Les expressions rationnelles

Que sont les expressions régulières ?

Une expression régulière ou expression rationnelle est une chaîne de caractères, qui décrit, en utilisant une syntaxe précise, un ensemble de chaînes de caractères possibles.

Elles résultent d'un travail important de plusieurs chercheurs en mathématiques dans les années 1940 et 1950 et ont abouti à l'obtention d'un prix `Turing`.

Ces expressions régulières sont très utilisées en informatique et vous les retrouvez dans tous les langages, la plupart des éditeurs de code (dont `VS code`) et dans tous les terminaux `Unix` (avec des commandes comme `grep` ou `sed` que nous apprendrons dans une autre formation).

Elles permettent de facilement rechercher, substituer ou ajouter des chaînes de caractères.

Vous les utiliserez dès que vous aurez à manipuler des quantités importantes de texte ou de données, quel que soit le langage (`JavaScript, HTML` etc), la langue ou le format utilisé.

Pour vous donner quelques exemples d'utilisation : elles se retrouvent largement dans les filtres anti-spam, le traitement automatique du langage naturel (`machine learning`), les bots conversationnels, les robots des moteurs de recherche et plein d'autres domaines.

Les expressions régulières sont longues à apprendre, aussi nous vous conseillons de passer brièvement la leçon pour le moment et d'y revenir lorsque vous en aurez besoin.

Nous vous recommandons ce site pour tester vos expressions régulières, n'oubliez pas de sélectionner `JavaScript`.

Ce qui est intéressant c'est qu'une fois que vous connaissez les expressions régulières vous pourrez les utiliser dans tous les langages : en `Java, Python, PHP, C` etc. La syntaxe est commune avec des différences extrêmement minimes.

L'objet `JavaScript RegExp`

En `JavaScript` les expressions régulières sont des objets `RegExp`.

Elles se déclarent de trois façons, soit en utilisant **le constructeur avec le mot clé `new`** (que nous étudierons en détails plus tard dans la formation), soit **en utilisant `//`** (déclaration littérale), soit enfin en utilisant simplement un objet `RegExp`.

```
const regex1 = /.+/;
const regex2 = new RegExp('\w+');
const regex3 = RegExp('\w+');
```

Il faut utiliser la notation littérale dans les boucles, ou lorsqu'elle est constante dans votre contexte.

En effet, la notation littérale est évaluée une seule fois par l'interpréteur, alors que si vous utilisez le constructeur l'interpréteur recompilera l'expression à chaque itération de la boucle.

Les méthodes utilisables sur des chaînes de caractères

Nous allons commencer par les méthodes utilisables sur les chaînes de caractères et auxquelles nous passons une expression régulière.

Ces méthodes sont toutes sur l'objet `String`, mais nous les voyons maintenant car elles s'utilisent avec des expressions régulières.

Obtenir la première ou toutes les correspondances avec `match()`

La méthode `match()` permet de récupérer la première ou toutes les correspondances (avec le marqueur `g` que nous étudierons plus tard) d'une expression régulière avec une chaîne de caractères.

```
'abc'.match(/a/); // ["a", index: 0, input: "abc", groups: undefined]
'abc arge'.match(/a/g); // ["a", "a"]
```

Si le marqueur `g` est utilisé, tous les résultats correspondants à l'expression rationnelle seront retournés mais pas les groupes capturants.

Si le marqueur `g` n'est pas utilisé, seule la première correspondance et ses groupes capturants seront renvoyés.

Nous verrons les groupes capturants en détails dans la leçon. Retenez simplement que la propriété `groups` est utilisée pour les groupes capturants nommés.

Obtenir toutes les correspondances et les groupes capturants avec `matchAll()`

La méthode `matchAll()` retourne un **itérateur** contenant toutes les correspondances entre une chaîne de caractères et une expression rationnelle.

La première différence avec `match()` et le marqueur `g` est **qu'elle retourne également les groupes capturants**.

La seconde différence est que toutes les correspondances retournent la position de la correspondance (avec la propriété `index`) et la chaîne de caractères en entrée (avec la propriété `input`).

C'est un peu avancé car il s'agit d'un itérateur mais pas d'un tableau. C'est à dire qu'il est possible d'itérer dessus mais que ce n'est pas un tableau de l'objet `Array`.

Vous pourrez y revenir plus tard, mais cette méthode est très puissante :

```
const regexp = RegExp('test.','g');
const chaine = 'Une chaîne pour tester et effectuer des tests';
const matches = chaine.matchAll(regexp);

for (const match of matches) {
  console.log(match);
}
// ["teste", index: 16, input: "Une chaîne pour tester et effectuer des tests", groups: undefined]
// ["tests", index: 40, input: "Une chaîne pour tester et effectuer des tests", groups: undefined]
```

Remplacer les correspondances à une expression régulière par une chaîne de caractères avec `replace()`

La méthode `replace()` permet d'utiliser une expression régulière pour remplacer toutes les correspondances par une chaîne de caractères.

Elle peut aussi utiliser une simple chaîne de caractères mais c'est souvent moins utilisé.

Par exemple :

```
'Mon numéro de téléphone est 0142441222'.replace(/\d/g, '*');
// Mon numéro de téléphone est **********

'Il y a  un espace en trop'.replace(' ', ''); // Il y a un espace en trop
```

Trouver la première occurrence d'une correspondance à une expression régulière dans une chaîne de caractères avec `search()`

La méthode `search` permet de trouver la position de la première occurrence d'une correspondance à une expression régulière dans une chaîne de caractères.

Si aucune occurrence est n'est trouvée elle retourne `-1`.

Elle peut être utilise quand vous recherchez un motif mais dans les autres cas il faut mieux utiliser la méthode `includes()`.

```
'Mon numéro de téléphone est 0142441222'.search(/tél/); // 14
```

Séparer une chaîne de caractères en utilisant une expression régulière avec `split()`

La méthode `split()` permet de diviser une chaîne de caractères comme nous l'avons vu. Elle permet d'utiliser une expression régulière comme séparateur également :

```
'Paul Durant ; Bob Sinclar ; Jean Dupont'.split(/ ;|$ /);
// ["Paul Durant", " Bob Sinclar", " Jean Dupont"]
```

Les méthodes utilisables sur des `RegExp`

D'autres méthodes sont sur l'objet `RegExp` et sont donc utilisables non pas sur les chaînes de caractères mais sur les `RegeExp` .

Rechercher les correspondances à une expression régulière dans une chaîne de caractères avec la méthode `exec()`

La méthode `exec()` permet de rechercher les correspondances à une expression régulière dans une chaîne de caractères.

Si il n'y a pas de correspondance la méthode renvoie `null` .

En utilisant le marqueur `g` on peut utiliser la méthode `exec()` plusieurs fois afin de trouver les correspondances successives dans la chaîne.

Voici un exemple :

```
const regexp = RegExp('test.','g');
const chaine = 'Une chaîne pour tester et effectuer des tests';
const match1 = regexp.exec(chaine);
// ["teste", index: 16, input: "Une chaîne pour tester et effectuer des tests", groups: undefined]
const match2 = regexp.exec(chaine);
// ["tests", index: 40, input: "Une chaîne pour tester et effectuer des tests", groups: undefined]
```

Si on souhaite obtenir toutes les correspondances on utilisera plutôt la méthode `matchAll()` vue précedemment.

Vérifier si il y a au moins une correspondance entre une expression régulière et une chaîne de caractères avec `test()`

La méthode `test()` permet de vérifier si il y a au moins une correspondance entre une expression régulière et une chaîne de caractères. Elle retourne un booléen suivant le résultat du test.

```
/test/.test('Une chaîne pour tester et effectuer des tests');
// true
```

```
/test/.test('Une chaîne qui ne contient pas le motif recherché')
// false
```

La syntaxe des expressions régulières : caractères littéraux et classes

Nous allons commencer par la syntaxe pour apprendre à écrire des expressions régulières.

Les caractères littéraux

Commençons par le plus simple, les caractères littéraux sont extrêmement simple :

```
/a/.test('abc'); // true
RegExp(12).test(8768912); // true
```

Les classes prédéfinies pour les chiffres : `\d` et `\D`

Il existe certains caractères spéciaux, qui sont échappées pour les distinguer des caractères normaux, et qui sont des raccourcis permettant de créer des expressions régulières plus rapidement.

`\d` signifie n'importe quel chiffre entre `0 et 9`.

`\D` signifie n'importe quel caractère qui n'est pas un chiffre.

Exemples :

```
RegExp('\d').test('fezezf2'); // true
/\d/.test(4); // true
/\d/.test('bonjour'); // false

/\D/.test('bonjour'); // true
/\D/.test(4); // false
```

Les classes prédéfinies pour les mots : `\w` et `\W`

Il existe une classe plus large, `\w` (pour `word`) qui va correspondre à tous les chiffres entre 0 et 9, et aussi toutes les lettres entre a et z (en minuscule ET en majuscule) et l'`underscore (_)`.

Attention cependant, elle n'inclue pas tous les caractères non utilisés en anglais comme `éèçù….`.

L'inverse est `\W`.

```
/\w/.test(4); // true
/\w/.test('é'); // false
```

```
/\w/.test('!'); // false

/\W/.test('!'); // true
```

Les classes prédéfinies pour les espaces : `\s` et `\S`

Il existe également une classe pour les espaces, **`\s`** (pour `space`) qui va correspondre à tous les types d'espace, de tabulations et de sauts de lignes :

```
/\s/.test(' '); // true
RegExp('\s').test('\n'); // true
RegExp('\s').test('hello'); // false

/\S/.test('!'); // true
```

N'importe quel caractère avec `.`

Le caractère `.` est le caractère indiquant **tous les caractères sauf les sauts de ligne**.

Il faut activer le marqueur **`s`** pour qu'il devienne tous les caractères y compris les sauts de ligne :

```
/./.test('!'); // true
/./.test('1'); // true
/./.test(' '); // true
/./.test('\n'); // false
/./s.test('\n'); // true
```

Comment faire un point littéral dans ce cas ?

Il suffit de l'échapper : `\.` :

```
/\./s.test('.'); // true
```

Construire ses classes

Il est possible de définir exactement quels caractères vous souhaitez faire correspondre en utilisant les crochets `[ ]`.

Par exemple, `[a1;]` signifie match avec un seul caractère qui peut être un `a`, un `1` ou un `;` :

```
/[a1;]/.test('.'); // false
/[a1;]/.test('2'); // false
```

```
/[a1;]/.test(';'); // true
```

Inversement si vous voulez avoir n'importe quel caractères sauf ceux spécifiés il faut utiliser les crochets avec un `^` : `[^ ]`.

Par exemple, `[^a1;]` signifie match avec un seul caractère qui peut être n'importe quel caractère sauf un `a`, un `1` ou un `;` :

```
/[^a1;]/.test('.'); // true
/[^a1;]/.test('2'); // true
/[^a1;]/.test(';'); // false
```

Enfin, vous pouvez utiliser des intervalles permettant de matcher tous les caractères entre deux caractères spécifiés avec **[-]**.

A noter que les correspondances sont inclusives des deux limites :

```
/[c-g]/.test('g'); // true
/[2-8]/.test('1'); // false
/[2-8]/.test('4'); // true
```

Vous pouvez combiner plusieurs intervalles de caractères.

Ainsi par exemple, `[A-Za-z0-9_]` équivaut à `\w`.

`\d` équivaut à `[0-9]`.

`\D` équivaut à `[^0-9]`.

Vous pouvez également utiliser les raccourcis dans vos classes :

Par exemple `[\da-z]` :

```
/[\da-z]/.test('4'); // true
/[\da-z]/.test('b'); // true
/[\da-z]/.test('Z'); // false
/[\da-z]/.test(';'); // false
```

La syntaxe des expressions régulières : quantificateurs et alternative

Nous avons vu comment matcher un caractère en utilisant des intervalles de correspondances.

Nous allons voir maintenant comment quantifier le nombre de répétitions des caractères que nous souhaitons inclures dans nos correspondances.

Pour le moment si nous voulons avoir trois chiffres d'affilé nous avons à faire : `\d\d\d`. N'existe t-il pas une meilleure syntaxe ? C'est ce que nous allons apprendre.

Quantification exacte avec les accolades `{}`

Avec les `{ }` vous pouvez spécifier exactement le nombre de répétitions que vous souhaitez.

Par exemple `{2, 3}` signifie deux ou trois répétitions de l'expression précédente spécifiée :

```
/\d{2,3}/.test('1'); // false
/\d{2,3}/.test('12'); // true
/\d{2,3}/.test('123'); // true
/\d{2,3}/.test('1234'); // true
/\d{2,3}/.test('12a4'); // true
/\d{2,3}/.test('1aa4'); // false
```

Notez que `1234` match car nous avons au moins 3 chiffres successifs.

Pour signifier que vous voulez avoir le nombre exact de répétition vous pouvez utiliser un seul chiffre : `{3}`.

Si vous voulez avoir minimum x répétitions sans maximum il vous faudra utiliser `{2, }`.

Exemple :

```
/\d{2,}/.test('1aa4'); // false
/\d{2,}/.test('14123'); // true
'14123'.match(/\d{2,}/);
  // ["14123", index: 0, input: "14123", groups: undefined]
```

Quantification 1 ou plus avec `+`

Il existe un raccourci pour `{1, }` qui est souvent utile : **+**. Il signifie au moins une fois l'expression spécifiée.

Exemples :

```
'aabc'.match(/a+/);
// ["aa", index: 0, input: "aabc", groups: undefined]
'bc'.match(/a+/); //null
```

Quantification 0 ou plus avec `*`

Il existe un raccourci pour `{0, }` qui est également très utilisé : **`*`**. Il signifie zéro ou plus de fois l'expression spécifiée.

Rendre une correspondance optionnelle avec `?`

Parfois vous voudrez rendre un caractère ou un intervalle optionnel, il faudra utiliser `?` :

```
les chats'.match(/les? chats?/);
// ["les chats", index: 0, input: "les chats", groups: undefined]
'le chat'.match(/les? chats?/);
// ["le chat", index: 0, input: "le chat", groups: undefined]
```

Utiliser une alternative avec `|`

L'alternative correspond au OU logique. Je veux tel caractère ou tel autre :

```
/a|b|cdef/.test('adef'); // true
/a|b|cdef/.test('bdef'); // true
/a|b|cdef/.test('def'); // false
/a|b|cdef/.test('gdef'); // false
```

Rendre un quantificateur paresseux avec `?`

Par défaut, tous les quantificateurs sont gourmands (`greedy`), ils vont matcher le maximum de caractères possibles en respectant votre expression.

Cependant, dans certains cas, vous voudrez qu'ils matchent le minimum de caractères tout en respectant votre expression :

```
'abc'.match(/a\w+?/);
// ["ab", index: 0, input: "abc", groups: undefined]
'abc'.match(/a\w+/);
// ["abc", index: 0, input: "abc", groups: undefined]
```

La syntaxe des expressions régulières : les ancres

Les ancres permettent de spécifier le début ou la fin. Cela peut être le début ou la fin d'une ligne ou d'un text. Cela peut également être la fin d'un mot.

Début ou fin d'un texte avec `^` et `$`

Pour marquer le début d'une entrée il faut utiliser `^` et pour marquer la fin il faut utiliser `$`.

```
'Il était une fois...
 abc
 the end.'.match(/^Il était une fois.* the end\.$/s);
// 0: Il était une fois... \n abc \n the end.
```

Ici nous sommes sûr que le texte commence exactement par `Il était une fois` (grâce à `^`), suivi d'un nombre indéfinis de n'importe quel caractère (notez l'utilisation du marqueur `s` pour les nouvelles lignes).

Nous sommes ensuite certain que le texte finit exactement par `the end.` grâce à `$`.

Début ou fin d'une ligne avec `^` et `$` avec le marqueur `m`

Nous pouvons utiliser le marqueur `m` qui va changer le comportement de `^` et `$`. Ils ne marqueront plus la fin du texte d'entrée mais le début et la fin des lignes.

```
'Du texte de préambule qui ne sera pas inclus.
Il était une fois.....
 the end.'.match(/^Il était une fois.* the end\.$/sm);
// 0: "Il était une fois..... \n the end."
```

Le début ou la fin d'un mot avec `\b`

Le caractère spécial `\b` pour `word boundary` permet de marquer le début ou la fin d'un mot. Il n'est pas très souvent utilisé.

Par exemple `\be` signifie : match tous les caractères `e` qui se situent en début de mot.

Et `e\b` signifie : match tous les caractères `e` qui se situent en fin de mot.

Les groupes

Une fonctionnalité extrêmement puissante des `RegExp` sont les groupes. Ils permettent notamment d'extraire des caractères pour pouvoir les manipuler.

Capturer des groupes avec `()`

Les groupes sont extrêmement utiles pour récupérer par exemple une extension ou un nom de fichier :

```
'mon_super_fichier.pdf'.match(/(^\w+)\.([a-z]{3}$)/);
// 0: "mon_super_fichier.pdf"
// 1: "mon_super_fichier"
```

```
// 2: "pdf"
// index: 0
// input: "mon_super_fichier.pdf"
// groups: undefined
// length: 3
```

Nous allons étudier cet exemple en détails.

Etape 1 : Vérifie déjà que `^\w+\.[a-z]{3}$` a au moins une correspondance. C'est-à-dire vérifie que nous avons un ou plus caractère contenu dans l'intervalle `[a-zA-Z0-9_]` et qui commence le texte en entrée, puis un point littéral, puis exactement trois lettres minuscules.

Etape 2 : `(^\w+)` signifie capture le groupe suivant : un ou plus caractère contenu dans l'intervalle `[a-zA-Z0-9_]` et qui commence le texte en entrée.

Etape 3 : `([a-z]{3}$)` signifie capture le groupe suivant : exactements trois lettres contenu dans l'intervalle `[a-z]` et qui termine le texte en entrée.

La méthode `match()` retourne ensuite à l'`index 0` la correspondance globale de l'étape 1, à l'`index 1` le groupe capturé à l'étape 2, à l'`index 2` le groupe capturé à l'étape 3.

La propriété `index` est la position de la première correspondance.

La propriété `groups` contient les groupes nommés (nous les verrons juste après).

La propriété `input` contient le texte en entrée.

La propriété `length` est la correspondance plus le nombre de groupes donc ici 3.

Capturer des groupes nommés avec `(?nom)`

Il est possible également de nommer ses groupes de capture ce qui permet un traitement beaucoup plus simples car nous pouvons accéder aux groupes par propriété et non plus par `index`.

```
'mon_super_fichier.pdf'.match(/(?<nomFichier>^\w+)\.(?<extension>[a-z]{3}$)/);
// 0: "mon_super_fichier.pdf"
// 1: "mon_super_fichier"
// 2: "pdf"
// index: 0
// input: "mon_super_fichier.pdf"
// groups:
//   nomFichier: "mon_super_fichier"
//   extension: "pdf"
```

Vous pouvez ainsi faire :

```
const match = 'mon_super_fichier.pdf'.match(/(?<nomFichier>^\w+)\.(?<extension>[a-z]{3}$)/);

match.groups.extension; // "pdf"
match.groups.nomFichier; // "mon_super_fichier"
```

Les sous-groupes

Vous pouvez imbriquer les groupes de capture les uns dans les autres si vous avez besoin d'extraire plusieurs informations à partir des mêmes caractères.

Prenons un exemple :

```
'09/12/2019 à 12:57:47'.match(/(?<date>(?<jour>\d{2})\/\d{2}\/(?<annee>\d{4})) à (?
<heure>\d{2}:\d{2}:\d{2})/);
// 0: "09/12/2019 à 12:57:47"
// 1: "09/12/2019"
// 2: "09"
// 3: "2019"
// 4: "12:57:47"
// index: 0
// input: "09/12/2019 à 12:57:47"
// groups:
//   date: "09/12/2019"
//   jour: "09"
//   annee: "2019"
//   heure: "12:57:47"
```

C'est à ce genre de chose que vous voyez que les expressions régulières sont extrêmements puissantes. Il vous faudrait plusieurs de ligne pour obtenir un résultat équivalent sans expression régulière, et la performance serait bien moins importante.

Délimiter les alternatives avec les groupes

Prenons un exemple :

```
'bonjour tout le monde'.match(/hello|bonjour|salut tout le monde/);
// 0: bonjour
```

Ce n'est pas ce que nous voulons ! Nous voulions récupérer `xxx tout le monde`. Mais l'alternative est entre `hello` ou `bonjour` ou `salut tout le monde`.

Nous souhaitons `hello` ou `bonjour` ou `salut` et `tout le monde`.

Dans ce cas il faut utiliser un groupe pour correctement limiter les alternatives :

```
'bonjour tout le monde'.match(/(hello|bonjour|salut) tout le monde/);
// 0: "bonjour tout le monde"
// 1: "bonjour"
...
```

Les groupes non capturants `(?:)`

Lorsque vous utilisez les groupes pour délimiter les alternatifs ou les quantificateurs vous ne souhaitez pas forcément les récupérer dans vos groupes.

Il est possible d'utiliser un groupe non capturant de ce cas avec `(?:)`.

Ainsi en reprenant notre exemple :

```
'bonjour tout le monde'.match(/(?:hello|bonjour|salut) tout le monde/);
// 0: "bonjour tout le monde"
…
```

Nous n'avons plus le groupe en `index 1`, il n'est pas venu polluer nos groupes capturés.

Les marqueurs

Les marqueurs sont des paramètres pour l'expression rationnelle. Ils peuvent tous s'utiliser en même temps car ils portent sur des caractéristiques différentes.

La seule exception est le marqueur `y` et le marqueur `g` qui sont incompatibles, le marqueur `g` sera ignoré.

Recherche globale avec le marqueur `g`

Une recherche globale permet de rechercher toutes les occurrences correspondants à l'expression régulière plutôt que de s'arrêter à la première.

```
const regex1 = /.+/g;
const regex2 = new RegExp('\w+', 'g');
const regex3 = RegExp('\w+', 'g');
```

Ignorer la casse avec le marqueur `i`

Le marqueur `i` permet de rendre la recherche non sensible à la casse (`insensitive case`), cela permet de faire abstraction des majuscules et minuscules.

```
const regex1 = /.+/i;
const regex2 = new RegExp('\w+', 'i');
const regex3 = RegExp('\w+', 'i');
const regex3 = RegExp('a+', 'gi');
```

Les marqueurs n'ont pas d'ordre, si vous en utilisez plusieurs vous pouvez les mettre dans l'ordre que vous voulez : `gi` ou `ig` reviendront exactement au même.

Recherche multiligne avec `m`

Le marqueur `m` permet d'agir sur les caractères de début `^` et de fin `$`.

Ils permettent d'être considérés comme **marquant le début et la fin des lignes** et non pas le **début et la fin du texte d'entrée** sans le marqueur.

Activer l'`Unicode` avec `u`

Le marqueur `u` permet de dire à l'interpréteur `JavaScript` d'activer l'`Unicode`. Vous pourrez ainsi rechercher des séquences de points de code.

Nous verrons des exemples plus tard.

Activer l'adhérence avec `y`

Le marqueur `y` permet d'utiliser la propriété `lastIndex` dans une expression régulière. Il permet de ne rechercher si il y a une correspondance à la position précisée par la propriété `lastIndex`.

```
const chaine = ("bonjour à tous !");
const regex = /à.+/y;
regex.lastIndex = 8;
regex.test(chaine); // true
regex.lastIndex = 2;
regex.test(chaine); // false
regex.lastIndex = 10;
regex.test(chaine); // false
```

Activer le `dotAll` avec `s`

Comme nous l'avons vu le caractère `.` permet de matcher tous les caractères sauf les sauts de ligne. En activant le marqueur `s`, les `.` correspondent à tous les caractères **y compris les sauts de ligne** (les

quatre types de saut de ligne : `line feed`, `carriage return`, `line separator` et `paragraph separator`).

Avec `s` le `.` correspondra à tous les caractères encodés sur une unité de code `UTF-16`.

Si vous voulez qu'il corresponde également aux caractères encodés sur deux unités de code `UTF-16` il est nécessaire d'activer également le marqueur `u`.

Regarder avant ou après (`lookaround`)

Une fonctionnalité très puissante des expressions régulières est la possibilité de vérifier qu'une expression est présente ou absente, avant ou après l'expression visée.

Vérifier qu'une expression est trouvée dans la suite du texte (`positive lookahead`)

Pour vérifier qu'une expression existe dans la suite du texte mais que nous ne voulons pas matcher il faut utiliser `(?=)`.

```
/\d+(?=%)/.exec('100% des présidents français et américains ont été des hommes');
// 0: "100"
...
```

Ici nous voulons récupérer un ou plusieurs chiffres suivis par un `%`.

C'est très puissant pour par exemple vérifier que l'utilisateur a entré au moins une majuscule, une minuscule, un chiffre et un caractère spécial :

```
/(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[\W_])/.test('hello'); // false
/(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[\W_])/.test('hE'); // false
/(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[\W_])/.test('hE1'); // false
/(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[\W_])/.test('hE1!'); // true
```

Vous pouvez jouer un peu pour par exemple ajouter la condition, le mot de passe doit faire au moins 10 caractères :

```
/(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[\W_]).{10,}/.test('hE1!'); // false
/(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[\W_]).{10,}/.test('hE1!frezzfe'); // true
```

Vérifier qu'une expression n'est pas trouvée dans la suite du texte (`negative lookahead`)

Pour vérifier qu'une expression n'existe pas dans la suite du texte mais que nous ne voulons pas matcher il faut utiliser `(?!)`.

```
/(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[\W_])(?!.*\s).{10,}/.test('hE1!frezzfe'); // true
/(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?=.*[\W_])(?!.*\s).{10,}/.test('hE1!fre zzfe'); // false
```

Nous avons ajouté la vérification que le mot de passe ne contient pas de caractères de type espace (tabulation, saut de ligne, espace etc) avec `?!.*\s`

Vérifier qu'une expression est trouvée avant une expression (`positive lookbehind`)

Pour vérifier qu'une expression existe avant une expression il faut utiliser `(?<=)`.

Par exemple :

```
/(?<=\$)\d+/.test('Cela coûte $100'); // true
/(?<=\$)\d+/.test('Cela coûte 100$'); // false
```

Vérifier qu'une expression n'est pas trouvée avant une expression (`negative lookbehind`)

Pour vérifier qu'une expression n'existe pas avant une expression il faut utiliser `(?<!)`.

```
/(?<!\$)\d{3}/.test('Cela coûte $100'); // false
/(?<!\$)\d{3}/.test('Cela coûte 100$'); // true
```

Utilisation des propriétés `Unicode`

Cette fonctionnalité est extrêmement puissante pour les langages comme le français qui ont des caractères comme `ç é è à ï ù É` etc.

Elle est aussi puissante si vous voulez par exemple extraire le sens en utilisant les emojis (colère, joie, rire etc). Bref, les cas d'utilisation sont très nombreux.

Nous allons voir plusieurs exemples en lien avec notre magnifique langue française :

```
/\w+/.test('ééééé !'); // false
/\p{L}/u.test('ééééé !'); // true
```

Les propriétés `Unicode` permettent de cibler de grands ensembles de caractères, nous allons en voir les principaux ensemble :

`\p{L}` signifie toutes les lettres, y compris les lettres accentués etc.

`\p{Ll}` signifie toutes les lettres minuscules, y compris les lettres accentués etc.

`\p{Lu}` signifie toutes les lettres majuscules, y compris les lettres accentués etc.

`\p{Number}` signifie tous les nombres y compris décimaux, y compris les chiffres romains, les fractions etc.

`\p{Ps}` signifie toutes les ponctuations d'ouverture de crochets (`{ [ (` etc).

`\p{Pe}` signifie toutes les ponctuations de fermeture de crochets.

`\p{Sc}` signifie tous les symboles de monnaie (`$ €` etc).

`\p{Sm}` signifie tous les symboles mathématiques.

`\p{Emoji}` signifie tous les emojis.

`\p{Script=Greek}` signifie tous les caractères grecs.

Il y en a vraiment énormément et vous pouvez les rechercher en tapant `unicode character property`.

```
/\p{Emoji}/u.test('😁'); // true
/^\p{Script=Greek}+$/u.test('μετά');
/^\p{Number}+$/u.test('²³¹¼½¾123456789𝟎𝟏𝟐𝟑456⑪⑫⑬IIIIIIIVVVIVIIVIIIIXXXIXIIILCDMiiiiiiivvviviiviiiixxxixiilcdm'); // true
```

Les substitutions

Nous allons enfin voir les substitutions qui permettent une utilisation très puissante de la méthode `replace()`.

Il existe quelques symboles à connaître avant d'étudier les exemples :

`$&` permet d'insérer toute la correspondance avec l'expression régulière à l'endroit spécifié.

`$n` permet d'insérer le groupe capturé `n` (où `n` est un nombre à 1 ou 2 chiffres) par l'expression régulière à l'endroit spécifié.

`$<nom>` permet d'insérer le groupe nommé `nom` par l'expression régulière à l'endroit spécifié.

`$'` permet d'insérer la partie du texte d'entrée avant la correspondance.

`$$` permet d'insérer le caractère littéral `$`.

Prenons un exemple pour insérer le caractère littéral `$` juste à l'endroit où une correspondance avec l'expression régulière est trouvée :

```
'Je veux garder ma phrase mais ajouter le signe dollar, et ce après le chiffre 42. Je veux garder le reste aussi'.replace(/(\d+)/, '$&$$');
```

```
// Je veux garder ma phrase mais ajouter le signe dollar, et ce après le chiffre 42$. Je veux garder le reste aussi
```

Autre exemple, où l'objectif est d'inverser un prénom et un nom et d'ajouter une virgule entre les deux :

```
'Paul DUPONT'.replace(/(\w+) (\w+)/i, '$2, $1'); // DUPONT, Paul
```

Autre exemple avec un groupe capturé :

```
'Je veux extraire et ajouter un préfixe à mon nom de fichier mon_fichier.pdf'.replace(/(.*?)(?<fichier>\w+\.pdf)/, '123_$<fichier>');
// 123_mon_fichier.pdf
```

Nous utilisons `?` après `.*` pour rendre le quantificateur paresseux. Nous voulons en effet que le second groupe capturé l'emporte sur le premier pour la partie du nom du fichier (vous pouvez essayer sans pour observer la différence).

Nous utilisons un groupe capturé pour ne pas garder la partie précédent le nom de fichier. Ensuite, nous utilisons un groupe nommé pour capturer le nom du fichier `pdf`.

Enfin, nous insérons le préfixe souhaité et nous accolons le nom du fichier en utilisant la substitution de groupe nommé avec `$<nom`.

Quelques exemples de `RegExp` utiles

Nous allons prendre quelques exemples de `RegExp` que vous rencontrerez souvent.

Expression régulière pour les `URLs`

Voici une expression régulière possible pour les `URLs` utilisant des caractères latins (ne fonctionnera pas avec des `URL` contenant des caractères chinois ou arabes par exemple) :

```
https?:\/\/(www\.)?[-\w.]{2,253}\.[a-z]{1,15}\b[-\w()\[\]@!$&'*:%+.,;~#?&\/=]*
```

`http` : nous voulons avoir le début de l'`URL` qui commence toujours par `http`.

`s?` : l'`URL` peut être sécurisée (`https`) ou non. Le `s` est donc optionnel.

`\/\/` : nous devons échapper les deux `/` car il s'agit de caractères spéciaux dans les `regex`.

`(www\.)?` nous utilisons un groupe uniquement pour rendre l'ensemble de l'expression optionnelle. En effet, le `www.` peut être présent ou absent dans une `URL`.

`[-\w.]{2,253}` : le nom de domaine des `URL` peut contenir jusqu'à 253 caractères. Il peut contenir des `.` pour les sous-domaines, des tirets, des chiffres et des lettres minuscules (mais les lettres majuscules seront converties automatiquement par les navigateurs donc l'`URL` fonctionnerait). Il permettra d'inclure les domaines de deuxième niveau également (par exemple `.aeroport.fr, .co.uk`).

`\.[a-z]{1,13}\b` : l'extension finale (`TLD`) peut faire entre `0` et `13` caractères (`.international`) pour le moment. Nous avons ensuite forcément une `word boundary` qui peut être un `?` si il y a une partie `query`.

`[-\w()\[\]@!$&'*:%+.,;~#?&//=]*` est la partie chemin ou requête. Elle peut comporter tous les caractères listés exactement. Les autres caractères sont encodés avec `%`. Elle peut également commencer par un port, par exemple `:3000`.

Cette expression régulière n'est pas parfaite, c'est nous qui l'avons faites et pas 10 ingénieurs de `Google`, mais elle est suffisamment robuste pour la plupart des cas d'utilisation.

Expression régulière pour les `IPv4`

Prenons un second exemple d'expression rationnelle simple pour les `IPv4` :

```
^(?:\d{1,3}\.){3}\d{1,3}$
```

`^(?:\d{1,3}\.){3}` : nous avons exactement 3 fois (`{3}`), un à trois chiffres suivi d'un point littéral (`\.`). Nous utilisons un groupe non capturant pour ne pas polluer nos groupes et pouvoir utiliser un quantificateur (ici `{3}`).

`\d{1,3}$` : nous finissons par (`$`) un à trois chiffres (`\d{1,3}`).