Les chaînes

1. Construction

Toute chaîne est un objet de classe `String`. Elles sont généralement représentées entre apostrophes ou guillemets. Elles se composent d'un ensemble de caractères sur 1 octet. Un caractère quelconque c peut s'écrire `?c`.

Exemples :

```
puts  hello
puts "world"
```

La forme entre apostrophes n'est jamais interprétée alors que la forme entre guillemets peut comporter des séquences particulières de caractères. La première séquence est l'antislash (ou backslash) suivi d'un caractère ou d'un nombre. Ainsi le `\n` et `\r` participent à un retour à la ligne (`\n` pour saut de ligne et `\r` pour retour en début de ligne), `\t` insère une

tabulation, `\\` insère l'antislash et `\xHH` intègre un caractère suivant sa représentation hexadécimale :

Exemples :

```
puts  hel\nlo
# Nous obtiendrons le mŒme rØsultat : hel\nlo
puts "wo\nrld"
# Nous avons un retour à la ligne :
# wo
# rld
puts "wo\trl\x42"
# wo rlB
```

Comme nous l'avons déjà vu, la séquence `#{&}` sert à interpréter une expression dans une chaîne et mettre le résultat à la place.

Exemple :

```
puts "il est #{1+1} heures"
```

Nous obtenons alors :

```
il est 2 heures
```

Il est possible de créer une chaîne sur plusieurs lignes :

```
puts "il est #{1+1}
heures"
```

Cela donne en résultat :

```
il est 2
heures
```

On peut également utiliser d'autres délimiteurs grâce aux séquences %q et %Q. %q est pour une chaîne sans interprétation et %Q pour le contraire. Les caractères délimitant la chaîne sont à déterminer par vous-même, cela offre l'avantage de limiter un conflit entre les caractères de la chaîne et les caractères de délimitation.

Exemples :

```
puts %Q|Hello\nWorld|
puts %q\OK\
```

Nous obtenons alors en sortie :

```
Hello
World
OK
```

Grâce à l'opérateur `<<` vous pouvez créer votre propre séquence de caractères pour démarrer et arrêter une chaîne.

Exemple :

```
puts <<FIN
"hello
world"
FIN
```

Ce qui donne en sortie :

```
"hello
```

```
world"
```

2. Opérateurs

Les opérateurs `*` et `+` vont respectivement multiplier et additionner une chaîne, le résultat est toujours une nouvelle chaîne.

Exemples :

```
puts " terre en vue !" * 3
puts "quo " + "vadis"
```

Nous obtenons :

```
terre en vue ! terre en vue ! terre en vue !
quo vadis
```

L'opérateur `<<` est assez semblable au `+`, il effectue une concaténation, mais il est en plus capable d'effectuer une conversion d'un numéro de caractère.

Exemples :

```
a="tu quoque"
a<<" mi fili"
a<<63
puts a
```

Et nous obtenons :

```
tu quoque mi fili?
```

Pour comparer deux chaînes, il suffit d'utiliser l'opérateur `==`, sinon l'opérateur `<=>` indiquera une relation d'ordre entre deux chaînes en retournant une valeur négative, à zéro ou positive. Il existe également une variante de ce dernier opérateur par la méthode `casecmp` qui ignore les majuscules/minuscules.

Exemples :

```
puts "bonjour" == "bonjou"+"r"
# Le rØsultat est vrai
puts "aa" <=> "bb"
```

```
# -1 est retournØ car aa est avant bb
puts "bb" <=> "aa"
# +1 est retournØ car bb est aprŁs aa
```

L'opérateur `[]` permet d'accéder à une partie de la chaîne. Il prend un indice (commençant à 0) en argument et retourne alors la valeur entière du caractère correspondant. En spécifiant deux indices (borne inférieure et borne supérieure non incluse), nous obtenons la chaîne correspondante à ces positions, cela est également valable en passant un intervalle. Si une chaîne est passée en argument, la valeur `nil` sera retournée si cette chaîne n'est pas trouvée et sinon la chaîne elle-même. Une expression régulière peut également être utilisée.

Exemples :

```
a="bonjour"
puts a[0]
# Nous obtenons 98
if a[0] == ?b then
```

```
  puts "cela commence par  b"
end
# Nous avons bien le message  cela commence par b
puts ""<<a[0]
#nous avons la conversion en caractŁre : b
puts a[0,3]
# bon
puts a[0..2]
# bon
puts a["bon"]
# bon
puts a["joo"]
# nil
puts a[/b(.*)j/]
# Nous avons toute la chaî ne correspond à l expression rØguliŁre
puts a[/b(.*)j/, 1]
# Nous avons uniquement le groupe (grâce au deuxiŁme argument) :
```

3. Principales fonctions

Nous n'allons pas détailler toutes les fonctions à disposition mais présenter celles d'usage courant.

a. Fonctions de lecture

`count`

Cette fonction retourne le nombre de caractères de la chaîne en argument, si plusieurs arguments sont présents, l'intersection des caractères est utilisée. Enfin il est possible de déterminer une négation par `^` et un intervalle de caractères par `-`.

Exemples :

```
a="bonjour"
puts a.count( "o" )
# RØsultat = 2
puts a.count( "^o" )
# RØsultat = 5
puts a.count( "ou" )
# RØsultat = 3
puts a.count( "o", "ou" )
# RØsultat = 2
puts a.count( "a-j" )
# RØsultat = 2
```

`each`

En passant en argument un séparateur, elle sert à parcourir avec un bloc d'instructions, les ensembles de chaînes correspondantes.

Exemple :

```
a="bonjour"
a.each( o ) { |substr| puts substr }
```

On obtient en sortie :

```
bo
njo
ur
```

`each_byte`

Parcours de tous les caractères en passant à un bloc d'instruction le code de chaque caractère.

Exemple :

```
a="bonjour"
a.each_byte { |c| puts c  }
```

On obtient en sortie :

```
98
111
110
106
111
117
114
```

`each_line`

Elle sert à parcourir les lignes de la chaîne. Si on passe un argument, on se retrouve dans la configuration de la méthode `each`.

Exemples :

```
a="bonjour"
(a+"\nMonde").each_line() { |substr| puts substr  }
```

On obtient :

```
bonjour
```

```
Monde

(a+"\nMonde").each_line(  o  ) { |substr| puts substr  }
```

On obtient :

```
bo
njo
ur
Mo
nde
```

`empty?`

Indique s'il s'agit d'une chaîne vide.

Exemples :

```
puts "bonjour".empty?
# RØsultat : false
puts "".empty?
# RØsultat : true
```

`include?`

Indique si la chaîne inclut une autre chaîne.

Exemple :

```
a="bonjour"
puts a.include?( "bon" )
# RØsultat : true
```

`index`

Retourne la position d'une sous-chaîne passée en premier argument, il est également possible de passer une expression régulière ou un numéro de caractère, le deuxième argument est optionnel et représente une position de recherche de départ (supérieure à zéro). La variante `rindex` donne toujours la dernière position.

Exemples :

```
a="bonjour"
puts a.index(  jour  )
# RØsultat : 3
puts a.index(  o , 2 )
```

```
# RØsultat : 4
puts a.index( /[rj]/ )
# RØsultat : 3
```

`length`

Longueur de la chaîne (très similaire à count).

Exemple :

```
a="bonjour"
puts a.length
# RØsultat : 7
```

`match`

Prend en argument une expression régulière (soit par la syntaxe entre slash, soit sous forme de chaîne) et retourne la chaîne correspondante. Si des groupes sont positionnés dans l'expression régulière, un tableau avec les différentes valeurs est retourné.

Exemples :

```
a="bonjour"
puts a.match( /[on]/ )
# RØsultat : o
puts a.match(  [bo] )
# RØsultat : b
puts a.match( /(bon)(.*)$/ )[2]
# RØsultat : jour
```

scan

Cette fonction prend en argument une expression régulière et un bloc d'instructions en option. Elle parcourt toutes les parties de la chaîne correspondantes à l'expression régulière. S'il n'y a pas de bloc d'instructions, elle retourne un tableau résultat.

Exemples :

```
a="bonjour"

puts a.scan(/(o.)/ )[1]
# RØsultat : ou
puts a.scan( /(o.)/ ) {
```

```
  |res|
    puts res
}
```

Résultat :

```
on
ou
bonjour
```

`split`

Cette fonction découpe la chaîne en mots et retourne le résultat dans un tableau. Sans argument, elle utilise donc un séparateur « blanc ». Ce séparateur peut être passé en argument soit sous la forme d'une chaîne, soit sous la forme d'une expression régulière. Si l'expression régulière est vide (`//`), chaque caractère devient un résultat. Un deuxième argument peut être ajouté pour spécifier le nombre d'élément attendu dans le tableau résultat. Si ce nombre est inférieur au nombre de résultat, le dernier élément du tableau résultat est le reste de la chaîne.

Exemples :

```
a="bonjour"

a.split(  o  ).each { |i| puts i  }
```

Résultat :

```
b
nj
ur
```

```
a.split( /o./ ).each { |i| puts i  }
```

Résultat :

```
b
j
r
```

```
a.split( // ).each { |i| puts i  }
```

Résultat :

```
b
o
n
j
o
u
r
```

```
a.split( //, 4 ).each { |i| puts i  }
```

Résultat :

```
b
o
n
jour
```

```
d = "Dupond;Jean;01/01/45;Paris"
nom,prenom,dateNaissance,lieu = d. split( /;/ )
puts nom
puts prenom
puts dateNaissance
puts lieu
```

Résultat :

```
Dupond
Jean
01/01/45
Paris
```

b. Fonctions d'écriture

`capitalize`

Cette fonction convertit la première lettre en majuscule et le reste en minuscule.

Exemple :

```
puts "BONJour".capitalize
#RØsultat : Bonjour
```

`center`

Cette fonction sert à créer une chaîne d'une certaine taille en agglomérant si nécessaire et de manière homogène des caractères en début et fin de chaîne. Un premier argument, désigne le nombre de caractère

à atteindre, un deuxième argument en option désigne les caractères à ajouter. La fonction `ljust` effectue un travail similaire mais ajoute les nouveaux caractères à droite uniquement et la fonction `rjust` effectue la même chose mais en ajoutant les caractères à gauche.

Exemples :

```
puts "[" + "bonjour".center(20) + "]"
#RØsultat : [      bonjour       ]
puts "[" + "bonjour".center(20, - ) + "]"
#RØsultat : [------bonjour-------]
```

`chomp`

Cette fonction supprime par défaut les retours à la ligne (`\n`, `\r`, et `\r\n`). Si un argument est passé, il correspond à cette partie supprimée. La fonction `chop` est assez similaire, mais supprime toujours au moins le dernier caractère (elle est donc plus dangereuse à utiliser).

Exemples :

```
puts "Bonjour\n\n".chomp
#RØsultat : Bonjour (sans les deux retours à la ligne)
puts "Bonjour".chomp
#RØsultat : Bonjour
puts "Bonjour".chomp( "our")
#RØsultat : Bonj
```

`crypt`

Cette fonction crypte la chaîne. Elle prend en argument au maximum deux caractères pour altérer la forme cryptée. Une même chaîne et un même argument donnera toujours le même résultat. Il n'existe pas de fonction de décryptage, cette fonction sert à stocker une valeur sous sa forme cryptée comme un mot de passe et s'emploie ensuite à garantir qu'un mot de passe saisi, sera bien identique par comparaison avec la forme cryptée.

Exemples :

```
puts "Bonjour".crypt("ab")
#RØsultat : abqP9Re8TfQ7s
```

```
puts "Bonjour".crypt("abc")
#RØsultat : abqP9Re8TfQ7s
puts "Bonjour".crypt("ac")
#RØsultat : acUsv1z05/qzM
```

`delete`

Cette fonction supprime un ensemble de caractères. Cet ensemble de caractères est fourni de manière similaire à la fonction `count`.

Exemples :

```
puts "Bonjour".delete( "o" )
#RØsultat : Bnjur
puts "Bonjour".delete( "on" )
#RØsultat : Bjur
puts "Bonjour".delete( "^Bo")
#RØsultat : Boo
puts "Bonjour".delete( "a-n")
#RØsultat : Boour
```

`downcase`

Cette fonction effectue une conversion en minuscule, l'opposé étant la fonction `upcase`.

Exemples :

```
puts "Bonjour".downcase
#RØsultat : bonjour
puts "Bonjour".upcase
#RØsultat : BONJOUR
```

`gsub`

Cette fonction prend en argument une expression régulière et une chaîne ou un bloc d'instructions représentant la chaîne à substituer. C'est un peu comme une sorte de "chercher/remplacer". Pour utiliser le résultat d'un groupe de l'expression régulière, on utilise `\m` en remplaçant dans `m` le numéro du groupe. La fonction `gsub` effectue tous les *chercher/remplacer* possibles alors que la fonction `sub` ne l'effectue qu'une seule fois.

Exemples :

```
puts "Bonjour".gsub( /[on]/,  -  )
#RØsultat : B--j-ur

puts "Bonjour".gsub( /([on])/,  [\1]  )
#RØsultat : B[o][n]j[o]ur

puts "Bonjour".gsub( /([on])/ ) {
  |lettre|
  lettre.capitalize
}
#RØsultat : BONjOur
```

Attention au deuxième argument car si vous mettez des guillemets, le contenu est évalué avant les substitutions. Par exemple `\1` est interprété comme représentant un caractère numéro un.

`strip`

Cette fonction supprime les blancs (y compris les retours à la ligne) d'en-tête et de fin. Il existe

également une variante avec les fonctions `lstrip` et `rstrip` pour conserver ou non les blancs de début ou de fin.

Exemples :

```
puts "  ok  ".strip
# RØsultat :  ok
puts "  ok  ".lstrip
#RØsultat :  ok
puts "  ok  ".rstrip
#RØsultat :    ok
```

`reverse`

Cette fonction inverse l'ordre des caractères.

Exemple :

```
puts "bonjour".reverse
# RØsultat : ruojnob
```

`squeeze`

Cette fonction supprime les occurrences successives

d'un même caractère. Il est possible de passer en argument le ou les caractères concernés, par défaut tous les caractères sont pris en compte.

Exemples :

```
puts "boonjoour".squeeze
#RØsultat : bonjour
puts "boonnjoor".squeeze(  n  )
#RØsultat : boonjoor
```

`tr`

Cette fonction convertit les caractères du premier argument par les caractères du deuxième argument. Les caractères sont exprimés dans la même syntaxe que la fonction `count`.

Exemples :

```
puts "bonjour".tr(  bj ,  BJ )
# RØsultat : BonJour
puts "bonjour".tr(  a-n ,  *  )
# RØsultat : *o**our
```