Karnaj

Zeste de savoir

Une introduction à Ruby

3 novembre 2021

Table des matières

Introduction 9

I. Remerciements 11

II. Les bases 13

II.1. Introduction 14
Introduction . . . 14
II.1.1. Qu'est-ce que Ruby ? . . . 14
II.1.2. Installation . . . 15
II.1.2.1. Installation sous Windows . . . 15
II.1.2.2. Installation sous Linux . . . 16
II.1.2.3. Installation sous OS X . . . 16
II.1.3. Prise en main . . . 17
II.1.3.1. IRB . . . 17
II.1.3.2. Opérations mathématiques . . . 18
II.1.3.3. Entiers et flottants . . . 20
II.1.4. Exercices . . . 20
Conclusion . . . 21
Contenu masqué . . . 21

II.2. Variables et chaînes de caractères 23
Introduction . . . 23
II.2.1. Variables . . . 23
II.2.1.1. Déclaration de variables et opérations . . . 23
II.2.1.2. La valeur `nil` . . . 24
II.2.1.3. Constantes . . . 25
II.2.1.4. Conventions de nommage . . . 25
II.2.2. Chaînes de caractères . . . 26
II.2.2.1. Opérations sur les chaînes de caractères . . . 27
II.2.3. Conversion de variables . . . 29
II.2.4. Saisir et afficher des informations . . . 31
II.2.4.1. Afficher des informations . . . 31
II.2.4.2. La méthode `gets` . . . 32
II.2.4.3. Les caractères spéciaux . . . 33
II.2.5. Exercices . . . 34
II.2.5.1. Exercice 1 . . . 35
II.2.5.2. Exercice 2 . . . 35

II.2.5.3. Exercice 3 . . . 35
II.2.5.4. Exercice 4 . . . 36
Conclusion . . . 36
Contenu masqué . . . 36

II.3. Les conditions **39**
Introduction . . . 39
II.3.1. Opérateurs de comparaison . . . 39
II.3.1.1. Mise entre parenthèses . . . 40
II.3.2. La structure if-else . . . 41
II.3.2.1. Le mot-clé `elsif` . . . 42
II.3.2.2. La structure `unless` . . . 43
II.3.3. La structure case-when . . . 44
II.3.3.1. Les intervalles . . . 45
II.3.4. Les conditions ternaires . . . 46
II.3.5. Exercices . . . 47
II.3.5.1. Exercice 1 . . . 47
II.3.5.2. Exercice 2 . . . 47
II.3.5.3. Exercice 3 . . . 48
Conclusion . . . 48
Contenu masqué . . . 49

II.4. Les boucles **52**
Introduction . . . 52
II.4.1. La boucle while . . . 52
II.4.1.1. La structure `begin-while` . . . 53
II.4.1.2. La structure `until` . . . 54
II.4.2. La boucle for . . . 55
II.4.3. Contrôler l'exécution d'une boucle . . . 56
II.4.3.1. Le mot-clé `break` . . . 56
II.4.3.2. Le mot-clé `next` . . . 57
II.4.3.3. Le mot-clé `redo` . . . 57
II.4.4. La boucle loop . . . 58
II.4.5. Les itérateurs . . . 59
II.4.5.1. La méthode `each` . . . 59
II.4.5.2. La méthode `times` . . . 60
II.4.5.3. La méthode `upto` . . . 60
II.4.5.4. La méthode `downto` . . . 60
II.4.5.5. La méthode `step` . . . 61
II.4.6. Exercices . . . 62
II.4.6.1. Exercice 1 . . . 62
II.4.6.2. Exercice 2 . . . 62
II.4.6.3. Exercice 3 . . . 62
Conclusion . . . 63
Contenu masqué . . . 63

II.5. Les tableaux **65**
Introduction . . . 65

II.5.1. Généralités sur les tableaux . . . 65
II.5.1.1. Qu'est-ce qu'un tableau . . . 65
II.5.1.2. Déclarer un tableau . . . 66
II.5.2. Opérations sur les tableaux . . . 67
II.5.2.1. Accéder à un élément . . . 67
II.5.2.2. Concaténation et ajout d'éléments . . . 68
II.5.2.3. Parcourir le tableau . . . 70
II.5.3. Lien avec les chaînes de caractères . . . 71
II.5.3.1. Accéder à un élément . . . 72
II.5.3.2. Parcourir une chaîne de caractères . . . 72
II.5.3.3. De la chaîne au tableau . . . 73
II.5.4. Exercices . . . 74
II.5.4.1. Exercice 1 . . . 74
II.5.4.2. Exercice 2 . . . 74
II.5.4.3. Exercice 3 . . . 74
Conclusion . . . 75
Contenu masqué . . . 75

II.6. Les méthodes **78**
Introduction . . . 78
II.6.1. Principe et schéma d'une méthode . . . 78
II.6.1.1. Qu'est-ce qu'une méthode? . . . 78
II.6.1.2. Déclarer une méthode . . . 78
II.6.2. Écrire des méthodes . . . 80
II.6.2.1. Procédure . . . 80
II.6.2.2. Valeur de retour . . . 81
II.6.2.3. Redéfinition de méthode . . . 82
II.6.3. Les paramètres . . . 83
II.6.3.1. Méthodes à un argument . . . 83
II.6.3.2. Méthodes à plusieurs arguments . . . 84
II.6.3.3. Valeurs par défaut . . . 85
II.6.4. Exercices . . . 86
II.6.4.1. Exercice 1 . . . 86
II.6.4.2. Exercice 2 . . . 87
II.6.4.3. Exercice 3 . . . 87
Conclusion . . . 88
Contenu masqué . . . 88

II.7. Les hachages **91**
Introduction . . . 91
II.7.1. Des tableaux associatifs . . . 91
II.7.1.1. Qu'est-ce qu'un hachage . . . 91
II.7.1.2. Déclarer un hachage . . . 92
II.7.2. Opérations sur les hachages . . . 93
II.7.2.1. Accéder à un élément . . . 93
II.7.2.2. Ajout d'éléments . . . 94
II.7.2.3. Parcourir le hachage . . . 95

II.7.3. Hachages et tableaux . . . 96
II.7.3.1. La clarté . . . 97
II.7.3.2. L'ordre . . . 97
II.7.3.3. La flexibilité . . . 98
II.7.4. Exercices . . . 98
Conclusion . . . 100
Contenu masqué . . . 100

II.8. Retour sur les variables **103**
Introduction . . . 103
II.8.1. Une histoire de références . . . 103
II.8.1.1. Qu'est-ce qu'une variable? . . . 103
II.8.1.2. Une histoire d'identifiant . . . 104
II.8.1.3. Modification de variables . . . 105
II.8.2. Les variables globales . . . 106
II.8.2.1. Un problème: passage de variables aux méthodes . . . 106
II.8.2.2. Les variables globales . . . 109
II.8.2.3. Variables globales réservées . . . 110
II.8.3. Les symboles . . . 110
II.8.3.1. Garder le même identifiant . . . 110
II.8.3.2. Les symboles et les chaînes de caractères . . . 111
II.8.3.3. Les symboles et les hachages . . . 112
II.8.4. Modifier nos variables dans des méthodes . . . 113
II.8.4.1. Utiliser le retour des méthodes . . . 113
II.8.4.2. Retourner plusieurs variables? . . . 113
II.8.4.3. Copier un objet . . . 115
Conclusion . . . 116

III. Organisation de code **117**

Introduction **118**

III.1. Les objets **119**
Introduction . . . 119
III.1.1. Ruby et les objets . . . 119
III.1.1.1. Qu'est-ce qu'un objet . . . 119
III.1.1.2. Orientation objet de Ruby . . . 119
III.1.1.3. Création du premier objet . . . 120
III.1.2. Construction d'un objet . . . 121
III.1.2.1. Méthodes . . . 121
III.1.2.2. Avec les objets existants . . . 122
III.1.2.3. Conversions . . . 122
III.1.3. Analyse d'un objet . . . 123
III.1.3.1. Inspecter un objet . . . 123
III.1.3.2. Comparaison . . . 125
III.1.3.3. Envoi de message . . . 127
III.1.4. Exercices . . . 129
Conclusion . . . 129

III.2. Les classes **130**
Introduction . . . 130
III.2.1. Attributs . . . 130
III.2.1.1. Accesseurs et mutateurs . . . 131
III.2.1.2. Conventions pour les méthodes . . . 131
III.2.2. Un modèle de construction . . . 133
III.2.2.1. Création de classe . . . 133
III.2.2.2. Des méthodes . . . 135
III.2.2.3. Initialisation d'objets . . . 136
III.2.3. Du nouveau sur les classes? . . . 138
III.2.3.1. Travailler avec une classe . . . 138
III.2.3.2. Visibilité . . . 140
III.2.3.3. Interface . . . 143
III.2.4. Exercices . . . 145
Conclusion . . . 145
Contenu masqué . . . 145

III.3. Les modules **149**
Introduction . . . 149
III.3.1. Les modules . . . 149
III.3.1.1. Création de module . . . 149
III.3.1.2. Des espaces de noms . . . 150
III.3.1.3. Déverser le tiroir . . . 152
III.3.2. Le mot-clé self . . . 152
III.3.2.1. À quoi se rapporte le mot-clé `self` . . . 152
III.3.2.2. Du `self` implicite dans les méthodes . . . 154
III.3.2.3. Retour sur `include` et `module_function` . . . 155
III.3.3. Modules et classes . . . 157
III.3.3.1. Classes ou modules? . . . 157
III.3.3.2. De l'imbrication de classes et de modules . . . 158
III.3.3.3. Bonnes pratiques . . . 160
III.3.4. Exercices . . . 162
Conclusion . . . 162

III.4. Des objets plus complexes **163**
Introduction . . . 163
III.4.1. Héritage et composition . . . 163
III.4.1.1. Héritage . . . 163
III.4.1.2. L'arbre d'héritage de Ruby . . . 166
III.4.1.3. Composition . . . 167
III.4.2. Mixer un module dans une classe . . . 168
III.4.2.1. Les *mixins* . . . 168
III.4.2.2. Un peu de *lookup*? . . . 170
III.4.3. Quelle technique utiliser? . . . 175
III.4.3.1. Bien utiliser l'héritage . . . 175
III.4.3.2. Le typage de canard . . . 179
III.4.3.3. Héritage, mixins ou composition? . . . 181
III.4.4. Exercices . . . 182

Conclusion . . . 182

III.5. La gestion des erreurs **183**
Introduction . . . 183
III.5.1. Les exceptions . . . 183
III.5.1.1. Qu'est-ce qu'une exception . . . 183
III.5.1.2. Attraper une exception . . . 184
III.5.1.3. Mais quelle est donc cette exception? . . . 186
III.5.2. Gérer les exceptions . . . 187
III.5.2.1. La classe des exceptions . . . 187
III.5.2.2. Rattraper des exceptions particulières . . . 188
III.5.2.3. Bonne gestion des exceptions . . . 190
III.5.3. De l'autre côté du miroir . . . 193
III.5.3.1. Lever des exceptions . . . 193
III.5.3.2. Créer nos propres exceptions . . . 194
III.5.3.3. Bonne gestion des exceptions . . . 195
III.5.4. Exercices . . . 196
Conclusion . . . 196

III.6. Les blocs **198**
Introduction . . . 198
III.6.1. La notion de fermeture . . . 198
III.6.1.1. Les fermetures . . . 198
III.6.1.2. Des fonctions d'ordre supérieur . . . 199
III.6.1.3. Et Ruby dans tout ça? . . . 200
III.6.2. Les blocs . . . 202
III.6.2.1. Utiliser un bloc dans une fonction . . . 202
III.6.2.2. Des blocs avec des paramètres . . . 204
III.6.2.3. Appel explicite . . . 205
III.6.2.4. Quelques subtilités . . . 205
III.6.3. Proc et lambda . . . 206
III.6.3.1. Réutiliser des blocs? . . . 206
III.6.3.2. Les `Procs` . . . 208
III.6.3.3. Les lambdas . . . 209
III.6.4. Exercices . . . 212
III.6.4.1. Exercice 1 . . . 212
III.6.4.2. Exercice 2 . . . 212
III.6.4.3. Exercice 3 . . . 212
III.6.4.4. Exercice 4 . . . 213
Conclusion . . . 213
Contenu masqué . . . 213

IV. Les outils de Ruby **216**

Introduction **217**

IV.1. Les objets énumérables **218**
Introduction . . . 218

IV.1.1. Le module Enumerable . . . 218
IV.1.1.1. Vérifier des conditions sur un tableau . . . 218
IV.1.1.2. Récupérer des éléments du tableau . . . 220
IV.1.1.3. Autres méthodes utiles . . . 221
IV.1.2. Créer des objets énumérables . . . 226
IV.1.2.1. Mixer le module `Enumerable` . . . 226
IV.1.2.2. Un exemple . . . 227
IV.1.2.3. Des structures de données . . . 228
IV.1.3. La classe Enumerator . . . 228
IV.1.4. Exercices . . . 228

IV.2. Les expressions régulières **229**
Introduction . . . 229
IV.2.1. Construire une expression régulière . . . 229
IV.2.1.1. Tester la présence d'un mot dans une chaîne . . . 229
IV.2.1.2. Les symboles . . . 231
IV.2.2. Des informations supplémentaires . . . 236
IV.2.2.1. La classe `MatchData` . . . 236
IV.2.2.2. Capture d'expressions . . . 238
IV.2.2.3. Contexte de comparaison . . . 240
IV.2.3. Et encore? . . . 241
IV.2.3.1. Plusieurs comportements? . . . 241
IV.2.3.2. Variables globales pour les expressions régulières . . . 242
IV.2.3.3. Tous les problèmes ne sont pas des clous . . . 243
IV.2.4. Exercices . . . 244
IV.2.4.1. Exercice 1 . . . 244
IV.2.4.2. Exercice 2 . . . 244
IV.2.4.3. Exercice 3 . . . 244
Conclusion . . . 245
Contenu masqué . . . 245

V. Annexes **248**

V.1. Écrire le code dans des fichiers **249**
Introduction . . . 249
V.1.1. Les éditeurs de texte . . . 249
V.1.1.1. Pourquoi choisir un éditeur de texte? . . . 249
V.1.1.2. Des éditeurs de texte adaptés à la programmation . . . 249
V.1.1.3. Exécuter le code . . . 250
V.1.2. Windows—Notepad++ . . . 251
V.1.2.1. Installer Notepad++ . . . 251
V.1.2.2. Exécuter le code . . . 251
V.1.3. Linux—gedit . . . 253
V.1.3.1. Installer gedit . . . 253
V.1.3.2. Exécuter le code . . . 253
V.1.4. Les problèmes possibles . . . 254
V.1.4.1. Voir les erreurs obtenues . . . 254

V.1.4.2. La console se ferme directement 254
V.1.4.3. Le programme ne se lance pas 255
Conclusion 255

Conclusion **256**

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.

Première partie

Remerciements

I. 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 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.

Deuxième partie

Les bases

II.1. Introduction

Introduction

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

II.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.

II.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.

II.1.2.1. Installation sous Windows

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

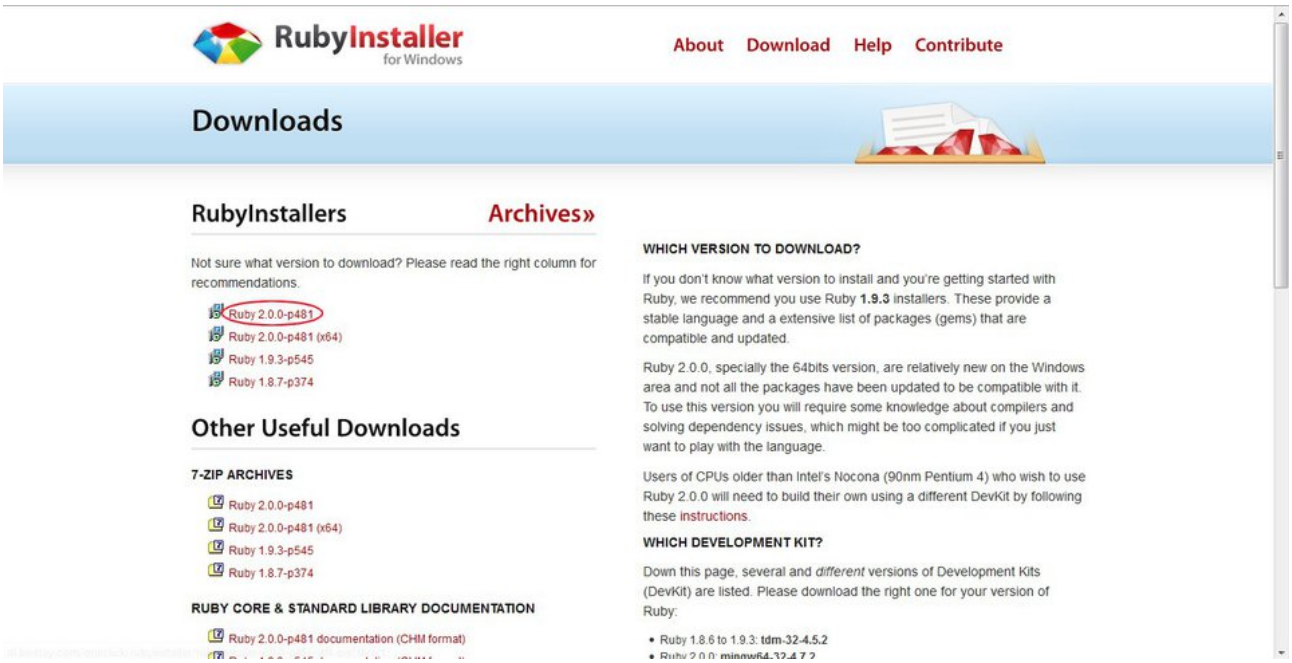

FIGURE II.1.1. – Capture du site.

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.

FIGURE II.1.2. – 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.

Notons quand même que bien que Ruby soit multi-plateforme, Linux est particulièrement adapté à son utilisation, bien plus que Windows.

II.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.
```

II.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.

II.1.3. Prise en main

II.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.

```
1 3; 2
```

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

```
1 3 4
```

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

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

II.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`.

II.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.

```
> 10   => 10
> 0d10 => 10
> 0b10 => 2
> 010  => 8
> 0o10 => 8
```

```
> 0x10 => 16
> 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`.

II.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!

II.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

Conclusion

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.

II.2. Variables et chaînes de caractères

Introduction

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

II.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.

II.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.

```
> x = 3
=> 3
x = 5
=> 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).

```
> a = 3
=> 3
> b = 2
=> 2
> c = a + b
=> 5
```

`c` vaut 5.

```
> a = 3
=> 3
> b = 2
=> 2
> a = a + b
=> 5
```

`a` vaut 5.

II.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 =`).

```
> variable =
> variable
=> 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.

II.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.

II.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	unles
=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.

```
1 ma_super_variable
2 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.

i

Dans ce tutoriel, nous allons suivre les bonnes pratiques de Ruby. Elles demandent des noms de variables en anglais et nos noms de variables seront donc en anglais. Mais pas d'inquiétude, ce n'est pas de l'anglais compliqué et les codes seront bien sûr expliquées. De plus, c'est une bonne manière de travailler un peu son anglais, langue qui sera indispensable dans l'informatique.

II.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.

```
1 > "Bonjour"
2 => "Bonjour"
3 > 'Bonjour'
4 => "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.

```
> word = 'zeste'
=> "zeste"
> Constant = 'vrai'
=> "vrai"
```

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

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

```
> word = %(zeste)
=> "zeste"
> word = %q(zeste)
=> "zeste"
> word = %Q(zeste)
=> "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 `%[]`.

II.2.2.1. Opérations sur les chaînes de caractères

Nous pouvons effectuer quelques opérations sur les chaînes de caractères.

```
> 'Bonjour ' + 'tout le monde'
=> "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.

```
> word = 'zeste'
=> "zeste"
> 'Faites un ' + word + ' envers votre prochain.'
=> "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 `*`.

```
> word = 'zeste '
=> "zeste "
> word * 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 `word` 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 "
> repeated = mot * 3
=> "zeste zeste zeste "
```

La variable `repeated` vaut `zeste zeste zeste`.

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

```
> word = '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"
> word = "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.

II.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
> x.to_s
=> "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.

```
> x = 'Bonjour'
=> "Bonjour"
> x.to_i
=> 0
> y = 'zeste'
=> "zeste"
> y.to_f
=> 0.0
```

Alors que…

```
> x = '3'
=> "3"
> x.to_i
=> 3
> x.to_f
=> 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.

II.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.

II.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.

```ruby
puts "Chaîne"
puts 2
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.

```
1 print 'Bonjour'
2 print 2
3 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…

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

… plutôt que…

```
1 print 'Bonjour'
2 print variable1
3 print variable2
```

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

II.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.

```
1 print 'Entrez votre nom : '
2 name = gets.chomp
3 print "Bonjour #{name}"
```

Le mieux est d'essayer. Après avoir affiché «Entrez votre nom:», le programme se stoppe. Là, nous saisissons quelque chose avant de valider en appuyant sur la touche Entrée. Nous devrions obtenir une sortie ressemblant àceci (ici, nous avons saisi «Zesteur»).

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

? 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 `name = gets`. Affichons maintenant `name`. 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.

```
name = gets
name = name.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.

II.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…

```
print "zeste\n"         # Affiche « zeste » et va à la ligne.
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}"}`).

II.2.5. Exercices

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

II.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

II.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

II.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

II.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

Conclusion

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.

```ruby
print 'Entrez votre nom : '
last_name = gets.chomp
print 'Entrez votre prénom : '
first_nom = gets.chomp
puts "Bonjour #{las_name} #{first_name}."
```

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`.

```ruby
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.

```ruby
print 'Entrez votre âge : '
age = gets.chomp.to_i # On convertit la saisie en entier.
puts age
```

```
4  age = age * 365        # On passe aux jours.
5  puts âge
6  age = age * 24         # On passe aux heures.
7  puts age
8  age = age * 60         # On passe aux minutes.
9  puts age
10 age = age * 60         # On passe aux secondes.
11 puts age
```

Dans celui-ci on affiche la variable `age` 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.

```
1  print 'Entrez votre âge : '
2  age = gets.chomp.to_i # On convertit la saisie en entier.
3  days = age * 365
4  hours = days * 24
5  minutes = heures * 60
6  seconds = minutes * 60
7  puts age
8  puts days
9  puts hours
10 puts minutes
11 puts seconds
```

Retourner au texte.

Contenu masqué n°5

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

```
1  print 'Entrez le premier nombre : '
2  number1 = gets.chomp.to_f # On demande le nombre 1 et on le
       convertit en flottant.
3  print 'Entrez le deuxième nombre : '
4  number2 = gets.chomp.to_f # On demande le nombre 2 et on le
       convertit en flottant.
5
6  puts "#{number1} + #{number2} = #{number1 + number2}"
7  puts "#{number1} - #{number2} = #{number1 - number2}"
8  puts "#{number1} * #{number2} = #{number1 * number2}"
9  puts "#{number1} / #{number2} = #{number1 / number2}"
```

Retourner au texte.

II.3. Les conditions

Introduction

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.

II.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…

```
1 boolean = true
2 false_boolean = 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.

II.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`.

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

Ici, nous vérifions si la variable `age` 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 age = gets.chomp.to_i # On récupère une saisie et on la convertit
      en entier.
2
3 if age < 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 age = gets.chomp.to_i # On récupère une saisie et on la convertit
      en entier.
```

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

II.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 age = gets.chomp.to_i
2
3 if age < 18    # Si âge est inférieur à 18…
4   print 'Vous êtes mineur.'
5 elsif age > 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-clé `then`.

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

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

```
age = gets.chomp.to_i

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

II.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.

```
age = 19

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

Là encore, nous pouvons utiliser la structure condensée.

```
print 'Vous êtes majeur.' unless age < 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`.

II.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.

```
number = gets.chomp

case number     # 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
number = gets.chomp

case number
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
number = gets.chomp

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

II.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.

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

case mark
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`.

II.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.

```
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.

```
age = 19

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

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

```
1 age = 19
2
3 (age < 18) ? (print 'Mineur') : (age < 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.

II.3.5. Exercices

Voilà une nouvelle série d'exercices.

II.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

II.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

II.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

Conclusion

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.'
number = gets.chomp.to_i
max = number
number = gets.chomp.to_i
max = number if number > max
number = gets.chomp.to_i
max = number if numbet > 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.'
number1 = gets.chomp.to_i
number2 = gets.chomp.to_i
number3 = gets.chomp.to_i
if number1 > number2 # Si number1 > number2, on les échange.
  tmp = number1
  number1 = number2
  number2 = tmp
end
if number2 > number3 # Si number2 > number3, on les échange.
  tmp = number2
  number2 = number3
  number3 = tmp
  # l'ancien number3 était plus petit que l'ancien number2, alors
      il peut
  # être plus petit que number 1.
  if number1 > number2
    tmp = number1
    number1 = number2
    number2 = tmp
  end
end
puts number1, number2, number3
```

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 : '
year = gets.chomp()
year = année.to_i()
if year % 400 == 0
  puts 'L'année est bissextile.'
elsif year % 4 == 0 && year % 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 year % 400 == 0 || (year % 4 == 0 && year % 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 : '
priceHT = gets.chomp.to_f
print 'Entrez le code de votre produit : '
code = gets.chomp.to_i

if code == 1
  priceTTC = priceHT + TVA1 / 100 * priceHT
elsif code == 2
  priceTTC = priceHT + TVA2 / 100 * priceHT
elsif code == 3
  priceTTC = priceHT + TVA3 / 100 * priceHT
end

case code
when 1..3
  print "Le prix TTC de votre produit est #{priceTTC}."
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 : '
priceHT = gets.chomp.to_f
price 'Entrez le code de votre produit : '
code = gets.chomp.to_i

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

Retourner au texte.

II.4. Les boucles

Introduction

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.

II.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.

```
1 n = 1
2 while n <= 10 # Tant que n est inférieur ou égal à 10, le code est
      exécuté.
3   print "#{n * 8} "
4   n = n + 1 # On ajoute 1 à n à chaque tour pour que sa valeur
        atteigne 10.
5 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.

```
1 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.

II.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.

```
1 n = 11
2
3 begin (print "#{n * 8} "; n = n + 1) end while n <= 10
```

II.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`.

```
1 n = 1
2 until n > 10 # Jusqu'à ce que n soit plus grand que 10, le code
      est exécuté.
3   print "#{n * 8} "
4   n = n + 1
5 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.

```
1 n = 1
2
3 (print "#{n * 8} "; n = n + 1) until n > 10
```

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

```
1 # Mauvais code.
2 while !condition
3   # Instructions.
```

```ruby
end

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

II.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.

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

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

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

```
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…

```ruby
for n in 0...5
  print "#{n} " # Instruction(s).
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.

II.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.

II.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.

II.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
```

II.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`.

II.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`.

II.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.

II.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 (1..8).each do |i|
2   puts i
3 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 (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.

II.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`.

II.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
```

II.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.

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

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

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

II.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.

```ruby
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.

```ruby
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.

```ruby
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.

II.4.6. Exercices

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

II.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

II.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

II.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

Conclusion

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.

```ruby
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

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

II.5. Les tableaux

Introduction

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.

II.5.1. Généralités sur les tableaux

II.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** .

II.5.1.2. Déclarer un tableau

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

```
1 tab = []
```

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

```
1 tab = []
2 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.

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

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

II.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.

```
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.

```
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.

II.5.2. Opérations sur les tableaux

II.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.

```
grammar = %w[mais ou et donc or ni car]
print grammar[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.

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

II.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.

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

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

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

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

Avec ce code, on obtientceci.

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

II.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).

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

```
3 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`.

II.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.

```
1 meat_menu = ["un steak haché", "une entrecôte", "un rôti",
      "un faux-filet"]
2 for meat in meat_menu
3   puts "Voulez-vous #{meat} pour le dîner de ce soir ?"
4 end
```

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

```
1 Voulez-vous un steak haché pour le dîner de ce soir ?
2 Voulez-vous une entrecôte pour le dîner de ce soir ?
3 Voulez-vous un rôti pour le dîner de ce soir ?
4 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.

```
1 meat_menu = ["un steak haché", "une entrecôte", "un rôti",
      "un faux-filet"]
2 4.times { |i| puts
      "Voulez-vous #{meat_menu[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.

II.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 meat_menu = ["un steak haché", "une entrecôte", "un rôti",
      "un faux-filet"]
2 meat_menu.each { |meat| puts
      "Voulez-vous #{meat} 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.

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

II.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.

II.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.

```
1 str = 'Bonjour'
2 print str[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»…

```
1 str = 'Bonjour'
2 print str[1..3]
```

II.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.

```
1 str = 'bonjour'
2 7.times { |i| print "#{str[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.

```
1 str = 'bonjour'
2 str.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).

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

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

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

II.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`, quant 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 chaîne 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"
```

II.5.4. Exercices

II.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

II.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

II.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

Conclusion

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

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

```
13 end
14 puts
15
16 tab.each { |x| puts "Votre tableau contient #{x}." }
```

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  str = gets.chomp
3
4  current = chaîne[0]
5  n = 0
6  tab = []
7  str.each_char do |char|
8    if char == current
9      n = n + 1
10   else
11     tab << n
```

```
12      tab << current
13      current = char
14      n = 1
15    end
16  end
17  tab << n
18  tab << current
19  result = tab.join
20  print result
```

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

II.6. Les méthodes

Introduction

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.

II.6.1. Principe et schéma d'une méthode

II.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 `repeatMessage` 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*.

II.6.1.2. Déclarer une méthode

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

```
1 # Définition de la méthode add.
2 def add
```

```
3   # Instructions.
4 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.

```
1 print 'J'appelle la méthode add : '
2 add
3 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.

```
1 a = 4
2 b = 5
3 hello
```

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

```
1 def hello
2   # Instructions.
3 end
4
5 a = 4
6 b = 5
7 hello
```

Une méthode doit être définie avant son utilisation.

Ainsi, ce code ne fonctionne pas.

```
1 a = 4
2 b = 5
3 hello
4
5 def hello
6   # Instructions.
```

```
7 end
```

On pourrait pourtant penser que ce code est le même que le précédent. Cependant, quand l'interpréteur arrive à `hello`, il ne connaît pas encore `hello`, 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).

II.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.

II.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`.

```
1 > print
2 => nil
```

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

```
1 # Définition de la méthode bonjour.
2 def hello
3   print 'Hello World!'
4 end
5
6 hello # 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.

II.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.

```
1 # Définition de la méthode add.
2 def add
3   x = 9
4   y = 5
5   return x + y # Retour.
6 end
7
8 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.

```
1 def add
2   x = 9
3   y = 5
4   x + y # Retour.
5 end
6
7 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.

```ruby
def hello
  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.

```ruby
def hello
  print 'Bonjour.'
  return
  print 'Merci.'
end
```

II.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 `hello` fera deux choses différentes.

```ruby
def hello
  puts 'Bonjour.'
  return 5
end

puts hello

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

print hello
```

Le premier appel à `hello` affiche «Bonjour.» et retourne 5. Le second appel à `hello` 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.

II.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.

II.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.

```
1 def method_name(arg_name)
2
3 end
```

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

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

On obtient alors ceci.

```
1 Bonjour utilisateur.
2 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.

```
1 def hello(name)
2   puts "Bonjour #{name}."
3 end
4
5 hello('utilisateur')
6 hello('')
7
8 print name
```

II.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.

```
1 def method_name(arg1_name, arg2_name, arg3_name, argn_name)
2
3 end
```

Ce n'est pas vraiment 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.

```
1 def add(x, y)
2   return x + y
3 end
4
5 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`.

II.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 `hello`, 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 `héllo` s'écrit alors ainsi.

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

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 f(arg1, arg2 = 10, arg3`), comment savoir à quoi correspond `12` dans cet appel: `f(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.

```ruby
def hello(last_name, first_name)
   puts "Bonjour #{last_name} #{first_name}."
end

hello('moi', 'utilisateur')
hello(first_name = 'utilisateur', last_name = 'moi')
```

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

II.6.4. Exercices

II.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

II.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

II.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.

Conclusion

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}." }
  choice = 0
  while choice < 1 || choice > 4
    print "\nVotre choix : "
    choice = gets.chomp.to_i
  end
  puts "\nVous avez choisi de faire une #{tab[choix - 1]}."
  return choice
end

choice = menu
print "\nEntrez le premier nombre : "
number1 = gets.chomp.to_f
print 'Entrez le second nombre : '
number2 = gets.chomp.to_f
puts
if choix == 1
  print "#{number1} + #{number2} = #{number1 + number2}"
elsif choix == 2
  print "#{number1} - #{number2} = #{number1 - number2}"
elsif choix == 3
  print "#{number1} * #{number2} = #{number1 * number2}"
elsif number2 != 0
```

```
27    print "#{number1} / #{number2} = #{number1 / number2}"
28  else
29    print 'Diviser par 0, c'est mal.'
30  end
```

Retourner au texte.

Contenu masqué n°19

```
1   def pgcd(a, b)
2     if a > b
3       tmp = a
4       a = b
5       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    choice = 0
8    while choice < 1 or choice > 5
9      print "\nVotre choix : "
10     choice = gets.chomp.to_i
11   end
12   return choice
13 end
14
15 PI = 3.14159265359
16 DEG_TO_RAD = 180 / PI
17 MS_TO_KMH = 3.6
18 choice = menu
19 if choice != 5
20   print "\nEntrez le number : "
21   number = gets.chomp.to_f
22   if choice == 1
23     print number * DEG_VERS_RAD
24   elsif choice == 2
25     print number / DEG_VERS_RAD
26   elsif choice == 3
27     print number / MS_VERS_KMH
28   else
29     print number * MS_VERS_KMH
30   end
31 end
```

Retourner au texte.

II.7. Les hachages

Introduction

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**.

II.7.1. Des tableaux associatifs

II.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.

```
1 tab[2]      # Un tableau normal.
2 hash['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.

```
1 person = ['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 `'last_name'` associée à son nom;
— la case `'first_name'` associée à son prénom;
— la case `'age'` 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, `last_name`, `first_name` et `age` seraient donc des clés, alors que le nom de la personne, son prénom et son âge seraient des valeurs.

II.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.

```
1 hash = {}
```

On peut l'afficher avec `print`.

```
1 hash = {}
2 print hash
```

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.

```
1 hash = { 'last_name'  => 'Mon nom',
2          'first_name' => 'Mon prénom',
3          'age'        => 2015 }
4 print hash
```

Cette fois, on obtient affiché à l'écran: `{"last_name"=>"Mon nom", "first_name"=>"Mon prénom", "age"=>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.

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

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

II.7.2. Opérations sur les hachages

II.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.

```
hash = { 'last_name'  => 'Mon nom',
         'first_name' => 'Mon prénom',
         'age'        => 2015 }
print hash['last_name']
```

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.

```
hash = { 'last_name'  => 'Mon nom',
         'first_name' => 'Mon prénom',
         'age'        => 2015 }
print hachage['Last_name']
```

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.

II.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.

```
hash = { 'last_name'  => 'Mon nom',
         'first_name' => 'Mon prénom',
         'age'        => 2015 }
hash['other'] = "nouveau"

print hash
```

La clé `'other'` 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.

```
hash = { 'last_name'  => 'Mon nom',
         'first_name' => 'Mon prénom',
         'age'        => 2015
         'last_name'  => 'Mon nouveau nom' }
print hash
```

Et le résultat.

```
{"last_name"=>"Mon nouveau nom", "first_name"=>"Mon prénom",
    "age"=>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.

II.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`.

```
hash = { 'last_name'  => 'Mon nom',
         'first_name' => 'Mon prénom',
         'age'        => 2015 }

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

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

```
["last_name", "Mon nom"]
["first_name", "Mon prénom"]
["age", 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.

II.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.

```
hash = { 'last_name'  => 'Mon nom',
         'first_name' => 'Mon prénom',
         'age'        => 2015 }

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

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

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 [”last_name”, ”Mon nom”] est associée à l'indice 0».

```ruby
hash = { 'last_name'  => 'Mon nom',
         'first_name' => 'Mon prénom',
         'age'        => 2015 }

hash.each_with_index { |v, i| puts
    "La valeur #{v} est associée à l'indice #{i}." }
```

II.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`.

```ruby
hash = { 'last_name'  => 'Mon nom',
         'first_name' => 'Mon prénom',
         'age'        => 2015 }

hash.each_value { |value| puts
    "La valeur #{value} est dans le hachage." }
```

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

II.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.

```ruby
hash = { 'last_name'  => 'Mon nom',
         'first_name' => 'Mon prénom',
         'age'        => 2015 }

hash.each_key { |clé| puts
    "La clé #{key} est une des clés du hachage." }
```

II.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.

II.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.

II.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.

```
1 hash = { 1 => 1,
2          2 => 2,
3          3 => 3 }
4
5 hash.each { |key, value| puts "#{key} : #{value}" }
```

Les éléments sont affichés dans le bon ordre. Super!

Mais, regardons maintenant ce code.

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

hash[5] = 5
hash[4] = 4

hash.each { |key, value| puts "#{key} : #{value}" }
```

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.

II.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.

II.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.

Conclusion

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 `menu` qui affiche le menu et renvoie le choix de l'utilisateur, une méthode `add_student` qui ajoute un élève au tableau, et une méthode `print_students` qui affiche la liste des élèves.

```ruby
def add_student(tab)
  print "\nNom : "
  last_name = gets.chomp
  print 'Prénom : '
  first_name = gets.chomp
  print 'Âge : '
  age = gets.chomp.to_i
  tab << { 'last_name'    => nom,
           'first_name' => prénom,
           'âge'    => âge }
  print "\nL'élève #{last_name} #{first_name} a été ajouté.\n\n"
end

def print_students(tab)
  tab.each { |e| puts "- #{e['last_name']} #{e['first_name']}" }
  print "\n\n"
end

def menu
  print "1. Ajouter un élève.\n2. Afficher la liste des élèves.\n
      3. Quitter.\n\n"
  print 'Votre choix ? '
  return gets.chomp.to_i
end

tab = []
choice = 0

while choice != 3
  choice = menu
  if choice == 1 then
    add_student(tab)
  elsif choice == 2 then
    print_students(tab)
  end
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.

```ruby
def student_data(tab)
  puts 'Nom de l'élève : '
  last_name = gets.chomp
  print "\n\n"
  tab.each do |e|
    puts "#{e["last_name"]} #{e["first_name"]} a #{e["age"]} ans."
        if e["last_name"] == last_name
  end
  print "\n\n"
end
```

Retourner au texte.

II.8. Retour sur les variables

Introduction

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

II.8.1. Une histoire de références

II.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.

II.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.

```
1 a = 4
2 b = 4
3 puts a.object_id
4 puts b.object_id
```

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

Nous pouvons même aller encore plus loin.

```
1 a = 4
2 b = 4
3 puts a.object_id
4 puts b.object_id
5 puts 4.object_id
```

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

```
1 a = 4
2 b = a
3 puts a.object_id
4 puts b.object_id
```

II.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.

```
1 a = [122, 32]
2 b = [122, 32]
3 puts a.object_id
4 puts b.object_id
5 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).

II.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.

```ruby
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).

```ruby
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.

II.8.2. Les variables globales

II.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.

```ruby
def increment(a)
  a = a + 1
end

x = 6
increment(x)
```

```
7 print x
```

On aimerait que la valeur de `x` soit `7` après l'appel de la méthode `increment`, 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 `increment` (et dans ce cas, l'argument n'est plus nécessaire).

```
1 def increment
2   x = x + 1
3 end
4
5 x = 6
6 increment
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 `increment` n'existe que dans la méthode `increment` et pas en dehors;
— la variable `x` du reste du code n'existe qu'en dehors de la méthode `increment`.

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

Ainsi, dans la méthode `increment`, 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.
```

```ruby
end

x = 1
f(x)
puts x # x vaut toujours 1 en dehors de la méthode.
g(x)
puts x # x vaut toujours 1.
h(x)
puts x # x vaut toujours 1.
x = h(x)
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 `increment` 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.

```ruby
def f(x)
  puts "Au début de la méthode, l'id de x est #{x.object_id}."
  x = x + 1
  puts "À la fin de la méthode, l'id de x est #{x.object_id}."
end

x = 2
puts "En dehors de la méthode avant son appel, l'id de x est #{x.o
    bject_id}."
f(x)
puts "En dehors de la méthode après son appel, l'id de x est #{x.o
    bject_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.

```ruby
def f(tab)
  tab << 3
end
```

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

II.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 print_x
2   print $x
3 end
4
5 $x = 'Voici une variable globale.'
6 print_x
```

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

```
1 def increment
2   $x = $x + 1
3 end
4
5 $x = 1
6 increment
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.

II.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 actuellement à la ligne #{__LINE__}."
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
      #{__LINE__}."
```

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.

II.8.3. Les symboles

II.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 symbol = :symbol
```

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

```ruby
x = :symbol
y = :symbol
print x.object_id == y.object_id
```

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

```ruby
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).

i 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.

II.8.3.2. Les symboles et les chaînes de caractères

Les symboles sont surtout utilisés 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.

```
name1 = 'Nom'
nation1 = :fr
name_2 = 'Nom2'
nation2 = :en
name3 = 'Nom3'
nation3 = :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…

```
name = 'nom'
nation = :fr
x = name.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 `nations = %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).

II.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.

```
hash = { :last_name  => 'Mon nom',
         :first_name => 'Mon prénom',
         :age        => 2015 }
```

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.

```
hash = { last_name: 'Mon nom',
         first_name: 'Mon prénom',
         age: 2015 }
```

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.

II.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?

II.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 square(x)
  return x * x
end

print 'Entrez un nombre : '
number = gets.chomp.to_i
puts "Nous allons calculer le carré de #{number}."
number = square(number)
puts "Son carré est #{number}."
```

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.

II.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.

```
1 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.

```
1 a, b = b, a
```

Pour plus de lisibilité, nous pouvons ajouter des crochets autour des valeurs assignées. On écrit alors notre méthode de *swap*.

```
1 def swap(a, b)
2   return [b, a]
3 end
4
5 a, b = [1, 2]
6 a, b = swap(a, b)
```

Bien sûr, notre méthode `swap` n'a pas grande utilité, puisqu'il 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.

```
1 tab = [1, 2, 3]
2 a, b, c = tab
3 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 affectées 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).

II.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 copy_tab(tab)
  copy = []
  tab.each { |e| copy << e }
  copy
end

tab = [1, 2, 3]
copy = copy_tab(tab)
copy[0] = 3
print tab
print copy
```

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]
```

```
copy = tab
f(copy)
print tab
print copy

# Fonctionne

tab = [1, 2, 3]
copy = tab.dup
f(copy)
print tab
print copy
```

Conclusion

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.

Troisième partie

Organisation de code

Introduction

La première partie nous a appris ce qui était nécessaire pour écrire un programme. La deuxième partie nous apprendra à organiser notre code de manière efficace. Il y aura donc un peu plus de théorie que dans la première partie, mais rien de bien grave. Il nous faudra prendre le temps de bien lire toutes les informations et de bien les comprendre (et donc il nous faudra être bien organisé), mais comme d'habitude, la pratique sera l'élément-clé pour progresser. Il ne faut donc pas hésiter à chercher des exercices, à créer des exercices, ni à poster sur le forum.

Voici par exemple un billet avec une liste d'exercices de programmation .

En parallèle de cette partie, nous pouvons lire la partie 3. Certains chapitres nécessitent des connaissances de la partie 2, mais quand ce sera le cas, ce sera précisé. Cette troisième partie présente certains outils de Ruby qu'il est utile de connaître.

III.1. Les objets

Introduction

Dans ce chapitre, nous allons voir ce qu'est un objet et comment l'utiliser. Nous allons également créer nos premiers objets et voir quelques généralités sur les objets. Puisque nous allons tester beaucoup de petits bouts de code, il peut être plus simple d'utiliser l'interpréteur.

III.1.1. Ruby et les objets

III.1.1.1. Qu'est-ce qu'un objet

Pour commencer, il nous faut voir ce qu'est un objet. On ne va pas faire compliqué, on va regarder le monde réel. Un objet, c'est un truc, une chose. Autour de nous, il y a pleins d'objets. Des livres, des ordinateurs, des feuilles, etc. On interagit avec eux, on leur fait faire des actions, on les manipule, on les fait interagir entre eux, etc.

Un objet a des propriétés (un livre a un titre, un auteur, etc.) et on peut le manipuler (on peut l'ouvrir, le fermer, le lire, ~~le déchirer~~, etc.) et tout cela le caractérise. Si nous considérons un autre livre, c'est aussi un objet. C'est aussi un livre, mais c'est un objet différent.

Maintenant, nous voyons mieux à quoi correspond un objet de la vie réelle. Les objets dont nous allons parler sont à peu près les mêmes. Donc pour commencer, nous allons considérer qu'un objet est un truc, un bidule un machin.

III.1.1.2. Orientation objet de Ruby

En Ruby, un objet est souvent manipulé sous la forme d'une variable. Pour interagir avec lui, on effectue des opérations.

Ruby est un langage dit **orienté objet**. De nombreux langages de programmation sont orientés objets (Java, Python, etc.). Avec eux, on manipule les données grâce à des objets qu'on fait communiquer entre eux. On leur demande des informations, on les fait faire une action. On peut par exemple imaginer un objet `avion` à qui l'on demande de voler.

Ruby est de plus un langage de programmation où tout est objet. Les entiers sont des objets, `true` et `false` sont des objets, les tableaux sont des objets, absolument tout ce que nous utilisons sont des objets. Pour manipuler ces objets, les opérations que nous utilisons sont les méthodes. Par exemple, nous pouvons parcourir tous les éléments d'un tableau avec la méthode `each` ou obtenir le PGCD de deux nombres avec la méthode `gcd`.

```
tab = [1, 2, 3]                 # Création d'un objet
tab.each { |e| print e + 1 }    # Appel de méthode

nb = 36                         # Création d'un objet
print nb.gcd(12)                # Appel de méthode
```

Quand nous créons deux tableaux, ils ont les mêmes méthodes, des méthodes que les entiers ne possèdent pas forcément, que les hachages ne possèdent pas forcément et la réciproque est aussi vraie. En fait, on peut créer des objets sur des modèles préexistants. Nous verrons dans le chapitre suivant comment cela se passe, mais il nous faut garder cela en tête. Nous pouvons partir d'un modèle d'objet , d'un moule, pour créer de nouveaux objets.

Bien sûr, certains objets du monde réel ne seront pas implémentés comme on les voit dans la vraie vie. Déjà, cela dépend de notre vision de l'objet, et surtout parce que le but n'est pas de représenter la vraie vie mais de construire un programme robuste, maintenable, et qui fonctionne bien. En fait, certaines bonnes pratiques sont quasiment non naturelles. Néanmoins, elles permettent d'avoir du code bien organisé. Il faudra notamment regarder les principes SOLID , la loi de Déméter et certains patrons de conception . Pour avoir une idée rapide de ce que préconisent ces principes, nous pouvons lire cet article qui les présente de manière humoristique.

III.1.1.3. Création du premier objet

Pour créer un objet, nous allons tout simplement écrire ce code.

```
obj = Object.new
```

On appelle la méthode `new` de l'objet `Object` qui nous permet de créer un nouvel objet. Notons que `Object` est un ce que nous avons appelé plus tôt une classe, c'est-à-dire qu'elle est un modèle sur lequel est construit notre objet `obj`. `obj` a été construit en utilisant le moule `Object`. Cependant, `obj` ne représente rien de spécifique, ce n'est pas un point, ni un personnage, ni un livre. C'est juste un objet banal, un truc.

Ce sera en effet à nous de spécifier le comportement de notre objet, pour qu'il devienne ce que nous voulons qu'il soit. Pour le moment, notre objet est tout à fait quelconque. On peut regarder s'il est égal à quelque chose d'autre, l'afficher, mais sinon, il n'y a rien de bien folichon à faire.

```
> print obj
#<Object:un_nombre>
> obj == 2
false
```

Le second résultat est tout ce qu'il y a de plus normal (ben oui, notre objet n'est pas égal à 2), mais le premier est quand même assez embêtant. On demande l'affichage de `obj` et c'est ça qu'on obtient! En fait, c'est quand même assez normal, on demande l'affichage de `obj`, mais à quoi s'attend-on? `obj` c'est à peu près rien, que pourrait-on afficher?

III.1.2. Construction d'un objet

III.1.2.1. Méthodes

Pour la suite, nous allons créer un objet `message`. Commençons par le créer.

```
1 message = Object.new
```

Pour le moment, notre objet est très triste et à vrai-dire, on ne peut pas vraiment dire que c'est un message. Personnalisons-le. Pour commencer, donnons-lui quelques méthodes. Nous savons déjà définir des méthodes, nous pouvons définir une méthode pour notre objet de la même manière. Il nous suffit d'écrire le nom de l'objet suivi d'un point avant le nom de la méthode. Définissons quelques méthodes pour notre objet.

```
1  def message.author
2    'Karnaj'
3  end
4
5  def message.recipient
6    'Zeste de Savoir'
7  end
8
9  def message.content
10   'Quoi ? Un message ? Tout de suite ?'
11 end
12
13 def message.date
14   '27/12/4096'
15 end
```

On peut alors appeler les méthodes que l'on vient d'écrire.

```
1 puts "Message a été écrit par #{message.author} pour #{message.rec⌋
    ipient}."
2 print "Contenu : « #{message.content} »."
```

Nous pouvons de même définir des méthodes avec paramètres comme nous l'avons vu, etc.

III.1.2.2. Avec les objets existants

En fait, ceci est également possible avec quasiment n'importe quel autre variable. Par exemple, on va définir une méthode `renverse` pour notre variable `str`.

```
str = 'abcd'
def str.invert
  'dcba'
end
print str.invert
```

Il nous faut cependant faire attention. La méthode n'a été définie que pour la variable `str`. Si nous l'essayons sur une autre variable, nous obtiendrons une erreur et ce même si l'autre variable est une chaîne de caractère.

```
other = 'abcd'
print other.invert # => NoMethodError: undefined method `invert`
```

En effet, dans ce code, nous avons créé une nouvelle chaîne de caractère. Mais lorsque nous définissons `other` en lui affectant `str`, cela fonctionne puisqu'il fait alors référence au même objet.

```
other
print other.invert
```

Bien sûr, la même règle s'applique aux objets que nous créons.

```
message_2 = message
message_2.author         # => Karnaj
message_3 = Object.new
message_3.author         # => NoMethodError
```

III.1.2.3. Conversions

Cependant, il reste un souci à effacer. Nous voudrions bien avoir le contenu du message lorsque nous utilisons `print`. Pour cela, il nous faut savoir comment la méthode `print` fonctionne. En fait, elle appelle la méthode `to_s` de l'objet. Ainsi, elle obtient une chaîne de caractère à afficher. Il nous suffit donc de définir `to_s` pour faire ce qu'on veut.

Notons que notre objet `message` a déjà une méthode `to_s` (grâce à la classe `Object` à partir de laquelle il a été créé). Elle ne renvoie pas ce que l'on veut, nous allons la redéfinir.

```
1 def message.to_s
2   ⌋
      'Message de #{author} à destination de #{recipient}. Contenu secret.'
3 end
```

Maintenant, lorsque l'on affiche notre objet avec `print` ou `puts`, on a bien le résultat voulu. Comme quoi ce n'était pas compliqué comme problème.

En écrivant `author`, nous appelons la méthode `author` de l'objet courant (donc de `message`). De manière générale, on peut appeler n'importe quelle méthode de l'objet courant de cette façon.

Dans le même genre, nous pouvons définir des méthodes pour convertir notre message dans un autre type. Nous pourrions par exemple faire une méthode `convert_into_integer`, mais pour respecter les conventions de Ruby, nous allons garder les noms que nous connaissons déjà. Bien sûr, nous n'allons définir que les méthodes qui ont un sens (définir `to_i` ou `to_f` n'aurait pas beaucoup de sens).

```
1 def message.to_a
2   [author,
3    recipient,
4    content,
5    date]
6 end
```

III.1.3. Analyse d'un objet

Dans cette partie, nous allons surtout faire de la théorie. Nous allons pratiquer sur des choses qui nous seront utiles en théorie et qui sont toujours bonnes à savoir.

III.1.3.1. Inspecter un objet

Il est temps d'inspecter notre objet, de voir ce qu'il a dans les entrailles. Nous nous doutons bien que notre objet n'est pas vide, et qu'il est créé avec quelques méthodes. D'ailleurs, nous avons pu remarquer qu'il avait déjà une méthode `to_s` (sinon, utiliser `puts` avec notre message n'aurait pas fonctionné). En fait, notre objet a été créé avec quelques méthodes. Pour connaître les méthodes d'un objet, nous utilisons la méthode `methods` qui nous renvoie le tableau des méthodes d'un objet.

```
print message.methods
```

Nous pouvons constater que les méthodes que nous avons créées sont aussi présentes, mais ce qui doit le plus nous marquer, c'est le nombre de méthodes déjà présentes. Il y en a déjà beaucoup.

Ici, nous allons nous intéresser à quelques-une de ces méthodes (moins d'une dizaine).

La méthode `methods` nous donne la liste des méthodes d'un objet, la méthode `respond_to?`, quant à elle, nous permet de savoir si un objet a une méthode. Elle prend en paramètre un symbole (ou une chaîne de caractère), celui correspondant à la méthode dont on veut tester l'existence, et renvoie un booléen, `true` si la méthode existe, `false` sinon. Par exemple, notre objet possède une méthode `respond_to`.

```
print 'Je possède une méthode respond_to?.' if
    message.respond_to?(:respond_to?)
```

On peut alors afficher le contenu du message s'il existe et un autre message sinon.

```
if message.respond_to?(:content)
  print message.content
else
  print
      'Ce message n'a pas de contenu. On devrait peut-être le supprimer ?'
end
```

Ces deux méthodes nous permettent de faire ce qu'on appelle de l'**introspection**, c'est-à-dire qu'elles nous permettent d'examiner les objets manipulés. En fait, l'objet s'examine lui-même, d'où le nom d'introspection.

Une autre méthode qui est à connaître est la méthode `inspect`. Cette méthode renvoie une chaîne de caractères et ressemble beaucoup à `to_s`. En fait, quand `irb` affiche le contenu d'un objet ou d'une opération, c'est la méthode `inspect` qui est appelé. Nous avons également la méthode `p` qui est le pendant de `puts`, mais avec la méthode `inspect`.

```
puts message
# => Message de Karnaj à destination de Zeste de Savoir. Contenu
    secret.
p message
# => <Object:un_nombre>
```

On voit bien qu'on n'obtient pas le même résultat, et c'est normal, nous n'avons pas redéfini `inspect`, mais seulement `to_s`.

? Mais à quoi sert `inspect`?

C'est vrai que la question se pose, d'autant plus que `to_s` et `inspect` donnent souvent le même résultat. En gros, `to_s` est là pour de l'affichage là où `inspect` est majoritairement utilisé pour du débogage.

III.1.3.2. Comparaison

Ici, nous allons faire un retour sur la comparaison d'objets. En effet, une question que nous pouvons nous poser et à laquelle il est important de répondre est comment comparer deux objets. Deux nombres sont égaux s'ils sont les mêmes, deux chaînes de caractères sont égales si elles ont les mêmes caractères, deux tableaux sont égaux s'ils ont les mêmes éléments, mais à quel moment deux objets que l'on vient de créer sont égaux?

```
a = Object.new
b = Object.new
a == b          # => false
```

Ici, on obtient `false`. Pourtant ces deux objets sont exactement la même chose. On pourrait penser que c'est parce que ce sont deux objets distincts. Cette réflexion n'est pas bête, et c'est en effet à cause de ça. Mais dans ce cas, le résultat obtenu avec les chaînes posent problème.

```
a = 'abc'
b = 'abc'
a == b    # => true
a.object_id == b.object_i # false
```

Ici, `a` et `b` ne font pas référence au même objet puisqu'ils n'ont pas le même identifiant.

En fait, pour vérifier que deux objets sont égaux nous utilisons juste… une méthode. Écrire `a == b` est strictement équivalent à écrire `a.== b`. Et donc, puisque c'est une méthode, on peut redéfinir `==`. Par exemple, on peut redéfinir la méthode d'un message.

```
def message.==(other)
  message.content == other.content
end
```

La méthode `==` prend en paramètre un autre message et retourne `true` si les contenus des deux messages sont égaux.

En fait, nous disposons également d'autres manières de tester si deux objets sont égaux.

III.1.3.2.1. Égalité d'identité

Pour commencer, on a l'égalité d'identité, associé à la méthode `equal?`. Elle permet de tester si deux objets ont la même identité.

```
a = 2
b = 2
a.equal?(b)                  # => true
a.object_id == b.object_id   # => true
a = [1, 2]
b = [1, 2]
a.equal?(b)                  # => false
a.object_id == b.object_id   # => false
```

III.1.3.2.2. Égalité sricte

L'égalité stricte, associée à la méthode `eql?`, permet de savoir si deux objets sont strictement les mêmes. Par exemple, `1 == 1.0` vaut `true` mais `1.eql?(1.0)` vaut `false`, car les deux objets ne valent pas strictement la même chose.

Cette méthode est rarement utilisée, parce que généralement `==` nous suffit. En pratique, cette comparaison stricte est rarement utile.

III.1.3.2.3. Égalité du `case`

L'égalité du `case` associée à la méthode `===` est présente quand on utilise `case`. En effet, `(1..10) === 7` renvoie `true`. On peut alors écrire ces deux codes équivalents.

```
case hour
when 0..6
  print 'Nuit.'
when 7..12
  print 'Matin'
when 12..18
  print 'Après-midi.'
when 18..24
  print 'Soir.'
else
  print 'Euh, chez nous les journées ont 24 heures'
end
```

```ruby
if (0..6) === hour
  print 'Nuit.'
elsif (6..12) === hour
  print 'Matin'
elsif (12..18) === hour
  print 'Après-midi.'
elsif (18..24) === hour
  print 'Soir.'
else
  print 'Euh, chez nous les journées ont 24 heures'
end
```

Le symbole `===` n'est quasiment jamais utilisé et son utilisation est à éviter. Conformément à son nom, il ne sera utilisé que quand on utilisera des `case` et son utilisation sera donc implicite (nous ne l'écrirons pas).

III.1.3.2.4. Redéfinir toutes ces égalités ?

En fait, tout ce que nous venons de dire là est relatif à l'objet considéré puisque nous pouvons redéfinir ces méthodes comme nous l'avons fait plus haut avec le `==` de notre message. Ainsi, nous pouvons donner à ces égalités la signification que l'on veut. Néanmoins, nous resterons raisonnables et respecterons ces quelques règles.

— Généralement, on ne redéfinit pas `equal?`.
— On peut redéfinir `===` conformément à l'utilisation qu'on fait de l'objet dans nos conditions et donc avec `case`.

On rajoute à ces règles celles que l'on a vu plus haut, ce qui nous donne ces règles d'utilisation.

— On n'utilise pas `===` explicitement.
— On utilise rarement `equal?`.
— On privilégie `==` à `eql?` à moins que l'on ait vraiment besoin de `eql?`.

En résumé, sauf cas particulier, on ne s'occupe que de `==`.

III.1.3.3. Envoi de message

En Ruby, plutôt que de parler d'appel de méthode comme dans beaucoup d'autres langages, on préfère parfois parler d'**envoi de message**. Le point est alors l'opérateur d'envoi de message. Quand on écrit `message.auteur`, on envoie alors un message à l'objet `message` pour lui demander son auteur. Le message, à droite de l'opérateur d'envoi de message, est envoyé à l'objet, situé à sa gauche.

En fait, ce nom se justifie encore plus lorsqu'on utilise les méthodes `public_send`, `__send__` ou `send`. Tout comme `respond_to?`, ces fonctions prennent en paramètre un symbole ou une chaîne de caractère, et demandent à l'objet d'appeler la méthode qui y est associé.

```
message.public_send(:author)
```

Cette ligne se comprend aisément comme «envoyer à `message` le message `author`».

Il y a quelques différences entre les trois méthodes vues. Si nous devons les utiliser, nous préférerons l'utilisation de `public_send` (pour des raisons que nous verrons plus tard). Dans les cas où, elle ne peut pas être utilisée (là encore nous verrons ces cas plus tard), nous préférerons `__send__` pour deux raisons.

1. Certains objets redéfinissent `send` (par exemple, on pourrait redéfinir la méthode `send` de notre `message` pour qu'elle envoie le contenu du message à son destinataire.
2. Certains objets n'implémentent pas la méthode `send`.

On peut alors écrire ce genre de code grâce à ces méthodes (on demande à l'utilisateur un nom de méthode et on envoie ce message à l'objet).

```
print('Quelle information à propos du message voulez-vous ? ')
request = gets.chomp.to_i
print message.public_send(request) if message.respond_to?(request)
```

Tout ceci est par exemple utilisé pour faire de la **métaprogrammation** (en gros, écrire du code qui écrit lui-même du code).

?

Nous avons dit plus haut qu'en Ruby, tout était objet, et que les méthodes servaient à manipuler les méthodes. Qu'en est-il de `puts`? N'est-ce pas une vraie méthode?

En fait, lorsque nous écrivons `puts`, Ruby comprend `$stdout.puts`. `$stdout` est une variable globale représentant la sortie standard (*standard output*). En écrivant `puts`, on demande donc un affichage sur la sortie standard (par défaut la console). Notons également l'existence de l'entrée standard `$stdin` (*standard input*) qui nous permet d'écrire `$stdin.gets`.

Il existe également des constantes, `STDOUT` et `STDIN`, qui représentent la sortie et l'entrée standard. La différence avec `$stdin` et `$stdout` est que même si on modifie ces deux derniers, on pourra toujours afficher notre message sur la console avec `STDOUT.puts`. Pour le moment, cela ne nous est guère utile, mais nous apprendrons plus tard à rediriger `$stdout` et `$stdin` (par exemple vers des fichiers).

III.1.4. Exercices

Pas de vrai exercices pour ce chapitre, pour nous exercer, nous allons juste créer des objets et les méthodes pour communiquer avec eux. Le plus simple est de rester dans IRB et de jouer avec quelques objets.

Conclusion

Dans ce chapitre très long et très général, nous avons appris beaucoup (vraiment beaucoup) de choses. Tout n'est pas à retenir par cœur, mais il ne faut pas perdre ces informations de vue, elles pourront nous être utiles par la suite.

— En Ruby, tout est objet et donc il n'y a que des méthodes. Même `puts` et `gets` sont des méthodes (l'objet auquel elles s'appliquent est juste implicite).
— On peut créer un objet avec `Object.new`. On peut ensuite lui rajouter des méthodes à la volée (ce qu'on peut aussi faire avec la plupart des objets existants).
— On peut inspecter un objet (connaître ses méthodes, savoir si une méthode existe). Ces opérations relèvent de ce que l'on appelle l'introspection.

III.2. Les classes

Introduction

Dans ce chapitre, nous allons compléter nos objets en les rendant moins statiques et nous allons découvrir un moyen de les créer plus facilement.

III.2.1. Attributs

Nos objets sont quand même bien triste pour le moment. Nous ne pouvons même pas changer le contenu d'un message sans redéfinir la méthode `content`. C'est parce que pour le moment, nous avons laissé de côté un aspect important des objets: un objet a des **propriétés**, des **attributs**. Si les méthodes servent à manipuler l'objet, ses propriétés le caractérisent. Par exemple, si nous reprenons l'exemple du chapitre précédent, l'auteur, le destinataire, et même le contenu d'un message ne servent pas à manipuler le message mais sont des attributs du message.

Ces propriétés sont représentées sous la forme de ce que l'on appelle des **variables d'instances** (nous verrons pourquoi elles portent ce nom dans la section suivante). Une variable d'instance est simple à déclarer, son nom est précédé du symbole `@`. Par exemple, on pourrait modifier notre objet `message` ainsi.

```
1 m = Object.new
2 def m.change_content(new_content)
3   @content = new_content
4 end
5
6 def m.to_s
7   @content
8 end
```

On peut alors modifier le contenu d'un message bien plus facilement.

```
1 m.change_content('Contenu du message.')
2 puts m
3 m.change_content('Nouveau contenu.')
4 puts m
```

Les attributs nous permettent donc d'avoir des objets moins statiques.

III.2.1.1. Accesseurs et mutateurs

Dans des langages comme Java, on écrit des méthodes pour obtenir la valeur d'un attribut et pour la modifier. Ces méthodes ont des noms suivant le modèle `get_attribute` et `set_attribute`. Pour notre message par exemple, plutôt que `change_content`, on écrirait `set_content`.

```ruby
def m.set_content(content)
  @content = content
end

def m.get_content
  @content
end

# Et la même chose pour `author`, etc.
```

Les méthodes que nous venons de définir (pour obtenir un attribut ou donner une valeur à un attribut) sont ce que l'on appelle des accesseurs et des mutateurs. Par convention, on n'écrit pas l'accesseur `get_content`, mais plutôt `content` pour pouvoir écrire `puts m.content`. De même, on n'écrit pas le mutateur `set_content`, on fait en sorte de pouvoir écrire `m.content = other_content`.

?

Comment on fait pour écrire ce mutateur. La méthode `content` est déjà associée à l'accesseur, non?

En fait, lorsqu'on écrit `m.content = other_content`, c'est la méthode `content=` de l'objet `m` qui est appelée avec le paramètre `other_content`. Il nous faut donc écrire la méthode `content=`.

```ruby
def m.content
  @content
end

def m.content=(content)
  @content = content
end

# Et la même chose pour `author`, etc.
```

III.2.1.2. Conventions pour les méthodes

Nous allons maintenant nous pencher sur la conception des méthodes et sur quelques conventions de Ruby en la question.

III.2.1.2.1. Opérateurs

Nous avons déjà vu que nous pouvions définir les opérateurs de comparaison. Tous les autres opérateurs sont aussi des méthodes que l'on peut définir au besoin (et bien sûr `a + b` est équivalent à `a.+(b)`). Bien sûr, nous pouvons définir la méthode pour faire ce que l'on veut (on peut parfaitement définir la méthode `+` pour qu'elle affiche un message), mais nous allons rester à peu près cohérent et garder les opérateurs pour les opérations qu'elles semblent représenter. L'opérateur `+` sert alors à additionner pour des nombres, mais à concaténer pour les tableaux ou les chaînes de caractères.

S'il n'y en a pas besoin, nous n'allons pas définir ces méthodes (ainsi, les hachages n'ont pas de méthode `+`), mais quand nous en définissons un, nous appelons son paramètre `other` comme dans le chapitre précédent.

```
def message.==(other)
 message.content == other.content
end
```

Tout le long du tutoriel, nous verrons d'autres opérateurs, mais nous en connaissons déjà plusieurs. Listons-les ici.

Pour commencer, nous avons les opérateurs arithmétiques classiques (`+`, `-`, `*` et `/`) accompagné de `%` et `**`. Quand nous définissons par exemple l'opérateur `+` de l'objet `a`, nous pouvons écrire `a += b`, c'est la méthode `a` qui sera appelée. Attention, il ne faut pas oublier que le résultat de l'opération est ensuite affecté à `a`. Cela peut mener à des problèmes.

```
a = Object.new

def a.+(other)
  other
end

a += 3
# Maintenant, `a` vaut 3.
```

Puis, pour comparer, nous avons les opérateurs de comparaison que nous avons vu dans le chapitre précédent (`==`, `===`) sans oublier `<`, `>`, `<=` et `>=`.

Et puis, il y a des opérateurs que nous utilisons pour manipuler les tableaux et les chaînes de caractères. Oui, `[]` et `<` sont bien des opérateurs. Si nous en avons besoin pour un objet, il ne faut pas hésiter à les définir. Le cas de `[]` est un peu spécial dans le sens où il sert à accéder à un élément, mais aussi à modifier un élément.

Ce problème se règle de la même manière que celui posé par les accesseurs et mutateurs, nous définissons la méthode `[]` et la méthode `[]=`, `[]=` étant appelé quand on écrit `a[0] =` . Le nombre entre crochets correspond au paramètre qui est passé à la méthode.

```ruby
a = Object.new

def a.[](i)
  print "On a écrit « a[#{i}] »."
end

a[3]
```

Remarquons qu'ici, le paramètre n'a pas été nommé `other`. `[]` et `<` sont deux exceptions à cette règle car leur sémantique est différente de celle des autres opérateurs.

III.2.1.2.2. Méthodes retournant un booléen

Au niveau des méthodes renvoyant un booléen, la convention est très simple, elle se finit par un point d'interrogation. De plus, il est d'usage de ne pas utiliser de verbes inutiles à la compréhension. Par exemple, pour vérifier qu'un message est vide, plutôt que de définir une méthode `is_empty`, nous allons définir la méthode `empty?`.

```ruby
def m.empty?
  @contenu == ''
end
```

III.2.2. Un modèle de construction

Pour le moment, créer des messages est assez rébarbatif. Nous devons à chaque fois créer les mêmes méthodes pour chaque objet créé. Ce serait bien de les avoir dès le départ, tout comme nous avons par exemple une méthode `each` dès que nous créons un tableau. Pour ce faire, nous allons créer une classe. Nous en avons déjà parlé dans le chapitre précédent en disant qu'elles étaient des modèles sur lesquelles les objets étaient construits.

III.2.2.1. Création de classe

Nous pouvons alors créer une classe `Message` et les objets de cette classe auraient déjà les méthodes nécessaires dès leur création.

La syntaxe pour créer une classe est aussi simple que le reste du langage. Créons notre classe `Message`.

```ruby
class Message
end
```

On utilise le mot-clé `class` suivi du nom de la classe. Le mot-clé `end` sert à indiquer la fin de notre classe.

Les noms des classes sont par convention écrit en «Camel_Case » avec la première lettre en majuscule.

Ça y est, nous avons créé notre première classe. Elle est vide, mais nous pouvons déjà créer des objets avec eux. Cette fois, nous n'allons pas utiliser la méthode `new` de `Object`, mais la méthode `new` de `Message` pour indiquer que l'on veut créer un nouveau message.

```
1 m = Message.new
```

III.2.2.1.1. Un peu de vocabulaire

En Ruby, chaque objet a une classe (avec `Object.new`, on créait un objet de classe `Object`). Lorsque nous créons un objet d'une certaine classe, on dit que l'on crée une **instance de la classe** ou que l'on **instancie la classe**. L'objet créé est une **instance** de la classe, et l'opération de création s'appelle une **instanciation**.

C'est de cette opération que vient l'expression «**variable d'instances**» rencontrée plus haut, une variable d'instance étant une variable d'une instance d'une classe.

En fait, les classes peuvent être vu comme des modèles, mais aussi comme des types. Chaque objet a une classe, cette classe représente son type. Ainsi, le type d'un message est `Message`. Nous avons bien entendu la possibilité de savoir de quelle classe est un objet grâce à la méthode `class`.

```
1 m = Message.new
2 m.class # => Message
```

Nous pouvons aussi regarder la classe d'objets usuels.

```
1 10.class  # => Integer
2 [].class  # => Array
3 ''.class  # => String
4 {}.class  # => Hash
```

Ceci renforce l'idée que les classes sont comme des types. Un tableau est de la classe `Array`, une chaîne de caractères de classe `String`, etc.

III.2.2.2. Des méthodes

Nous avons dit que les classes nous permettaient de ne pas réécrire les méthodes pour chaque objet. Pour cela, nous déclarons tout simplement la méthode à l'intérieur de la classe. Notre classe `Message` est donc la suivante.

```
class Message
  def content=(content)
    @content = content
  end

  def to_s
    @content
  end

  # And the other methods.
end
```

Et on peut maintenant utiliser nos objets comme on le souhaite.

```
m1 = Message.new
m2 = Message.new
m1.content = 'Voici un message.'
print m1
m1 == m2
m2.content = 'Voici un message.'
m1 == m2
```

Les méthodes que nous déclarons comme cela sont appelés **méthodes d'instances**. Chaque instance de la classe dispose de ces méthodes. Au contraire, lorsque nous définissons une méthode pour un objet seul, comme nous l'avons fait dans le chapitre précédent, on parle de **méthode singleton**.

III.2.2.2.1. Rajouter des méthodes à la volée

Tout comme nous pouvons rajouter des méthodes à un objet, nous pouvons rajouter des méthodes à une classe. Imaginons par exemple que nous avons déjà notre classe `Message` et que pour les messages qui suivent, on veut rajouter une méthode `erase` qui supprime le contenu. Il nous suffit d'écrire ça.

```
class Message
  def erase
    @content = ''
```

```
4     end
5   end
```

En faisant ça, nous n'avons pas défini une nouvelle classe, nous avons modifié la classe en lui ajoutant la méthode `supprimer`. Lorsque nous faisons cette opération, nous **ouvrons la classe**. En Ruby, les classes sont ouvertes. Lorsque nous ouvrons une classe, les objets déclarés avant cette classe ne sont pas modifiés et ne disposent pas des méthodes créées lors de l'ouverture (normal, ils n'ont pas été créés avec la classe modifiée).

L'ouverture de classe n'est pas très utilisée sur les classes créées par le développeur, mais elle peut être très utile sur les classes pré-définies. Par exemple, nous pourrions définir une méthode sur la classe `String` pour mélanger une chaîne de caractères.

III.2.2.2.2. Méthodes pour les accesseurs et mutateurs

Ruby nous fournit des méthodes pour faciliter la définition d'accesseurs et de mutateurs. Elles prennent en paramètre des symboles (ou des chaînes de caractères), autant que l'on veut, et créent l'accesseur ou le mutateur associé à ce symbole. `attr_writer` (pour *attribute writer*) crée le mutateur, `attr_reader` (pour *attribute reader*) crée l'accesseur et `àttr_accessor` (pour *attribute accessor*) crée les deux. Notre classe `Message` va alors se réécrire ainsi.

```
1  class Message
2    attr_reader :author, :date, :recipient, :content
3
4    def to_s
5      @contenu
6    end
7
8    # And the other methods.
9  end
```

Ici, tout a été placé en «lecture seule», l'idée étant qu'une fois qu'un message a été écrit et envoyé, sa date, son destinataire, son contenu et son auteur sont fixés. Ils ne peuvent pas être modifiés. Bien sûr, ce n'est qu'une vision personnelle et on pourrait très bien imaginer une classe où tous ces attributs, ou une partie d'entre eux, sont modifiables.

III.2.2.3. Initialisation d'objets

Nous avons maintenant des objets plus personnalisables et nous savons le manipuler. Mais nous ne savons pas **créer** l'objet comme nous le voulons. Nous ne pouvons même pas donner de valeurs aux attributs dès sa création. Ceci pose de plus un gros problème, nous ne pouvons pas donner de valeurs aux attributs en lecture seule. Si nous créons un message avec la dernière classe `Message` que nous avons créé nous ne pouvons pas modifier ses attributs.

```ruby
m1 = Message.new
print m1.content
m1.content = 'Nouveau contenu.' # Erreur content= n'est pas défini
```

Il nous faut **initialiser** notre objet. Cela se fait avec la méthode `initialize`. Elle est appelée automatiquement à la création de l'objet.

```ruby
class Message
  attr_reader :author, :date, :recipient, :content

  def initialize
    @content = 'Contenu'
    @date = '19/19/1919'
    @recipient = 'Destinataire'
    @author = 'Auteur'
  end
end
```

Testons la nouvelle classe.

```ruby
m = Message.new
print "Écrit par #{m.auteur} pour #{m.destinataire} le #{m.date}."
```

La méthode `initialize` est une méthode qu'on appelle **constructeur** de la classe, parce qu'elle permet de construire l'objet tout simplement. Nous avons écrit un constructeur sans paramètre, mais il est tout à fait possible d'en créer un avec paramètres (et il est bien là l'intérêt).

```ruby
class Message
  attr_reader :author, :date, :recipient, :content

  def initialize(content, author, recipient, date = '21/12/2112')
    @content = content
    @date = date
    @recipient = recipient
    @author = author
  end
end

m1 = Message.new('Un message', 'Moi', 'Le monde')
m2 = Message.new('Un autre message', 'Moi', 'Quelqu'un',
    '24/04/2024')
```

III.2.3. Du nouveau sur les classes ?

III.2.3.1. Travailler avec une classe

Nous pouvons créer des instances de classe, mais nous pouvons aussi travailler directement avec la classe.

III.2.3.1.1. Constantes de classe

Pour commencer, nous pouvons déclarer des constantes dans les classes, la portée de ces variables étant la classe. Cela permet de définir une constante qui sera utilisée dans le code. Nous pouvons par exemple ajouter une constante `LIVREUR` à notre classe `Message`. C'est Zeste de Savoir qui livre les messages.

```
class Message
  LIVREUR = 'ZdS'

  def livreur
    LIVREUR
  end

  # etc.
end

m = Message.new('Un message', 'Moi', 'Le monde')
m.livreur # => 'Zds'
```

Et, en fait, on peut avoir accès aux constantes de classe en dehors de la classe grâce à l'opérateur `::`.

```
print "Le livreur est #{Message::LIVREUR}."
```

III.2.3.1.2. Variables de classes

Les constantes c'est bien, mais nous pourrions peut-être vouloir avoir des variables. Par exemple, pour ne pas mélanger les messages, nous allons leur ajouter un identifiant. Pour avoir des identifiants différents, nous allons utiliser une variable dans notre classe. Elle sera initialisée à 0 et incrémentée à chaque nouveau message. Il nous suffira alors de donner cette valeur à l'identifiant dans la méthode `intialize`.

Malheureusement, utiliser une variable normale ne fonctionnera pas, nous obtiendrons une erreur disant que la variable n'est pas définie. Pour faire ce travail, il nous faut utiliser une **variable de classe**. Si les variables d'instances ont leurs noms précédés du symbole `@`, les variables de classes ont leurs noms précédés du symbole `@@`.

```ruby
class Message
  @@messages_number = 0

  attr_reader :author, :date, :recipient, :content, :id

  def initialize(content, author, recipient, date = '21/12/2112')
    @content = content
    @date = date
    @recipient = recipient
    @author = author
    @id = (@@messages_number += 1)
  end

  # etc.
end

m1 = Message.new('Contenu', 'Auteur', 'Destinataire')
Message.new
m2 = Message.new('Contenu', 'Auteur', 'Destinataire')
puts "m1 est le message #{m1.id}."  # => id = 1
puts "m2 est le message #{m2.id}."  # => id = 3
```

Les variables de classes sont à éviter car elles peuvent s'avérer dangereuses. Nous verrons pourquoi dans les chapitres suivants, mais il nous faut savoir que c'est en partie parce qu'elles peuvent être modifiées là où on ne s'y attend pas.

Nous allons donc les éviter dans la plupart des cas. Ici, son usage est simple et ne pose pas de problèmes

III.2.3.1.3. Méthodes de classes

Finalement, nous allons parler des méthodes de classes. Si les méthodes d'instance sont les méthodes que les instances possèdent, les méthodes de classe sont les méthodes de la classe (comme `new`). Pour déclarer une méthode de classe, nous allons utiliser le mot-clé `self`. Par exemple, définissons une méthode pour avoir accès au nombre de messages écrits.

```ruby
class Message
  @@messages_number = 0

  attr_reader :author, :date, :recipient, :content, :id

  def self.messages_number
    @@messages_number
  end
```

```ruby

  # etc.
end

Message.messages_number
m = Message.new
m.nb_messages  # => Erreur
```

La méthode de classe n'est définie que pour la classe et pas pour ses instances. De même, lorsque nous déclarons une méthode d'instance, elle ne peut pas être utilisée par la classe.

Une classe est aussi un objet, donc on peut déclarer une méthode de classe singleton pour une classe.

```ruby
def Message.messages_number
  @@messages_number
end
```

III.2.3.2. Visibilité

Tout comme nous avons vu que les variables avaient une portée, les méthodes ont ce qu'on appelle une **visibilité**. On ne les «voit» pas partout c'est-à-dire qu'on ne peut pas les utiliser partout. En Ruby, il y a trois types de visibilité, représenté par les méthodes `public`, `private` et `protected`.

L'utilisation d'une de ces méthodes sans argument change la visibilité de toutes les méthodes déclarées à partir de là (donc jusqu'à ce qu'on utilise une autre méthode qui change cela). L'utiliser dans la classe change donc la visibilité de toutes les méthodes déclarées à partir de là.

```ruby
class Example
  private
  # Les méthodes sont maintenant privées.

  public
  # Les méthodes sont maintenant protégées.
end
```

On peut également utiliser ces méthodes après avoir écrit `def nom`. En agissant comme cela, nous agissons seulement sur la visibilité de la méthode `nom`.

```ruby
class Example
  def example_method private
  end
end
```

Cette syntaxe n'est pas trop utilisée (et nous ne l'utiliserons pas), mais elle peut être croisée à l'occasion donc il faut l'avoir déjà vue.

Et finalement, il est possible d'utiliser ces méthodes de la même manière que `attr_reader` et compagnie, en leur donnant en argument les méthodes que l'on veut déclarer sous la forme de symboles ou de chaînes de caractères.

```ruby
class Example
  def example_method
  end

  private :example_method
end
```

Maintenant, nous savons changer la visibilité des méthodes de trois manières différentes… Mais nous ne savons toujours pas à quoi correspondent ces visibilités. Il faudrait quand même le savoir.

III.2.3.2.1. Les méthodes publiques

Une méthode est dite publique si on peut l'utiliser partout. Par défaut, les méthodes que nous déclarons dans une classe sont publiques.

III.2.3.2.2. Les méthodes privées

Une méthode est dite privée si on peut l'utiliser uniquement dans les autres méthodes de l'objet. Cela permet d'empêcher d'accéder à cette méthode depuis l'extérieur. Par exemple, rendons privée la méthode `erase` de nos messages et rajoutons une méthode `delete_content` qui prend en paramètre une chaîne de caractère et efface le contenu du message uniquement si cette chaîne de caractère correspond à l'auteur.

```ruby
class Message
  # Méthodes précédentes

  def delete_content(name)
    name == author ? erase : puts 'Ce n'est pas le bon auteur.'
  end

```

```ruby
  private

  def erase
    @content = ''
  end
end

m = Message.new('Contenu', 'Auteur', 'Destinataire')
m.delete_content('Auteur')  => OK.
m.erase                     => Erreur.
```

Notre méthode `delete_content` peut appeler `erase`, mais on ne peut appeler `erase` en dehors de la classe. De plus, quand une méthode est privée, on ne peut l'appeler que pour l'objet courant. Ainsi, on ne peut pas appeler la méthode `erase` d'un message `m1` depuis un autre message `m2`. On peut par exemple imaginer une méthode fantaisiste `clean` qui prend en paramètre un autre message et supprime son contenu s'ils sont de même auteur.

```ruby
class Message
  # Méthodes précédentes

  def clean(other)
    other.erase if other.author == @author
  end

  private

  def erase
    @content = ''
  end
end

m1 = Message.new('Contenu', 'Auteur', 'Destinataire')
m2 = Message.new('Autre contenu', 'Auteur', 'Destinataire')
m2.erase(m1)  # => Erreur.
```

III.2.3.2.3. Les méthodes protégées

Les méthodes protégées sont un peu moins strictes que les méthodes privées et permettent d'écrire la méthode `erase`. Si une méthode est protégée, n'importe quel objet de la classe peut appeler cette méthode. En faisant de `erase` une méthode protégée, on peut écrire ce code qui ne fonctionnait pas quand elle était privée.

```ruby
m1 = Message.new('Contenu', 'Auteur', 'Destinataire')
m2 = Message.new('Autre contenu', 'Auteur', 'Destinataire')
```

```
3 m2.erase(m1)  # C'est OK.
```

Pour ne pas mélanger les différentes choses définies dans les classes, nous allons adopter la structure suivante. Les constantes, puis les accesseurs et mutateurs, puis les méthodes de classe publiques, puis la méthode `initialize`, puis les autres méthodes d'instances publiques et les méthodes privées et protégées viennent à la fin.

Contenu masqué n°23

III.2.3.3. Interface

Parlons un peu de conception maintenant. Nous allons appeler «interface» l'ensemble des méthodes publiques d'un objet, c'est-à-dire la face que cet objet nous présente pour le manipuler. Nous pourrions d'abord penser à mettre toutes nos méthodes publiques et à créer au moins des accesseurs pour toutes nos variables d'instances. Pourtant c'est une mauvaise idée.

Ce que nous allons faire au contraire, c'est limiter les méthodes publiques et les accesseurs au strict minimum, donc à ce que l'utilisateur de la classe a besoin de savoir. Imaginons par exemple une classe `Chat`. Un chat sait faire le beau (comment ça, ce sont les chiens qui font le beau), dormir, et manger. Ça nous fait autant de méthodes.

Pour savoir s'il faut lui donner à manger, ou s'il doit dormir, nous allons lui attribuer des attributs `estomac` et `énergie`, des entiers de 0 à 10. Quand ces entiers sont inférieurs à 4, le chat mange ou dort. On a donc le code suivant.

```
1  class Cat
2    attr_reader :, :energy, :name
3
4    def initialize(nom)
5      @estomac = 7
6      @energy = 7
7    end
8
9    def eat
10     puts 'Le chat mange.'
11     @estomac += 6
12     @energy -= 2
13   end
14
15   def sleep
16     puts 'Le chat dort.'
17     @energy += 6
```

```
18      @estomac -= 2
19    end
20
21    def act
22      puts 'Le chat fait le beau.'
23    end
24  end
25
26  c = Cat.new('Sylvestre')
27  c.eat if c.estomac < 4
```

Le problème: `estomac` et `energy` font partie du fonctionnement interne de `Chat`, on ne devrait pas savoir combien d'énergie il a. De plus, imaginons que nous changeons de technique, nous décidons que le chat a faim s'il n'a pas mangé depuis un certain nombre d'heures. Il nous faudrait changer tous les endroits où nous utilisons `estomac`; c'est ballot. Ici, ça va, on ne l'a utilisé qu'une fois, mais dans un vrai code, ça ferait mal.

La solution: définir une méthode `hungry?` et une méthode `tired?` et supprimer les accesseurs `estomac` et `energy`. L'utilisateur de la classe n'a pas à connaître ces informations.

```
1   class Cat
2     attre_reader :name
3
4     def hungry?
5       @estomac < 4
6     end
7
8     def tired?
9       @energy < 4
10    end
11
12    # etc.
13  end
14
15  c = Cat.new('Tom')
16  c.sleep if c.tired?
17  c.eat if c.hungry?
```

Nous pouvons remarquer qu'il manque par exemple une méthode `asleep?` pour ne pas le faire manger pendant qu'il dort ce qui serait absurde (quoique, Garfield le fait bien, lui), mais c'est une autre histoire.

i

Remarquons que cela nous permet également de respecter la Loi de Démeter (l'histoire du gars qui donnait son pantalon au vendeur pour payer sa canette). Ici, les utilisateurs de la classe `Cat` ont juste besoin de connaître son état (s'il est fatigué ou affamé), mais ils n'ont pas du tout besoin de connaître son estomac!

III.2.4. Exercices

Comme exercice, nous allons créer une classe `Duration` pour représenter une durée. Nous pouvons additionner des durées, les multiplier ou les diviser par des nombres (par exemple, cinq fois une durée de 2 heures). Il nous faut bien sûr une manière de les afficher et de les comparer. Dans notre implémentation, nous pourrons choisir d'autoriser ou non les durées négatives. Il nous faudra alors faire attention à comment nous gérons les multiplications et divisions par un nombre négatif et la soustraction de durée.

Correction.

Contenu masqué n°24

Bien sûr, ce n'est pas la seule implémentation possible et on pourrait penser à d'autres méthodes pour notre classe `Duration`.

Conclusion

C'est la fin d'un chapitre encore une fois long, et avec beaucoup de théories. Qu'avons-nous appris?

- Nos objets ont des attributs qu'on manipule sous la forme de variables d'instances.
- Les méthodes pour les opérateurs et les conversions doivent être cohérentes.
- Les classes permettent de créer des objets avec des méthodes prédéfinies. Ces objets peuvent être initialisés plus facilement.
- L'interface de nos objets ne doit donner accès qu'au nécessaire. Pour cela, il ne faut définir que les accesseurs et les mutateurs nécessaires et donner aux méthodes la visibilité qui leur convient.

! Ruby est souvent comparé à Python, mais sur le point de la visibilité, ils sont complètement différents. Là où en Python on suit la philosophie «nous sommes entre adultes consentants», en Ruby on cache ce qu'il y a à cacher.

Contenu masqué

Contenu masqué n°23

```ruby
class ClassExample
  CONSTANT = 20

```

```ruby
  attr_reader :variable

  def self.class_method
  end

  def initialize
  end

  def instance_method
  end

  protected

  def protected_method
  end

  private

  def private_method
  end
end
```

Retourner au texte.

Contenu masqué n°24

Ici, on a des méthodes pour:

— additionner, soustraire, comparer des durées;
— multiplier et diviser des durées par des nombres;
— convertir une durée en secondes, minutes, heures;
— convertir une durée en chaîne de caractères ou en tableau.

On gère les durées négatives et pour continuer à illustrer le concept d'interfaces, on ne gère la durée qu'avec les secondes en attribut, mais on a accès à des méthodes pour connaître le nombre d'heures, le nombre de minutes et le nombre de secondes (et même le constructeur prend en paramètre ces trois choses. L'utilisateur de la classe, lui n'a pas besoin de savoir qu'en fait seul un nombre de secondes est stockée dans la classe. Ce qui l'intéresse, ce sont les services auxquels il a accès.

```ruby
# Un petit exemple de classe.
class Duration
  def initialize(hours, minutes, seconds)
    @seconds = (seconds + minutes * 60 + hours * 3600).to_i
  end

  def +(other)
```

```ruby
    Duration.new(0, 0, other.seconds + @seconds)
  end

  def -(other)
    Duration.new(0, 0, @seconds - other.seconds)
  end

  def -@
    Duration.new(0, 0, -@seconds)
  end

  def *(other)
    Duration.new(0, 0, @seconds * other)
  end

  def /(other)
    Duration.new(0, 0, @seconds / other)
  end

  def ==(other)
    @seconds == other.seconds
  end

  def >(other)
    @seconds > other.seconds
  end

  def >=(other)
    self > other || self == other
  end

  def <(other)
    self < other
  end

  def <=(other)
    self <= other
  end

  def positive?
    @seconds > 0
  end

  def h
    @seconds / 3600
  end

  def m
    (@seconds % 3600) / 60
  end
```

```ruby

  def s
    (@seconds % 60)
  end

  def to_s
    "#{h}:#{m}:#{s}"
  end

  def to_a
    [h, m, s]
  end

  def to_second
    @seconds
  end

  def to_minute
    @seconds.to_f / 60
  end

  def to_hour
    @seconds.to_f / 3600
  end

  protected

  attr_reader :seconds
end

a = Duration.new(0, 0, 3686)
b = Duration.new(0, 0, 59)
c = 3
puts a
puts a + b
puts a - b
puts a * c
puts a / c
```

Retourner au texte.

III.3. Les modules

Introduction

Dans ce chapitre, nous allons continuer sur la lancée du chapitre précédente en voyant un nouvel outil pour regrouper et organiser du code.

III.3.1. Les modules

III.3.1.1. Création de module

Un module est une structure regroupant à la fois des constantes et des méthodes. On peut alors regrouper des éléments qui partagent un thème commun dans une même structure. On peut par exemple imaginer un module de messagerie dans lequel on mettrait des méthodes ayant un lien avec notre classe `Message` ou encore un module mathématique contenant divers outils mathématiques.

Un module se construit de cette façon.

```
1 module Football
2 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 `Football`). 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.

Dans le module, on pourra ensuite définir des méthodes et des constantes (un peu comme dans une classe). Comme un module ne s'instancie pas (on ne crée pas d'objets à partir du «moule» du module), on ne va pas définir de méthodes d'instances, mais des méthodes de module (un peu comme des méthodes de classe), en utilisant `def self.méthode`.

Les méthodes d'instance de module ne sont pas inutiles et nous pouvons les croiser dans plusieurs codes Ruby. Elles servent notamment à créer ce que l'on appelle des **mixins** et que nous verrons au prochain chapitre.

Néanmoins, plutôt que d'écrire `self.fonction`, nous allons préférer utiliser `module_function` qui s'utilise comme `private` par exemple, et permet d'indiquer que les méthodes qui suivent sont des méthodes de module. Utiliser `module_function` n'est pas tout à fait similaire à utiliser `self`, mais ces subtilités ne nous intéressent pas pour le moment.

```
1  module Multiplication
2
3    PLAYERS_NB = 11
4
5    module_function
6
7    def rules
8      puts 'On a demandé les règles du football.'
9    end
10 end
```

Ici, nous avons défini la constante `MAX` et la méthode `table`. 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 `nom_méthode`.

```
1  module Multiplication
2
3    PLAYERS_NB = 11
4
5    module_function
6
7    def rules
8      puts 'On a demandé les règles du football.'
9      puts "C'est un sport qui se joue à #{PLAYERS_NB}."
10     puts '...'
11   end
12
13   def presentation
14     puts 'Le football est un sport d'équipe.'
15     puts 'Voici ses règles.'
16     rules
17   end
18 end
```

III.3.1.2. Des espaces de noms

Jusqu'à maintenant un module ressemble beaucoup à une classe (sauf qu'il n'y a pas d'attributs). ET l'utilisation ne fait pas exception. Pour utiliser une méthode d'un module, on écrit `NomMo`

`dule.nom_méthode`. Par exemple, pour utiliser la méthode `table` du module `Multiplication` défini précédemment, nous allons utiliser ce code.

```
print Football.rules
```

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.

```
print Football::PLAYERS_NB
```

Notons que cette syntaxe fonctionne également pour utiliser les méthodes du module (et pour les méthodes de classe). 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`.

```
puts "Au football il y a  #{Football::PLAYERS_NB} joueurs."
Football.rules                          # Bonne écriture.
Football::rules                         # Écriture déconseillée.
```

En fait, un avantage des modules, en plus de permettre un regroupement thématique est que nous pouvons avoir dans chaque module une méthode ou un objet qui ont le même nom sans que cela ne pose de problème. Par exemple, on pourrait tout à fait imaginer une méthode `rules` dans un autre module `Handball`.

```
module Handball

   PLAYERS_NB = 7

  module_function

  def rules
    puts 'Voici les règles du handball.'
    puts '...'
  end
end

puts 'Dans le module Football.'
Football.rules
puts 'Dans le module Handball.'
Handball.rules
```

Les modules permettent alors de former ce que l'on appelle des **espaces de noms** (*namespace* en anglais). Un espace de noms est comme un tiroir dans lequel on met des objets (méthodes, constantes, etc.). En appelant `Fooball.rules`, on appelle la méthode `rules` du tiroir `Football`, et donc il n'y a pas de risques de la confondre avec celle du tiroir `Handball`.

III.3.1.3. Déverser le tiroir

Notre programme est une chambre, les modules sont des tiroirs. Ainsi, nous avons un tiroir avec tous les outils du football (et en particulier les règles) et un autre tiroir avec tous les outils du handball (dont les règles). Mais, comme tout enfant gaffeur, nous allons déverser notre tiroir dans la chambre.

? Euh, mais ça retire tout l'intérêt des espaces de noms, non?

Oui, mais dans certains cas, on sait qu'il n'y a pas de conflit possible et on veut écrire `rules` plutôt que `Football.rules`. Et dans ce cas, on voudra peut-être juste écrire `rules` plutôt que `Football.rules`.

Pour ce faire, nous allons utiliser la méthode `include` qui permet nous d'inclure un module dans l'«espace courant». On peut alors écrire ce code.

```
1 include Football
2
3 puts 'Dans le module Football.'
4 Football.rules
5 puts 'Comme on a inclus ce module.'
6 rules
7 puts 'Dans le module Menuiserie.'
8 Handball.rules
```

Et voilà, l'affaire est faite.

III.3.2. Le mot-clé self

Cela fait plusieurs fois que nous utilisons le mot-clé `self`. Il est temps de voir ce qu'il signifie.

III.3.2.1. À quoi se rapporte le mot-clé `self`

L'objet `self` se rapporte à l'objet courant et le représente. Par exemple, écrire `self.m` dans un module signifie que l'on définit la méthode `m` du module et c'est la même chose pour les classes. En fait, écrire `self.m` équivaut à écrire `ModuleName.m`.

Utiliser le mot-clé `self` plutôt que le nom du module est une bonne habitude. Elle permet par exemple de changer le nom du module sans se faire de souci. Nous allons donc adopter cette habitude.

Notons que puisque `self` fait référence à l'objet courant, écrire `self.m` revient à appeler la méthode `m` de l'objet courant. Dans une méthode de classe ou de module, cela revient à appeler la méthode `m` de la classe ou du module, dans une méthode d'instance, cela revient à appeler la méthode `m` de l'objet courant donc de l'instance de la classe.

Cela est très utile et permet d'utiliser l'objet et de le modifier. Par exemple, écrivons pour la classe `Array` une méthode `encrypt` qui ajoute à tous les éléments du tableau une certaine valeur (le code) fournit en paramètre. Pour cela, nous allons ouvrir la classe `Array` et lui ajouter la méthode.

```
1 class Array
2   def encrypt(code)
3     each_with_index { |x, i| self[i] += code }
4   end
5 end
6
7 tab = [2, 7, 8, 4]
8 tab.encrypt(9)
9 print tab
```

On obtient bien le résultat attendu (et on illustre au passage le fonctionnement de l'ouverture de classe).

Lorsque l'on écrit une méthode «dangereuse» ou qui modifie l'objet, il est d'usage de terminer le nom de la méthode par un point d'exclamation. On parle alors de méthode *bang*. Généralement on définit une méthode *bang* (celle qui change l'objet) et une méthode normale, la méthode normale étant définie en utilisant la méthode *bang* quand c'est possible (en utilisant `dup`). Pour notre exemple, on obtient ce résultat.

```ruby
class Array
  def encrypt!(code)
    each_with_index { |x, i| self[i] += code }
  end

  def encrypt(code)
    dup.encrypt!(code)
  end
end

tab = [2, 7, 8, 4]
puts "chiffrer retourne un tableau : #{tab.chiffrer(9)}."
puts "Mais tab n'a pas été modifié : #{tab}."
```

Ici, on utilise le fait que `each_with_index` (et `each`), renvoie le tableau. Ainsi, `crypt!` renvoie le tableau chiffré. En écrivant `dup.crypt!(code)`, on copie le tableau et on chiffre cette copie avec `crypt!`, le retour de `crypt!` sur cette copie (donc le tableau chiffré) étant renvoyé. Le tableau original n'est pas modifié.

III.3.2.2. Du `self` implicite dans les méthodes

Avec tout ce que nous venons de voir, nous devons remarquer qu'utiliser `m` revient à utiliser `self.m`. En effet, utiliser `m` dans une méthode d'instance revient à appeler la méthode `m` de l'instance de même que dans une classe ou un module ça revient à appeler la méthode de la classe, ou du module. En gros, cela revient à appeler la méthode de l'objet courant, donc à écrire `self.m`. Le `self` est donc **implicite**.

Il est donc équivalent d'écrire `m` et d'écrire `self.m`. Nous allons éviter d'utiliser `self`. Cela mène à un problème que nous n'avons jamais abordé jusqu'à maintenant.

Que se passe-t-il quand une fonction a le même nom qu'une variable?

La question est légitime ici. Si nous avons une méthode `f` et une variable `f`, que se passe-t-il quand on écrit `f`? Le mieux reste d'essayer, et là, on remarque que dans ce cas, c'est la variable qui est utilisée. En fait, pour utiliser la méthode `f`, nous sommes obligés d'utiliser des parenthèses.

```ruby
def f
  0
end

f = 1
puts f    # => 1.
puts f()  # => 0.
```

Pour éviter cela, nous allons éviter de nommer de la même manière des variables et des méthodes. Bien sûr, dans certains cas, ça reste plus pratique de les nommer de la même manières (et donner à nos variables et à nos méthodes une bonne portée aide à limiter ces problèmes de même que la bonne utilisation des modules et des classes). Par exemple, dans le cas de la méthode `initialize`, avoir les paramètres qui ont le même nom que les attributs (et donc que l'accesseur potentiel) ne pose pas beaucoup de problèmes et permet de garder un code lisible.

```ruby
class Message
  attr_reader :author, :date, :recipient, :content

  def initialize(content, author, recipient, date = '21/12/2112')
    @content = content
    @date = date
    @recipient = recipient
    @author = author
  end
end
```

Ici, les noms des paramètres ne posent pas problème, mais dans le cas général, il faut éviter de **masquer la méthode** avec une variable locale.

? Et pour les attributs, dans une méthode d'instance faut-il utiliser la variable d'instance ou les accesseurs et mutateurs potentiels?

Là, c'est un choix personnel. Le mieux est de s'en tenir à son choix. Par contre, si une variable n'a pas à être modifié, il ne faudra pas créer d'accesseurs ou de mutateurs (même en les mettant en privé), sous prétexte qu'on a décidé de ne pas utiliser les variables d'instances dans la classe. Par exemple, l'énergie de notre chat n'avait pas à être connue en dehors de l'objet, et nous n'avions pas créé d'accesseurs.

III.3.2.3. Retour sur `include` et `module_function`

Maintenant que nous savons à quoi correspond `self`, nous allons expliquer un peu mieux ce que font `include` et `module_function`. Et nous allons être un peu surpris.

En fait, `include` copie les méthodes d'instances du module pour les ajouter aux méthodes d'instances de l'objet courant. C'est pour cela que lorsque nous utilisons `include`, nous pouvons ensuite avoir accès aux méthodes d'un module. Et là, nous pouvons déjà voir un problème.

? Mais, nous avons dit qu'avec `module_function`, on indiquait qu'on définissait des méthodes de module. Si utiliser `module_funtion` est similaire à utiliser `self`, on ne définit jamais de méthodes d'instance dans le module et alors `include` ne copie aucune méthode, non?

Il faut bien croire que si. N'oublions pas, nous avons dit qu'utiliser `module_function` n'était pas tout à fait similaire à utiliser `self`. En fait, avec `module_function` on déclare deux méthodes, une méthode d'instance et une méthode de module. Avec `module_function` on serait alors plus proche de ça.

```
module Handball
   PLAYERS_NB = 7

  def self.rules
    puts 'Voici les règles du handball.'
    puts '...'
  end

  def rules
    puts 'Voici les règles du handball.'
    puts '...'
  end
end
```

Et ceci nous explique comment fonctionne `include`. Pour vérifier ce comportement, nous pouvons tester d'inclure un module, mais où la méthode a juste été définie en tant que méthode de module.

```
module Handball
   PLAYERS_NB = 7

  def self.rules
    puts 'Voici les règles du handball.'
    puts '...'
  end
end

include Handball

puts rules
```

Et on obtient une erreur «*undefined local variable or method 'rule' for main:Object (NameError)*». La méthode d'instance `rules` du module n'existait pas, et `include` n'inclut pas les méthodes de module.

III.3.2.3.1. Un peu de subtilité

Nous avons dit que la méthode `include` **copiait** les méthodes d'instances de son argument pour les rajouter aux méthodes d'instance de l'objet courant. En fait c'est faux, Ruby ne les copie pas, mais crée simplement une référence vers le module inclus. Cela signifie que tous les

objets qui incluent un module `M` ont une référence vers ce module `M` et qu'en particulier toute modification du module sera répercutée sur ces objets.

```ruby
module M
  module_function

  def f
    'Version 1 du module.'
  end
end

include M

puts f # => Version 1 du module.

module M
  module_function

  def f
    'Version 2 du module.'
  end
end

puts f # => Version 2 du module
```

Puisqu'il ne s'agit pas de copies mais bien de référence au module, nous n'avons même pas eu à l'inclure, la modification était déjà effective.

Nous allons éviter de modifier des modules de la sorte. Nous les déclarerons une fois, avec leurs méthodes et leurs constantes, et n'y toucherons plus.

III.3.3. Modules et classes

III.3.3.1. Classes ou modules ?

Les modules et les classes sont très liées et il est facile de confondre les deux. Pourtant les deux sont complètement différents dans le sens ou un module regroupe des objets alors qu'une classe représente vraiment quelque chose. D'ailleurs, on remarquera qu'un module ne peut pas être instancié.

```ruby
module M
end

```

```
m = M.new
# => undefined method `new' for M:Module
```

En fait, si nous avons juste besoin d'une structure pour regrouper des méthodes et des constantes, nous avons sûrement besoin d'un module et non d'une classe. De même, si nous commençons à vouloir faire une classe sans attributs, c'est sûrement un module qu'il nous faut. Généralement, il est plutôt simple de savoir qu'est-ce qui correspond à nos besoins.

De plus, les classes et les modules ne sont pas incompatibles comme nous allons le voir dans la suite du tutoriel.

III.3.3.2. De l'imbrication de classes et de modules

Une des premières manière de combiner les classes et les modules est de définir une classe dans un module. La syntaxe pour cela est plutôt naturelle.

```
module Football
  class Player
  end
end
```

Pour avoir accès à la classe, nous écrirons ensuite `Football::Player` (comme pour avoir accès une constante de modules). Ceci peut servir à avoir des espaces de noms pour les classes. On peut par exemple imaginer deux classes `Player`, la première dans un module de football, la seconde dans un module de handball. Reste à imaginer une application qui aurait besoin de ces deux classes, mais l'idée est là (en fait, même sans risque éventuel de conflits, on peut tout simplement vouloir que la classe soit liée à un espace de nom).

```
module Football
  class Player
  end
end

module Handball
  class Player
  end
end

player = Handball::Player.new
```

(ou exemple classe `Client` pour module `Network` et classe `Client` pour module `Trade`).

C'est l'utilisation la plus courante de classes imbriquées dans un module. Notons qu'en anglais on parle de *nested class* (classe emboîtée, imbriquée).

De même, nous pouvons imbriquer des modules dans des modules, ce qui permet de former en quelque sorte des sous modules.

```ruby
module Sport
  module Football
  end
end
```

Accéder aux méthodes et aux constantes du module principal

Les méthodes (et les constantes) du module principal ne sont pas des méthodes (ni des constantes) du module imbriqué ou de la classe imbriquée. Si on veut les appeler, on veillera à utiliser la syntaxe appropriée.

```ruby
module Football
  PLAYERS_NB = 11

  class Team
    def introduce
      puts "Et voici les #{Football::PLAYERS_NB}␣
           joueurs de l'effectif"
      puts '...'
    end
  end

  module_function

 def rules
   puts 'On a demandé les règles du football.'
   puts "C'est un sport qui se joue à #{PLAYERS_NB}."
   puts '...'
 end

Football.rules
Football::Team.new.introduce
```

De même, on peut imbriquer des classes dans des classes. Par exemple, on peut imaginer une classe pour représenter un moteur dans une classe pour représenter un véhicule.

```ruby
class Vehicle
  class Engine
  end

  def initialize
```

```ruby
    @engine = Engine.new
  end
end

engine = Vehicle::Engine.new
```

Dans certains langages, cela permet de n'autoriser l'instanciation de `Engine` que dans la classe `Vehicle`. En Ruby ce n'est pas le cas et dans le code précédent nous l'utilisons pour créer une variable `engine` en dehors de la classe `Vehicle`. Mais c'est une indication pour le développeur (peut-être qu'il ne veut définir des moteurs que dans la classe `Véhicle`), et là encore cela peut servir d'espaces de noms.

III.3.3.3. Bonnes pratiques

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 `Sport` dans un fichier `sport.rb`.

```ruby
module Sport
  module Handball
    # The content of the Handball module.
  end

  module Football
    # The content of the Handball module.
  end
end
```

Ensuite, dans notre fichier principal, disons `main.rb`, nous allons indiquer qu'il a besoin de `multiplication.rb`.

```ruby
require_relative 'sport.rb'

puts "Les règles du hand : #{Sport::Handball.rules}"
Sport::Football::Team.new.introduce
```

Notons qu'avec ce code, le fichier `sport.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 `sport.rb` dans un dossier `module`, il faudrait alors écrire `require_relative 'sport/multiplication.rb'`. Notons de plus que l'extension du fichier n'est pas obligatoire et qu'il est parfaitement possible d'écrire `require_relative 'sport'`.

Il existe également des méthodes `require` et `load` qui sont un peu différentes et dont nous n'allons pas parler en détail ici.

III.3.3.3.1. 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`. De même, il est conseillé d'avoir une classe par fichier et de le nommer par le nom de la classe en «snake_case».

De plus, notons que généralement nous faisons un fichier par classe et un fichier par module. Ceci permet alors de bien répartir le code. Les règles de nommage pour les fichiers contenant les classes sont les mêmes que celles pour les modules (par exemple la classe `MyClass` dans le fichier `my_class.rb`).

III.3.3.3.2. Principe de responsabilité unique

Le principe de responsabilité unique (SRP pour *Single Responsibility Principle*) est le premier principe de SOLID. En gros il indique qu'une classe ou une méthode ne doit avoir qu'une seule responsabilité (elle ne fait qu'une chose et elle le fait bien). On l'exprime souvent en disant qu'«une classe ne doit avoir qu'une seule raison de changer» (si elle avait plusieurs responsabilités, elle aurait plusieurs raisons de changer).

L'image de l'article humoristique était celle du couple qui dans sa maison avait un interrupteur était chargé d'allumer ou éteindre la lumière de l'entrée, mais aussi de contrôler le broyeur de la cuisine… Assez gênant si nous pensons à la fin de l'histoire.

Considérons notre classe `Message`, elle devrait par exemple juste se charger du message et de son auteur, si un jour nous voulons envoyer des messages, ça ne devra pas être sa responsabilité, mais celle d'un module ou d'une autre classe. De même, si nous voulons formater le message de plusieurs manières différentes (en HTML, en Markdown, en LaTeX, etc.), ça ne devrait pas être la responsabilité de la classe `Message` d'effectuer ce formatage, mais plutôt de différents modules ou classes de formatage.

Ce principe permet d'avoir du code plus robuste car plus simple à modifier et à déboguer. En nous exerçant, en faisant des projets et en discutant avec d'autres programmeurs, nous comprendront progressivement comment écrire du code respectant ses principes.

III.3.4. Exercices

Avec tout ce que nous avons vu, nous avons déjà de quoi faire un bon paquet d'exercices et de projets. En fait, les classes et les modules ne nous permettent pas de faire beaucoup plus de projets que ce que nous pouvions déjà faire… Mais ces notions nous permettent de le faire de manière plus organisée et nous permettent donc d'avoir du code plus simple à écrire et surtout à lire (un programmeur passe beaucoup plus de temps à lire du code qu'à en écrire, autant que le code soit simple à lire).

Pour mettre à l'épreuve ces notions, nous pouvons faire l'exercice de la pharmacie , tiré de la banque d'exercices proposée au début de cette partie du tutoriel.

Maintenant, nous allons de plus en plus faire des petits projets en guise d'exercice, et l'avis d'autres programmeurs sera donc un très grand plus, et donc poster son code sur le forum pour obtenir des avis est vraiment une bonne chose.

Conclusion

Entre modules et classes, nous avons fort à faire. Et nous n'avons pas fini d'explorer tout ce que Ruby a à offrir! Mais avant de continuer, voyons un peu ce que nous avons appris.

— Les modules nous permettent de regrouper des méthodes, des constantes, et en gros du code d'une même thématique.
— Les méthodes «dangereuses» (celles qui modifient l'objet) se terminent par un `!`. La méthode non dangereuse est généralement définie à l'aide de la méthode *bang* associée et de la `dup` (on appelle la méthode dangereuse sur une copie de l'objet et on renvoie cette copie).
— Le mot-clé `self` représente l'objet courant. D'ailleurs qu'affiche `puts self` dans un programme (ou même dans IRB) et quelle est la classe de cet objet?
— Nous créons un fichier par classe et un fichier par module.

III.4. Des objets plus complexes

Introduction

Dans les chapitres précédents, nous avons appris à créer des objets, des classes, puis des modules. Ici, nous allons approfondir notre maîtrise des classes et apprendre à spécialiser des classes et partager du code entre plusieurs classes.

III.4.1. Héritage et composition

III.4.1.1. Héritage

L'héritage est une relation entre deux classes, la super-classe (ou classe mère) et la sous-classe (ou classe fille). Lorsqu'une classe `F` hérite d'une classe `C`, toutes les instances de la sous-classes possèdent alors les méthodes définies dans la superclasse.

L'idée est de spécialiser la classe mère pour créer une nouvelle classe. Par exemple, on pourrait spécialiser une classe `Book` en une classe `Novel`. Pour cela, nous allons utiliser cette syntaxe lors de la définition de la classe `Novel`.

```ruby
class Book
  attr_reader :content

  def initialize(content)
    @content = content
  end
end

class Novel < Book
  attr_reader :author
end
```

Avec `<`, on indique que `Novel` est une sous-classe de `Book` et donc lorsqu'on crée un objet de classe `Novel`, il a une méthode `content` et une méthode `initialize` qui prend en paramètre un argument et fait l'opération `@content = content`.

```ruby
novel = Novel.new('Un beau roman.')
puts novel.content
```

III.4.1.1.1. Redéfinition de méthodes

L'héritage nous permet de disposer des méthodes de la classe mère dans la classe fille. Cependant, il peut arriver que l'on veuille que la méthode soit un peu différente dans la classe fille. Pour cela, on peut redéfinir la méthode dans la classe fille. C'est alors cette nouvelle méthode qui sera appelée si elle est utilisée sur une instance de la classe fille. Par exemple, nous pouvons définir la méthode `initialize` de `Novel` pour qu'elle prenne en paramètre l'auteur.

```ruby
class Novel < Book
  attr_reader :author

  def initialize(content, author)
    @content = content
    @author = author
  end
end

class Magazine < Book
  attr_reader :publisher

  def initialize(content, publisher)
    @content = content
    @publisher = publisher
  end
end

novel = Novel.new('Un beau roman.', 'Clem')
magazine = Novel.new('Contenu.', 'ZdS')
book = Book.new('Un beau livre.')
```

III.4.1.1.2. La méthode super

Lorsque nous redéfinissons une méthode, nous pouvons faire appel à la méthode correspondante de la super-classe. Pour cela, nous allons utiliser le mot-clé `super`.

```ruby
class Book
  # ...
  def read
    puts "Contenu du livre : #{@content}"
```

```ruby
  end
end

class Novel < Book
  # ...
  def read
    puts "Lecture d'un roman de #{@author}."
    super
  end
end

Novel.new("contenu du roman de Zeste de Savoir", "Clem")
```

De plus, il est à noter qu'en utilisant`super`, la méthode de la super-classe est appelée automatiquement avec les arguments passés à la méthode de la classe.

```ruby
class Book
  def read_page(n)
    puts "Lecture de la page #{n}."
  end
end

class Novel < Book
  def read_page(n)
    puts "Lecture du roman de #{@author}."
    super # Ici, `read_page` de `Book` prend `n` en argument.
  end
end
```

Cette «magie» de Ruby peut cependant poser problème si la méthode de la super-classe ne prend pas exactement le même nombre de paramètres. Nous obtiendrons alors une erreur disant que la méthode est appelée avec le mauvais nombre d'arguments. Pour régler cela, il nous suffit de passer à `super` les paramètres voulus.

```ruby
class Book
  def initialize(content)
    @content = content
  end
end

class Novel < Book
  def initialize(content, author)
    super
    @author = author
  end
end
```

Si nous voulons appeler `super` et qu'il nous faut préciser qu'aucun argument ne doit être envoyé à la méthode, nous devrons utiliser une paire de parenthèse; nous emploierons donc `super()`.

Utiliser `super` est par exemple utile quand la méthode de la super-classe fait déjà une partie de ce qui doit être fait dans la méthode correspondante de la sous-classe. En particulier, cela permet de ne pas écrire plusieurs fois le même code (respect du DRY) et de ne changer que ce qui est nécessaire.

III.4.1.2. L'arbre d'héritage de Ruby

En Ruby, nous avons la méthode `is_a?` et son alias `kind_of?` qui prend en paramètre une classe et permettent de savoir si un objet est une instance d'une sous-classe de cette classe (ou une instance de cette classe), et `instance_of?` qui permet de savoir si un objet est **exactement** une instance d'une classe.

```
1 Novel.new.kind_of?(Novel) # => true
2 Novel.new.kind_of?(Book) # => true
3 Novel.new.instance?(Novel) # => false
4 Novel.new.instance?(Book) # => false
```

En fait, lorsque nous utilisons l'héritage, par exemple pour avoir une classe `Novel` qui hérite de `Book`, nous devons noter qu'une instance de `Novel` **est aussi** une instance de `Book` (en clair, un roman est en particulier un livre).

Sachant cela, nous pouvons créer ce que l'on appelle un **arbre d'héritage**. L'idée est simplement que les nœuds sont des classes et que si `F` est une sous-classe de `M`, on a une branche qui va de `M` à `F` (en fait, c'est l'arbre généalogique des classes). Concrètement, on pourrait quasiment construire cet arbre dans Ruby grâce à la méthode `superclass` qui nous donne la super-classe d'une classe.

```
1 Novel.superclass # => Book
```

Et en fait, cet arbre a une racine; il existe une classe qui est un ancêtre de toutes les classes! Il s'agit de la classe `Object` (ce n'est pas si étonnant, tous les objets sont des objets). En fait, quand on crée une classe, elle hérite par défaut de `Object`. Ainsi, `Object` est la super-classe de `Book`.

```
1 Novel.superclass # => Book
2 Book.superclass # => Object
```

On a donc que toutes les méthodes disponibles pour `Object` le sont pour les classes que nous créons. En particulier, cela explique que nous ayons par défaut des méthodes `to_s`, `inspect`, etc. En fait, on pourrait s'amuser à regarder l'arbre d'héritage de certains objets.

III.4.1.2.1. Les classes

En réalité c'est un peu plus compliqué. En particulier, il y a deux classes, `BasicObject` et `Class` dont nous n'allons pas parler en détail. Toutes les classes sont de la classe `Class` (`Object.class`, `Class.class`, `Array.class` valent `Class`). De même, il y a une classe `Module` dont tous les modules sont des instances. Et en fait, on a que `Object` est la super-classe de `Module` qui est elle-même la super-classe de `Class`.

L'idée c'est que les classes sont des `Class`, les modules sont des `Module`, et les instances de classes sont des `Object`. Et avec cet arbre d'héritage, on retrouve que tout est objet.

Avec ça on a un petit paradoxe; `Object` est un objet, mais est aussi une classe, et `Class` est un objet, mais aussi une classe. Ruby fait sa tambouille en interne pour que tout ceci ne pose pas problème. Nous n'allons donc pas plus nous en occuper.

III.4.1.3. Composition

La composition, c'est quelque chose que nous avons déjà utilisé plusieurs fois (et que nous utilisons en fait quasiment à chaque fois que nous utilisons un objet). La composition est le fait d'avoir un objet qui est un attribut d'un autre objet. Vu que les nombres, les chaînes de caractères, et tout le reste en fait, sont des objets, nous comprenons qu'elle est omniprésente.

Si l'héritage correspond à une relation «**être un**», la composition correspond à une relation «**a un**». Par exemple, un message a un auteur. Et on représente cela en donnant à la classe `Message` un attribut `author`. Et c'est de la composition… Oui, la composition c'est juste ça.

Néanmoins, si pour le moment nous avons juste fait de la composition avec des objets «simples» (nombres et chaînes de caractères), mais nous pouvons bien sûr le faire avec n'importe quel type d'objets. Par exemple, nous pouvons tout à fait créer une classe `Person` et faire en sorte que l'auteur d'un message et son destinataire soient des instances de `Person`.

```
1  class Person
2    attr_reader :name
3
4    def initialize(name)
5      @name = name
6    end
7
8    def to_s
9      @name
10   end
11 end
12
```

```ruby
class Message
  attr_reader :author, :content, :date, :recipient

  def initialize(content, author, recipient, date = '21/12/2112')
    # ...
  end
end

clem = Person.new("Clem")
moi = Person.new("Moi")
m = Message.new("Bonjour", moi, clem)
```

Ici, ça permet en particulier de permettre de différencier facilement deux personnes qui ont le même nom. De même, on pourrait avoir une classe `Date` pour représenter des dates avec des opérations dessus, notamment de comparaison (à noter que des classes pour gérer les dates existent déjà en Ruby).

III.4.2. Mixer un module dans une classe

III.4.2.1. Les *mixins*

Comme nous l'avons vu, nous ne pouvons pas instancier de modules; la création d'objets nécessite une classe. Néanmoins, grâce à la méthode `include` qui, rappelons-le, permet de copier les méthodes d'instances d'un module pour les rajouter aux méthodes d'instance de l'objet courent, nous pouvons rajouter les méthodes d'instance d'un module à une classe. On parle alors du module en tant que *mixin*.

```ruby
module M
  def f(x, y)
    x + y
  end
end

class C
  include M
end

puts C.new.f(1, 2)
```

Vu comme ça, les *mixins* n'ont en effet pas l'air très utile. L'idée première que nous devons retenir est qu'ils nous permettent de réunir du code utilisé dans plusieurs classes à l'intérieur d'un module (principe DRY).

? Mais qu'est-ce que cela apporte par rapport à l'héritage? Pourquoi ne pas juste créer une classe et faire de l'héritage?

Il y a plusieurs réponses à cette question, et c'est autant une question de philosophie de Ruby que de conception de programmes. Pour commencer, contrairement à certains langages comme C++, il n'y a pas d'héritage multiple en Ruby. Cela signifie qu'on ne peut pas faire une classe hériter de plus d'une classe. Ainsi, partager du code entre deux classes en utilisant l'héritage nécessite de créer une *god*-classe, une classe avec plein de méthodes. Et cette classe n'a pas forcément de sens au niveau de la conception (en plus de sûrement contrevenir au principe du SRP).

Les *mixins* permettent de contrevenir à cela. En effet, on peut mixer autant de modules que l'on veut à une classe. L'idée est alors d'avoir une **relation d'héritage qui a du sens**, et des *mixins* qui ont eux aussi du sens. Ainsi, on a du code modulable et plus simple. En fait, les *mixins* servent souvent à implémenter des comportements.

Ceci est d'autant plus puissant que dans le module, nous pouvons tout à fait utiliser des variables d'instances puisque qu'en incluant le module, les méthodes d'instances sont copiés).

Supposons par exemple que l'on ait toujours des messages, mais on rajoute des lettres (des messages livrables), des SMS, des colis, etc. On pourrait avoir un module pour les objets livrables (avec une méthode pour livrer, une méthode pour suivre le colis, etc.).

```ruby
class Message
end

module Deliverable
  def deliver
    puts "Livraison de la part de #{@author} pour #{@recipient}."
  end
end

class Letter < Message
  include Deliverable
end

class SMS < Message
end

class PostalPacket
  include Deliverable
end
```

Ici, les classes `Letter` et `PostalPacket` ont accès à la méthode `deliver`, mais pas la classe `SMS`. Ceci est d'autant plus puissant que si une méthode avait déjà été définie dans la classe `Message` (disons par exemple une méthode `send`), nous pouvons la définir dans `Deliverable`

et dans les sous-classes de `Message` qui incluent `Deliverable`, c'est la méthode `send` de `Deliverable` qui sera appelée.

```
1  class Message
2    def send
3      puts "Envoi du message de #{@author} pour #{@recipient}."
4    end
5  end
6
7  module Deliverable
8    def send
9      puts "Envoi du message de #{@author} pour #{@recipient}
           par service postal."
10   end
11 end
12
13 class Letter < Message
14   include Deliverable
15 end
16
17 Letter.new('Contenu', 'Auteur', 'Destinataire').send
```

III.4.2.2. Un peu de *lookup* ?

Jusqu'à maintenant, l'envoi d'un message `m` à un objet `obj` semblait assez simple: `obj` reçoit le message `m` et il exécute la méthode `m` si sa classe définit cette méthode. Mais en rajoutant l'héritage, puis les **mixins* les choses se compliquent un peu. Si la classe de `obj` ne définit pas `obj`, il va regarder dans sa super-classe, si elle na la définit pas, il va encore remonter et ce jusqu'à arriver à `Object`. Ou il va regarder si la méthode n'est pas disponible dans un module qui est mixé par la classe.

?

Mais dans quel ordre regarde-t-il cela? Si une classe et un module mixé dans cette classe définissent tous les deux une méthode `m`, laquelle de ces méthodes est appelée?

L'ordre dans lequel Ruby cherche la méthode est défini par un algorithme dit de *lookup*. Cet algorithme permet donc de savoir quelle méthode sera prioritaire, et nous allons voir un peu comment il fonctionne ici.

III.4.2.2.1. Héritage et *mixins*

Commençons par créer un module `M` avec une méthode `f`.

```ruby
module M
  def f
    puts 'Méthode du module M.'
  end
end
```

Le cas le plus simple est celui où la méthode est également déclarée dans la classe de l'objet. Dans ce cas, on s'attend à ce que ce l'appel se fasse avec la méthode de la classe, et c'est bien ce qui se passe.

```ruby
class C
  include M

  def f
    puts 'Méthode de la classe C.'
  end
end

C.new.f # => Méthode de la classe C
```

Et si maintenant on crée une classe `D` qui hérite de `C`?

```ruby
class D < C
end

D.new.f
```

Là encore, c'est la méthode de la classe `C` qui est appelée et pas celle du module.

Les méthodes définies dans les classes semblent prioritaires. Néanmoins, comme nous l'avons vu plus haut avec l'exemple de `send` définie dans `Message` et dans `Deliverable` mais pas dans `Letter`, si une classe inclut un module, les méthodes de ce module sont prioritaires sur les méthodes de la super-classe.

```ruby
class C
  def f
    puts 'Méthode de la classe C.'
  end
end

class E < C
  include M
end
```

```
10
11 E.new.f
```

En fait, Ruby regarde dans les méthodes de la classe de l'objet, puis dans les méthodes des modules inclus dans cette classe, et s'il ne trouve pas, alors il passe à la super-classe et tente la même chose, et ce jusqu'à trouver la méthode ou à arriver à `BasicObject` et là il ne peut pas monter plus haut dans la hiérarchie des classes s'il ne trouve pas et nous obtenons une erreur.

III.4.2.2.2. Plusieurs modules avec la même méthode

Dans le cas où plusieurs modules inclus dans la classe définissent la même méthode, c'est le dernier module inclus qui a la priorité.

```ruby
module M
  def f
    puts 'Méthode du module M.'
  end
end

module N
  def f
    puts 'Méthode du module N.'
  end
end

class C
  include M
  include N
end

C.new.f # => Dans le module N
```

Ici, c'est le module `N` qui est inclus en dernier, c'est donc sa méthode qui est appelée.

? Que se passe-t-il si un module est inclus plusieurs fois?

Faisons le test.

```ruby
class C
  include M
  include N
  include M
end
```

```
6
7 C.new.f
```

Ici, le module `M` est inclus en dernier, on s'attend donc à ce que sa méthode soit exécutée, et pourtant ce n'est pas le cas. En fait, lorsqu'un module est déjà inclus, la seconde inclusion ne fait rien.

Notons que cela est aussi vrai si le module a été inclus par un ancêtre de la classe.

```ruby
module M
  def f
    puts 'Méthode du module M.'
  end
end

class C
  include M

  def f
    'Méthode de la classe C'
  end
end

class D < C
  include M
end

D.new.f
```

Ici, on pourrait s'attendre à ce que ce soit la méthode de `M` qui soit exécutée, conformément aux règles vues plus haut, mais l'inclusion de `M` dans `C` n'a rien fait, vu que `M` est déjà inclus dans `C` et que `D` hérite de `C`. Ainsi, la recherche de la méthode `f` d'un objet de la classe `D` va mener grosso-modo à ceci.

1. On regarde dans les méthodes d'instance de `D`, non trouvé.
2. On regarde dans les modules inclus dans `D`, non trouvé (`M` n'a pas été inclus à nouveau dans `D`).
3. On passe à la super-classe, on regarde dans les méthodes d'instance de `C`, on trouve.

III.4.2.2.3. Les méthodes singletons

Les méthodes singletons sont un peu à part dans tout ça. En fait, elles ont la priorité absolue, même sur les méthodes d'instances.

```ruby
class C
  def f
    puts 'Méthode de la classe C.'
  end
end

o = C.new

def o.f
  puts 'Méthode singleton de f.'
end

puts o.f # => Méthode singleton de f.
```

III.4.2.2.4. Et finalement ?

Finalement, voici l'ordre utilisé par Ruby pour trouver la méthode à appeler.

1. Regarder dans les méthodes singletons de l'objet.
2. Regarder dans les méthodes définies dans la classe de l'objet.
3. Regarder dans les modules inclus dans la classe de l'objet dans l'ordre inverse d'inclusion.
4. Retourner à l'étape 2, mais avec la super-classe.

Notons que la méthode `super` suit également ce même chemin. Ceci nous permet d'écrire ce code (qui nous permet en passant de vérifier que les étapes sont bien celles que nous avons données.

```ruby
module M
  def f
    puts 'Dans le module M.'
    super
  end
end

class C
  def f
    puts 'Dans la classe C.'
  end
end

class D < C
  include M

  def f
    puts 'Dans la classe D.'
```

```
19      super
20    end
21  end
22
23  o = D.new
24
25  def o.f
26    puts 'Méthode singleton de o'
27    super
28  end
29
30  o.f
```

Nous n'avons pas encore tout vu sur ce sujet. En particulier, d'autres méthodes `extend` et `prepend` viennent avec `include` et permettent d'inclure des méthodes de module dans des objets et rajoutent d'autres étapes dans l'algorithme de *lookup* de Ruby. Nous n'en parleront pas pour le moment, mais nous pouvons tout à fait aller nous renseigner à leur sujet; nous avons le niveau pour comprendre ce qu'elles font. Nous pourrons alors regarder ce document par exemple pour l'algorithme de *lookup*.

III.4.3. Quelle technique utiliser ?

III.4.3.1. Bien utiliser l'héritage

III.4.3.1.1. Le Principe de substitution de Liskov

Après avoir vu toutes ces techniques, il convient d'apprendre à bien les utiliser. Commençons par voir quelques notions de conception liées à l'héritage. C'est l'heure de voir un autre principe SOLID. Cete fois il s'agit du principe de substitution de Liskov (LSP pour *Liskov Substitution Principle*). On l'exprime grossièrement en disant que si F hérite de M, alors on peut donner un objet de la classe F à n'importe quel endroit où un objet de classe M est attendu.

Regardons comme d'habitude par l'image de l'article humoristique. Cette fois, il s'agit de quelqu'un qui dîne de la salade. Il y a donc des aliments comestibles dans son assiette… Regardons-y de près: dans sa salade, il y a de la tomate, de l'oignon, des carottes, **mais aussi de la ciguë (qui est une plante toxique)**.

Ici, on considère une méthode qui prend en paramètre une liste d'aliments comestibles et en fait une salade. On a alors une classe mère pour les aliments comestibles. La carotte, l'oignon, la tomate sont des aliments comestibles. Malheureusement, si on fait la ciguë hériter de cette classe mère (et c'est en gros ce qui s'est passé dans ce malheureux dîner), alors on peut l'utiliser pour faire de la salade, ce qui est clairement une mauvaise idée.

Un exemple classique de violation du SRP est celui du carré et du rectangle. En effet, si on a une classe `Rectangle` dont on peut modifier la hauteur et la largeur, on peut imaginer une classe

`Square` qui hérite de `Rectangle` (un carré est un rectangle). Malheureusement, les méthodes de modification de la hauteur et de la largeur ne peuvent pas être utilisées directement car modifier la largeur d'un carré sans en modifier la hauteur pose problème (ce n'est plus un carré dans ce cas).

III.4.3.1.2. L'héritage et les variables de classe

Nous allons revenir sur les variables de classe. Rappelons-nous, plus tôt dans le tutoriel nous avions dit ceci.

> Les variables de classes sont à éviter car elles peuvent s'avérer dangereuses. Nous verrons pourquoi dans les chapitres suivants, mais il nous faut savoir que c'est en partie parce qu'elles peuvent être modifiées là où on ne s'y attend pas.
>
> *Chapitre sur les classes*

Ici, nous allons écrire une classe `M` et une classe `F` qui hérite de `M`. `M` a une variable de classe `@@number` qui comptabilise le nombre d'instances de `M` créées. Ainsi, `F` aura également une variable de classe `@@number` qui comptabilisera le nombre d'instances de `F` créées.

```
1  class M
2    @@number = 0
3
4    def initialize
5      @@number += 1
6    end
7
8    def self.number
9      @@number
10   end
11 end
12
13 class F < M
14   def initialize
15     @@number += 1
16   end
17 end
```

Un code simple, n'est-ce pas? Nous définissons les constructeurs pour qu'ils incrémentent `@@number` et nous définissons également une méthode de classe `number` pour avoir accès à `@@number`. Testons-le.

```
1 M.new
2 M.new
3 F.new
4 puts M.number
```

```
5 puts F.number
```

Le résultat qui est attendu est le suivant.

```
1 2
2 1
```

En effet, deux instances de `M` ont été créées, et donc le constructeur de `M` a été appelée deux fois, et une instance de `F` a été créé, appelant le constructeur de `F` une fois (notons qu'il serait peut-être préférable d'avoir `3` pour le nombre d'instances de `M`, vu qu'une instance de `F` est une instance de `M`). Et pourtant, le résultat obtenu est le suivant!

```
1 3
2 3
```

En fait, les variables `@@number` de `M` et de `F` sont les mêmes. Et avec ça, nous comprenons le problème. Une modification dans `M` affectera aussi `F` puisque ce sont les mêmes variables. Ce code (tiré de ce guide) l'illustre bien.

```
1  class Parent
2    @@class_var = 'parent'
3
4    def self.print_class_var
5      puts @@class_var
6    end
7  end
8
9  class Child < Parent
10   @@class_var = 'child'
11 end
12
13 Parent.print_class_var # => will print 'child'
```

La solution à ce problème est également donnée dans le guide, il faut utiliser des **variables d'instance de classe**.

? Quoi, des variables d'instance de classe? Mais qu'est-ce que c'est?

Tous les objets on des variables d'instance. Mais une classe est aussi un objet, et a donc des variables d'instance.

```ruby
class C
  @x = 2
end

puts C.instance_variables
```

Ici, nous créons une classe `C` avec une variable d'instance `@x`. Remarquons qu'il s'agit de la seule variable d'instance de `C`.

En utilisant les variables d'instance de classe, nous avons de quoi régler notre souci. En fait, on utilise les variables d'instance de classe pour simuler des variables de classe; chaque classe aura sa variable d'instance et elles seront indépendantes. On écrit alors ce code.

```ruby
class M
  @number = 0

  def initialize
    M.number += 1
  end

  def self.number
    @number
  end

  private def self.number=(x)
    @number = x
  end
end

class F < M
  @number = 0

  def initialize
    self.class.number += 1
    super
  end
end
```

Ici, nous avons également appelé `super` dans la méthode `initialize` de `F`. Ainsi, le compteur de `M` sera incrémenté même lors de la création d'une instance de `F`. Nous pouvons alors essayer le code qui posait problème.

```ruby
M.new
M.new
F.new
```

```ruby
puts M.number
puts F.number
```

Et nous obtenons le résultat attendu!

La conclusion de tout ceci est que nous préférerons utiliser des variables d'instance de classe plutôt que des variables de classe.

III.4.3.2. Le typage de canard

Maintenant que nous avons vu comment bien faire de l'héritage, il nous reste à parler du typage de canard ou *duck typing*. L'idée du typage de canard est de profiter du typage de Ruby pour créer des méthodes qui acceptent des paramètres ayant un certain comportement et non un certain type.

? Et concrètement, qu'est-ce que ça veut dire?

Plus concrètement, plutôt que de considérer qu'une méthode prend en paramètre un objet de telle classe, nous allons considérer qu'il prend en paramètre un objet qui dispose d'une certaine interface, et présente donc certaines méthodes.

```ruby
def duck_typing(duck)
  duck.fly
  duck.swim
end
```

Ici, on se dit que seul un objet de la classe `Duck` peut être envoyé à la méthode `duck_typing`, mais c'est faux. Il suffit que l'objet passé en paramètre ait une méthode `fly` et une méthode `swim`.

> Si ça vole comme un canard, que ça nage comme un canard, et que ça fait coin-coin comme un canard, alors c'est un canard.
>
> *Le typage de canard.*

Grâce à ceci, nous pouvons nous concentrer sur l'interface de nos objets plutôt que sur nos types.

III.4.3.2.1. Supplanter l'héritage ?

Le *duck-typing* permet de supplanter l'héritage sur certains points. Par exemple, plutôt que de créer une classe `Animal`, des classes `Duck` et `Cat` héritant de `Animal`, et une méthode `speak`, nous pouvons juste créer les classes `Duck` et `Cat`.

```ruby
class Duck
  def speak
    puts 'Coin-coin.'
  end
end

class Cat
  def speak
    puts 'Miaou.'
  end
end

def stroke(animal)
  puts 'On caresse un animal.'
  animal.speak
end

stroke(Duck.new)
stroke(Cat.new)
```

Ici, on peut envoyer n'importe quel argument à `stroke`, pourvu qu'il implémente `speak`. En fait, créer une classe `Animal` ici, juste pour que `Cat` et `Duck` en héritent est une mauvaise idée. Nous allons donc éviter de créer des classes juste pour faire de l'héritage et préférer du *duck-typing* dans ce genre de cas. Plus précisément, l'héritage sera préféré s'il y a **vraiment** des caractéristiques, des attributs, des méthodes à partager.

Par exemple, si nous ne considérons que des animaux domestiques qui ont un propriétaire, un nom, etc., nous pouvons créer une classe `Pet`, dont héritera `Dog` et `Cat`.

```ruby
class Pet
  attr_reader :name, :master, :age

  def initialize(name, master, age)
    @name = name
    @master = master
    @age = age
  end

  def welcome
    puts "#{name} accueille #{master}."
    speak
  end
end

class Dog < Pet
  def speak
    puts 'Ouaf !'
```

```
19    end
20  end
21
22  class Cat < Pet
23    def speak
24      puts 'Miaou !'
25    end
26  end
27
28  Dog.new('Rex', 'Clem', 3).welcome
```

Ici, nous avons même fait encore mieux, nous avons utilisé de l'héritage (avec `initialize`, `welcome` et les attributs) et du *duck-typing* avec l'utilisation de `speak` dans `welcome`.

Néanmoins, les classes qui héritent de `Pet` doivent implémenter une méthode `speak`, sinon on aura un problème. En fait, `Pet.new('N', 'M', 2).welcome` provoque une erreur, et nous ne voulons pas de cela, donc il faudrait mieux avoir une méthode `speak` pour `Pet`…

III.4.3.3. Héritage, mixins ou composition ?

E rajoutant le typage de canard, nous avons un arc plutôt bien fourni entre l'héritage, les mixins et la composition. Et il peut donc être compliqué de savoir lequel privilégier. Pour commencer, il est raisonnable de dire que cela dépendra de la situation. Mais rappelons quelques idées que nous avons déjà abordées.

Néanmoins, ce qu'il nous faut comprendre, c'est que l'utilisation d'une technique a un sens et donne à notre code du sens. Par exemple, lorsque nous faisons une classe `F` hériter d'une classe `M`, le sens donné est que les instances de `F` sont aussi des instances de `M` (ceci est encore plus vrai avec le LSP).

Ainsi, cela n'aurait pas de sens de faire une classe pour une roue hériter d'une classe représentant une voiture. Entre une roue et une voiture, il y aurait plutôt une relation de composition; une voiture a une roue.

Et finalement, mixer un module `M` dans une classe `C` indique que la classe se comporte comme `M`, agit comme `M`. C'est la sémantique principale du *mixin*. Par exemple, on pourrait avoir un module `Drivable` qu'on mixerait dans la classe `Car` (une voiture se comporte comme quelque chose qu'on peut conduire).

Pour bien appréhender les subtilités derrière tout cela, nous renvoyons vers cet article de Synbioz . Bien entendu, il n'est pas suffisant et c'est la pratique qui nous permettra de progresser. De plus, il nous faut bien garder à l'esprit qu'il y a rarement une seule bonne solution à un problème.

III.4.4. Exercices

Pour ce chapitre, nous pouvons commencer par jouer avec la classe `Message` (définir d'autres classes et modules pour les personnes, les livreurs, mais aussi pour avoir d'autres types de messages). Par exemple, nous pourrions gérer des carnets d'adresses, des répertoires de téléphones, et lorsqu'on décide d'envoyer une lettre, on l'envoie à une adresse.

Après s'être amusé avec les messages, voici un exercice assez connu ici pour pratiquer la POO: le Javaquarium (pour l'occasion, nous le renommerons *Rubaquarium* qui est un nom bien plus classe). Nous n'allons pas redonner les règles ici, il suffit d'aller voir le lien. Bonne programmtion!

Conclusion

Ce chapitre conclut quasiment les bases que nous avons besoin de connaître pour organiser du code et devenir des Rubyistes convaincus. Bien sûr, il nous faut mettre tout ça en pratique, mais récapitulons déjà tout ce que nous avons appris ici.

— L'héritage permet d'étendre (ou plutôt de spécialiser) une classe (la classe-mère) en créant une nouvelle classe (la classe-fille) qui bénéficie des méthodes et attributs de la classe-mère qu'elle peut redéfinir.
— Le typage de canard est à préférer à l'héritage.
— Les *mixins* permettent d'ajouter les méthodes d'un module à une classe; ils permettent d'implémenter des comportements.
— La composition est elle aussi un outil appréciable. Avec l'héritage et la composition, elle nous permet de concevoir du code robuste, lisible et extensible.

III.5. La gestion des erreurs

Introduction

Pour le moment, lorsque nos programmes rencontraient une erreur, ils se terminaient immédiatement. Dans un vrai programme, c'est quand même plutôt gênant. Dans ce chapitre, nous allons donc voir comment gérer les erreurs.

III.5.1. Les exceptions

III.5.1.1. Qu'est-ce qu'une exception

Pour commencer ce chapitre, il nous faut bien sûr commencer par voir ce qu'est une exception. En fait, il s'agit tout simplement d'un évènement qui interrompt le flux d'exécution normal du programme. Il s'agit très souvent d'une erreur qui survient dans le programme. En effet, quand une erreur a lieu, une exception est levée par le programme (et son flux d'exécution est interrompu). Un exemple nous permettra d'y voir plus clair.

```
1 a = gets
2 b = a + 2
3 print 'Addition faite.'
```

En exécutant ce programme, nous obtenons l'erreur «*no implicit conversion of Integer into String (TypeError)*» (nous avons oublié de convertir `a` en nombre). Ben c'est qu'une exception a été levée. Le flux normal d'exécution du programme a bien été interrompu (il était censé afficher «Addition faite» et il ne l'a pas fait, il s'est terminé directement).

Le but de ce chapitre est de nous apprendre à prendre en compte ces erreurs (pour par exemple afficher un vrai message d'erreur, réessayer, etc.) et donc à gérer les erreurs proprement.

III.5.1.1.1. Trace de l'erreur

Une exception vient avec une «trace». Cette trace correspond au chemin du programme qui a mené à l'erreur. C'est un peu obscur dit ainsi, un exemple aidera à mieux comprendre.

```ruby
def f(x)
  x + 2
end

print f('a')
```

Ici, une exception est encore lancée et notre interpréteur nous affiche un résultat de ce type.

```
Traceback (most recent call last):
2: from file.rb:5:in `<main>'
1: from file.rb:2:in `f'
file.rb:2:in `+': no implicit conversion of Integer into String
    (TypeError)
```

C'est la trace de notre erreur, et elle se lit ainsi (de bas en haut):

- À la ligne 2 du fichier, dans la méthode `+`, il y a eu un problème de conversion de type.
- La méthode `+` avait été appelée par la méthode `f` à la ligne 2.
- La mééthode `f` avait été appelée dans `<main>` (le programme principal) à la ligne 5.

Cette trace permet de savoir quand et où une erreur a lieu dans un script Ruby.

III.5.1.2. Attraper une exception

Comme nous l'avons dit, quand une exception survint, elle interrompt le flux d'exécution de notre programme et l'interpréteur Ruby nous dit ensuite qu'une exception a interrompu notre programme. Cependant, il est possible d'**attraper** une exception. Cela permet d'empêcher sa propagation, de traiter l'erreur et d'éventuellement continuer le programme.

Pour attraper une exception, nous utilisons la syntaxe la syntaxe `begin-rescue-end`. Entre le `begin` et le `rescue`, nous mettons le code que nous voulons exécuter et qui va potentiellement lancer une exception. Puis, entre le `rescue` et le `end`, nous mettons le code à exécuter en cas d'exception. Voyons un exemple que nous allons décortiquer.

```ruby
begin
  print "Entrez un nombre : "
  number = gets
  result = 1 / number
rescue
  puts 'Exception attrapée dans le programme.'
end
puts 'Fin du programme.'
```

Ici, nous demandons à l'utilisateur de rentrer un nombre. Nous récupérons sa saisie dans `number` et nous lui rajoutons `1`. Si ce bout de code échoue, une exception sera levée, et elle est attrapée par le bout de code dans le `rescue`, c'est-à-dire l'affichage d'un message d'erreur.

Lorsque nous essayons ce programme, il y a bien sûr une exception qui est levée (nous ne convertissons pas la saisie de l'utilisateur en entier, donc nous essayons d'additionner un entier et une chaîne de caractères). Le message d'erreur est donc affiché.

? Pourquoi le message «Fin du programme» est-il également affiché?

Si une exception est attrapée, elle ne se propage pas et le programme continue son cours normalement (à moins que dans le bloc `rescue` on ferme le programme ou quelque chose de ce genre). Il est donc normal que le message indiquant la fin du programme soit affiché.

III.5.1.2.1. Rattraper les exceptions dans une méthode ou un bloc de code

Lorsque nous écrivons une méthode ou un bloc de code, lee début de ce bloc ou de la méthode nous donne un `begin-end` implicite. Ainsi, on peut y utiliser le mot-clé `rescue` directement. Il sera alors associé à toute la méthode ou à tout le bloc.

```ruby
def divide(x, y)
  return x / y
  rescue
    return 0
end
```

Nous essayons de retourner `x / y`, si une exception est lancée, on la rattrape et on retourne `0`.

Si l'on veut avoir une gestion plus fine, nous devons cette fois retourner au `begin-rescue-end`. En effet, avec ce que nous venons de voir, tout ce qui est du `rescue` au `end` de la méthode n'est exécuté qu'en cas d'exception, et toutes les exceptions lancées entre le `def` et le `rescue` sont rattrapées.

```ruby
def divide(x, y)
  result = x / y
  rescue
    result = 0
  puts "#{x} / #{y} = #{result}"
end
```

Ici, si aucune exception n'est lancée, l'affichage n'aura pas lieu. Pour obtenir le comportement souhaité, nous modifions le code de la manière suivante.

```ruby
def divide(x, y)
  begin
    result = x / y
  rescue
    result = 0
  end
  puts "#{x} / #{y} = #{result}"
end
```

Nous essayons de faire des méthodes courtes et qui n'ont qu'une seule fonction. Par exemple, notre méthode `divide` se contentrait de renvoyer le résultat de la division (ce serait donc notre première version). Ainsi, il arrive souvent qu'on puisse utiliser le `rescue` directement dans nos méthodes.

III.5.1.3. Mais quelle est donc cette exception ?

Bon, on sait qu'on a une exception, mais on n'est pas très avancé. Nous aimerions bien avoir des informations sur l'exception qu'on vient d'attraper. Pour cela, nous utilisons l'opérateur `>=` avec `rescue`. Il nous permet de capturer l'exception qui a été attrapé.

```ruby
begin
  print "Entrez un nombre : "
  number = gets
  result = 1 / number
rescue => e
  puts "Exception #{e.class} attrapée : #{e.message}."
end
puts 'Fin du programme.'
```

Ici, en exécutant le programme, nous obtenons le résultat suivant.

```
Exception TypeError attrapée : String can't be coerced into Fixnum.
Fin du programme.
```

Ici nous apprenons que (comme tout le reste) les exceptions sont des objets (nous aurions pu nous y attendre). L'opérateur `>=` nous permet de récupérer une exception (ici de la classe `TypeError`) et nous utilisons sa méthode `message` pour obtenir le message associé à l'exception.

Essayons le même programme, mais en rajoutant la conversion en entier.

```ruby
begin
  print "Entrez un nombre : "
  number = gets.to_i
  result = 1 / number
rescue => e
  puts "Exception #{e.class} attrapée : #{e.message}."
end
puts 'Fin du programme.'
```

Cette fois, on peut causer plusieurs erreurs. Par exemple, en ne rentrant pas un entier, ou en entrant zéro (ce qui causera une division par 0). Cette fois, on obtient une exception différente.

```
Exception ZeroDivisionError attrapée : divided by 0.
```

On a donc plusieurs classes d'exceptions.

Notons de plus que les exceptions ont une méthode `backtrace` qui nous donne accès à la trace dont nous avons parlé plus haut.

```ruby
def f(x)
  1 / x
end

begin
  f(0)
rescue => e
  puts e.backtrace
end
```

III.5.2. Gérer les exceptions

III.5.2.1. La classe des exceptions

Nous avons déjà rencontré deux exceptions différentes, une de la classe `TypeError`, et l'autre de la classe `ZeroDivisionError`. En Ruby, les noms des classes des exceptions sont plutôt explicites. En fait, toutes les exceptions descendent de la classe `Exception`. Voici un code tiré de cet article de Synbioz qui nous montre les différentes classes existantes.

```ruby
def subclasses_tree(klass, level = 0)
  puts level.zero? ? klass.to_s : "#{' ' * (4 * (level - 1))}└── #{klass}"
  descendants_of(klass).select { |k| k.superclass == klass }.each do |k|
    subclasses_tree(k, level + 1)
  end
end

def descendants_of(klass)
  ObjectSpace.each_object(Class).select { |k| k < klass }
end

subclasses_tree(Exception)
```

On peut alors observer la longue liste d'exceptions de Ruby. Notons par exemple l'existence de `ArgumentError` (lancée par exemple lorsqu'une méthode est appelée avec le mauvais nombre d'arguments) ou encore de `NameError` (lancée par exemple lorsqu'un identifiant est utilisée et que l'interpréteur ne le connaît pas).

Notons que l'arbre d'héritage est plutôt bien fait. Par exemple, `NoMethodError` est une classe fille de `NameError`. En effet, `NameError` est lancée lorsqu'un identifiant inconnu est utilisé, et `NoMethodError` lorsqu'une méthode inconnue est utilisée (et cette méthode a bien un identifiant). Pour voir cela, testons ces deux codes.

```ruby
put 'On affiche ceci' # => NoMethodError, put est inconnue.
```

Ici, on sait que `put` est censée être une méthode puisqu'on a un argument, l'exception `NoMethodError` est lancée.

```ruby
put # => NameError, put est inconnue
```

Ici, `put` pourrait être une méthode, mais pourrait aussi juste être une variable normale, ce serait donc une erreur de lancer une exception de la classe `NoMethodError`.

III.5.2.2. Rattraper des exceptions particulières

Il y a plusieurs types d'exceptions, et nous avons la possibilité de rattraper une exception en particulier. Pour cela, il nous suffit de préciser à `rescue` la classe de l'exception à rattraper.

```ruby
def divide(x, y)
  return x / y
  rescue ZeroDivisionError
    puts 'Division par 0 !'
    return 0
end
```

Bien entendu, nous pouvons toujours capturer l'exception rattrapée dans une variable.

```ruby
def divide(x, y)
  return x / y
  rescue ZeroDivisionError => e
    puts "Division par 0 : #{e.message} !"
    return 0
end
```

III.5.2.2.1. Rattraper plusieurs types d'exceptions...

Dans les deux codes précédents, seules les exceptions de la classe `ZeroDivisionError` seront rattrapées. Néanmoins, nous voudrons peut-être rattraper plusieurs types d'exceptions. Par exemple, nous avons plus haut obtenu une exception `TypeError` lorsque nous essayions de diviser `1` par une chaîne de caractère. Cette exception peut arriver, rattrapons là.

```ruby
def divide(x, y)
  return x / y
  rescue ZeroDivisionError, TypeError => e
    puts "Impossible de calculer #{x} / #{y} : #{e.message} !"
    return 0
end

divide(1, 0)
divide(1, 'x')
```

De manière générale, on rattrape les exceptions `E1`, `E2`, `E3`, et plus encore, en utilisant le code `rescue E1, E2, E3` et en ajoutant `=> variable` pour capturer l'exception si besoin est.

III.5.2.2.2. ... Avec des traitements différents

Quand nous voulons rattraper plusieurs types d'exceptions, il peut arriver que l'on veuille avoir des traitements différents suivant de l'exception rattrapée. Pour cela, il suffit d'utiliser plusieurs fois le mot-cle `rescue`. On va ainsi créer un bloc de code à exécuter pour chaque `rescue`. Par

exemple, affichons un message différent si la division échoue à cause d'une erreur de type ou d'une division par zéro.

```ruby
def divide(x, y)
  return x / y
  rescue ZeroDivisionError => e
    puts 'Pas de division par zéro !'
    return 0
  rescue TypeError => e
    puts "Erreur de type : #{e.message}"
end

divide(1, 0)
divide(1, 'x')
```

Cette fois, nous obtenons des messages différents lors des deux appels à la méthode.

i L'infini en Ruby

En fait, en Ruby, nous avons une constante `INFINITY` dans la classe `Float` de Ruby. Une division par zéro avec des flottants (`1.0 / 0` par exemple) renvoie cette constante. On pourrait fait notre méthode la renvoyer en cas de division par zéro (ou même essayer de faire `x / y.to_f`).

```ruby
def divide(x, y)
  return x / y
  rescue ZeroDivisionError => e
    return Float::INFINITY
  rescue TypeError => e
    puts "Erreur de type : #{e.message}"
    return 0
end
```

III.5.2.3. Bonne gestion des exceptions

Avec tout ce que nous avons vu, nous pouvons légitimement nous demander comment bien gérer les exceptions. Quand faut-il les rattraper? Comment faut-il les rattraper? Ce sont à ces questions et à d'autres encore que nous répondrons ici.

Une des premières choses à retenir (et des plus importantes) est de **ne pas utiliser les exceptions pour le flux de contrôle**. Comme son nom l'indique, une exception est censée être lancée **lorsqu'une situation** exceptionnelle, qu'on n'attendait pas arrive**. Par exemple, l'impossibilité d'ouvrir un fichier ou de se connecter à un réseau.

Si on prend l'exemple de la division par 0 que nous utilisons depuis le début, elle ne devrait pas être gérée avec une exception si c'est quelque chose qui peut arriver (par exemple si c'est l'utilisateur qui rentre les données), si on veut afficher un message d'erreur si cela arrive, c'est un `if` qu'il nous faut utiliser.

```
n == 0 ? puts 'Impossible de diviser par 0 !' else puts n / d
```

Bien sûr, si `n` est fourni et ne devrait pas pouvoir valoir 0 (par exemple, si on essaie de récupérer la date du jour et qu'on tombe sur 0), alors là, une exception n'est pas une mauvaise chose.

III.5.2.3.1. Les exceptions du système

Certaines exceptions de Ruby sont dédiés à son bon fonctionnement et il est déconseillé de les rattraper. Nous pouvons comprendre assez aisément pourquoi ce n'est pas une bonne idée de rattraper une exception `SystemExit` par exemple (elle est déclenchée lorsque l'utilisateur décide de fermer le programme). Un autre exemple est celui de `LoadError` qui est lancée quand le chargement d'un fichier (avec `require`) échoue.

```
begin
  require 'bad_file'
  main
rescue Exception => e
  puts 'Exception, on quitte le programme.'
end
```

Dans cet exemple (adapté du même article de Synbioz), on ne sait pas si le problème est causé par le `require` ou par le `main`; le problème de chargement est masqué alors que c'est une erreur qu'on préférerait avoir.

C'est pourquoi nous n'attraperons que les exceptions qui descendent de `StandardError`.

III.5.2.3.2. Du moins général au plus général

De plus, s'il y a plusieurs exceptions à rattraper, nous les rattraperont du plus général au moins général. Ce conseil se comprend bien: si le `rescue` d'une exception est placée avant celui d'une autre exception plus spécifique, le code du second bloc `ensure` ne sera jamais exécuté, puisque les exceptions de ce type seront rattrapées par le bloc `ensure` plus général.

```
begin
  puts 1 / 0
rescue StandardError
  puts 'Une erreur standard'
rescue ZeroDivisionError
```

```
6     puts 'Division par 0 !'
7   end
```

Ici, il nous faut donc placer le `rescue` du `ZeroDivisioError` avant celui du `StandardError`.

III.5.2.3.3. La clause `ensure`

Le mot-cle `ensure` nous permet de nous assurer qu'un traitement est effectué même si une exception a lieu. Par exemple, il permet de s'assurer qu'un fichier est bien fermé même après son ouverture. On place le code à eexécuter dans tous les cas entree le `rescue` et le `end`. On a ainsi la structure `begin-rescue-ensure-end`. Ici, nous allons l'utiliser pour afficher un message de fin dans tous les cas. Le `rescue` n'est pas obligatoire (nous pouvons parfaitement ne pas vouloir rattraper l'exception, mais quand même faire quelque chose si elle survient.

```
1   def main
2     loop do
3       str = gets.chomp
4       break if str == 'quit'
5       n = str.to_i
6       puts "1 / #{n} = #{1 / n}"
7     end
8   end
9
10  begin
11    main
12  ensure
13    puts 'Fin du programme !'
14  end
15
16  puts 'Aucune exception, bravo !'
```

Ici, si on rentre une saisie invalide (zéro ou encore une chaîne différente de `exit`), une exception `ZeroDivisionError` est lancée et n'est pas rattrapée, mais le code du `ensure` est quand même exécuté, contrairement au code qui suit en dehors du `bloc`.

Quand on dit que ce code est toujours exécuté, c'est vraiment toujours. On pourrait tout à fait rattraper l'exception et faire un `return` si elle a lieux que le message serait quand même affiché (mais la trace de l'erreur ne serait plus affichée).

```
1   begin
2     main
3   rescue StandardError => e
4     return -1
5   ensure
6     puts 'Fin du programme !'
```

```
end

puts 'Aucune exception, bravo !'
```

III.5.3. De l'autre côté du miroir

III.5.3.1. Lever des exceptions

Ruby nous laisse la possibilité de lever nous même des exceptions. Pour cela, nous utilisons le mot-clé `raise` avec le nom de l'exception à lever.

```
def f(x)
  raise ArgumentError if x == 0
  1 / x
end

f(10)
f(0) # ArgumentError
```

Nous pouvons donner à `raise` un second argument qui correspond au message de l'exception levée.

```
def f(x)
  raise ArgumentError, 'Impossible de diviser par 0 !' if x == 0
  1 / x
end
```

Notons de plus que le premier argument peut être omis auquel cas c'est une exception de la classe `RuntimeError` qui sera lancée. On peut alors juste écrire `raise 'Une erreur RuntimeError'` si on veut lancer une exception `RuntimeError`.

```
begin
  raise 'Error'
rescue StandardError => e
  puts e.class
  puts e.message
end
```

Ici, l'exception lancée est bien une `RuntimeError` et son message est bien «*Error*».

III.5.3.2. Créer nos propres exceptions

Nous pouvons lancer nous même des exceptions, et bien entendu, nous pouvons aussi créer nos propres exceptions. Il nous suffit de créer une classe classique qui hérite de la classe `Exception` (ou d'une de ses descendantes). Nous pouvons alors la lancer et la rattraper comme n'importe quel autre type d'exception.

```ruby
class NewError < Exception
end

raise NewError
```

? Mais pourquoi créer ses propres exceptions?

Créer ses exceptions offre plusieurs avantages. Premièrement, cela permet de donner plus de sémantique à notre code. Par exemple, si nous créons une exception `FileNotFoundError`, si une méthode lance une exception `FileNotFoundError`, on a plus d'informations que si c'était, disons une exception `RuntimeError`.

De plus, nous obtenons une certaine granularité au niveau du rattrapage d'exception puisque nous pouvons rattraper l'exception en utilisant son nom, et en particulier, nous pouvons avoir un traitement particulier si cette nouvelle exception est attrapée.

Créons une méthode qui prend un paramètre `x` et lance une exception si `x / 2` est supérieur à 2.

```ruby
def new_method(x)
  raise NewError, "#{x} / 2 > 2" if x / 2 > 2
  x
end
```

Nous pouvons alors rattraper spécifiquement l'exception `NewError` quand on appelle cette méthode.

```ruby
begin
  new_method(6) # => Lance exception NewError
rescue NewError => e
  puts "Exception : #{e.message}"
end
```

Et dans le même temps, si c'est un autre type d'exception qui est lancée, l'exception ne sera pas rattrapée.

```ruby
begin
  new_method('6') # => Lance exception NoMethodError
rescue NewError => e
  puts "Exception : #{e.message}"
end
```

III.5.3.3. Bonne gestion des exceptions

Après avoir vu comment bien rattraper les exceptions, il nous faut voir les bonnes pratiques quant au lancement et à la création des exceptions.

III.5.3.3.1. Quand créer des exceptions

Notons que le nombre d'exceptions standard est plutôt grand et que leurs noms sont suffisamment bien choisis pour couvrir la plupart des erreurs que l'on pourrait vouloir gérer. Dans la plupart des cas, elles seront donc suffisantes et nous allons favoriser l'usage d'exceptions standard.

Néanmoins, il ne faut pas hésiter à créer ses propres exceptions quand aucune des exceptions standards ne semble assez expressive pour l'erreur que l'on veut indiquer.

De plus, la plupart des temps, nous créerons nos exceptions en tant que classe-fille de `StandardError` plutôt que juste en tant que classe-fille de `Exception`. Nous donnons ainsi plus de sens à l'exception créée (c'est une exception standard) et nous permettons à celle-ci d'être rattrapée lorsque nous rattrapons les exceptions standards (rappelons que c'est une mauvaise pratique de rattraper **toutes** les exceptions).

III.5.3.3.2. Exceptions et espace de noms

Une bonne idée quand on crée des exceptions est de les avoir dans un espace de noms qui indique bien à quoi l'exception se rapporte. Ainsi, on évite les conflits de nommage d'exceptions. De plus, on peut ainsi spécifier plus précisément à quoi se rapporte une exception.

Ainsi, on ne laisserait l'espace de nom global que pour les exceptions standards. On pourrait par exemple avoir une exception `NameError` dans un module de gestion d'utilisateurs et ce sans avoir de conflits avec l'exception standard `NameError`.

III.5.3.3.3. Relancer une exception rattrapée

Un comportement que nous voudrons parfois avoir est le suivant: quand une certaine exception est levée, nous l'attrapons et la traitons, mais ne souhaitons pas qu'elle soit «oubliée» (en clair, nous voulons savoir que l'exception a été levée).

La solution à ce problème est de relancer l'exception dans le bloc `ensure`. Pour ce faire, il nous suffit de relancer l'exception. Et Ruby nous offre la possibilité de le faire simplement, juste avec le mot-clé `raise`. On peut alors écrire ce programme.

```
1  def main
2    loop do
3      str = gets.chomp
4      break if str == 'quit'
5      n = str.to_i
6      begin
7        puts "1 / #{n} = #{1 / n}"
8      rescue
9        puts 'Votre saisie est invalide !'
10       raise
11     end
12   end
13 end
14
15 begin
16   main
17 rescue => e
18   puts "Il y a une exception : e.message."
19 else
20   puts 'Aucune exception, bravo !'
21 ensure
22   puts 'Fin du programme !'
23 end
```

III.5.4. Exercices

Nous n'allons pas faire d'exercices spécifiques aux exceptions. Mais comme depuis quelques chapitres, nous pouvons nous lancer dans l'implémentation de petits projets. Bien sûr, à partir de maintenant, nous pouvons gérer les exceptions dans nos programmes (en fait, nous pouvons même reprendre certains de nos anciens programmes).

Conclusion

Ça y est, nous ne sommes plus les exceptions des Rubyistes, nous savons nous aussi gérer les erreurs. Pour plus de bonne pratiques liées aux exceptions, nous pourront lire ce guide de bonnes

pratiques .

— Les exceptions servent à gérer des erreurs imprévues (comme leur nom l'indique, ce sont des exceptions).
— Nous rattrapons les exceptions avec le mot-clé `rescue` et il vaut mieux préciser la classe de l'exception que l'on veut rattraper.
— Il est déconseillé de rattraper des exceptions qui ne descendent pas de `StandardError`.
— Le mot-clé `ensure` nous permet de nous assurer qu'un bout de code sera exécuté même si une exception vient interrompre le flux d'exécution du programme.

III.6. Les blocs

Introduction

Lorsqu'on entend parler de Ruby, les blocs sont souvent présentés comme une fonctionnalité qui permettra de faire le café, de faire du sport et de manger équilibré, et tout ça en même temps. En clair, ce serait une fonctionnalité clé de Ruby. Nous avons bien entendu déjà utilisé des blocs, mais dans ce chapitre nous allons les voir plus en profondeur et voir les notions qui se cachent derrière. En fait, nous allons plus généralement voir la notion de fermeture (*closure* en anglais).

III.6.1. La notion de fermeture

Rappelons-nous que nous sommes dans un langage où tout est objet. Les méthodes sont des objets, les classes sont des objets, les modules sont des objets, etc. Mais qu'en est-il du code? Vu qu'on veut que tous les éléments du langage soit des objets, pourquoi ne pas faire du code lui-même un objet?

Dans certains langages comme Smalltalk dont Ruby tire beaucoup de ses principes, c'est le cas, et Ruby reprend bien sûr ce principe. Ceci permet alors de traiter le code comme n'importe quel objet (c'est-à-dire qu'on pourra le passer en argument à des méthodes, utiliser des méthodes dessus, etc.

Néanmoins, il nous reste à savoir comment un objet peut être créé à partir d'un bout de code, et c'est ceci qui nous mène à la notion de fermeture.

Ici, nous allons rapidement faire un peu de théorie pour comprendre comment les fermetures fonctionnent. Nous pouvons lire cette partie sur les fermetures un peu rapidement, cela ne sera pas très dérangeant pour la suite de l'apprentissage.

III.6.1.1. Les fermetures

Pour commencer, voyons ce qu'est une fermeture. Prenons donc la définition de Wikipédia.

> Une fermeture ou clôture (en anglais: *closure*) est une fonction accompagnée de son environnement lexical. L'environnement lexical d'une fonction est l'ensemble des variables non locales qu'elle a capturé, soit par valeur (c'est-à-dire par copie des valeurs des variables), soit par référence (c'est-à-dire par copie des adresses mémoires des variables).

> *Wikipédia*

C'est un peu obscur. Décortiquons un peu tout ça. La phrase importante est la première, une fermeture est une **fonction** accompagnée de son **environnement lexical**. En gros, cela signifie que c'est un bout de code (en fait ce n'est pas nécessairement une fonction), accompagné de tout l'environnement qu'il y avait au moment où ce bout de code a été écrit (en particulier, toutes les variables qui existaient au moment de la création du bout de code).

L'idée derrière tout ça, c'est que le bout de code que l'on capture dans la fermeture a peut-être besoin de variables existantes pour fonctionner. Par exemple si nous créons une fermeture avec un bout de code qui affiche la valeur d'une variable `x`, cette fermeture a besoin de cette variable `x` pour «fonctionner».

Ainsi, une fermeture est en quelque sorte une structure contenant un bout de code (donc des instructions) et les variables existants à sa création.

Cela signifie notamment que les changements de variable qui ont lieu **après** la création de la fermeture n'affectent pas cette dernière.

```
x = 10
Création fermeture f avec le bout de code « afficher x »
x = 12
```

Ici, même si `x` a changé de valeur, le `x` de la fermeture vaut toujours `10`. L'environnement a été copié pour créer la fermeture et le `x` de la fermeture n'a plus rien à voir avec le `x` initial.

Si nous revenons du côté de Ruby, nous avons déjà utilisé des fermetures, et ce à chaque fois que nous avons utilisé des blocs.

```ruby
a = [1, 2, 3]
y = 5
a.each do |x|
  puts x
  puts y
end
```

Ici, nous avons créé une fermeture avec le bout de code `puts x puts y` et un environnement contenant les variables `a` et `y`.

III.6.1.2. Des fonctions d'ordre supérieur

Dans la théorie des langages de programmation, nous disons qu'une fonction est d'ordre supérieur si prend en paramètre des fonctions ou renvoie des fonctions. C'est une fonctionnalité plutôt puissante qui permet par exemple d'avoir ce genre de code en Python.

```python
def filter(ary, filter_function):
    result = []
    for x in ary:
        if filter_function(x):
            result.append(x)
    return result

def method(x):
    return x % 3 == 0

filter([2, 3, 5, 8, 12], method)
```

Nous avons une fonction `filter` qui prend en paramètre un tableau et une fonction `filter_function` et qui renvoie le tableau des éléments tels que `filter_function` renvoie `True`.

Plus généralement, nous allons dire que les fonctions d'un langage sont d'ordre supérieur si elles peuvent prendre en paramètre des fonctions et renvoyer des fonctions. De même, nous dirons qu'une entité du langage est de première classe si elle peut être passé en paramètre à une méthode et si elle peut être renvoyé à une méthode. Par exemple, les nombres, les chaînes de caractères ou encore les tableaux sont de première classe en Ruby.

III.6.1.3. Et Ruby dans tout ça ?

En fait, en Ruby tous les objets sont de première classe (on peut de manière générale passer n'importe quel objet à une méthode et retourner n'importe quel objet). En particulier, et c'est ce qui va nous intéresser, **les fermetures sont des objets de première classe** et peuvent donc être passés en paramètre à des méthodes.

Ainsi, lorsque nous écrivons `5.times { |puts 'Bonjour' }`, nous envoyons la fermeture contenant le code `puts 'Bonjour'` à la méthode `times`. C'est comme si nous lui avions donné une fonction `hello` en argument.

Les blocs nous permettent donc de «simuler» des fonctions d'ordre supérieur. Et en Ruby, nous n'avons donc pas besoin de fonctions d'ordre supérieur et nous n'avons pas besoin de pouvoir passer des méthodes en argument à d'autres méthodes. Remarquons d'ailleurs que nous ne pouvons pas le faire. Écrire le nom d'une méthode permet d'appeler la méthode. Nous ne pouvons donc pas écrire ceci.

```ruby
def filter(ary, filter_method)
  result = []
  ary.each { |x| result << x if filter_method(x) }
  result
end

def method(x)
```

```ruby
  return x % 3 == 0
end

filter([2, 3, 5, 8, 12], method)
```

En effet, écrire `filter(ary, method)` appelle `method` et donc est équivalent à exécuter `method`et appeler `filter` avec comme argument `ary` et le résultat de `method` (sachant qu'ici l'appel `method` va causer une erreur vu qu'elle prend un argument). Les fermetures nous offrent une solution à ce problème.

De plus, elles permettent de faire ce que l'on appelle des **fonctions anonymes**. Ce sont des fonctions que l'on va utiliser sans leur donner de nom. On parle aussi souvent de **fonction lambda**. Et c'est exactement ce qu'on fait lorsqu'on utilise un bloc. Reprenons l'exemple `5.times { puts 'Hello' }`; c'est comme si nous avions envoyé la fonction dont le corps est `puts 'Hello'` à `times`, mais sans jamais lui donner de nom.

Si nous reprenons l'exemple de `filter` en Python, on pourrait le réécrire comme ceci avec une fonction anonyme.

```python
def filter(ary, filter_function):
    result = []
    for x in ary:
        if filter_function(x):
            result.append(x)
    return result

def method(x):
    return x % 3 == 0

filter([2, 3, 5, 8, 12], lambda x: x % 3 == 0)
```

Avec ce chapitre, nous apprendrons à utiliser les fermetures pour faire de même en Ruby. Nous saurons alors écrire la méthode `filter` en Ruby de telle sorte que l'on puisse l'appeler ainsi.

```ruby
filter([2, 3, 5, 8, 12]) { |x| x % 3 == 0 }
```

Voilà pour la longue introduction. Il est maintenant temps de rentrer dans le vif du sujet et de voir comment manipuler ces fermetures. Il est à noter que Ruby dispose de différentes manière de gérer les fermetures. Elles diffèrent sur quelques points, mais ne sont pas très compliquées à appréhender.

III.6.2. Les blocs

III.6.2.1. Utiliser un bloc dans une fonction

Nous n'allons pas revoir comment donner un bloc en argument à une fonction, ce que nous voulons voir, c'est comment utiliser ce bloc dans la fonction. Pour illustrer cela, nous allons partir d'un exemple simple; notre but est d'écrire une méthode `day` qui exécute les actions d'une journée. Comme chaque journée est potentiellement différente, elle prendra en paramètre un bloc avec l'action du jour.

Pour commencer, faisons une méthode `day_description` qui affiche les actions de la journée. Nous lui passerons donc en paramètre une chaîne de caractère correspondant à l'action à effectuer.

```
1 def day_description(action_description)
2   puts "Se lever"
3   puts action_description
4   puts "Dormir"
5 end
6
7 day_desctiption('Travailler')
8 day_description('Regarder la télévision')
```

Ici, ça va. Maintenant, écrivons la méthode `day` qui n'affiche pas la description des actions, mais les fait. En particulier, elle devra aussi faire l'action passée en paramètre à la fonction, et nous allons donc utiliser un bloc. Pour utiliser un bloc passé en paramètre à une méthode, il suffit d'utiliser le mot-clé `yield`. Il n'y a rien d'autre à faire.

```
1 def day
2   wake_up
3   yield
4   spend_the_night
5 end
6
7 day { work }
8 day { watch_tv }
```

Pour tester notre code précédent, nous allons créer des méthodes `wake_up`, `work` qui… Affichent l'activité (oui, nous avions dit qu'on allait faire l'activité, mais bon…).

```
1 def wake_up
2   puts 'Baillement...'
3 end
4
```

```ruby
def spend_the_night
  puts 'Ron... Pi...'
end

def work
  puts 'Le pouvoir de la procrastination'
end

def watch_tv
  puts 'Ron... Pi...'
end
```

En fait, **toutes les méthodes prennent un bloc en paramètre**, c'est implicite. C'est juste qu'elles ne l'utilisent pas tous.

```ruby
puts('Hello') { puts 22 }
```

Ici, le message est affiché et le bloc est ignoré.

Nous pouvons appeler le bloc plusieurs fois si nous le voulons. Par exemple, modifions `day` pour faire l'activité deux fois dans la journée avec une petite pause déjeuner entre les deux.

```ruby
def day
  wake_up
  yield
  lunch_break
  yield
  spend_the_night
end
```

III.6.2.1.1. Savoir si un bloc est passé en paramètre

Il peut parfois être nécessaire de savoir si un bloc a été passé en paramètre. Pour ce faire, nous utilisons `block_given?` qui renvoie `true` si un bloc a été passé en paramètre. C'est une bonne pratique d'utiliser `block_given`. En effet, l'utilisation de `yield` sans bloc passé en paramètre crée une erreur «*LocalJumpError (no block given (yield))*». Nous corrigeons donc notre code de la manière suivante.

```ruby
def day
  wake_up
  block_given? ? yield : procrastinate
  lunch_break
```

```
5   block_given? ? yield : procrastinate
6   spend_the_night
7 end
```

Ici, si un bloc est donné, il est exécuté, sinon la méthode `procrastinate` est appelé.

III.6.2.2. Des blocs avec des paramètres

Nous avons déjà plusieurs fois utilisé des blocs avec paramètres. Pour exécuter un bloc en lui passant en paramètre, il suffit d'utiliser `yield` en lui donnant ce paramètre. Par exemple, rajoutons un argument `nom` à notre méthode `day`, le bloc prendra en paramètre ce nom.

```
1  def wake_up(name)
2    puts "#{name} wake up."
3  end
4
5  def work(name)
6    puts "#{name} fait semblant de travailler."
7  end
8
9  # etc.
10
11 def day(name)
12   wake_up(name)
13   block_given? ? yield(name) : procrastinate(name)
14   lunch_break(name)
15   block_given? ? yield(name) : procrastinate(name)
16   spend_the_night(name)
17 end
18
19 day('Nom') { |name| work(name) }
```

Et nous avons le résultat voulu. Ça y est, nous savons utiliser des blocs!

Les blocs ne vérifient pas le nombre d'arguments qui leur est passé. S'il y en a trop, le surplus est ignoré, et s'il n'y en pas assez, les arguments non fournis vaudront `nil`.

```
1 day('Nom') { work('Lui') }
2 day('Nom') { |name, n| n.times { work(name) } }
```

ici, avec le premier appel le bloc sera appelé avec l'argument `name` dans la fonction `day`, mais vu que le bloc ne prend aucun argument, il sera ignoré. Le deuxième appel, lui causera une erreur car `n` vaudra `nil` (et l'objet `nil` n'a pas de méthode `times`). Pour

pallier cela, nous pourrions donner une valeur par défaut à `n` et même à `name`.

```
day('Nom') { |name, n=1| n.times { work(name) } }
```

Et là, nous avons bien le résultat.

III.6.2.3. Appel explicite

Avec ce que nous avons écrit, nous ne pouvons pas, en voyant le prototype d'une fonction (son nom et ses arguments), savoir qu'elle attend un bloc en paramètre. Nous pouvons demander explicitement un bloc en argument. Pour cela, nous utilisons la syntaxe suivante.

```
def day(name, &block)
  wake_up(name)
  block_given? ? yield(name) : procrastinate(name)
  lunch_break(name)
  block_given? ? yield(name) : procrastinate(name)
  spend_the_night(name)
end
```

L'esperluette devant le nom du paramètre indique qu'il s'agit d'un bloc. Ce paramètre ne peut apparaître qu'en dernier, et bien sûr, il n'y en a qu'un seul (on ne peut passer qu'un seul bloc en argument d'une fonction).

Notons bien que ce paramètre ne force pas l'appel de la fonction avec un bloc; le bloc est toujours facultatif. Il nous permet juste, à nous, de savoir que la fonction attend sûrement un bloc. De plus, il nous permet d'appeler le bloc autrement qu'en utilisant `yield`, en utilisant la méthode `call` de la variable `block`.

```
def day(name, &block)
  wake_up(name)
  block_given? ? yield(name) : procrastinate(name)
  lunch_break(name)
  block_given? ? block.call(name) : procrastinate(name)
  spend_the_night(name)
end
```

III.6.2.4. Quelques subtilités

Maintenant que nous savons utiliser les blocs, revoyons quelques subtilités liées à leur utilisation.

Pour commencer, rappelons qu'une fermeture est un bout de code et son environnement (les variables locales, etc.). Sachant cela, que produit le code suivant.

```
def f(&block)
  x = 10
  block.call
end

x = 2
f { puts x }
```

Si nous avons bien suivi tout ce qui a été dit, le bloc a été créé avec son environnement et donc avec la variable `x` qui vaut 2, et c'est bien cette valeur qui est affichée.

En fait, c'est même plus fort; un bloc est censé contenir ce qui est nécessaire à son exécution (modulo les variables qu'on lui passe en paramètre). Ainsi, l'environnement dans lequel le bloc est évalué n'est même pas pris en compte lors de son évaluation, seul l'environnement capturé lors de sa création existe à ce moment. Le code suivant ne fonctionne donc pas. La variable `x` n'existe pas dans l'environnement du bloc.

```
def f(&block)
  x = 10
  block.call
end

f { puts x }
```

Par contre, nous pouvons avoir la variable `x` en paramètre du bloc.

```
def f(&block)
  x = 10
  block.call(x)
end

f { |x| puts x }
```

III.6.3. Proc et lambda

III.6.3.1. Réutiliser des blocs ?

Si nous voulons des fermetures, c'est notamment pour pouvoir les passer en argument à une fonction. Néanmoins, les blocs ont un petit problème: nous ne pouvons pas les récupérer dans

des variables (pour par exemple les réutiliser lors d'autres appels. Disons que nous avons par exemple une fonction `map` qui prend en paramètre un tableau et applique à chacun de ses éléments une fonction, et une autre `filter` (dont nous avons déjà parlé) qui ne garde que les éléments qui respectent une certaines conditions.

```
1  def map(ary, &block)
2    new_ary = []
3    ary.each { |x| new_ary << block.call(x) }
4  end
5
6  def filter(ary, &block)
7    new_ary = []
8    ary.each { |x| new_ary << x if block.call(x) }
9  end
10
11 a = [1, 2, 3, 4]
12 map(a) { |x| x.even? }    # => [false, true, false, true]
13 filter(a) { |x| x.even? } # => [2, 4]
```

Ici, nous voudrions bien garder le bloc dans une variable et le réutiliser, mais ce n'est pas possible. Avec le code qui suit, nous obtenons une erreur.

```
1  block = { |x| x.even? }
```

Quelqu'un de malin pourrait penser à faire une fonction qui prend en paramètre un bloc de manière explicite, et le retourne.

```
1  def create_block(&block)
2    block
3  end
```

En exécutant ce code dans une console, et en appelant cette méthode avec un bloc, nous constatons qu'elle nous renvoie un objet de la classe `Proc`. La classe de `block` serait donc `Proc`. Vérifions cela.

```
1  def block_class(&block)
2    block.class
3  end
```

Et en exécutant `block_class {}`, nous obtenons bien `Proc`

III.6.3.2. Les `Procs`

Une mystérieuse classe `Proc` apparaît au milieu de nos blocs. Mais qu'est-elle donc? En fait, nous ne pouvons pas réutiliser un bloc, et nous pouvouns quasiment dire que ce ne sont pas des objets réels. Ils n'existent que le temps de la méthode. Les `Proc`, eux, peuvent être liées à des variables et sont réutilisables. Pour définir un `Proc`, nous donnons un bloc en paramètre à `Proc.new` ou à `proc`.

Quant à son utilisation… Nous avons déjà utilisé des `Proc` lorsque nous utilisions la syntaxe explicite des blocs. La variable `block` que nous utilisions était un `Proc`. Nous pouvons alors écrire ce code.

```
1  def map(ary, proc)
2    new_ary = []
3    ary.each { |x| new_ary << proc.call(x) }
4  end
5
6  def filter(ary, proc)
7    new_ary = []
8    ary.each { |x| new_ary << x if proc.call(x) }
9  end
10
11 a = [1, 2, 3, 4]
12 proc_method = proc { |x| x.even? }
13 map(a, proc_method)   # => [false, true, false, true]
14 filter(a, proc_mehod) # => [2, 4]
```

Ici, il n'y a pas d'esperluette à l'argument `proc` des méthodes `map` et `filter`. Ce n'est pas un bloc, c'est un argument normal de la méthode. Cela signifie notamment qu'il est encore possible de passer un bloc à la fonction.

Notons qu'il est possible de créer un `Proc` à partir d'une méthode ou d'un symbole à l'aide de la méthode `to_proc`.

```
1  def execute(f, args)
2    f.call(args)
3  end
4
5  execute(:puts.to_proc, "Bonjour")
```

La classe `Proc` nous permet donc d'obtenir des fonctions d'ordre supérieur.

III.6.3.2.1. De bloc à `Proc`

Revenons un peu sur la relation bloc-`Proc`.

?

Que se passe-t-il quand nous écrivons `def f(args, &block)`? Pourquoi partons-nous d'un bloc, pour finalement obtenir un `Proc`.

Il y a tout simplement conversion du bloc en `Proc`. L'esperluette devant `block` indique à Ruby que nous voulons récupérer le bloc dans une variable, et il le convertit donc en `Proc` (les blocs ne peuvent pas être stockés dans une variable).

L'opérateur `&` permet également de faire la conversion inverse et de passer de `Proc` à bloc. Ceci est utile si nous voulons utiliser un `Proc` avec une méthode qui prend en paramètre un bloc.

```
1 fun = proc { |x| print "#{x + 1} "}
2 5.times(&fun) # => 1 2 3 4 5
```

Notons également qu'avec un objet qui n'est ni un `Proc`, ni un bloc, l'opérateur unaire tente d'appeler la méthode `to_proc` de l'objet pour ensuite transformer le `Proc` correspondant en bloc. Ceci nous permet d'utiliser cette syntaxe plutôt concise.

```
1 ary = [2, 1, 32, 6]
2 map(ary, &:to_s) # => ["2", "1", "32", "6"]
```

La méthode `to_s` est appelée pour chaque élément du tableau. En fait, écrire `:method` revient à écrire le Proc suivant.

```
1 Proc.new { |x, arguments| x.method(arguments) }
```

Et donc, pour notre exemple avec `map`, le bloc `{ |x| x.to_s }` est passé à la méthode.

III.6.3.3. Les lambdas

Un autre moyen de créer une fermeture à partir d'un bloc est de créer une lambda. Elle se crée à partir de deux syntaxes.

```
1 fun1 = lambda { |x| x + 1 }
2 fun2 = -> (x) { x + 1}
```

La première syntaxe emploie le mot-clé `lamda` suivi du bloc à transformer en lambda. La seconde suit la forme `-> (arg1, arg2) block`. L'utilisation est ensuite la même que celle des `Proc`. Ces objets sur lesquels nous pouvons appeler `call` sont dits *callables* (nous pouvons les appeler).

```
1 fun2 = -> (x) { x + 1}
2 puts fun2.call(4)
```

Nous préférerons utiliser la seconde syntaxe avec des blocs sur une ligne, et le mot-clé `lambda` avec des blocs multi-lignes.

III.6.3.3.1. Quelles différences entre `Proc` et lambda ?

La question se pose en effet. Pourquoi avoir ces deux objets alors qu'ils se comportent de la même manière?

Pour commencer, en regardant la classe d'une lambda, nous constatons qu'il s'agit d'un objet de la classe `Proc`. En fait, si nous regardons la valeur d'une `lambda` et que nous la comparons à celle d'un `Proc`, nous constatons qu'une `lambda` est un objet de la classe `Proc` mais avec une information indiquant que c'est une `lambda`.

```
1 ld = -> (x, y) { x + 2 * y }
2 pr = Proc.new { |x, y| x + 2 * y }
3 puts ld  # => #<Proc:value (lambda)>
4 puts pr  # => #<Proc:value>
```

Mais ça ne répond pas à notre interrogation sur leurs différences. En fait, il y a deux différences majeures entre les `Proc` et les lambdas.

III.6.3.3.1.1. Vérification du nombre d'arguments La première différence est que les lambdas vérifient que le nombre d'arguments passés est correct alors que les `Proc` met les paramètres non passés à `nil`.

```
1 ld = -> (x) { puts x }
2 pr = Proc.new { |x| puts x}
3
4 pr.call
5 ld.call # ArgumentError (wrong number of arguments)
```

III.6.3.3.1.2. Le comportement du `return`

Lorsqu'un `return` est croisé dans un lambda, il interrompt l'exécution de la fonction qui l'avait appelé', ce n'est pas le cas des `Proc`.

```ruby
def f
  ld = -> { return 10 }
  ld.call
  puts "Ici"
  return 0
end

f # => 0

def g
  pr = Proc.new { return 10 }
  pr.call
  puts "Ici"
  return 0
end

g # => 10
```

Ici, lors de l'appel de `f`, l'affichage est fait et la méthode renvoie `0`, lors de l'appel de `g`, ce n'est pas le cas.

En fait, c'est même pire, un `return` dans un `Proc` est lié à l'endroit de sa création. Dit comme ça c'est un peu obscur, mais un exemple permettra de comprendre.

```ruby
def f(pr)
  pr.call
  return 0
end

pr = Proc.new { return 10 }
f(pr)
```

Ici, nous obtenons une erreur `LocalJumpError (unexpected return)` car le `return` n'est pas liée à la fonction `f` puisque le `Proc` a été créé en dehors.

De même dans le code qui suit, puisque le `Proc` a été créé dans la méthode `f`.

```ruby
def f
  Proc.new { return 10 }
end

pr = f
pr.call
```

Finalement, les lambdas peuvent plus être vus comme de vrais fonctions du premier ordre dans le sens ou un `return` va agir comme dans une méthode et que le nombre de paramètres est vérifié.

Les cas où nous aurons vraiment besoin de faire cette distinction sont plutôt rares; en fait ce sont les blocs que nous utiliserons le plus fréquemment. Dans les rares cas où nous devront choisir entre `Proc` et lambda, il nous suffira généralement de nous dire que les lambdas agissent comme de vraies méthodes.

III.6.4. Exercices

Les fermetures sont finalement très utiles et nous pouvions déjà le constater en voyant l'utilité des blocs. Chose n'est pas coutume, nous allons reprendre une vielle habitude et faire dans ce chapitre quelques méthodes pour nous entraîner à utiliser les fermetures.

III.6.4.1. Exercice 1

Pour commencer, écrivons une fonction `min(a, b, &block)` qui renvoie le minimum de `a` et `b` d'après le bloc passé en paramètre. Par exemple, `min(-2, 1, &:abs)` doit renvoyer `1`).

Correction.

 Contenu masqué n°25

III.6.4.2. Exercice 2

Continuons avec un exercice simple. Écrire une méthode qui prend en paramètre deux méthodes `f` et `g` et renvoie la méthode `h` qui fait somme de ces deux méthodes (*ie.* `h(x) = f(x) + g(x)`).

Correction.

 Contenu masqué n°26

III.6.4.3. Exercice 3

Toujours dans la veine du deuxième exercice, nous allons cette fois écrire une méthode qui compose deux méthodes. Rappelons que la composition de deux méthodes `f` et `g` est la méthode `h` telle que `h(x)` vaut `f(g(x))`.

Correction.

Contenu masqué n°27

III.6.4.4. Exercice 4

Nous allons prendre le troisième exercice, et le complexifier un peu. Cette fois, nous voulons une méthode qui prend en paramètre une liste de fonctions `[f0, f1, ...]` et renvoie la fonction composition `f0(f1(...))`. Par convention, nous posons que si la liste est vide, la fonction à renvoyer ne fait rien et renvoie l'argument (c'est la fonction identité).

Correction.

Contenu masqué n°28

Bien sûr, nous pouvons aussi faire la somme d'une liste de fonctions par exemple.

Conclusion

Dans ce chapitre, nous avons démystifié les blocs, et ajouté à notre panoplie du Rubyiste les `Proc` et les lambdas.

- Une fermeture est un objet construit avec un bout de code et son environnement.
- Les fermetures permettent de simuler des fonctions d'ordre supérieure, c'est-à-dire qu'on peut les passer en paramètre à des méthodes.
- L'opérateur unaire `&` permet de passer de `Proc` à bloc et inversement.
- L'opérateur unaire `&` permet aussi de créer un bloc à partir d'une méthode (avec `&:method_name`).

Contenu masqué

Contenu masqué n°25

L'idée est bien sûr de comparer les valeurs renvoyées par l'appel du bloc plutôt que les valeurs `a` et `b` passées en paramètre. Ici, nous décidons de comparer les deux valeurs si aucun bloc n'est donné.

```
1 def min(a, b, &block)
2  return (block.call(a) < block.call(b) ? a : b) if block_given?
3  (a < b) ? a : b
4 end
```

Retourner au texte.

Contenu masqué n°26

Ici, nous n'allons pas utiliser de blocs, mais directement demander des `Procs` ou des lambdas (en tout cas des objets *callables*) en paramètre. Ça paraît plus logique de faire ainsi (notre méthode prend en gros deux méthodes sous la forme de fermeture et renvoie une troisième méthode sous la forme d'une fermeture).

```
def sum(f, g)
  -> (arg) { f.call(arg) + g.call(arg) }
end
```

Nous renvoyons une lambda, en considérant qu'on doit bien donner le bon nombre d'arguments à la fermeture qu'on renvoie et que si dans la fermeture renvoyée on fait un `return`, on veut le même comportement que dans une méthode normale. Retourner au texte.

Contenu masqué n°27

L'idée est la même que pour l'exercice 2, notre fonction devra appeler `f` avec comme argument la valeur retournée par l'appel de `g` sur l'argument initial.

```
def composition(f, g)
  -> (arg) { f.call(g.call(arg)) }
end
```

Retourner au texte.

Contenu masqué n°28

Cet exercice est un bon exemple d'utilisation de récursivité. Ici, soit la liste est vide (on le teste avec `fun_list.empty?`), soit ce n'est pas le cas, auquel cas, il faut faire ceci.

1. Calculer le résultat `r` de l'appel de la composition des fonctions sauf la première avec l'argument `arg`.
2. Appeler la première fonction avec l'argument `r`.

En effet, si nous calculons `r = f1(f2(...))`, alors nous voulons juste renvoyer `f0(r)`. Et pour calculer `f1(f2(...)))`, nous allons faire la même chose et calculer `f2(f3(...))`, et ainsi de suite jusqu'à tomber sur `fn(arg)` qui va lui s'évaluer en `fn.call(composition([]).call(arg))`, et `composition([])` est l'identité, donc nous aurons `fn.call(arg)`, et nous pouvons alors remonter pour avoir le résultat final. Pour de plus amples et explicites explications, nous pourrons nous renseigner sur la récursivité .

```
def composition(fun_list)
  return -> (arg) { arg } if fun_list.empty?
  -> (arg) {
      fun_list[0].call(composition(fun_list[1..-1].call(arg)) }
end
```

Retourner au texte.

Quatrième partie

Les outils de Ruby

Introduction

La première partie du tutoriel nous apprenait les bases du langage et de sa syntaxe; la deuxième partie nous apprenait à organiser notre code; la troisième partie, elle, complète notre arsenal en rajoutant plusieurs choses à notre boîte à outils du programmeur. Nous verrons notamment quelques nouveaux éléments de la bibliothèque standard de Ruby (expressions régulières, gestion de fichiers, etc.) et en approfondiront d'autres (méthodes des classes usuelles, etc.).

Cela signifie notamment que cette partie ne doit pas forcément être lue linéairement. Nous pouvons parfaitement sauter à un chapitre qui nous intéresse particulièrement. De plus, comme nous l'avons dit dans l'introduction de la partie 2, cette partie peut être lue en parallèle de la partie 2, même si certains chapitres nécessitent des connaissances de chapitres de cette partie 2.

IV.1. Les objets énumérables

Introduction

Dans ce chapitre, nous allons traiter d'objets énumérables. Un objet énumérable est tout un objet qu'on peut parcourir (les tableaux ou les hachages par exemple). Nous allons voir le module `Enumerable` qui donne accès à beaucoup de méthodes sur ces objets, puis nous verrons comment créer nos propres objets énumérables.

IV.1.1. Le module Enumerable

Comme nous l'avons dit en introduction, le module `Enumerable` est destiné aux objets qui savent énumérer leur contenu; il leur rajoute de nombreuses méthodes. En fait, ce module est inclus dans les classes de ces objets. 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 et en fait pour tous les objets disposant d'une méthode `each`. Nous conseillons d'utiliser IRB pour faire les tests sur `Enumerable`.

IV.1.1.1. Vérifier des conditions sur un tableau

IV.1.1.1.1. 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).

```
enumerable = [1, 2, 3]
is_even = enumerable.all? do |e|
  print e
  e.even?
end
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.
```

IV.1.1.1.2. 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.
```

IV.1.1.1.3. 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.

```ruby
enumerable = [1, 2, 3]
enumerable.any? { |e| e.even? }  # => true, car 2 est pair.
```

IV.1.1.1.4. 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`.

```ruby
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.
enumerable2.one? { |e| e == 1 }  # => false, il y a deux 1 dans le
    tableau.

[false, nil, true].one?          # => true, seul true est évalué à
    true.
```

IV.1.1.1.5. 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.

```ruby
enumerable = [1, 2, 3]
enumerable.include?(2)  # => true
enumerable.include?(5)  # => false
```

IV.1.1.2. Récupérer des éléments du tableau

IV.1.1.2.1. La méthode `find`

Cette méthode permet de chercher le premier élément satisfaisant un critère donné.

```ruby
enumerable = [1, 2, 3]
n = enumerable.find { |e| e.odd? }  # => 1
```

IV.1.1.2.2. La méthode `select`

Cette fois ci, cette méthode retourne tous les éléments satisfaisant un critère.

```ruby
enumerable = [1, 2, 3]
enumerable.select { |e| e.odd? }  # => [1, 3]
```

IV.1.1.2.3. 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`.

```ruby
enumerable = [1, 2, 3]
enumerable.reject { |e| e.even? }  # => [1, 3]
```

IV.1.1.2.4. 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.

```ruby
enumerable = [1, 2, 3]
enumerable.map { |e| e + 1 }  # => [2, 3, 4].
```

IV.1.1.3. Autres méthodes utiles

IV.1.1.3.1. 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.

```ruby
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.

IV.1.1.3.2. 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.

```ruby
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.
```

IV.1.1.3.3. 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.

```ruby
enumerable = [1, 2, 3]
enumerable.first     # => 1
enumerable.first(2)  # => [1, 2]
```

IV.1.1.3.4. 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.

IV.1.1.3.5. 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 }
```

IV.1.1.3.6. 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 }
```

IV.1.1.3.7. 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"]
```

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.

```
enumerable = [1, 2, 3, 4, 5, 6, 7, 8, 9]
enumerable.sort_by { |a| a - (a - 4) * (a - 4) }
```

IV.1.1.3.8. 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`.

IV.1.1.3.9. 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`. Elle fait la somme des éléments et prend en paramètre facultatif la valeur initiale (qui est 0 par défaut). On pourrait par exemple additionner des tableaux en écrivant ceci.

```
[1, 2, 3].sum                          # => 6
enumerable = [[1, 2], [3, 4], [5, 6]]
enumerable.sum([])                     # => [1, 2, 3, 4, 5, 6]
# enumerable.sum est incorrect, au début il essaye d'additionner 0
    et [1, 2].
```

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.

IV.1.2. Créer des objets énumérables

IV.1.2.1. Mixer le module `Enumerable`

Nous avons dit précédemment que le module `Enumerable` était «destiné aux éléments qui savent **déjà** comment énumérer leur contenu» (avec la méthode `each`). En fait, ce module est juste mixé à ces classes; les méthodes de ce module partent simplement du postulat suivant: je serais inclus dans une classe qui possède une méthode `each` et je peux donc utiliser cette méthode `each` pour faire mon travail.

La seule chose à faire pour pouvoir utiliser le module `Enumerable` sur une classe est de lui fournir une méthode `each`.

```
class C
  include Enumerable

  def each
  end
end
```

Et là, nous avons accès à toutes les méthodes.

```
c = C.new
c.each { |n| puts n }
puts c.sum
c.select { |n| n > 2 }
```

Ça fonctionne… Mais ce n'est pas très intéressant. Normal, notre méthode `each` ne fait rien. En fait, pour faire quelque chose d'intéressant, il nous faut une méthode `each` qui exécute un bloc passé en paramètre avec les éléments que l'on veut.

```
class C
  include Enumerable

  def each
    yield 2
    yield 5
    yield 3
  end
end
```

Et là on le code précédent est déjà plus intéressant. Le `sum` nous donne 10, le `select` nous renvoie `[2, 3]`, etc. En fait, ici on a simplement fait une méthode `each` qui permet de parcourir les éléments `2`, `3` et `5`.

IV.1.2.2. Un exemple

Cet exemple n'est pas très intéressant. Allons un peu plus loin. Nous allons gérer les utilisateurs d'un serveur de discussion. Un serveur est composé de plusieurs salles, et dans chaque salle, il y a des utilisateurs (des chaînes de caractères pour les représenter). Nous allons écrire `each` de `Server` de manière à pouvoir itérer sur tous les utilisateurs du serveur.

```ruby
class Room
  attr_reader :users

  def initialize(users=[])
    @users = users
  end

  def each
    @users.each { |u| yield u }
  end
end

class Server
  include Enumerable
  attr_reader :rooms

  def initialize(rooms=[])
    @rooms = rooms
  end

  def each(&block)
    @rooms.each { |r| r.each(&block) }
  end
end
```

Nous pouvons alors écrire ceci.

```ruby
r1 = Room.new(["User5", "User2", "User3"])
r2 = Room.new(["User1", "User6", "User4"])
s = Server.new([r1, r2])

puts s.count # Affiche le nombre d'utilisateurs
puts s.select { |u| u[0] == "s" }
puts s.sort
```

Ici, nous avons donc inclus `Enumerable` dans `Server` ce qui nous permet de faire des opérations sur les utilisateurs du serveur. Notons que nous ne l'avons pas inclus dans `Room` (nous aurions pu le faire). Ainsi, `puts r1.count` ne fonctionnera pas par exemple.

IV.1.2.3. Des structures de données

Pour bien illustrer le fonctionnement de `Enumerable`, nous allons créer des structures de données et les parcourir.

IV.1.2.3.1. Liste chaînée

```ruby
class LinkedList
  include Enumerable
  attr_reader :head, :tail

  def initialize(head, tail = nil)
    @head = head
    @tail = tail
  end

  def to_s
    "[#{@head}#{", " if !@tail.nil?}#{@tail}]"
  end

  def each(&block)
    block.call(@head)
    @tail.each(block) if @head
  end
end
```

IV.1.2.3.2. Créer un arbre binaire de recherche

sert aussi d'exercices arbre généalogique (=> plus binaire).

IV.1.3. La classe Enumerator

IV.1.4. Exercices

Recoder quelques méthodes de Enumerable?

IV.2. Les expressions régulières

Introduction

Dans ce chapitre, nous allons traiter des expressions régulières. Le système d'expressions régulières est un système très puissant qui nous permet de faire des opérations sur les chaînes de caractères. Avec lui, il est possible defaire ceci.

1. Chercher un motif (*pattern* en anglais) dans la chaîne (par exemple, chercher les expressions entre guillemets).
2. Traiter (éventuellement) les éléments trouvés. Nous pouvons les remplacer, les extraire, les récupérer, etc.

Par exemple, nous pourrons faire une expression régulière qui vérifie qu'une adresse courriel est valide.

IV.2.1. Construire une expression régulière

Pour commencer, il nous faut voir ce qu'est une expression régulière. Nous avons vu en introduction ce à quoi elles pouvaient servir, mais nous ne savons toujours pas comment elles se présentent. En fait, nous pouvons voir une expression régulière comme un modèle. Certaines chaînes correspondent au modèle, d'autres non. Par exemple, le mot «chat» correspond au modèle des mots qui commencent par un «c» et se terminent par un «t», mais ne correspond pas à celui des mots qui commencent par un «c» et comportent un «y».

Chercher un motif dans une chaîne signifie simplement vérifier que la chaîne testée correspond au modèle. Il ne nous reste plus qu'à voir à quoi ressemblent ces modèles et comment les écrire.

Notons qu'une expression régulière est de la classe `Regexp` (pour *regular expression*).

IV.2.1.1. Tester la présence d'un mot dans une chaîne

Le moyen le plus simple d'écrire une expression régulière est de l'écrire entre `/`. Pour tester si un mot correspond à cette expression, nous utilisons ensuite le symbole `=~`. Par exemple, pour vérifier qu'une chaîne contient le mot «chat», nous écrirons ceci.

```
1 /chat/ =~ 'Le chat est sur la table.'
```

Ici, `/chat/` est une expression régulière, et ce code vérifie que cette expression est présente dans la chaîne.

Cette expression ne renvoie pas un booléen comme nous pouvions nous y attendre. Elle renvoie l'indice du caractère où la correspondance a été établie. Ici, le `c` est le quatrième caractère, soit le caractère d'indice 3.

? Et si notre chaîne ne correspond pas au modèle, il se passe quoi?

Essayons, et voyons ce qui se passe.

```
1 /chat/ =~ 'Le chien est sur la table.'
```

La chaîne ne comprend pas le mot «chat», l'expression renvoie `nil`.

Notons que nous pouvons inverser l'écriture. Il est équivalent (à un détail près) d'écrire `'str' =~ /regex/` et `/regex/ =~ 'str'`. Nous pouvons donc choisir celle qui nous plaît le plus.

Avec `!~`, nous faisons l'opération inverse, à savoir que nous vérifions que notre chaîne ne contient pas le modèle testé. Cette fois, c'est un booléen qui est retourné (`true` si la chaîne ne vérifie pas le modèle, `false` sinon).

```
1 /chat/ !~ 'Le chien est sur la table.'  # => true
2 /chat/ !~ 'Le chat est sur la table.'   # => false
```

La méthode `match?` permet également de savoir si une chaîne correspond au modèle. Elle s'utilise indifféremment sur les chaînes de caractères ou les expressions régulières et renvoie `True` s'il y a correspondance et `false` sinon.

```
1 /chat/.match('Le chien est sur la table.')  # => false
2 'Le chat est sur la table.'.match(/chat/)   # => true
```

! Ces exemples ne sont présents que pour introduire les expressions régulières. Si dans un programme, nous volons vérifier la présence d'une chaîne dans une autre chaîne, nous utiliserons plutôt cette syntaxe.

```
str = 'Le chat est sur la table.'
str['chat']   # => 'chat'
```

Les expressions régulières sont à utiliser dans le cas où l'on a un motif plus compliqué à rechercher.

Dans le cas où l'on a une expression régulière simple et qu'on veut obtenir la sous-chaîne qui y correspond dans une chaîne, nous pouvons utiliser la syntaxe `str[regex]`. Elle renvoie `nil` si la chaîne ne vérifie pas l'expression régulière et renvoie la partie de la chaîne qui la vérifie sinon.

```
str = 'Le chat est sur la table'
str[/chien/]   # nil
str[/chat/]    # chat
```

Notons que nous pouvons également créer une expression régulière avec la syntaxe `%r`. Dans ce cas, nous privilégierons les accolades comme délimiteur. Nous n'utiliserons cette syntaxe que quand notre expression régulière contiendra le caractère `/`.

IV.2.1.2. Les symboles

Maintenant que nous connaissons la syntaxe de base, nous pouvons nous pouvons nous lancer à la découverte de ce qui fait la puissance des expressions régulières.

IV.2.1.2.1. Les quantificateurs

Jusqu'à maintenant, nous testions un nombre exact de caractères. Mais nous pouvons aussi chercher si quelque chose apparaît plusieurs fois dans une chaîne.

Il y a trois symboles qui servent de quantificateur.

— Le symbole `*` indique que le caractère qu'il suit doit être présent zéro, une, ou plusieurs fois.
— Le symbole `+` indique que le caractère qu'il suit doit être présent au moins une fois.
— Le symbole `?` indique que le caractère qu'il suit doit être présent au plus une fois.

Ainsi, l'expression régulière `chats?` est valide pour les mots `chat` et `chats`.

Ces caractères sont appelés métacaractères. Ce sont des caractères qui ont une signification particulière dans la construction des motifs de recherche. Nous ne pouvons donc pas les utiliser tels quel si nous voulons les rechercher. Pour les rechercher, nous les échappons avec une barre oblique (par exemple, la présence d'un point d'interrogation se teste avec `\?`).

Nous pouvons également rechercher un caractère un certain nombre de fois grâce aux intervalles de reconnaissance. Pour cela, nous utilisons les accolades (qui sont aussi des métacaractères). Nous pouvons les utiliser de plusieurs manières.

— `a{n}` signifie que `a` doit être présent `n` fois.
— `a{n,}` signifie que `a` doit être présent au moins `n` fois.
— `a{,n}` signifie que `a` doit être présent au plus `n` fois.
— `a{n,m}` signifie que `a` doit être présent entre `n` et `m` fois.

Nous ne pouvons pas utiliser d'espaces dans les intervalles de reconnaissance. `"aaa" =~ /{2, }/` renverra `nil` contrairement à `"aaa" =~ /{2,}/` qui renverra `0`.

Avec tout ça, nous pouvons déjà écrire des expression régulières intéressantes. Par exemple, nous pouvons tester l'existence de caractères entre `<` et `>` dans une chaîne de caractères avec l'expression `<.*>`. Nous regardons s'il y a `<` puis n'importe quels caractères et enfin `>`.

IV.2.1.2.2. Les ancrages

Parfois, nous voulons chercher quelque chose à une position particulière de la chaîne vérifiée. Par exemple, nous pouvons vouloir vérifier que notre chaîne se termine par «er». Pour cela, nous allons utiliser `\z`.

```
1 "Aller" =~ /er\z/  # => 3
2 "Finir" =~ /er\z/  # => nil
```

Ce symbole permet de préciser la recherche et il n'est pas le seul.

— `\A` signifie le début de la chaîne.
— `\Z` signifie tout comme `\z` la fin de la chaîne. Cependant, dans le cas où la chaîne se termine par un retour à la ligne, `\Z` ne prend pas en compte ce retour à la ligne. Ainsi, `"Aller\n" =~ /er\z/` vaut `nil` alors que `"Aller\n" =~ /er\Z/` vaut `3`.
— `^` signifie le début d'une ligne. Si, notre chaîne est composée de plusieurs lignes, ce symbole permet de vérifier ce qui se trouve au début de chacune de ces lignes.
— `$` signifie la fin d'une ligne.

Le dollar, l'accent circonflexe et la barre oblique sont, comme nous pouvions nous en douter, des métacaractères. Si nous voulons tester la présence d'un de ces caractères, il ne faudra pas oublier de l'échapper.

Nous pouvons par exemple imaginer un fichier où l'on range des gens avec leur numéro. Ce fichier est organisé de la manière suivante.

```
NOM PRÉNOM NUMÉRO
NOM PRÉNOM NUMÉRO
etc.
```

Imaginons la situation suivante. Nous voulons récupérer le numéro d'une personne dont le nom se termine par «arnaj», mais nous avons oublié la première lettre de son nom. Pour connaître le nom de la personne et récupérer son numéro, nous cherchons dans le fichier une ligne qui correspond à ce qu'on cherche (pour cela, nous pourrons au préalable charger tout le contenu du fichier dans une chaîne de caractères ou encore créer un tableau avec une case par ligne du fichier).

```
regex = /^.arnaj /
str =~ regex
```

Nous cherchons, au début d'une ligne, n'importe quelle caractère suivie de «arnaj».

IV.2.1.2.3. Les classes de caractères

Les classes de caractères nous permettent de tester plusieurs caractères. Nous pouvons regarder si un caractère est dans une liste de caractères en utilisant les crochets. Ainsi, `[oe]` signifie «un o ou un e». Pour rechercher «bon» ou «ben», nous utiliserons l'expression régulière `b[oe]n`.

```
'bon' =~ /b[oe]n/  # => 0
'ben' =~ /b[oe]n/  # => 0
```

Nous pouvons spécifier un intervalle plutôt que de lister toutes les possibilités grâce au symbole `-`. Ainsi, nous pouvons rechercher un caractère entre `a` et `m` ou un chiffre entre `2` et `7` sans écrire toutes ces lettres.

```
'bon' =~ /b[a-m]n/      # => 0
'187' =~ /[2-7]/        # => 2
'abo34' =~ /[a-m2-7]/   # => 0
```

Avec la dernière expression, nous recherchons un caractère entre `a` et `m` ou entre `2` et `7`.

Ceci nous permet de faire les vérifications suivantes.

— Vérifier qu'un caractère est alphabétique, avec `[a-zA-Z]`.
— Vérifier qu'un caractère est alphabétique et minuscule avec `[a-z]`.
— Vérifier qu'un caractère est numérique avec `[0-9]`.
— Vérifier qu'un caractère est alphanumérique, avec `[a-zA-Z0-9]`.

Avec ce que l'on sait, nous pouvons chercher l'existence d'une balise `<hn>` dans un texte avec `n` un chiffre entre 1 et 6 (balises de titres en HTML) avec l'expression régulière `<h[1-6]>`.

Si nous voulons placer `-` dans notre classe de caractères, il nous faudra le placer en premier. Dans le cas contraire, il sera considéré comme un séparateur pour un intervalle. Ainsi `[-15]` signifie «-», «1» ou «5», alors que `[1-5]` signifie un chiffre entre «1» et «5».

IV.2.1.2.4. Le complémentaire

Parfois, il est plus simple de donner la liste des caractères qu'on ne veut pas plutôt que ceux qu'on veut. Et c'est possible. Par exemple, si nous voulons n'importe quel caractère sauf la lettre `a`, nous écrirons `[^a]`. En utilisant le symbole `^`, nous indiquons que nous voulons tous les caractères sauf ceux énumérés.

```
1 "voiture" ~= /[^aeiouy]/ # => 0
```

Nous pouvons déjà construire des expressions compliquées.

```
1 regex = / [aeiouy]{2}[^aeiouy][aeiouy]{3} /
```

Ici, la chaîne sera valide si elle contient un mot de deux voyelles suivies d'une consonne puis de trois voyelles, ce mot devant être entouré d'espaces.

```
1 "Voici une phrase." =~ regex  # => nil
2 "Un oiseau vole." =~ regex    # => 2
```

IV.2.1.2.5. Classes préexistantes

Mais pour nous faciliter la vie, des classes ont été inventées. Ainsi, nous pouvons vérifier plus facilement qu'un caractère est alphabétique, est numérique, etc.

— Pour indiquer que le symbole peut être n'importe quel caractère, nous utilisons le symbole `.` (il est vérifié pour tous les caractères sauf le retour à la ligne). Le point est l'un des métacaractères les plus utiles. Ainsi, pour vérifier qu'un mot de trois lettres a le `o` comme lettre du milieu, nous utiliserons `.o.`.

```
1 'bon' =~ /.o./  # => 0
2 'ben' =~ /.o./  # => 0
```

— `\w` signifie un caractère alphanumérique. Il est équivalent à `[a-zA-Z0-9_]` (notons qu'il y a les lettres mais aussi le tiret bas).
— `\d` signifie un caractère numérique. Il est équivalent à `[0-9]`.
— `\s` signifie un caractère d'espacement. Il est équivalent à `[ \t\r\n\f\v]`.

En mettant la lettre en majuscule, nous obtenons la classe complémentaire. Ainsi, `\W` signifie un caractère non alphanumérique (équivalent à `[^a-zA-Z0-9_]`) et `\D` signifie un caractère non numérique (équivalent à `[^0-9]`)

En utilisant cela, nous pouvons faire une expression régulière qui vérifie qu'une chaîne contient un numéro de téléphone.

```
regex = /0\d{9}/
str = "0123456789"
str =~ regex # => 0
```

Nous cherchons d'abord un zéro, puis neuf chiffres quelconques. Nous pouvons faire mieux et vérifier qu'une chaîne contient quelque chose comme ça `<nom> : <numéro>`.

```
regex = /\w+ : 0\d{9}/
str = "Karnaj : 0123456789"
str =~ regex  # => 0
```

Nous avons rajouté `\w+ :` avant le numéro: nous cherchons un mot et les deux-points avant le numéro.

Nous dispose de plusieurs autres classes. Pour avoir plus d'informations à leur sujet, nous pouvons aller regarder la documentation . Nous noterons les classes suivantes.

— `[[:alpha:]]` signifie un caractère numérique. Contrairement à `\w`, il ne prend pas en compte le tiret bas. Il est donc équivalent à `[a-zA-Z]`
— `[[:alnum:]]` signifie un caractère alphanumérique. Tout comme `[[:alpha:]]`, il ne prend pas en compte le tiret bas, et est donc équivalent à `[a-zA-Z0-9]`.
— `[[:lower:]]` signifie un caractère alphabétique en minuscule (équivalent à `[a-z]`) et `[[:upper:]]` signifie un caractère alphabétique en majuscule (équivalent à `[A-Z]`).
— `[[:punct:]]` signifie un symbole de ponctuation.

Pour tester qu'une chaîne contient un mot de quatre lettres précédées d'un espace et suivie d'une ponctuation, nous pouvons alors utiliser l'expression régulière suivante.

```
regex = / [[:alpha:]]{4}[[:punct:]]/
"Ça marche pas avec cette chaîne" =~ regex  # => nil
"Pour elle, oui" =~ regex                   # => 4
```

IV.2.1.2.6. L'alternative

Supposons que l'on veuille vérifier qu'un mot contienne un «b» suivi de «eau» ou de «on» (soit vous êtes beau, soit vous êtes bon) puis d'un point. Pour ce faire, nous allons utiliser le métacaractère `|` qui nous permet de représenter une alternative.

```
1 regex = /on|eau\./
2 "Voici de l'eau."[regex]  # => eau
3 "C'est un don."[regex]    # => on
```

En écrivant `/bon|eau/`, les deux possibilités sont «bon» et «eau». Si l'on veut que le «b» ne soit pas dans le premier choix, il nous faut isoler les choix en les entourant de parenthèses.

Ainsi, avec l'expression régulière qui suit, nous reconnaissons «bon» ou «beau».

```
1 regex = /b(on|eau)\./
2 "On est beau."[regex]  # => "beau"
3 "On est bon."[regex]   # => "bon"
```

Nous pouvons proposer plus d'un choix, il suffit d'utiliser le métacaractère `|` autant de fois que nécessaire.

```
1 regex = /b(on|eau|rillant)\./
2 "On est beau."[regex]          # => "beau"
3 "On est bon."[regex]           # => "bon."
4 "On est brillant."[regex]      # => "brillant."
```

S'il n'y a que des lettres comme alternative, nous n'utiliserons pas ceci mais plutôt une classe de caractères.

IV.2.2. Des informations supplémentaires

IV.2.2.1. La classe `MatchData`

Pour le moment, nous n'avons fait que vérifier la correspondance entre une expression régulière et une chaîne de caractères. Pourtant, nous pouvons récupérer plusieurs autres informations sur la correspondance. Nous pouvons notamment «capturer» des expressions, c'est-à-dire récupérer

une partie de la chaîne lue qui correspond à une partie de l'expression régulière lue. Pour cela, nous n'allons plus utiliser les opérateurs, mais la méthode `match`.

```
regex = %r{<h[1-6]>.*</h[1-6]>}
str = "<h1>Un titre</h1>"
regex.match(str)
```

La méthode `match` renvoie une instance de la classe `MatchData` si la correspondance a été trouvée et `nil` sinon. Notons que nous pouvons utiliser la méthode `match` avec les chaînes de caractères, mais aussi avec les expressions régulières (auquel cas le paramètre devra être une chaîne de caractères).

```
"abcd".match(/.{4}/)
/.{4}/.match("abcdefgh")
/.{4}/.match("abc")    # => nil
```

De plus, nous pouvons rechercher une correspondance à partir d'un caractère particulier, en donnant à `match` comme second argument l'indice de ce caractère dans la chaîne.

```
/a.{2}/.match("abcdeafg", 3)
```

Cette fois, on obtient `#<MatchData "afg">`, la correspondance ayant été recherchée à partir de la quatrième lettre (celle d'indice 3).

L'objet obtenu nous donne accès à l'expression régulière et à la chaîne dont il est issue (grâce aux méthodes `regexp` et `string`), et donne également accès à la partie de la chaîne qui correspond à l'expression régulière. Pour l'obtenir, nous allons utiliser la méthode `to_s`.

```
m = /a.{2}/.match("abcdeafg", 3)
puts "#{m.to_s} est présent dans #{m.string}" if m
```

Pour compléter cela, nous pouvons obtenir la partie de la chaîne qui précède la correspondance trouvée (avec la méthode `pre_match`) et celle qui la suit (grâce à la méthode `post_match`).

```
m = /a.{2}/.match("abcdeafg", 3)
puts "#{m.string} se compose de #{m.pre_match}, #{m.to_s} et #{m.p
    ost_match}."
```

IV.2.2.2. Capture d'expressions

La capture d'expression se fait grâce à ce que l'on appelle des «groupes de capture». Un groupe de capture est une expression entourée de parenthèses dans une expression régulière. L'instance de `MatchData` obtenue avec la méthode `match` nous permet ensuite de récupérer la partie de la chaîne correspondant au groupe. Reprenons notre exemple de titre en HTML.

```
1 regex = %r{<h([1-6])>(.*)</h([1-6])>}
2 str = "<h6>Un titre</h6>"
3 m = regex.match(str)
```

Nous pouvons voir que `m` a changé. Nous pouvons accéder aux différents groupes capturés en utilisant les crochets. L'élément d'indice 0 est la chaîne toute entière et les indices des chaînes capturées commencent à partir de l'indice 1. Ainsi, `m[1]` donne `"6"`, `m[2]` donne `"Un titre"` et `m[3]` donne `"6"`.

Même si une instance de `MatchData` ressemble en ce sens a un tableau, ce n'en est pas un et en fait, elle ne sait même pas comment énumérer ses éléments (elle n'a pas de méthode `each`).

On peut néanmoins la transformer en tableau en utilisant la méthode `to_a`. Cependant, la chaîne totale ne nous intéresse parfois pas. En utilisant la méthode `captures`, on a accès aux différents groupes capturés. Elle retourne un tableau contenant les chaînes capturées.

```
1 m.captures  # => ["6", "Un titre", "6"]
```

Par exemple, on peut créer une fonction pour vérifier qu'une chaîne contient un titre correct en HTML. On vérifie que la chaîne vérifie l'expression régulière. Si oui, on vérifie également que les deux chiffres des balises sont les mêmes.

```
1 class String
2   def html_title?
3     regex = %r{<h([1-6])>.*</h([1-6])>}
4     m = match(regex)
5     false unless m
6     l = m.captures
7     l[0] == l[1]
8   end
9 end
```

Pour représenter une alternative, nous utilisons également des parenthèses, et avec ces parenthèses, on récupère également un groupe. Cependant, ce n'est pas toujours notre but. Heureusement, il existe un moyen pour ne pas capturer un groupe. Il suffit d'écrire `?:` après la parenthèse ouvrante.

```
regex = /b(?:on|eau)\./
l = "On est beau.".match(regex).captures
puts l # => []
```

De manière générale, quand nous n'avons pas besoin du résultat d'un groupe, il vaut mieux ne pas capturer l'expression de ce groupe.

IV.2.2.2.1. Nommer les groupes

Dans certaines expressions compliquées où l'on récupère plusieurs groupes, nommer ces groupes peut être une bonne idée. Et c'est possible. Pour cela, on utilise la syntaxe `?<name>` dans le groupe qui aura alors pour nom `name`.

```
regex = %r{<h[?<nb_1>1-6]>.*</h[?<nb_2>1-6]>}
str = "<h1>Un titre</h1>"
str.match(regex)
```

Cette fois, l'objet obtenu contient des données supplémentaires. On voit en utilisant `irb` que `nb_1` et `nb_2` valent 1. La méthode `named_captures` nous permet alors d'obtenir un hachage dont les clés sont les noms des groupes et les valeurs la partie de la chaîne y correspondant. On écrit alors la méthode `titre_html` de la manière suivante.

```
class String
  def html_title?
    regex = %r{<h[?<nb_1>1-6]>.*</h[?<nb_2>1-6]>}
    m = match(regex)
    false unless m
    l = m.named_captures
    l[nb_1] == l[nb_2]
  end
end
```

Notons de plus la fonction `names` qui renvoie la liste des noms des groupes.

IV.2.2.3. Contexte de comparaison

Les expressions régulières ne se limitent pas à ça puisqu'on peut rajouter un contexte à notre recherche. Par exemple, nous pouvons rechercher quelque chose au début ou à la fin d'une chaîne grâce à `\A` et `\Z`. Ces deux symboles sont des **assertions**. Les autres ancrages sont aussi des assertions, mais il s'agit là d'assertions plutôt simples.

Nous avons la possibilité de préciser un contexte beaucoup plus complexe. Prenons par exemple une phrase comme celle là.

> Chez nous, les gens ont généralement deux prénoms, et on les appelle rarement par les deux. On préfère utiliser le premier prénom ou un surnom, mais quand on utilise le nom (par exemple, dans les situations formelles), il faut obligatoirement utiliser les deux prénoms.

Notre but est d'obtenir les occurrences de «noms», mais seulement dans les mots «prénom» (pour cela, nous allons utiliser la méthode `scan` qui peut aussi prendre une expression régulière en paramètre).

Dans notre expression, nous allons utiliser une assertion arrière positive (*positive lookbehind assertion* en anglais). Cette assertion est dite «arrière positive», car on vérifie la présence de quelque chose avant ce que l'on veut récupérer. Elle se construit avec `(?<=pattern)`. Notre expression est alors la suivante.

```
1 regex = /(?<=pré)nom/
2 str.scan(regex)
3 => ["nom", "nom", "nom"]
```

On obtient trois occurrences, les mot «nom» et «surnoms» n'ont pas été pris en compte.

Il y a également des assertions arrière négative (dans ce cas, on vérifie la **non-présence** de quelque chose) et des assertions avants (dans ce cas, on vérifie la présence ou la non-présence après ce qu'on cherche). Les quatre assertions que nous venons de voir sont appelés *lookaround assertions* puisqu'elles permettent de «regarder ce qui se passe autour».

Voici un tableau récapitulatif de ces assertions.

Nom français	Nom anglais	Motif
Assertion avant positive	positive lookahead assertion	`(?=pattern)`
Assertion avant négative	negative lookahead assertion	`(?!pattern)`
Assertion arrière positive	positive lookbehind assertion	`(?<=pattern)`
Assertion arrière négative	negative lookbehind assertion	`(?<!pattern)`

TABLE IV.2.2. – Les *lookaround assertions*.

Nous pouvons remarquer que les symboles ne sont pas choisies au hasard, `=` indique que l'assertion est positive, `!` qu'elle est négative et `<` signifie qu'il s'agit d'une assertion arrière.
Utilisons ces assertions pour récupérer les occurrences de «nom» qui commencent par un «pré» et ne se terminent pas par un «s».

```
regex = /(?<=pré)nom(?!s)/
str.scan(regex)
=> ["nom"]
```

Cette fois, on n'obtient plus qu'une seule occurrence.

IV.2.3. Et encore ?

IV.2.3.1. Plusieurs comportements ?

Les expressions régulières peuvent avoir plusieurs comportements, et c'est à nous de choisir laquelle nous convient. Il y a trois types de comportements.

IV.2.3.1.1. La gourmandise

Par défaut, une expression régulière est **paresseuse**. Cela signifie qu'elle essaye de trouver la plus grande correspondance possible. Un exemple simple de ce comportement est l'utilisation de l'expression régulière `.+\.`. On veut n'importe quoi, puis un point.

```
str = 'Une phrase. Une autre phrase.'
str[/.+\./] => "Une phrase. Une autre phrase."
```

Alors que «Une phrase.» correspond à l'expression, c'est toute l'expression qui est renvoyée. Avec une expression régulière gourmande, on cherche la plus grande correspondance possible. On teste toute la chaîne, si ça ne correspond pas, on enlève un caractère, etc. On continue ainsi jusqu'à ce que l'expression soit vérifiée (ou qu'il ne reste plus de mots).
Ce recul progressif consomme donc des ressources (même s'il est optimisé). Une expression régulière gourmande est donc… Gourmande en ressources.

IV.2.3.1.2. La paresse

Dans le cas où on ne recherche pas la plus grande correspondance, on peut utiliser une expression régulière paresseuse. Comme son nom l'indique, une expression paresseuse en fait le moins possible et retourne la correspondance minimale. Avec une expression paresseuse, la correspondance est d'abord testée au début de la chaîne avec le nombre minimum de caractères.
Pour utiliser un quantificateur fainéant, il suffit de lui ajouter un point d'interrogation. Ainsi, `+?` est la version fainéante de `+`. Regardons notre exemple précédent avec lui.

```
str = 'Une phrase. Une autre phrase.'
str[/.+?\./] => "Une phrase."
```

Cette fois, nous obtenons la correspondance minimale. Les expressions fainéantes consomment en général moins de ressources.

IV.2.3.1.3. La possessivité

Un troisième «mode» existe, il s'agit du mode possessif. Celui-ci tente de trouver la plus grande correspondance possible, comme le mode gourmand, mais si elle ne correspond pas, il échoue directement sans tenter autre chose.
Son avantage est donc cet échec rapide qui permet de consommer moins de ressources que le mode gourmand. Cependant, il peut donc arriver qu'il échoue alors qu'une correspondance existe, comme nous allons le voir.
Pour utiliser un quantificateur possessif, nous allons cette fois lui ajouter un symbole `+`. Regardons cet exemple qui échoue.

```
1 str = 'Une phrase. Une autre phrase'.
2 str[/.++\./] => nil
```

Ici, nous n'obtenons aucun résultat. En effet, le `.++` va essayer de correspondre avec toute la chaîne, puis une correspondance va être testée entre `\.` et ce qui reste (une chaîne vide) et cette dernière correspondance va échouer. Là où avec le quantificateur gourmand nous serions retourné en arrière, avec le quantificateur possessif nous échouons directement.
Les quantificateurs possessifs sont rarement utilisés, mais qui sait. Ils sont parfois utilisés lors de la recherche d'une petite expression dans une plus grande expression.

IV.2.3.2. Variables globales pour les expressions régulières

Les expressions régulières viennent avec un gros lot de variables globales. Pour commencer, il y a toutes les variables globales `$n` avec `n` un nombre. Elles contiennent le n-ième groupe qui a été récupéré la dernière fois qu'une expression régulière a été testée (que ce soit avec `match`, avec `=~` ou même avec `[]`).

```
1 'abc'[/(.)(.)/]
2 puts $1  => a
3 puts $2  => b
4 'abc'[/(.)/]
5 puts $1  => a
6 puts $2  => nil
```

Remarquons dans l'exemple précédent que `$2` n'a pas gardé sa valeur après le second test. Comme nous l'avons dit, `$n` contient le n-ième groupe récupérée lors du **dernier test**.
Ces variables globales permettent de se passer de l'utilisation de `match`, et ce sont pas les seules puisque l'instance de `MatchData` associée à la dernière expression régulière testée est disponible grâce à la variable globale `$~`. De la même manière, `$&` permet d'avoir accès au dernier texte vérifié.

```
1 str = '<h1>Un titre</h1>'
2 str =~ %r{<h([1-6])>(.*)</h([1-6])>}
3 print $&  => "<h1>Un titre</h1>"
4 print $~[1]
```

Nous disposons encore d'autres variables globales comme `$`` qui correspond à la méthode

`pre_match` de l'instance de `MatchData`, `$'` qui correspond lui à la méthode `post_match` et `$+` qui correspond à l'expression associée au dernier groupe capturé.

? Que faut-il utiliser, les variables globales, la méthode `match`, autre chose?

Les variables globales comme `$1` sont tirées du langage Perl et nous allons éviter de les utiliser et préférer utiliser `match` quand c'est possible. Si malgré tout nous avons besoin de récupérer des informations sur la dernière expression vérifiée et que `match` n'a pas été utilisée, nous pouvons utiliser la méthode `last_match` de la classe `Regexp`. Elle nous renvoie l'instance de `MatchData` associée au dernier test. On peut même lui passer en paramètre un entier `i`, pour avoir l'élément d'indice `i` de ce `MatchData`.

```
1 str = '<h1>Un titre</h1>'
2 str =~ %r{<h([1-6])>(.*)</h([1-6])>}
3 print Regexp.last_match(0)
```

Comme nous pouvons le voir, nous avons une multitude de moyens pour nous passer de l'utilisation des variables globales, utilisons-les. Si nous voyons ces variables, c'est parce qu'il faut savoir qu'elles existent et qu'elles peuvent être croisées dans certains codes.

IV.2.3.3. Tous les problèmes ne sont pas des clous

Les expressions régulières font partie de ces choses qui ont l'air un peu magique et semblent résoudre tous les problèmes une fois qu'on sait s'en servir. En effet, elles sont très puissantes et sont en fait quasiment un autre langage qui, une fois maîtrisé, offre de nombreuses possibilités. Néanmoins, il faut garder en tête qu'elles sont consommatrices de ressources. Ainsi, il faut éviter de les utiliser à tort et à travers, car non, tous les problèmes ne sont pas des clous. Par exemple, pour savoir si une chaîne contient une autre chaîne, on utilisera plutôt `chaîne_1[chaîne_2]`. De même, si nous pouvons utiliser `scan` avec une chaîne de caractères plutôt qu'une expression régulière, il ne faut pas hésiter à le faire. Notons que tout comme `scan`, `gsub` et `sub` peuvent s'utiliser avec des expressions régulières et que là encore nous allons préférer les utiliser avec des chaînes de caractères lorsque c'est possible et utiliser les fonctions adaptées.

```
1  str = '10 - 11 - 12 - 13'
2
3  # Pour remplacer un caractère par un autre, on utilise `tr`.
4  str.tr('-', '|')
5  => "10 | 11 | 12 | 13"
6
7  # Pour remplacer un groupe de caractères fixés, on utilise
8  # gsub avec une chaîne de caractères.
9  str.gsub(' -', ',')
10 => "10, 11, 12, 13"
11
12 # Pour remplacer un modèle par un autre, on utilise
13 # gsub avec une expression régulière et un bloc si nécessaire.
14 str.gsub(/\d+/) { |n| (n.to_i + 100).to_s }
```

```
15 => "110, 111, 112, 113"
```

Nous allons finir sur cette citation qui résume ce que nous racontons là.

> Some people, when confronted with a problem, think ”I know, I’ll use regular expressions.” Now they have two problems.
>
> *Jamie Zawinski*

En français: Des gens, quand ils font face à un problème, se disent «je sais, je vais utiliser les expressions régulières». Ils ont alors deux problèmes.

IV.2.4. Exercices

Il est temps de pratiquer un peu et de voir par la même occasion à quel point les expressions régulières peuvent être puissantes.

IV.2.4.1. Exercice 1

Le premier exercice est, comme d’habitude, plutôt simple. Notre objectif va être de valider une plaque d’immatriculation. Nous allons demander à l’utilisateur une chaîne de caractères et lui afficher si elle correspond à une plaque d’immatriculation valide ou pas. Une plaque est valide si elle suit le modèle «AA-111-AA» (donc deux lettres, un tiret, trois chiffres, un tiret, et de nouveau deux lettres).
Correction.

Contenu masqué n°29

IV.2.4.2. Exercice 2

Le but de cet exercice est de gérer un carnet d’adresse. Dans ce carnet, on gardera les noms, numéros de téléphone, et adresse courriel. Nous pouvons faire le programme plus ou moins complet (ajout et retrait d’un contact au carnet, modification, etc.). Dans tous les cas, nous devrons vérifier que son nom et son adresse courriel sont bien valides. C’est une bonne occasion pour travailler plusieurs notions et pas seulement les expressions régulières.
Correction.

Contenu masqué n°30

IV.2.4.3. Exercice 3

Après un exercice complet, mais finalement pas très axé sur les expressions régulières, en voici un qui donne à réfléchir. Ici, nous allons vérifier qu’une expression mathématique est correcte syntaxiquement et sémantiquement. Il nous faudra donc nous assurer de ces trois choses.

1. Il n’y a que des caractères autorisés.
2. L’expression est correctement parenthésée.

3. Tous les termes sont bien placés (par exemple, il n'y a pas un `+` à côté d'un `*`.

Vu que nous aurons peut-être besoin d'expressions régulières complexes, précisons ici que l'interpolation fonctionne aussi dans les expressions régulières.

```
two_char = "[A-Z]{2}"
even_digit = "[02468]"
regex = %r{#{two_char} - #{even_digit} #{even_digit}}
"CD - 6 2".match(regex)
```

Correction.

Contenu masqué n°31

Conclusion

Et c'est terminé pour ce chapitre. Bien sûr, les motifs des expressions régulières ne sont pas à connaître par cœur.

— Les expressions régulières permettent de vérifier qu'une chaîne correspond à un modèle.
— Elles permettent également de récupérer certains motifs dans une chaîne (avec `scan` par exemple).
— Nous ne devons pas abuser des variables globales associées aux expressions régulières.
— Les expressions régulières ne permettent pas de résoudre tous les problèmes.

Pour finir, nous pouvons aller visiter les pages de documentation de `MatchData` et de `Regexp`. En particulier, les parties «*options*», «*free spacing mode and comments*» et «*encoding*» qui nous proposent quelques options pour modifier un peu nos expressions régulières (par exemple les écrire sur plusieurs lignes et y mettre des commentaires).

Contenu masqué

Contenu masqué n°29

```
regex = /\A[A-Z]{2}-\d{3}-[A-Z]{2}\z/

print "Entrez la plaque : "
str = gets.chomp
print str.match?(regex) ? "Plaque valide." : "Plaque invalide."
```

Ici, un point qu'il ne fallait pas oublier est la présence de `\A` et de `\z`. Sans eux, la chaîne «je suis AA-111-AA.» serait considérée comme valide. Retourner au texte.

Contenu masqué n°30

```ruby
class String
  def mail?
    match?(/\A[\w.-]+@[\w.-]{2,}\.[a-z]{2,4}\z/)
  end

  def phone?
    match?(/\A0\d{9}\z/)
  end

  def alpha?
    match?(/\A[[:alpha:]]+\z/)
  end
end

class Contact
  attr_reader :name
  attr_accessor :mail, :phone

  def initialize(name, mail, phone)
    @name = name
    @mail = mail
    @phone = phone
  end

  def to_s
    "#{@name} - #{@phone} - #{@mail}"
  end
end

def get_contact
  print "Entrez le nom du contact :"
  name = gets.chomp until name.alpha?
  print "Entrez son numéro de téléphone : "
  phone = gets.chomp until phone.phone?
  print "Entrez son adresse courriel : "
  mail = gets.chomp until mail.mail?
end
```

Il n'y a pas trop de difficulté ici, nous avons des méthodes pour vérifier qu'une chaîne est un nom, un numéro de téléphone ou une adresse courriel, et nous créons une classe pour les contacts. Retourner au texte.

Contenu masqué n°31

```ruby
nombre = '\d+(\.\d+)?'
opérateur = '[+-/*]'
variable = '[A-Z]'

atomique = "(#{nombre}|#{variable})"
atomique_signé = "[+-]?(#{nombre}|#{variable})"

expression = "#{atomique_signé}(#{opérateur}(#{atomique}|\\(#{atomique_signé}\\)))*"
expression_parenthésée = "\\(#{expression}\\)"

regex =  %r{#{expression_parenthésée}}

str = gets
str.gsub!(/\s+/, '')
str.gsub!(regex, "X") until str.scan(regex).empty?

puts str
puts str[/\A#{expression}\z/] ? "Valide." : "Invalide."
```

L'idée ici est qu'une expression correcte est composée d'expressions elles-mêmes correctes. Ainsi, nous remplaçons toutes les expressions de la forme `(n1 op1 n2 op2 n3 op3 n4 ...)` par la variable `X`. Et on fait ceci tant que des expressions de ce type existent. Ensuite, si l'expression est correcte, il nous restera une seule expression valide. Testons le principe sur un exemple.

```
((2 + 3) * (3 - 5 + 8)) + (1 + 2 * (3 + 8))
(X * X) + (1 + 2 * X)
X + X
```

À noter qu'ici, nous ne considérons pas les expressions de la forme `(1 + 2)(3 + 4)` (où le signe `*` n'est pas mis). Retourner au texte.

Cinquième partie

Annexes

V.1. Écrire le code dans des fichiers

Introduction

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.

V.1.1. Les éditeurs de texte

V.1.1.1. 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.

V.1.1.2. 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).

i

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.

V.1.1.3. 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»).

V.1.2. 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.

V.1.2.1. 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».

V.1.2.2. Exécuter le code

Pour lancer le code écrit, il suffit de l'enregistrer avec l'extension .rb puis de double-cliquer dessus.

V.1.2.2.1. 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 V.1.1. – 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.

```
1 NPP_CONSOLE OFF                                  // Ne pas afficher la
      console NppExec.
2 NPP_SAVE                                         // Sauvegarder le
      fichier courant.
3 CD $(CURRENT_DIRECTORY)                          // Se placer dans le
      dossier du fichier courant.
4 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 V.1.2. – Création du script.

Cliquez sur «Compléments» → «NppExec» → «Advanced Options». Choisissez le script que vous avez créé dans «Associated script» et cliquez sur «Add/Modify».

Figure V.1.3. – 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 V.1.4. – Ajout du raccourci.

Ça y est, vous avez votre super raccourci (notez que vous pouvez utiliser le même raccourci pour Python par exemple).

V.1.3. 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+.

V.1.3.1. 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…

```
1 sudo apt-get install gedit
```

… ou…

```
1 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.

V.1.3.2. 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.

V.1.4. Les problèmes possibles

V.1.4.1. 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.

```
1 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.

```
1 cd programmation
2 cd ruby
3 test.rb
```

La commande `cd` est une commande bash. Elle n'est donc pas à utiliser dans IRB, mais directement dans la console.

V.1.4.2. 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…

```
1 a = 'Bonjour'
2 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.

V.1.4.3. 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`.

Conclusion

Vous pouvez maintenant écrire vos codes dans des fichiers et les exécuter.

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

HT Hors taxes. 48

IRB Interactive Ruby. 1, 17, 18, 20, 21, 31, 39, 80, 129, 218, 251

PGCD Plus Grand Commun Diviseur. 87, 89

TTC Toutes taxes comprises. 48

TVA Taxe sur la valeur ajoutée. 48, 51