2ème partie

Exemples de programmes

On peut utiliser n'importe quel éditeur de texte pour écrire des programmes en Ruby, mais certains d'entre eux « comprennent » la syntaxe du langage, ce qui leur permet de coloriser le code, d'indenter automatiquement ce qu'on frappe, de vérifier la syntaxe de ce qu'on écrit, etc. Ils reconnaissent souvent le langage utilisé à l'extension du nom du fichier. Avec Ruby, c'est **rb**.

SciTE (*http://www.scintilla.org/SciTE.html*) et jEdit (*http://www.jedit.org*) sont populaires. SciTE est livré avec la version de Ruby pour Windows. Tous deux sont multi-plate-forme.

Le choix d'un éditeur est essentiellement une affaire de goût, mais il faut vérifier qu'il reconnaît le langage qu'on utilise. Les meilleurs sont même capables de faire des propositions pour compléter le code, ce qui diminue les risques d'erreur et fait gagner du temps :

```
17
   puts fact(nombre.to
               to                      RSS::UnknownConversionMethodError
               to                      URI::MailTo
               to                      DOT::DOTEdge
               to                      RSS::ConversionError
               to=(v)                  URI::MailTo
               toUTF8(str, encoding)   Tk
               to_a                    Array
               to_a                    Gem::Platform
               to_a                    Hash
               to_a                    Set
               to_a                    Resolv::DNS::Name
               to_a                    OptionParser
               to_a                    XMLRPC::DateTime
               to_a(xpath)             REXML::Elements
               to_a                    Shell::Filter
               to_a                    YAML::DBM
               to_a                    Benchmark::Tms
```

Pour le développement d'applications, on a essentiellement le choix entre Eclipse et NetBeans. Ce sont les deux IDE les plus populaires (IDE signifie *integrated development environments*). Ils ont d'abord été conçus pour Java, mais ils ont été étendus à d'autres langages actuels (C/C++, Javascript, PHP, Perl, Ruby, Python, etc.). Ils sont multi-plate-forme.

Eclipse est né chez IBM et NetBeans chez Sun, mais tous deux sont aujourd'hui open source.

Eclipse (*http://www.eclipse.org*) est le plus utilisé. Il offre beaucoup de fonctions natives et il peut en outre être complété par un grand nombre de plug-ins (d'extensions). Pour Ruby, il existe une distribution appelée EasyEclipse for Ruby and Rails (*http://www.easyeclipse.org/site/distributions/ruby-rails.html*). Eclipse est un bon choix si on recherche avant tout la richesse en fonctions, mais la configuration des plug-ins est parfois difficile.

NetBeans (*http://www.netbeans.org*) est également open source. Comme Eclipse, il offre un grand nombre de fonctions natives, mais le choix de plug-ins est plus restreint. Il y en a toutefois un pour

Ruby (*http://www.netbeans.org/features/ruby/index.html*) et il reconnaît les principaux langages actuels. Il me semble que NetBeans est plus facile d'emploi qu'Eclipse. Les exemples ci-dessous ont été réalisés avec NetBeans.

Remarque préliminaire

Contrairement à Windows, qui gère la correspondance entre les fichiers et les applications en se basant sur les extensions des noms de fichiers, les systèmes d'exploitation de la famille Unix ne connaissent pas la notion d'extension. Ils gèrent l'appartenance des fichiers en consultant les deux premiers caractères de la première ligne, qui s'appelle la *shebang line* (parce que le « # » se dit *sharp* et le « ! » se dit *bang*). Dans le cas de Ruby, c'est :

```
#!/usr/bin/env ruby
```

Attention, les signes *#!* doivent être les deux *premier*s caractères de la *première* ligne du fichier. Même une ligne vide ou un espace a pour résultat que Unix ne la trouve pas.

La shebang line ne sert à rien sous Windows, mais on peut sans problème la laisser dans le fichier parce que le signe « # » la signale comme étant un commentaire. En fait, la shebang line n'est comprise que par le système d'exploitation Unix. Ni les applications qui tournent sous Unix, ni Windows, ni Ruby n'en tiennent compte.

Calculer une moyenne

La méthode la plus simple pour calculer une moyenne consiste à demander à l'utilisateur quelle est la première valeur (*print "Note 1 ? "*), à placer sa réponse dans une variable (*rep1 = gets.to_f*), à refaire cela le nombre de fois nécessaire et à terminer en calculant la moyenne et en affichant le résultat :

```
1  #!/usr/bin/env ruby
2
3  # Calculer une moyenne, version 1
4
5  print "Note 1 ? "
6  rep1 = gets.to_f
7
8  print "Note 2 ? "
9  rep2 = gets.to_f
10
11 print "Note 3 ? "
12 rep3 = gets.to_f
13
14 puts "Moyenne : " + ((rep1 + rep2 + rep3) / 3).to_s
15
```

Mais cette façon de faire n'est pas bonne parce qu'elle n'est pas **générique**. Par exemple, si on veut calculer une moyenne sur 4 valeurs à partir de ce programme, il faut ajouter des lignes et ne pas oublier de corriger le calcul final en ajoutant *+ rep4* et en divisant le total par 4 au lieu de 3 :

```
13
14 print "Note 4 ? "
15 rep4 = gets.to_f
16
17 puts "Moyenne : " + ((rep1 + rep2 + rep3 + rep4) / 4).to_s
18
```

En fait, il faudrait modifier ce programme pour chaque changement dans le nombre des valeurs ou créer plusieurs versions (*moyenne_sur_3.rb*, *moyenne_sur_4.rb*, etc.).

Il faut mieux avoir un programme qui fonctionne avec n'importe quel nombre de valeurs à calculer. Or l'itération a justement pour but d'exécuter une suite d'instructions plusieurs fois.

```
1  #!/usr/bin/env ruby
2
3  # Calculer une moyenne, version 2
4
5  total = 0.0
6  i     = 1
7  rep   = 1
8
9  puts "\nCalcul moyenne (Enter pour terminer)"
10
11 while rep != 0
12   print "Note #{i.to_s} ? "
13   rep = gets.to_f
14   total = rep + total
15   i += 1
16 end
17
18 puts "Moyenne : " + (total/(i-2)).to_s
19
```

Dans cette deuxième version, on commence par initialiser trois variables. On a *total*, un nombre en virgule flottante, pour stocker la somme des notes, *i*, un nombre entier, pour afficher le numéro de la valeur (messages *Note 1 ?*, puis *Note 2 ?*, etc.), et *rep* pour initialiser la variable qui contient la réponse de l'utilisateur.

On utilise en outre *rep* pour commander la boucle : si l'utilisateur tape Enter, cela signifie qu'il n'a plus de valeur à entrer et qu'il faut sortir de la boucle.

À cause de son usage comme commande de la boucle, *rep* ne peut pas être initialisée à la valeur 0. Sinon, le traitement n'entrerait jamais dans l'itération puisque l'instruction de la ligne 11 signifie « faire la boucle tant que *rep* n'est pas égale à 0 ».

À la ligne 18, remarquer que la moyenne se calcule en divisant le total par *i-2* et non par *i*. Cela s'explique par le fait que *i* est initialisée à 1 et non à 0 et qu'elle est encore incrémentée de 1 à la ligne 15.

Au lieu de *while... end*, on peut utiliser la boucle *loop do*, mais cette boucle est inconditionnelle, ce qui veut dire qu'il faut ajouter un test à l'intérieur de l'itération. Ce sont les lignes 15 à 17 de la version ci-dessous du programme. Elles vérifient que la réponse n'est pas zéro et elles font sortir le traitement de la boucle si c'est le cas.

```
10
11 loop do
12   print "Note #{i.to_s} ? "
13   rep = gets.to_f
14   total = rep + total
15   if rep == 0
16     break
17   end
18   i += 1
19 end
20
21 puts "Moyenne : " + (total/(i-1)).to_s
22
```

Cette façon de faire est plus verbeuse — plus bavarde — que la boucle while puisqu'elle nécessite l'ajout des lignes 15, 16 et 17. En pratique, la construction while est donc préférable dans la majorité des cas.

Le fait que la sortie a lieu à la ligne 16 signifie que la ligne 18 n'est pas traitée à la fin du dernier tour, ce qui veut dire que *i* n'est pas incrémenté de 1 et qu'il faut donc calculer cette fois la moyenne en divisant le total par *i-1* plutôt que par *i-2*.

```
10
11  until rep == 0
12    print "Note #{i.to_s} ? "
13    rep = gets.to_f
14    total = rep + total
15    i += 1
16  end
17
```

Au lieu de while, on peut aussi utiliser until (voir ci-contre).

En fait, toutes les constructions itératives peuvent s'employer. Même un *for... end* est utilisable si on demande préalablement à l'utilisateur sur combien de notes il veut faire sa moyenne (ligne 7) :

```
1   #!/usr/bin/env ruby
2
3   # Calculer une moyenne, version 4
4
5   total = 0.0
6
7   print "\nSur combien de notes faut-il faire la moyenne ? "
8   n = gets.to_i
9
10  for i in 1..n
11    print "Note #{i.to_s} ? "
12    rep = gets.to_f
13    total = rep + total
14  end
15
16  puts "Moyenne : " + (total/i).to_s
17
```

Cette version est la plus compacte. Par contre, elle implique que l'utilisateur sache d'avance combien il va prendre de valeurs.

À la ligne 16, le calcul de la moyenne se fait en divisant *total* par *i*, ce qui est plus élégant que le *i-1* ou le *i-2* des versions précédentes. Cela s'explique par le fait que *i* s'incrémente à la ligne 10 et non pas à la fin de la boucle.

Suivre la valeur des variables qui s'incrémentent ne va pas de soi. Pour cela, on peut activer la fonction « exécuter pas à pas » de l'IDE qu'on utilise ou placer ici et là une ligne qui sert seulement à savoir où en est une variable. Par exemple, avec le programme ci-dessus, on pourrait ajouter une ligne après la 13 avec un simple *puts i* (il n'est pas nécessaire d'écrire *puts i.to_s* parce qu'il n'y a que le nombre à afficher ; Ruby ne se plaint donc pas d'un mélange comme il le ferait si on avait, par exemple, *puts i + " "*).

Remarquer en outre qu'on n'a plus besoin d'initialiser *i* comme on le faisait à la ligne 6 des versions précédentes. Elle s'auto-initialise à la ligne 10.

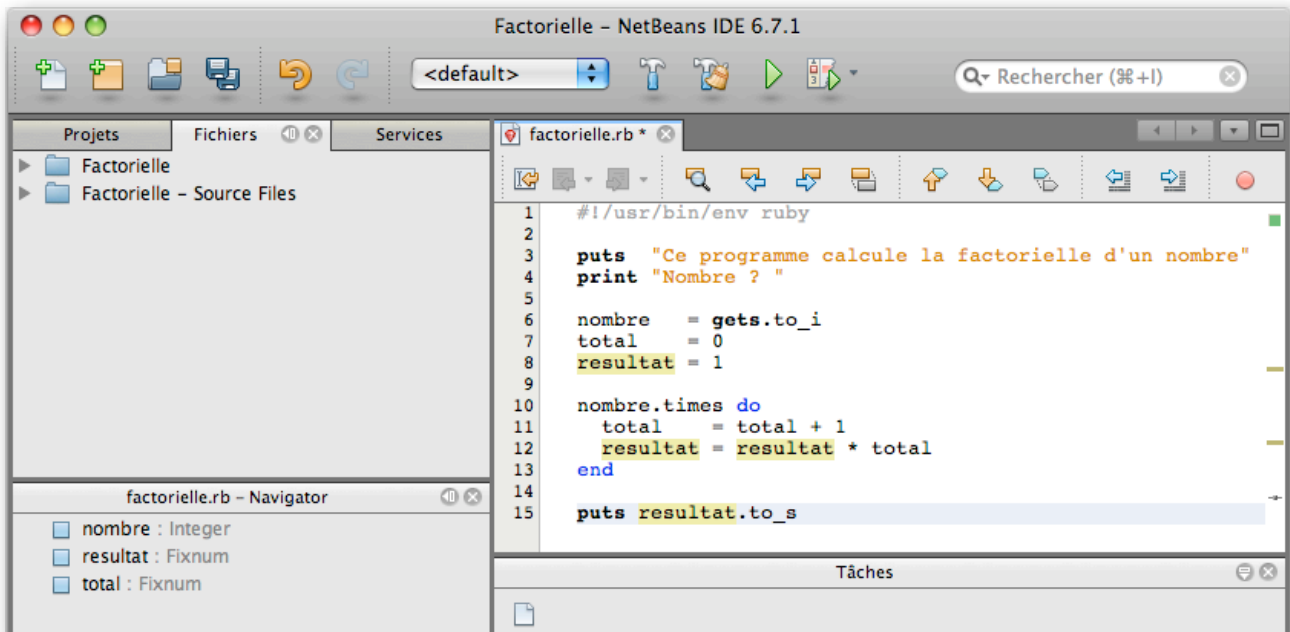
```
6    i = 0
7
8    print "\nSur c
9    n = gets.to_i
10
11   for i in 1..n
12     print "Note
```

Un IDE voit même ce genre de petites fautes (voir ci-contre). Si on initialise inutilement le compteur, NetBeans souligne le *i* de la boucle d'une vaguelette jaune et affiche un avertissement à la place du numéro de ligne sous la forme d'une ampoule électrique (« vérifier cette ligne ») et d'un signal triangulaire (« attention »).

Factorielle d'un nombre

La factorielle *n!* de *n* est le produit des nombres entiers de 1 à *n*. Par exemple, *5! = 1 x 2 x 3 x 4 x 5 = 120*.

En Ruby, ce calcul peut s'effectuer de plusieurs manières. Voici une version en une douzaine de lignes.



Remarquer que les IDE surveillent ce qu'on tape et signalent les erreurs qu'il est possible de découvrir pendant la frappe. De plus, ils répertorient les variables (au coin de la fenêtre en bas à gauche).

La ligne 1 est (par définition) la shebang line.

Les lignes 3 et 4 affichent les messages « Ce programme calcule la factorielle d'un nombre » et « Nombre ? ». À la ligne 4, on utilise *print* au lieu de *puts* pour éviter de passer à la ligne.

La ligne 6 lit la réponse de l'utilisateur au moyen de *gets* (*get string*). Comme la variable obtenue par cette méthode est une chaîne de caractères, il faut la convertir en nombre entier pour pouvoir l'utiliser pour des calculs. C'est ce qu'on fait avec la méthode *to_i* avant de placer le nombre entier résultant dans la variable *nombre*.

Par sécurité, on pourrait ajouter la méthode *abs* (valeur absolue) à la fin de cette ligne :

```
nombre = gets.to_i.abs
```

Cela évite qu'une erreur se produise si l'utilisateur tape un nombre négatif.

Les lignes 7 et 8 initialisent les variables *total* et *resultat* avec les valeurs 0 et 1.

La ligne 10 lance une boucle qui va s'exécuter le nombre de fois correspondant au nombre donné par l'utilisateur (c'est encore un autre itérateur : la méthode *times*, "fois"). Par exemple, si le nombre est *7*, le code des lignes 10 à 13 s'exécute 7 fois. La ligne 13 marque le bas de la boucle.

La ligne 11 incrémente de 1 la valeur de la variable *total*. On pourrait aussi écrire *total += 1*.

La ligne 12 multiplie la valeur de *total* par celle de *resultat* et stocke le résultat dans *resultat*.

La ligne 15 affiche le résultat après l'avoir converti en chaîne de caractère au moyen de la méthode *to_s*.

Avec cette solution, on crée un programme qui calcule les factorielles. Une autre manière de faire consiste à créer une *méthode*, ce qui permet ensuite de l'utiliser dans le programme aussi souvent que l'on veut :

```
1   nombre = 3
2
3   def fact(nombre)
4     if nombre == 0
5       1
6     else
7       nombre * fact(nombre-1)
8     end
9   end
10
11  puts fact(nombre.to_i)
12
```

Une fonction intéressante des langages de programmation est utilisée ici : la méthode *fact* s'appelle elle-même. C'est ce qu'on appelle la **récursion**.

Bien entendu, dans un programme réel, on n'emploierait que les lignes 3 à 9 et la méthode *fact* serait appelée avec l'argument souhaité et non pas avec un *nombre = 3* rigide.

Ruby permet d'écrire du code très concis et la méthode *fact* peut s'exprimer en une seule ligne (voir ci-contre). Le résultat est amusant, mais c'est généralement une mauvaise idée parce que cela rend les programmes difficiles à lire.

```
1
2  def fact(nombre)
3    nombre == 0 ? 1 : nombre * fact(nombre-1)
4  end
5
```

Jeu de la pierre, de la feuille de papier et des ciseaux

Dans le jeu de la pierre, de la feuille de papier et des ciseaux, deux joueurs s'affrontent. Au signal, ils tendent la main soit en faisant le poing (c'est la pierre), soit en mettant la main à plat (c'est la feuille de papier), soit en imitant une paire de ciseaux avec le majeur et l'index. La pierre gagne contre les ciseaux et perd contre la feuille de papier, les ciseaux gagnent contre la feuille de papier et perdent contre la pierre, la feuille de papier gagne contre la pierre et perd contre les ciseaux.

À la page suivante, voici un programme possible en Ruby, écrit sous NetBeans. Le logiciel permet à tout instant de tester le programme (c'est la partie grise en bas) au moyen de la touche F6. Eclipse a bien sûr aussi cette fonction.

La ligne 1 est la shebang line et les lignes 3, 4 et 5 contiennent des commentaires.

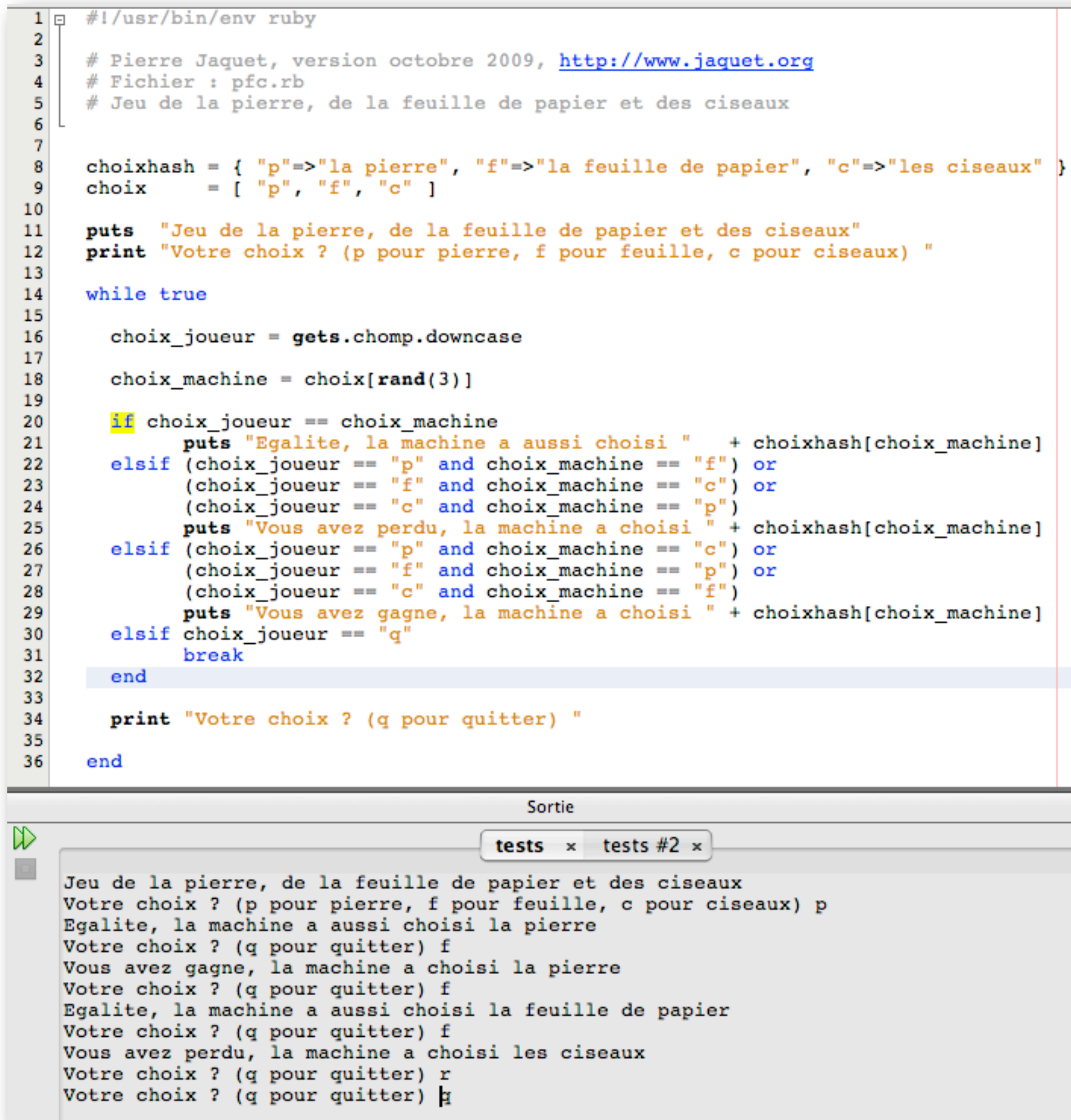
La ligne 8 crée un tableau associatif (un hash) et la ligne 9 un tableau simple.

Les lignes 11 et 12 affichent des messages.

De la ligne 14 à la 36, c'est la boucle centrale du programme. L'expression *while true* signifie « faire indéfiniment ».

À la ligne 16, le caractère entré par le joueur est stocké dans la variable *choix_joueur* après avoir été amputé des caractères « \n » au moyen de la méthode *chomp* puis mis en minuscule avec *downcase* pour le cas où la personne l'aurait tapé en majuscule. La méthode chomp est très utilisée. Sans elle, un "c" et un *Enter* tapés par l'utilisateur deviennent "c\n". Or, on est généralement intéressé uniquement par les caractères tapés avant le *Enter*.

À la ligne 18, la machine fait son choix au hasard (*rand*) entre les trois éléments du tableau *choix*, éléments qui sont "p" (pierre), "f" (feuille de papier) et "c" (ciseaux).


```
 1  #!/usr/bin/env ruby
 2
 3  # Pierre Jaquet, version octobre 2009, http://www.jaquet.org
 4  # Fichier : pfc.rb
 5  # Jeu de la pierre, de la feuille de papier et des ciseaux
 6
 7
 8  choixhash = { "p"=>"la pierre", "f"=>"la feuille de papier", "c"=>"les ciseaux" }
 9  choix     = [ "p", "f", "c" ]
10
11  puts  "Jeu de la pierre, de la feuille de papier et des ciseaux"
12  print "Votre choix ? (p pour pierre, f pour feuille, c pour ciseaux) "
13
14  while true
15
16    choix_joueur = gets.chomp.downcase
17
18    choix_machine = choix[rand(3)]
19
20    if choix_joueur == choix_machine
21          puts "Egalite, la machine a aussi choisi "   + choixhash[choix_machine]
22    elsif (choix_joueur == "p" and choix_machine == "f") or
23          (choix_joueur == "f" and choix_machine == "c") or
24          (choix_joueur == "c" and choix_machine == "p")
25          puts "Vous avez perdu, la machine a choisi " + choixhash[choix_machine]
26    elsif (choix_joueur == "p" and choix_machine == "c") or
27          (choix_joueur == "f" and choix_machine == "p") or
28          (choix_joueur == "c" and choix_machine == "f")
29          puts "Vous avez gagne, la machine a choisi " + choixhash[choix_machine]
30    elsif choix_joueur == "q"
31          break
32    end
33
34    print "Votre choix ? (q pour quitter) "
35
36  end
```



```
Jeu de la pierre, de la feuille de papier et des ciseaux
Votre choix ? (p pour pierre, f pour feuille, c pour ciseaux) p
Egalite, la machine a aussi choisi la pierre
Votre choix ? (q pour quitter) f
Vous avez gagne, la machine a choisi la pierre
Votre choix ? (q pour quitter) f
Egalite, la machine a aussi choisi la feuille de papier
Votre choix ? (q pour quitter) f
Vous avez perdu, la machine a choisi les ciseaux
Votre choix ? (q pour quitter) r
Votre choix ? (q pour quitter) q
```

De la ligne 20 à la 32, le programme compare le caractère entré par le joueur à la ligne 16 avec celui que le programme a choisi à la ligne 18.

La ligne 20 examine si le joueur et le programme ont fait le même choix, auquel cas il y a égalité. L'expression *choixhash[choix_machine]* consulte le contenu du tableau associatif *choixhash* pour afficher la chaîne de caractère complète plutôt que le caractère choisi. Par exemple, si le programme a choisi "c", le message sera « Égalité, la machine a aussi choisi les ciseaux », ce qui est plus clair que « la machine a aussi choisi "c" ».

Les lignes 22 à 24 déterminent les cas où le joueur perd contre la machine et la ligne 25 affiche le message qui annonce au joueur qu'il a perdu.

Remarquer que les lignes 22 à 24 et 26 à 28 forment une seule ligne logique :

22 : si le joueur a choisi la pierre et le programme la feuille de papier ou
23 : si le joueur a choisi la feuille de papier et le programme les ciseaux ou
24 : si le joueur a choisi les ciseaux et le programme la pierre

Avec la plupart des langages, une erreur serait générée à la fin de la ligne 22 parce que l'instruction est incomplète. Ruby, lui, voit que le dernier mot est « ou » et il en déduit que la ligne suivante fait partie de la même instruction.

Remarquer aussi l'importance des parenthèses, qui, comme en mathématiques, servent à grouper les éléments. Par exemple, *(1 + 2) * 3* donne *9* alors que *1 + 2 * 3* ou *1 + (2 * 3)* donnent *7*.

La ligne 30 détermine si le joueur a tapé "q" pour quitter le programme. Si c'est le cas, le *break* de la ligne 31 le fait sortir de la boucle et donc du programme.

La ligne 34 affiche le message qui demande son choix au joueur sous une forme plus compacte que le message de la ligne 12.

Ce programme est minimal et il y a plusieurs améliorations possibles. Par exemple, on peut accepter *x* (exit) comme moyen de sortie en plus de *q* (quitter) et on peut ajouter un *else* à la fin pour afficher un message d'erreur plus clair que « Votre choix ? (q pour quitter) » si le joueur tape autre chose que les lettres *p*, *f*, *c*, *q* ou *x*.

```
30      elsif choix_joueur == ("q" or "x")
31          break
32      else
33          puts "Tapez P pour pierre, F pour feuille ou C pour ciseaux "
34      end
```

Pour répéter les tables de multiplication

À la page suivante, voici un programme qui peut être très utile : il permet de répéter les tables de multiplication.

Les lignes 1 à 6 contiennent la shebang line et les commentaires.

Les lignes 9 et 10 affichent un message d'explication.

La ligne 12 demande à l'utilisateur d'indiquer la première table de multiplication à exercer et la 13 stocke sa réponse dans la variable qui s'appelle *premier*.

La ligne 15 fait de même pour la dernière table à exercer et la 16 stocke sa réponse dans *dernier*.

On voit qu'on n'a pas besoin de déclarer les variables en Ruby.

La ligne 19 inverse les valeurs des variables *premier* et *dernier* si *premier* est plus grand que *dernier*. Cela peut se faire en une seule instruction parce que Ruby reconnaît l'affectation parallèle. Sinon, il faudrait passer par une variable intermédiaire (de la même manière qu'on ne peut pas intervertir le contenu de deux verres sans utiliser un troisième verre).

Les lignes 22 et 23 stockent le nombre d'essais et le nombre d'essais ratés effectués par l'utilisateur. Ces informations sont utilisées pour afficher une statistique quand l'utilisateur sort du programme.

```ruby
#!/usr/bin/env ruby

# Pierre Jaquet, octobre 2009, http://www.jaquet.org
# Fichier : calculmental.rb
# Petit programme tout simple d'exercice des tables de multiplication
# écrit en Ruby (http://www.ruby-lang.org/fr/)


puts "\n\nIndiquez quelles tables de multiplication vous voulez exercer."
puts "Exemple : de 7 a 12  (faites Enter pour sortir)"

print "\nDepuis la table de  "
premier = gets.to_i

print "jusqu'a la table de "
dernier = gets.to_i

if premier > dernier
  premier, dernier = dernier, premier
end

essais    = 0
faux      = 0
continuer = true

while continuer

  nombre1 = (2..12).sort_by{rand}.first
  nombre2 = (premier..dernier).sort_by{rand}.first

  resultat = nombre1 * nombre2

  print "\n " + nombre1.to_s + " x " + nombre2.to_s + " = "

  reponse = gets.to_i

  unless (reponse == resultat) or (reponse == 0)
    faux += 1
    while resultat != reponse
      print "\nEssayez encore  "
      reponse = gets.to_i
      if reponse == 0
        essais += 1
        continuer = false
        break
      end
    end
  end

  break if reponse == 0

  essais += 1

end

justes = essais - faux
puts "Nombre de calculs  #{essais.to_s}"
puts "Calculs justes     #{justes.to_s}"
unless essais == 0
  puts "\nSoit             #{(100*justes/essais).to_i.to_s} % \n\n"
end

sleep 4
```

Pour l'initialisation de ces variables, on a le choix entre trois formes :

```ruby
essais = 0
faux   = 0
```

```ruby
essais = faux = 0
```

```ruby
essais = 0 ; faux = 0
```

Le point-virgule est le séparateur d'instructions quand on veut placer plusieurs instructions sur une seule ligne.

À la ligne 24 apparaît la variable *continuer* qui va être utilisée pour gérer la boucle principale du programme.

De la ligne 26 à la ligne 54, c'est la boucle principale du programme.

À la ligne 28, le programme prend la suite des nombres entiers de 2 à 12, arrange l'ordre de ces nombres au hasard (*sort by random*) et prend le premier de ces nombres avec la méthode *first*. Le résultat de ces opérations est le premier des deux nombres à multiplier.

À la ligne 29, il prend la fourchette des tables choisies par l'utilisateur (par exemple, de 7 à 9), il les réarrange également au hasard, ce qui donne par exemple 8, 9 et 7, puis il prend le premier de ces nombres (ici, c'est 8). Le résultat est le second des deux nombres à multiplier.

La ligne 31 fait le calcul et stocke le résultat dans la variable *resultat*.

La ligne 33 affiche le calcul à faire, par exemple *11 x 8 =* .

La ligne 35 place la réponse donnée par l'utilisateur dans la variable *reponse*. La méthode *to_i* en fait un nombre entier.

Les lignes 37 à 48 gèrent deux cas :

1° celui où la réponse de l'utilisateur est fausse (*unless reponse == resultat*, ce qui veut dire « sauf si reponse = resultat ») ;

2° celui où l'utilisateur fait *Enter* pour sortir (*unless reponse == 0*, « sauf si reponse = zéro ») .

La ligne 38 incrémente de 1 la valeur de la variable *faux* (qui a été initialisée à 0 à la ligne 23). L'expression *faux += 1* est synonyme de *faux = faux + 1*. C'est juste une affaire de goûts.

Les lignes 39 à 47 forment une boucle qui tourne tant que la réponse donnée par l'utilisateur diffère du résultat correct. L'expression *resultat != reponse* signifie « résultat différent de réponse ».

Aux lignes 40 et 41, le programme demande à l'utilisateur de faire un nouvel essai et il stocke la nouvelle réponse dans la variable *reponse*.

Les lignes 42 à 46 gèrent le cas où l'utilisateur veut sortir. Si la touche Enter a été pressée à la ligne 41, *reponse* contient la valeur 0. Le nombre d'essais est alors incrémenté de 1 et la valeur de la variable *continuer* est mise à *faux*. L'instruction *break* fait sortir l'utilisateur de la boucle et le fait que *continuer* soit faux cause aussi sa sortie de la boucle principale.

Si l'utilisateur a donné la réponse juste au calcul ou qu'il a fait *Enter*, il n'est pas passé par la condition *unless... end* des lignes 37 à 48. La ligne 50 le fait sortir s'il a fait *Enter*.

La ligne 52 incrémente la variable *essais*.

La ligne 54 termine la boucle principale du programme.

La ligne 56 calcule le nombre de réponses justes.

Les lignes 57 et 58 affichent le nombre de calculs faits et le nombre de calculs réussis.

S'il y a eu au moins un essai (*unless essais == 0*), le pourcentage de réussite est affiché à la ligne 60. On ne peut pas calculer ce pourcentage s'il n'y a eu aucun essai parce que cela générerait une

erreur, *justes/essais* étant une division par zéro. Pour la logique humaine aussi, cela n'a pas de sens de calculer un pourcentage sur le néant.

La ligne 63 suspend la sortie du programme pendant 4 secondes pour laisser à l'utilisateur le temps de lire les statistiques des lignes précédentes.

Le fait que les IDE répertorient les variables et les tableaux est très pratique pour limiter les risques d'erreurs. Par exemple, si on a écrit par erreur *résultat* quelque part dans le programme à la place de *resultat*, on peut le voir en consultant la liste (voir ci-contre).

```
continuer : TrueClass
dernier : Integer
essais : Fixnum
faux : Fixnum
justes
nombre1
nombre2
premier : Integer
reponse : Integer
resultat
résultat : Fixnum
```

Remarquer, pour terminer, à quel point l'indentation est importante pour la lisibilité du code : [1]

```
37    unless (reponse == resultat) or (reponse == 0)
38    faux += 1
39    while resultat != reponse
40    print "\nEssayez encore  "
41    reponse = gets.to_i
42    if reponse == 0
43    essais += 1
44    continuer = false
45    break
46    end
47    end
48    end
```

```
37    unless (reponse == resultat) or (reponse == 0)
38      faux += 1
39      while resultat != reponse
40        print "\nEssayez encore  "
41        reponse = gets.to_i
42        if reponse == 0
43          essais += 1
44          continuer = false
45          break
46        end
47      end
48    end
```

Web application frameworks

Ruby est très utilisé comme langage de *web application frameworks*, c'est-à-dire de logiciels de développement de sites et d'applications destinées au web.

Ruby on Rails (http://rubyonrails.org) est le plus connu et le plus utilisé de ces frameworks. Il existe des dizaines de milliers de sites Rails sur le web. Twitter est un exemple.

Références

Black, David A., *The Well-Grounded Rubyist*, Manning, New York, 2009 — c'est une très bonne introduction au langage et aux meilleures pratiques de programmation en Ruby.

Flanagan, David, Matsumoto, Yukihiro, *The Ruby Programming Language*, O'Reilly, New York, 2008 — un excellent livre, mais des compétences préalables en programmation sont nécessaires.

Brown, Gregory, *Ruby Best Practices*, O'Reilly, New York, 2009 — très intéressant, mais réservé aux personnes à l'aise avec les bases de Ruby.

http://www.ruby-doc.org : documentation sur le langage ; voir notamment http://www.ruby-doc.org/core — la référence complète sur le langage.

http://zamples.com/JspExplorer/content/rubyUG/index.html — un manuel de Ruby en ligne.

---

[1] Certains langages rendent l'emploi de l'indentation nécessaire : les programmes ne fonctionnent pas s'ils n'ont pas été indentés correctement. C'est le cas de Python, Occam et Haskell. Ce mode n'est pas activable dans les autres langages, mais les éditeurs de texte ont une option pour automatiser l'indentation.

En Ruby, la tradition veut qu'on indente de 2 caractères, mais rien n'empêche qu'on indente de 3 caractères, ou 4, ou plus, selon les goûts.

http://phrogz.net/ProgrammingRuby/frameset.html — version en ligne de la première édition du livre *Programming Ruby* de D. Thomas et A. Hunt (la 2ème édition est parue chez Pragmatic Bookshelf, New York, 2009).

http://www.ruby-lang.org/fr — site officiel du langage.

http://www.rubyfrance.org — site de l'association francophone des utilisateurs de Ruby.