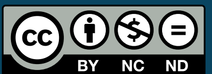
tleb
Karnaj
GaaH
Lalla
Dominus Carn...



Zeste de savoir

Une introduction à Ruby

22 mars 2019

Table des matières

I. Introduction 5

II. Remerciements 7

III. Les bases 9

1. Introduction 10
- 1.1. Qu'est-ce que Ruby ? 10
- 1.2. Installation 11
  - 1.2.1. Installation sous Windows 11
  - 1.2.2. Installation sous Linux 12
  - 1.2.3. Installation sous OS X 13
- 1.3. Prise en main 13
  - 1.3.1. IRB 13
  - 1.3.2. Opérations mathématiques 14
  - 1.3.3. Entiers et flottants 16
- 1.4. Exercices 17
- Contenu masqué 18

2. Variables et chaînes de caractères 19
- 2.1. Variables 19
  - 2.1.1. Déclaration de variables et opérations 19
  - 2.1.2. La valeur `nil` 20
  - 2.1.3. Constantes 21
  - 2.1.4. Conventions de nommage 21
- 2.2. Chaînes de caractères 22
  - 2.2.1. Opérations sur les chaînes de caractères 23
- 2.3. Conversion de variables 25
- 2.4. Saisir et afficher des informations 26
  - 2.4.1. Afficher des informations 27
  - 2.4.2. La méthode `gets` 28
  - 2.4.3. Les caractères spéciaux 29
- 2.5. Exercices 30
  - 2.5.1. Exercice 1 30
  - 2.5.2. Exercice 2 31
  - 2.5.3. Exercice 3 31
  - 2.5.4. Exercice 4 31
- Contenu masqué 32

3. Les conditions **35**
3.1. Opérateurs de comparaison . . . 35
3.1.1. Mise entre parenthèses . . . 36
3.2. La structure if-else . . . 37
3.2.1. Le mot-clé `elsif` . . . 38
3.2.2. La structure `unless` . . . 39
3.3. La structure case-when . . . 40
3.3.1. Les intervalles . . . 41
3.4. Les conditions ternaires . . . 42
3.5. Exercices . . . 43
3.5.1. Exercice 1 . . . 43
3.5.2. Exercice 2 . . . 43
3.5.3. Exercice 3 . . . 44
Contenu masqué . . . 45

4. Les boucles **48**
4.1. La boucle while . . . 48
4.1.1. La structure `begin-while` . . . 49
4.1.2. La structure `until` . . . 50
4.2. La boucle for . . . 51
4.3. Contrôler l'exécution d'une boucle . . . 52
4.3.1. Le mot-clé `break` . . . 52
4.3.2. Le mot-clé `next` . . . 53
4.3.3. Le mot-clé `redo` . . . 53
4.4. La boucle loop . . . 54
4.5. Les itérateurs . . . 54
4.5.1. La méthode `each` . . . 55
4.5.2. La méthode `times` . . . 55
4.5.3. La méthode `upto` . . . 56
4.5.4. La méthode `downto` . . . 56
4.5.5. La méthode `step` . . . 57
4.6. Exercices . . . 57
4.6.1. Exercice 1 . . . 57
4.6.2. Exercice 2 . . . 58
4.6.3. Exercice 3 . . . 58
Contenu masqué . . . 59

5. Les tableaux **61**
5.1. Généralités sur les tableaux . . . 61
5.1.1. Qu'est-ce qu'un tableau . . . 61
5.1.2. Déclarer un tableau . . . 61
5.2. Opérations sur les tableaux . . . 63
5.2.1. Accéder à un élément . . . 63
5.2.2. Concaténation et ajout d'éléments . . . 64
5.2.3. Parcourir le tableau . . . 66
5.3. Lien avec les chaînes de caractères . . . 67
5.3.1. Accéder à un élément . . . 68
5.3.2. Parcourir une chaîne de caractères . . . 68

5.3.3. De la chaîne au tableau 69
5.4. Exercices 70
5.4.1. Exercice 1 70
5.4.2. Exercice 2 70
5.4.3. Exercice 3 70
Contenu masqué 71

6. Les méthodes **74**
6.1. Principe et schéma d'une méthode 74
6.1.1. Qu'est-ce qu'une méthode ? 74
6.1.2. Déclarer une méthode 74
6.2. Écrire des méthodes 76
6.2.1. Procédure 76
6.2.2. Valeur de retour 77
6.2.3. Redéfinition de méthode 78
6.3. Les paramètres 79
6.3.1. Méthodes à un argument 79
6.3.2. Méthodes à plusieurs arguments 80
6.3.3. Valeurs par défaut 81
6.4. Exercices 82
6.4.1. Exercice 1 82
6.4.2. Exercice 2 82
6.4.3. Exercice 3 83
Contenu masqué 84

7. Les hachages **87**
7.1. Des tableaux associatifs 87
7.1.1. Qu'est-ce qu'un hachage 87
7.1.2. Déclarer un hachage 88
7.2. Opérations sur les hachages 89
7.2.1. Accéder à un élément 89
7.2.2. Ajout d'éléments 90
7.2.3. Parcourir le hachage 91
7.3. Hachages et tableaux 92
7.3.1. La clarté 93
7.3.2. L'ordre 93
7.3.3. La flexibilité 94
7.4. Exercices 94
Contenu masqué 96

8. Retour sur les variables **99**
8.1. Une histoire de références 99
8.1.1. Qu'est-ce qu'une variable? 99
8.1.2. Une histoire d'identifiant 100
8.1.3. Modification de variables 101
8.2. Les variables globales 102
8.2.1. Un problème: passage de variables aux méthodes 102
8.2.2. Les variables globales 105
8.2.3. Variables globales réservées 106

8.3. Les symboles . . . 106
8.3.1. Garder le même identifiant . . . 106
8.3.2. Les symboles et les chaînes de caractères . . . 107
8.3.3. Les symboles et les hachages . . . 108
8.4. Modifier nos variables dans des méthodes . . . 109
8.4.1. Utiliser le retour des méthodes . . . 109
8.4.2. Retourner plusieurs variables ? . . . 109
8.4.3. Copier un objet . . . 111

9. Le module Enumerable **113**
9.1. Les modules . . . 113
9.1.1. Les modules . . . 113
9.1.2. Utiliser un module . . . 114
9.1.3. Mettre un module dans un fichier séparé . . . 115
9.2. Le module Enumerable . . . 116
9.2.1. Vérifier des conditions sur un tableau . . . 117
9.2.2. Récupérer des éléments du tableau . . . 119
9.2.3. Autres méthodes utiles . . . 120
9.3. Exercices . . . 124
Contenu masqué . . . 125

10. Écrire le code dans des fichiers **127**
10.1. Les éditeurs de texte . . . 127
10.1.1. Pourquoi choisir un éditeur de texte? . . . 127
10.1.2. Des éditeurs de texte adaptés à la programmation . . . 127
10.1.3. Exécuter le code . . . 128
10.2. Windows — Notepad++ . . . 129
10.2.1. Installer Notepad++ . . . 129
10.2.2. Exécuter le code . . . 129
10.3. Linux — gedit . . . 133
10.3.1. Installer gedit . . . 133
10.3.2. Exécuter le code . . . 133
10.4. Les problèmes possibles . . . 134
10.4.1. Voir les erreurs obtenues . . . 134
10.4.2. La console se ferme directement . . . 134
10.4.3. Le programme ne se lance pas . . . 135

IV. Conclusion **136**

% UNE INTRODUCTION À RUBY % Karnaj, GaaH % 11 juin 2018

Première partie

Introduction

Vous rêvez d'apprendre à programmer, mais vous ne savez pas quel langage apprendre? Pourquoi ne pas apprendre le Ruby? Ruby est un langage de programmation **libre** et **dynamique**, qui met l'accent sur la **simplicité et la productivité**. Sa **syntaxe** élégante en facilite la lecture et l'écriture.

Prérequis
Connaître les quatre opérations mathématiques élémentaires.
Prérequis optionnels
Avoir déjà lu ce tutoriel qui introduit la programmation et aide à choisir un premier langage.
Connaître la notion de nombres relatifs.
Avoir des bases dans l'utilisation de la ligne de commande (savoir l'ouvrir et savoir entrer une commande).
Objectifs
Apprendre les bases de la programmation en Ruby.

L'apprentissage de la programmation peut-être une course pour certains. Mais cette idée n'est bonne que s'il s'agit d'une course d'**endurance**. En effet, le but n'est pas de finir le tutoriel le plus rapidement possible. Le but est de prendre son temps pour comprendre toutes les notions et surtout pour les **appliquer**. Seule la pratique permet de progresser, et il vaut mieux rester une semaine sur un chapitre pour maîtriser toutes les notions qui y sont traitées, plutôt que de passer une semaine sur tout le tutoriel.

Deuxième partie

Remerciements

Nous voulons remercier plusieurs personnes avant de commencer:

- baptisteguil le créateur originel et premier rédacteur de ce tutoriel;
- Titi_Alone et tleb pour leur participation;

Dominus Carnufex (https ://zestedesavoir.com/membres/voir/Dominus Carnufex/) pour son travail monstrueux et très rapide de corrections orthotypographiques;

- Arius pour son formidable travail de validation;
- Tous les membres qui ont apporté leurs conseils, leurs avis et leurs corrections.

Troisième partie

Les bases

1. Introduction

Dans ce chapitre, nous allons découvrir ce qu'est Ruby, comment l'installer, et nous ferons une courte introduction à sa syntaxe.

1.1. Qu'est-ce que Ruby ?

Avant de nous lancer dans cette grande aventure qu'est l'apprentissage de Ruby, nous devons déjà savoir ce qu'est Ruby.

Pour commencer, Ruby est un langage de programmation.

Mais au fait, qu'est-ce qu'un langage de programmation?

En lisant ce tutoriel, nous avons une réponse à cette question.
Pour résumer, un **langage de programmation** est un langage destiné à écrire des **programmes informatiques**.

Il existe deux grandes catégories de langages:

- avec les **langages compilés**, le code que nous écrivons passe par un compilateur et ce compilateur génère un **exécutable** qui peut ensuite être exécuté par notre ordinateur;
- avec les **langages interprétés**, le code que nous écrivons doit être lancé dans ce que l'on appelle une **machine virtuelle**, il n'y a pas d'exécutable de créé.

Ruby est un langage interprété. Nous aurons donc besoin d'une machine virtuelle pour l'exécuter. Sans elle, les programmes ne fonctionneront pas. Cela signifie également que si nous voulons distribuer un programme écrit en Ruby, il faudra que les personnes qui utilisent notre programme aient cette machine virtuelle d'installée (cette machine virtuelle **est** en fait Ruby). Ils devront donc télécharger et installer Ruby (ou nous devrons distribuer Ruby avec notre programme).

Cette distribution est possible parce que Ruby est également un langage (et un logiciel) libre. Nous n'allons pas être longs sur ce point (pour plus d'informations, voir la licence GPL), mais retenons en gros que nous pouvons utiliser et redistribuer Ruby gratuitement et même créer des programmes payants avec.

Ruby est un langage complet et permet de faire beaucoup de choses (on peut par exemple travailler sur un site Web en utilisant **Ruby On Rails**). De plus, il permet de faire les choses de plusieurs manières différentes. C'est l'une de ses forces, mais paradoxalement, c'est aussi l'une des choses qui lui sont parfois reprochées: on peut faire les choses de plusieurs manières différentes, quelle méthode faut-il alors utiliser?

En tout cas, il reste un langage pratique, facile à utiliser et il est même portable. Nous pourrons donc l'utiliser sur différents systèmes.

1.2. Installation

Avant de coder en Ruby, il va de soi qu'il faut l'installer. Heureusement, nous allons voir pas à pas comment faire.

Il y a plusieurs façons d'installer Ruby. Elles sont détaillées sur cette page . Nous allons voir la méthode la plus simple.

1.2.1. Installation sous Windows

Pour commencer, téléchargeons la dernière version de RubyInstaller sur cette page .

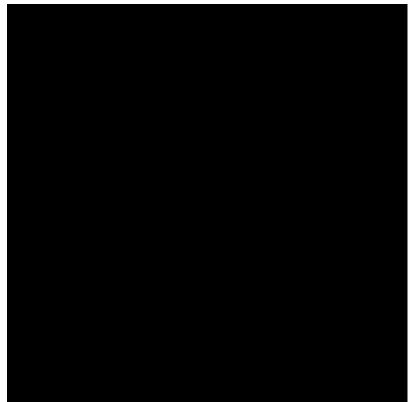
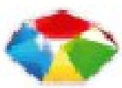


Figure : Capture du site.

Ensuite, il nous suffit d'exécuter l'installateur téléchargé.
Il nous faut cliquer sur « Suivant » → « Suivant » → « Installer » en prenant soin de cocher les cases comme ci-dessous.

Installation Destination and Optional Tasks



Setup will install Ruby 2.2.2-p95 into the following folder. Click Install to continue or click Browse to use a different one.

Please avoid any folder name that contains spaces (e.g. Program Files).

C:\Ruby22 | Browse...

- [ ] Install Tcl/Tk support
- [x] Add Ruby executables to your PATH
- [x] Associate .rb and .rbw files with this Ruby installation

TIP: Mouse over the above options for more detailed information.

Required free disk space: ~42,8 MB

< Back | Install | Cancel

Figure : Exécutable.

Pour vérifier que Ruby est bien installé, nous pouvons utiliser la console. Ouvrons l'application «Invite de commande» et tapons `ruby -v` avant de valider en appuyant sur Entrée. Nous sommes censés obtenir le numéro de la version de Ruby qui est installée.

1.2.2. Installation sous Linux

Ruby est déjà installé sur certaines distributions. Pour savoir s'il est déjà installé, nous pouvons utiliser la commande `ruby`. Pour cela, il faut ouvrir un terminal (généralement le terminal est l'application «Konsole» ou encore «Terminal»). Après l'avoir ouvert, tapons `ruby` avant de valider en appuyant sur Entrée.

Si la commande ne renvoie pas d'erreur, alors Ruby est installé (Notons également la commande `ruby -v` qui renvoie le numéro de version).

Si Ruby n'est pas encore installé, le plus simple est d'utiliser notre gestionnaire de paquets pour l'installer. Nous pouvons utiliser un gestionnaire de paquets graphiques ou en ligne de commandes au choix.

```
1 $ sudo apt-get install ruby-full # Sous Ubuntu, Linux Mint, Debian,
    etc.
2 $ sudo pacman -S ruby            # Sous Arch Linux.
```

1.2.3. Installation sous OS X

Nous avons de la chance, depuis quelques versions, Ruby est installé de base sur OS X. Nous n'avons donc rien à faire.
Lançons quand même la commande `ruby -v` dans notre terminal pour connaître la version qui est installée. Pour cela, ouvrons le terminal (il s'agit de l'application «Terminal.App»), tapons `ruby -v` et validons en appuyant sur Entrée. Nous obtenons le numéro de la version de Ruby qui est installée.

Ruby est maintenant installé sur notre ordinateur. Il nous faut maintenant apprendre à l'utiliser.

1.3. Prise en main

1.3.1. IRB

Ruby possède ce que l'on appelle un environnement interactif, ici appelé IRB.
IRB interprète notre code au fur et à mesure que nous l'écrivons: nous entrons une instruction et nous appuyons sur Entrée, cette instruction est alors exécutée.

> Pour lancer IRB, il faut passer par la ligne de commande que nous avons déjà utilisée précédemment. Il s'agit de l'application «Invite de commande» sous Windows et de «Terminal.app» sous OS X. Une fois l'invite de commande ouverte, il faut taper `irb`. Nous avons maintenant une magnifique console qui interprète ce que nous écrivons en Ruby.

En lançant IRB, nous nous retrouvons face au symbole `>`. Ce chevron signifie une chose: IRB est à nos ordres, il attend nos instructions. Lorsqu'il nous «répond», sa réponse est située après le signe `=>` sur une nouvelle ligne. Par exemple, en tapant `2` il répond `2` (oui, il répond bien `2`).

```
1 > 2
2 => 2
```

En Ruby, les instructions sont séparées par un point-virgule. Celui-ci peut cependant être omis s'il n'y a qu'une instruction par ligne.

Ainsi, nous pouvons écrire cela.

```
3; 2
```

Mais ceci ne fonctionnera pas et nous donnera une belle erreur.

```
3 4
```

Nous conseillons alors de ne mettre qu'une seule instruction par ligne.

Pour quitter IRB, nous devons entrer `quit` et valider.

1.3.2. Opérations mathématiques

Les premières instructions que nous allons voir sont les opérations mathématiques. IRB fonctionne comme une calculatrice et nous donne le résultat à la ligne juste en dessous. Il respecte même les priorités (avec ou sans parenthèses). Essayons d'écrire des opérations mathématiques. Nous obtenons quelque chose de ce genre.

```
> 3 + 2
=> 5
> 2 * 3 + 5
=> 11
> (2 * 3) + 5
=> 11
> (6 - 2)*(8 + 25)
=> 132
```

Nous pouvons même assigner des valeurs à une lettre pour effectuer des opérations sur celle-ci.

```
> x = 4
=> 4
> y = 7
=> 7
> x + y
=> 11
```

> ! Il faut faire attention aux divisions. L'invite donne toujours un résultat entier: par exemple, `5 / 3 = 1`.

Nous allons voir quel est le problème de la division. Mais avant cela, voici un tableau des opérations mathématiques que nous avons vues.

Action	Code	Affichage
Additionner	`3 + 2`	Affiche `5`.
Soustraire	`3 - 2`	Affiche `1`.
Multiplier	`3 * 2`	Affiche `6`.
Diviser	`3 / 2`	Affiche `1`.
Modulo	`3 % 2`	Affiche `1` (le reste de la division euclidienne).

Nous pouvons également y ajouter l'opération **puissance** qui s'utilise avec `**`. En tapant `2 ** 2`, on obtient donc `4`.

1.3.2.1. Un peu de sucre syntaxique

L'expression sucre syntaxique désigne des parties d'un langage de programmation qui permettent d'écrire et de lire du code plus facilement. À ce niveau du cours, nous n'avons pas encore de quoi voir du sucre syntaxique, mais il existe des raccourcis aux opérateurs mathématiques que l'on peut considérer comme du sucre syntaxique.

Prenons l'exemple d'une variable `variable` qui vaut 4. On veut la multiplier par 3. Le code qui s'impose estle suivant.

```
variable = 4
variable = variable * 3
```

Ceci est parfaitement correct. Cependant, on peut raccourcir cette écriture en écrivant ceci.

```
variable = 4
variable *= 3
```

Et cette syntaxe ne marche pas que pour la multiplication. Voici un tableau des opérations et de leurs équivalents.

Opération	Équivalent
`variable = variable + nombre`	`variable += nombre`

`variable = variable - nombre`	`variable -= nombre`
`variable = variable * nombre`	`variable *= nombre`
`variable = variable / nombre`	`variable /= nombre`
`variable = variable % nombre`	`variable %= nombre`

Notons de plus que nous pouvons utiliser le symbole `_` dans nos nombres. Ainsi, `12_3_2` et `1232` sont les mêmes nombres. Ceci nous permet de regrouper certains chiffres, donc d'avoir une meilleure lisibilité. Par exemple, `10_000_000` est beaucoup plus lisible que `10000000`.

Nous avons également la possibilité de noter les nombres en binaire (grâce au préfixe `0b`), en octal (grâce aux préfixes `0` et `0o`) et en hexadécimal (grâce au préfixe `0x`). En fait, il y a aussi le préfixe `0d` pour écrire les nombres en décimal.

```
1 > 10   => 10
2 > 0d10 => 10
3 > 0b10 => 2
4 > 010  => 8
5 > 0o10 => 8
6 > 0x10 => 16
7 > 0xab => 171
```

Nous pouvons écrire les lettres des préfixes en majuscule ou en minuscule. De même, Ruby n'est pas sensible à la casse pour la notation hexadécimale. Ainsi, `0B10` et `0b10` sont équivalents, de même que `0xAb1` et `0XAB1`.

1.3.3. Entiers et flottants

En voyant ce que sont les entiers et les flottants, nous verrons qu'en fait, le résultat que nous obtenons avec la division n'est pas une erreur.

On appelle un entier un nombre comme 3, 37 ou -12. Un flottant est un nombre avec une partie décimale comme 5, 6 ou 32,5.

Notre problème est directement lié aux notions d'entiers et de flottants: en fait, quand on tape `3 / 2`, on divise deux entiers. Le résultat est donc aussi un entier. En toute logique, pour obtenir un résultat à virgule (donc un flottant), il faut diviser deux flottants. Donc pour obtenir le résultat attendu, il nous aurait fallu taper `3.0 / 2.0`. Entrons `3.0 / 2.0` dans IRB, et admirons le résultat.

> **!** Nous n'avons pas pu le manquer. Pour écrire un nombre décimal, on utilise la notation anglo-saxonne. Il faut donc utiliser un point et non une virgule! De plus, nous sommes obligés de rajouter la partie décimale même si elle est nulle pour préciser qu'on utilise un flottant.

Lorsque nous additionnons un entier et un flottant, nous obtenons un flottant. Ainsi, en tapant `2 + 3.5`, on obtient bien `5.5`. La conversion est faite implicitement.

Notons que nous pouvons utiliser la notation `E` pour les puissances de 10 (là encore, la casse n'a pas d'importance). Ainsi, `12e2` vaut `1200` et `1.2E-2` vaut `0.012`.

Il nous faut faire des essais, regarder les résultats des calculs et nous habituer à les utiliser. Nous en aurons besoin au prochain chapitre!

1.4. Exercices

Faisons quelques exercices avant de passer au chapitre suivant. Voyons si la notion de flottants est bien comprise. Nous allons donner quelques calculs. Le but de l'exercice est de deviner le résultat, avant de l'exécuter pour vérifier le résultat. Simple, non?

Commençons :

— `2 + 3` ;
— `3.2 + 4` ;
— `4 / 3` ;
— `4 / 3.0` ;
— `4,0 / 3,0`.

Jusque là, il n'y a que de petits calculs. Passons à d'autres choses:

— `4.0 % 2.4`;
— `5 * (2 + 5)`;
— `3 + 2 * (2 + 5))`;
— `3(2 + 2) * 3`.

Correction :

Contenu masqué n°1

Nous avons maintenant tous les outils pour nous lancer dans la programmation en Ruby. Pour le moment, nous n'avons rien fait d'extraordinaire, mais c'est quand même une base à avoir, et il ne faut pas oublier de nous entraîner.

— On peut exécuter des instructions via IRB, qui interprète le code Ruby que nous écrivons.
— Ruby nous permet de faire des opérations mathématiques élémentaires (addition, soustraction, multiplication, division) mais aussi le modulo (reste de la division euclidienne) et la puissance.

— Les nombres peuvent être notés de plusieurs manières (plusieurs bases sont disponibles, l'écriture scientifique est disponible, etc.).
— On n'utilise pas la virgule mais le point comme séparateur décimal.
— Les entiers sont à différencier des flottants (notamment une division d'entiers donne un entier) même si les entiers sont automatiquement convertis en flottants lors d'une opération avec un flottant.

Contenu masqué

Contenu masqué n°1

— `2 + 3` donne `5` comme résultat (c'est une addition de deux entiers);
— `3.2 + 4` donne `7.2`;
— `4 / 3` donne `1` (c'est une division entre entiers le résultat est donc un entier);
— `4 / 3.0` donne `1.3333` (lorsque l'on fait une opération entre un entier et un flottant, le résultat est un flottant);
— `4,0 / 3,0` donne une erreur car le séparateur décimal est le point et non la virgule.

Et pour les autres:

— `4.0 % 2.4` donne étonnement `1.6`. Le reste est calculé pour une division de flottants quand avec d'autres langages on obtiendrait une erreur ;
— `5 * (2 + 5)` donne `35`: le calcul entre parenthèses est effectué avant la multiplication. Le calcul effectué est donc `5 * 7` ;
— `3 + 2 * (2 + 5))` donne une erreur: une des parenthèses n'est pas ouverte ;
— `3(2 + 2) * 3` donne aussi une erreur: nous ne pouvons pas ne pas écrire de signes, le `*` doit être précisé.

Retourner au texte.

2. Variables et chaînes de caractères

Dans ce chapitre, nous allons aborder un concept important dans tous les langages: les variables.

2.1. Variables

Nous allons aborder le sujet des variables, mais il faut savoir que nous avons déjà déclaré deux variables dans le chapitre précédent: `x` et `y`.

Mais, c'est quoi une variable?

Une variable est une donnée de notre programme qu'on pourra modifier, lire, etc. En fait, dans notre programme, ce sera juste un nom (on parle d'**identifiant**) auquel on associe une valeur, un mot... On peut donc avoir une variable qui s'appelle `âge` et qui représenter l'âge de l'utilisateur. Dans le chapitre précédent, nous avons utilisé `x` et `y` afin de faire des calculs. Plus loin dans le tutoriel, nous reviendrons sur la définition des variables et verront plus précisément comment elles se présentent. Pour le moment, nous nous contenterons de les utiliser.

2.1.1. Déclaration de variables et opérations

Comme nous avons pu le voir, la déclaration d'une variable est enfantine: un simple signe égal. Nous pouvons ensuite effectuer les mêmes opérations avec vos variables qu'avec des nombres: additions, soustractions, multiplications et divisions.

```
> x = 3
=> 3
> y = 2
=> 2
> x + y
=> 5
> x - y
=> 1
```

On peut changer la valeur d'une variable au cours d'un programme. On dit qu'on lui **assigne** une valeur. Une variable a pour valeur la dernière valeur qui lui a été assignée. Par exemple, regardons ce code.

```
1 > x = 3
2 => 3
3 x = 5
4 => 5
```

Après cela, la variable `x` ne vaudra plus 3 mais vaudra bien 5.

Nous pouvons récupérer le résultat d'une opération entre variables dans une nouvelle variable (ou même dans l'une des variables avec laquelle nous faisons l'opération).

```
1 > a = 3
2 => 3
3 > b = 2
4 => 2
5 > c = a + b
6 => 5
```

`c` vaut 5.

```
1 > a = 3
2 => 3
3 > b = 2
4 => 2
5 > a = a + b
6 => 5
```

`a` vaut 5.

2.1.2. La valeur `nil`

Que se passe-t-il lorsque nous essayons d'utiliser une variable sans lui donner une valeur en la déclarant? Voici ce que l'on obtient (pour déclarer la valeur sans l'initialiser, il suffit d'écrire `variable =`).

```
1 > variable =
2 > variable
3 => nil
```

? Que signifie cette valeur `nil` ?

`nil` est une valeur particulière qui signifie l'absence de valeur, justement. Notre valeur n'a ici aucune valeur.

Pour le moment, savoir cela ne nous est pas très utile, mais gardons cette idée en tête: `nil` représente l'absence de valeur.

2.1.3. Constantes

Une constante est une variable dont la valeur ne peut pas être modifiée. Elle reste la même pendant tout le programme.

Pour déclarer une constante, il nous faut écrirececi.

```
> PI = 3.14159265359
=> 3.14159265359
```

En fait, il suffit de commencer le nom d'une variable par une majuscule. C'est imposé par le langage. Ainsi, nous ne pouvons pas commencer le nom d'une variable normale par une majuscule, car nous aurons alors déclaré une constante. En essayant de modifier la valeur d'une constante nous aurons un avertissement (mais la valeur sera quand même changée). Ainsi si j'essaye de modifier la constante `PI`, j'obtiensce message.

```
warning: already initialized constant PI
warning: previous definition of PI was here
```

Notons que nous avons écrit le nom de la constante tout en majuscules, entre autres parce que c'est la convention de nombreux langages, mais aussi pour pouvoir la repérer plus facilement.

2.1.4. Conventions de nommage

Nous avons commencé à parler de conventions de nommage pour les constantes. Continuons à en parler. Tout d'abord, nous devons préciser que tous les noms de variable ne sont pas possibles. Il y a des règles:

- un nom de variable ne peut pas commencer par un chiffre;
- un nom de variable ne peut pas contenir de signe de ponctuation;
- un nom de variable ne peut pas contenir de symbole opératoire (`+`, `-`, `*`, `/`, `%`).

À cela, il faut ajouter qu'un certain nombre de mots ne peuvent pas être utilisés comme noms de variable. Ce sont des mots réservés pas Ruby. Ils font partie du langage et ont un sens propre qui fait qu'on ne peut pas les utiliser. Les voici.

`=begin`	`break`	`elsif`	`module`	`retry`	`unless`
`=end`	`case`	`end`	`next`	`return`	`until`

BEGIN	class	ensure	nil	self	when
END	def	false	not	super	while
alias	defined?	for	or	then	yield
and	do	if	redo	true	__FILE__
begin	else	in	rescue	undef	__LINE__

De plus, il est conseillé d'écrire ses variables en « snake_case » . Cela signifie que toutes vos variables seront écrites en minuscules et que les changements de mot seront signalés par un tiret bas.

Les constantes, elles, doivent être écrites en « SCREAMING_SNAKE_CASE » . Elles doivent donc être écrites en majuscule, les changements de mot signalés par un tiret bas.

```
ma_super_variable
MA_SUPER_CONSTANTE
```

Finalement, et là il ne s'agit que de logique simple, il faut donner à nos variables des noms clairs et explicites. Il faut par exemple préférer `nom` à `n`. Évitons également de mélanger les noms en anglais et en français.

2.2. Chaînes de caractères

Pour afficher une chaîne de caractères (soit un mot ou une phrase), il faut l'entourer de guillemets simples ou doubles.

```
> "Bonjour"
=> "Bonjour"
> 'Bonjour'
=> "Bonjour"
```

Pour déclarer une variable contenant une chaîne de caractères, nous devons toujours utiliser le signe «=». Il faut cependant faire attention à ne pas oublier les guillemets. De la même manière que pour les nombres, nous devons choisir un **identifiant** et lui assigner une valeur. De plus, nous pouvons également déclarer des chaînes de caractères constantes, toujours en écrivant la première lettre de l'identifiant en majuscule.

```
> mot = 'zeste'
=> "zeste"
> Constante = 'vrai'
```

```
4 => "vrai"
```

Ici, la variable `mot` a alors pour valeur la chaîne de caractères `zeste`.

Nous pouvons également utiliser les syntaxes `%()`, `%q()` et `%Q()` pour déclarer une chaîne de caractères.

```
1 > mot = %(zeste)
2 => "zeste"
3 > mot = %q(zeste)
4 => "zeste"
5 > mot = %Q(zeste)
6 => "zeste"
```

Nous pouvons remplacer les parenthèses par des crochets, des accolades, ou encore des chevrons. En fait, nous pouvons utiliser plusieurs caractères non alphanumériques comme `%`, `|`, `^`, `*`, etc. On a alors équivalence entre `%()`, `%||` et `%[]`.

2.2.1. Opérations sur les chaînes de caractères

Nous pouvons effectuer quelques opérations sur les chaînes de caractères.

```
1 > 'Bonjour ' + 'tout le monde'
2 => "Bonjour tout le monde"
```

Cette opération s'appelle la **concaténation** (on dit qu'on **concatène** les deux chaînes). Elle permet de mettre deux chaînes bout à bout. Nous pouvons aussi insérer une variable dans une chaîne de caractères grâce à la concaténation.

```
1 > mot = 'zeste'
2 => "zeste"
3 > 'Faites un ' + mot + ' envers votre prochain.'
4 => "Faites un zeste envers votre prochain."
```

Ici, nous avons concaténé trois chaînes de caractères.

La concaténation se fait avec l'opérateur `+`. Si nous voulons obtenir plusieurs fois la même chaîne, nous pouvons utiliser l'opérateur `*`.

```
1 > mot = 'zeste '
2 => "zeste "
```

```
> mot * 3
=> "zeste zeste zeste "
```

La même chaîne a été concaténée trois fois.

Une autre opération utile permet de rajouter une chaîne à la fin d'une chaîne de caractères. On peut aussi dire que c'est une concaténation. Cependant, le résultat de cette concaténation est directement affecté à la variable de départ.

```
> mot = 'zeste'
=> "zeste "
> mot << ' de savoir'
=> "zeste de savoir"
```

La variable `mot` vaut alors `zeste de savoir`. Notons que cette concaténation ne s'utilise pas qu'avec des variables. Il est possible d'écrire `'zeste' < ' de savoir'`.

> *i* Tout comme pour les autres variables, nous pouvons récupérer le résultat de l'opération dans une variable.

Ainsi…

```
> mot = 'zeste '
=> "zeste "
> répété = mot * 3
=> "zeste zeste zeste "
```

La variable `répété` vaut `zeste zeste zeste`.

Et vérifions que nous pouvons bien écrire `'zeste' < ' de savoir'`.

```
> mot = 'zeste' << ' de savoir'
=> "zeste de savoir"
```

On obtient alors la valeur `"zeste de savoir"` pour la variable `mot` sans étape intermédiaire. Il est conseillé d'utiliser `<` plutôt que `+`. En effet, `<` est plus rapide, car il modifie la chaîne de caractères directement.

Après la concaténation, parlons de l'interpolation de chaînes de caractères. Elle permet tout comme la concaténation de «rajouter des données dans une chaîne», mais s'avère parfois plus efficace. L'interpolation s'effectue de cette manière: on utilise `#` suivi de la variable entre accolades à l'endroit de la chaîne où nous voulons l'ajouter. Un exemple.

```
> variable = 'de savoir'
=> "de savoir"
> mot = "zeste #{variable}"
=> "zeste de savoir"
```

L'interpolation marche avec tous les types de variables que nous avons vus, pas seulement avec les chaînes de caractères. On peut même effectuer des calculs. Notons qu'il est conseillé de ne pas mettre d'espaces autour du code interpolé.

```
> variable = 123
=> 123
> "123 * 2 = #{variable * 2}"
=> "123 * 2 = 246"
> "123 + 2 = #{123 + 2}"
=> 123 + 2 = 125"
```

> ! L'interpolation ne s'utilise qu'avec les guillemets doubles. Avec les guillemets simples, nous obtiendrons «#{variable}».

Il faut privilégier l'interpolation et ne pas trop utiliser la concaténation.

2.3. Conversion de variables

En Ruby, nous pouvons convertir des variables (changer le type de la variable) en utilisant une **méthode**.
Nous verrons ce qu'est une **méthode** dans un autre chapitre. Pour le moment, nous pouvons considérer qu'une méthode nous permet d'effectuer une action. Ici nous devrons juste mettre à la fin de notre variable l'une de ces expressions.

Code	Action
`.to_s`	Convertir en chaîne de caractères (`string`).
`.to_i`	Convertir en entier (`int`).
`.to_f`	Convertir en flottant (`float`).

Exemple.

```
> x = 3
=> 3
```

```
3  > x.to_s
4  => "3"
```

Si nous essayons de convertir une chaîne de caractères qui n'est pas un nombre en entier (`int`) ou en flottant (`float`), nous obtiendrons 0 pour la conversion en entier, et 0.0 pour la conversion en flottant.

```
1  > x = 'Bonjour'
2  => "Bonjour"
3  > x.to_i
4  => 0
5  > y = 'zeste'
6  => "zeste"
7  > y.to_f
8  => 0.0
```

Alors que…

```
1  > x = '3'
2  => "3"
3  > x.to_i
4  => 3
5  > x.to_f
6  => 3.0
```

Cette conversion peut s'avérer très utile lorsque nous voulons par exemple concaténer un nombre à une chaîne de caractères.

Cependant, cette conversion ne modifie pas la variable. Elle nous donne une nouvelle valeur que nous pouvons récupérer dans une variable, mais la variable initiale garde son ancienne valeur.

2.4. Saisir et afficher des informations

À partir de maintenant, nos programmes seront plus conséquents et le nombre de lignes sera donc plus important. Le chapitre «Écrire le code dans des fichiers» nous apprendra à écrire notre code dans des fichiers.

Dès à présent dans nos exemples, nous utiliserons des commentaires. Les commentaires sont des messages qui sont ignorés par l'interpréteur et qui permettent de donner des indications sur le

code (car relire ses programmes après plusieurs semaines d'abandon, sans commentaires, peut parfois être plus qu'ardu).

En Ruby, un commentaire débute par un croisillon (`#`) et se termine par un saut de ligne. Tout ce qui est compris entre ce `#` et ce saut de ligne est ignoré. Un commentaire peut donc occuper la totalité d'une ligne (on place le `#` en début de ligne) ou une partie seulement, après une instruction (on place le `#` après la ligne de code pour la commenter plus spécifiquement).

Nous pouvons également faire des commentaires multi-lignes. Pour cela, il faut placer notre commentaire entre `=begin` et `=end`. Mais nous n'allons pas les utiliser et allons plutôt utiliser plusieurs lignes commençant par `#` pour écrire un commentaire sur plusieurs lignes.

2.4.1. Afficher des informations

Depuis le début, nous utilisons IRB pour programmer en Ruby. Cependant, maintenant, lorsque nous n'utiliserons pas IRB et mettrons nos programmes dans des fichiers, il nous faudra un moyen d'afficher les résultats obtenus et le contenu de nos variables. Pour cela, nous allons encore une fois utiliser une **méthode**.

Nous avons deux méthodes pour afficher des informations, `puts` et `print`. Elles ont un fonctionnement similaire (les deux affichent ce qui est demandé sur la sortie standard qui est par défaut la console) à la différence près que contrairement à `print`, `puts` va à la ligne après avoir effectué l'affichage. Leur utilisation est par contre parfaitement similaire. Nous allons donc voir l'exemple de `puts`.

Pour afficher une chaîne de caractères ou un nombre voici le code à utiliser.

```
1 puts "Chaîne"
2 puts 2
3 puts 3.5
```

Ces lignes de codes vont afficher « Chaîne », aller à la ligne, afficher 2, aller à la ligne et finalement afficher 3.5 et… aller à la ligne.

Nous pouvons également afficher le contenu d'une variable de la même manière.

```
1 a = 3
2 puts a
```

Ce programme nous affichera 3 et un saut de ligne.

Notons finalement qu'en utilisant la virgule pour séparer des éléments, la méthode est appelée plusieurs fois et affiche chacun des éléments. Le code `print 'Bonjour', 2, 45.8` équivaut à ce code.

```ruby
print 'Bonjour'
print 2
print 45.8
```

Ceci nous sera très utile. Cependant, lorsque nous voulons afficher une chaîne de caractères et des variables, il vaut mieux utiliser l'interpolation de chaînes de caractères. Ainsi, nous écrirons…

```ruby
print "Bonjour #{variable1} #{variable2}"
```

… plutôt que…

```ruby
print 'Bonjour'
print variable1
print variable2
```

Les méthodes `puts` et `print` ne sont pas très compliquées. Nous allons maintenant voir comment demander des informations à l'utilisateur.

2.4.2. La méthode `gets`

Si nous voulons demander des informations à l'utilisateur, la méthode `gets` est toute désignée. Voici comment l'utiliser.

```ruby
print 'Entrez votre nom : '
prénom = gets.chomp
print "Bonjour #{prénom}"
```

Le mieux est d'essayer. Nous devrions obtenir une sortie ressemblant àceci.

```
Entrez votre prénom : Zesteur
Bonjour Zesteur
```

Après `prénom = gets.chomp`, le programme se stoppe et attend une saisie. Il suffit de rentrer l'information à la ligne suivante et de valider en appuyant sur Entrée.

Pourquoi rajouter `.chomp` après le `gets`?

En fait, c'est la méthode `gets` qui permet de demander une saisie à l'utilisateur. Elle nous renvoie une chaîne de caractères. Essayons donc `prénom = gets`. Affichons maintenant `prénom`. Et là, nous voyons le souci: `gets` nous laisse un `\n`. Nous ne voulons pas nous embarrasser d'un caractère gênant. Autant le supprimer tout de suite. `chomp` s'applique à une chaîne de caractères (ici la chaîne renvoyée par `gets`) et supprime ce `\n`. Ainsi, nous aurions également pu fairececi.

```
1 prénom = gets
2 prénom = prénom.chomp
```

Ici, on récupère d'abord la saisie de l'utilisateur et seulement à la ligne d'après on supprime le `\n`.

> !
> `gets.chomp` renvoie une chaîne de caractères. Si on veut par exemple faire des calculs, il faudra faire une conversion.

2.4.3. Les caractères spéciaux

Ce `\n`, que nous venons de croiser, qu'est-ce que c'est? En fait, `\n` est un caractère spécial qui représente un retour à la ligne. On parle de **séquences d'échappement** (les caractères sont échappés par le caractère `\`). Nous pouvons même l'utiliser pour aller à la ligne dans nos affichages. Par exemple...

```
1 print "zeste\n"        # Affiche « zeste » et va à la ligne.
2 print "zeste\n savoir" # Affiche « zeste », va à la ligne et
      affiche « savoir » .
```

Les séquences d'échappement peuvent être très utiles. Voyons quelques autres séquences d'échappement.

Nom	Description	Code de test
`\a`	**Fait un bip.**	**`print "\a"`**
`\b`	Efface le caractère précédent.	`print "x\b"`
`\n`	Va à la ligne.	`print "x\nz"`
`\r`	Retourne en début de ligne.	`print "xxx \rzzz"`
`\s`	Affiche un espace.	`print "x \s z"`
`\t`	Affiche une tabulation horizontale.	`print "x \t z"`

> **!** Puisque `\` permet d'échapper des caractères pour obtenir des caractères spéciaux, si nous voulons afficher un antislash, il nous faudra utiliser `\\`.

Notons que les caractères spéciaux ne sont compris qu'avec les guillemets doubles (on dit qu'ils sont échappés). `print 'Bonjour\nzz'` affichera `Bonjour\nzz`. Le seul caractère qui est échappé avec les guillemets simples est `\'` qui permet d'afficher des guillemets simples. De même, `\"` permet d'afficher des guillemets doubles lorsqu'on les utilise pour délimiter la chaîne de caractère.

Les bonnes pratiques de Ruby veulent que l'on garde un style de guillemets constant. Voici deux styles que l'on peut utiliser.

1. Préférer les chaînes à guillemets simples lorsqu'il n'y a pas d'interpolation, de caractères spéciaux ou de guillemets simples.
2. Préférer les chaînes à guillemets doubles lorsque la chaîne ne contient pas de caractères à échapper ou de guillemets doubles.

L'important est de choisir une règle et de la respecter.

Certaines bonnes pratiques sont également posées sur l'utilisation des syntaxes `%`. Les syntaxes `%` et `%Q` sont équivalentes (`%` est un raccourci) et correspondent à des guillemets doubles, la syntaxe `%q` à des guillemets simples. On a alors ces règles:

— il faut utiliser `%` pour les chaînes de caractères qui contiennent le caractère `"` et de l'interpolation;
— il ne faut pas utiliser `%q` à moins que notre chaîne ne contienne les caractères `"` et `'`;
— il faut préférer `()` comme délimiteur (on écrira `%(une chaîne avec écrit "#{variable}")` plutôt que `%{une chaîne avec écrit "#{variable}"}`).

2.5. Exercices

C'est l'heure de pratiquer. Nous allons donc faire quelques petits exercices.

2.5.1. Exercice 1

Commençons par un petit exercice: demandons à l'utilisateur son nom et son prénom et affichons «Bonjour nom prénom» suivi d'un retour à la ligne.

Correction.

> Contenu masqué n°2

2.5.2. Exercice 2

Nous devons maintenant demander à l'utilisateur d'entrer deux nombres `x` et `y`, échanger leurs valeurs, et afficher leur nouvelle valeur. Voici ce que nous devons obtenir.

```
Entrez x : 23
Entrez y : 12
L'échange a été effectué : x vaut 12 et y vaut 23.
```

! Le but est de vraiment échanger les valeurs des deux variables, pas juste d'afficher la valeur de l'une pour l'autre.

Correction.

Contenu masqué n°3

2.5.3. Exercice 3

Cette fois, nous devons demander à l'utilisateur son âge en années (un entier) et afficher sur différentes lignes:

- son âge en années;
- son âge en jours (on va considérer que toutes les années ont 365 jours);
- son âge en heures;
- son âge en minutes;
- son âge en secondes.

Oui, ça va demander plusieurs calculs. Mais c'est bien ça le but.

Correction.

Contenu masqué n°4

2.5.4. Exercice 4

Finissons avec un exercice pour bien travailler les conversions: demandons à l'utilisateur deux nombres, puis affichons sur des lignes différentes le résultat de l'addition, de la soustraction, de la multiplication et de la division du premier nombre par le deuxième. Si l'utilisateur rentre 6 et 4, le résultat attendu estcelui-ci.

```
6.0 + 4.0 = 10.0
6.0 - 4.0 = 2.0
6.0 * 4.0 = 24.0
6.0 / 4.0 = 1.5
```

L'affichage nous donne déjà un indice: nous devrons utiliser des flottants.

Correction.

Contenu masqué n°5

Maintenant que nous maîtrisons les variables, nous allons passer au chapitre sur les conditions et les boucles.

— Ruby nous permet de déclarer des variables et des constantes, mais certains noms sont interdits. Ces variables peuvent être des nombres, mais aussi des chaînes de caractères.
— Plusieurs opérations sont possibles sur les chaînes de caractères (concaténation, interpolation) et il y a plusieurs manières de les déclarer (en fonction des situations, nous allons en privilégier certaines).
— Les méthodes `print` et `puts` permettent d'afficher quelque chose sur la sortie standard (par défaut la console). Certains caractères sont spéciaux et doivent être échappés pour être affichés. Les guillemets simples et doubles ne se comportent pas de la même manière face aux caractères spéciaux et à l'interpolation.
— La méthode `gets` permet de demander à l'utilisateur de rentrer une chaîne de caractères. Nous pouvons ensuite la convertir en nombre si besoin en utilisant une méthode comme `to_i`.

Contenu masqué

Contenu masqué n°2

Nous allons utiliser deux variables `nom` et `prénom` et demander leur valeur à l'utilisateur, puis les afficher grâce à une interpolation.

```
print 'Entrez votre nom : '
nom = gets.chomp
print 'Entrez votre prénom : '
prénom = gets.chomp
puts "Bonjour #{nom} #{prénom}"
```

Notons que nous avons utilisé `print` quand nous ne voulions pas aller à la ligne et `puts` quand nous voulions aller à la ligne. Retourner au texte.

Contenu masqué n°3

La difficulté est l'échange des deux variables. Pour ce faire, nous allons utiliser une variable temporaire que nous appellerons `tmp`. Nous stockerons la valeur de `x` dans `tmp`, puis nous affecterons à `x` la valeur de `y` et enfin `y` prendra la valeur de `tmp`.

```
print 'Entrez x : '
x = gets.chomp.to_i
print 'Entrez y : '
y = gets.chomp.to_i

tmp = x
x = y
y = tmp
print "L'échange a été effectué : x vaut #{x} et y vaut #{y}."
```

Retourner au texte.

Contenu masqué n°4

Nous pouvons effectuer les calculs de plusieurs manières. Mais, le but est de ne pas faire de calculs inutiles. Ainsi, nous pouvons soit créer plusieurs variables pour contenir l'âge dans les différentes unités, soit avoir une seule variable et afficher l'âge dans une unité puis la convertir dans l'unité suivante et afficher et ce jusqu'à avoir tout affiché.

Dans tous les cas, pour ne pas faire de calculs inutiles, il vaut mieux utiliser le résultat du calcul précédent pour faire le calcul suivant. Il faut donc utiliser l'âge en années pour calculer celui en jours, celui en jours pour calculer celui en heures...

Voici le programme, fait des deux manières que nous avons évoquées.

```
print 'Entrez votre âge : '
âge = gets.chomp.to_i # On convertit la saisie en entier.
puts âge
âge = âge * 365       # On passe aux jours.
puts âge
âge = âge * 24        # On passe aux heures.
puts âge
âge = âge * 60        # On passe aux minutes.
puts âge
âge = âge * 60        # On passe aux secondes.
puts âge
```

Dans celui-ci on affiche la variable `âge` avant de passer à l'unité suivante et ainsi de suite.

Voici le deuxième, dans lequel on va créer des variables pour les jours, les heures, les minutes et les secondes.

```ruby
print 'Entrez votre âge : '
âge = gets.chomp.to_i # On convertit la saisie en entier.
jours = âge * 365
heures = jours * 24
minutes = heures * 60
secondes = minutes * 60
puts âge
puts jours
puts heures
puts minutes
puts secondes
```

Retourner au texte.

Contenu masqué n°5

Nous demandons les deux nombres et les convertissons en flottants, puis nous affichons toutes les données.

```ruby
print 'Entrez le premier nombre : '
nombre1 = gets.chomp.to_f # On demande le nombre 1 et on le convertit en flottant.
print 'Entrez le deuxième nombre : '
nombre2 = gets.chomp.to_f # On demande le nombre 2 et on le convertit en flottant.

puts "#{nombre1} + #{nombre2} = #{nombre1 + nombre2}"
puts "#{nombre1} - #{nombre2} = #{nombre1 - nombre2}"
puts "#{nombre1} * #{nombre2} = #{nombre1 * nombre2}"
puts "#{nombre1} / #{nombre2} = #{nombre1 / nombre2}"
```

Retourner au texte.

3. Les conditions

Dans ce chapitre nous apprendrons à créer des conditions. Ceci permettra d'adapter le déroulement de nos programmes à certains paramètres, tels que la valeur des variables.

3.1. Opérateurs de comparaison

Les conditions (ainsi que les boucles) introduisent de nouveaux opérateurs.
Voici le tableau des opérateurs de comparaison.

Opérateur	Signification
`<`	Strictement inférieur à
`>`	Strictement supérieur à
`<=`	Inférieur ou égal à
`>=`	Supérieur ou égal à
`==`	Égal à
`!=`	Différent de

Ainsi, on a `2 < 3` qui est vrai et `3 != 2` qui est aussi vrai.

Essayons de taper quelques expressions avec ces signes sur IRB.

```
2 > 1
=> true
3 > 6.0
=> false
'Trois' == 'Trois'
=> true
```

Ces expressions renvoient `true` ou `false`. Ce sont ce que l'on appelle des **booléens**. Une variable booléenne ne peut prendre comme valeur que `true` ou `false`, c'est-à-dire vrai ou faux. Parfait pour faire des conditions.

> Notons que la comparaison entre chaînes de caractères se fait suivant l'ordre lexicographique (l'ordre du dictionnaire). Ainsi, `'Trois' > 'Quatre'` renverra `true`. L'ordre de Ruby est le suivant: d'abord les chiffres (0, 1, 2… 9), puis les lettres majuscules (A, B… Z), puis les lettres minuscules (a, b… z).

En fait, Ruby compare les caractères des chaînes en utilisant la valeur de leur code UTF-8 (par exemple, `'é' > '#'` vaut `true`).

Nous pouvons bien sûr créer des variables de type booléen. Par exemple…

```
booléen = true
faux = false
```

Ces booléens nous servent justement à construire nos conditions. Toutes les comparaisons que nous ferons vaudront soit `true`, soit `false` et il nous faudra juste évaluer cette valeur.

On peut aussi combiner ces opérateurs et faire des comparaisons plus avancées avec trois mots-clés:

- `and` qui signifie « et » permet d'écrire `3 > 2 and 3 < 5` (ce qui est vrai);
- `or` qui signifie « ou » permet d'écrire `3 > 2 or 2 < 5` (ce qui est vrai);
- `not` qui signifie « non », c'est-à-dire la condition contraire, permet d'écrire `not 3 < 2` (ce qui est vrai).

L'on peut remplacer `and` par `&&`, `or` par `||` et `not` par `!`. En fait, nous allons même privilégier `!` `&&` et `||` pour les expressions booléennes, conformément aux bonnes pratiques de Ruby.

3.1.1. Mise entre parenthèses

Nous pouvons placer des parenthèses autour de vos comparaisons. Si nous les plaçons autour de toute la condition, cela ne change pas sa valeur. `(1 == 2 || 2 == 2)` est strictement équivalent à `1 == 2 || 2 == 2`. Cependant, si nous plaçons des parenthèses autour d'une partie de la condition, sa valeur peut changer. En effet, cela change les priorités. Ainsi :

- `(1 == 1 || 2 == 3) && 3 == 4` vaut `false`;
- `1 == 1 || (2 == 3 && 3 == 4)` vaut `true`.

Voyons ce qui se passe dans le premier cas.

`1 == 1 || 2 == 3` vaut `true`, donc c'est équivalent à `true && 3 == 4`, avec 3 4, donc `(1 == 1 || 2 == 3) && 3 == 4` vaut `false`.

Dans le deuxième cas maintenant.

`2 == 3 && 3 == 4` vaut `false`, donc c'est équivalent à `1 == 1 or false`, avec $1 = 1$, donc `1 == 1 || (2 == 3 && 3 == 4)` vaut `true`.

3.2. La structure if-else

La structure `if-else` est la plus simple des structures conditionnelles. On évalue une expression (on regarde si elle vaut `true` ou `false`) ; si (`if`) elle vaut `true`, on exécute certaines instructions, sinon (`else`), on en exécute d'autres.

Prenons le cas d'un programme qui demande l'âge de l'utilisateur et nous dit s'il est majeur ou mineur. S'il a moins de 18 ans, il est mineur, sinon, il est majeur.

```
1  âge = gets.chomp.to_i # On récupère une saisie et on la convertit
       en entier.
2
3  if âge < 18
4    print 'Vous êtes mineur.'
5  else      # Sinon…
6    print 'Vous êtes majeur.'
7  end
```

Ici, nous vérifions si la variable `âge` est inférieure à 18 grâce à `<`. Rentrons différents âges inférieurs ou supérieurs à 18 et admirons le travail.

Cette structure conditionnelle se termine toujours par le mot-clé `end`. C'est très important, et il ne faut pas l'oublier.

L'indentation n'est pas obligatoire comme en Python. Cependant, il faut quelque chose pour séparer le `if condition` des instructions à exécuter. Le retour à la ligne est l'un des moyens de faire cette séparation. Le mot clé `then` (« alors ») est l'autre manière de la faire. On peut alors écrire ce code strictement équivalent au précédent.

```
1  âge = gets.chomp.to_i # On récupère une saisie et on la convertit
       en entier.
2
3  if âge < 18 then print 'Vous êtes mineur.' else print
       'Vous êtes majeur.' end
```

Mais nous allons plutôt privilégier la première écriture. Notons de plus qu'il est conseillé d'indenter en utilisant deux espaces. Nous pouvons combiner le `then` et le retour à ligne, mais ce n'est pas une pratique conseillée.

Notons que le `else` n'est pas obligatoire. Nous pouvons parfaitement écrire ceci.

```
1  âge = gets.chomp.to_i # On récupère une saisie et on la convertit
       en entier.
```

```
2
3 if âge < 18
4   print 'Vous êtes mineur.'
5 end
```

3.2.1. Le mot-clé `elsif`

Si nous avons plus de deux possibilités, nous pouvons compléter notre structure `if`-`else` avec un `elsif` (sinon si). Il permet d'ajouter une autre possibilité à la condition `if`. Prenons le code précédent et rajoutons un `elsif`.

```
1 âge = gets.chomp.to_i
2
3 if âge < 18     # Si âge est inférieur à 18…
4   print 'Vous êtes mineur.'
5 elsif âge > 80 # Si âge est supérieur à 80…
6   print 'Vous êtes senior.'
7 else           # Sinon…
8   print 'Vous êtes majeur et votre âge est inférieur à 80 ans.'
9 end
```

On regarde si la première condition est vraie. Si elle est vraie, on affiche que l'utilisateur est mineur, sinon, on regarde la seconde condition, et enfin, on arrive au `else` qui s'exécute si aucune des conditions précédentes n'est vraie.

Nous pouvons utiliser autant de `elsif` que nous voulons.

Ce n'est vraiment pas compliqué à utiliser. La seule subtilité est que les remarques que nous avons tenues à propos du `if` sont aussi valables pour le `elsif`: là encore, le retour à la ligne est obligatoire à moins que nous n'utilisions le mot-cle `then`.

Notons qu'il existe une façon plus concise d'écrire un `if`.

```
1 âge = gets.chomp.to_i
2
3 print 'Vous êtes majeur.' if âge >= 18
```

Dans ce cas, il n'y a pas de `end` et l'instruction à exécuter doit être unique. Cependant, nous pouvons placer plusieurs instructions en utilisant le point-virgule et des parenthèses de cette manière.

```
âge = gets.chomp.to_i

(print 'Vous êtes majeur.'; print
    'Peut-être même que vous êtes senior.') if âge >= 18
```

Cette notation est conseillée pour les conditions sur une seule ligne, en particulier s'il n'y a qu'une seule instruction à exécuter et qu'elle est simple. Cependant, si la condition — ou même l'instruction — est trop compliquée, nous allons privilégier un `if` classique.

3.2.2. La structure `unless`

La structure `unless` fonctionne exactement de la même manière que `if`, et est en fait la condition contraire de `if`. Avec `unless`, les instructions sont toujours exécutées **sauf** lorsque la condition est vérifiée. On a donc `unless (condition)` qui est équivalent à `if !(condition)`. Pour l'âge, on peut donc écrire ce programme.

```
âge = 19

unless âge < 18
  print 'Vous êtes majeur.'
else
  print 'Vous êtes mineur.'
end
```

Là encore, nous pouvons utiliser la structure condensée.

```
print 'Vous êtes majeur.' unless âge < 18
```

Une bonne pratique consiste à privilégier `unless` pour les conditions négatives, c'est-à-dire que nous allons préférer écrire ce genre de code.

```
#### Mauvais code.
if !condition
  # Instructions.
end

#### Bon code.
unless condition
  # Instructions.
end
```

Cependant, nous n'allons pas utiliser `unless` avec `else`, mais préférer réécrire la condition en changeant le `unless` par un `if` et en inversant les conditions.

```
#### Mauvais code.
unless condition
  # Autres instructions.
else
  # Instructions.
end

#### Bon code.
if condition_inverse
  # Instructions.
else
  # Autres instructions.
end
```

Hormis cela, les bonnes pratiques pour `unless` sont les mêmes que celles pour `if`.

3.3. La structure case-when

La structure `case-when` est une autre structure de condition. Elle permet de tester une suite de conditions sur une variable. Si la variable vaut telle valeur, alors fais ceci, sinon, si elle vaut telle autre valeur, alors fais cela… Par exemple, écrivons un programme qui demande à l'utilisateur d'entrer «Un», «Deux» ou «Trois» et qui affiche la valeur numérique du nombre entré.

```
nombre = gets.chomp

case nombre     # On teste la valeur de nombre.
when 'Un'     # Si c'est "Un"…
  print 1
when 'Deux'   # Sinon, si c'est "Deux"…
  print 2
when 'Trois' # Sinon, si c'est "Trois"…
  print 3
end
```

Là encore, il y a un `end`. Il est aussi **obligatoire**.

Notons que nous pouvons rajouter un `else` pour faire quelque chose si aucun des cas testés n'est vérifié.

```ruby
nombre = gets.chomp

case nombre
when 'Un'
  print 1
when 'Deux'
  print 2
when 'Trois'
  print 3
else
  print 'La saisie n’est pas bonne.'
end
```

Si la saisie de l’utilisateur n’est ni «Un», ni «Deux», ni «Trois», alors on affiche «La saisie n’est pas bonne.».

Notons que nous pouvons écrire le `when` et l’action à effectuer sur une seule ligne en utilisant le mot-clé `then`. Ainsi, le code qui suit est strictement équivalent à celui que nous avons écrit plus haut.

```ruby
nombre = gets.chomp

case nombre
when 'Un' then print 1
when 'Deux' then print 2
when 'Trois' then print 3
else print 'La saisie n’est pas bonne.'
end
```

3.3.1. Les intervalles

Cependant, là où `case` s’avère vraiment utile, c’est qu’il permet de tester si une valeur est dans un intervalle.

En Ruby, nous allons noter un intervalle `x..y`, qui représente l’intervalle $[x, y]$ (donc x et y sont inclus). Pour voir comment nous pouvons utiliser cela, prenons l’exemple d’un programme qui vérifierait l’admission ou la mention selon une note à un examen. Dans notre cas, avoir en dessous de 6 est un échec, au-dessus de 12 la mention « Assez bien », de 14 la mention « Bien » et de 16 la mention « Très bien ».

Faire ce programme avec des `if` et des `else` serait long et contraignant. La structure `case` permet de simplifier les choses en offrant plusieurs choix.

```ruby
note = 18 # Nous pouvons modifier la note et voir ce qu'on obtient.

case note
when 0..6
  print 'Vous n'avez pas réussi l'examen.'
when 6..12
  print 'Vous avez réussi l'examen.'
when 12..14
  print 'Vous avez réussi l'examen avec mention « Assez bien ».'
when 14..16
  print 'Vous avez réussi l'examen avec mention « Bien ».'
when 16..20
  print 'Vous avez réussi l'examen avec mention « Très bien ».'
else
  print 'La note entrée est incorrecte.'
end
```

Ça aurait été vraiment embêtant de faire tout ça avec des `if` et des `else`.

> *i* Les intervalles peuvent aussi se faire avec des valeurs flottantes. Ainsi, nous pouvons choisir d'attribuer la mention `when 16.5..20`.

3.4. Les conditions ternaires

Dans plusieurs langages, il y a ce que l'on appelle des conditions ternaires. Elles sont un moyen condensé d'écrire des conditions. Voici le schéma qu'elles suivent.

```ruby
condition ? (si true) : (sinon)
```

Ce n'est pas aussi impressionnant que ça en a l'air. Il faut bien sûr s'habituer à la syntaxe, mais une fois celle-ci assimilée, il faut s'exercer pour réussir à la maîtriser.

```ruby
âge = 19

(âge < 18) ? (print 'Mineur') : (print 'Majeur')
```

Et on peut même utiliser des conditions ternaires imbriquées pour obtenir du `else if`.

```
âge = 19

(âge < 18) ? (print 'Mineur') : (âge < 80 ? (print 'Majeur') :
    (print 'Senior'))
```

Là c'est plus compliqué, non? Il faut bien prendre le temps de comprendre le code.

Les conditions ternaires sont rares dans d'autres langages, mais sont assez appréciées en Ruby pour leur compacité. Ainsi, nous aurons peut-être l'occasion de les croiser dans un code. En fait, la bonne pratique en Ruby est de privilégier les conditions ternaires si la condition est de la forme `if-then-else-end` c'est-à-dire dans les cas comme celui que nous avons traité.

Cependant, il faut éviter d'imbriquer plusieurs conditions ternaires et utiliser une structure `if` ou `unless` dans ce cas-là. De même, il faut privilégier les structures `if` et `unless` si la condition est sur plusieurs lignes.

3.5. Exercices

Voilà une nouvelle série d'exercices.

3.5.1. Exercice 1

Le premier exercice est encore une fois plutôt simple: demandons à l'utilisateur d'entrer trois nombres, puis affichons le plus grand des trois nombres.

Correction.

Contenu masqué n°6

Maintenant, modifions ce programme pour qu'il affiche les trois nombres triés par ordre croissant.

Correction.

Contenu masqué n°7

3.5.2. Exercice 2

Cet exercice est plus un mini TP. Nous devons demander une année à l'utilisateur et déterminer si elle est ou non bissextile. Donc, il nous faut savoir la condition suivant laquelle une année peut être qualifiée de bissextile.

> Une année est dite bissextile si elle est divisible par 400 ou si elle est divisible par 4 et non divisible par 100.

Un peu d'aide…

Contenu masqué n°8

Correction.

Contenu masqué n°9

3.5.3. Exercice 3

Nous allons maintenant devenir commerçants. Notre objectif: demander à l'utilisateur de rentrer le prix HT d'un objet et son code (compris entre 1 et 3), et calculer le prix TTC de l'objet sachant que le code 1 correspond à une TVA de 20 %, le code 2 à une TVA de 10 % et le code 3 à une TVA de 5.5 % (on considère que la seule taxe est la TVA).

Nous allons donc devoir manier des pourcentages.

Correction.

Contenu masqué n°10

Voilà, le chapitre sur les conditions est terminé. Maintenant, nos programmes n'ont plus un comportement linéaire, mais peuvent faire une action suivant la valeur d'une variable.

- Ruby dispose d'opérateurs de comparaison (supérieur, inférieur, différent, etc.) que l'on peut combiner avec les symboles `||`, `&&` et `!`.
- Une expression faite avec ces symboles s'appelle une expression booléenne et vaut soit `true` soit `false`.
- On peut exécuter certaines instructions en fonction de la valeur d'un booléen en utilisant des structures conditionnelles (`if`-`elsif`-`else` ou `unless`-`else` par exemple).
- La structure `case`-`when` permet de tester une suite de conditions sur une même variable. Elle permet également de tester si une variable appartient à un intervalle donné.
- Les conditions ternaires sont un moyen condensé d'écrire des conditions et sont assez appréciées en Ruby.

Contenu masqué

Contenu masqué n°6

Voici notre correction. Bien sûr, on peut tout à fait avoir un code différent. On a été malin : on compare le nouveau nombre entré au maximum des nombres déjà entrés.

```
puts 'Entrez trois nombres.'
nombre = gets.chomp.to_i
max = nombre
nombre = gets.chomp.to_i
max = nombre if nombre > max
nombre = gets.chomp.to_i
max = nombre if nombre > max
print max
```

Retourner au texte.

Contenu masqué n°7

Pour faire cela, nous allons demander nos trois nombres, les trier, et les afficher ensuite.

```
puts 'Entrez trois nombres.'
nombre1 = gets.chomp.to_i
nombre2 = gets.chomp.to_i
nombre3 = gets.chomp.to_i
if nombre1 > nombre2 # Si nombre1 > nombre2, on les échange.
  tmp = nombre1
  nombre1 = nombre2
  nombre2 = tmp
end
if nombre2 > nombre3 # Si nombre2 > nombre3, on les échange.
  tmp = nombre2
  nombre2 = nombre3
  nombre3 = tmp
  # ancien_nombre3 était plus petit que ancien_nombre2, alors il
      peut être plus petit que nombre1
  if nombre1 > nombre2
    tmp = nombre1
    nombre1 = nombre2
    nombre2 = tmp
  end
end
puts nombre1, nombre2, nombre3
```

Pour bien comprendre l'idée qui est mise en place, il faut prendre un papier et un crayon. Retourner au texte.

Contenu masqué n°8

Une question reste problématique: comment tester si un nombre `a` est multiple d'un nombre `b`? En fait, il suffit de tester le reste de la division entière de `b` par `a`. Si ce reste est nul, alors `a` est un multiple de `b`.

```
5 % 2 # 5 n'est pas un multiple de 2.
=> 1
8 % 2 # 8 est un multiple de 2.
=> 0
```

Il suffit d'appliquer cette méthode à notre cas. Retourner au texte.

Contenu masqué n°9

Voici une correction possible (d'autres code peuvent aussi fonctionner).

```
puts 'Entrez une année : '
année = gets.chomp()
année = année.to_i()
if année % 400 == 0
  puts 'L'année est bissextile.'
elsif année % 4 == 0 && année % 100 != 0
  puts 'L'année est bissextile.'
else
  puts 'L'année n'est pas bissextile.'
end
```

On peut le simplifier.

```
puts 'Entrez une année : '
année = gets.chomp()
année = année.to_i()
if année % 400 == 0 || (année % 4 == 0 && année % 100 != 0)
  puts 'L'année est bissextile.'
else
  puts 'L'année n'est pas bissextile.'
end
```

Retourner au texte.

Contenu masqué n°10

On va déclarer trois constantes correspondant aux trois taux de TVA et les utiliser pour nos calculs.

```
TVA1 = 20.0
TVA2 = 10.0
TVA3 = 5.5

print 'Entrez le prix de votre produit : '
prixHT = gets.chomp.to_f
print 'Entrez le code de votre produit : '
code = gets.chomp.to_i

if code == 1
  prixTTC = prixHT + TVA1 / 100 * prixHT
elsif code == 2
  prixTTC = prixHT + TVA2 / 100 * prixHT
elsif code == 3
  prixTTC = prixHT + TVA3 / 100 * prixHT
end

case code
when 1..3
  print "Le prix TTC de votre produit est #{prixTTC}."
else
  print 'Votre code n’est pas valide.'
end
```

Le code peut être raccourci, par exemple de cette manière.

```
print 'Entrez le prix de votre produit : '
prixHT = gets.chomp.to_f
print 'Entrez le code de votre produit : '
code = gets.chomp.to_i

if code == 1
  print "Le prix TTC de votre produit est #{prixHT * 1.2}."
elsif code == 2
  print "Le prix TTC de votre produit est #{prixHT * 1.1}."
elsif code == 3
  print "Le prix TTC de votre produit est #{prixHT * 1.055}."
else
  print 'Votre code n’est pas valide.'
end
```

Retourner au texte.

4. Les boucles

Il est maintenant temps d'apprendre à répéter des instructions un certain nombre de fois. Les boucles ont pour objectif de permettre de faire ceci. Il existe plusieurs boucles différentes, et ce sont elles que nous allons voir durant ce chapitre.

4.1. La boucle while

La boucle `while` est une structure de boucle plutôt simple. Elle permet de répéter des instructions tant qu'une condition est vraie. Encore une fois, remarquons que `while` est l'équivalent anglais de «tant que».

Prenons le cas d'un programme qui affiche la table de multiplication d'un nombre. Il affiche les produits de ce nombre par les entiers de 1 à 10.

```
n = 1
while n <= 10 # Tant que n est inférieur ou égal à 10, le code est exécuté.
  print "#{n * 8} "
  n = n + 1 # On ajoute 1 à n à chaque tour pour que sa valeur atteigne 10.
end
```

Ici, nous affichons la table de 8. Pour cela, nous initialisons une variable à 1. Cette variable s'appelle un **compteur**. Tant que cette variable sera inférieure ou égal à 10, on affichera le résultat du produit de 8 et de cette variable, puis on ajoutera 1 à cette variable (on dit qu'on **incrémente** la variable).

> ! Là encore, le mot-clé `end` est obligatoire.

Exécutons le programme et observons le résultat.

```
8 16 24 32 40 48 56 64 72 80
```

La structure d'une boucle `while` est vraiment très proche de celle d'un `if`. On pourrait presque se dire que c'est un `if` qui se répète tant que la condition est vraie.

D'ailleurs, nous pouvons aussi utiliser une forme condensée de `while`. On pourrait alors écrire le programme précédent de cette manière.

```
n = 1

(print "#{n * 8} "; n = n + 1) while n <= 10
```

Tout comme pour le `if` et sa forme condensée, nous allons préférer la forme condensée du `while` à sa forme classique s'il n'y a qu'une seule instruction à exécuter et que celle-ci est courte.

4.1.1. La structure `begin-while`

Il peut arriver que nous voulions qu'une instruction se répète en fonction d'une condition (donc une boucle `while`) , mais que nous voulions également que cette instruction soit exécutée au moins une fois. Nous pouvons nous dire qu'il suffit de placer l'instruction une fois avant la boucle, et une autre fois dans la boucle, et c'est bien vrai. Mais il y a mieux.

La structure `begin-end while` équivaut à un `while`, sauf que le code est exécuté au moins une fois. Avec cette structure, le `while` se retrouve à la toute fin de la boucle. Reprenons l'exemple précédent en utilisant cette structure.

```
n = 1
begin
  print "#{n * 8} "
  n = n + 1
end while n <= 10
```

Cet exemple n'est peut-être pas très parlant. Regardons alors celui-là.

```
n = 6

begin
  print "#{n} "
end while n < 5
```

La condition n'est même pas vraie au premier tour de boucle, mais les instructions sont toujours exécutées au moins une fois. En fait, la condition n'est évaluée qu'après le premier tour de boucle, et ceci car elle est évaluée après l'exécution des instructions (le `while` est à la fin pour cette raison).

> Encore une fois, le `end` est obligatoire, et là, il est à placer **avant** le while qui se retrouve alors à la fin de la boucle.

En dehors de ceci, la structure `begin-while` fonctionne exactement comme la structure `while`. Il faut cependant faire attention à la forme condensée et ne pas oublier d'y intégrer le `end` de cette manière.

```
n = 11

begin (print "#{n * 8} "; n = n + 1) end while n <= 10
```

4.1.2. La structure `until`

Rendons à César ce qui est à César: la boucle `until` est à la boucle `while` exactement ce que la condition `unless` est à la condition `if`. Ainsi, la boucle `until` est une boucle qui s'exécute jusqu'à ce que la condition soit vraie, c'est-à-dire tant qu'elle est fausse. On a donc `until (condtition)` qui est équivalent à `while !(condition)`. Réécrivons le programme précédent avec une boucle `until`.

```
n = 1
until n > 10 # Jusqu'à ce que n soit plus grand que 10, le code est
    exécuté.
  print "#{n * 8} "
  n = n + 1
end
```

Remarquons que l'inégalité large que l'on avait avec la boucle `while` devient stricte avec la boucle `until`. En effet, on veut s'arrêter quand `n` sera strictement supérieur à 10.

Encore une fois, et nous pouvions nous en douter, nous pouvons utiliser la structure condensée.

```
n = 1

(print "#{n * 8} "; n = n + 1) until n > 10
```

Nous allons préférer `until` à `while` dans le cas d'une condition négative.

```
#### Mauvais code.
while !condition
  # Instructions.
end

#### Bon code.
until condition
  # Instructions.
```

```
9 end
```

4.2. La boucle for

Pour le moment, nous avons vu la boucle `while` et ses dérivés. Elle permet d'exécuter des instructions en fonction d'un condition. La boucle `for` est relativement similaire: elle permet d'exécuter des instructions pour chaque élément d'un ensemble.

Mais de quoi voulons-nous parler en parlant d'«ensemble» ? En fait, nous en avons déjà utilisé dans la partie précédente lorsque nous avons parlé des intervalles de nombres. Oui, c'est un ensemble. La boucle `for` nous permet alors d'effectuer des instructions pour chaque élément d'un intervalle.

Le code suivant exécute l'instruction `print n` de manière répétée.

```
1 for n in 0..5
2   print "#{n} " # Instruction(s).
3 end
```

> ! Nous devons encore le redire, le `end` est obligatoire.

Sans surprise, ce code affiche tous les entiers entre 0 et 5.

```
1 0 1 2 3 4 5
```

Il existe une autre façon d'écrire des intervalles. Elle ressemble beaucoup à celle que nous avons déjà vue. En fait, pour l'utiliser, il suffit d'écrire `...` plutôt que `..`. Cependant, il y a une différence entre les deux syntaxes:

— `l..n` désigne l'ensemble des nombres entre `l` et `n`, `l` et `n` inclus ;
— `l...n` désigne aussi l'ensemble des nombres entre `l` et `n`, mais `n` est cette fois exclu.

Ainsi, le code…

```
1 for n in 0...5
2   print "#{n} " # Instruction(s).
3 end
```

… n'affichera que…

```
0 1 2 3 4
```

> **!** Notons que si dans la partie précédente, le `when` évaluait si le nombre proposé se trouvait dans l'intervalle, nombres réels compris, ici, l'intervalle est parcouru d'entier en entier.

En y réfléchissant, ceci est tout à fait normal: si on commence à 2.3 par exemple, comment savoir si on doit avancer de 1 en 1, ou si on doit commencer à 2.3 puis passer à 3…

Les deux façons de noter les intervalles peuvent troubler et il est facile d'oublier celui qui inclut la borne supérieure. Voici un moyen mnémotechnique simple pour ne pas se tromper. Quand on écrit `a...b`, le troisième point est là pour «pousser `b` dehors»; avec `a...b`, `b` est donc exclu.

4.3. Contrôler l'exécution d'une boucle

Ce serait bien de pouvoir rajouter un peu de contrôle sur nos boucles. Tant mieux, Ruby dispose de mots-clés réservés au contrôle de l'exécution des boucles.

4.3.1. Le mot-clé `break`

Le mot-clé `break` permet d'interrompre l'exécution d'une boucle à n'importe quel moment. Son fonctionnement est vraiment très simple.

```ruby
i = 0

while true
  print "#{i} "
  break if i > 6
  i = i + 1
end
```

Voici le résultat obtenu.

```
1 2 3 4 5 6 7
```

Quand `i` vaut 7, on rentre dans le `if` et le `break` est exécuté: on sort de la boucle au septième tour de boucle.

4.3.2. Le mot-clé `next`

Le mot-clé `next` permet d'interrompre un tour de boucle. Le programme passe alors directement au tour suivant. Voici, par exemple, une manière (pas la plus futée) d'afficher les nombres impairs entre 0 et 10.

```
for i in 0..10
  next if i % 2 == 0
  print "#{i} "
end
```

On obtient sans surprise ce résultat.

```
1 3 5 7 9
```

Pour ce faire, on passe au tour de boucle suivant si `i` est pair, c'est-à-dire si `i % 2 = 0`. L'instruction `print i` n'est donc exécutée que si `i` est impair.

Notons que l'utilisation de `next` dans ce genre de situation est préférée à l'utilisation d'un bloc conditionnel dans une boucle. Donc, le code que nous avons écrit sera préféré à celui-là.

```
for i in 0..10
  print "#{i} " if i % 2 != 0
end
```

4.3.3. Le mot-clé `redo`

Le mot-clé `redo`, lui, permet de recommencer un tour de boucle. Si on est au troisième tour de boucle et que le programme croise `redo`, le troisième tour de boucle recommence. Affichons deux fois tous les entiers de 1 à 5.

```
k = 1
for i in 1..5
  puts i
  if i == k
    k = k + 1
    redo
  end
end
```

Ici, on a un compteur `i` pour la boucle et un autre `k` qui permet de savoir quand on doit reprendre un tour de boucle. Et on incrémente `k` à chaque fois que `i == k`.

4.4. La boucle loop

Il est maintenant temps de parler d'une boucle particulière: la boucle `loop`. Elle a de particulier qu'elle n'a pas de condition d'arrêt, c'est une boucle infinie. Elle est donc équivalente à `while true`. Mais nous pouvons quand même en sortir. Comment? Tout simplement en utilisant un des mots-clés que nous venons de voir. Oui, le mot-clé `break` permet de sortir d'une boucle infinie. Testons d'abord ce code (attention, ce programme boucle indéfiniment, il nous faudra appuyer sur Ctrl + C pour quitter le programme).

```
i = 0
loop do
  puts i
  i = i + 1
end
```

Et regardons celui-là qui s'arrête à 10 grâce à `break`.

```
i = 0
loop do
  puts i
  break if i == 10
  i = i + 1
end
```

!

Le `do` est obligatoire, de même que le `end`. En fait, `do` peut aussi être utilisé de la même manière avec les boucles `while`, `until` et `for`. Le code de la boucle est alors encadré par `do` et `end` de même qu'il est encadré par `begin` et `end` dans la structure `begin-while`.

Tout comme nous avons dit que les structures conditionnelles que nous utiliserons le plus sont `if-else` et `case-when`, nous devons avouer que les boucles les plus courantes sont les boucles `while` et `for`. La boucle `loop` que nous venons de voir est préférée à la structure `begin-while`.

4.5. Les itérateurs

En fait, la boucle `for` est rarement utilisée pour faire des itérations. À la place, nous utilisons ce que nous appelons des **itérateurs**. Chaque type d'ensemble a ses propres itérateurs (même s'ils en ont parfois en commun) qui permettent de faire plusieurs opérations.

4.5.1. La méthode `each`

Pour parcourir un intervalle, nous utiliserons la méthode `each`. Par exemple, avec ce code, nous affichons tous les nombres de l'intervalle `1..8`.

```
(1..8).each do |i|
  puts i
end
```

Ici, pour chaque (*each*) élément de l'intervalle `1..8`, nous affichons `i`, avec `i` qui correspond à chaque fois à un élément de l'intervalle parcouru.

La structure `do`-`end` délimite ce qu'on appelle un **bloc**. On donne un bloc à la méthode `each` et elle l'exécute pour chaque élément de l'ensemble que l'on parcourt. Un bloc peut également être délimité par des accolades. Il est d'usage d'utiliser les accolades quand notre bloc s'écrit sur une ligne et la structure `do`-`end` lorsqu'il est plus long. Ainsi, notre code précédent peut également s'écrire ainsi.

```
(1..8).each { |i| puts i }
```

Notons que contrairement au code interpolé, le code mis entre accolades pour un bloc est ici entouré d'espaces. C'est une bonne pratique qu'il nous faut essayer de respecter.

Il y a quelques différences subtiles entre la délimitation par `do`-`end` ou par les accolades, mais nous n'en parlerons pas ici. Nous devons juste savoir que l'on donne à notre itérateur un bloc à exécuter pour chaque élément de notre intervalle.

Dans notre code, `i` est une variable du bloc. Lorsque nous donnons un bloc avec une variable à `each`, cette variable prend tour à tour toutes les valeurs de l'ensemble parcouru.

Les blocs sont une structure puissante de Ruby dont nous ne parlerons pas plus pour le moment.

4.5.2. La méthode `times`

Pour faire une opération un certain nombre de fois, nous utiliserons la méthode `times`. Elle s'applique à un entier `k` et exécute le bloc qu'on lui fournit `k` fois.

```
5.times { |i| print "#{i} " }
```

Ici, la variable `i` variera de 0 à 4, et à chaque fois, nous allons afficher `i`. Voici l'équivalent de ce code en utilisant une boucle `for`.

```
for n in 0...5
  print "#{n} "
end
```

`times` s'exécute `n` fois, et la variable `i` de notre code va de `0` à `4`, pas de `0` à `5`.

4.5.3. La méthode `upto`

Si nous ne voulons par partir de 0 mais d'un autre nombre, nous pouvons utiliser la méthode `upto`. Elle s'applique à un entier, l'incrémente et exécute un bloc d'instructions jusqu'à atteindre le nombre passé en paramètre.

```
2.upto(5) { |i| print "#{i} " }
```

Ce code est équivalent à celui-là.

```
for n in 2..5
  print "#{n} "
end
```

4.5.4. La méthode `downto`

Et si nous voulons aller de 5 à 2, nous utilisons la méthode `downto` qui s'applique à un entier, le décrémente et exécute un bloc de code jusqu'à atteindre le nombre passé en paramètre.

```
5.downto(2) { |i| print "#{i} " }
```

Qui avec une boucle `for` pourrait s'écrire ainsi, avec un code plus compliqué.

```
for n in 2..5
  print "#{5 - n + 2} "
end
```

4.5.5. La méthode `step`

Il est également possible d'aller de 2 en 2 ou de -4 en -4 grâce à la méthode `step`. Elle s'applique à un entier `début`, prend en paramètre deux entiers `fin` et `pas`, et permet d'aller de `début` à `fin` en allant de `pas` en `pas`. Par exemple, avec le code suivant, nous comptons de 5 en 5 en commençant à 2 pour aller jusqu'à 18.

```
2.step(18, 5) { |i| puts i }
```

À chaque fois que nous avons utilisé un itérateur, nous avions une variable entourée du symbole `| |` qui prenait successivement toutes les valeurs parcourues. C'est un **paramètre du bloc**. Mais elle n'est pas obligatoire. Par exemple, si nous voulons juste afficher 4 fois le message «Voici un message», nous pouvons parfaitement écrire ceci.

```
4.times { puts 'Voici un message.' }
```

Cependant, nous allons préférer toujours mettre le paramètre de bloc, et s'il est inutilisé, nous allons le préfixer par `_` (il pourra même être `_`). Nous allons donc plutôt écrire ce code.

```
4.times { |_| puts 'Voici un message.' }
```

En fait, dès qu'une variable sera inutilisée (oui, ça arrive parfois), nous allons la préfixer par le symbole `_`.

Nous verrons plus tard d'autres itérateurs lorsque nous verrons d'autre types d'ensembles. Mais, il nous faut savoir que les itérateurs sont à privilégier sur les boucles.

4.6. Exercices

Comme d'habitude, terminons ce chapitre par une petite série d'exercices.

4.6.1. Exercice 1

Écrivons un programme qui demande un entier `n` à l'utilisateur et qui affiche la somme des nombres de 1 à `n`, si `n` est positif, et retourne une erreur sinon.

Correction.

Contenu masqué n°11

4.6.2. Exercice 2

Écrire un programme qui demande un entier `n` à l'utilisateur et affiche un triangle de `n` lignes composées d'étoiles. Pour `n = 4`, le programme doit afficher ceci.

```
*
**
***
****
```

Correction.

Contenu masqué n°12

4.6.3. Exercice 3

Nous en avons l'habitude, il faut aussi des exercices compliqués. Ici, nous allons écrire un programme qui demande un nombre **entier** à l'utilisateur et qui affiche ensuite ce nombre à l'envers. Un exemple d'utilisation du programme.

```
Entrez un nombre entier : 1974
4791
```

Aide.

Contenu masqué n°13

Correction.

Contenu masqué n°14

Maintenant, nos programmes peuvent être encore plus vivants.

— Les boucles permettent de répéter des instructions. La boucle `while` permet de répéter tant qu'une condition est satisfaite et la boucle `until` jusqu'à ce qu'elle soit satisfaite.
— Certains mots-clés nous permettent de contrôler l'exécution d'une boucle pour, par exemple, en sortir (mot-clé `break`) ce qui nous permet de sortir de la boucle infinie `loop`.
— La boucle `for` nous permet de parcourir un ensemble, mais nous préférons utiliser des itérateurs comme `each`.

Contenu masqué

Contenu masqué n°11

Pour ce faire, nous allons initialiser une variable `somme` à 0, puis nous allons ajouter les entiers à cette variable pour `k` allant de 1 à n.

```
print 'Entrez un nombre positif : '
n = gets.chomp.to_i
if n < 0
  print 'Erreur, votre nombre n’est pas positif.'
else
  somme = 0
  1.upto(n) { |k| somme += k }
  print somme
end
```

Retourner au texte.

Contenu masqué n°12

Le principe est un peu le même que pour l’exercice 1. Cependant, il faut cette fois utiliser deux boucles imbriquées (ou deux itérateurs). La première pour le nombre de lignes à afficher, la seconde pour le nombre d’étoiles à afficher par ligne, en sachant que lorsqu’on est à la ligne `n`, on affiche `n` étoiles.

```
print 'Entrez un entier positif : '
n = gets.chomp.to_i
if n < 0
  print 'Votre nombre n’est pas positif.'
else
  1.upto(n) do |k|
    k.times { print "*" }
    puts
  end
end
```

Retourner au texte.

Contenu masqué n°13

Un indice: utiliser le modulo. En effet, on a:

— `4791 % 10 = 1`;

— `479 % 10 = 9`;
— `47 % 10 = 7`;
— `4 % 10 = 4`.

Nous pouvons ainsi obtenir tous les chiffres à afficher. Retourner au texte.

Contenu masqué n°14

```
print 'Entrez un nombre : '
nombre = gets.chomp.to_i
until nombre == 0
  print nombre % 10
  nombre = nombre / 10
end
```

En cas de non-compréhension de ce programme, il faut prendre un nombre, et dérouler l'algorithme sur lui à l'aide d'un crayon et d'un papier. Retourner au texte.

5. Les tableaux

Dans ce chapitre, nous allons voir un nouveau type de variables: les tableaux. Les tableaux sont un type bien spécial de variables puisqu'ils permettent de conserver en mémoire plusieurs valeurs plutôt qu'une.

5.1. Généralités sur les tableaux

5.1.1. Qu'est-ce qu'un tableau

C'est l'heure pour nous de voir les tableaux. Un tableau est une super variable. En fait, c'est une structure de données. Cette structure est un ensemble d'éléments ordonnés auxquels on peut accéder par un indice, d'où le nom «tableau». Les tableaux sont des variables qui contiennent d'autres variables. Ce sont donc des «séquences» d'éléments tout comme les intervalles `n..l` vus dans le chapitre précédent.

Le premier indice d'un tableau est l'indice 0 et non 1.

On a donc l'élément d'indice 0, l'élément d'indice 1… Et ainsi de suite, potentiellement à l'infini.

Un tableau permet alors de rassembler des informations. Par exemple, nous pourrions vouloir les âges de cinquante personnes. Récupérer ces cinquante âges dans un tableau peut être judicieux.

En effet, les tableaux permettent d'agir très facilement sur l'ensemble des éléments, et en continuant avec l'exemple des cinquante âges, nous pourrions multiplier tous ces âges par deux en moins de dix lignes en utilisant un tableau, alors qu'avec cinquante variables, nous aurions besoin de cinquante lignes.

Les tableaux permettent donc de s'affranchir de beaucoup de contraintes (et de copier-coller) et favorisent le développement.

Et pour mettre un mot sur les choses, il nous faut préciser que les tableaux sont ce que l'on appelle un **conteneur** .

5.1.2. Déclarer un tableau

Pour déclarer un tableau, nous devons utiliser les crochets `[]`. Cela permet de déclarer un tableau vide.

```
tab = []
```

Pour afficher le tableau, nous pouvons utiliser la méthode `print` comme d'habitude.

```
tab = []
print tab
```

On affiche `tab`, on obtient `[]`, le tableau vide.

Les éléments du tableau sont placés entre les crochets, séparés par des virgules. Voici comment déclarer le tableau contenant les éléments 1, 2, et 3.

```
tab = [1, 2, 3]
print tab
```

On obtient cette fois cet affichage: `[1, 2, 3]`.

> Les crochets ne sont pas obligatoires. Nous pouvons ne pas les utiliser et juste séparer les éléments par des virgules. Cependant, ils sont obligatoires dans le cas où il y a moins de deux éléments.

Ainsi, nous aurions pu parfaitement faire notre dernier exemple de cette manière.

```
tab = 1, 2, 3
print tab
```

Cependant, pour des raisons de compréhension, nous conseillons de toujours utiliser les crochets.

Maintenant, nous pouvons faire des tableaux d'entiers, des tableaux de flottants et même des tableaux de chaînes de caractères. C'est bien, mais il y a mieux: en Ruby, les tableaux mixtes sont possibles. Cela veut dire que nous pouvons parfaitement faire un tableau contenant des entiers **et** des chaînes de caractères, voire d'autres tableaux. Ce code est parfaitement valide.

```
tab = [3.23, 'Trois virgule vingt-trois', [3, '.', 2, 3]]
print tab
```

5.1.2.1. Tableaux de chaînes de caractères

De même qu'il y a la syntaxe `%q` pour écrire des chaînes de caractères, nous pouvons utiliser les syntaxes `%w` et `%W` pour écrire des tableaux de chaînes de caractères. Dans ce cas, les chaînes sont séparées par des espaces. `%w` correspond à des chaînes en guillemets simples et `%W` à des chaînes en guillemets doubles.

```
1 a = %w[un deux trois]  # Équivalent à a = ['un', 'deux', 'trois'].
```

Ces syntaxes sont beaucoup plus simples à écrire et sont à privilégier lorsque nous voulons écrire un tableau de chaînes de caractères. La seule chose qui peut poser problème est l'écriture d'une chaîne avec des espaces. En effet, `%W[une chaîne deux]` ne donne pas `["une chaîne", "deux"]` mais `["une", "chaîne", "deux"]`. Pour que l'espace soit prise en compte comme telle et non comme séparateur, il faut l'échapper.

```
1 a = %W[une\ chaîne deux]  # Équivalent à a = ["une chaîne",
    "deux"].
```

En fait, nous n'allons privilégier ces syntaxes que dans le cas où nous voulons un tableau de mots. Notons de plus qu'il est préférable d'utiliser cette syntaxe avec les crochets comme délimiteur (notamment parce que cela rappelle bien qu'il s'agit d'un tableau) alors que dans le cas des chaînes de caractères, il est d'usage d'utiliser les parenthèses.

5.2. Opérations sur les tableaux

5.2.1. Accéder à un élément

Les tableaux et tout, c'est bien joli, mais pour le moment, on ne peut pas accéder à un élément du tableau. Comment faire? Ne nous moquons pas, il faut encore utiliser les crochets `[]`. Pour accéder à l'élément d'indice 2 du tableau `tab`, il nous faut juste écrire `tab[2]`. Voyons cela avec un exemple.

```
1 grammaire = %w[mais ou et donc or ni car]
2 print grammaire[2]
```

Ce code permet d'afficher la chaîne de caractères «et».

> Nous aurions pu penser que ce code afficherait «ou» mais n'oublions pas, le premier indice d'un tableau est l'indice 0. L'élément d'indice `n` est donc l'élément `n + 1` du tableau.

Nous pouvons maintenant extraire n'importe quel élément d'un tableau. Cependant, lorsque que nous accédons à un élément du tableau, il faut faire attention qu'il y ait bien un élément qui corresponde à l'indice demandé. L'élément d'indice 5 d'un tableau de taille 3 n'existe pas. Contrairement à d'autres langages, Ruby ne provoque pas d'erreur et retourne `nil` lorsque cela est fait, mais cela ne veut pas dire qu'il faut le faire.

On peut même afficher une partie du tableau seulement, en utilisant les intervalles vus dans le chapitre précédent. Ils nous permettent d'obtenir un sous-tableau contenant tous les éléments du tableau dont les indices appartiennent à l'intervalle.

```
grammaire = %w[mais ou et donc or ni car]
print grammaire[1..5]
```

Le code précédent affiche `["ou", "et", "donc", "or", "ni"]`, c'est-à-dire les éléments dont l'indice est compris entre 1 et 5.

5.2.2. Concaténation et ajout d'éléments

La concaténation de tableaux est l'ajout de deux tableaux. Cela permet de deviner de suite comment effectuer la concaténation. Oui, la concaténation de tableau se fait avec l'opérateur `+`. Son utilisation nous est maintenant habituelle.

```
tab = [1, 2, 3]
tab = tab + [4, 5, 6]
print tab
```

Nous nous en doutons, avec ce code, on obtient `[1, 2, 3, 4, 5, 6]`. Pas compliqué, non?

Pour concaténer plusieurs fois le même tableau, on utilise l'opérateur `*`. Pour avoir trois fois `[1, 2, 3]`, on utilise ce code.

```
tab = [1, 2, 3]
tab = tab * 3
print tab
```

L'ajout d'éléments dans un tableau est un peu plus subtil. Une manière de voir les choses est de se dire que rajouter un élément à un tableau, c'est concaténer deux tableaux, l'un des tableaux n'ayant qu'un seul élément.

Rajoutons l'élément 7 à notre tableau `tab`.

```
tab = [1, 2, 3, 4, 5, 6]
tab += [7]
print tab
```

On obtient « `[1, 2, 3, 4, 5, 6, 7]` » comme prévu. L'autre méthode, à privilégier, est d'utiliser l'opérateur `<`. Il modifie directement le tableau, ce n'est donc pas la peine de récupérer le résultat de l'opération.

L'opérateur `<` ajoute au tableau l'élément qui est à droite contrairement à l'opérateur `+` qui concatène les deux tableaux.

Ainsi, `tab < [3]` est différent de `tab + [3]`. Dans le premier cas, on ajoute au tableau un élément qui est un tableau, dans le second cas, on concatène deux tableaux, donc l'élément qui est ajouté au premier tableau est `3`, et non `[3]`.

```
tab1 = [1, 2]
tab2 = [1, 2]
tab1 << [3]
tab2 = tab2 + [3]
print "#{tab1} \n#{tab2}"
```

Avec ce code, on obtientceci.

```
[1, 2, [3]]
[1, 2, 3]
```

Cela signifie que pour ajouter un élément au tableau avec `<`, il faut utiliser la syntaxe `tab < élément`, syntaxe qui ne marche pas avec l'opérateur `+`, car cela signifierait additionner un tableau et un entier par exemple. Pour ajouter 3 à notre tableau précédent, il fallait écrire `tab < 3`.

5.2.2.1. Ajouter un élément à un indice précis

Nous avons un tableau de 3 cases. Et nous voulons ajouter un élément à la sixième case du tableau. Pour cela, nous allons utiliser les crochets (très utiles ceux-là).

```
tab = [1, 2, 3]
tab[5] = 6
print tab
```

Ce code affiche `[1, 2, 3, nil, nil , 5]`: la case qu'on voulait remplir a pris la bonne valeur et toutes les cases qui n'existaient pas ont pris la valeur `nil`.

5.2.3. Parcourir le tableau

Maintenant que nous savons comment accéder à chaque élément d'un tableau, nous pouvons envisager de parcourir le tableau en entier et de, par exemple, faire une même opération sur tous les éléments du tableau. Pour cela, il suffit d'utiliser une boucle `for`. Nous avons en effet dit qu'un tableau était une séquence, nous pouvons donc parcourir le tableau de la même manière que nous avons parcouru les intervalles.

Prenons l'exemple d'un menu de restaurant.

```
menu_viande = ["un steak haché", "une entrecôte", "un rôti",
    "un faux-filet"]
for m in menu_viande
  puts "Voulez-vous #{m} pour le dîner de ce soir ?"
end
```

Tout d'abord, nous déclarons le tableau `menu_viande`. Ensuite nous parcourons notre tableau ; la boucle `for` permet de répéter les instructions pour chaque élément du tableau. On obtient doncceci.

```
Voulez-vous un steak haché pour le dîner de ce soir ?
Voulez-vous une entrecôte pour le dîner de ce soir ?
Voulez-vous un rôti pour le dîner de ce soir ?
Voulez-vous un faux-filet pour le dîner de ce soir ?
```

En réfléchissant bien, on peut trouver une autre manière de faire cela en jouant avec les indices.

```
menu_viande = ["un steak haché", "une entrecôte", "un rôti",
    "un faux-filet"]
4.times { |i| puts
    "Voulez-vous #{menu_viande[i]} pour le dîner de ce soir ?" }
```

Ici, nous affichons l'élément d'indice `i` du tableau avec `i` qui va de 0 à 3.

Cela nous fait deux façons de parcourir le tableau. La première est plus lisible. Cependant ce n'est pas celle que nous utiliserons.

5.2.3.1. La méthode each

Pour parcourir un tableau, nous utiliserons la méthode each. Eh oui, elle fonctionne également pour les tableaux! Gardons l'exemple d'un menu de restaurant.

```
1 menu_viande = ["un steak haché", "une entrecôte", "un rôti",
      "un faux-filet"]
2 menu_viande.each { |m| puts
      "Voulez-vous #{m} pour le dîner de ce soir ?" }
```

Tout d'abord, nous déclarons le tableau menu_viande. Ensuite nous parcourons notre tableau; do répète les instructions pour chaque élément du tableau. On obtient le même résultat que précédemment.

5.2.3.2. La méthode each_with_index

Si nous avons également besoin des indices du tableau, nous utiliserons l'itérateur each_with_index qui ressemble beaucoup à each. La seule différence est qu'il y a une seconde variable, l'indice.

```
1 menu_viande = ["un steak haché", "une entrecôte", "un rôti",
      "un faux-filet"]
2 menu_viande.each_with_index { |e, i| puts
      "Voulez-vous le menu #{i} (#{e}) pour le dîner de ce soir ?" }
```

5.3. Lien avec les chaînes de caractères

Les tableaux peuvent rappeler les chaînes de caractères. Après tout, la concaténation se fait de la même manière pour les tableaux que pour les chaînes de caractères. De même, l'ajout d'élément se fait de la même manière, à l'aide de l'opérateur <. On est alors en droit de se demander si cela va plus loin.

C'est vrai ça, quel est le lien entre les tableaux et les chaînes des caractères?

Si nous disions qu'une chaîne de caractères est un tableau de caractères, nous serions en train de mentir. Mais le fait est là, on peut presque voir une chaîne de caractères comme un tableau de caractères. De là, on voit la multitude d'opérations que l'on peut effectuer sur les chaînes de caractères.

5.3.1. Accéder à un élément

Nous pouvions facilement le deviner, on accède à un élément d'une chaîne de caractères à l'aide des crochets. Ainsi, si on veut obtenir le cinquième caractère d'une chaîne de caractères, on peut utiliser ce code.

```
chaîne = 'Bonjour'
print chaîne[3] # Affiche « j ».
```

De même que pour les tableaux, l'élément d'indice `i` correspond à l'élément `i + 1` de la chaîne de caractères.

De plus, nous pouvons toujours, à la manière des tableaux, afficher une partie d'une chaîne de caractères à l'aide des intervalles. Pour afficher «onj» de la chaîne «bonjour»…

```
chaîne = 'Bonjour'
print chaîne[1..3]
```

5.3.2. Parcourir une chaîne de caractères

Nous avons déjà vu dans le chapitre précédent qu'il était possible de parcourir une chaîne de caractères avec une boucle `for`. On parcourait ainsi un par un les éléments, le séparateur étant le `\n`. Maintenant que nous savons accéder à un caractère grâce à son indice, on peut parcourir toutes les lettres de la chaîne de caractères.

```
chaîne = 'bonjour'
7.times { |i| print "#{chaîne[i]} " }
```

Mais là encore, ce n'est pas cette méthode que nous utiliserons pour parcourir tous les caractères d'une chaîne. Quelle méthode utiliserons-nous dans ce cas?

La méthode `each`? Perdu, mais presque. Nous utiliserons la méthode `each_char` (chaque caractère). Elle s'emploie de la même manière que `each` (et est aussi un itérateur), mais est spécifique aux chaînes de caractères. Nous voyons bien là que même si les chaînes de caractères et les tableaux se ressemblent, ce ne sont pas les mêmes objets. Réécrivons le code précédent.

```
chaîne = 'bonjour'
chaîne.each_char { |c| print "#{c} " }
```

Il existe également une méthode `each_line` qui permet de parcourir une chaîne de caractères par groupes de caractères. Elle prend en paramètre le séparateur (l'élément qui sépare les groupes

de caractères), le séparateur par défaut étant le retour chariot `\n` (ce qui explique le nom de la méthode qui, par défaut, permet de parcourir «chaque ligne» d'une chaîne de caractères).

```
chaîne = "bonjour\nle\nmonde."
chaîne.each_line { |l| print "#{l} " }
```

On peut aussi lire les groupes de mots séparés par exemple par une espace.

```
chaîne = 'bonjour le monde.'
chaîne.each_line(' ') { |l| print "#{l}" }
```

5.3.3. De la chaîne au tableau

Notons que nous pouvons convertir une chaîne de caractères en tableau de caractères avec la méthode `chars`. Ainsi, `'bonjour'.chars` renverra `['b', 'o', 'n', 'j', 'o', 'u', 'r']`. La méthode `split`, quand à elle donne un tableau contenant les mots de la chaîne de caractère (le délimiteur est l'espace). Ainsi, `'je suis clem'.split` renverra `['je', 'suis', 'clem']`.

> *i* Nous pouvons choisir un autre caractère que l'espace comme délimiteur. Pour cela, il suffit de le passer en paramètre de `split`. Ainsi, `'un, deux, trois'.split(', ')` renverra `['un', 'deux', 'trois']` et `'bonjour'.split('')` agira comme `chars` et renverra `['b', 'o', 'n', 'j', 'o', 'u', 'r']`.

L'opération contraire (passer du tableau à la chaîne de caractère) se fait à l'aide de la méthode `join`. Elle crée une chaîne de caractères en joignant les éléments du tableau. On peut lui donner en paramètre une chaînes de caractère qui servira de séparateur entre les éléments du tableau.

```
["un", "deux", "trois"].join              # => "undeuxtrois"
#### On place une virgule entre les éléments.
["un", "deux", "trois"].join(", ")        # => "un, deux, trois"
#### On peut l'utiliser avec des éléments qui ne sont pas des
    chaînes de caractères.
[1, 2, 3].join(", ")                      # => "1, 2, 3"
```

5.4. Exercices

5.4.1. Exercice 1

Ce premier exercice va nous faire travailler les boucles et les tableaux. Nous devons demander à l'utilisateur combien de cases il veut pour un tableau, le nombre maximum de cases étant de 10 (nous pouvons soit lui redemander un nombre de cases, soit choisir 10 par défaut s'il ne donne pas un nombre valide). Nous devons ensuite lui demander les valeurs de son tableau. Et enfin, nous devons afficher pour chaque valeur du tableau «Votre tableau contient (la valeur en question ici)». Un exemple.

```
Combien de cases (entre 1 et 10) : 2

Entrez un nombre : 2
Entrez un nombre : 4

Votre tableau contient 2.
Votre tableau contient 4.
```

Correction.

Contenu masqué n°15

5.4.2. Exercice 2

Maintenant, manipulons plusieurs tableaux. Notre programme doit afficher les valeurs de deux tableaux de même taille dans l'ordre croissant (les deux tableaux sont déjà triés à l'origine). Nous pourrons demander à l'utilisateur de nous donner des valeurs pour créer notre tableau.

Correction.

Contenu masqué n°16

5.4.3. Exercice 3

En troisième exercice, nous allons faire un petit exercice de compression. Nous devons demander à l'utilisateur une chaîne de caractères composée de 1 et de 0 et afficher le résultat de la compression (ce résultat doit être un tableau). Voici comment nous allons procéder pour la compression: nous représenterons `n` chiffres 1 d'affilée par `n1` et ferons de même pour les 0. Voici un exemple.

```
Entrez la chaîne à compresser : 110000111000110
Résultat : 214031302110
```

Nous avons de quoi bien travailler.

Correction.

Contenu masqué n°17

Ça y est, nous savons maintenant utiliser les tableaux. Grâce à ça, nous pourrons manipuler des données plus facilement.

— Un tableau est un ensemble d'éléments indexés par des entiers. On peut mettre n'importe quel type de variable dans un tableau.
— Pour parcourir un tableau, on utilise la méthode `each` ou la méthode `each_with_index` lorsqu'on a besoin des indices.
— Les chaînes de caractères peuvent être vues comme des tableaux de caractères (mais n'en sont pas). On peut parcourir une chaîne de caractères en utilisant les méthodes `each_char` et `each_line`, et on peut la convertir en tableau en utilisant la méthode `chars`.

Contenu masqué

Contenu masqué n°15

Nous décidons de lui redemander tant que la valeur entrée n'est pas bonne.

```ruby
nombre = 0
while nombre > 10 || nombre <= 0
  print 'Combien de cases (entre 1 et 10) :'
  nombre = gets.chomp.to_i
end
puts

tab = []
nombre.times do
  print 'Entrez un nombre :'
  a = gets.chomp.to_i
  tab << a # On ajoute l'élément au tableau.
end
puts

```

```
16 tab.each { |élément| print "Votre tableau contient #{élément}.\n" }
```

Retourner au texte.

Contenu masqué n°16

Pour faire cet exercice, parcourir les tableaux à l'aide de leurs indices peut être astucieux.

```
1  tab1 = [-3, 0, 5, 12, 23]
2  tab2 = [-2, 1, 2, 3, 8]
3  n = 5
4  i = 0
5  j = 0
6  while i < 5 || j < 5
7    if j == 5 || tab1[i] < tab2[j]
8      print "#{tab1[i]} "
9      i = i + 1
10   else
11     print "#{tab2[j]} "
12     j = j + 1
13   end
14 end
```

La seule chose qui pourrait nous déboussoler est le `if j == 5`. Ici, on vérifie la valeur de `j` pour rester dans le tableau. Là encore, dérouler l'algorithme à la main sur un papier aide à sa compréhension. Retourner au texte.

Contenu masqué n°17

```
1  print "Entrez la chaîne à compresser : "
2  chaîne = gets.chomp
3
4  actuel = chaîne[0]
5  nombre = 0
6  tab = []
7  chaîne.each_char do |caractère|
8    if caractère == actuel
9      nombre = nombre + 1
10   else
11     tab << nombre
12     tab << actuel
13     actuel = caractère
14     nombre = 1
```

```
15     end
16   end
17   tab << nombre
18   tab << actuel
19   résultat = tab.join
20   print resultat
```

Ce que nous avons fait fonctionne avec des 1 et des 0, mais aussi avec n'importe quel autre caractère. Retourner au texte.

6. Les méthodes

Nous allons maintenant aborder une notion importante: celle de **méthode**. Les méthodes nous permettent de factoriser notre code, c'est-à-dire d'écrire un bloc de code une seule fois pour l'utiliser à plusieurs endroits facilement. Elles permettent donc d'écrire du code plus rapidement et plus facilement.

6.1. Principe et schéma d'une méthode

6.1.1. Qu'est-ce qu'une méthode ?

Une méthode est un ensemble d'instructions permettant de réaliser une certaine tâche (additionner deux nombres, par exemple). Nous pouvons créer des méthodes pour faire tout et n'importe quoi. Si nous voulons faire une méthode `répéterMessage` qui affiche un message 10 fois, nous pouvons. Les méthodes `puts` et `print` affichent des messages, par exemple.

Lorsque nous utilisons une méthode dans un programme, on dit que l'on appelle cette méthode. Tout comme nous pouvons utiliser `print` autant de fois que nous le voulons, nous pouvons appeler n'importe quelle méthode autant de fois que nous le voulons dans notre programme.

Les méthodes permettent de n'écrire le code qu'une seule fois. Imaginons que nous ayons un code de 20 lignes qui fasse une action, et que nous fassions cette même action à trois endroits différents de notre programme. Plutôt que de recopier le code à chaque fois, nous allons faire une méthode qui effectue cette action et l'appeler à chaque fois. Ainsi, plutôt que d'écrire 60 lignes, nous allons écrire 23 lignes, 20 lignes pour la méthode, et une ligne par appel à la méthode (en fait ça fera 25 lignes).

On peut donc voir les méthodes comme des sous-programmes. Elles permettent de rendre le programme plus *modulable*.

6.1.2. Déclarer une méthode

Voyons maintenant comment déclarer une méthode en Ruby. Voici le schéma d'une méthode.

```ruby
#### Définition de la méthode add.
def add
  # Instructions.
end
```

La définition d'une méthode débute par le mot-clé `def` et se termine par `end`. Ici, nous avons déclaré la méthode `add`. Une fois qu'une méthode est déclarée, nous pouvons l'appeler dans notre programme comme n'importe quelle autre méthode. Par exemple, on pourrait appeler la méthode `add` de cette manière.

```
print 'J'appelle la méthode add : '
add
print 'C'est fait.'
```

Bien sûr, si une méthode n'est pas définie, nous ne pouvons pas l'appeler et nous obtiendrons une erreur. Ainsi, ce code ne fonctionnera pas.

```
a = 4
b = 5
bonjour
```

Le code qui va suivre est par contre parfaitement valable. La méthode `bonjour`, même si elle ne fait rien du tout, y est définie.

```
def bonjour
  # Instructions.
end

a = 4
b = 5
bonjour
```

> ! Une méthode doit être définie avant son utilisation.

Ainsi, ce code ne fonctionne pas.

```
a = 4
b = 5
bonjour

def bonjour
  # Instructions.
end
```

On pourrait pourtant penser que ce code est le même que le précédent. Cependant, quand l'interpréteur arrive à `bonjour`, il ne connaît pas encore `bonjour`, et donc on obtient une erreur.

En Ruby, il est conseillé d'écrire les noms des méthodes en convention « snake_case » (comme les noms de variable).

6.2. Écrire des méthodes

Nous allons maintenant écrire nos premières vraies méthodes. Certaines méthodes retournent une valeur particulière (la méthode `gets`, par exemple, retourne la chaîne de caractères entrée par l'utilisateur). On peut donc demander à nos méthodes de retourner une valeur particulière, mais ce n'est pas obligatoire. Celles qui n'en renvoient pas renvoient `nil`. Ce sont ces méthodes que nous allons voir, pour commencer.

6.2.1. Procédure

Nous avons dit qu'une méthode qui ne retourne pas de valeur particulière retourne `nil`. On peut alors voir `nil` comme une valeur par défaut qui **signifie** qu'aucune valeur **réelle** n'est renvoyée (il faut comprendre que `nil` est renvoyée si le concepteur de la méthode n'a spécifié aucune valeur de retour). La méthode `print`, par exemple, renvoie `nil`. Pour voir cela, retournons sur IRB et tapons `print`.

```
> print
=> nil
```

Une méthode qui retourne `nil` s'appelle une procédure. Prenons l'exemple d'une méthode qui affiche «Hello World!».

```
#### Définition de la méthode bonjour.
def bonjour
  print 'Hello World!'
end

bonjour # On appelle la méthode. Elle affiche « Hello World! ».
```

Cette procédure ne fait qu'un bête affichage. Nous pouvons bien sûr changer cet affichage par ce que nous voulons et mettre autant d'instructions que nous le voulons dans notre méthode.

Les procédures nous permettent d'éviter de devoir réécrire plusieurs fois des instructions identiques que nous utilisons souvent: nous écrivons une fois la procédure et nous l'appelons autant de fois que nous le voulons.

6.2.2. Valeur de retour

Les procédures sont très utiles, mais une méthode qui renvoie quelque chose, c'est tout aussi bien. Pour étudier ce type de méthodes, nous allons reprendre l'exemple du début de ce chapitre: nous allons écrire une méthode qui permet d'additionner deux nombres (nous l'appellerons `add`). Cette méthode devra retourner le résultat de l'addition. Voici comment cela se fait.

```
#### Définition de la méthode add.
def add
  x = 9
  y = 5
  return x + y # Retour.
end

print add # Nous devons utiliser print pour afficher la valeur de
    retour.
```

Nous avons vu comment nous retournons la valeur: avec le mot-clé `return`. `return x + y` permet donc de retourner la valeur de `x + y`. Dans cet exemple, `add` retourne `x + y`, soit `9 + 5`, et donc `14`. `print add` signifie afficher la valeur retournée par `add`, c'est-à-dire afficher `x + y` et donc, ici, `14`.

Le `return` n'est pas obligatoire. Nous aurions parfaitement pu écrire ceci.

```
def add
  x = 9
  y = 5
  x + y # Retour.
end

print add
```

> **!** Le mot-clé `return` met fin à la méthode. Les instructions après le `return` ne seront pas exécutées.

Ainsi, dans ce code, le second `print` ne sera pas exécuté.

```
def méthode
  print 'Bonjour.'
  return 0
  print 'Merci.'
end
```

En fait, on peut utiliser `return` sans aucune valeur. Dans ce cas, aucune valeur ne sera retournée, le `return` servira juste à mettre fin à la méthode. On peut donc écrire ceci.

```
def méthode
  print 'Bonjour.'
  return
  print 'Merci.'
end
```

6.2.3. Redéfinition de méthode

Quand on définit des méthodes, il faut faire attention. Si on définit plusieurs fois la même méthode, la version qui sera retenue quand on appellera la méthode sera la dernière que l'on aura définie. Ainsi, dans le code suivant, la méthode `bonjour` fera deux choses différentes.

```
def bonjour
  puts 'Bonjour.'
  return 5
end

puts bonjour

def bonjour
  print 'Bonjour, utilisateur.'
  return 10
end

print bonjour
```

Le premier appel à `bonjour` affiche «Bonjour.» et retourne 5. Le second appel à `bonjour` affiche «Bonjour, utilisateur.» et retourne 10. Finalement, ce code affichececi.

```
Bonjour.
5
Bonjour, utilisateur.10
```

Il faut donc faire attention à comment on définit nos méthodes. En fait, il vaut mieux ne jamais redéfinir une méthode et juste faire attention aux noms qu'on leur donne.

6.3. Les paramètres

Tout à l'heure, nous avons affiché «Bonjour» suivi de rien ou de «utilisateur» en redéfinissant la méthode. Nous allons maintenant voir la façon correcte de faire ceci grâce aux arguments des méthodes.

Les arguments sont des paramètres qui peuvent influer sur l'exécution d'une méthode. Les méthodes `print` et `puts`, par exemple, prennent en argument des données à afficher. En fonction des données passées, ce n'est pas la même chose qui est affichée.

6.3.1. Méthodes à un argument

Commençons par voir comment passer un argument à une méthode. Nous devons juste ajouter le nom que nous voulons donner à notre paramètre après le nom de la méthode. Nous pouvons le mettre entre parenthèses. Ce n'est pas obligatoire, mais c'est conseillé et nous le ferons (en fait, nous pouvons aussi utiliser les parenthèses pour les procédures et écrire `def f()`, mais c'est déconseillé). Voici le squelette d'une méthode avec argument.

```
def nom_méthode(nom_argument)

end
```

Nous pouvons alors utiliser l'argument dans notre méthode comme n'importe quelle autre variable. Ainsi, le bon code pour la méthode `bonjour` serait celui-ci.

```
def bonjour(nom)
  puts "Bonjour #{nom}."
end

bonjour('utilisateur')
bonjour('')
```

On obtient alors ceci.

```
Bonjour utilisateur.
Bonjour .
```

Notons que l'on appelle alors la méthode en écrivant l'argument à côté entre parenthèses. Là encore, les parenthèses ne sont pas obligatoires, mais elles sont conseillées lorsque nous utilisons des méthodes que nous avons écrites. Elles sont déconseillées dans de rares cas (par exemple avec les méthodes comme `print` et `puts`) Bien sûr, si nous n'appelons pas la méthode en lui passant le bon nombre d'arguments, nous aurons une erreur.

> **!** Les arguments ne peuvent pas être utilisés en dehors de la méthode. Ils sont définis uniquement dans la méthode.

Ainsi, ce code provoquera une erreur.

```
def bonjour(nom)
  puts "Bonjour #{nom}."
end

bonjour('utilisateur')
bonjour('')

print nom
```

6.3.2. Méthodes à plusieurs arguments

Les méthodes à plusieurs arguments ne sont pas plus compliquées à utiliser. La seule différence avec les méthodes à un argument est que nous devons utiliser une virgule pour séparer les arguments, autant dans la définition de la méthode que dans son appel. Les règles à suivre restent les mêmes. Le squelette de la méthode devient alors celui-ci.

```
def nom_fontion(nom_argument1, nom_argument2, nom_argument3,
    nom_argument_n)

end
```

Ce n'est vraiment pas plus compliqué que les méthodes à un argument.

Reprenons l'exemple de la méthode `add`. Les arguments permettent de définir la valeur de `x` et de `y` lors de l'appel de la méthode. Comme ceci.

```
def add(x, y)
  return x + y
end

print add(4, 5) # Nous affichons la valeur retournée par add(4, 5).
```

> **!** Là encore, le nombre d'arguments lors de l'appel doit être le même que lors de la définition de la méthode. Dans notre cas, il n'y en que deux : `x` et `y`.

6.3.3. Valeurs par défaut

Nous avons dit qu'il fallait passer le bon nombre d'arguments à la méthode lors de son appel. Néanmoins, ce serait bien de pouvoir se passer de certains arguments lors de certains appels, non ? Par exemple, pour notre méthode `bonjour`, de ne pas avoir à passer une chaîne vide en paramètre lorsque nous ne voulons pas afficher de nom.

Nous sommes fiers d'annoncer que nous pouvons faire tout cela en donnant à notre argument une valeur par défaut lorsque nous définissons notre méthode. En faisant cela, lorsque nous appellerons la méthode sans lui donner cet argument, sa valeur sera la valeur par défaut.

Pour définir une méthode avec une valeur par défaut, il faut écrire l'argument dans la définition de la méthode en lui donnant déjà une valeur. Notre méthode `bonjour` se code alors ainsi.

```
1 def bonjour(nom = '')
2   puts "Bonjour #{nom}."
3 end
4
5 bonjour('utilisateur')
6 bonjour
```

Nous obtenons bien le résultat attendu.

Nous pouvons faire des méthodes avec plusieurs arguments facultatifs. Cependant, ces derniers doivent être les derniers paramètres de notre méthode.

Quand on y réfléchit, c'est logique. Si on crée une méthode comme ceci (`def méthode(argument1, argument2 = 10, argument3)`), comment savoir à quoi correspond `12` dans cet appel : `méthode(3, 12)`. Ici, on devine qu'il correspond au troisième argument, mais dans d'autres cas, nous voyons bien que c'est impossible.

Notons tout de même que nous pouvons choisir l'ordre dans lequel nous transmettons nos paramètres à une méthode. Il faut dans ce cas préciser le nom de l'argument.

Regardons ce code, par exemple.

```
1 def bonjour(nom, prénom)
2    puts "Bonjour #{nom} #{prénom}."
3 end
4
5 bonjour('moi', 'utilisateur')
6 bonjour(prénom = 'utilisateur', nom = 'moi')
```

L'argument `prénom` vaut `"utilisateur"` dans les deux cas et l'argument `nom` vaut `"moi"` dans les deux cas. Les deux appels à la méthode donnent donc le même résultat.

6.4. Exercices

6.4.1. Exercice 1

Voici le premier exercice: faire une calculatrice! Nous devons gérer les quatre opérations usuelles: l'addition, la soustraction, la multiplication et la division. L'utilisateur devra choisir une opération, puis saisir deux nombres. Nous afficherons le résultat de l'opération. Voici un exemple de ce que doit afficher notre programme.

```
Choisissez une opération.

1. Addition.
2. Soustraction.
3. Multiplication.
4. Division.

Votre choix : 3

Vous avez choisi la Multiplication.

Entrez le premier nombre : 4
Entrez le second nombre : 23

4.0 * 23.0 = 92
```

Correction.

Contenu masqué n°18

6.4.2. Exercice 2

Exercice un peu plus difficile cette fois. Notre programme doit demander deux nombres à l'utilisateur et afficher le PGCD de ces deux nombres. Nous pouvons aller faire un tour sur la page Wikipédia du PGCD pour nous rafraîchir la mémoire.

Correction.

Contenu masqué n°19

6.4.3. Exercice 3

Nous allons finir par faire un convertisseur. Nous devons demander à l'utilisateur de choisir une conversion et afficher le résultat de cette conversion. Voici les conversions que la première version de notre programme doit gérer (dans les deux sens bien sûr):

— conversion de degrés en radians (1 radian = 180/ degrés);
— conversion de km/h en m/s (1 m/s = 3.6 km/h).

Ces deux conversions sont très utiles en physique. Nous devons également réaliser un menu dans lequel nous proposerons à l'utilisateur de réaliser une des conversions ou de quitter le programme.

Correction.

Contenu masqué n°20

Maintenant que nous avons réalisé ce programme, nous pouvons le compléter en rajoutant quelques autres conversions:

— conversion de pieds en mètres (1 pied = 0.3048 mètres);
— conversion de grammes en livres (1 gramme = 0.002205 livres);
— conversion de degrés Fahrenheit en degrés Celsius (température en degrés Fahrenheit = 32 + 1.8 × température en degrés Celsius);
— conversion de diverses devises (euro, dollar, livre, etc.).

Nous pouvons même (et c'est un bon entraînement) faire un programme de conversion d'un nombre vers une autre base. Gérer le binaire, l'octal, le décimal et l'hexadécimal est suffisant pour commencer, mais notre programme devrait à terme être capable de convertir un nombre de n'importe quelle base en n'importe quelle autre base.

En cas de pépin, n'oublions pas que le forum est là pour nous aider.

C'est la fin de ce chapitre. Nous savons maintenant suffisamment de choses pour bien faire nos méthodes.

Nous pouvons reprendre l'exercice de compression du chapitre précédent et l'améliorer en utilisant les méthodes. Nous pouvons écrire une méthode qui compresse et une autre qui décompresse… Nous pouvons même rendre notre calculatrice plus performante. Dans tous les cas, il est nécessaire de pratiquer un peu avant de passer à la suite. Les méthodes sont un élément très important qu'il faut maîtriser.

— Les méthodes permettent d'éviter de réécrire le même code plusieurs fois.
— Les méthodes peuvent retourner des valeurs, une méthode retournant `nil` étant une procédure.
— Les méthodes peuvent prendre des arguments, dont certains peuvent être facultatifs.

Contenu masqué

Contenu masqué n°18

```ruby
def menu
  puts 'Choisissez une opération.'
  tab = %w[Addition Soustraction Multiplication Division]
  tab.each_with_index { |op, idx| puts "#{idx + 1}. #{op}." }
  choix = 0
  while choix < 1 || choix > 4
    print "\nVotre choix : "
    choix = gets.chomp.to_i
  end
  puts "\nVous avez choisi la #{tab[choix - 1]}."
  return choix
end

choix = menu
print "\nEntrez le premier nombre : "
nombre1 = gets.chomp.to_f
print 'Entrez le second nombre : '
nombre2 = gets.chomp.to_f
puts
if choix == 1
  print "#{nombre1} + #{nombre2} = #{nombre1 + nombre2}"
elsif choix == 2
  print "#{nombre1} - #{nombre2} = #{nombre1 - nombre2}"
elsif choix == 3
  print "#{nombre1} * #{nombre2} = #{nombre1 * nombre2}"
elsif nombre2 != 0
  print "#{nombre1} / #{nombre2} = #{nombre1 / nombre2}"
else
  print 'Diviser par 0, c’est mal.'
end
```

Retourner au texte.

Contenu masqué n°19

```ruby
def pgcd(a, b)
  if a > b
    tmp = a
    a = b
    b = tmp
```

```
6    end
7    if a == 0
8      return b
9    end
10   while b % a != 0
11     tmp = b % a
12     b = a
13     a = tmp
14   end
15   return a
16 end
```

On peut aussi faire cette fonction de manière récursive en utilisant le fait que le PGCD de deux entiers est le PGCD du plus petit entier et du reste de la division euclidienne de ces deux entiers, les cas terminaux étant celui où l'un des deux nombres est nul (auquel cas le PGCD est l'autre nombre) et celui où lorsque le reste est nul (le PGCD est alors le plus petit des nombres).

```
1 def pgcd(a, b)
2   return b if a == 0
3   return a if b == 0
4   r = a % b
5   r == 0 ? b : pgcd(b, r)
6 end
```

En fait, la méthode `gcd` qui s'applique à un entier nous permet déjà d'avoir le PGCD de deux nombres (et oui, la boîte à outils de Ruby est très complète). Ainsi, `36.gcd(30)` renvoie `6`. Retourner au texte.

Contenu masqué n°20

Le but des méthodes est d'avoir un code bien découpé. Utilisons-les donc pour avoir un beau programme (en plus, c'est un peu le but, vu le nom du chapitre).

```
1 def menu
2   puts '1. Convertir des radians en degrés.'
3   puts '2. Convertir des degrés en radians.'
4   puts
       '3. Convertir des kilomètres par heure en mètres par seconde.'
5   puts
       '4. Convertir des mètres par seconde en kilomètres par heure.'
6   puts '5. Quitter.'
7   choix = 0
8   while choix < 1 or choix > 5
```

```
    print "\nVotre choix : "
    choix = gets.chomp.to_i
  end
  return choix
end

PI = 3.14159265359
DEG_VERS_RAD = 180 / PI
MS_VERS_KMH = 3.6
choix = menu
if choix != 5
  print "\nEntrez le nombre : "
  nombre = gets.chomp.to_f
  if choix == 1
    print nombre * DEG_VERS_RAD
  elsif choix == 2
    print nombre / DEG_VERS_RAD
  elsif choix == 3
    print nombre / MS_VERS_KMH
  else
    print nombre * MS_VERS_KMH
  end
end
```

Retourner au texte.

7. Les hachages

Nous venons de terminer un chapitre sur les conteneurs. Il est maintenant temps de faire un chapitre sur les conteneurs.

Quoi, un autre chapitre sur les conteneurs? Mais, nous en avons déjà fait un, non?

Oui, nous avons fait le chapitre sur les tableaux. Dans ce chapitre, nous allons aborder un autre type de conteneurs, les **tableaux associatifs** aussi appelés ***hashs*** ou encore **hachages**.

7.1. Des tableaux associatifs

7.1.1. Qu'est-ce qu'un hachage

Nous avons dit que nous allions voir un nouveau type de conteneurs. En fait, un hachage n'est rien d'autre qu'un tableau spécial. Le nom tableau associatif peut nous aider à comprendre en quoi ils sont spéciaux. En fait, au lieu d'associer une valeur à un nombre entier comme dans un tableau normal, nous allons associer une valeur à… ce que nous voulons. Nous pourrons par exemple associer à une chaîne de caractères une autre chaîne de caractères.

```
tableau[2]      # Un tableau normal.
hachage['deux'] # Ce qu'on peut faire avec un hachage.
```

L'idée générale est simple à comprendre.

Mais, à quoi ça sert?

Bonne question. Supposons que notre but soit de stocker des informations sur une personne. Nous voulons son nom, son prénom et son âge. Nous pourrions créer un tableau normal.

```
personne = ['nom de la personne', 'prénom de la personne',
    'âge de la personne']
```

Mais ce ne serait pas très évident à utiliser. Un tableau associatif serait bien mieux. On aurait un tableau associatif à trois cases.

- la case `'nom'` associée à son nom;
- la case `'prénom'` associée à son prénom;
- la case `'âge'` associée à son âge.

Ce sera non seulement plus facile à écrire, mais aussi à lire (si on passe beaucoup de temps à écrire un programme, on passe encore plus de temps à le lire).

On appelle clé (*key* en anglais) ce qu'on a choisi comme identifiant, et on appelle valeur ce à quoi on accède grâce à la clé (les éléments du hachage).

Dans notre exemple, `nom`, `prénom` et `âge` seraient donc des clés, alors que le nom de la personne, son prénom et son âge seraient des valeurs.

7.1.2. Déclarer un hachage

Maintenant que nous savons ce que sont les hachages et que nous savons à quoi ils peuvent servir, il ne nous reste plus qu'à les utiliser. Pour déclarer un hachage, il faut utiliser des accolades à la place des crochets du tableau. Pour déclarer un hachage vide, il faut donc utiliser ceci.

```
hachage = {}
```

On peut l'afficher avec `print`.

```
hachage = {}
print hachage
```

Code qui nous affiche bien entendu `{}`.

De plus, pour indiquer qu'on associe une valeur à une autre, il faut utiliser `=>`. Les éléments sont, comme pour les tableaux, séparés par une virgule. Déclarons le hachage de notre exemple précédent.

```
hachage = { 'nom'    => 'Mon nom',
            'prénom' => 'Mon prénom',
            'âge'    => 2015 }
print hachage
```

Cette fois, on obtient affiché à l'écran: `{"nom"=>"Mon nom", "prénom"=>"Mon prénom", "âge"=>2015}`.

> ! Contrairement aux tableaux, les hachages ne peuvent pas être déclarés sans utiliser les accolades, elles sont obligatoires.

Nous avons dit que nous pouvions utiliser n'importe quoi comme clé. Faisons un hachage avec des nombres et des chaînes de caractères comme clé.

```
hachage = { 'abc' => 'dcvs',
            2     => 'deux',
            3.4   => 23 }
print hachage
```

Et on obtient: `{"abc"=>"dcvs", 2=>"deux", 3.4=>23}`.

7.2. Opérations sur les hachages

7.2.1. Accéder à un élément

Si nous retournons à la partie précédente, nous verrons que nous l'avons déjà fait. Comme pour les tableaux, il faut utiliser les crochets, et écrire entre eux la clé de l'élément auquel on veut accéder. Nos clés peuvent être tout et n'importe quoi.

```
hachage = { 'nom'    => 'Mon nom',
            'prénom' => 'Mon prénom',
            'âge'    => 2015 }
print hachage['nom']
```

Grâce à ce code, nous affichons la chaîne « Mon nom ».

On peut se dire que nous sommes en train de radoter, mais ce n'est pas de notre faute si les tableaux et les hachages se ressemblent tant. En fait, ils se ressemblent tellement que même en essayant d'accéder à une valeur associée à une clé qui n'existe pas, on a la même réponse.

```
hachage = { 'nom'    => 'Mon nom',
            'prénom' => 'Mon prénom',
            'âge'    => 2015 }
print hachage['Nom']
```

Ce code affiche en effet `nil`, tout comme `print tab[5]` lorsque `tab` a moins de 6 cases.

Encore une fois on se répète, mais ce n'est pas parce que cet accès ne provoque pas d'erreur qu'on peut le faire. Il faut essayer d'éviter toutes erreurs de ce type.

7.2.2. Ajout d'éléments

Les opérateurs `+` et `<` ne marchent pas avec les hachages. En fait, la seule manière de rajouter un élément est d'utiliser les crochets pour modifier un élément qui n'existe pas encore.

```
hachage = { 'nom'    => 'Mon nom',
            'prénom' => 'Mon prénom',
            'âge'    => 2015 }
hachage['autre'] = "nouveau"

print hachage
```

La clé `'autre'` n'existe pas encore, on lui assigne une valeur.

D'ailleurs, qu'est-ce qui se passe si on assigne deux fois une valeur à la même clé lorsqu'on déclare un hachage?

Faisons le test.

```
hachage = { 'nom'    => 'Mon nom',
            'prénom' => 'Mon prénom',
            'âge'    => 2015,
            'nom'    => 'Mon nouveau nom' }
print hachage
```

Et le résultat.

```
{"nom"=>"Mon nouveau nom", "prénom"=>"Mon prénom", "âge"=>2015}
```

La valeur qui a été gardée pour la clé est la dernière que l'on avait associée. On ne peut donc associer qu'une seule valeur à une clé.

Pour faire une analogie avec un autre langage, les hachages de Ruby sont les dictionnaires de Python.

7.2.3. Parcourir le hachage

Là on va arrêter de radoter! Parcourir un hachage ne se fait pas du tout de la même manière que parcourir un tableau. Et c'est bien normal, on ne peut pas parcourir un hachage grâce aux indices puisqu'il n'y a pas d'indices. En fait, la seule méthode commune est celle qui consiste à parcourir directement le hachage avec `in`.

```
hachage = { 'nom'     => 'Mon nom',
            'prénom' => 'Mon prénom',
            'âge'     => 2015 }

for i in hachage
  print "#{i}\n"
end
```

On déclare notre hachage, et on le parcourt avec une boucle `for`, pour finalement obtenir ceci.

```
["nom", "Mon nom"]
["prénom", "Mon prénom"]
["âge", 2015]
```

Et là, nous sommes au regret de dire que tout comme pour les tableaux, nous n'allons pas utiliser cette technique, nous allons plutôt utiliser des méthodes.

7.2.3.1. Parcourir les clés et les valeurs

Les hachages ont, comme les tableaux, une méthode `each`. Cependant, celle des hachages permet de parcourir les clés et les valeurs. Voyons un exemple.

```
hachage = { 'nom'     => 'Mon nom',
            'prénom' => 'Mon prénom',
            'âge'     => 2015 }

hachage.each { |clé, valeur| puts
    "La valeur #{valeur} est associée à la clé #{clé}." }
```

On parcourt tout le hachage en stockant chaque clé et chaque valeur dans les variables `clé` et `valeur`.

La méthode `each_with_index` existe également pour les hachages. Cependant, avec elle, les clés sont associées à des indices. Ainsi, avec le code qui suit, nous obtiendrons «La valeur [”nom”, ”Mon nom”] est associée à l'indice 0».

```
1 hachage = { 'nom'    => 'Mon nom',
2             'prénom' => 'Mon prénom',
3             'âge'    => 2015 }
4
5 hachage.each_with_index { |v, i| puts
      "La valeur #{v} est associée à l’indice #{i}." }
```

7.2.3.2. Parcourir les valeurs

Avoir les valeurs et les clés c’est bien, mais ce serait bien de ne récupérer que les valeurs. Pour cela, nous allons utiliser la méthode `each_value`.

```
1 hachage = { 'nom'    => 'Mon nom',
2             'prénom' => 'Mon prénom',
3             'âge'    => 2015 }
4
5 hachage.each_value { |valeur| puts
      "La valeur #{valeur} est dans le hachage." }
```

Le code se passe de description. Nous devrions pouvoir le comprendre tranquillement.

7.2.3.3. Parcourir les clés

C’est plus rare de vouloir accéder aux clés d’un hachage, mais cela arrive. Heureusement, la méthode `each_key` est là pour nous aider à faire cela.

```
1 hachage = { 'nom'    => 'Mon nom',
2             'prénom' => 'Mon prénom',
3             'âge'    => 2015 }
4
5 hachage.each_key { |clé| puts
      "La clé #{clé} est une des clés du hachage." }
```

7.3. Hachages et tableaux

Bon, maintenant que nous avons vu les hachages et certaines opérations possibles sur eux, il nous faut savoir quand les utiliser. Nous n’avons en effet vu qu’un seul exemple jusqu’à maintenant, et nous l’avons utilisé pour toutes nos opérations.

Quelles sont les avantages et les inconvénients des hachages face aux tableaux? Lequel choisir?

C'est ce que nous allons voir dans cette partie. Pour cela, nous allons nous attarder sur trois points:

— la clarté;
— l'ordre;
— la flexibilité.

7.3.1. La clarté

Sur ce point, les hachages sont mieux que les tableaux. En effet, alors que les clés des tableaux ne sont que des nombres, avec les hachages, on peut choisir tout et n'importe quoi comme clé (même des nombres donc). Les hachages permettent donc de toujours avoir des clés qui ont un sens.

7.3.2. L'ordre

Sur ce point, les tableaux sont nettement mieux que les hachages. En effet, les tableaux disposent d'un ordre déjà établi du fait de leur définition (l'élément d'indice 0, l'élément d'indice 1, etc.). Au contraire, les clés des hachages ne permettent pas d'établir un ordre clair. Comment Ruby ferait-il pour savoir que l'élément associé à telle clé doit venir avant l'élément associé à telle autre clé?

Cette absence d'ordre dans les hachages est d'ailleurs ce qui produit l'impossibilité de certaines actions possibles sur les tableaux. En effet, nous avons vu par exemple que l'opérateur `+` ne s'utilisait pas sur les hachages. Quelle est la raison de cela? En fait, avec deux tableaux, l'opérateur `+` ajoute les éléments du deuxième tableau aux éléments du premier tableau **en conservant l'ordre**. Ainsi, en faisant `[2, 3] + [4, 5]`, on obtient bien `[2, 3, 4, 5]` et non `[2, 3, 5, 4]` ni `[4, 5, 2, 3]`. Cela n'est pas possible avec les hachages. De même, l'opérateur `<` qui ajoute un élément à la fin d'un tableau n'est pas disponible avec les hachages (on ne sait pas où est la fin d'un hachage).

Et si on donnait comme clé à nos hachages des nombres, ça marcherait, non?

Allons-y, testons ce code.

```ruby
hachage = { 1 => 1,
            2 => 2,
            3 => 3 }

hachage.each { |clé, valeur| puts "#{clé} : #{valeur}" }
```

Les éléments sont affichés dans le bon ordre. Super!

Mais, regardons maintenant ce code.

```
hachage = { 1 => 1,
            2 => 2,
            3 => 3 }

hachage[5] = 5
hachage[4] = 4

hachage.each { |clé, valeur| puts "#{clé} : #{valeur}" }
```

La clé `5` est passée avant la clé `4`. On perd notre ordre, et pourtant, nous n'avons fait rien d'autre qu'un simple ajout de valeurs.

Il ne faut pas essayer de faire passer un tableau pour un hachage. Les propriétés d'ordre du tableau ne seraient pas du tout respectées.

7.3.3. La flexibilité

Voyons un peu les différentes opérations possibles sur les tableaux et les hachages et comparons-les:

- l'ajout d'élément est possible sur les deux;
- la modification est possible sur les deux;
- la concaténation n'est possible que sur les tableaux mais correspond juste à une suite d'ajouts d'éléments.

Les hachages et les tableaux ont l'air aussi flexibles l'un que l'autre. Les opérations qu'on peut effectuer sur les deux sont complètes et permettent de modifier radicalement la structure.

Finalement, aucune structure n'est mieux que l'autre. Elles ont toutes les deux leurs points forts et leurs points faibles. Tout simplement, parce qu'elles ne sont pas adaptées aux même situations. Les tableaux sont utiles dans certains cas, les hachages dans d'autres.

7.4. Exercices

Plutôt que de faire plusieurs exercices, nous allons en faire un seul en plusieurs étapes. Notre but va être de gérer une liste d'élèves. Un élève aura un nom, un prénom et un âge (nous pourrons tester que l'âge de l'élève est valide, mais ce n'est pas obligatoire). Notre programme devra proposer à l'utilisateur trois choix:

- ajouter un élève;
- afficher la liste des élèves;

— quitter.

Un exemple.

```
1. Ajouter un élève.
2. Afficher la liste des élèves.
3. Quitter.

Votre choix ? 1

Nom : ZdS
Prénom : Clem
Âge : 2

L'élève ZdS Clem a été ajouté.

1. Ajouter un élève.
2. Afficher la liste des élèves.
3. Quitter.

Votre choix ? 1

Nom : SdZ
Prénom : Zozor
Age : 10

L'élève SdZ Zozor a été ajouté.

1. Ajouter un élève.
2. Afficher la liste des élèves.
3. Quitter.

Votre choix ? 2

- SdZ Zozor
- ZdS Clem
```

Correction.

Contenu masqué n°21

Maintenant, rajoutons une fonctionnalité pour lire les informations d'un élève en particulier, pour une sortie de ce genre.

```
1. Ajouter un élève.
2. Afficher la liste des élèves.
```

```
3. Informations d’un élève.
4. Quitter.

Votre choix ? 3

Nom de l’élève : ZdS

L’élève Zds Clem a 2 ans.
```

Si plusieurs élèves ont le même nom, les descriptions de tous ces élèves devront être affichées.

Correction.

Contenu masqué n°22

Nous pouvons encore ajouter d’autres options à notre menu, comme la possibilité de supprimer un élève. Nous pouvons également complexifier notre structure en ajoutant, par exemple, la liste des matières suivies par chaque élève à son hachage (cette liste serait un tableau). Nous aurions alors un tableau contenant un hachage qui contient lui-même un tableau.

Et voilà c’est la fin de ce chapitre qui sera très utile pour la suite, les hachages étant un élément important de Ruby.

— Un hachage est un ensemble d’éléments indexés par ce qu’on veut (on peut associer chaque élément à ce qu’on veut) et se trouve pour cela appelé tableau associatif.
— Pour parcourir un hachage, on utilise la méthode `each`. Les méthodes `each_value` et `each_key` nous permettent respectivement de parcourir les valeurs et les clés.
— Les hachages et les tableaux sont complémentaires et il faut choisir lequel utiliser en fonction des besoins et de la situation.

Contenu masqué

Contenu masqué n°21

Nous allons faire un tableau dont les éléments sont des hachages. Ces hachages seront nos différents élèves. Notre programme consistera alors à gérer ce tableau.

Nous allons faire une méthode `choisir` qui affiche le menu et renvoie le choix de l’utilisateur, une méthode `ajouterÉlève` qui ajoute un élève au tableau, et une méthode `afficherListe` qui affiche la liste des élèves.

```ruby
def ajouterÉlève(tab)
  print "\nNom : "
```

```
3    nom = gets.chomp
4    print 'Prénom : '
5    prénom = gets.chomp
6    print 'Âge : '
7    âge = gets.chomp.to_i
8    tab << { 'nom'     => nom,
9             'prénom' => prénom,
10            'âge'     => âge }
11   print "\nL'élève #{nom} #{prénom} a été ajouté.\n\n"
12 end
13
14 def afficherListe(tab)
15   tab.each { |e| puts "- #{e["nom"]} #{e["prénom"]}" }
16   print "\n\n"
17 end
18
19 def(choisir)
20   print
       "1. Ajouter un élève.\n2. Afficher la liste des élèves.\n3. Quitter.\n\n"
21   print 'Votre choix ? '
22   return gets.chomp.to_i
23 end
24
25 tab = []
26 choix = 0
27
28 while choix != 3
29   choix = choisir
30   if choix == 1 then
31     ajouterÉlève(tab)
32   elsif choix == 2 then
33     afficherListe(tab)
34   end
35 end
```

Retourner au texte.

Contenu masqué n°22

Pour cela, nous allons créer une méthode `informationsÉlève`. Il ne faudra pas non plus oublier de mettre à jour la méthode `choisir` et le code principal. On a le code suivant.

```
1 def informationsÉlève(tab)
2   puts 'Nom de l'élève : '
3   nom = gets.chomp
4   print "\n\n"
```

```
5   tab.each { |e| puts
        "L’élève #{e["nom"]} #{e["prénom"]} a #{e["âge"]} ans." if
        e["nom"] == nom }
6   print "\n\n"
7 end
```

Retourner au texte.

8. Retour sur les variables

C'est parti pour un nouveau chapitre. Ici, nous reviendrons aux variables en approfondissant ce que nous avons vu dans le deuxième chapitre.

8.1. Une histoire de références

8.1.1. Qu'est-ce qu'une variable ?

Nous les utilisons depuis le premier chapitre de ce tutoriel et nous nous n'avons jamais vraiment répondu à cette question? Bon, cela doit changer, voyons un peu ce qui se passe quand on déclare une variable.

Dans d'autres langages, on peut apprendre qu'une variable est une boîte. Une simple boîte dans laquelle on range une valeur. Pour accéder à cette valeur, on ouvre la boîte. Pour changer la valeur d'une variable, on change le contenu de cette boîte. Simple, non?

Mais cette idée est à supprimer. En Ruby, une variable n'est rien d'autre qu'un nom.

Quoi, un nom? Dans ce cas, pourquoi ce nom représente une valeur?

Le verbe utilisé dans la question est intéressant. Le nom **représente** une valeur. C'est exactement ça. Une variable n'est pas une boîte qui contient quelque chose, c'est juste un nom qui représente une valeur.

Bien sûr, pour cela, la variable doit contenir quelque chose pour savoir ce qu'elle est censée représenter.

Une représentation plus juste des variables pourrait être les pointeurs dans les langages comme le C (à un plus haut niveau). Un pointeur est une variable qui contient une adresse mémoire. Si deux pointeurs sont égaux, alors non seulement la valeur pointée est la même, mais en plus, l'adresse mémoire est la même.

En Ruby, cela se vérifie ainsi, si `a = 4` et `b = 4`, alors non seulement les valeurs `a` et `b` sont égales, mais en plus, elles représentent le même objet en mémoire (on ne peut pas vérifier leur adresse mémoire, mais on peut vérifier leur identifiant). Il n'y a pas un objet créé pour `a` et un autre pour `b`, leur identifiant est le même. Ce sont des **références** au même objet.

8.1.2. Une histoire d'identifiant

Mettons maintenant un nom sur ce dont nous parlons. Nous avons dit qu'une variable servait juste à représenter une valeur. Puis nous avons parlé d'identifiant. Le mot «identifiant» est le bon. D'ailleurs, il existe une méthode qu'on peut utiliser avec tout en Ruby, la méthode `object_id`. Cette méthode donne l'identifiant d'une variable.

Utilisons cette méthode pour vérifier ce que l'on a dit dans la partie précédente.

```
a = 4
b = 4
puts a.object_id
puts b.object_id
```

Comme nous l'avons dit, les deux variables ont le même identifiant.

Nous pouvons même aller encore plus loin.

```
a = 4
b = 4
puts a.object_id
puts b.object_id
puts 4.object_id
```

Bien sûr, en affectant `a` à `b`, ils ont également le même identifiant.

```
a = 4
b = a
puts a.object_id
puts b.object_id
```

8.1.2.1. Tableaux, chaînes de caractères...

Cependant, l'affaire est différente pour les tableaux, chaînes de caractères ou encore hachages. Lorsque nous créons deux fois le même tableau, par exemple, ils n'ont pas le même identifiant. Regardons ce code.

```
a = [122, 32]
b = [122, 32]
puts a.object_id
puts b.object_id
puts [122, 32].object_id
```

Comme nous pouvons le remarquer en exécutant ce code, les trois identifiants sont différents. Mais ne nous inquiétons pas, c'est dû à la manière dont nous pouvons modifier ces variables (et donc ce sera traité dans la partie qui suit).

8.1.3. Modification de variables

Dans cette partie, nous allons tenter de répondre à une question simple.

Que se passe-t-il lorsque l'on modifie une variable?

La question est simple, mais nous allons voir que la réponse ne l'est pas forcément.

Commençons par le cas de variables simples. Regardons le résultat de ce code.

```
1 a = 12
2 puts a.object_id
3 a = 15
4 puts a.object_id
```

Ce code conforte notre idée précédente: `a` ne fait plus référence au même objet, son identifiant n'est donc plus le même.

Maintenant que nous avons vu ce qui se passe lors de la modification de variable, nous pouvons passer à la partie suivante. Ah, non, il nous reste le cas des tableaux, des chaînes de caractères et des hachages qui comme tout à l'heure sont particuliers.

```
1 a = []
2 puts a.object_id
3 a = [1, 2]
4 puts a.object_id
```

Les identifiants sont différents comme tout à l'heure.

Quoi? Pourquoi? Pourtant, nous avons dit que ce cas était différent?

Oui, mais le fait est qu'ici on change de tableau. Au départ, `a` faisait référence au tableau `[]`, après la seconde affectation, il fait référence au tableau `[1, 2]`.

Cependant, lorsque nous modifions le tableau directement (donc sans affectation), la variable fait toujours référence au même tableau. Aucun nouveau tableau n'a été créé, c'est l'ancien tableau qui a été modifié.

```
a = []
puts a.object_id
a << 1
a << 2
a[0] = 23
puts a.object_id
```

Dans ce code, nous ne réaffectons pas notre variable `a`. Elle fait toujours référence au même tableau, tableau qui en revanche a été beaucoup modifié. L'identifiant reste néanmoins le même.

Et ceci nous permet de répondre à la question que nous nous posions à propos de la différence entre les tableaux et les variables plus simples: si en déclarant deux fois le même tableau, les deux variables avaient le même identifiant, alors en modifiant l'un on modifierait l'autre, ce qui n'est pas trop voulu. Voilà ce que l'on veut (et c'est ce qui se passe).

```
a = []
b = [] # L'identifiant de b est différent de celui de a.
a << 1 # a est modifié, cela ne change pas b.
c = a  # c = a donc c et a ont le même identifiant.
c << 2 # On modifie c, a est également modifié, car ils font référence au même tableau.
a = [] # On donne une nouvelle valeur à a ; c garde son ancienne valeur, par contre.
```

Cette partie était remplie de nouvelles informations, alors il faut prendre le temps de bien tout assimiler et faire des tests avant de passer à la suite.

8.2. Les variables globales

8.2.1. Un problème : passage de variables aux méthodes

Nous allons maintenant nous intéresser à un problème: on passe une variable en paramètre à une méthode, comment faire pour que les modifications effectuées sur la variable de la méthode affectent la variable que l'on a passée en argument. Un exemple de code pour voir le problème.

```
def incrémenter(a)
  a = a + 1
end

x = 6
incrémenter(x)
```

```
7 print x
```

On aimerait que la valeur de `x` soit `7` après l'appel de la méthode `incrémenter`, or elle a gardé son ancienne valeur `6`.

On pourrait alors se dire que c'est normal, qu'il suffit d'incrémenter `x` plutôt que `a` dans la méthode `incrémenter` (et dans ce cas, l'argument n'est plus nécessaire).

```
1 def incrémenter
2   x = x + 1
3 end
4
5 x = 6
6 incrémenter(x)
7 print x
```

Et ce code ne fonctionne pas, et nous obtenons une erreur. Cette erreur est due à ce que l'on appelle la **portée des variables**. Il s'agit de définir dans quelle partie du code une variable existe. Il faut donc savoir que les variables n'existent que dans le bloc dans lequel elles ont été déclarées. Ainsi, dans notre code précédent, les deux variables `x` sont différentes:

— la variable `x` de la méthode `incrémenter` n'existe que dans la méthode `incrémenter` et pas en dehors;
— la variable `x` du reste du code n'existe qu'en dehors de la méthode `incrémenter`.

On dit que ce sont des variables **locales**.

Ainsi, dans la méthode `incrémenter`, on essaie d'incrémenter la valeur de `x`, or `x` n'existe pas encore (et a donc la valeur `nil`) et l'opération `nil + 1` ne peut pas être faite.

Pour mieux voir le phénomène de portée, testons des codes de ce genre.

```
1 def f(x)
2   x = x + 1 # Fonctionne, car x existe, étant un paramètre.
3   puts x
4 end
5
6 def g
7   x = 3      # On est obligé de déclarer x avant.
8   x = x + 1
9   puts x
10 end
11
12 def h(x)
13   x = x + 1
14   puts x
15   return x    # On retourne x.
```

```
16 end
17
18 x = 1
19 f(x)
20 puts x # x vaut toujours 1 en dehors de la méthode.
21 g(x)
22 puts x # x vaut toujours 1.
23 h(x)
24 puts x # x vaut toujours 1.
25 x = h(x)
26 puts x # x vaut maintenant 2, car on a récupéré la valeur retournée
     par la méthode h.
```

Ce code nous montre qu'il est possible de faire la méthode `incrémenter` grâce au retour de la méthode (nous le savions déjà). Mais si notre méthode doit changer plusieurs valeurs, c'est déjà plus embêtant à faire.

En fait, en Ruby, les valeurs sont passées par référence (normal, puisqu'en Ruby, **tout** est référence). Ainsi, voici ce qui se passe.

```
1  def f(x)
2    puts "Au début de la méthode, l'id de x est #{x.object_id}."
3    x = x + 1
4    puts "À la fin de la méthode, l'id de x est #{x.object_id}."
5  end
6
7  x = 2
8  puts
     "En dehors de la méthode avant son appel, l'id de x est #{x.object_id}."
9  f(x)
10 puts
     "En dehors de la méthode après son appel, l'id de x est #{x.object_id}."
```

On remarque que l'identifiant de `x` au début de la fonction est le même que celui du `x` extérieur (normal, puisqu'elles font référence au même objet et que le passage de paramètre se fait par référence). Par contre, une fois l'incrémentation effectuée, l'identifiant du `x` de la fonction a changé et puisqu'il ne s'agit pas du même `x` que celui à l'extérieur, alors la valeur du `x` extérieur n'a pas changé.

Cependant, cela nous permet de découvrir quelque chose: si on sait modifier un objet sans modifier son identifiant, alors on peut le modifier dans une fonction. En particulier, on sait modifier un tableau dans une fonction.

```
1 def f(tab)
2   tab << 3
3 end
```

```
4
5 tab = [1, 2]
6 f(tab)
7 print tab
```

8.2.2. Les variables globales

Pour régler ce problème, nous pouvons utiliser une variable globale. Les variables globales sont des variables qui, contrairement aux variables locales, sont accessibles dans tout le programme. Pour déclarer une variable globale, il suffit de préfixer son nom du caractère `$`.

```
1 $x # x est une variable globale.
```

> ! Les variables `x` et `$x` sont bien sûr différentes.

Donc, ce code fonctionne.

```
1 def afficher
2   print $x
3 end
4
5 $x = 'Voici une variable globale.'
6 afficher
```

Nous sommes maintenant capables de faire une méthode qui incrémente la variable globale `$x`.

```
1 def incrémenter
2   $x = $x + 1
3 end
4
5 $x = 1
6 incrémenter
7 print $x # Affiche 2.
```

Les variables globales semblent être une solution pertinente aux problèmes que nous pourrons avoir, et nous pourrions penser sûrement à les utiliser tout le temps. Pourtant, leur utilisation peut s'avérer dangereuse et ne conduit pas à une bonne conception du programme. On peut presque toujours se passer des variables globales et c'est ce que nous ferons.

8.2.3. Variables globales réservées

Il existe des variables globales dont le nom est réservé. Cela veut dire que nous ne pourrons pas utiliser ces noms de variable dans notre programme. Ces variables ont chacune leur utilité et nous pouvons les utiliser dans nos programmes. Certaines de ces variables sont utiles pour le débogage ou pour apporter d'autres fonctionnalités. D'autres ont des usages plus simples. Voyons les deux plus simples d'entre elles:

— la variable `__FILE__` est une chaîne de caractères qui représente le nom du fichier courant;
— la variable `__LINE__` est un entier qui représente la ligne du fichier courant que l'interpréteur est en train d'exécuter.

On peut ainsi écrire ce petit script qui affiche juste le nom du fichier et la ligne à laquelle on se trouve.

```
1 puts
      "Le fichier interprété est le fichier #{__FILE__} et nous sommes actuelleme
2 puts "Nous sommes maintenant à la ligne #{__LINE__}."
3
4 puts
      "Nous avons laissé une ligne vide dans le fichier, nous sommes à la ligne #
```

Nous avons dit que les variables globales étaient à éviter et c'est vrai. En fait, les seules variables globales dont l'utilisation est normale et tout à fait conseillée sont les variables globales réservées. Tout au long du tutoriel, nous en verrons d'autres, mais nous pouvons déjà nous renseigner sur elles et sur leur utilité.

8.3. Les symboles

8.3.1. Garder le même identifiant

Il peut arriver que l'on veuille attribuer un objet unique à une variable, c'est-à-dire garder le même identifiant. C'est le but des symboles. On peut alors voir les symboles comme un nom associé à un identifiant. Donc chaque fois qu'une variable aura pour valeur ce symbole, ce sera toujours le même identifiant qui lui sera associé. Un symbole en Ruby c'est un nom précédé du caractère `:`. Donc…

```
1 symbole = :symbole
```

C'est un symbole. Vérifions alors que deux symboles identiques ont le même identifiant.

```
x = :symbole
y = :symbole
print x.object_id == y.object_id
```

Ce code affiche `True`. Toutes les variables qui auront pour valeur `:symbole` sont le même objet. Que ce soit dans une méthode ou autre part.

```
def f
  y = :s
  puts y.object_id
end

def g
  z = :s
  puts z.object_id
end

x = :s
puts x.object_id
puts :s.object_id
f
g
```

Nous obtenons bien le même identifiant partout.

Mais quel est l'intérêt de cette démarche?

Imaginons par exemple que nous devions utiliser un très grand nombre de variables avec le même contenu. En utilisant les symboles, nous faisons une économie de mémoire (un seul objet plutôt que plusieurs).

Nous pouvons également déclarer des symboles avec la syntaxe `%s`, mais conformément aux bonnes pratiques, nous allons préférer utiliser `:`. Si nous devions malgré tout utiliser `%s`, nous privilégierons son utilisation avec les parenthèses comme délimiteurs.

8.3.2. Les symboles et les chaînes de caractères

Les symboles sont surtout utilisées en lieu et place des chaînes de caractères. La question qui se pose alors est quand utiliser les symboles et quand au contraire utiliser des chaînes de caractères. La réponse est en rapport avec l'information qui est importante:

— si c'est le contenu qui est important, il faut préférer une chaîne de caractères;

— si c’est l’identité qui importe, il faut préférer un identifiant.

Prenons l’exemple d’une application qui gère les nationalités de plusieurs individus. Ce qui nous intéresse, c’est l’information de la nationalité et non pas comment cette nationalité s’écrit. En fait, on pourrait tout aussi bien écrire `français` que `France` ou encore `fr`. Par contre, le nom de l’individu sera une chaîne de caractère. On va donc plutôt écrirececi.

```
nom_1 = 'Nom'
nation_1 = :fr
nom_2 = 'Nom2'
nation_2 = :en
nom_3 = 'Nom3'
nation_3 = :it # it c’est Italie, pas informatique, on est bien
    d’accord…
```

Pour convertir un symbole en chaîne de caractères, on peut toujours utiliser la méthode `to_s`. Cependant, nous pouvons également utiliser la méthode `id2name`, plus idiomatique.

L’opération inverse (à savoir convertir une chaîne de caractères en symboles) est faite à l’aide de la méthode `intern`. Donc…

```
nom = 'nom'
nation = :fr
x = nom.intern     # x = :nom
y = nation.to_s    # y = "fr"
z = nation.id2name # z = "fr"
```

Nous avons déclaré un symbole pour chacune des nations que nous voulions représenter. Utiliser un tableau de symboles aurait été plus simple. Pour cela, nous pouvons déclarer notre tableau normalement, mais aussi utiliser la syntaxe `%i`. Ainsi, nous allons écrire `nation = %i[fr en it]`. La syntaxe `%i` est d’ailleurs à privilégier pour écrire un tableau de symboles (remarquons que là encore, puisqu’il s’agit d’un tableau, les crochets sont les délimiteurs à privilégier).

8.3.3. Les symboles et les hachages

Les symboles sont communément utilisés en tant que clés pour les hachages. Ainsi, il est courant de voir ceci.

```
hachage = { :nom => 'Nom',
            :prénom => 'Prénom',
            :âge => 18 }
```

En fait, les hachages sont généralement utilisés de cette manière et, dorénavant, c’est ce que nous allons faire. Ce choix est de plus parfaitement logique. En effet, ce qui compte pour une

clé, c'est bien son identité, et non sa valeur. Le contenu de la clé nous importe peu, pourvu qu'il s'agisse de la bonne clé. Il existe même une syntaxe plus simple pour utiliser des symboles en tant que clés de hachages.

```
hachage = { nom: 'Nom',
            prénom: 'Prénom',
            âge: 18 }
```

C'est une bonne pratique en Ruby d'utiliser des symboles comme clés de hachages et c'est aussi une bonne pratique de privilégier la dernière syntaxe que nous avons vue pour cela.

8.4. Modifier nos variables dans des méthodes

Nous avons parlé des variables globales et des symboles. Pourtant, nous n'avons toujours pas répondu à la question originelle: comment modifier nos variables dans des méthodes?

8.4.1. Utiliser le retour des méthodes

Il est possible de modifier directement une variable dans une méthode. Mais nous n'allons pas voir comment le faire et allons, pour le moment, nous contenter de renvoyer la valeur modifiée et de la récupérer. Par exemple, si l'on veut une méthode qui met une variable au carré, nous allons faire ce code.

```
def carré(x)
  return x * x
end

print 'Entrez un nombre : '
nombre = gets.chomp.to_i
puts "Nous allons calculer le carré de #{nombre}."
nombre = carré(nombre)
puts "Son carré est #{nombre}."
```

Ainsi, on détourne le problème. Plutôt que de chercher à modifier la variable dans la méthode, on renvoie sa valeur modifiée et on la récupère.

8.4.2. Retourner plusieurs variables ?

Maintenant que nous avons établi que nous allions utiliser la valeur retournée par les méthodes, une question peut nous venir à l'esprit.

Comment modifier plusieurs variables?

Un exemple simple serait une méthode dite de *swap*, c'est-à-dire une méthode qui échange la valeur de deux variables. Comment peut-on faire cette méthode?

En fait, nous allons utiliser ce que l'on appelle l'**affectation multiple**. Elle consiste, comme son nom l'indique, à faire plusieurs affectations en une seule fois. Voyons un exemple d'affectation multiple.

```
a, b = 2, 3
```

Ici, `a` vaut 2 et `b` vaut 3. Et on peut alors échanger les valeurs de `a` et de `b` directement.

```
a, b = b, a
```

Pour plus de lisibilité, nous pouvons ajouter des crochets autour des valeurs assignées. On écrit alors notre méthode de *swap*.

```
def swap(a, b)
  return [b, a]
end

a, b = [1, 2]
a, b = swap(a, b)
```

Bien sûr, notre méthode `swap` n'a pas grande utilité, puisqu'il aurait suffit d'écrire `a, b = [b, a]`. Cependant, elle illustre parfaitement le principe de retour multiple, puisque dans cette méthode, nous avons renvoyé deux valeurs.

Si les crochets nous rappellent les tableaux, ce n'est pas sans raison. En effet, nous pouvons de cette manière affecter les éléments d'un tableau à des variables.

```
tab = [1, 2, 3]
a, b, c = tab
print "a = #{a}, b = #{b} et c = #{c}."
```

En fait, c'est même plus que ça. Lorsqu'on écrit `a, b = b, a` (ou `a, b = [b, a]`) `b, a` et `[b, a]` **sont** des tableaux. Donc, pour renvoyer plusieurs valeurs, nous renvoyons un tableau.

Notons finalement que nous ne sommes pas obligés d'affecter tous les éléments du tableau à une variable. Si nous affectons moins de valeurs qu'il n'y a d'éléments dans le tableau, les valeurs

sont affectés suivant l'ordre du tableau. Si nous en affectons plus, les variables qui sont en trop vaudront `nil`. Le code qui suit illustre ce comportement.

```
a, b, c = [1, 2, 3, 4]
puts "a = #{a}, b = #{b} et c = #{c}."
a, b, c = [5, 6]
puts "a = #{a}, b = #{b} et c = #{c}."
```

Cela signifie notamment que nous pouvons faire une méthode qui retourne un tableau, et ne récupérer que les éléments qui nous intéressent. Il faudra alors faire attention à la place des éléments dans notre tableau. Ceux que l'on voudra toujours récupérer seront placés en premier, et les optionnels en dernier (ce qui rappelle fortement le fonctionnement des arguments optionnels de méthodes).

8.4.3. Copier un objet

Si au contraire on veut copier un tableau par exemple, le signe égal ne fonctionne pas puisque la variable que l'on créera de cette manière fera référence au tableau d'origine. Dès lors, si on veut copier un tableau, il nous faut créer un tableau vide et copier les éléments du premier tableau dedans. Ceci est valable pour les tableaux, les hachages, les chaînes de caractères, etc.

```
def copier_tab(tab)
  retour = []
  tab.each { |e| retour << e }
  retour
end

tab = [1, 2, 3]
copie = copier(tab)
copie[0] = 3
print tab
print copie
```

Ruby nous fournit la méthode `dup` qui nous permet de nous affranchir de tout ça.

```
def f(tab)
  tab << 2
  tab << 4
  tab[0] = 0
end

#### Ne Fonctionne pas
tab = [1, 2, 3]
```

```
copie = tab
f(copie)
print tab
print copie

#### Fonctionne

tab = [1, 2, 3]
copie = tab.dup
f(copie)
print tab
print copie
```

Ce chapitre très théorique est enfin fini. N'oublions pas que les variables globales sont à proscrire dans la plupart des cas, contrairement aux symboles qui sont très utiles et très utilisés, avec les hachages par exemple. Pour renvoyer plusieurs valeurs dans une méthode, nous renvoyons des tableaux et nous récupérons ensuite les valeurs voulues en faisant une affectation multiple.

— Les variables sont passées par référence, et une variable passée en paramètre dans une fonction n'est pas modifiée.
— Pour modifier des variables, nous allons utiliser les retours des fonctions.
— Les variables globales sont à éviter (sauf celles qui sont réservées).
— Les symboles permettent d'associer un objet unique à une variable. Ils sont souvent utilisés avec les hachages.

9. Le module Enumerable

Dans ce chapitre, nous allons aborder une nouvelle notions en Ruby : les **modules**. Nous ne nous attarderons pas trop dessus puisque ce sera un des rôles de la seconde partie.

Par contre, nous allons apprendre à nous servir d'un module en particulier : le module `Enumerable`.

9.1. Les modules

9.1.1. Les modules

Un module est une structure regroupant à la fois des variables et des méthodes. Il permet de partager du code avec d'autres structures du langage, comme d'autres modules. Cela permet de réduire le nombre de ligne de code à écrire, et de coller au principe DRY en évitant de réécrire plusieurs fois le même code. On peut par exemple imaginer un module mathématique contenant des méthodes trigonométriques et divers outils mathématiques.

Un module se construit de cette façon.

```ruby
module Multiplication
  MAX = 10

  def Multiplication.table(x)
    puts "On a demandé la table de #{x}."
  end
end
```

Un module commence par le mot-clé `module` et se finit par le mot-clé `end`. Après le mot-clé `module`, on écrit le nom du module (ici `Multiplication`). Ce nom doit commencer par une majuscule sinon nous obtiendrons l'erreur «*class/module name must be CONSTANT*». De plus, il est conseillé d'écrire le nom des modules en «CamelCase», c'est-à-dire en mettant en majuscule la première lettre de chaque mot (`ExempleDeModule` plutôt que `Exemple_De_Module`). Ensuite, on va à la ligne et on écrit comme dans un fichier normal.

Ici, nous avons défini la constante `MAX` et la méthode `table`. Notons que pour définir la méthode `table`, nous avons écrit `Multiplication.table`. Cela s'explique par le fait qu'il faut que Ruby sache que nous sommes en train de définir une méthode du module `Multiplication`. Finalement, pour écrire une méthode dans un module, nous écrirons `def NomModule.nom_méthode`.

Nous avons un niveau d'indentation dans notre module. C'est une bonne pratique qui facilite la relecture.

Le nom du module peut être remplacé par le mot-clé `self`, qui signifie «soi», dans la définition de ses méthodes. Ainsi, nous pouvons écrire ceci.

```
def self.table(x)
  puts "On a demandé la table de #{x}."
end
```

Dans une méthode d'un module, nous pouvons faire appel à d'autres méthodes du module et aux constantes du module. Pour utiliser la constante, il suffit d'écrire son nom et pour faire appel à une méthode, on écrit `self.nom_méthode`.

```
module Multiplication
  MAX = 10

  def self.table(x)
    puts "On a demandé la table de #{x}."
  end

  def self.présente
    puts "La constante MAX de ce module vaut #{MAX}."
    puts 'Essayons Multiplication.table 3.'
    self.table(3)
  end
end
```

Utiliser le mot-clé `self` plutôt que le nom du module est une bonne habitude. Elle permet par exemple de pouvoir changer le nom du module sans se faire de souci.

9.1.2. Utiliser un module

Utiliser un module n'est pas très compliqué. Pour utiliser une méthode d'un module, on écrit `NomModule.nom_méthode`. Par exemple, pour utiliser la méthode `table` du module `Multiplication` défini précédemment, nous allons utiliser ce code.

```
print Mutltiplication.table(2)
```

Bien sûr, le module doit avoir déjà été défini précédemment.

> ! Nous ne pouvons pas utiliser le mot-clé `self` ici car Ruby ne saurait alors pas à quel module il se rapporte. Le nom du module est donc obligatoire.

Nous pouvons aussi accéder aux constantes du module. Pour cela, nous devons utiliser la syntaxe `NomModule::NOM_CONSTANTE`. Ainsi, pour accéder à la constante définie dans notre code précédent, nous allons utiliser ce code.

```
1 print Multiplication::MAX
```

Notons que cette syntaxe fonctionne également pour utiliser les méthodes du module. Nous pouvons donc écrire `NomModule::nom_méthode`. (en revanche, nous ne pouvons pas écrire `NomModule.NOM_CONSTANTE` sous peine d'obtenir une erreur). Cependant, pour bien identifier les appels aux méthodes et les utilisations des constantes, nous allons privilégier l'écriture `NomModule.nom_méthode`.

```
1  module Multiplication
2    MAX = 10
3
4    def self.table(x)
5      puts "On a demandé la table de #{x}." if x <= MAX
6    end
7  end
8
9  puts "le maximum est #{Multiplication::MAX}"
10 Multiplication.table(3)                          # Bonne écriture.
11 Multiplication::table(3)                         # Écriture
       déconseillée.
```

9.1.3. Mettre un module dans un fichier séparé

Nous avons dit que le but d'un module était de ne pas avoir à réécrire plusieurs fois le même code et de pouvoir utiliser le même module dans plusieurs programmes. Cependant, pour le moment, nous avons toujours écrit le module dans le même fichier que notre programme, ce qui signifie qu'il faudra le recopier chaque fois que nous voudrons l'utiliser. Or, c'est justement ce que nous voulons éviter.

En fait, ce qu'il nous faudrait, c'est pouvoir écrire le module dans un autre fichier. Ainsi, on pourrait utiliser ce même fichier dans plusieurs projets.

Ceci est possible grâce à la méthode `require_relative` qui nous permet d'indiquer qu'un fichier Ruby requiert un autre fichier Ruby. En l'utilisant, nous pourrions écrire notre module `Multiplication` dans un fichier `multiplication.rb`.

```ruby
module Multiplication
  MAX = 10

  def self.table(x)
    puts "On a demandé la table de #{x}." if x <= MAX
  end
end
```

Ensuite, dans notre fichier principal, disons `main.rb`, nous allons indiquer qu'il a besoin de `multiplication.rb`.

```ruby
require_relative 'multiplication.rb'

puts "le maximum est #{Multiplication::MAX}"
Multiplication.table(3)
```

Notons qu'avec ce code, le fichier `multiplication.rb` doit être placé dans le même dossier. En effet, le paramètre de `require_relative` est le chemin relatif du fichier requis. Si nous voulions placer notre fichier `multiplication.rb` dans un dossier `module`, il faudrait alors écrire `require_relative 'module/multiplication.rb'`. Notons de plus que l'extension du fichier n'est pas obligatoire et qu'il est parfaitement possible d'écrire `require_relative 'multiplication'`.

9.1.3.1. Conventions de nommage pour les dossiers et les fichiers

De même que pour les noms de variables et de modules, il y a des conventions de nommage pour les noms de fichiers et de dossiers. Elles ne sont pas compliquées et les suivre ne devrait pas nous poser de problèmes. Les voici:

— il est conseillé d'écrire les noms des fichiers en «snake_case»;
— il est conseillé d'écrire les noms des dossiers en «snake_case».

Il est également conseillé d'avoir un seul module par fichier et de nommer ce fichier par le nom du module (en «snake_case» pour rester en cohérence avec la règle sur les noms des fichiers). Ainsi, nous avons le module `Multiplication` dans le fichier `multiplication.rb`.

9.2. Le module Enumerable

Maintenant, nous savons comment créer des modules. Cependant, la création de module n'est pas l'objet principal de ce chapitre. L'objet principal de ce chapitre est un module en particulier, le module `Enumerable`.

Comme son nom le suggère, ce module semble permettre *d'énumérer*. Mais il n'en n'est rien. En fait, ce module est destiné aux éléments qui savent déjà comment énumérer leur contenu. C'est

par exemple le cas pour les tableaux, les hachages ou encore les intervalles que nous savons parcourir avec la méthode `each`.

```
1 [1, 2, 3, 4, 5].each do |i|
2   # …
3 end
```

Et c'est justement cette méthode qu'utilise le module `Enumerable`. Nous allons voir que cette seule méthode permet de faire des tas de choses! Grâce à lui, nous pouvons faire des opérations de recherche, de tri, de comptage, etc. Par exemple, il nous permet de connaître les éléments d'un tableau qui satisfont une condition.

Dans la suite, nous traiterons uniquement des tableaux. Mais ce que nous verrons sera aussi valable pour les hachages ou encore pour les intervalles. Nous conseillons d'utiliser IRB pour faire les tests sur `Enumerable`.

9.2.1. Vérifier des conditions sur un tableau

9.2.1.1. La méthode `all?`

Cette méthode permet de vérifier si tous les éléments d'un tableau correspondent à un critère donné. Elle prend en paramètre un bloc d'instructions, la dernière instruction devant renvoyer un booléen. Si le booléen renvoyé est évalué à `true` pour chaque élément du tableau, alors la méthode `all` renvoie `true`. Ici, nous testons la parité des éléments d'un tableau grâce à la méthode `even?` qui s'applique à un entier et renvoie `true` s'il est pair et `false` s'il est impair (notons que la méthode `odd?` fait le contraire).

```
1 enumerable = [1, 2, 3]
2 is_even = enumerable.all? do |e|
3             print e
4             e.even?
5 end
6 print is_even  # => false car 1 est impair.
```

Ici, bien entendu, `is_even` vaut `false`. Nous pouvons également remarquer grâce au `print` que tous les éléments du tableau n'ont pas été testés; puisque le premier élément, 1, n'est pas pair, ce n'est pas la peine de tester les autres.

Si nous ne donnons aucun bloc à `all?`, elle retourne `true` uniquement si tous les éléments du tableau sont évalués à `true`, autrement dit si le tableau ne contient ni `false`, ni `nil`.

```
[1, nil].all?       # => false.
[5, 'salut'].all?   # => true.
```

9.2.1.2. La méthode none?

Cette méthode est la complémentaire de `all?`. Elle renvoie `true` si aucun élément ne satisfait le critère donné. Si aucun bloc ne lui est donné, elle retourne vrai uniquement si tous les éléments du tableaux sont évalués à `false`, donc si le tableau ne contient que `false` ou `nil`.

```
enumerable = [1, 2, 3]
enumerable.none? { |e| e.even? }  # => false.
enumerable.none? { |e| e > 5 }    # => true.

[false, nil].none?                # => true.
[true, nil].none?                 # => false.
```

9.2.1.3. La méthode any?

Cette méthode ressemble beaucoup à `all?`, à la différence que `any?` retourne `true` si **au moins un élément correspond au critère**. Autrement dit, `any?` retourne `false` si aucun élément ne correspond au critère.

```
enumerable = [1, 2, 3]
enumerable.any? { |e| e.even? }  # => true, car 2 est pair.
```

9.2.1.4. La méthode one?

Cette méthode est le complémentaire de `any?`. Elle retourne `true` si un, et seulement un, élément vérifie le critère déterminé par le bloc qui lui est donné. Si on ne lui donne pas de bloc, alors elle retourne `vrai` si un seul élément est évalué à `true`.

```
enumerable = [1, 2, 3]
enumerable.one? { |e| e == 1 }   # => true, il y a un seul 1 dans
    le tableau.

enumerable2 = [1, 1, true]
enumerable2.one?                 # => false, tous les éléments sont
    évalués à true.
```

```
6 enumerable2.one? { |e| e == 1 }  # => false, il y a deux 1 dans le
      tableau.
7
8 [false, nil, true].one?          # => true, seul true est évalué à
      true.
```

9.2.1.5. La méthode `include?`

Cette méthode retourne `true` si la valeur passée en paramètre est présente dans le tableau, c'est-à-dire si le tableau contient la valeur passée en paramètre.

```
1 enumerable = [1, 2, 3]
2 enumerable.include?(2)  # => true
3 enumerable.include?(5)  # => false
```

9.2.2. Récupérer des éléments du tableau

9.2.2.1. La méthode `find`

Cette méthode permet de chercher le premier élément satisfaisant un critère donné.

```
1 enumerable = [1, 2, 3]
2 n = enumerable.find { |e| e.odd? }  # => 1
```

9.2.2.2. La méthode `select`

Cette fois ci, cette méthode retourne tous les éléments satisfaisant un critère.

```
1 enumerable = [1, 2, 3]
2 enumerable.select { |e| e.odd? }  # => [1, 3]
```

9.2.2.3. La méthode `reject`

Cette méthode retourne la liste des éléments ne satisfaisant pas un critère donné. Elle fait le contraire de ce que fait `select`.

```
enumerable = [1, 2, 3]
enumerable.reject { |e| e.even? }  # => [1, 3]
```

9.2.2.4. La méthode `map`

Cette méthode permet d'effectuer un traitement sur tous les éléments d'un tableau, et de retourner tous les résultats dans un tableau. Nous lui donnons un bloc, et elle va créer un tableau contenant ce qu'a renvoyé le bloc pour chaque élément. Avec le code qui suit, nous allons créer un tableau contenant les éléments du premier tableau incrémentés. C'est simple et très pratique.

```
enumerable = [1, 2, 3]
enumerable.map { |e| e + 1 }  # => [2, 3, 4].
```

9.2.3. Autres méthodes utiles

9.2.3.1. La méthode `reverse_each`

Nous avons vu la méthode `each` pour parcourir un tableau. La méthode `reverse_each` parcourt le tableau… à l'envers. Avec le code qui suit, on affiche les éléments du tableau dans l'ordre inverse.

```
enumerable = [1, 2, 3]
enumerable.reverse_each { |e| print "#{e} " }
```

> **!** Cette méthode crée un tableau temporaire contenant les éléments du tableau initial, mais dans le sens inverse.

Cette méthode peut donc être assez lourde sur de gros tableaux. Elle illustre bien la façon dont les méthodes du module `Enumerable` fonctionnent: elles ne font appel qu'à `each`, or ce dernier renvoie les éléments dans le bon ordre, c'est pour cela qu'un tableau temporaire inversé est nécessaire.

9.2.3.2. La méthode count

La méthode `count` fait partie des méthodes les plus simples que nous verrons dans ce chapitre. Elle retourne le nombre d'éléments du tableau satisfaisant un critère donné. Si on ne lui donne pas de bloc, elle retourne le nombre d'éléments du tableau (pour obtenir la taille du tableau, nous utiliserons plutôt la méthode `size`). On peut également lui donner un élément `x` en argument, auquel cas elle comptera le nombre de `x` du tableau.

```
enumerable = [1, 2, 3, 4]
enumerable.count                  # => 4, il y a 4 éléments.
enumerable.count { |e| e.even?}   # => 2, car 2 éléments sont pairs.
enumerable.count(3)               # => 1, il y a un seul 3.
```

9.2.3.3. La méthode first

Cette méthode prend en paramètre facultatif un entier `k` et retourne les `k` premiers éléments du tableau sous forme d'un tableau. Si aucun paramètre ne lui est donné, elle retourne le premier élément du tableau.

```
enumerable = [1, 2, 3]
enumerable.first     # => 1
enumerable.first(2)  # => [1, 2]
```

9.2.3.4. Les méthodes min, max et minmax

Ces méthodes retournent respectivement le maximum d'un tableau, son minimum, et un tableau contenant le minimum et le maximum.

```
enumerable = [1, 2, 3]
enumerable.max     # => 3
enumerable.min     # => 1
enumerable.minmax  # => [1, 3]
```

On peut leur donner un bloc, dont la dernière instruction doit renvoyer un booléen, pour indiquer comment comparer deux éléments. Par exemple, nous pouvons chercher le mot le plus long avec le code qui suit (la méthode `size` permet d'obtenir la longueur d'une chaîne de caractères).

```
enumerable = %w(un deux trois quatre cinq six sept huit neuf dix)
enumerable.max { |x, y| x.size <=> y.size }  # => "quatre"
```

`<=>` est une méthode qui compare deux objets. `a <=> b` retourne `-1` si `a < b`, `1` si `a > b` et `0` sinon. Ici, nous l'utilisons pour comparer la longueur de deux mots.

9.2.3.5. Les méthodes `min_by`, `max_by` et `minmax_by`

Les méthodes `min_by` et consort permettent d'obtenir les minimums et maximums suivant une propriété que l'on indique dans un bloc. Par exemple, pour obtenir le plus grand mot, nous pourrions écrire ce code qui se lit littéralement «donner le mot `x` dont `x.size` est le plus grand».

```
enumerable.max_by { |x| x.size }
```

9.2.3.6. La méthode `sort`

Cette méthode trie simplement un tableau.

```
enumerable = [2, 3, 1, 4]
enumerable.sort  # => [1, 2, 3, 4]
```

En fait, nous pouvons l'utiliser avec un bloc de la même manière que la méthode `min`. Pour trier par ordre décroissant, on pourrait par exemple écrire le code suivant en utilisant la méthode `<=>`.

```
enumerable.sort { |a, b| b <=> a }
```

9.2.3.7. La méthode `sort_by`

La méthode `sort_by` est à la méthode `sort` ce que `min_by` est à `min`; elle trie un tableau selon quelque chose (généralement les attributs de ses éléments). Par exemple, nous pouvons trier un tableau de chaînes de caractères suivant la taille des chaînes.

```
enumerable = %w(poire abricot cerise)
enumerable.sort_by { |e| e.size }  # => ["poire", "cerise",
    "abricot"]
```

<–COMMENT Mais on peut trier selon n'importe quoi. Par exemple, dans le code qui suit, nous trions le tableau en utilisant une fonction `f` complètement fantaisiste.

```
def f(a)
  return a - (a - 4) * (a - 4)
end

enumerable = [1, 2, 3, 4, 5, 6, 7, 8, 9]
enumerable.sort_by { |a| f(a) }
```

COMMENT–>

9.2.3.8. La méthode `to_a`

Cette méthode retourne un `Enumerable` sous forme de tableau (`to_a` signifie *to array* soit «vers tableau»).

```
[1, 2, 3].to_a       # => [1, 2, 3]
(1..3).to_a          # => [1, 2, 3]
{ a: 1, b: 2}.to_a   # => [[:a, 1], [:b, 2]]
```

Déclarer un tableau d'entiers qui va de 0 à 100 est beaucoup trop long. À la place, nous pouvons écrire `a = (1.100).to_a`.

9.2.3.9. La méthode `reduce`

Cette méthode permet de faire une opération dite de **réduction**. On lui donne un bloc dans lequel on va considérer deux variables. La première servira d'accumulateur et la seconde sert à parcourir le tableau. La variable qui sert d'accumulateur prend à chaque fois la valeur retournée par le bloc et `reduce` retourne la valeur finale de l'accumulateur. Nous comprendrons mieux ce qu'elle fait avec des exemples.

Voici un code pour calculer une somme.

```
enumerable = [1, 2, 3]
enumerable.reduce { |a, e| a + e }  # => 6 (1 + 2 + 3).
```

Ici, `a` sert d'accumulateur et `e` parcourt `enumerable`. Chaque élément `e` de `enumerable` est ajoutée à `a`. Le code pourrait s'écrire ainsi en utilisant `each`.

```
a = 0
enumerable.each { |e| a = a + e }
```

La méthode `reduce` prend en paramètre facultatif la valeur initiale de l'accumulateur (qui est 0 par défaut). On peut alors calculer le produit des éléments d'un tableau en initialisant l'accumulateur à 1.

```
enumerable.reduce(1) { |a, e| e * a }  # => 6
```

La méthode `reduce` est vraiment très utile. On pourrait l'utiliser pour trouver le maximum d'un tableau de nombres positifs ou encore pour trouver le plus grand mot d'un tableau (bien sûr, il vaut mieux utiliser les méthodes `max` et `max_by` pour cela).

Notons que pour faire la somme des éléments d'un tableau nous pouvons utiliser directement la méthode `sum`.

Certaines méthodes parmi celles que nous avons vues ont des alias (un alias est une méthode équivalente). Ainsi:

— `detect` est un alias de `find`;
— `find_all` est un alias de `select`;
— `collect` est un alias de `map`;
— `inject` est un alias de `reduce`.

Nous allons préférer utiliser les premières méthodes que nous avons vues plutôt que leurs alias.

De plus, nous avons pu voir dans ce chapitre plusieurs méthodes se terminant par le symbole `?`. Nous pouvons remarquer qu'il s'agit de méthodes retournant un booléen. En Ruby, par convention, les noms des méthodes qui renvoient un booléen, et elles seules, se terminent par un point d'interrogation. C'est une bonne pratique que nous allons nous aussi adopter.

9.3. Exercices

En guise d'exercice, nous allons créer un petit module dans lequel nous regrouperons plusieurs méthodes. Ces méthodes feront des actions simples pour lesquelles nous devrons utiliser des méthodes du module `Enumerable`. Nous allons faire les méthodes suivantes.

— `tri_par_distance` qui prend en paramètre obligatoire un tableau et en paramètre facultatif un nombre `origine`. Elle trie les éléments de ce tableau par leur distance à l'origine. La valeur par défaut de `origine` est 0. Ainsi, `tri_par_distance([1, 4, 2, 3], 2.3)` doit renvoyer `[2, 3, 1, 4]`.
— `tri_par_occurrence` qui prend en paramètres un tableau de nombres et un chiffre. Elle trie les éléments de ce tableau par le nombre de fois que le chiffre passé en paramètre apparaît dans chaque nombre du tableau. Par exemple, après appel de `tri_par_occurrence(tab, 3)` où `tab = [33, 876, 32, 3343]`, on doit avoir `tab = [876, 32, 33, 3343]`.
— `intersection` qui prend en paramètres deux tableaux et renvoie un tableau contenant tous les éléments du premier tableau qui sont également présents dans le second tableau.

— `présent_dans_intervalle` qui prend en paramètres un tableau et deux entiers `min` et `max` tels que `min` est inférieur à `max` et qui retourne un tableau contenant tous les éléments du tableau compris entre `min` et `max`.
— `inférieur?` qui prend en paramètres deux tableaux et renvoie `true` si tous les éléments du premier tableau sont inférieurs à ceux du second tableau et `false` sinon.
— `tous_multiples?` qui prend en paramètres un tableau et un entier et vérifie que tous les nombres du tableaux sont des multiples de l'entier.

Correction.

Voici la correction. Nous allons la mettre dans un fichier `exo.rb`.

Contenu masqué n°23

Nous pouvons trouver d'autres exercices à ce propos sur ce projet. La plupart des exercices que nous avons résolus dans ce chapitre viennent de ce projet.

Le module `Enumerable` est extrêmement utile et permet de faire plein de choses. Toutes les méthodes n'ont pas été présentées et il nous en reste beaucoup à voir. Pour cela, nous pouvons aller voir la documentation officielle.

— Les modules permettent d'éviter de réécrire plusieurs fois la même chose et de réunir du code par thématiques.
— Le module `Enumerable` permet de faire des opérations sur les variables qui ont une méthode `each`. Il permet de les trier, de faire des recherches, des opérations, du filtrage, etc.

Contenu masqué

Contenu masqué n°23

```ruby
module Exo

  # Pour faire tri_par_distance, nous aurons besoin d'une méthode qui calcule la
  # valeur absolue d'un nombre. La méthode abs qui s'applique à un nombre existe
  # déjà pour faire ce travail. Ensuite, il nous suffit de trier le tableau par
  # valeur absolue des éléments moins l'origine (ce qui correspond à la distance).
  def self.tri_par_distance(tab, origine = 0)
    tab.sort_by { |e| (e - origine).abs }
  end
```

```
10
11    # Nous allons être malin. Nous transformons le nombre en chaîne
        de caractère,
12    # puis la chaîne de caractère en tableau de caractère, il nous
        suffit alors
13    # de compter combien de fois le caractère associé au chiffre
        apparaît dans
14    # le tableau.
15    def self.tri_par_occurrence(tab, nb)
16      tab.sort_by { |e| e.to_s.chars.count(nb.to_s) }
17    end
18
19    # Pas de difficulté pour écrire intersection. On sélectionne tous
        les éléments de
20    # tab1 qui sont présent dans tab2.
21    def self.intersection(tab1, tab2)
22      tab1.select { |e| tab2.include?(e) }
23    end
24
25    # Pour présent_dans_intervalle, on sélectionne tous les éléments
        du tableau qui
26    # sont inférieurs à max et supérieurs à min.
27    def self.présent_dans_intervalle(tab, min, max)
28      tab.select { |e| e >= min && e <= max}
29    end
30
31    # Pour inférieur, il suffit de vérifier que tous les éléments de
        tab1 sont inférieurs
32    # au minimum de tab2.
33    def self.inférieur?(tab1, tab2)
34      min = tab2.min
35      tab1.all? { |e| e <= min }
36    end
37
38    # On vérifie que le reste de la division euclidienne de tous les
        éléments du tableau
39    # par le nombre passé en paramètre vaut 0.
40    def self.tous_multiples?(tab, nb)
41      tab.all? { |e| e % nb == 0 }
42    end
43
44  end
```

Retourner au texte.

10. Écrire le code dans des fichiers

Ce chapitre annexe a pour but de nous apprendre à exécuter du code écrit dans un fichier. En effet, dès que l'on commence à écrire des programmes un peu long, ou que l'on veut le distribuer, il nous est indispensable d'écrire notre code dans un fichier.

10.1. Les éditeurs de texte

10.1.1. Pourquoi choisir un éditeur de texte ?

Pour exécuter un programme Ruby depuis un fichier, il suffit le code et de l'enregistrer dans un fichier au format `rb`. Pour exécuter le code, il faut ouvrir ce fichier avec l'interpréteur Ruby (qui est installé en même temps que Ruby). L'interpréteur Ruby fait son travail et interprète le code du fichier. Normalement, les fichiers au format `rb` se lancent avec l'interpréteur Ruby par défaut.

Il nous reste juste à voir comment écrire le code dans un fichier. Pour cela, nous allons utiliser un éditeur de texte. Un éditeur de texte est un programme qui sert à… Éditer des textes. Il n'est pas à confondre avec un traitement de texte, qui lui, permet également de faire de la mise en forme et de la mise en page. Le logiciel Bloc-notes de Windows est par exemple un éditeur de texte.

Nous ne pouvons pas utiliser de traitement de texte pour écrire notre code parce qu'ils n'enregistrent pas le **texte brut** que nous écrivons, mais rajoutent des informations sur la mise en page par exemple, or l'interpréteur n'a besoin que du texte brut.

Il nous faut alors choisir un éditeur de texte. Et ils sont nombreux. Très nombreux. En fait, il y en a pour tous les goûts. Certains sont simples et sobres. D'autres sont très complexes et personnalisables. Certains ne s'utilisent qu'à la souris ou qu'au clavier.

10.1.2. Des éditeurs de texte adaptés à la programmation

Cependant, si nous pouvons choisir n'importe quel éditeur de texte pour coder (la seule condition étant que nous puissions écrire du code brut), certains éditeurs sont plus adaptés à la programmation que d'autre. Un simple bloc-note comme celui de Windows par exemple, n'est pas adapté à la programmation. En effet, il ne permet pas de mettre en place un environnement agréable pour programmer. Comparons par exemple les deux codes qui suivent.

variable = 43
chaine = "#{variable} *2 = #{variable* 2}"
"43 + 2 = #{43 + 2}"

```
variable = 43
chaine = "#{variable} * 2 = #{variable * 2}"
"43 + 2 = #{43 + 2}"
```

Le premier code est beaucoup moins lisible. En fait, un point important pour un éditeur de code est le respect de la mise en forme que' l'on donne à notre code et en particulier de l'indentation . Notre éditeur doit alors bien aérer les textes et si possible adopter une police à chasse fixe (et c'est aussi notre devoir d'écrire un code clair et compréhensible).

Un autre point qui est important est la coloration syntaxique. Comparons ces deux codes.

```
variable = 43
chaine = "#{variable} * 2 = #{variable * 2}"
"43 + 2 = #{43 + 2}"
```

```
variable = 43
chaine = "#{variable} * 2 = #{variable * 2}"
"43 + 2 = #{43 + 2}"
```

Là encore, les deux codes sont strictement identiques, et pourtant, le premier code est plus agréable à lire puisqu'il met en valeur certains éléments-clés de notre code (ici, le code interpolé).

> Les éditeurs de texte adaptés à la programmation associent souvent des extensions de fichiers à un langage et utilisent automatiquement la coloration syntaxique approprié. Les fichiers à l'extension `.rb` sont associés à Ruby; dès lors, nous enregistrons nos fichiers en utilisant cette extension.

10.1.3. Exécuter le code

Pour exécuter le code, il suffit de suivre ces trois étapes:

- ouvrir une console;
- se rendre dans le dossier de notre fichier;
- exécuter la commande `ruby nom_fichier`.

Pour se déplacer dans les dossiers avec la console, nous utilisons la commande `cd` (pour *change directory*). Supposons par exemple que nous soyons dans le dossier `/home/user/` au démarrage de la console et que notre fichier qui s'appelle `test.rb` soit dans le dossier `/home/user/programmation/ruby`. Il nous faudra alors taper `cd programmation/ruby`, puis `ruby test.rb`. On peut aussi le faire en plus d'étapes en tapant ce qui suit.

```
cd programmation
cd ruby
ruby test.rb
```

La commande `cd` est une commande bash. Elle n'est donc pas à utiliser dans IRB, mais directement dans la console.

Faisons un test. En utilisant un éditeur de texte (le Bloc Note de Windows par exemple), écrivons `puts "Test."` dans un fichier. Enregistrons-le sous le nom `test.rb`. Ouvrons une console, et rendons-nous dans le dossier où le fichier a été enregistré. Après avoir utilisé la commande `ruby test.rb`, le programme devrait s'exécuter (et donc afficher le mot «Test»).

10.2. Windows — Notepad++

Sous Windows, l'éditeur de texte que nous allons présenter est Notepad++. Il s'agit d'un logiciel gratuit sous licence GPL . Il intègre la coloration syntaxique pour de nombreux langages et dispose de nombreuses extensions.

10.2.1. Installer Notepad++

Avant d'installer Notepad++, il nous faut le télécharger. Le mieux pour cela est de se rendre sur la page de téléchargement de Notepad++. Il suffit alors de cliquer sur le bouton de téléchargement (le gros bouton vert «Download»). Une fois ceci fait, nous exécutons le programme téléchargé pour installer Notepad++.

Notepad++ existe aussi en version portable. L'utilisation de la version portable d'un logiciel ne nécessite pas d'installation. Nous pouvons transporter cette version sur une clé USB, donc l'utiliser sur n'importe quel ordinateur avec nos paramètres.

Pour obtenir la version portable de Notepad++, retournons sur la page de téléchargement du logiciel. Nous pourrONS y trouver deux liens qui correspondent aux versions portables: «Notepad++ zip package» et «Notepad++ 7z package». Au moment où nous écrivons, ces deux liens sont sous le bouton «Download».

10.2.2. Exécuter le code

Pour lancer le code écrit, il suffit de l'enregistrer avec l'extension .rb puis de double-cliquer dessus.

10.2.2.1. NppExec

Quoi? On vient de me dire dans l'oreillette que c'est quand même fatigant de réduire Notepad++ puis double-cliquer sur votre fichier. Dans ce cas, voyons comment lancer le fichier avec un raccourci clavier. Pour cela, nous allons utiliser une extension de Notepad++ appelé NppExec.

Pour commencer, installons NppExec. Cliquez sur «Compléments» → «Plugin Manager» → «Show Plugin Manager» → «Available». Là, cherchez NppExec, cochez-le et cliquez sur «Install».

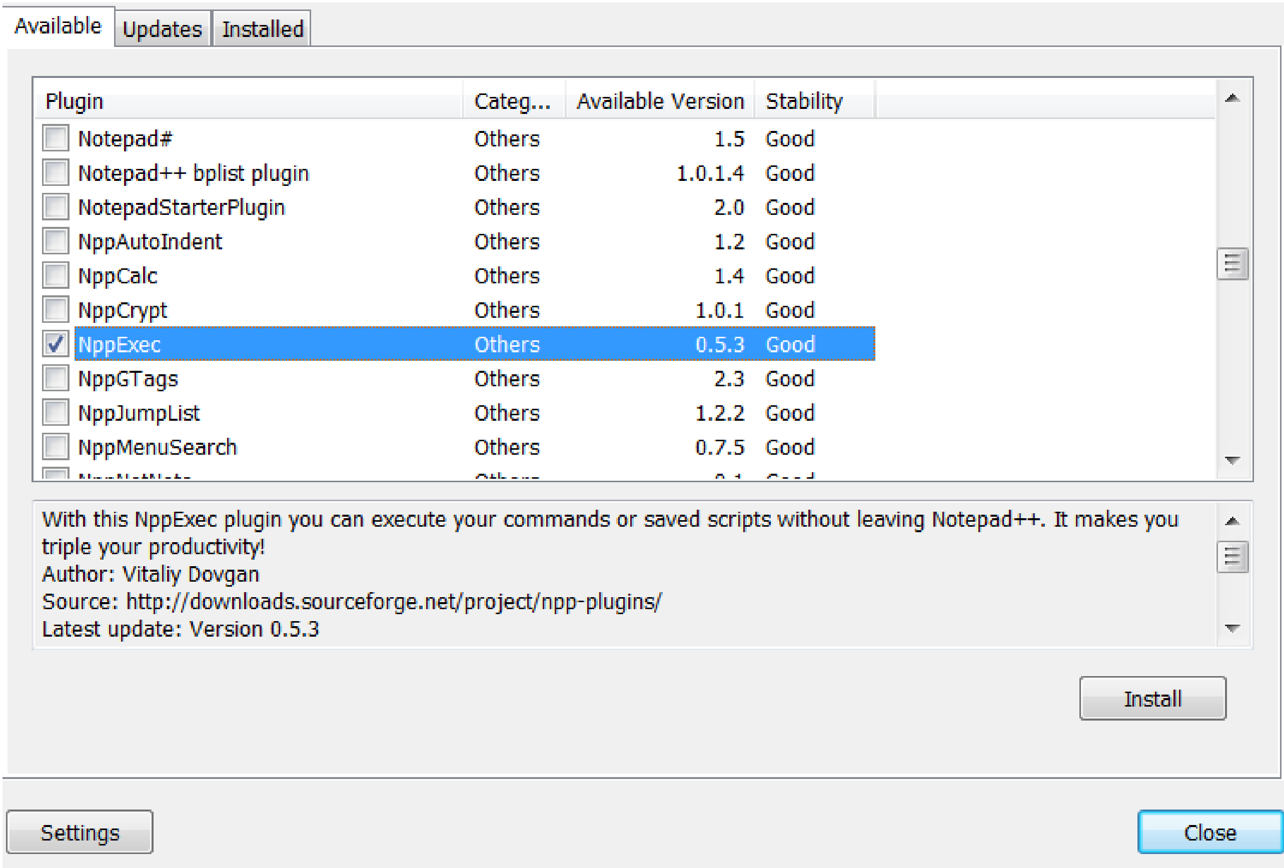


Figure : Installer NppExec.

Maintenant que vous avez installé le plugin, il faut créer un script. Pour cela, appuyez sur F6 (ou Fn + F6) et tapez ceci.

```
NPP_CONSOLE OFF                                 // Ne pas afficher la console NppExec.
NPP_SAVE                                        // Sauvegarder le fichier courant.
CD $(CURRENT_DIRECTORY)                         // Se placer dans le dossier du fichier courant.
NPP_RUN cmd /c $(FULL_CURRENT_PATH) && pause    // Exécuter le fichier puis mettre en pause la console.
```

Cliquez sur «Save» et choisissez un nom pour votre script (pourquoi pas «Ruby»).

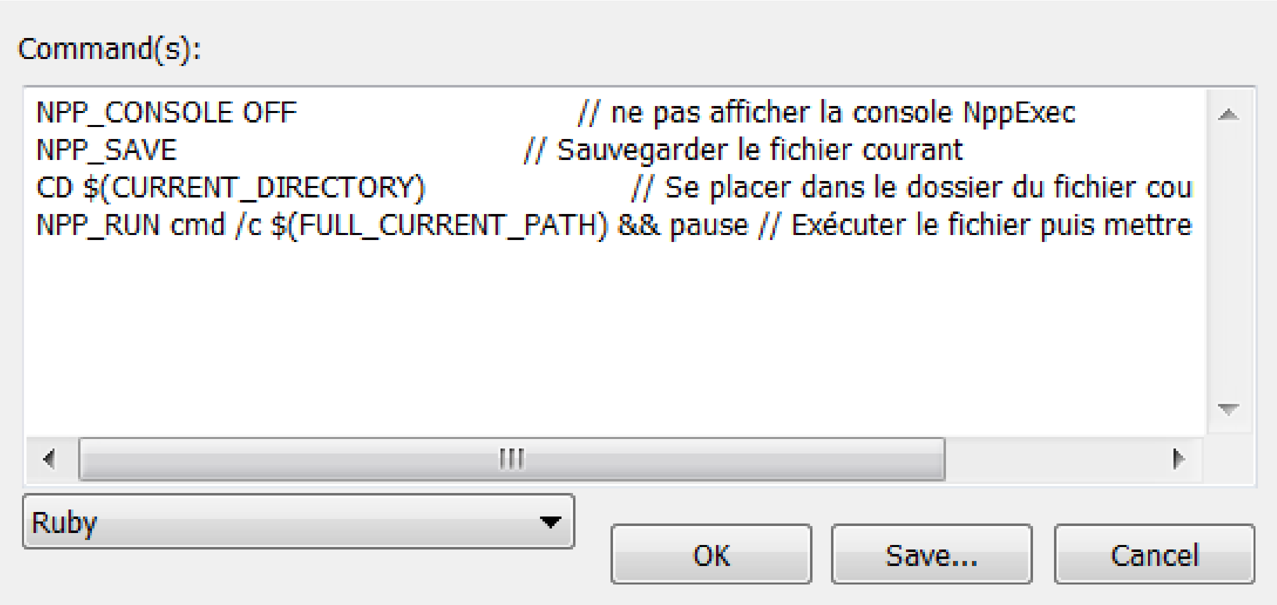


Figure : Création du script.

Cliquez sur «Compléments» → «NppExec» → «Advanced Options». Choisissez le script que vous avez créé dans «Associated script» et cliquez sur «Add/Modify».

Menu items *
Place to the Macros submenu
Ruby :: Ruby
Move up
Move down
* Assign shortcut keys using Settings: Shortcut Mapper -> Plugin commands
Menu item
Item name:
Ruby
Associated script:
Ruby
Add/Modify
Delete
Advanced options
Execute this script when Notepad++ starts:
Execute this script when Notepad++ exits:
[Options]
HotKey F6
ToolbarBtn Console
[Console]
Visible (on start) Auto
ShowHelp (on start) No
SaveCmdHistory Yes
CommentDelimiter //
TextColorNormal (RR GG BB) 00 00 00
TextColorError (RR GG BB) A0 10 10
TextColorMessage (RR GG BB) 20 80 20
BackgroundColor (RR GG BB) 0
OK
Cancel

Figure : Ajout du script.

Il ne nous reste plus qu'à faire un raccourci clavier pour ce script : cliquez sur «Paramétrage» → «Raccourcis clavier...» → «Plugin commands», cherchez le nom du script, double-cliquez dessus (ou cliquez sur «Modify») et choisissez un raccourci : F9 est souvent utilisé, mais la seule règle à respecter est de ne pas utiliser des touches déjà utilisées. Ctrl + Alt + R peut être une bonne idée.

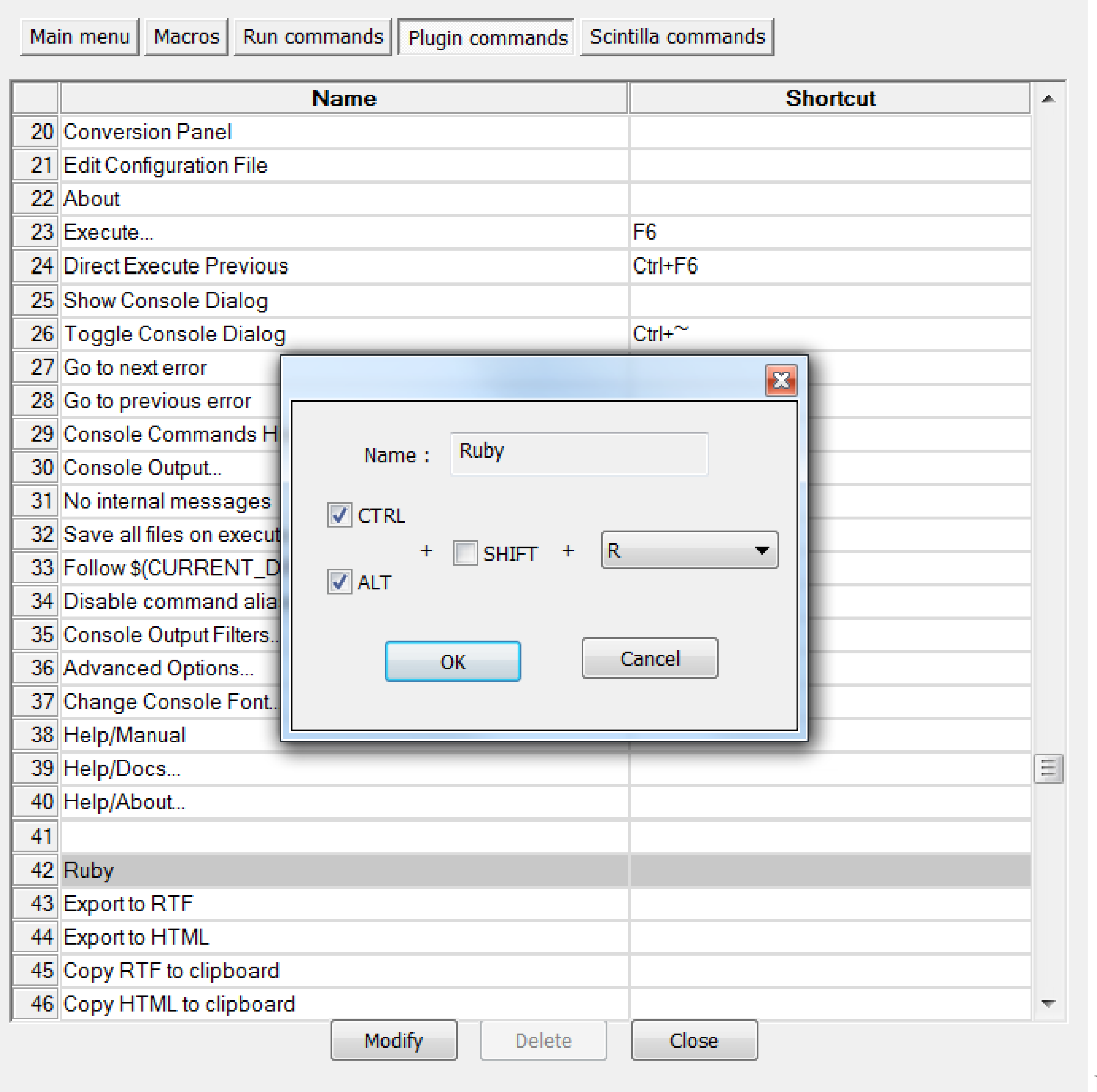


Figure : Ajout du raccourci.

Ça y est, vous avez votre super raccourci (notez que vous pouvez utiliser le même raccourci pour Python par exemple).

10.3. Linux — gedit

Si vous êtes sous Linux, vous disposez sûrement déjà d'un éditeur de texte avec coloration syntaxique et support du langage Ruby. S'il s'agit d'un éditeur que **vous** avez choisi, que vous avez l'habitude d'utiliser et qui vous convient, ne le changez pas. Sinon, vous devriez vous intéresser au large panel de choix qui est disponible sous Linux. Il y en a vraiment pour tous les goûts. Ici, nous allons vous présenter gedit, un éditeur de texte très simple, qui propose juste le peu de fonctionnalités dont nous avons effectivement besoin. gedit utilise GTK+.

10.3.1. Installer gedit

Sur la plupart des distributions, gedit est installé par défaut. Il est par exemple fourni avec l'environnement de bureau GNOME. S'il n'est pas installé, il suffit d'utiliser votre gestionnaire de paquets pour l'installer. Par exemple…

```
sudo apt-get install gedit
```

… ou…

```
pacman gedit
```

Vous savez quoi faire pour l'installer. Nous vous laissons faire.

N'oubliez pas d'installer GTK+ pour obtenir gedit avec la meilleure interface graphique.

Une fois gedit installé, lancez-le et habituez-vous à l'interface. La coloration est choisie en fonction de l'extension du fichier ouvert (la coloration pour le Ruby est donc associée aux fichiers `.rb`), mais vous pouvez bien sûr choisir la coloration dans les paramètres de gedit.

10.3.2. Exécuter le code

La façon la plus simple d'exécuter le script Ruby est d'utiliser le terminal : on ouvre un terminal, on se rend dans le dossier du script (à l'aide la commende `cd`) et on utilise la commande `ruby nom_du_script.rb`. Le fichier sera alors interprété par l'interpréteur Ruby, c'est-à-dire qu'il sera exécuté.

Vous pouvez bien sûr essayer d'autres éditeurs de texte. Geany, par exemple, offre plus de fonctionnalités, tout en restant assez simple et ergonomique, mais choisissez avant tout un éditeur qui vous plaît et qui vous permet de coder facilement. Essayez-en plusieurs, lisez les avis des autres, faites-vous vos propres avis et choisissez le vôtre.

10.4. Les problèmes possibles

10.4.1. Voir les erreurs obtenues

Vous ne l'avez peut-être pas encore remarqué, mais... écrivez donc un code erroné, et essayez de l'exécuter en cliquant sur le fichier ou en utilisant les méthodes ci-dessus. Exécutez par exemple ce code tout simple.

```
a = 'Hello'
```

Problème, la console s'ouvre et se referme aussitôt, et nous n'avons pas le temps de voir l'erreur. Ici, pas de problème, nous avons délibérément fait une erreur, mais imaginez dans un code de plusieurs centaines de lignes où une petite erreur s'est glissée. Si l'on n'a pas accès à l'erreur, la régler sera très compliqué et demandera de relire tout le code. Il nous faut donc faire en sorte que la console ne se ferme pas.

Pour cela, nous allons donc utiliser un peu la console. La seule commande de la console que vous devez connaître est la commande `cd` (pour *change directory*), qui permet de changer de répertoire. Voici ce que nous allons faire :

- ouvrir une console ;
- se rendre dans le dossier de notre fichier `.rb` ;
- exécuter notre fichier `.rb`.

Ouvrir une console, vous savez faire. Pour vous rendre dans le dossier, nous allons utiliser la commande `cd`. Supposons que nous soyons dans le dossier `/home/user/` au démarrage de la console et que notre fichier qui s'appelle `test.rb` soit dans le dossier `/home/user/programmation/ruby`. Il nous faudra alors taper `cd programmation/ruby`, puis `test.rb`. On peut aussi le faire en plus d'étapes en tapant ce qui suit.

```
cd programmation
cd ruby
test.rb
```

> ! La commande `cd` est une commande bash. Elle n'est donc pas à utiliser dans IRB, mais directement dans la console.

10.4.2. La console se ferme directement

Si en cliquant sur le programme, la console se ferme directement, c'est tout à fait normal, notre programme s'exécute, puis se ferme. Pour qu'il reste ouvert, nous avons deux solutions :

- utiliser la méthode du point précédent et ouvrir le programme en passant par la console (utiliser `cd` pour se rendre dans le dossier et l'exécuter) ;
- demander au programme d'attendre, par exemple, une saisie de l'utilisateur.

Nous avons déjà vu la première méthode, voyons maintenant la seconde. Elle n'est pas beaucoup plus compliquée (en fait, elle est même plus simple), il suffit d'utiliser `gets` à la fin de notre code. Comme cela, le programme attendra une saisie de l'utilisateur et se fermera dès que celui-ci en fournira une. Par exemple…

```
a = 'Bonjour'
gets
```

> ! Utiliser `gets` ne fonctionne pas dans le cas d'un code erroné. En effet, dès que Ruby rencontre une erreur dans le programme, il l'arrête.

Ainsi, pour corriger les erreurs de votre code, par exemple, il vaut mieux se déplacer dans le dossier de votre programme à l'aide de `cd` et exécuter le programme pour trouver les erreurs.

Utilisez `gets` lorsque, par exemple, vous avez fini un programme, et que vous voulez pouvoir l'utiliser directement (ou le passer à un ami). Pour le moment, nous ne sommes pas dans ce cas, et n'avons pas le niveau de faire des programmes nécessitant cela, mais dans quelques temps, nous pourrons déjà faire des petits scripts intéressants.

10.4.3. Le programme ne se lance pas

Il peut aussi arriver que, lorsque vous cliquez sur le programme (ou l'exécutez grâce au script de Notepad++, par exemple), celui-ci ne se lance pas, ou encore que le fichier s'ouvre avec votre éditeur de texte ou un autre programme. Cela signifie tout simplement que le programme avec lequel les fichiers `.rb` sont lancés n'est **pas** Ruby. Pour régler cela, il vous suffit donc de choisir l'interpréteur Ruby en tant que programme par défaut.

Cependant, il peut arriver que vous vouliez justement que le programme s'ouvre par défaut avec votre éditeur. Dans ce cas, lorsque vous voulez lancer le programme, il faut indiquer à votre ordinateur que vous voulez le lancer avec l'interpréteur Ruby. S'il s'agit de le lancer en cliquant dessus, les OS ont toujours une option « Ouvrir avec » accessible depuis un clic droit. Si vous voulez le lancer depuis la console, il faut alors utiliser la commande `ruby nom_du_fichier.rb` plutôt que `nom_du_fichier.rb` pour lancer le fichier avec Ruby. De même, il faudra alors changer vos scripts et indiquer que le fichier est à ouvrir avec Ruby. Ainsi, la dernière ligne du script de NppExec devient `NPP_RUN cmd /c ruby $(FULL_CURRENT_PATH) && pause`.

Vous pouvez maintenant écrire vos codes dans des fichiers et les exécuter.

Quatrième partie

Conclusion

Ce tutoriel d'introduction à Ruby est maintenant terminé. Mais la route à parcourir est encore longue. Vous pouvez regarder les autres tutoriels sur Ruby ou encore apprendre l'algorithmique (qui vous sera vraiment utile) avec ce tutoriel et les autres tutoriels d'algorithmique .

N'oubliez pas, la pratique est votre meilleure amie. Exercez-vous, exercez-vous et… amusez-vous!

Dans ce tutoriel, nous avons souvent dit de certaines pratiques qu'elles étaient bonnes ou mauvaises. Pour cela, nous nous sommes appuyés en particulier sur ce document , notamment pour les pratiques de mise en page (nommage, indentation…).

Liste des abréviations

DRY Don't Repeat Yourself. 113

HT Hors taxes. 44

IRB Interactive Ruby. 1, 13, 14, 16, 17, 27, 35, 76, 117, 129, 134

PGCD Plus Grand Commun Diviseur. 82, 85

TTC Toutes taxes comprises. 44

TVA Taxe sur la valeur ajoutée. 44, 47