Syntaxe des expressions régulières – REGEX

Maj : 30/05/25

Pour créer une expression régulière, on utiliser une syntaxe particulière, à savoir des caractères spéciaux et des règles de construction. Par exemple, l'expression régulière simple ci-dessous correspond à un numéro de téléphone à 10 chiffres présenté sous la forme nnn-nnn-nnnn :

```
\d{3}-\d{3}-\d{4}
```

Le tableau ci-dessous présente certains des caractères spéciaux les plus couramment utilisés dans les expressions régulières. Ces caractères sont répartis selon les catégories suivantes :

Caractères	Description
Ancrages, Métacaractères	
^	(accent circonflexe) Établit une correspondance avec les éléments que l'expression régulière recherche au début d'une ligne ou d'une chaîne de texte. Prenons pour exemple une règle de contenu s'appliquant à l'objet du message qui contient l'expression régulière suivante : ^abc Cette règle capture tout e-mail dont l'objet commence par les lettres abc
$	(dollar) Établit une correspondance avec les éléments que l'expression régulière recherche à la fin d'une ligne ou d'une chaîne de texte. Prenons pour exemple une règle de contenu s'appliquant à l'objet du message qui contient l'expression régulière suivante : xyz$ Cette règle capture tout e-mail dont l'objet se termine par les lettres xyz
Quantificateurs	
?	0 ou 1 fois le caractère (ou le bloc) précédent le symbole
+	Au moins 1 fois le caractère (ou le bloc) précédent le symbole
*	Correspond à 0 ou un nombre quelconque d'occurrences du caractère précédent.
.	(point) Établit une correspondance avec n'importe quel caractère unique, à l'exception d'une nouvelle ligne.
|	(barre verticale) Indique une alternative, avec la valeur d'un "ou". Par exemple : chat|chien correspond au mot chat ou chien
\	Indique que le caractère qui suit doit être lu littéralement et n'est pas un caractère d'expression régulière. Cela permet d'échapper les caractères spéciaux. Exemple : \. correspond à un vrai point, et non à n'importe quel caractère unique (métacaractère du point).
Classes de caractères	
[...]	Établit une correspondance avec n'importe quel caractère d'un ensemble de caractères. Séparez le premier et le dernier caractère d'une série par un trait d'union. Par exemple : [123] correspond aux chiffres **1**, **2** ou **3** [a-f] correspond à n'importe quelle lettre de **A** à **F** **Remarques** : Les expressions régulières dans les règles de conformité du contenu sont sensibles à la casse.

[^...]	Établit une correspondance avec tout caractère ne figurant pas dans la liste de caractères indiqués. Par exemple : [^af] correspond à tout caractère qui n'est pas une lettre de **a** à **f :** **Remarque** : Les expressions régulières contenues dans les règles de conformité du contenu sont sensibles à la casse.
[:alnum:]	Établit une correspondance avec des caractères alphanumériques (lettres ou chiffres) : az , AZ ou 0-9 Remarque : Cette classe de caractères doit être entourée d'une autre paire de crochets lorsqu'elle est utilisée dans une expression régulière. Par exemple : [[:alnum:]].
[:alpha:]	Établit une correspondance avec tout caractère alphabétique (lettres) : **a-z** ou **A-Z** **Remarque** : Cette classe de caractères doit être entourée d'une autre paire de crochets lorsqu'elle est utilisée dans une expression régulière. Par exemple : [[:alpha:]].
[:digit:]	Établit une correspondance avec des chiffres : 0-9 Remarque : Cette classe de caractères doit être entourée d'une autre chaîne entre crochets lorsque vous l'utilisez dans une expression régulière. Par exemple : [[:digit:]].
[:xdigit]	N'importe quelle valeur hexadécimale → [0-9a-Fa-f]
[:graph:]	Établit une correspondance avec des caractères visibles uniquement, c'est-à-dire tout caractère à l'exception d'un espace, d'un caractère de contrôle, etc. Remarque : Cette classe de caractères doit être entourée d'une autre paire de crochets lorsqu'elle est utilisée dans une expression régulière. Par exemple : [[:graph:]].
[:punct:]	Établit une correspondance avec tout signe de ponctuation ou symbole : **! " # $ % & ' ( ) * + , \ -. / : ; < = > ? @ [ ] ^ _ ` { | }** Remarque : Cette classe de caractères doit être entourée d'une autre paire de crochets lorsqu'elle est utilisée dans une expression régulière : [[:punct:]].
[:print:]	Établit une correspondance avec tout caractère visible ou espace. **Remarque** : Cette classe de caractères doit être entourée d'une autre paire de crochets lorsqu'elle est utilisée dans une expression régulière. Par exemple : [[:print:]].
[:space:]	Établit une correspondance avec tout caractère d'espacement, y compris un espace, une tabulation et un saut de ligne. **Remarque** : Cette classe de caractères doit être entourée d'une autre paire de crochets lorsqu'elle est utilisée dans une expression régulière. Par exemple : [[:space:]].
[:word:]	Établit une correspondance avec tout caractère utilisé dans les mots en anglais, à savoir les lettres non accentuées, les chiffres ou le trait de soulignement : a-z, A-Z, 0-9, ou _ Remarque : Cette classe de caractères doit être entourée d'un autre ensemble de crochets lorsqu'elle est utilisée dans une expression régulière. Par exemple : [[:word:]].
Raccourcis pour les classes de caractères	
\w	Établit une correspondance avec tout caractère utilisé dans les mots en anglais, à savoir les lettres non accentuées, les chiffres ou le trait de soulignement : az, AZ, 0-9 ou _ Équivalent de [:word:]
\W	Établit une correspondance avec tout caractère qui n'est pas utilisé dans les mots en anglais, à savoir les caractères autres que les lettres non accentuées, les chiffres ou le trait de soulignement. Équivalent de [^[:word:]]
\s	Établit une correspondance avec tout caractère d'espacement. Utilisez ce caractère pour spécifier un

	espace entre les mots d'une expression. Par exemple : portefeuille\sboursier correspond à l'expression portefeuille boursier Équivalent de [:space:]
\S	Établit une correspondance avec tout caractère autre qu'un caractère d'espacement. Équivalent de [^[:space:]]
\d	Établit une correspondance avec tout chiffre compris entre 0 et 9. Équivalent de [:digit:]
\D	Établit une correspondance avec tout caractère autre qu'un chiffre entre 0 et 9. Équivalent de [^[:digit:]]
Groupe	
(...)	Regroupe les parties d'une expression. Utilisez les parenthèses pour appliquer un quantificateur à un groupe ou pour appliquer une classe de caractère avant ou après un groupe.
Quantificateurs	
{n}	Indique le nombre exact n d'occurrences consécutives de l'expression qui précède. Par exemple : [a-c]{2} correspond à toute lettre de **A** à **C** à condition ces lettres apparaissent deux fois de suite exactement. Ainsi, l'expression peut établir une correspondance avec **ab** et **ac**, mais pas **abc** ni **aabbc**.
{n,m}	Indique un minimum n et un maximum m d'occurrences consécutives de l'expression qui précède. Par exemple : [ac]{2,4} correspond à n'importe quelle lettre entre **a** et **c** à condition que les lettres apparaissent entre deux fois et quatre fois de suite. Ainsi, l'expression peut établir une correspondance avec **ab** et **abc**, mais pas **aabbc**.
?	Indique que le caractère ou l'expression qui le précède peut être présent ou non. Équivalent de la plage {0,1}. Prenons pour exemple l'expression régulière suivante : colou?r Elle peut correspondre à **colour**, mais aussi à **color**, car le **?** indique que la lettre **u** est facultative.

Expressions rationnelles avec antislash

Expression	Signification
\b	Chaine de début ou fin de mot (délimiteur de mots)
\B	Chaine qui n'est pas début ou fin de mot
\d	Un chiffre (équivalent de [0-9])
\D	Un non-chiffre (équivalent [^0-9])
.	Tout caractère
\<	Chaine vide en début de mot
\>	Chaine vide en fin de mot
\s	Caractères espace
\S	Non caractères espace
\w	Caractère alphanumérique : lettre, chiffre ou underscore
\W	Caractère qui n'est pas lettre, chiffre ou underscore

Exemples

BASH

```
#Suite de caracteres hexadecimauX
if expr "0BA2" : '^[[:xdigit:]][[:xdigit:]]*$' ; then echo "ok"; else echo "ko" ; fi
4
ok

#exemple
 if [[ "Wikibooks" =~ o+ ]]; then echo "ok"; else echo "ko"; fi
ok
 if [[ "Wikibooks" =~ a+ ]]; then echo "ok"; else echo "ko"; fi
ko
```

Exemple avec egrep (l'option -E permet d'utiliser le regex étendu) : recherche d'une @ip

```
tail -20 /var/log/nginx/access.log | egrep -Eo "([0-9]{1,3}\.){3}[0-9]{1,3}"
```

Using Regular Expression with SED

While matching patterns, you can use the regular expression which provides more flexibility. Check the following example which matches all the lines starting with *daemon* and then deletes them −

```
$ cat testing | sed '/^daemon/d'
```

```
root:x:0:0:root user:/root:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
```

Following is the example which deletes all the lines ending with **sh** −

```
$ cat testing | sed '/sh$/d'
```

```
sync:x:4:65534:sync:/bin:/bin/sync
```

The following table lists four special characters that are very useful in regular expressions.

Sr.No.	Character & Description
1	^ Matches the beginning of lines
2	**$** Matches the end of lines
3	. Matches any single character
4	*** Matches zero or more occurrences of the previous character
5	**[chars]** Matches any one of the characters given in chars, where chars is a sequence of characters. You can use the - character to indicate a range of characters.

Exemple concret de substitution :

Une ligne commence par 0 ou plusieurs espaces, suivi d'un mot en majuscule, suivi de 0 ou 1 espace, puis le caractère accolade ouvrante { pour finir. La ligne de substitution doit être identique à celle-ci, mais le mot "class" suivi d'un espace doit la précéder.

```
sed -E 's/^ *([A-Z]+) ?\{$/class \1 {/g'
```

- `^ *` : début de ligne suivi de 0 ou plusieurs espaces
- `([A-Z]+)` : capture un mot en majuscules
- `?` : 0 ou 1 espace
- `\{` : une accolade ouvrante
- `$` : fin de ligne

Matching Characters

Look at a few more expressions to demonstrate the use of **metacharacters**.
For example, the following pattern −

Sr.No.	Expression & Description
1	**/a.c/** Matches lines that contain strings such as **a+c**, **a-c**, **abc**, **match**, and **a3c**
2	**/a*c/** Matches the same strings along with strings such as **ace**, **yacc**, and **arctic**
3	**/[tT]he/** Matches the string **The** and **the**
4	**/^$/** Matches blank lines
5	**/^.*$/** Matches an entire line whatever it is
6	**/ */** Matches one or more spaces
7	**/^$/** Matches **blank** lines

Following table shows some frequently used sets of characters −

Sr.No.	Set & Description
1	**[a-z]** Matches a single lowercase letter
2	**[A-Z]** Matches a single uppercase letter
3	**[a-zA-Z]** Matches a single letter
4	**[0-9]** Matches a single number
5	**[a-zA-Z0-9]** Matches a single letter or number

Character Class Keywords

Some special keywords are commonly available to **regexps**, especially GNU utilities that employ **regexps**. These are very useful for sed regular expressions as they simplify things and enhance readability.

For example, the characters **a through z** and the characters **A through Z**, constitute one such class of characters that has the keyword **[[:alpha:]]**

Using the alphabet character class keyword, this command prints only those lines in the **/etc/syslog.conf** file that start with a letter of the alphabet −

```
$ cat /etc/syslog.conf | sed -n '/^[[:alpha:]]/p'
authpriv.*                          /var/log/secure
mail.*                              -/var/log/maillog
cron.*                              /var/log/cron
uucp,news.crit                      /var/log/spooler
local7.*                            /var/log/boot.log
```

The following table is a complete list of the available character class keywords in GNU sed.

	Character Class & Description
1	**[[:alnum:]]** Alphanumeric [a-z A-Z 0-9]
2	**[[:alpha:]]** Alphabetic [a-z A-Z]
3	**[[:blank:]]** Blank characters (spaces or tabs)
4	**[[:cntrl:]]** Control characters
5	**[[:digit:]]** Numbers [0-9]
6	**[[:graph:]]** Any visible characters (excludes whitespace)
7	**[[:lower:]]** Lowercase letters [a-z]
8	**[[:print:]]** Printable characters (non-control characters)
9	**[[:punct:]]** Punctuation characters
10	**[[:space:]]** Whitespace
11	**[[:upper:]]** Uppercase letters [A-Z]
12	**[[:xdigit:]]** Hex digits [0-9 a-f A-F]

Aampersand Referencing

The **sed metacharacter &** represents the contents of the pattern that was matched. For instance, say you have a file called **phone.txt** full of phone numbers, such as the following −

```
5555551212
5555551213
5555551214
6665551215
6665551216
7775551217
```

You want to make the **area code** (the first three digits) surrounded by parentheses for easier reading. To do this, you can use the ampersand replacement character −

```
$ sed -e 's/^[[:digit:]][[:digit:]][[:digit:]]/(&)/g' phone.txt
(555)5551212
(555)5551213
(555)5551214
(666)5551215

(666)5551216cette ligne commence par 0 ou plusieurs espaces, suivi d'un mot en majuscule, suivi de 0 ou 1 espace, puis le caractère accolade ouvrante { pour finir.
La ligne de substitution doit être identique à celle-ci, mais le mot "class" suivi d'un espace doit la précéder.
ChatGPT a dit :

Bien sûr ! Voici une commande sed qui fait exactement ce que tu demandes :

sed -E 's/^ *([A-Z]+) ?\{$/class \1 {/g'

Explication de l'expression régulière :

    ^ * : début de ligne suivi de 0 ou plusieurs espaces

    ([A-Z]+) : capture un mot en majuscules

    ? : 0 ou 1 espace

    \{ : une accolade ouvrante

    $ : fin de ligne

La substitution ajoute class suivi d’un espace avant le mot en majuscules capturé.
```

Here in the pattern part you are matching the first 3 digits and then using **&** you are replacing those 3 digits with the surrounding **parentheses**.

Using Multiple sed Commands

You can use multiple sed commands in a single sed command as follows −

```
$ sed -e 'command1' -e 'command2' ... -e 'commandN' files
```

Here **command1** through **commandN** are sed commands of the type discussed previously. These commands are applied to each of the lines in the list of files given by files.

Using the same mechanism, we can write the above phone number example as follows −

```
$ sed -e 's/^[[:digit:]]\{3\}/(&)/g' -e 's/)[[:digit:]]\{3\}/&-/g' phone.txt
(555)555-1212
(555)555-1213
(555)555-1214
(666)555-1215
(666)555-1216
(777)555-1217
```

Note − In the above example, instead of repeating the character class keyword **[[:digit:]]** three times, we replaced it with **\{3\}**, which means the preceding regular expression is matched three times.

Back References

The **ampersand metacharacter** is useful, but even more useful is the ability to define specific regions in regular expressions. These special regions can be used as reference in your replacement strings. By defining specific parts of a regular expression, you can then refer back to those parts with a special reference character.

To do **back references**, you have to first define a region and then refer back to that region. To define a region, you insert **backslashed parentheses** around each region of interest. The first region that you surround with backslashes is then referenced by **\1**, the second region by **\2**, and so on.

Assuming **phone.txt** has the following text −

```
(555)555-1212
(555)555-1213
(555)555-1214
(666)555-1215
(666)555-1216
(777)555-1217
```

Try the following command −

```
$ cat phone.txt | sed 's/\(.*)\)\(.*-\)\(.*$\)/Area code: \1 Second: \2 Third: \3/'

Area code: (555) Second: 555- Third: 1212
Area code: (555) Second: 555- Third: 1213
Area code: (555) Second: 555- Third: 1214
Area code: (666) Second: 555- Third: 1215
Area code: (666) Second: 555- Third: 1216
Area code: (777) Second: 555- Third: 1217
```

Note − In the above example, each regular expression inside the parenthesis would be back referenced by **\1**, **\2** and so on.

JAVA

Utilisation des classes Pattern et Matcher

Importations nécessaires des librairies : java.lang.Boolean; import java.util.regex.*;

```
public static boolean matches(String regex, CharSequence input) {
    Pattern p = compile(regex);
    Matcher m = p.matcher(input);
    return m.matches();
}
```

```
package org.o7planning.tutorial.regex.stringmatches;

public class EitherOrCheck {

    public static void main(String[] args) {

        String s = "The film Tom and Jerry!";

        // Check the whole s
        // Begin by any characters appear 0 or more times
        // Next Tom or Jerry
        // End with any characters appear 0 or more times
        // Combine the rules:., *, X | Z
        // ==> true
        boolean match = s.matches(".*(Tom|Jerry).*");
        System.out.println("s=" + s);
        System.out.println("-Match .*(Tom|Jerry).* " + match);

      }
}
```

Trouver le nombre d'occurences grâce à la méthode find() :

```
 public static int runTest(String regex, String text) {
    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(text);
    int matches = 0;
    while (matcher.find()) {
        matches++;
    }
    return matches;
}
```

When we get 0 matches, the test should fail; otherwise, it should pass.

PHP

Pour rechercher la présence d'un caractère particulier, il suffit de l'inclure entre des guillemets simples ou doubles. Pour rechercher le caractère #, vous écrivez le modèle :

```
$modele="/#/";
```

```
$modele="/[abc]/";
```

pour rechercher un ou plusieurs des caractères a, b, c

```
$modele = "/[0-9]/";
```

recherche la présence d'au moins un chiffre entre 0 et 9.

```
$modele="/[[:digit:]]/";
```

vous permet de rechercher la présence de chiffres.

Recherche par motif : le motif suivant correspondant au mot qui sera recherché, en respectant la casse :

```
$modele = "/Amsterdam/";
```

La fonction de recherche (parmi d'autres) permettent de retourner un booléen, en fonction d'un motif.
Avec les fonctions standard de PHP :

bool preg_match (string $modele, string $ch [, array $tab])
Fonction élémentaire de recherche

Exemple : Recherche dans une chaîne

```php
<?php
$ch="1515 Bataille de Marignan";
$modele= "/([[:digit:]]{4}) (.*)/"; if(preg_match($modele,$ch,$tab))
{
echo "La chaîne \"$ch\" est conforme au modèle \"$modele\" <br />";
echo "Date : ",$tab[1],"<br />";
echo "Événement : ",$tab[2],"<br />";
}
else echo "La chaîne \"$ch\" n'est pas conforme au modèle \"$modele\" ";
?>
```

L'exemple affiche le résultat suivant :

```
La chaîne "1515 Bataille de Marignan" est conforme au modèle "/([[:digit:]]{4})(.*)/"
Date: 1515
Événement: Prise de la Bastille
```

Remplacement de chaîne :
string preg_replace (string $modele, string $remp, string $ch)

$modele : le motif (pattern) qui va servir au remplacement
$remp : le motif de remplacement
$ch : chaîne à remplacer

$ch="ceci est la phrase 12";
$modele="/[0-9]{2}/";
$ch2[]=preg_replace($modele, 30+1, $ch);
var_dump($ch2) ;

```
array(2) {
  [0]=>
  string(20) "ceci est la phrase 1"
  [1]=>
  string(21) "ceci est la phrase 31"
}
```

SQL

---

expression REGEXP 'motif'

Négation de l’opérateur LIKE
Retourne TRUE si le motif de l’expression régulière est trouvé dans la valeur d’une colonne.
Par exemple, le code suivant sélectionne tous les noms dont le prénom commence par la lettre V suivie d’une voyelle puis d’un nombre quelconque de caractères :

```
SELECT nom FROM client WHERE prenom REGEXP 'V[aeiouy].*'
```

---

expression **NOT** REGEXP 'motif'

Négation de l’opérateur REGEXP