

# Codage de texte et Expressions régulières

---

## Sommaire

Codage de texte et Expressions régulières .....	1
1 Introduction.....	2
2 Les expressions régulières.....	3
2.1 Les méta-caractères des expressions régulières.....	5
2.2 Le Backreferencing .....	8
2.3 Les Regex et le .NET Framework .....	9
2.3.1 Les méthodes de comparaison.....	10
2.3.2 La méthode de remplacement .....	12
2.3.3 Obtention de sous chaînes .....	14
3 Codage et décodage de texte.....	15
3.1 L'histoire des tables de codages.....	16
3.2 Le codage dans le .NET Framework.....	17
3.3 Utilisation des classes de codage .....	17
3.4 Le Codage et les flux.....	20
4 Conclusion .....	21



## 2 Les expressions régulières

Imaginons que nous créons un nouveau projet dans lequel l'utilisateur peut saisir des informations dont son adresse email par exemple. Nous pourrions vérifier la validité de son adresse via le code suivant:

```
'VB
Sub Main()
    Console.WriteLine("Entrez votre adresse email :")
    Dim valeur As String = Console.ReadLine()
    If (valeur.Contains("@")) Then
        Console.WriteLine("Adresse email valide")
    Else
        Console.WriteLine("Adresse email NON valide")
    End If
    Console.Read()
End Sub

//C#
static void Main(string[] args)
{
    Console.WriteLine("Entrez votre adresse email :");
    string valeur = Console.ReadLine();
    if (valeur.Contains("@"))
        Console.WriteLine("Adresse email valide");
    else
        Console.WriteLine("Adresse email NON valide");
    Console.Read();
}
```

Ce code fonctionne dans ce cas:

```
Entrez votre adresse email :
kikoo@lol.fr
Adresse email valide
```

Mais également dans ces cas là:

```
Entrez votre adresse email :
lol@
Adresse email valide
Entrez votre adresse email :
@
Adresse email valide
```

Le code nous indique qu'il s'agit d'adresse email valide alors qu'en réalité, elles ne le sont pas du tout.

En réutilisant le code ci-dessus, nous pourrions ajouter un test afin de vérifier si la chaîne de caractères contient également un point. Mais le problème reste entier puisqu'une adresse telle que "43-è\_ç@.kl^ù" ou encore "@." fonctionnerait également.

Si nous devons ainsi tester toutes les formes d'adresses email possibles et imaginables, nous perdrons beaucoup de temps et de performance à l'exécution.

Pour éviter ces tests successifs, les développeurs Unix et Perl utilisent depuis longtemps ce qu'on appelle les expressions régulières (Ou "Regex" en raccourci). Les regex permettent, non plus de tester si une chaîne contient ou non un ou plusieurs caractères, mais de vérifier si une chaîne de caractères concorde avec un "prototype" de chaîne de caractères.

Par exemple, si on traduit le code présenté ci-dessus, nous pourrions dire "si la chaîne de caractères contient un @".

Avec les regex, nous pourrions effectuer un test beaucoup plus complet qui se traduirait par : "si la chaîne de caractères commence par un maximum de 12 caractères alphanumériques, suivis du symbole @, lui-même suivi par une suite de 5 lettres au maximum, également suivie d'un point et se terminant par 3 lettres au maximum" et le tout en une seule ligne (ne vous occupez pas de la signification des caractères pour le moment):

```
'VB
Imports System.Text.RegularExpressions
Sub Main()
    Console.WriteLine("Entrez votre adresse email :")
    Dim valeur As String = Console.ReadLine()

    If (Regex.IsMatch(valeur, "^(\\w{1,12})@([a-z]{1,5})\\.([a-z]{1,3})$"), RegexOptions.None)) Then
        Console.WriteLine("Adresse email valide")
    Else
        Console.WriteLine("Adresse email NON valide")
    End If
    Console.Read()
End Sub

//C#
static void Main(string[] args)
{
    Console.WriteLine("Entrez votre adresse email :");
    string valeur = Console.ReadLine();
    if (Regex.IsMatch(valeur, @"^(\\w{1,12})@([a-z]{1,5})\\.([a-z]{1,3})$", RegexOptions.None))
        Console.WriteLine("Adresse email valide");
    else
        Console.WriteLine("Adresse email NON valide");
    Console.Read();
}
```

Cette fois-ci, si nous testons à nouveau les adresses précédentes, nous obtiendrons:

```
Entrez votre adresse email :
kikoo@lol.fr
Adresse email valide

Entrez votre adresse email :
lol@
Adresse email NON valide

Entrez votre adresse email :
abcdefghijklmnop@free.lol
Adresse email NON valide
```

Seule la première adresse remplit tous les critères à remplir pour concorder avec la forme de chaîne de caractères que l'on souhaite tester.

Les expressions régulières permettent de faire beaucoup plus (notamment sur le web où elles permettent de créer du pseudo-code (BBCode) afin de faciliter le traitement de texte sur les forums ou les blogs). Cependant, vous aurez vite remarqué qu'elles ne sont pas très simples à écrire et encore moins simples à décrire.

Il est important ici de comprendre que, par défaut, les expressions régulières sont sensibles à la casse, c'est-à-dire qu'elles font la différence entre les lettres majuscules et minuscules. ('a' et 'A' ont une casse différente).

## 2.1 Les méta-caractères des expressions régulières

Vous aurez pu constater grâce au code ci-dessus, que les expressions régulières sont construites grâce à une chaîne de caractères (un prototype) dont chaque caractère a une spécification particulière qui est indiquée dans ces tableaux. Dans la mesure du possible, chaque caractère sera illustré par un exemple rapide :

➤ Caractères de positionnement des recherches

Caractère	Description
<sup>1</sup> ^	Indique le début de la chaîne de caractères (Ex: "^a" concordera avec n'importe quelle chaîne de caractères commençant par la lettre "a"). Sur une chaîne multi ligne, indique le début de n'importe quelle ligne.
\$	Indique la fin de la chaîne de caractères (Ex: "a\$" concordera avec n'importe quelle chaîne de caractères se terminant par un "a"). Sur une chaîne multi ligne, indique la fin de n'importe quelle ligne.
<sup>2</sup> \A	Indique que la recherche doit commencer sur le premier caractère de la chaîne. Les retours à la ligne sont ignorés (Ex: "\A^a" concordera uniquement si le premier caractère de la première ligne de la chaîne de caractères est un "a").
<sup>2</sup> \Z	Indique que la recherche doit commencer sur le dernier caractère de la chaîne ou le dernier caractère situé avant un <sup>1</sup> \n terminal. Les retours à la ligne sont ignorés (Ex: "a\$\Z" concordera si les derniers caractères de la dernière ligne de la chaîne de caractères sont "a" ou "a\n").
<sup>2</sup> \z	Indique que la recherche doit commencer sur le dernier caractère de la chaîne. Les retours à la ligne sont ignorés (Ex: "a\$\z" concordera uniquement si le dernier caractère de la dernière ligne de la chaîne de caractères est "a").
<sup>2</sup> \G	Spécifie que la recherche débute là où la précédente recherche s'est arrêtée. Utilisée avec Match.NextMatch(), cela garantit qu'on ne saute aucune occurrence du prototype recherché.
<sup>2</sup> \b	Spécifie une recherche de mots composés de caractères alpha-numériques (Ex: "\bje\b" concordera uniquement si la chaîne de caractères contient le mot "je" séparé à droite et à gauche par des caractères non-alphanumériques).
<sup>2</sup> \B	Spécifie une recherche de morceaux de mots (Ex: "\Bjour\B" concordera uniquement si la chaîne de caractères contient le mot "jour" entouré à droite et à gauche par des caractères alphanumériques).

1 : Le caractère \n indique un retour à la ligne. En VB.NET, il s'agit du caractère vbCrLf (pour VB Line Feed)

2 : En C#, vous serez obligé de précéder vos prototypes de regex par le symbole @. De cette façon, s'ils contiennent un backslash, ils seront interprétés comme des caractères normaux et pas comme un caractère d'échappement.

## ➤ Caractères spéciaux

Caractère	Description
\a	Indique le caractère d'alarme (code Unicode: 0x0007).
\b	Indique le caractère de retour en arrière (code Unicode: 0x0008) uniquement s'il est utilisant dans un ensemble [].
\t	Indique le caractère de tabulation (code Unicode: 0x0009).
\r	Indique le caractère de retour chariot (code Unicode: 0x000D).
\v	Indique le caractère de tabulation verticale (code Unicode: 0x000B).
\f	Indique le caractère de défilement de ligne (code Unicode: 0x000C).
\n	Indique le caractère de retour à la ligne (code Unicode: 0x000A).
\e	Indique le caractère d'échappement (code Unicode: 0x001B).
\040	Représente un caractère de la table ASCII en représentation octale (les 3 octets sont obligatoires). Ici, \040 représente un espace.
\x20	Représente un caractère de la table ASCII en représentation hexadécimale (les 2 valeurs sont obligatoires). Ici, \x20 représente aussi un espace.
\cC	Représente un caractère de contrôle ASCII. Ici, \cC représente la combinaison de touches "CTRL+C".
\u0020	Représente un caractère de la table Unicode en représentation hexadécimale (les 4 valeurs sont obligatoires). \u0020 code aussi pour un espace.
\	Lorsqu'on souhaite rechercher un caractère d'expression régulière, permet d'indiquer à la Regex de ne pas traiter le caractère suivant en tant que prototype mais en tant que caractère normal (Ex: "\\" représente le caractère "\"). C'est le caractère d'échappement.
.	Indique n'importe quel caractère excepté un \n

## ➤ Quantificateurs

Caractère	Description
{x}	Spécifie que le caractère précédent doit apparaître x fois. (Ex: "a{5}" fonctionnera avec "cklaaaaalmp" ou "aaaaa").
{x,}	Spécifie que le caractère précédent doit apparaître au moins x fois (Ex: "a{2,}" fonctionnera avec "cklaa" ou "aa" mais pas avec "a").
{x,y}	Spécifie que le caractère précédent doit apparaître au moins x fois et au plus y fois (Ex: "a{2,3}" fonctionnera avec "aa" ou "aaa" mais pas avec "a" ou "aaaa").
+	Le caractère ou l'expression précédente peut apparaître 1 ou plusieurs fois. Equivalent à {1,}.
*	Le caractère ou l'expression précédente peut apparaître 0, 1 ou plusieurs fois. Equivalent à {0,}.
? <sup>1</sup>	Le caractère ou l'expression précédente peut apparaître 0 ou 1 fois. Equivalent à {0,1}. Si ce caractère est directement précédé d'un autre quantifieur, il retourne le nombre de répétition le plus petit possible (par exemple, sur la chaîne "oooo", si on teste "o+", il retournera tous les o trouvés alors "o+?" ne retournera qu'un seul o.

1 : Lorsque le ? suit un autre quantificateur comme \* ou +, il le transforme en mode « Ungreedy », c'est-à-dire que le quantificateur va s'arrêter dès la première occurrence contenant le moins de caractères possible. Cette technique peut être utile lorsqu'on souhaite effectuer du remplacement de chaînes de caractères (par exemple, remplacer <tbl.\* ?> dans la chaîne <tbl>test</tbl><tbl>test2</tbl> ne va affecter que le premier <tbl> disponible ce qui n'aurait pas été le cas avec <tbl.\*> qui aurait remplacé toute la chaîne).

## ➤ Conditions et ensembles de caractères

Caractère	Description
	Permet d'inclure un OU logique entre deux expressions (Ex: "ki oo" concordera avec n'importe quelle chaîne contenant soit "ki" soit "oo")
[abc]	Indique que la chaîne peut contenir l'une des lettres spécifiée dans les crochets (Ex: "[abc]" concordera avec n'importe quelle chaîne contenant un a, un b ou un c)
[a-z]	Indique que la chaîne peut contenir l'un des caractères situés entre a et z (Ex: "[1-9]" concordera avec n'importe quelle chaîne contenant un chiffre entre 1 et 9)
[^abcd]	Indique que la chaîne ne peut contenir aucun des caractères situés dans l'ensemble spécifié.
\d	Indique un nombre. Equivalent à [1-9]
\D	Indique tout ce qui n'est pas un nombre. Equivalent à [^1-9]
\s	Indique tout ce qui est un espace blanc. Equivalent à [\f\n\r\t\v]
\S	Indique tout ce qui n'est pas un espace blanc. Equivalent à [^\f\n\r\t\v]
\w	Indique n'importe quel caractère alphanumérique incluant l'underscore. Equivalent à [A-Za-z0-9_]
\W	Indique tout ce qui n'est pas un caractère alphanumérique incluant l'underscore. Equivalent à [^A-Za-z0-9_]
(abc)	Permet de spécifier un groupe indissociable de caractères (Ex: "t(ia ol)t" concordera avec tiat ou tolt)

## ➤ Les groupes nommés

Il est possible de créer des variables temporaires (appelées "groupes nommés") qui stockeront un résultat trouvé. Pour ce faire, vous devez utiliser la syntaxe "(?<nom>prototype)".

Par exemple, si nous cherchons "\[(?<valeur>\d)\]", cela concordera avec les chaînes contenant la forme de texte "[x]" où x est un chiffre décimal. Vous pourrez ensuite accéder à ce chiffre en utilisant la variable valeur.

## 2.2 Le Backreferencing

Le Backreferencing, c'est le fait d'utiliser les groupes nommés (explicitement ou non) avant de s'en servir à nouveau directement dans le prototype.

Pour cela, on utilise les caractères de regroupements (les parenthèses) pour définir la backreference. On utilise ensuite le caractère \, pour faire appel à un groupe nommé implicitement, ou le caractère \k, pour appeler un groupe nommé explicitement. Ainsi:

- `"(\d)\1"` crée une backreference nommée implicitement, indiquant n'importe quel chiffre décimal et réutilise la backreference de suite après. Ce code concorde donc avec toute chaîne contenant deux chiffres identiques qui se suivent.
- `"(?<char>\w)\k<char>"` crée une backreference nommée. Dans cet exemple, la backreference s'appelle char et est réutilisée de suite après. Ce code concorde donc avec toute chaîne contenant deux caractères identiques qui se suivent.

Comme les expressions régulières s'exécutent de manière procédurale, si deux backreferences nommées portant le même nom, sont définies dans la même Regex, seule la dernière backreference sera utilisée.

Nous avons également remarqué que les backreference sans nom sont implicitement nommées 1, 2, 3 etc... Il est tout à fait possible de nommer ces backreference en utilisant l'un de ces nombres et d'y accéder comme si c'était une backreference non nommée (Ex: `"(?<1>\w)\1"` est équivalent à `"(\w)\1"`).

Dotnet-France Assn



## 2.3 Les Regex et le .NET Framework

Tous les outils nécessaires à l'utilisation des Regex se trouvent dans l'espace de nom `System.Text.RegularExpressions`.

On y trouve notamment la classe `Regex` qui propose quelques méthodes statiques pour utiliser les regex:

Méthode	Description
<code>IsMatch</code>	Retourne un booléen indiquant si le prototype de chaîne de caractères a été trouvée dans la chaîne de caractères spécifiée.
<code>Match</code>	Retourne un objet de type <code>Match</code> contenant l'occurrence de la recherche du prototype dans la chaîne de caractères spécifiée.
<code>Matches</code>	Retourne un objet <code>MatchCollection</code> contenant les résultats de toutes les occurrences du prototype qui ont été trouvées dans la chaîne de caractères.
<code>Replace</code>	Dans une chaîne de caractères, remplace toutes les occurrences du prototype par une chaîne de caractères indiquée.
<code>Split</code>	Coupe une chaîne de caractères aux positions définies par un prototype.

Dans chacune de ces méthodes, nous pouvons indiquer l'une des valeurs de l'énumération `RegexOptions` (ou plusieurs valeurs en effectuant un OU logique entre chacune). A chaque valeur de l'énumération correspond également une lettre pour que l'on puisse spécifier des options supplémentaires à l'intérieur du prototype de recherche (toutes les options ne sont pas utilisables dans un prototype). Pour cela, nous utilisons les caractères de regroupement par exemple `"(?ix-ms)"` indique d'ignorer la casse et les espaces blancs du prototype et annule le multi-ligne ainsi que l'option `Singleline` (le caractère `-` indiquant la négation de l'option). Voici les options disponibles :

Valeur	Valeur Inline	Description
<code>None</code>		Aucune option
<code>IgnoreCase</code>	<code>i</code>	Spécifie la sensibilité à la casse.
<code>Multiline</code>	<code>m</code>	Spécifie le mode multi-ligne. Change le comportement des caractères <code>^</code> et <code>\$</code> .
<code>ExplicitCapture</code>	<code>n</code>	Spécifie que seule les captures nommées sont autorisées (avec la forme <code>"(?&lt;nom&gt;...)"</code> )
<code>Compiled</code>		Spécifie que le prototype est contenu dans un assembly compilé en code MSIL. Permet une exécution plus rapide.
<code>Singleline</code>	<code>s</code>	Spécifie le mode "une seule ligne". Change le comportement de <code>.</code> qui indique alors n'importe quel caractère excepté <code>\n</code> .
<code>IgnorePatternWhitespace</code>	<code>x</code>	Spécifie que n'importe quel espace blanc non échappé sera ignoré. Active les commentaires grâce au caractère <code>#</code> .
<code>RightToLeft</code>		Spécifie que la recherche sera effectuée en partant de la droite et non en partant de la gauche.
<code>ECMAScript</code>		Spécifie un comportement conforme à l'ECMAScript. Cette option ne peut être utilisée qu'avec <code>Multiline</code> et <code>IgnoreCase</code> .
<code>CultureInvariant</code>		Spécifie que les différences culturelles dans la langue ne sont pas prises en compte.

### 2.3.1 Les méthodes de comparaison

La méthode `IsMatch` est la plus simple de toutes. Elle prend en paramètre la chaîne que l'on souhaite tester, le prototype à tester et éventuellement des options.

```
'VB
Sub Main()
    Console.WriteLine("Entrez votre numero de telephone:")
    Dim numero As String = Console.ReadLine()
    Console.WriteLine("Validite du numero: {0}", Regex.IsMatch(numero,
    "^(\d{2}\u0020){4}\d{2}\u0020?$"))
End Sub

//C#
static void Main(string[] args)
{
    Console.WriteLine("Entrez votre numero de telephone:");
    string numero = Console.ReadLine();
    Console.WriteLine("Validite du numero: {0}", Regex.IsMatch(numero,
    @"^(\d{2}\u0020){4}\d{2}\u0020?$"));
}
```

Ce code se contente de vérifier qu'un numéro de téléphone est bien sous la forme "12 34 56 78 90". Toute autre forme indiquera que le numéro n'est pas correct.

La Regex utilisée est simple. Elle vérifie si la chaîne commence par 4 nombres de deux chiffres chacun, suivis d'un espace et se termine par un nombre de deux chiffres. Optionnellement, il est possible que le numéro soit terminé par un espace blanc.

Entrez votre numero de telephone:

05 61 45 67 89

Validite du numero: True

La méthode `Match` peut être utile si on souhaite récupérer des informations qui auraient pu être capturées au cours de la recherche. En effet, cette méthode retourne un objet de type `Match` qui contient tous les paramètres de la première occurrence trouvée dans la chaîne de caractères.

Dans la même optique, la méthode `Matches` retourne un objet `MatchCollection` (une liste d'objet `Match`) qui contient l'ensemble des occurrences du prototype dans la chaîne de caractères.

L'exemple ci-dessous demande d'entrer une phrase. Dans cette phrase, on peut saisir des tags sous la forme "tag:valeur". Ainsi, chaque tag pourra être récupéré par la suite et on pourra également récupérer la valeur entrée par l'utilisateur:

```
'VB
Sub Main()
    Console.WriteLine("Entrez une phrase (peut contenir des tag
'tag:valeur'):")
    Dim chaine As String = Console.ReadLine()
    Dim collection As MatchCollection = Regex.Matches(chaine,
"\u0020?(?<tag>\w+):( ?<valeur>\w+)\u0020?")
    For Each m As Match In collection
        Console.WriteLine("Le tag {0} contient la valeur {1}",
m.Groups("tag"), m.Groups("valeur"))
    Next
End Sub

//C#
static void Main(string[] args)
{
    Console.WriteLine("Entrez une phrase (peut contenir des tag
'tag:valeur'):");
    string chaine = Console.ReadLine();
    MatchCollection collection = Regex.Matches(chaine,
@"\u0020?(?<tag>\w+):( ?<valeur>\w+)\u0020?");
    foreach (Match m in collection)
    {
        Console.WriteLine("Le tag {0} contient la valeur {1}",
m.Groups["tag"], m.Groups["valeur"]);
    }
}
```

Ce qui donne:

```
Entrez une phrase (peut contenir des tag 'tag:valeur'):
Bonjour, je suis name:Kikoo firstname:LoL . Je suis un type:programme intelligen
t
Le tag name contient la valeur Kikoo
Le tag firstname contient la valeur LoL
Le tag type contient la valeur programme
```

### 2.3.2 La méthode de remplacement

Lors de vos recherches, vous pouvez remplacer uniquement des portions de textes tout en conservant des valeurs du prototype qui ont été capturées. Pour cela, vous devez créer une variable temporaire du mot que vous souhaitez conserver et dans le paramètre de remplacement de la méthode Replace, vous utiliserez "\${nom\_de\_variable}" pour récupérer ce qui a été sauvegardé.

Par exemple, nous pourrions vouloir remplacer tous les verbes à l'infinitif par leur impératif. Cet exemple ne va convertir que les verbes du premier groupe:

```
'VB
Sub Main()
    Dim verbe() As String = {"acheter", "achever", "priver", "manger",
    "poutrer"}
    For Each v As String In verbe
        Console.WriteLine(Regex.Replace(v, "(?<base>.+)(er)$",
    "${base}e", RegexOptions.Singleline))
    Next
End Sub

//C#
static void Main(string[] args)
{
    Console.WriteLine("---Retourne l'impératif--");
    String[] verbe = {"acheter", "achever", "priver", "manger",
    "poutrer"};
    foreach(string v in verbe)
    {
        Console.WriteLine(Regex.Replace(v, "(?<base>.+)(er)$",
    "${base}e", RegexOptions.Singleline));
    }
}
```

Plusieurs variables de substitution sont créées lorsque vous utilisez Replace:

Caractères	Description
\$nombre	Remplace par une capture nommée implicitement (en utilisant la forme "(expression)")
\${nom}	Remplace par une capture nommée explicitement (en utilisant la forme "(?<nom>expression)")
\$\$	Remplace par un seul \$
\$&	Remplace par une copie de tout le prototype.
\$`	Remplace tout le texte testé avant l'application du prototype
\$'	Remplace tout le texte testé après l'application du prototype
\$_	Remplace par le dernier groupe capturé
\$+	Remplace par toute la chaîne testée

Cet autre exemple, bien connu des concepteurs de forums, va ouvrir un fichier et convertir quelques balises BBcode simples de la façon suivante:

- `[code]donnee[/code]` en `<code>donnee</code>`
- `[code=valeur]donnee[/code]` en `<span style='code:valeur;'>donnee</span>`

```
'VB
Sub Main()
    Dim texte As String = File.ReadAllText("c:\Test\test1.txt")
    Console.WriteLine("---Avant transformation---" + vbNewLine + texte)
    texte = Regex.Replace(texte, "\[(?<code>[a-z]{1,3})\](?<donnee>.+)\[/\k<code>\]", "<${code}>${donnee}</${code}>",
    RegexOptions.Multiline)
    texte = Regex.Replace(texte, "\[(?<code>[a-z]{1,6})=(?<valeur>#([A-F]| [0-9]){6})\](?<donnee>.+)\[/\k<code>\]", "<span
    style='${code}=${valeur}'>${donnee}</${code}>", RegexOptions.Multiline)
    Console.WriteLine(vbNewLine + vbNewLine + "---Après transformation---" + vbNewLine + texte)
    Console.Read()
End Sub
```

```
//C#
static void Main(string[] args)
{
    string texte = File.ReadAllText(@"c:\Test\test1.txt");
    Console.WriteLine("---Avant transformation---" + "\n" + texte);
    texte = Regex.Replace(texte, @"\[ (?<code>[a-z]{1,3}) \] (?<donnee>.+ ) \[/\k<code>\]", "<${code}>${donnee}</${code}>",
    RegexOptions.Multiline);
    texte = Regex.Replace(texte, @"\[ (?<code>[a-z]{1,6}) = (?<valeur># ([A-F]| [0-9]) {6}) \] (?<donnee>.+ ) \[/\k<code>\]", "<span
    style='${code}=${valeur}'>${donnee}</${code}>", RegexOptions.Multiline);
    Console.WriteLine("\n" + "\n" + "---Après transformation---" + "\n" +
    texte);
}
```

---Avant transformation---

Bonjour,

Je [b]suis[/b] du texte

[b]traduit[/b] [i]en HTML[/i].

En plus, je peux être [color=#FF0000]rouge[/color]

---Après transformation---

Bonjour,

Je <b>suis</b> du texte

<b>traduit</b> <i>en HTML</i>.

En plus, je peux être <span style='color=#FF0000'>rouge</span>

### 2.3.3 Obtention de sous chaînes

Comme nous l'avons vu, il est également possible de scinder la chaîne d'entrée à chaque occurrence d'un prototype grâce à la méthode Split.

Voici un exemple qui coupe la phrase à chaque fois qu'elle trouve un chiffre décimal

```
'VB
Sub Main()
    Console.WriteLine("Entrez une phrase avec des chiffres:")
    Dim chaine As String = Console.ReadLine()
    Dim words() As String = Regex.Split(chaine, "\d",
RegexOptions.Singleline)
    Console.WriteLine(vbNewLine + vbNewLine + "----La chaine contient
les parties suivantes----")
    For Each s As String In words
        Console.WriteLine(s)
    NextEnd Sub
End Sub

//C#
static void Main(string[] args)
{
    Console.WriteLine("Entrez une phrase avec des chiffres:");
    chaine = Console.ReadLine();
    String[] words = Regex.Split(chaine, @"\d", RegexOptions.Singleline);
    Console.WriteLine("\n" + "\n" + "----La chaine contient les parties
suivantes----");
    foreach (string s in words)
    {
        Console.WriteLine(s);
    }
}
```

Ce qui donne:

```
Entrez une phrase avec des chiffres:
Bonjour je suis4une regex1très très5utile

----La chaine contient les parties suivantes----
Bonjour je suis
une regex
très très
utile
```

### 3 Codage et décodage de texte

Le codage et le décodage de textes n'est pas une notion facile à comprendre. Cela l'est d'autant moins avec le Framework .NET car il gère implicitement les codages et décodages par défaut.

Pourtant, le codage et le décodage de textes va nous être très utile dans plusieurs cas :

- Rendre nos documents interopérables avec d'autres systèmes d'exploitation
- Lire et écrire des documents dans plusieurs langues

Vous aurez par exemple besoin d'indiquer que vous utilisez une langue latine pour envoyer un e-mail à un japonais, ou bien d'utiliser un codage Unicode pour envoyer un document texte à une machine UNIX.

La table ci-contre nous montre le contenu de la table ASCII. Pour obtenir une valeur numérique correspondant à une lettre, on localise la lettre à coder, on regarde d'abord le numéro de colonne attribué puis le numéro de ligne ('A' est codé 41 en hexadécimal)

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

### 3.1 L'histoire des tables de codages

Le codage nous l'avons vu dans l'introduction tend à normaliser la manière de transformer du code machine en texte. Lorsque l'on code un texte, quelque soit le type de codage utilisé, celui-ci va être transformé en une séquence d'octets. Le décodage est le processus inverse, celui de transformer ces octets en texte.

Le premier type de codage normalisé est l'*American Standard Code For Information Interchange (ASCII)*. Il a été créé par l'*American National Standard Institute (ANSI)* et est le pilier des codages normalisés puisque la plupart s'en servent de base aujourd'hui.

L'ASCII est une table de caractères codée sur 7 bits (elle supporte donc 128 caractères). Chaque nombre compris entre 0 et 127 est assigné à un point de code. Cela peut être un caractère alphanumérique, un caractère spécial, ou une commande de contrôle. Par exemple, la lettre P correspondant en hexadécimal à 0x50, @ à 0x40, et la commande SPACE (espace) correspond à 0x20.

**Note :** Vous trouverez une représentation très détaillée de la table ASCII en suivant [ce lien](#).

L'organisme de normalisation international ISO a ensuite repris les bases de l'ANSI pour créer de nouveaux types de codage, notamment les ISO-8859-xx. L'ISO a redéfini la norme de codage en codant non plus sur 7 bits mais sur 8. Ainsi, de 0 à 127 correspond la table ASCII, puis de 128 à 255 correspondent tous les caractères spéciaux (lettres accentuées par exemple). Certaines langues contenant de nombreux caractères sont codées sur 16 bits (japonais, chinois...).

Le problème de ces codages en ASCII étendu à 8 bits est qu'ils font correspondre des caractères différents pour une même valeur sur la plage 127 à 255, il est alors impossible de mélanger dans un même document deux alphabets différents (Français et Hébreux par exemple).

Aussi, un nouveau standard appelé Unicode a été créé en 1991. Celui-ci couvre les caractères de plusieurs langages différents dont les caractères chinois, japonais, Cyrillique, Hébreux... Au total après la mise à jour de juillet 2006 (version 5.0), l'Unicode regroupait 245 000 points de codes. Vous pouvez bien entendu utiliser les caractères de plusieurs langues différentes dans un seul et même document !

Ainsi si vous envoyez deux versions d'un e-mail (l'un en codage ISO et l'autre en Unicode) à un Japonais, si celui-ci ne possède pas les informations nécessaires pour lire le codage ISO de votre langue, seul l'e-mail codé en Unicode aura gardé son intégrité.

Quand vous développez des programmes, veillez donc à toujours faire attention aux problèmes de codage. Une bonne pratique consiste à utiliser l'Unicode, en effet, il est géré sur toutes les machines par défaut.



### 3.2 Le codage dans le .NET Framework

Le Framework .NET gère plusieurs types de codages grâce à des classes contenues dans `System.Text`.

Codage	Description
ASCII/ISO/Unicode	Regroupe un large panel des codages ASCII, ISO et Unicode existants qui sont gérés par la classe <code>System.Text.Encoding</code> . Vous pouvez retrouver la liste en suivant <a href="#">ce lien</a> .
ASCII	Permet d'encoder en US ASCII sur 7 bits grâce à la classe <code>ASCIIEncoding</code> .
Unicode UTF-8	Permet d'encoder en Unicode UTF-8 grâce à la classe <code>UTF8Encoding</code> . L'UTF-8 est codé sur un à quatre octets et permet une transition facile en Unicode depuis un codage ANSI/ISO. Le Framework .NET utilise l'UTF-8 dans certains processus internes. Vous pouvez retrouver une large description de l'UTF-8 en suivant <a href="#">ce lien</a> .
Unicode UTF-16	Permet d'encoder en Unicode UTF-16 grâce à la classe <code>UnicodeEncoding</code> . L'UTF-16 est codé sur un ou deux mots de 16 bits. C'est le codage utilisé par le Framework .NET pour afficher un caractère. Vous pouvez retrouver une large description de l'UTF-16 en suivant <a href="#">ce lien</a> .
Unicode UTF-32	Permet d'encoder en Unicode UTF-32 grâce à la classe <code>UTF32Encoding</code> . L'UTF-32 est codé sur un mot double de 32 bits. Vous pouvez retrouver une large description de l'UTF-32 en suivant <a href="#">ce lien</a> .

### 3.3 Utilisation des classes de codage

Afin de comprendre comment fonctionne le codage d'un texte de façon explicite, voici une suite d'exemple :

```
'VB
Sub Main()
    Dim e As Encoding = Encoding.GetEncoding("iso-8859-1")
    Dim encode() As Byte
    encode = e.GetBytes("Bonjour!")
    For i As Integer = 0 To encode.Length
        Console.WriteLine("Octet {0}: {1}", i, encode(i))
    Next
    Console.Read()
End Sub

//C#
static void Main(string[] args)
{
    Encoding e = Encoding.GetEncoding("iso-8859-1");
    byte[] encode;

    encode = e.GetBytes("Bonjour!");

    for (int i = 0; i < encode.Length; i++)
        Console.WriteLine("Octet {0}: {1}", i, encode[i]);

    Console.Read();
}
```

Nous récupérons donc un objet de type `Encoding` grâce à la méthode `GetEncoding` de la classe `Encoding`. Puis nous codons grâce à cet objet le texte « Bonjour! ». Enfin nous affichons la représentation codée de notre texte. Voici ce que la console affiche :

```
Octet 0: 66
Octet 1: 111
Octet 2: 110
Octet 3: 106
Octet 4: 111
Octet 5: 117
Octet 6: 114
Octet 7: 33
```

Vous pouvez constater que les lettres ont toutes été transformées en valeurs binaires, qui sont affichées en décimal par la méthode `WriteLine` de la classe `Console`.

Comme nous avons utilisé les lettres contenues dans la table ASCII, vous pouvez voir que les valeurs sont comprises entre 0 et 127, le décodage sera possible dans n'importe quel codage utilisant au minimum un octet.

Si nous utilisons maintenant un texte codé en français et que nous le transformons en hébreu, voici ce qu'il va se passer :

```
'VB
Sub Main()
    Dim e As Encoding = Encoding.GetEncoding("iso-8859-1")
    Dim f As Encoding = Encoding.GetEncoding("iso-8859-8")
    Dim encode() As Byte

    encode = e.GetBytes("Bonjour! éèàôîûê")
    Console.WriteLine(f.GetString(encode))

    Console.Read()
End Sub

//C#
static void Main(string[] args)
{
    Encoding e = Encoding.GetEncoding("iso-8859-1");
    Encoding f = Encoding.GetEncoding("iso-8859-8");
    byte[] encode;

    encode = e.GetBytes("Bonjour! éèàôîûê");

    Console.WriteLine(f.GetString(encode));

    Console.Read();
}
```

```
Bonjour! ???????
```

La méthode `GetString` va transformer la suite d'octets en une chaîne de caractères. Comme vous pouvez le voir, tous les caractères accentués spécifiques à la langue française et qui n'existent pas en hébreu sont transformés en point d'interrogations, ce qui indique qu'ils n'ont pas pu être décodés.

Si vous souhaitez toutefois convertir dans un autre langage tout en gardant une certaine intégrité du texte, vous pouvez utiliser la méthode `Convert` de la classe `Encoding`.

```
'VB
Sub Main()
    Dim e As Encoding = Encoding.GetEncoding("iso-8859-1")
    Dim f As Encoding = Encoding.GetEncoding("iso-8859-8")
    Dim encode() As Byte

    encode = e.GetBytes("Bonjour! éèàôîûê")
    Dim decode() As Byte = Encoding.Convert(e, f, encode)
    Console.WriteLine(f.GetString(decode))

    Console.Read()
End Sub

//C#
static void Main(string[] args)
{
    Encoding e = Encoding.GetEncoding("iso-8859-1");
    Encoding f = Encoding.GetEncoding("iso-8859-8");
    byte[] encode;
    byte[] decode;

    encode = e.GetBytes("Bonjour! éèàôîûê");
    decode = Encoding.Convert(e, f, encode);

    Console.WriteLine(f.GetString(decode));

    Console.Read();
}
```

Bonjour! eeaoiue

### 3.4 Le Codage et les flux

Nous l'avons vu dans le chapitre précédent, il est possible de préciser l'encodage d'un fichier lorsque l'on ouvre un flux de type `StreamReader` ou `StreamWriter`. Nous allons effectuer un petit rappel en utilisant un type de codage plus exotique : l'UTF7.

```
'VB
Sub Main()
    Dim fluxEcriture As StreamWriter = New
StreamWriter("c:\Test\test.txt", False, Encoding.UTF7)
    fluxEcriture.WriteLine("Bonjour! éèàôîûê")
    fluxEcriture.Flush()
    fluxEcriture.Close()

    Dim fluxLecture As StreamReader = New
StreamReader("c:\Test\test.txt", Encoding.UTF7)
    Console.WriteLine(fluxLecture.ReadToEnd())
    fluxLecture.Close()

    Console.Read()
End Sub

//C#
static void Main(string[] args)
{
    StreamWriter fluxEcriture = new StreamWriter(@"c:\Test\test.txt",
false, Encoding.UTF7);
    fluxEcriture.WriteLine("Bonjour! éèàôîûê");
    fluxEcriture.Flush();
    fluxEcriture.Close();

    StreamReader fluxLecture = new StreamReader(@"c:\Test\test.txt",
Encoding.UTF7);
    Console.WriteLine(fluxLecture.ReadToEnd());
    fluxLecture.Close();

    Console.Read();
}
```

Nous écrivons ici un texte composé de caractères spéciaux (point d'exclamation et lettres accentuées) dans un fichier puis nous le lisons en utilisant le codage UTF7. La console indique le bon texte :

Bonjour! éèàôîûê

Mais si on tente de lire le fichier avec Notepad, voilà le résultat.

Bonjour+ACE- +AOkA6ADgAPQA7gD7AOo-

## 4 Conclusion

Nous avons vu qu'en informatique, les caractères tels qu'ils apparaissent à l'écran, sont codés grâce à des tables de caractères qui effectuent une correspondance entre un chiffre et une lettre, dans une ou plusieurs langues.

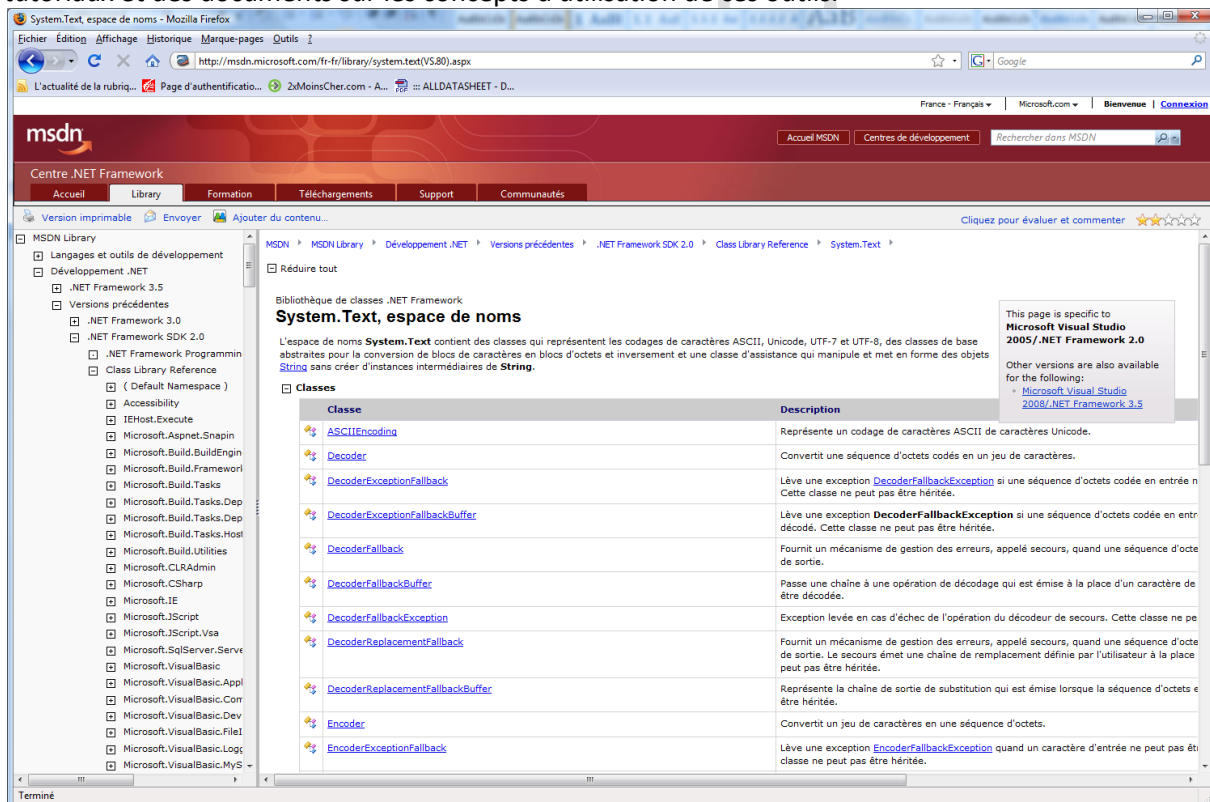
Nous avons également vu que le .NET Framework fournit un support assez complet permettant l'utilisation des expressions régulières. Ces expressions nous permettent d'effectuer des modifications sur du texte en utilisant non plus des séquences de caractères mais des prototypes de texte.

Concernant les expressions régulières, il ne vous est pas demandé de retenir absolument tous les caractères utilisables. Seuls les plus utiles doivent être connues.

Ainsi, à la fin de ce chapitre, vous devez être capable de :

- Comprendre l'utilité des expressions régulières
- Formuler des regex fonctionnelles (Savoir reproduire les exemples de ce chapitre devrait être une base).
- Comprendre l'intérêt des tables de codages.
- Savoir passer un texte d'un type de codage à un autre.

Dans tous les cas, vous pouvez aller visiter le [MSDN](http://msdn.microsoft.com) qui possède non seulement la documentation technique sur les outils d'encodage et d'expressions régulières mais également des tutoriaux et des documents sur les concepts d'utilisation de ces outils.



The screenshot shows the MSDN website interface. The main content area is titled 'System.Text, espace de noms' and describes the classes in the System.Text namespace. A table lists the following classes and their descriptions:

Classe	Description
<a href="#">ASCIIEncoding</a>	Représente un codage de caractères ASCII de caractères Unicode.
<a href="#">Decoder</a>	Convertit une séquence d'octets codés en un jeu de caractères.
<a href="#">DecoderExceptionFallback</a>	Lève une exception <a href="#">DecoderFallbackException</a> si une séquence d'octets codée en entrée n'est pas reconnue. Cette classe ne peut pas être héritée.
<a href="#">DecoderExceptionFallbackBuffer</a>	Lève une exception <a href="#">DecoderFallbackException</a> si une séquence d'octets codée en entrée n'est pas reconnue. Cette classe ne peut pas être héritée.
<a href="#">DecoderFallback</a>	Fournit un mécanisme de gestion des erreurs, appelé secours, quand une séquence d'octets de sortie n'est pas reconnue.
<a href="#">DecoderFallbackBuffer</a>	Passe une chaîne à une opération de décodage qui est émise à la place d'un caractère de sortie non reconnu.
<a href="#">DecoderFallbackException</a>	Exception levée en cas d'échec de l'opération de décodage de secours. Cette classe ne peut pas être héritée.
<a href="#">DecoderReplacementFallback</a>	Fournit un mécanisme de gestion des erreurs, appelé secours, quand une séquence d'octets de sortie n'est pas reconnue. Le secours émet une chaîne de remplacement définie par l'utilisateur à la place de la séquence non reconnue.
<a href="#">DecoderReplacementFallbackBuffer</a>	Représente la chaîne de sortie de substitution qui est émise lorsque la séquence d'octets de sortie n'est pas reconnue.
<a href="#">Encoder</a>	Convertit un jeu de caractères en une séquence d'octets.
<a href="#">EncoderExceptionFallback</a>	Lève une exception <a href="#">EncoderFallbackException</a> quand un caractère d'entrée ne peut pas être encodé. Cette classe ne peut pas être héritée.