3. Syntaxe des expressions régulières

3.1. Notion de classe

Nous arrivons maintenant au morceau de choix de ce chapitre. Comme nous l'avons déjà entrevu, les expressions régulières sont exprimées dans une syntaxe très précise, très hermétique, et très chatouilleuse. Des livres entiers y ont été consacrées, il ne s'agit donc pas ici d'en faire un cours complet, mais plutôt un aperçu qui permette de résoudre les cas simples, et d'éviter de tomber dans les pièges les plus fréquents. La syntaxe des expressions régulières Java est précisée dans la Javadoc à la page `Pattern`. Une expression régulière est une chaîne de caractères. Le premier principe est qu'un caractère se représente lui-même. Ainsi le pattern `"bonjour"` représente simplement le mot `"bonjour"`. Il est ensuite possible d'ajouter des caractères spéciaux à un pattern, de façon à enrichir ce qu'il représente. Par exemple, le pattern `"a*"` représente toutes les chaînes de caractères constituées d'un nombre quelconque de "a" (y compris la chaîne vide). Ajouter le caractère `"*"` à un pattern signifie que ce pattern peut se répéter. Nous avons également vu que le caractère `"."` pouvait représenter n'importe quel caractère. Nous en déduisons que le pattern `".*"`, que nous avons déjà utilisé, représente toutes les chaînes de caractères, y compris la chaîne vide. Il est possible ensuite de définir et d'utiliser des *classes de caractères* . Une classe de caractère est définie par une chaîne de caractères écrite entre crochets. Par exemple, la classe `"[abc]"` représente un *unique* caractère, qui peut être a ou b ou c. Voyons toutes les possibilités de définir une classe.

Tableau 3. Syntaxe d'une classe d'expression régulière

Classe de caractère	Signification
`[abc]`	Un unique caractère qui peut être a, b ou c.
`[^abc]`	Le ^ exprime la négation : cette classe représente un unique caractère, qui peut prendre toutes les valeurs, sauf a, b et c.
`[a-zA-Z]`	Le - signifie que tous les caractères entre ses bornes sont valides. Cette classe représente un unique caractère alphabétique, minuscule ou majuscule.
`[a-gmn]`	Autre exemple d'union : cette classe est constituée de tous les caractères compris entre a et g, du caractère m et du caractère n.
`[a-g[A-G]]`	On peut inclure des classes les unes dans les autres. Cette classe représente un unique caractère, compris entre a et g, en minuscule ou en majuscule. Elle est équivalente à [a-gA-G].
`[a-g&&[c-k]]`	Le signe && représente l'intersection. On fait donc là l'intersection entre la classe [a-g] et la classe [c-k]. Il s'agit donc de la classe [c-g].
`[a-g&&[^cd]]`	Ici on réalise l'intersection entre la classe qui représente tous les caractères de a à g, et celle qui représente tous les caractères, sauf c et d. Il reste donc a, b, e, f et g, que l'on peut aussi écrire `[abefg]` ou `[abe-g]`.
`[a-z&&[^m-p]]`	Autre exemple : ici on réalise l'intersection de tous les caractères compris entre a et z, et de tous les caractères sauf ceux compris entre m et p. Il nous reste donc `[a-lq-z]`.

Certains classes sont prédéfinies, et portent un nom particulier. Nous en avons en fait déjà vue une : `"."`. Cette classe particulière représente n'importe quel caractère. Voyons ici ces classes prédéfinies.

Tableau 4. Classes prédéfinies pour les expressions régulières

Classe prédéfinie	Classe équivalente	Signification
`.`		Un unique caractère quelconque.
`\d`	`[0-9]`	N'importe quel chiffre.
`\D`	`[^0-9]`	N'importe quel caractère qui n'est pas un chiffre.
`\s`	`[ \t\n\x0B\f\r]`	N'importe quel caractère blanc (espace, tabulation, retour-chariot, etc...).
`\S`	`[^\s]`	N'importe quel caractère qui n'est pas un blanc. Notons que l'on peut prendre la négation d'une classe prédéfinie.

Java API avancées
Cours & Tutoriaux
Retour au blog Java le soir
Sommaire
P. précédente P. suivante
Table des matières
API Collection
1. Introduction
2. Interface Collection
2.1. Notion de Collection
2.2. Détail des méthodes disponibles
2.3. Interface Iterator
2.4. Implémentation, exemples d'utilisation
3. Interface List
3.1. Notion de List
3.2. Détail des méthodes disponibles
3.3. Interface ListIterator
3.4. Implémentations, exemples d'utilisation
4. Interface Set
4.1. Notion de Set
4.2. Implémentations HashSet et LinkedHashSet
4.3. Exemples d'utilisation
5. Interface SortedSet
5.1. Notion de SortedSet

Classe prédéfinie	Classe équivalente	Signification
\w	[a-zA-Z_0-9]	N'importe quel caractère utilisable dans un mot (w est utilisé pour word). Cela représente les caractères alphabétiques minuscules et majuscules, les chiffres et le caractère souligné (underscore). Notons que les caractères accentués ne s'y trouvent pas...
\W	[^\w]	Inverse de la classe précédente.
\p{javaLowerCase}		N'importe quel caractère minuscule. Notons que cette fois, les caractères accentués s'y trouvent !
\p{javaUpperCase}		N'importe quel caractère majuscule. Même chose : les majuscules accentuées s'y trouvent.
\p{javaWhitespace}		N'importe quel espace.
\p{javaMirrored}		N'importe quel caractère écrit en miroir au sens de Unicode.

Ces quatre dernières classes font appel aux méthodes correspondantes de la classe `Character`. Tous les caractères `c` pour lesquels `Character.isLowerCase(c)` retourne `true` appartiennent à la classe `\p{javaLowerCase}`. On peut utiliser toutes les méthodes statiques de la classe `Character` du type `isProperty(char)` de cette façon, en utilisant la classe `\p{javaProperty}`. Il existe encore quelques caractères spéciaux, qui permettent de détecter des éléments particuliers d'un texte.

Tableau 5. Caractères spéciaux pour les expressions régulières

Caractère de début ou de terminaison	Signification
^	Un début de ligne.
$	Une fin de ligne.
\b	Le début ou la fin d'un mot.
\B	Le début ou la fin d'un élément qui n'est pas un mot.
\A	Le début d'une entrée.
\G	La fin du morceau de texte qui a été trouvé précédemment.
\Z	La fin d'une entrée, sauf s'il s'agit de la fin du texte.
\z	La fin d'une entrée, y compris s'il s'agit de la fin du texte.

Enfin voici les quantifieurs, avec leurs deux versions *greedy* et *reluctant* . Dans ces tableaux, *X* représente une classe quelconque.

Tableau 6. Caractères spéciaux pour les expressions régulières

Quantifieurs *greedy*	Quantifieurs *reluctant*	Signification
X ?	*X* ??	*X* apparaît 0 ou une fois.
X *	*X* *?	*X* apparaît un nombre quelconque de fois.
X +	*X* +?	*X* apparaît 1 fois et plus.
X {n}	*X* {n}?	*X* apparaît exactement *n* fois.
X {n,}	*X* {n,}?	*X* apparaît au moins *n* fois.
X {n, m}	*X* {n, m}?	*X* apparaît au moins *n* fois et au plus *m* fois.

Enfin signalons l'opérateur | : *X* | *Y* signifie que le caractère considéré doit appartenir soit à la classe *X* , soit à la classe *Y* . Voyons maintenant quelques exemples d'application, indispensables pour comprendre comment tout cela fonctionne.

3.2. Étude d'un cas réel

Prenons le texte suivant comme texte de référence.

```
String prevert =
        "Une pierre\n" +
        "deux maisons\n" +
        "trois ruines\n" +
        "quatre fossoyeurs\n" +
        "un jardin\n" +
        "des fleurs\n" +
        "\n" +
        "un raton laveur\n" +
        "\n" +
        "une douzaine d'huîtres un citron un pain\n" +
        "un rayon de soleil\n" +
        "une lame de fond\n" +
        "six musiciens\n" +
        "une porte avec son paillasson\n" +
        "un monsieur décoré de la légion d'honneur\n" +
        "\n" +
        "un autre raton laveur" ;
```

Appliquons le code suivant à ce texte.

Exemple 46. Exemple d'expressions régulières complexes

```
Pattern pattern = Pattern.compile(...) ;  // nous allons faire varier les paramètres de compile()

Matcher matcher = pattern.matcher(prevert) ;
 while (matcher.find()) {
    System.out.println(matcher.group()) ;
}
```

3.3. Recherche d'un mot précis

Tout d'abord, affichons tous les mots "un" de ce texte. Rien de plus simple, le pattern vaut "un". Si nous exécutons le code, on trouve bien les 10 "un" que compte ce texte. Hum ? En sommes-nous sûrs ? Combien y a-t-il de "un" dans ce texte ? Même après une nuit blanche passée à corriger des bugs, on doit arriver à 7, et non pas à 10 ! Les "une" n'auraient-ils pas été comptés avec les "un" ? Et si Prévert avait ajouté une "tunique" à son inventaire, n'aurait-elle pas été comptée dans le total ? Lorsque l'on cherche un mot précis, il faut toujours l'encadrer par un début de mot, et une fin de mot, sous peine de problèmes. Essayons à nouveau avec le pattern `"\\bun\\b"` (les double-barres sont là pour échapper la simple barre). Cette fois, l'on trouve bien 7. Notons que cette méthode ne permet pas de trouver les mots d'une lettre, comme les articles contractés suivis d'une apostrophe.

3.4. Recherche de deux mots précis

Compliquons à peine les choses : on veut maintenant les "un" du texte et aussi les "une". On peut construire le pattern suivant : `"(\\bun\\b)|(\\bune\\b)"`, qui va nous donner le bon résultat. Il signifie simplement : je cherche tous les mots valant exactement "un", ou valant exactement "une". On peut aussi demander les mots qui commencent par "un", et qui possèdent éventuellement un "e" ensuite, ce qui donne une écriture plus compacte. Un "e" qui peut être présent, ou ne pas être présent se note juste par "e?", ce qui donne le pattern `"\\bune?\\b"`. Dans ce cas, le point d'interrogation ne s'applique qu'au caractère qui le précède.

3.5. Recherche d'un mot commençant par une lettre donnée

Intéressons-nous à présent à tous les mots qui commencent par "r". Le premier pattern qui vient à l'esprit sera probablement celui-là : `"\\br.*\\b"` : un début de mot, un "r", suivi de n'importe quelle suite de caractères, puis une fin de mot. Malheureusement, le fonctionnement par défaut des expressions régulières est *greedy* , et le résultat est celui-ci :

```
ruines
raton laveur
rayon de soleil
raton laveur
```

Le matcher a bien pris les mots commençant par "r", mais il ne s'est arrêté qu'en fin de ligne. Il nous faut donc modifier le pattern. Une façon de faire est de dire qu'après le "r" on ne peut trouver que des caractères alphabétiques : `"\\br\\w*\\b"`. On trouve cette fois le bon résultat.

3.6. Cas de mots comportant des caractères accentués

Utilisons ce même pattern pour trouver tous les mots commençant par un "h" : `"\\bh\\w*\\b"`. Voici le résultat :

```
honneur
```

Le mot "huître" n'est pas trouvé ! Que s'est-il encore passé ? Rappelons-nous que la classe `\w` est équivalente à `[a-zA-Z]`. Le caractère "î" ne se trouve pas dedans, raison pour laquelle "huître" ne sort pas. Si l'on veut trouver n'importe quel caractère, il faut utiliser la méthode `isLetter()` de la classe `Character`, et donc écrire notre expression régulière de la façon suivante : `"\\bh\\p{javaLetter}*\\b"`. Avec ce pattern, le mot "huître" est bien détecté.

3.7. Recherche sur les lignes

Jouons à présent avec les débuts et fins de ligne. Pour ce faire, il faut tout d'abord indiquer à notre matcher que l'on prend en compte les passages à la ligne dans notre texte, en l'appelant avec le flag `Pattern.MULTILINE`. Sortons toutes les lignes commençant par un "d". Le code de début de ligne est le caractère `"^"`, celui de fin de ligne `"$"`. On peut donc construire le pattern suivant : `"^d.*$"`. Il signifie : un début de ligne, suivi du caractère "d", suivi d'une suite de caractères quelconques, suivi d'une fin de ligne. On trouve bien :

```
deux maisons
des fleurs
```

Si l'on n'avait voulu que le premier mot de chacune de ces lignes, on aurait utiliser le pattern suivant : `"^d\\p{javaLetter}*"` : un début de ligne, suivi du caractère "d", suivi d'un nombre quelconque de lettres. Le pattern suivant nous permet de sortir tous les derniers mots de chaque ligne : `"\\p{javaLetter}+$"`. Il signifie : un nombre non nul de lettres, suivi d'une fin de ligne. Là encore un piège nous guette, si des espaces ou des tabulations, bref des "blancs" se trouvent en fin de ligne, notre mot ne sortira pas. La tentation est grande d'écrire un pattern du type `"\\p{javaLetter}+\\s*$"`, mais malheureusement, les caractères de fin de retour à la ligne `"\n"` et `"\r"` se trouvent dans la classe `\s` ! On préfèrera donc une expression du type : `"\\p{javaLetter}+[ \t]*$"`.

Java API avancées

Cours & Tutoriaux

Retour au blog Java le soir

Sommaire

P. précédente **P. suivante**

Table des matières

API Collection

1. Introduction
2. Interface `Collection`
   - 2.1. Notion de Collection
   - 2.2. Détail des méthodes disponibles
   - 2.3. Interface `Iterator`
   - 2.4. Implémentation, exemples d'utilisation
3. Interface `List`
   - 3.1. Notion de `List`
   - 3.2. Détail des méthodes disponibles
   - 3.3. Interface `ListIterator`
   - 3.4. Implémentations, exemples d'utilisation
4. Interface `Set`
   - 4.1. Notion de `Set`
   - 4.2. Implémentations `HashSet` et `LinkedHashSet`
   - 4.3. Exemples d'utilisation
5. Interface `SortedSet`
   - 5.1. Notion de `SortedSet`