apprenez Ruby" disponible en français sur le site www.ruby-doc.org).

La bible: "Programming Ruby"

Les américains l'appellent "The Pickaxe", la pioche, parce qu'il y en a une de dessinée sur la couverture de l'ouvrage. C'est le livre à avoir absolument; "Programming Ruby, The Pragmatic Programmer's Guide" (Uniquement en langue anglaise, pour le moment), de David Thomas et Andrew Hunt (les programmeurs pragmatiques). Bien que je vous recommande l'achat de cet excellent ouvrage, vous pouvez néanmoins l'obtenir gratuitement sur Internet. (D'ailleurs, si vous avez installé la version Windows de Ruby, vous le possédez déjà; Andrew Hunt est le type qui a fait le programme d'installation de la version Windows.)

Vous y trouverez tout ce qui concerne Ruby, de la base au plus perfectionné. Il est facile à lire; compréhensible; il est tout simplement parfait. Je souhaiterais que chaque langage ait un livre de cette qualité. A la fin du livre, vous pourrez y trouver une grande section détaillant chaque méthode dans chaque classe, l'expliquant et donnant des exemples. J'aime tout simplement ce livre!

Il y a plusieurs endroits où vous pouvez l'obtenir (y compris le propre site des Programmeurs Pragmatiques), mais mon lieu préféré est : http://phrogz.net/ProgrammingRuby/. Cette version possède une super table des matières sur le côté, et un index. Mais le meilleur est le bas de la fenêtre. Depuis la première parution de la bible sur Ruby, ce dernier a évolué. La communauté de Ruby a gardé trace de ces changements et les a mis en ligne. Ainsi se trouvent en bas de page les additifs ou corrections au livre original.

Ruby-Talk: la liste de diffusion de Ruby.

Même avec **irb** et la bible, il peut arriver que vous soyez bloqué(e). Ou peut-être voudrez-vous savoir si quelqu'un a déjà buché sur votre travail actuel, et voir si vous pouvez utiliser ses recherches. Dans ces cas là, le lieu où aller est ruby-talk, la liste de diffusion de Ruby. Elle est remplie de gens charmants, amicaux et prêts à vous aider. Pour en savoir plus ou pour y souscrire, regardez à http://ruby-lang.org/en/ml.html

ATTENTION: Il y a pas mal de courrier chaque jour sur la liste de diffusion et toutes les discussions se font en anglais.

Tim Toady

Quelque chose dont j'ai essayé de vous protéger, mais que vous découvrirez sans tarder, est le concept de TMTOWTDIT (prononcez "Tim Toady"): There's More Than One Way To Do It. (Il y a plusieurs façons de faire ça)

Certains trouveront ce concept magnifique, d'autres seront indifférents. Je n'ai pas réellement une mauvaise opinion à ce sujet, mais il me semble que c'est une façon *terrible* pour apprendre à quelqu'un à programmer. (Comme si apprendre *une* façon de faire quelque chose n'était déjà pas suffisamment déconcertant en soi et ambitieux).

Cependant, maintenant que vous allez quitter ce tutoriel, vous allez découvrir des codes diversifiés. Par exemple, je suis en train de penser à au moins cinq façons différentes de celle que je vous ai montrée, pour construire une chaîne de caractères (en l'encadrant de guillemets simples), et chacune d'entre elles fonctionne différemment des autres. Je vous ai montré la plus simple des six.

Et quand nous avons parlé des branchements, je vous ai montré `if` (si), mais je n'ai rien dit de `unless` (à moins que). Je vous laisse l'essayer à l'aide de **irb**.

Une autre façon d'utiliser `if`, `unless` et `while` est la mignonne version qui consiste à les placer après le code pour le modifier (faire *ceci* si...):

```
puts 'Probablement que ce texte va s\'afficher'  if 5 ==2**2 +
1**1
puts 'certainement que celui-là non !' unless 'Chris'.length == 5
```

```
Probablement que ce texte va s'afficher
```

Et finalement il existe d'autres façons d'écrire des méthodes qui prennent des blocs (pas des procs). Nous avons vu celle où nous prenions le bloc et le faisions pénétrer dans une proc en utilisant l'astuce de `&bloc` dans la liste des paramètres lors de la définition de la fonction. Puis pour appeller le bloc, nous avons simplement utilisé `bloc.call`. Bon, et bien il y a une façon plus rapide (encore que j'estime prêtant plus à confusion). Au lieu de ceci:

```
def faitDeuxFois(&bloc)
  bloc.call
  bloc.call
end
```

```
faitDeuxFois do
  puts 'Là, je pense que je vais m'afficher deux fois...'
end
```

```
Là, je pense que je vais m'afficher deux fois...
Là, je pense que je vais m'afficher deux fois...
```

...faites cela:

```
def faitDeuxFois(&bloc)
  yield
  yield
end

faitDeuxFois do
  puts 'Là, je pense que je vais m'afficher deux fois...'
end
```

```
Là, je pense que je vais m'afficher deux fois...
Là, je pense que je vais m'afficher deux fois...
```

Je ne sais pas...qu'est-ce que vous en pensez? Peut-être que c'est moi mais...`yield`?! Si c'était quelque chose comme `appelle_le_bloc_en_haut` ou un truc dans ce goût-là qui ait un sens pour moi... Il y a des gens chez qui `yield` a un sens. Mais c'est à cause de cela qu'existe Tim Toady : ils font de leur façon, et je fais de la mienne.

Fin

Bonne utilisation. :) Si cela vous interesse, vous pouvez aussi lire le chapitre suivant "Au sujet de ce tutoriel", et si vous l'avez trouvé pratique (ou confus ou bien si vous avez trouvé une erreur), faites-le moi savoir : ruby.tutorial@hellotree.com

Au sujet de ce tutoriel

Ce tutoriel a été écrit par moi, Chris Pine (courriel : ruby.tutorial@hellotree.com). Vous pouvez l'utiliser, le modifier, le copier sur votre propre site...la seule chose que je vous demande est de mentionner un lien clair et sans ambiguïté vers l'original.

Les pages que vous voyez (compatible XHTML 1.1) ont été générées par le programme en Ruby disponible sur http://pine.fm/LearnToProgram/?ShowTutorialCode=true. Ce n'est peut-être pas le code du siècle, mais il possède quelques particularités ingénieuses. Par exemple, tous les exemples s'exécutent chaque fois que vous regardez la page, et la sortie que vous voyez est la sortie qu'ils génèrent. Je pense que c'est la meilleure façon d'être sûr que le code que vous allez lire fonctionne exactement comme je le dis. Vous n'avez donc pas à craindre un erreur de copie de code de ma part, ou un oubli de test; tout est testé lorsque vous le voyez. (Ainsi dans la section sur les nombres aléatoires, vous verrez les nombres modifiés à chaque rechargement de la page... c'est chic non?).

J'ai essayé de séparer les concepts autant que possible dans ce tutoriel, de telle sorte que l'étudiant n'en ait qu'un à apprendre à la fois. Ce fut difficile au début, mais de plus en plus facile au cours de l'avancement de l'ouvrage. Certaines choses ont du être apprises avant d'autres, et j'ai été étonné par le manque de hiérarchie. En fin de compte, j'ai tout simplement choisi un ordre, et tenté d'arranger les choses de telle sorte que chaque nouvelle section soit une suite motivée par la précédente. Je ne suis pas sûr d'y être toujours parvenu.

Un autre principe de ce tutoriel est d'apprendre une seule façon de faire quelque chose. C'est un avantage évident pour quelqu'un qui n'a jamais programmé. Pour une chose, une façon de la faire est plus facile à apprendre que deux. Peut-être que moins vous apprenez de choses à un nouveau programmeur, plus créatif et plus astucieux il devra être en programmant. Puisque la plupart du temps programmer c'est résoudre des problèmes, il est préférable d'encourager cela, autant que faire se peut, à chaque étape.

Notes sur la version française

J'ai fait cette traduction parce que la manière de procéder de Chris Pine m'a beaucoup plu. J'ai respecté son humour et ses exemples en essayant, autant que j'ai pu, de trouver des équivalents en français. Il se peut qu'il y ait des erreurs (c'est très vraisemblable!), n'hésitez pas à me le faire savoir.
Courriel : anghel-pierre@wanadoo.fr

Je ne sais si vous êtes comme moi, mais pour travailler, lire ou apprendre il me faut un support papier, la lecture sur écran devenant vite fastidieuse. Ce qui fait que j'aurais bien aimé utiliser le programme Ruby dont parle l'auteur dans le chapitre précédent (excellent exercice de programmation d'ailleurs) en ne changeant que le texte anglais, mais deux choses m'ont retenu. Ce sont deux modifications de la présentation pour les raisons qui suivent:

- Le format : la version française a été portée en .pdf car ce format est plus facilement imprimable que le XHTML d'origine. La plupart des gens utilisent Internet Explorer, et obtenir une impression correcte avec "ça" relève de la mission impossible. J'ai même lu dans une revue le conseil d'utiliser une copie d'écran et d'imprimer ensuite à l'aide de Paint Shop Pro !!!
- Les exemples ont été entourés de cadres de couleur, mais sans couleur de fond, ceci afin de diminuer le coût de l'impression. Ma dernière imprimante m'a coûté 59 euros en 2003 (Epson C42+) mais une recharge noire et une recharge couleur (c'est à peu près ce qu'il m'a fallu pour imprimer l'original) reviennent à 45 euros. Sachant que le prix des cartouches augmente et leur contenance diminue, j'ai choisi l'économie.

En ce qui concerne la littérature française sur Ruby, le tour va être vite fait :
"Ruby in a nutshell" (si, si c'est en français!) chez O'Reilly. Il s'agit du manuel de référence.
Point.
Du moins pour les livres en vente chez les libraires.
Sur Internet c'est un peu plus fourni (Hummm!) :

- la section française de ruby-doc.org

Un mot sur SciTE, l'éditeur de texte. Je ne saurai trop vous le recommander car je le trouve excellent. Et n'hésitez pas à télécharger la dernière version disponible, qui n'est pas forcément celle fournie avec Ruby, sur le site: http://prdownloads.sourceforge.net/scintilla/scite160.zip?download (pour la version 1.60)
Un additif pour la traduction française est disponible au même endroit.

Si vous estimez avoir les capacités (et les idées) pour augmenter la taille de cet ouvrage dans le même esprit, n'hésitez pas et contactez-moi. Ruby mérite bien d'être plus connu en France. Et peut-être bien que la programmation française a besoin de Ruby...