Apprendre à programmer avec Ruby

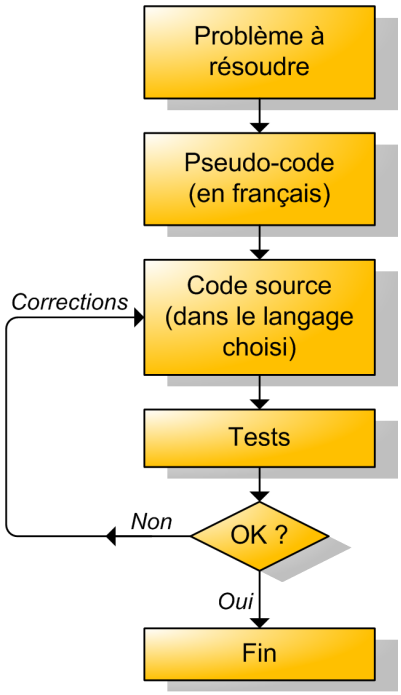
Apprendre à programmer peut se faire avec n'importe quel langage généraliste : les mêmes règles de base s'utilisent quel que soit le langage. Toutefois, certains sont plus proches que d'autres de la logique humaine et donc mieux adaptés à l'apprentissage de la programmation. C'est le cas de Ruby.

La méthodologie de la programmation est simple :

1° On exprime le problème à résoudre en un **algorithme**, une description de ce qu'il faut faire en français, de manière indépendante du langage utilisé. On parle de **pseudo-code**.

2° On écrit le **programme** dans le langage choisi. On fait cela au moyen d'un **éditeur de texte**, un logiciel de traitement de texte adapté à l'écriture de programmes informatiques. Le résultat est le **code source**. C'est du texte ordinaire, mais avec une syntaxe bien précise et spécifique au langage.

3° On teste le programme, on corrige les erreurs s'il y en a et on reteste. On continue ainsi jusqu'à ce qu'il n'y ait plus d'erreur.



Par la suite, quand on veut exécuter le programme, on va dans l'interface de commande et on tape (dans le cas de Ruby) :

```
ruby nom-du-fichier.rb
```

L'extension *rb* signifie qu'il s'agit de code source en Ruby.

La commande *ruby* indique à l'ordinateur qu'il doit lancer l'**interpréteur** Ruby. Un interpréteur est un **traducteur**, un programme chargé de transformer le texte du code source en une suite d'instructions directement compréhensibles par l'ordinateur.

Le fait que Ruby est un langage interprété veut dire qu'on ne peut pas lancer un programme Ruby si le langage n'a pas été installé préalablement sur la machine.

Installation de Ruby

Mac OS : Ruby est préinstallé et prêt à l'emploi.

Windows : par défaut, Ruby n'est pas installé. Aller sur le site *http://www.ruby-lang.org/fr/downloads*, cliquer sur *Téléchargez Ruby* , puis sur *Ruby xxx One-Click Installer* et suivre les instructions.

Linux : Ruby est souvent préinstallé, mais pas toujours. Il y a plusieurs moyens de le savoir, mais le mieux est d'utiliser la commande *which*, qui offre l'avantage d'être présente sur tous les systèmes d'exploitation de la famille Unix, et qui sert à indiquer à quel endroit se trouve un programme. Taper :

```
which ruby
```

Si le système répond par exemple */usr/bin/ruby*, c'est que Ruby est installé à l'endroit indiqué. S'il ne répond rien ou s'il affiche un message négatif (du genre *no ruby in...*), il faut installer le langage. Pour cela, on passe par le gestionnaire de paquets du système. Si c'est *apt-get*, taper :

```
sudo apt-get install ruby irb rdoc
```

Une fois l'installation terminée, on peut vérifier que Ruby est bien installé et savoir par la même occasion de quelle version du langage il s'agit en tapant :

```
ruby -v
```

L'interface de commande

Les manipulations décrites dans ce texte se font dans l'interface de commande du système d'exploitation.

Sous Windows, on s'y rend par *Démarrer* → *Tous les programmes* → *Accessoires* → *Invite de commande* ou par *Démarrer* → *Exécuter* en tapant *cmd.exe*. Attention, par défaut, l'option *Exécuter* n'existe plus depuis Vista. Pour l'activer, faire un clic droit sur l'icône du menu *Démarrer*, *Propriétés* → *Menu Démarrer* → *Personnaliser* et cocher la ligne *Commande Exécuter*.

Sous Mac OS, c'est *Finder* → *Applications* → *Utilitaires* → *Terminal.app*.

Sous Linux, on peut soit taper *Alt-Ctrl-F1* (pour revenir à l'interface graphique, c'est *Alt-Ctrl-F7*), soit passer par les menus : *Applications* → *Accessoires* → *Terminal*, soit cliquer sur l'icône qui représente un écran noir sur la barre des tâches, si cette icône existe, soit encore faire un clic droit sur le fond de l'écran et sélectionner *Ouvrir un terminal* dans le menu qui apparaît.

Premiers pas

La tradition veut que le premier exemple qu'on donne d'un programme soit celui qui affiche les mots *Hello World* à l'écran.

Pour cela, il faut ouvrir un éditeur de texte et taper le code source.

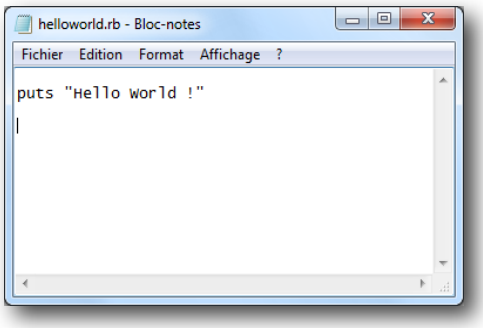
Ici, c'est simplement *puts "Hello World !"* (le mot *puts* est l'abréviation de *put string*, « afficher chaîne de caractères ») :

```
puts "Hello World !"
```

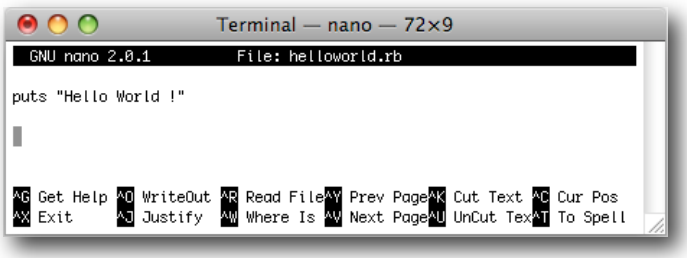
On termine en l'enregistrant sous un nom quelconque, par exemple *test1.rb* ou *helloworld.rb*. L'extension est *rb*.

Concrètement, voici ce que cela donne avec le Bloc-Note de Windows ou avec nano de Unix :

Avec le Bloc-Note de Windows



Avec nano



On exécute le programme en allant dans l'interface de commande et en tapant :

`ruby helloworld.rb`

Le système doit répondre en affichant *Hello World !*. Si ce n'est pas le cas, c'est qu'on a fait une erreur. Un problème commun est d'oublier un espace ou d'en mettre un là où il ne faut pas.

Le C++, le C# et Java

À ce stade, il peut être intéressant de voir le même programme dans les trois langages les plus utilisés actuellement : C++, C# et Java.[1] Ils descendent tous trois du langage C.

En C++ :

```
#include <iostream>
int main()
{
   std::cout << "Hello World !\n";
   return 0;
}
```

En Java :

```
class HelloWorld
{
   public static void main(String[] args)
   {
      System.out.println("Hello World !");
   }
}
```

[1] En réalité, comparer le nombre d'utilisateurs du C#, du C++ et de Java n'a pas beaucoup de sens. C'est un peu comme si on dit que l'anglais et le chinois sont les deux premières langues mondiales. En réalité, le chinois est bien parlé par un grand nombre de personnes, mais elles sont presque toutes en Chine, alors que l'anglais, lui, est parlé dans le monde entier. De la même façon, le C# ne s'emploie pratiquement que dans le monde Windows alors que le C++ et Java sont universels.

Et en C# :

```
using System;
public class HelloWorld
{
   public static void Main()
   {
      System.Console.WriteLine("Hello World !");
   }
}
```

On voit que ces langages sont plus lourds que Ruby, mais cela ne signifie pas qu'ils ne sont pas aussi bons. C'est une philosophie différente.

Remarquer l'**indentation**, le fait que les lignes sont décalées pour rendre le code plus lisible. Par exemple, dans le code C# ci-dessus, on voit que l'instruction *public class HelloWorld* concerne le code qui se situe entre les accolades de la troisième et de la dernière ligne et que l'instruction *public static void Main()* se trouve à l'intérieur.

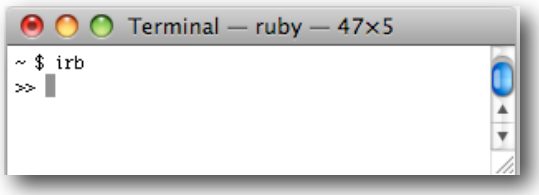
En passant, il faut noter que d'autres langages que Ruby sont légers. C'est le cas de Python. Voici le programme Hello World dans ce langage :

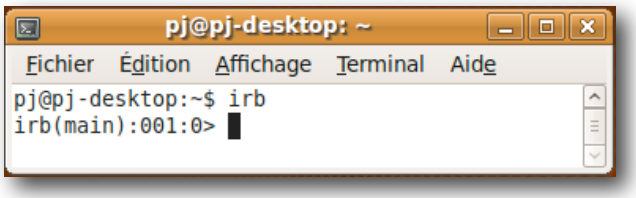
```
print "Hello World !"
```

On exécute ce programme au moyen de la commande *python helloworld.py*, pour autant que le langage soit présent sur le système ; c'est le cas avec Mac OS et souvent Linux, mais pas avec Windows. On peut le télécharger depuis le site officiel (*http://www.python.org/download*).

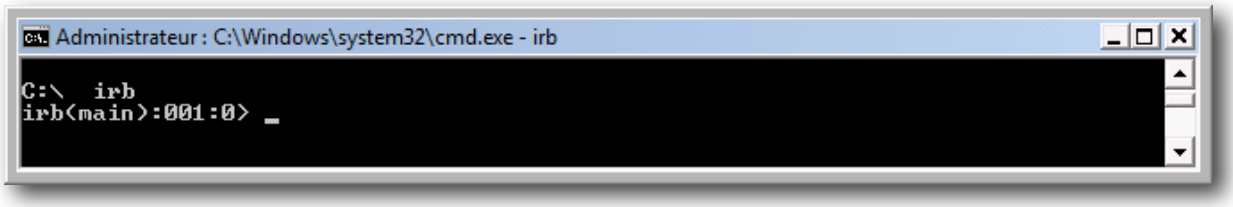
Ruby interactif

Ruby inclut une version interactive appelée **irb** (*interactive Ruby*). On la lance en tapant *irb* dans l'interface de commande. Voici ce que cela donne sous Mac OS (à gauche) et Ubuntu Linux (à droite) :





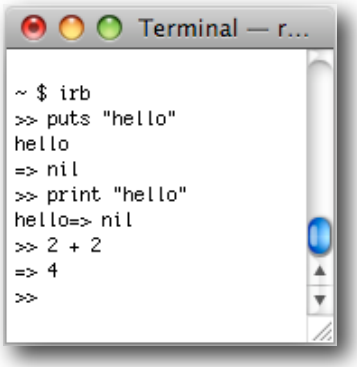
Et sous Windows :



Sous Linux et Windows, on peut taper *irb --simple-prompt* au lieu d'*irb* tout court si l'on préfère l'invite minimale « >> » comme sous Mac OS.

Pour que le programme Hello World s'exécute dans irb, on tape :

```
puts "Hello World !"
```



L'interpréteur répond en affichant le message demandé suivi de « => nil », ce qui veut dire qu'il n'y a pas de valeur à renvoyer. Par contre, si on tape *2 + 2*, le système renvoie la valeur *4* (voir l'image ci-contre).

Irb est très pratique comme bac à sable. Ne pas hésiter à faire beaucoup d'essais dans cet environnement, c'est de cette façon qu'on apprend à programmer.

On en sort en tapant *exit*.

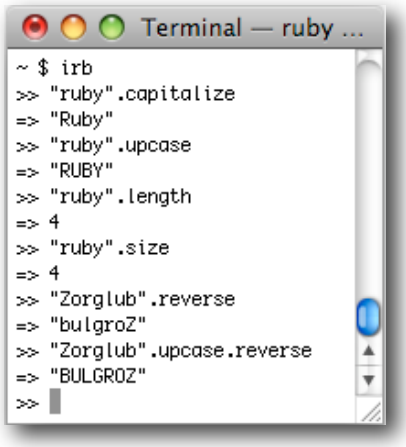
Un langage à objets

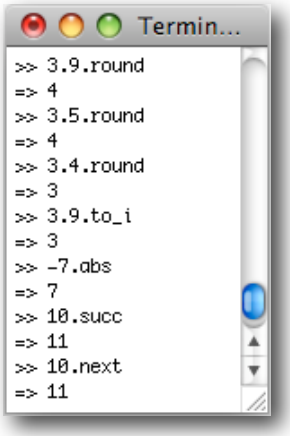
Ruby est un **langage à objets**. Qu'est-ce que cela veut dire ? En pratique, c'est tout simple : en Ruby, tout ce qu'on peut manipuler est un **objet**, et on peut exercer sur tout objet des actions prédéfinies appelées **méthodes**. En Ruby, quand on calcule *6 + 3 = 9*, on applique l'action *additionner* aux objets *6* et *3*.

La syntaxe de base est très simple elle aussi : le nom de l'objet est suivi de celui de la méthode avec un point de séparation entre les deux.

```
NomObjet.NomMéthode
```

Puisque, dans Ruby, tout est objet, les chaînes de caractères sont des objets. Que peut-on faire avec elles ? Par exemple, on peut mettre la première lettre en majuscule (*capitalize*), mettre toute la chaîne en majuscules (*upcase*) ou mesurer sa longueur (*size* ou *length*, ce sont des synonymes). On peut inverser les caractères de la chaîne et transformer l'objet *Zorglub* en *bulgroZ* en hommage à Franquin.[2]





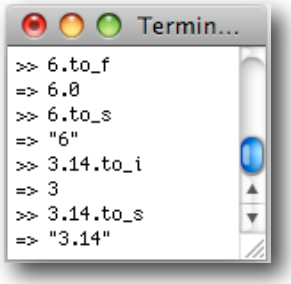
On peut aussi appliquer une succession de plusieurs méthodes à un objet. Ainsi, la ligne *"Zorglub".upcase.reverse* applique l'action *inverser les lettres* sur l'objet intermédiaire *ZORGLUB* obtenu par l'action *mettre en capitales* effectuée sur l'objet original *Zorglub*.

Bien entendu, les nombres sont également des objets, et on peut aussi appliquer des actions sur eux. Par exemple, on peut arrondir un nombre (*round*), demander l'entier suivant (*succ* ou *next*) ou prendre sa valeur absolue (*abs*).

---

[2] Franquin, Jidéhem et Greg, *Z comme Zorglub*, Dupuis, Bruxelles, 1961.

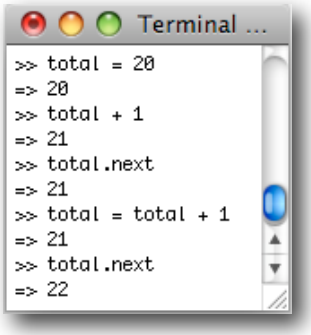
En Ruby, il existe deux types de nombres : les entiers (*integers*) et les nombres en virgule flottante (*floats*), les nombres qui ont des chiffres après la virgule comme *2,7* ou *3,1416*. On peut changer un entier en nombre en virgule flottante (*to_f*), ou faire l'inverse (*to_i*), ou même le changer en chaîne de caractères (*to_s*). Par exemple, *6* est un entier, *6.0* un nombre en virgule flottante et *"6"* une chaîne de caractères.



Attention, si l'arrondi de 3,9 donne bien 4, la transformation de 3,9 en nombre entier donne 3. Cela explique aussi que 3 / 2 donne 1. Si on veut la bonne réponse, il faut travailler avec des nombres en virgule flottante et non des entiers. Pour cela, il faut taper 3.0 / 2.0. Le résultat est 1.5.

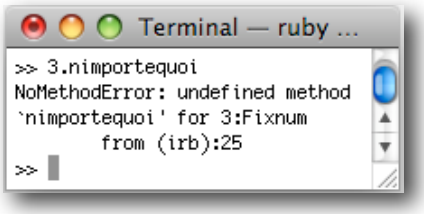
N.B.: comme toujours en informatique, la virgule française est remplacée par le point anglo-saxon. On n'écrit pas 1,5 mais 1.5.

On peut aussi agir sur des **variables**, qui sont des objets dont la valeur peut changer. Par exemple, on peut créer une variable appelée *total* et lui donner la valeur initiale *20* puis agir sur cette valeur.



Si on y ajoute *1* et qu'on demande son successeur, le résultat est le même : *21*. Cette dernière opération ne donne pas *22* parce que l'action *ajouter 1 à total* ne fait pas changer la valeur de la variable *total*. Par contre, c'est le cas avec la formule *total = total + 1* qui signifie *modifier la valeur actuelle de total en y ajoutant 1*.

On l'a dit, une méthode est une **action prédéfinie**. Une méthode qui n'est pas préalablement définie est incomprise et provoque le message d'erreur *undefined method*.



Toutefois, prédéfini ne veut pas dire bloqué. On va voir plus loin qu'on peut créer nos propres méthodes au moyen de la construction *def... end*.

Les objets font partie de **classes**. Une classe, c'est ce qui définit un ensemble d'objets, ce qui fait qu'ils se ressemblent entre eux et qu'ils sont différents des objets des autres classes. Par exemple, les chaînes de caractères forment une classe appelée *String*. Une classe décrit les **propriétés** (ou **attributs**) de ses objets ainsi que leur comportement tel qu'il est défini par les méthodes.

Il existe aussi des **sous-classes**. Ainsi, en Ruby, la classe *Numeric* définit ce qui caractérise un nombre en général, et des sous-classes développent ce qui caractérise chaque type de nombre (nombre entier, nombre en virgule flottante, etc.). Une classe mère est appelée une **métaclasse**.

Les sous-classes héritent des méthodes de leur classe mère — c'est la notion d'**héritage** — et y ajoutent les leurs, celles qui leurs sont spécifiques.

En Ruby, il y a une classe mère de toutes les classes : **Object**. C'est elle qui définit les méthodes essentielles, celles qui

> Un **objet** est **tout ce qui est manipulable** dans le langage.
>
> Une **méthode** est une **action possible** sur un objet.
>
> Une **variable** est un **objet** comme un autre.
>
> Une **classe** est un **ensemble d'objets** de la même famille.

concernent tous les types d'objets. Ainsi, le fait que les méthodes « == » (égalité) et « != » (inégalité) soient définies dans cette classe a pour effet qu'un objet de n'importe quel type peut être comparé à un autre objet de n'importe quel type.

Le **polymorphisme** est une autre notion importante. Il définit l'idée qu'une méthode donnée peut être appliquée à des classes d'objets différentes. Autrement dit, la méthode ne se comporte pas de la même façon selon la classe — elle s'adapte à sa clientèle. Par exemple, avec la méthode *Additionner*, 2 + 2 donne 4, mais "te" + "st" donne "test".

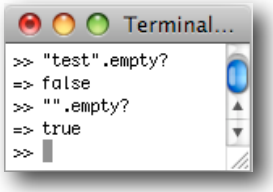
Méthodes prédéfinies

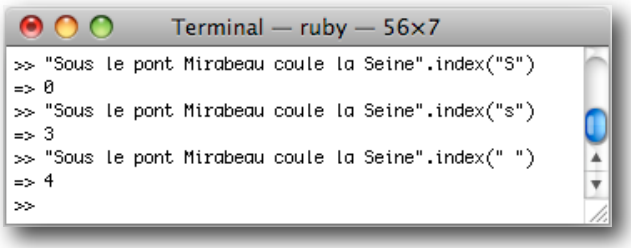
Pour chaque classe, il existe des dizaines de méthodes prédéfinies. On peut afficher la liste des méthodes applicables à une classe en tapant ce qui suit :

***NomClasse*.instance_methods**

Par exemple, si on tape *String.instance_methods* dans irb (le nom de la classe doit commencer par une majuscule), on obtient ceci :

Certaines méthodes se terminent par un point d'interrogation. Ce sont des méthodes **booléennes**, des tests qui ne peuvent retourner que la valeur *true* (vrai) ou *false* (faux). Par exemple, si on tape *"test".empty?* (y compris le point d'interrogation final), ce qui veut dire « la chaîne de caractères "test" est-elle de longueur nulle ? », la réponse sera *false*. Inversement, si on tape *"".empty?*, la réponse sera *true* puisqu'il n'y a aucune caractère entre les deux guillemets.



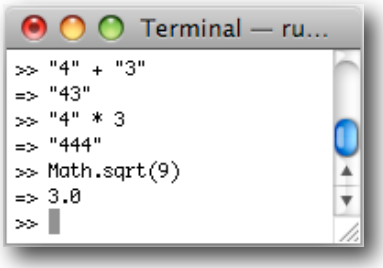


Pour Ruby, une chaîne de caractères est une liste d'éléments. Cela permet, par exemple, de rechercher la position d'un caractère. La méthode est *index("c")*. Le premier caractère occupe la position 0, le deuxième la position 1, etc.

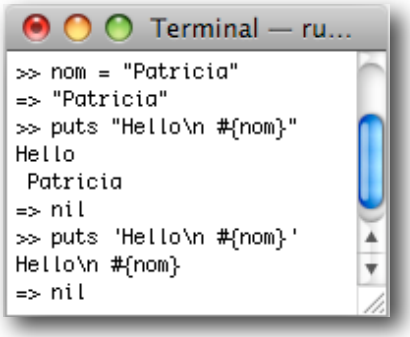
Le langage différencie minuscules et majuscules. Ainsi, dans l'exemple ci-contre, il voit la lettre majuscule *S* à la position 0 et la lettre minuscule *s* à la position 3. Il est important de noter que l'espace est un caractère lui aussi.

Les opérations numériques sont les opérations habituelles : l'addition (+), la soustraction (-), la multiplication (*), la division (/), la puissance (**), etc. Par exemple, l'expression *x**3* signifie x^3.

Si on place un nombre entre guillemets ou entre apostrophes, c'est une chaîne de caractères. Par exemple, *"4" + "3"* donne *"43"*. Cette opération est une **concaténation**. On peut aussi multiplier des caractères. Par exemple, *"4" * 3* donne *"444"* alors que *4 * 3* donne *12*.



La méthode *Math.sqrt(x)* donne la racine carrée (*square root*) de *x*. Le résultat de *sqrt* est automatiquement un nombre réel.



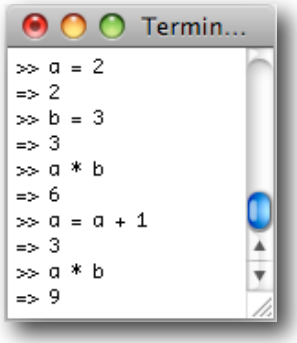
Pour enclore une chaîne de caractères, les guillemets ne sont pas la seule solution. On peut aussi utiliser les apostrophes, mais, dans ce cas, Ruby restitue la chaîne de caractères telle qu'il la reçoit. Avec les guillemets, il recherche d'abord d'éventuelles variables, qui doivent être précédées d'un signe dièse et entre accolades, ou des caractères de commande.

Le caractère de commande le plus répandu est « \n », qui commande un passage à la ligne (comme dans beaucoup d'autres langages). Il y en a une dizaine d'autres, notamment « \t » pour la tabulation et « \s » pour un espace, mais ils sont moins utiles.

Variables et constantes

Dans tous les langages informatiques, si on veut conserver des valeurs, on les place dans des variables.

Une variable normale est une **variable locale**, ce qui veut dire qu'elle n'existe que dans le bloc de code où elle est créée. En Ruby, une variable locale doit commencer une lettre minuscule ou un caractère de soulignement (c'est ainsi qu'il la reconnaît comme locale). Elle peut se composer de caractères non accentués, de chiffres et du caractère de soulignement. Par exemple, *resultat*, *somme1*, *_plus* , *totalBrut* et *total_net* sont des variables, mais pas *1er* ou *Nom* parce qu'une variable ne doit pas commencer par un chiffre ou une majuscule.



Cette règle est également valable pour les noms de méthodes et les noms de paramètres de méthodes.

On effectue sur les variables les mêmes opérations qu'avec des nombres.

Une **variable globale** existe dans l'ensemble du programme. Son nom commence par le signe $.

Une variable dont la valeur ne doit pas changer est une **constante**. Elle commence par une lettre majuscule. Un exemple est le nombre pi, qu'on peut écrire *PI* ou *Pi*.

Si on change la valeur d'une constante, Ruby obéit mais il affiche un message d'avertissement.

Tableaux et hashes

En Ruby, un **tableau** est un ensemble d'éléments indexés. Le terme « indexé » signifie simplement que chaque élément est numéroté et qu'on peut rechercher un élément en donnant son numéro, qu'on appelle sa **clé** (*key*).

Quand on crée un tableau, on sépare les éléments par des crochets :

```
nomTableau = [ "élément1", "élément2", etc. ]
```

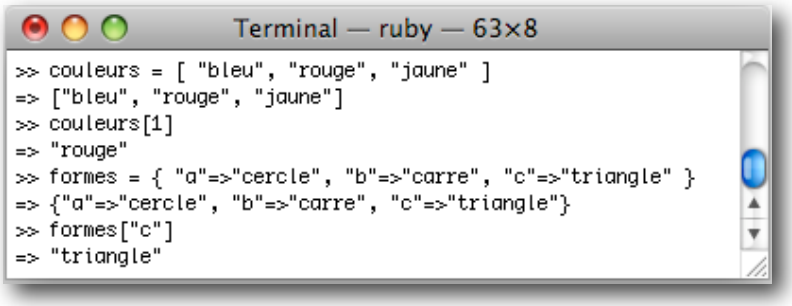
Ils se numérotent automatiquement en partant de zéro. Le premier élément reçoit le numéro 0, le deuxième le 1, etc.

Il existe un autre type de collection, le **tableau associatif** ou ***hash***, qui se différencie du tableau normal par le fait que la clé n'a pas besoin d'être un nombre entier. N'importe quoi convient.

Quand on crée un hash, on sépare les éléments par des accolades. Les clés doivent être indiquées :

```
nomHash = { "clé1"=>"élément1", "clé2"=>"élément2", etc. }
```

En revanche, à l'utilisation, on n'emploie les accolades, mais les crochets comme avec un tableau.



Créer une méthode

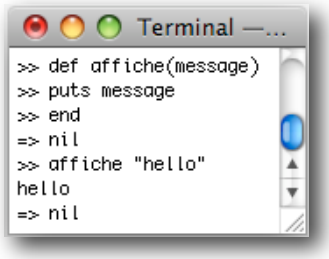
Pour créer une méthode, on utilise la construction **def... end**.

On peut spécifier des **arguments**, ou **paramètres**, c'est-à-dire des éléments sur lesquels l'action doit s'effectuer.

S'il y a des arguments, on les met entre parenthèses après le nom de la méthode, sous la forme *def nom-méthode(paramètre1, paramètre2, etc.)*. Voici un exemple :

```
def affiche(message)
  puts message
end
```

Ce code a pour effet d'afficher un message si l'utilisateur invoque la méthode *affiche*, qui n'est en fait rien d'autre qu'un synonyme de la méthode *puts*.

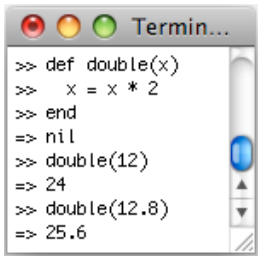


Si on tape l'instruction *affiche "hello"* après avoir créé la méthode, Ruby répond : *hello*.

Voici un autre exemple :

```
def double(x)
  x = x * 2
end
```

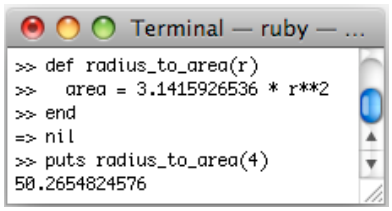
Si on tape *puts* ou *print double(12)*, Ruby répond *24*. À noter que Ruby s'adapte au type de données spécifié par l'utilisateur : si on tape *print double(12.8)*, irb répond *25.6*. Si le paramètre est un nombre en virgule flottante, Ruby donne la réponse en virgule flottante.



Si on travaille avec des cercles dans un programme, il peut être utile de créer une fonction *radius_to_area* qui calcule avec précision l'aire d'un cercle à partir de son rayon :

```
def radius_to_area(r)
  area = 3.1415926536 * r**2
end
```

Si on tape *puts radius_to_area(3)*, Ruby répond *28.27433388231* — si un cercle a un rayon de 3 cm, son aire est d'un peu plus de 28 cm^2.; si c'est 4 cm, elle est de 50 cm^2.



De la même manière, un économiste ou un comptable peut se constituer une bibliothèque de formules financières.

Étrangement, Ruby ne sait arrondir un nombre qu'à l'entier, ce qu'il fait avec la méthode *round* (arrondir). Par exemple, *2.518.round* donne *3*. Elle n'accepte pas d'argument, ce qui veut dire qu'on ne peut pas, par exemple, taper *2.518.round(1)* pour obtenir *2.5* (l'argument *1* étant pour « une décimale »). Pour résoudre ce problème, on peut créer une méthode appelée par exemple *round_to* à la classe des nombres en virgule flottante :

```
class Float
  def round_to(x)
    (self * 10**x).round.to_f / 10**x
  end
end
```

Ainsi, une fois les méthodes *radius_to_area* et *round_to* créées, si on tape ce qui suit :

```
puts radius_to_area(3).round_to(2)
```

Ruby répond *28.27* au lieu de *28.27433388231*.

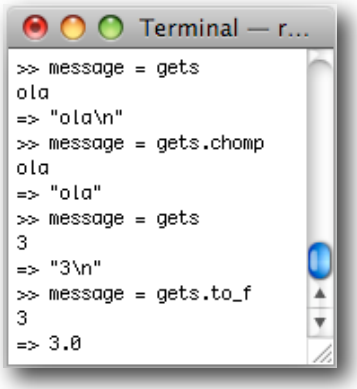
La méthode gets

Une méthode très utile est **gets** (*get string*). Elle permet au programme de lire ce que l'utilisateur tape au clavier.

La syntaxe classique est la suivante :

```
reponse = gets.chomp
```

La méthode *gets* place dans la variable *reponse* les caractères entrés par l'utilisateur, y compris le *Enter* avec lequel il a terminé la frappe. La méthode *chomp* a pour effet d'enlever les caractères finals *"\n"* qui représentent ce Enter. Par exemple, si l'utilisateur a tapé "o" pour "oui", *gets* enregistre "o\n". Il faut utiliser *gets.chomp* pour que la méthode se débarrasse du Enter et garde uniquement la lettre "o".



Si le programme attend une réponse sous forme de caractères, on ajoute généralement encore la méthode *downcase* ou *upcase* :

```
reponse = gets.chomp.downcase
```

Sans cela, quand l'utilisateur tape "o" ou "n", il faudrait traiter deux fois sa réponse : « si la réponse est "o" minuscule ou "O" majuscule, alors... ». Avec le downcase, on peut se contenter de « si la réponse est "o" minuscule, alors... ».

Si on veut que l'entrée de l'utilisateur soit un nombre au lieu d'une chaîne de caractères, il faut utiliser *gets.to_f* ou *gets.to_i* selon le type de nombre désiré :

```
reponse_numerique = gets.to_i
```

Dans ce cas, le chomp est inutile parce que les méthodes *to_i* et *to_f* se débarrassent de toute façon des signes finals *"\n"* puisque ce ne sont pas des chiffres.

Les conditions

Dans le déroulement d'un programme, on a souvent besoin de **branchements**. C'est le cas, par exemple, si on lance ou non un traitement selon la demande de l'utilisateur : « si la réponse est oui, faire ceci... sinon faire cela... ». Tous les langages informatiques permettent de gérer les branchements.

En Ruby, il existe trois constructions de branchement :

- *case... when... when... else... end* (« dans le cas 1... dans le cas 2... etc. »)
- *if... elsif... elsif... else... end* (« si... sinon... »)
- *unless... elsif... elsif... else... end* (« sauf si... sinon... » ; c'est le if inversé).

Voici un exemple avec la construction **if... end** :

```
reponse = gets.chomp.downcase

if reponse == "o"
   puts "la reponse est oui"
elsif reponse == "n"
   puts "la reponse est non"
else
   puts "reponse non comprise"
end
```

Remarquer que l'égalité s'écrit avec un double signe égal (« == »). Ce signe s'emploie pour dire qu'un objet est identique à un autre objet alors que le signe égal simple (« = ») sert à donner une valeur à un objet.

Voici un exemple avec la construction **unless... end** :

```
puts "voulez-vous continuer ? (o/n) "

reponse = gets.chomp.downcase

unless reponse == "o"
   puts "alors vous voulez arreter"
end
```

L'expression *unless reponse == "o"* signifie « sauf si la réponse est oui », ce qui veut dire, soit dit en passant, que toutes les réponses sauf la lettre "o" minuscule ou majuscule sont considérées comme "non".

Avec les deux constructions, le *elsif* et le *else* sont optionnels. Il y a donc quatre formes possibles :

- *if... end* (ou *unless... end*)
- *if... else... end*
- *if... elsif... end*
- *if... elsif... else... end*.

La construction **case... end** se présente ainsi :

```
reponse = gets.chomp.downcase

message = case reponse
   when "o" then "la reponse est oui"
   when "n" then "la reponse est non"
   else "reponse non comprise"
end

puts message
```

On voit que le case est une construction plus compacte et donc plus lisible que le if. Elle est préférable si on a plus de deux cas.

Dans les trois constructions, le *else* indique le résultat par défaut.

Les itérations

Les itérations sont les **boucles**, c'est-à-dire les suites d'instructions qui sont exécutées à plusieurs reprises. Cette situation est très commune en programmation. Elle s'emploie notamment quand il faut faire un total à partir de plusieurs montants. Par exemple, le montant d'une facture se calcule ainsi, sachant qu'une facture peut se composer de plusieurs articles :

1° aller à la première ligne de facture (au premier article) ;

2° mettre le montant de cette ligne dans la variable *total* ;

3° tant qu'il y a des lignes, aller à la ligne suivante et ajouter le montant à la variable *total* ;

4° quand il n'y a plus de ligne, la variable *total* contient le total réel.

Ruby comprend quatre formes principales de boucles :

- *while... end* (« faire la boucle tant que... »)
- *until... end* (« faire la boucle jusqu'à ce que... »)
- *for... in... end* (« faire la boucle le nombre de fois spécifié... »)
- *loop do... end* (« faire la boucle »).

Par exemple, l'itération **while... end** ci-dessous affiche les chiffres 1, 2 et 3 l'un après l'autre :

```
nombre = 1

while nombre < 4
  puts nombre
  nombre += 1
end
```

Au départ, *nombre* vaut *1*. À l'entrée dans la boucle, Ruby vérifie qu'il est plus petit que *4*, ce qui est le cas, ce qui permet à l'itération de commencer. L'instruction *puts nombre* affiche la valeur de la variable, soit *1*, puis *nombre += 1* incrémente cette valeur de 1, ce qui la porte à *2*, et la boucle recommence jusqu'à ce que la valeur de la variable arrive à 4, ce qui arrête la boucle. Le traitement se poursuit alors avec les instructions qui se trouvent après la boucle.

Au lieu de *nombre += 1*, on peut écrire *nombre = nombre + 1*.

L'**incrémentation** est l'opération qui consiste à augmenter la valeur d'une variable à chaque tour de boucle (en général, l'incrément est 1). La variable utilisée pour l'incrémentation s'appelle le **compteur** et son nom est traditionnellement *i*. Au lieu de *i = i + 1*, on peut aussi écrire *i += 1*. Cette notation est populaire. Elle vient du langage C.

On peut également décrémenter le compteur. Sa valeur baisse alors à chaque tour.

Avec la boucle **until... end**, voici comment on affiche les chiffres 1, 2 et 3 :

```
i = 1

until i > 3
  puts i
  i += 1
end
```

La boucle **for... end** est une troisième possibilité :

```
for i in 1..3
  puts i
end
```

Enfin, il y a la boucle **loop do... end** :

```
i = 1

loop do
  puts i
  i += 1
  break if i > 3
end
```

Il existe une forme courte de ces boucles. Exemple :

```
i = 0

print "#{i+=1} " while i < 4
```

L'expression « #{i} » signifie qu'il faut afficher la valeur de *i*. Cela peut être pratique. Les deux fragments de code suivants sont équivalents :

```
puts "Total : " + i.to_s
```

```
puts "Total : #{i.to_s}"
```

On peut aussi placer l'instruction de boucle en fin de bloc plutôt qu'au début. Exemple :

```
i = 0

begin
   print "#{i} "
   i += 1
end while i < 4
```

On peut même écrire quelque chose comme :

```
i = 4

puts (loop do i -= 1 ; print "#{i}, " ; break " Partez !" if i==1 ; end)
```

À quoi l'interpréteur répond « 3, 2, 1, Partez ! ».

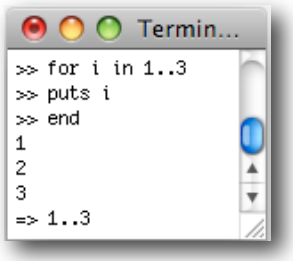
Les points-virgules remplacent le passage à la ligne pour séparer les instructions. Ils s'utilisent quand on veut écrire plusieurs instructions sur une seule ligne.

Comparaison des boucles

Les boucles for, while et until existent dans presque tous les langages.

Quelle est la différence entre les quatre formes de boucles ?

- *for... end* sert à faire un nombre prédéterminé de tours. C'est la forme de boucle la plus lisible et la plus compacte pour autant qu'on sache combien de tours il faut faire.
- *while... end* s'emploie quand le nombre de boucles n'est pas connu d'avance. Elle s'arrête quand une condition donnée cesse d'être vraie.
- *until... end* elle équivaut au while, mais l'arrêt se produit quand une condition donnée cesse d'être fausse (quand elle devient vraie).
- *loop do... end* est inconditionnelle. C'est l'itération la plus simple. Elle ne s'arrête que si l'on utilise l'instruction *break* pour placer une condition d'arrêt à l'intérieur de la boucle.



En résumé

Les notions essentielles de la programmation sont au nombre de quatre :

- les **objets**, soit tout ce qui se manipule : chaînes de caractères, nombres, tableaux, etc. ; les variables en font partie ;
- les **méthodes**, c'est-à-dire les actions possibles sur les objets ;
- les **classes** et l'organisation des objets qui va avec (héritage, polymorphisme,...) ;
- le **flux** du traitement : les branchements (si... sinon) et les boucles (répéter.... tant que...).