Les Expressions Régulières (classe Regex)

Mécanisme fourni par .NET pour vérifier qu'une suite de caractères (une chaîne) satisfait un "motif" (pattern) décrit par une expression régulière, c'est-à-dire une expression finie pouvant néanmoins décrire des répétitions et des choix. Grâce à ce mécanisme on peut vérifier qu'une chaîne de caractères satisfait un format imposé.

Note : les types numériques (int,float,decimal) fournissent déja des méthodes Parse/TryParse pour vérifier le format et convertir une chaîne en valeur si le format est correct.

Ce que permettent les expressions régulières c'est vérifier qu'une suite de caractères satisfait certaines contraintes :

est-ce une suite de caractères tous compris entre '0' et '9' (format valable pour un entier base 10) ?
"123" -> OK "1e4" -> KO
est-ce une suite de caractères, sans chiffre, d'au moins 2 lettres et au plus 5 lettres ?
"art" -> OK "rtaert" -> KO (6 lettres) "ra4te" -> KO (pas uniquement des lettres)
est-ce une suite de caractères sans espace commençant par une lettre ?
"1ef23" -> KO "zy4t" -> OK
est-ce une suite de caractères où apparaissent 3 sous-chaînes avec la structure "entier;" ?
"12;3;4;" -> OK "12;3;4;6;" -> KO

Les objets **`Regex`** vont permettre de décrire par une expression finie (pattern) ce qu'on vient de décrire littéralement par des phrases.

Les expressions régulières .NET utilisent une chaîne de caractères pour décrire le motif (pattern). Ce motif représente le format attendu au moyen de caractères spéciaux. Ensuite, ce motif peut être appliqué à différentes chaînes de caractères pour vérifier si le format est correct, voire même si plusieurs sous-chaînes satisfont le format.

```
// Y-a-t-il 2 caractères successifs dans l'intervalle 'a' – 'g' ?
Regex re = new Regex(@"[a-g]{2}");   // motif "[a-g]{2}"
Match m = re.Match("chaine de caracteres"); // chaîne testée
if (m.Success)  // s'il y a une concordance (match)

{  Console.WriteLine(m.Value);   // = "de"    (sous-chaîne trouvée)
   Console.WriteLine(m.Index);   // =  7 (position dans la chaîne)


}  Console.WriteLine(m.Length);  // =  2  (taille de la sous-chaîne)
```

Note : si plusieurs sous-chaînes concordent avec le motif, la concordance correspond par défaut à la première dans le sens d'exploration gauche->droite.

Pour trouver toutes les sous-chaînes qui concordent avec le motif :

```
Regex re = new Regex(@"[a-g]{2}");
MatchCollection mc = re.Matches("chaine de caracteres");
foreach(Match m in mc) // pour chaque concordance
{
     Console.WriteLine(m.Value);
   Console.WriteLine(m.Index);
   Console.WriteLine(m.Length);
}
```

Pour cet exemple, dans `"chaine de caracteres"` on a le motif `"[a-g]{2}"`
"de" en position 7, "ca" en position 10 et "ac" en position 13

Le langage des patterns

Pour l'essentiel, l'utilisation des expressions régulières recquiert de comprendre la construction des patterns. La chaîne de caractères du pattern est constituée de caractères "classiques" et de commandes spécifiques qui ont un sens particulier dans le langage des patterns. On fournit ci-dessous les différents éléments de construction des patterns.

(1) : les caractères . $ ^ { [(|) * + ? \ } ont un sens particulier dans les patterns

Commande	Type de concordance	Pattern	Chaîne et résultat
Tous les caractères en dehors de (1)	Ces caractères n'ont pas d'autre sens et testent la concordance avec eux-même	`"ta"`	`"ta" en positions 0 et 5 dans "ta patate"`
\t	Teste la concordance avec une tabulation		
\r	Teste la concordance avec retour chariot		
\f	Teste la concordance avec form feed		
\n	Teste la concordance avec new line		
\xHH	Teste la concordance avec un caractère ASCII dont le code hexa à 2 digits est HH	`"\x61"`	`"a" en positions 0,3,5,7,10 dans "abracadabra"`
\uHHHH	Teste la concordance avec un caractère UTF-16 dont le code hexa à 4 digits est HHHH	`"\u00E7"`	`"ç" en position 2 dans "leçon"`
\. \$ \{ \[\(| \) * \+ \? \\	Teste la concordance avec les caractères spéciaux (1) .,$ ...	`"\{1\}"`	`"{1}" en position 2 dans "a={1}"`

Ensemble de caractères

[character_group]	Teste la concordance avec un caractère du groupe. Sensible à la casse par défaut.	`"[a-e]"`	`"a" en position 3 dans "gray"` `"a","e" dans "lane"`
[^ character_group]	Négation : teste la concordance avec les caractères non présents dans le groupe	`"[^aei]"`	`"r","g","n" dans "reign"`
[first - last]	Character range: teste la concordance avec l'un des caractères de l'intervalle	`"[A-Z]"`	`"A","B" dans "AB123"`
.	Wildcard: teste la concordance avec n'importe quel caractère excepté \n.	`"a.e"`	`"ave" dans "nave"` `"ate" dans "water"`
\w	Teste la concordance avec n'importe quel "word character." (lettre + _)	`"\w"`	`"I","D","A","1", "3" dans "ID A1.3"`
\W	Négation de \w	`"\W"`	`" ", "." dans "ID A1.3"`
\p{ name }	Teste la concordance avec un caractère de la catégorie Unicode 'name'(voir les catégories Unicode).Ex : Lu (Letter, Uppercase), Ll(Letter, Lowercase), Nd (Number, decimal digit) ...	`"\p{Lu}"` `"p\{Ll}"`	`"Ä","Q" dans "Äbd Qè"` `"b","d","è" dans "Äbd Qè"`
\P{name}	Négation de \p{name}		

\s	Teste la concordance avec un white-space character (espace, tab, ...)	**"\w\s"**	"D " dans "ID A1.3"
\S	Négation de \s	**"\s\S"**	" _" dans "int __ctr"
\d	Digit décimal	**"\d\d"**	"42" dans "a=423"
\D	Négation de \d : caractère non digit décimal	**"\D"**	"a=" dans "a=423"

Quantificateurs : commandes et caractères spéciaux qui décrivent les motifs répétés

*	L'élément précédent 0 ou plusieurs fois (greedy : prend le maximum d'occurences)	**"\d*\.\d"**	"19.9" et ".7" dans "a=19.9 b=.7"
+	L'élément précédent 1 ou plusieurs fois (greedy)	**"be+"**	"bee" dans "been" "be" dans "bent"
?	L'élément précédent 0 ou 1 fois	**"rai?n"**	"ran" et "rain" concordent
{n}	L'élément précédent exactement n fois	**",\d{2}"**	",12" en position 1 dans "7,1234"
{n,}	L'élément précédent au moins n fois (greedy)	**"\d{2,}"**	"12" et "456" dans "1 3 a12 b3 456"
{n,m}	L'élément précédent au moins n fois et au plus m fois (greedy)	**"\d{3,5}"**	"166" et "17668" concordent
*?	L'élément précédent 0 ou plusieurs fois (lazy : prend le minimum d'occurences)		
+?	L'élément précédent 1 ou plusieurs fois (lazy)		
??	L'élément précédent 0 ou 1 fois (lazy)		
{ n ,}?	L'élément précédent au moins n fois (lazy)		
{n,m}?	L'élément précédent au moins n fois et au plus m fois (lazy)		

Greedy vs. Lazy : par défaut les quantificateurs sont "greedy", ils recherchent la sous -chaîne de taille maximale satisfaisant le motif. Si l'on ajoute un ? après le quantificateur, c'est la sous-chaîne de taille minimale satisfaisant le motif qui est cherchée (lazy).

Exemple :

Pattern = "<a>.*<a>" Chaîne = "x=<a>b<a>c<a>d<a>e"
La sous-chaîne qui concorde est "<a>b<a>c<a>d<a>"

Pattern = "<a>.*?<a>" Chaîne = "x=<a>b<a>c<a>d<a>e"
Deux sous-chaînes concordent "<a>b<a>" en position 2 et "<a>d<a>"

Note: quand une concordance a été trouvée, l'analyse reprend dans la sous-chaîne restante

Pattern = "<a>.*?<a>" Chaîne = "x=<a>b<a>c<a>d"
Une seule sous-chaîne concorde "<a>b<a>" en position 2 puisque dans la sous-chaîne restante "c<a>d" on ne retrouve plus le motif.

Jalons (Anchors) : commandes qui testent des repères particuliers correspondant à des concordances de taille nulle (ne correspond à aucune sous-chaîne) : début de mot, fin de mot, début de ligne, fin de ligne ...

^	La concordance doit avoir lieu en début de chaîne ou de ligne	`"^\d"`	`"1" dans "1a 2r"`
$	La concordance doit avoir lieu en fin de chaîne ou de ligne (avant \n)	`"\d{3}$"`	`"-333" dans "-901-333"`
\A	La concordance doit avoir lieu en début de chaîne		
\z	La concordance doit avoir lieu en fin de chaîne		
\G	La concordance doit avoir lieu là où la concordance précédente a eu lieu. Permet de vérifier que les concordances sont contigües	`"\G\ (\d\)"`	`"(1)" "(3)" et "(5)" dans "(1)(3)(5)[7](9)"`
\b	En dehors d'un groupe de caractères, teste la concordance avec le début ou la fin d'un mot (séparation entre un alphanumérique \w et un \W)	`"\be"`	`"e" en positions 0 et 8 dans "ete tee etre"`
\B	Négation de \b : n'est pas le début ou la fin d'un mot	`"\Be"`	`"e" en positions 2,5 et 6 "ete tee"`

Groupements

Dans les motifs reconnus, on peut isoler des groupes qui correpondent à des sous-chaînes de l'ensemble. Dans une chaîne concordant avec un pattern donné, il peut y avoir des sous-chaînes satisfaisant des parties du pattern complet. Les groupes sont délimités par des parenthèses

```
// Motif = 2 digits : 2 digits
// Les 2 paires de digits sont placées dans des groupes
Regex re = new Regex(@"(\d{2}):(\d{2})");
Match m = re.Match("13:24");
 if (m.Success){
          GroupCollection gc = m.Groups;
          for (int i = 0; i < gc.Count;i++ )
          {
              Console.WriteLine(gc[i].Value);
              Console.WriteLine(gc[i].Index);
          }

 }
```

Sortie

```
13:24
0
13
0
24
3
```

Dans cet exemple, on isole les deux paires de digits (avant et après le :). A chaque groupe dans le motif correspond une sous -chaîne dans la collection gc. Le groupe d'indice 0 est la chaîne concordant avec le pattern global. Le groupe d'indice 1 est le premier groupe (le premier jeu de parenthèses dans le motif). Le groupe d'indice 2 est le second groupe (le second jeu de parnthèses).

On peut aussi réutiliser les groupes dans les concordances du reste de la chaîne

```
/* \1 = 1er groupe Le motif est ici '2 paires identiques séparées par :' */
Regex re = new Regex(@"(\d{2}):\1");
MatchCollection mc = re.Matches("13:13 14:14");
//deux concordances "13:13" en position 0 et "14:14" en position 6
```

(subexpr)	Capture la sous-expression dans un groupe (repéré par un numéro : le premier groupe a le numéro 1, le suivant numéro 2 ...)	`"(\d{2}):(\d)"`	"23:1" concorde "23" et "1" sont les groupes 1 et 2
\num	Fait référence à un groupe déja reconnu et mémorisé précédemment (num entier)	`"(\d):\1"` `"(\d):\d:\1"` `"(\d)(\d)\1\2"`	"2:2" concorde "1:4:1" concorde "1212" concorde
(?<name>sexpr) \k<name>	Le groupe est nommé name. \k<name> représente le groupe nommé name	`"(?<pa>\d{2}):\d:\k<pa>"`	"13:1:13" concorde
(?: subexpr)	Groupe sans capture (ne figure pas dans la collection Groups)		
(?= subex)	Requiert d'être suivi par le sous-motif subex sans le placer dans le match. Les concordances suivantes sont cherchées à la suite du match.	`"\d(?=\d{2})"` Ne concordent que les digits suivis de 2 digits. L'analyse reprend à la fin du match précédent.	"1" "2" "4" et "5" dans "1234 4567"
(?! subex)	Négation : concorde seulement si le sous-motif subex ne suit pas	`"\b(?!un)\w+\b"`	"sure", "used" dans "unsure sure unity used"
(?<= subex)	Requiert d'être précédé du sous-motif sans le placer dans le match.	`"(?<=19)\d{2}\b"`	"99", "50", "05" dans "1851 1999 1950 1905 2003"
(?<! subex)	Négation :concorde seulement si le sous-motif subex ne précède pas	`"(?<!19)\d{2}\b"`	"51", "03" dans "1851 1999 1950 1905 2003"

Remarque (groupes nommés) : quand un groupe est nommé, on peut obtenir la sous-chaîne correspondant au groupe par son nom dans la collection **`Groups`**.

```
Regex re = new Regex(@"(?<h>\d{2}):(?<m>\d{2})");
Match m = re.Match("12:24");
if (m.Success){
     string s1 = m.Groups["h"].Value;   // "12"

     string s2 = m.Groups["m"].Value;   // "24"
}
```

Alternatives

|	Concordance avec un des patterns séparés par |	`"th(e|is|at)"`	"the" et "this" dans "this is the day. "
(?(exp) m_1| m_2)	Concordance sous condition. Si exp est trouvé, cherche si le motif m_1 est trouvé, sinon cherche si m_2 est trouvé. (exp est un sous-motif de m_1)	`"(?(A)A\d|\b\d{3})"` Si A est trouvé, cherche s'il est suivi d'un digit. Sinon, cherche s'il y a un mot commençant par 3 digits.	"A1","910" dans "A10 C103 910"
(?(name)m_1| m_2)	(name) est un motif nommé (voir les groupements) Si le motif name est trouvé, on cherche si m_1 est reconnu. Sinon, on cherche si m_2 est trouvé		