Les expressions régulières (partie 2/2)

Dans ce deuxième chapitre consacré aux regex, nous allons voir leur utilisation au sein du JavaScript. En effet, le premier chapitre n'était là que pour enseigner la base de la syntaxe alors que, une fois couplées à un langage de programmation, les regex deviennent très utiles. En JavaScript, elles utilisent l'objet `RegExp` et permettent de faire tout ce que l'on attend d'une regex : rechercher un terme, le capturer, le remplacer, etc.

Construire une regex

Le gros de la théorie sur les regex est maintenant vu, et il ne reste plus qu'un peu de pratique. Nous allons tout de même voir comment écrire une regex pas à pas, de façon à ne pas se tromper.
Nous allons partir d'un exemple simple : vérifier si une chaîne de caractères correspond à une adresse e-mail. Pour rappel, une adresse e-mail est de cette forme : `monadresse@email.com`.

Une adresse e-mail contient trois parties distinctes :

- La partie locale, avant l'arobase (ici « monadresse ») ;
- L'arobase @ ;
- Le domaine, lui-même composé du label « email » et de l'extension « com ».

Pour construire une regex, il suffit de procéder par étapes : faisons comme si nous lisions la chaîne de caractères et écrivons la regex au fur et à mesure. On écrit tout d'abord la partie locale, qui n'est composée que de lettres, de chiffres et éventuellement d'un tiret, un trait de soulignement et un point. Tous ces caractères peuvent être répétés plus d'une fois (il faut donc utiliser le quantificateur +) :

```
/^[a-z0-9._-]+$/
```

On ajoute l'arobase. Ce n'est pas un métacaractère, donc pas besoin de l'échapper :

```
/^[a-z0-9._-]+@$/
```

Après vient le label du nom de domaine, lui aussi composé de lettres, de chiffres, de tirets et de traits de soulignement. Ne pas oublier le point, car il peut s'agir d'un sous-domaine (par exemple @email.email.com) :

```
/^[a-z0-9._-]+@[a-z0-9._-]+$/
```

Puis vient le point de l'extension du domaine : attention à ne pas oublier de l'échapper, car il s'agit d'un métacaractère :

```
/^[a-z0-9._-]+@[a-z0-9._-]+\.$/
```

Et pour finir, l'extension ! Une extension de nom de domaine ne contient que des lettres, au minimum 2, au maximum 6. Ce qui nous fait :

```
/^[a-z0-9._-]+@[a-z0-9._-]+\.[a-z]{2,6}$/
```

Testons donc :

```
const email = prompt("Entrez votre adresse e-mail :",
"javascript@sitedemoncours.com");
if (/^[a-z0-9._-]+@[a-z0-9._-]+\.[a-z]{2,6}$/.test(email)) {
    alert("Adresse e-mail valide !");
} else {
    alert("Adresse e-mail invalide !");
}
```

L'adresse e-mail est détectée comme étant valide !

L'objet RegExp

L'objet `RegExp` est l'objet qui gère les expressions régulières. Il y a donc deux façons de déclarer une regex : via `RegExp` ou via son type primitif que nous avons utilisé jusqu'à présent :

```
const myRegex1 = /^Raclette$/i;
const myRegex2 = new RegExp("^Raclette$", « i");
```

Le constructeur `RegExp` reçoit deux paramètres : le premier est l'expression régulière sous la forme d'une chaîne de caractères, et le deuxième est l'option de recherche, ici `i`. L'intérêt d'utiliser `RegExp` est qu'il est possible d'inclure des variables dans la regex, chose impossible en passant par le type primitif :

```
const nickname = "Sébastien";
const myRegex = new RegExp("Mon prénom est " + nickname, « i");
```

Ce n'est pas spécialement fréquent, mais cela peut se révéler particulièrement utile. Il est cependant conseillé d'utiliser la notation littérale (le type primitif) quand l'utilisation du constructeur `RegExp` n'est pas nécessaire.

Méthodes

`RegExp` ne possède que deux méthodes : `test()`et `exec()`. La méthode `test()` a déjà été utilisée et permet de tester une expression régulière ; elle renvoie `true` si le test est réussi ou `false` si le test échoue. De son côté, `exec()`applique également une expression régulière, mais renvoie un tableau dont le premier élément contient la portion de texte trouvée dans la chaîne de caractères. Si rien n'est trouvé , `null` est renvoyé.

```
const sentence = "Si ton tonton";
const result = /\bton\b/.exec(sentence); // On cherche à récupérer
le mot « ton »
if (result) { // On vérifie que ce n'est pas null
    alert(result); // Affiche « ton »
}
```

Propriétés

L'objet `RegExp` contient neuf propriétés, appelées `$1`, `$2`, `$3`... jusqu'à `$9`. Comme nous allons le voir dans la sous-partie suivante, il est possible d'utiliser une regex pour extraire des portions de texte, et ces portions sont accessibles via les propriétés `$1` à `$9`.

Tout cela va être mis en lumière un peu plus loin en parlant des parenthèses !

Les parenthèses

Les parenthèses capturantes

Nous avons vu pour le moment que les regex servaient à voir si une chaîne de caractères correspondait à un modèle. Mais il y a moyen de faire mieux, comme extraire des informations. Pour définir les informations à extraire, on utilise des parenthèses, que l'on appelle **parenthèses capturantes**, car leur utilité est de capturer une portion de texte, que la regex va extraire.

Considérons cette chaîne de caractères : « Je suis né en mars ». Au moyen de parenthèses capturantes, nous allons extraire le mois de la naissance, pour pouvoir le réutiliser :

```
const birth = 'Je suis né en mars';
/^Je suis né en (\S+)$/.exec(birth);
alert(RegExp.$1); // Affiche : « mars »
```

Cet exemple est un peu déroutant, mais est en réalité assez simple à comprendre. Dans un premier temps, on crée la regex avec les fameuses parenthèses. Comme les mois sont faits de caractères qui peuvent être accentués, on peut directement utiliser le type générique \S. \S+ indique qu'on recherche une série de caractères, jusqu'à la fin de la chaîne (délimitée, pour rappel, par $) : ce sera le mois. On englobe ce « mois » dans des parenthèses pour faire comprendre à l'interpréteur JavaScript que leur contenu devra être extrait.

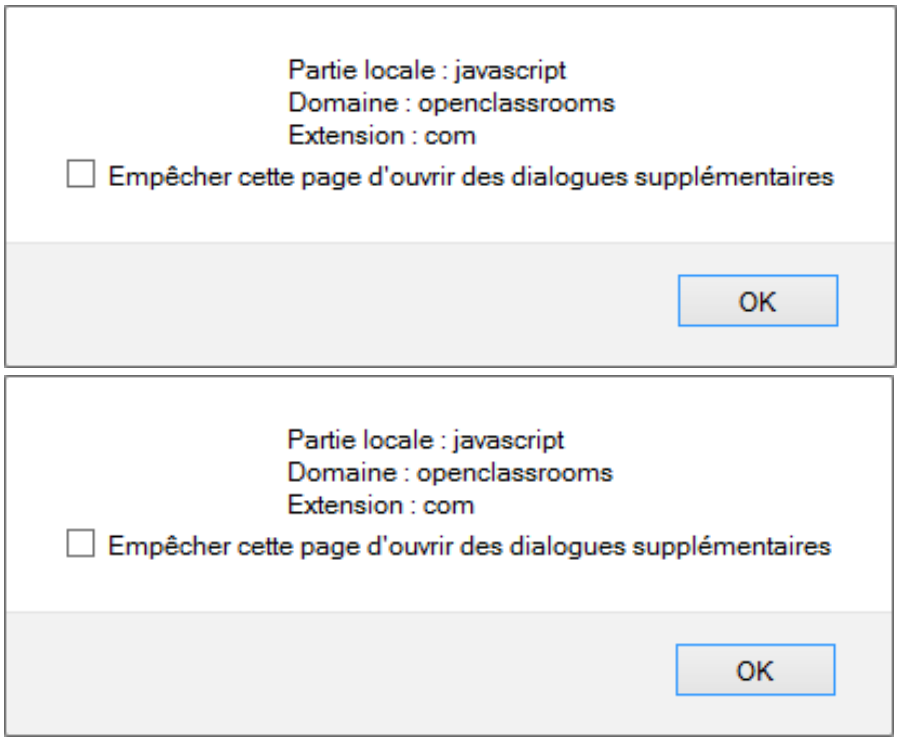
La regex est exécutée via `exec()`. Et ici une autre explication s'impose. Quand on exécute `test()` ou `exec()`, le contenu des parenthèses capturantes est

enregistré temporairement au sein de l'objet `RegExp`. Le premier couple de parenthèses sera enregistré dans la propriété `$1`, le deuxième dans`$2`et ainsi de suite, jusqu'au neuvième, dans `$9`. Cela veut donc dire qu'il ne peut y avoir qu'un maximum de neuf couples de parenthèses. Les couples sont numérotés suivant le sens de lecture, de gauche à droite.
Et pour accéder aux propriétés, il suffit de faire `RegExp.$1`, `RegExp.$2`, etc.

Voici un autre exemple, reprenant la regex de validation de l'adresse e-mail. Ici, le but est de décomposer l'adresse pour récupérer les différentes parties :

```
const email = prompt("Entrez votre adresse e-mail :",
"javascript@siteduzero.com");
if (/^([a-z0-9._-]+)@([a-z0-9._-]+)\.([a-z]{2,6})$/.test(email)) {
    alert('Partie locale : ' + RegExp.$1 + '\nDomaine : ' +
RegExp.$2 + '\nExtension : ' + RegExp.$3);
} else {
    alert('Adresse e-mail invalide !');
}
// TESTER CE CODE
```

Ce qui nous affiche bien les trois parties :



Les trois parties sont bien renvoyées

Remarquez que même si `test()` et `exec()` sont exécutés au sein d'un `if()` le contenu des parenthèses est quand même enregistré. Pas de changement de ce côté-là puisque ces deux méthodes sont quand même exécutées au sein de la condition.

Les parenthèses non capturantes

Il se peut que dans de longues et complexes regex, il y ait besoin d'utiliser beaucoup de parenthèses, plus de neuf par exemple, ce qui peut poser problème puisqu'il ne peut y avoir que neuf parenthèses capturantes exploitables. Mais toutes ces parenthèses n'ont peut-être pas besoin de capturer quelque chose, elles peuvent juste être là pour proposer un choix. Par exemple, si on vérifie une URL, on peut commencer la regex comme ceci :

```
/(https|http|ftp|steam):\/\//
```

Mais on n'a pas besoin que ce soit une parenthèse capturante et qu'elle soit accessible via `RegExp.$1`. Pour la rendre non capturante, on va ajouter ?: au début de la parenthèse, comme ceci :

```
/(?:https|http|ftp|steam):\/\//
```

De cette manière, cette parenthèse n'aura aucune incidence sur les propriétés `$` de `RegExp` !

Les recherches non-greedy

Le mot anglais greedy signifie « gourmand ». En JavaScript, les regex sont généralement gourmandes, ce qui veut dire que lorsqu'on utilise un quantificateur comme le +, le maximum de caractères est recherché, alors que ce n'est pas toujours le comportement espéré. Petite mise en lumière : nous allons construire une regex qui va extraire l'adresse Web à partir de cette portion de HTML sous forme de chaîne de caractères :

```
const html = '<a href="www.mon-adresse.be">Mon site</a>';
```

Voici la regex qui peut être construite :

```
/<a href=« (.+)">/
```

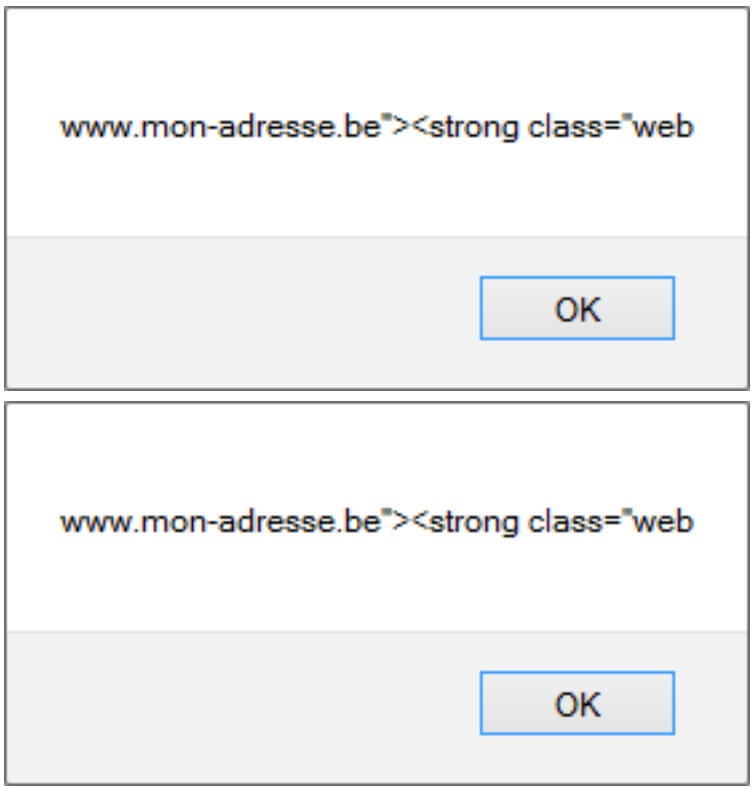
Et ça marche :

```
/<a href="(.+)">/.exec(html);
alert(RegExp.$1); // www.mon-adresse.be
```

Maintenant, supposons que la chaîne de caractères ressemble à ceci :

```
const html = '<a href="www.mon-adresse.be"><strong class="web">Mon site</strong></a>';
```

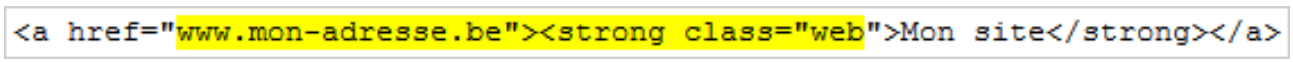
Et là, c'est le drame :



La valeur renvoyée n'est pas celle qu'on attendait

En spécifiant `.+` comme quantificateur, on demande de rechercher le plus possible de caractères jusqu'à rencontrer les caractères « "> », et c'est ce que le JavaScript fait :

```
<a href="www.mon-adresse.be"><strong class="web">Mon site</strong></a>
```



JavaScript s'arrête à la dernière occurrence souhaitée

Le JavaScript va trouver la partie surlignée : il cherche jusqu'à ce qu'il tombe sur la dernière apparition des caractères « "> ». Mais ce n'est pas dramatique, fort heureusement !

Pour pallier ce problème, nous allons écrire le quantificateur directement suivi du point d'interrogation, comme ceci :

```
const html = '<a href="www.mon-adresse.be"><strong class="web">Mon site</strong></a>';
/<a href="(.+?)">/.exec(html);
alert(RegExp.$1);
```

Le point d'interrogation va faire en sorte que la recherche soit moins gourmande et s'arrête une fois que le minimum requis est trouvé, d'où l'appellation non-greedy (« non gourmande »).

Rechercher et remplacer

Une fonctionnalité intéressante des regex est de pouvoir effectuer des « rechercher-remplacer ». Rechercher-remplacer signifie qu'on recherche des portions de texte dans une chaîne de caractères et qu'on remplace ces portions par d'autres. C'est relativement pratique pour modifier une chaîne rapidement, ou pour convertir des données. Une utilisation fréquente est la conversion de balises BBCode en HTML pour prévisualiser le contenu d'une zone de texte.

Un rechercher-remplacer se fait au moyen de la méthode `replace()` de l'objet `String`. Elle reçoit deux arguments : la regex et une chaîne de caractères qui sera le texte de remplacement. Petit exemple :

```
const sentence = 'Je m\'appelle Sébastien';
const result = sentence.replace(/Sébastien/, 'Johann');
alert(result); // Affiche : « Je m'appelle Johann »
```

Très simple : `replace()` va rechercher le prénom « Sébastien » et le remplacer par « Johann ».

Utilisation de `replace()` sans regex

À la place d'une regex, il est aussi possible de fournir une simple chaîne de caractères. C'est utile pour remplacer un mot ou un groupe de mots, mais ce n'est pas une utilisation fréquente, on utilisera généralement une regex. Voici toutefois un exemple :

```
const result = 'Je m\'appelle Sébastien'.replace('Sébastien',
'Johann');
alert(result); // Affiche : « Je m'appelle Johann »
```

L'option `g`

Nous avions vu l'option `i` qui permet aux regex d'être insensibles à la casse des caractères. Il existe une autre option, `g`, qui signifie « rechercher plusieurs fois ». Par défaut, la regex donnée précédemment ne sera exécutée qu'une fois : dès que « Sébastien » sera trouvé, il sera remplacé… et puis c'est tout. Donc si le prénom « Sébastien » est présent deux fois, seul le premier sera remplacé. Pour éviter ça, on utilisera l'option `g` qui va dire de continuer la recherche jusqu'à ce que plus rien ne soit trouvé :

```
const sentence = 'Il s\'appelle Sébastien. Sébastien écrit un
tutoriel.';
const result = sentence.replace(/Sébastien/g, 'Johann');
alert(result); // Il s'appelle Johann. Johann écrit un tutoriel.
```

Ainsi, toutes les occurrences de « Sébastien » sont correctement remplacées par « Johann ». Le mot occurrence est nouveau ici, et il est maintenant temps de l'employer. À chaque fois que la regex trouve la portion de texte qu'elle recherche, on parle d'occurrence. Dans le code précédent, deux occurrences de

« Sébastien » sont trouvées : une juste après « appelle » et l'autre après le premier point.

Rechercher et capturer

Il est possible d'utiliser les parenthèses capturantes pour extraire des informations et les réutiliser au sein de la chaîne de remplacement. Par exemple, nous avons une date au format américain : 05/26/2011, et nous souhaitons la convertir au format jour/mois/année. Rien de plus simple :

```
let date = '05/26/2011';
date = date.replace(/^(\d{2})\/(\d{2})\/(\d{4})$/, 'Le $2/$1/$3');
alert(date); // Le 26/05/2011
```

Chaque nombre est capturé avec une parenthèse, et pour récupérer chaque parenthèse, il suffit d'utiliser`$1`,`$2`et`$3`(directement dans la chaîne de caractères), exactement comme nous l'aurions fait avec `RegExp.$1`.

Et si on veut juste remplacer un caractère par le signe dollar, il faut l'échapper ? Pour placer un simple caractère $ dans la chaîne de remplacement, il suffit de le doubler, comme ceci :

```
const total = 'J\'ai 25 dollars en liquide.';
alert(total.replace(/dollars?/, '$$'); // J'ai 25 $ en liquide
```

Le mot « dollars » est effectivement remplacé par son symbole. Un point d'interrogation a été placé après le « s » pour pouvoir trouver soit « dollars » soit « dollar ».

Voici un autre exemple illustrant ce principe. L'idée ici est de convertir une balise BBCode de mise en gras (`[b]un peu de texte[/b]`) en un formatage HTML de ce type :`<strong>un peu de texte</strong>`. N'oubliez pas d'échapper les crochets qui sont, pour rappel, des métacaractères !

```
const text = 'bla bla [b]un peu de texte[/b] bla [b]bla bla en
gras[/b] bla bla';
text = text.replace(/\[b\]([\s\S]*?)\[\/b\]/g, '<strong>$1</
strong>');
alert(text);
```

Mais pourquoi avoir utilisé `[\s\S]` et non pas juste le point ?
Il s'agit ici de trouver tous les caractères qui se trouvent entre les balises. Or, le point ne trouve que des caractères et des espaces blanc hormis le retour à la ligne. C'est la raison pour laquelle on utilisera souvent la classe comprenant \s et \S quand il s'agira de trouver du texte comportant des retours à la ligne.

Cette petite regex de remplacement est la base d'un système de prévisualisation du BBCode. Il suffit d'écrire une regex de ce type pour chaque balise, et le tour est joué :

```
<script>
    function preview() {
        const value = document.getElementById("text").value;
        value = value.replace(/\[b\]([\s\S]*?)\[\/b\]/g,
'<strong>$1</strong>'); // Gras
        value = value.replace(/\[i\]([\s\S]*?)\[\/i\]/g, '<em>$1</
em>'); // Italique
        value = value.replace(/\[s\]([\s\S]*?)\[\/s\]/g,
'<del>$1</del>'); // Barré
        value = value.replace(/\[u\]([\s\S]*?)\[\/u\]/g, '<span
style="text-decoration: underline">$1</span>'); // Souligné
        document.getElementById("output").innerHTML = value;
    }
</script>
<form>
    <textarea id="text"></textarea><br />
    <button type="button" onclick="preview()">Prévisualiser</
button>
    <div id="output"></div>
</form>
// TESTER CE CODE
```

Utiliser une fonction pour le remplacement

À la place d'une chaîne de caractères, il est possible d'utiliser une fonction pour gérer le ou les remplacements. Cela permet, par exemple, de réaliser des opérations sur les portions capturées (`$1`, `$2`, `$3`…).

Les paramètres de la fonction sont soumis à une petite règle, car ils doivent respecter un certain ordre (mais leurs noms peuvent bien évidemment varier) : `function(str, p1, p2, p3 /* ... */, offset, s)`. Les paramètres `p1`, `p2`, `p3`… représentent les fameux `$1`, `$2`, `$3`… S'il n'y a que trois parenthèses capturantes, il n'y aura que trois « p ». S'il y en a cinq, il y aura cinq « p ». Voici les explications des paramètres :

- Le paramètre `str` contient la portion de texte trouvée par la regex ;
- Les paramètres `p*` contiennent les portions capturées par les parenthèses ;
- Le paramètre `offset` contient la position de la portion de texte trouvée ;
- Le paramètre `s` contient la totalité de la chaîne.

Si on n'a besoin que de `p1` et de `p2` et pas des deux derniers paramètres, ces deux derniers peuvent être omis.
Pour illustrer cela, nous allons réaliser un petit script tout simple, qui recherchera des nombres dans une chaîne et les transformera en toutes lettres. La transformation se fera au moyen de la fonction `num2Letters()` qui a été codée lors du TP vu durant la semaine 2 de votre formation frontend : Convertir un nombre en toutes lettres.

```
const sentence = 'Dans 22 jours, j\'aurai 24 ans';
let result = sentence.replace(/(\d+)/g, function(str, p1) {
    p1 = parseInt(p1);
    if (!isNaN(p1)) {
        return num2Letters(p1);
    }
});
alert(result); // Affiche : « Dans vingt-deux jours, j'aurai vingt-quatre ans »
```

L'exemple utilise une fonction anonyme, mais il est bien évidemment possible de déclarer une fonction :

```
function convertNumbers(str) {
    str = parseInt(str);
    if (!isNaN(str)) {
        return num2Letters(str);
    }
}
const sentence = 'Dans 22 jours, j\'aurai 24 ans';
const result = sentence.replace(/(\d+)/g, convertNumbers);
```

Autres recherches

Il reste deux méthodes de `String` à voir, `search()` et `match()`, plus un petit retour sur la méthode `split()`.

Rechercher la position d'une occurrence

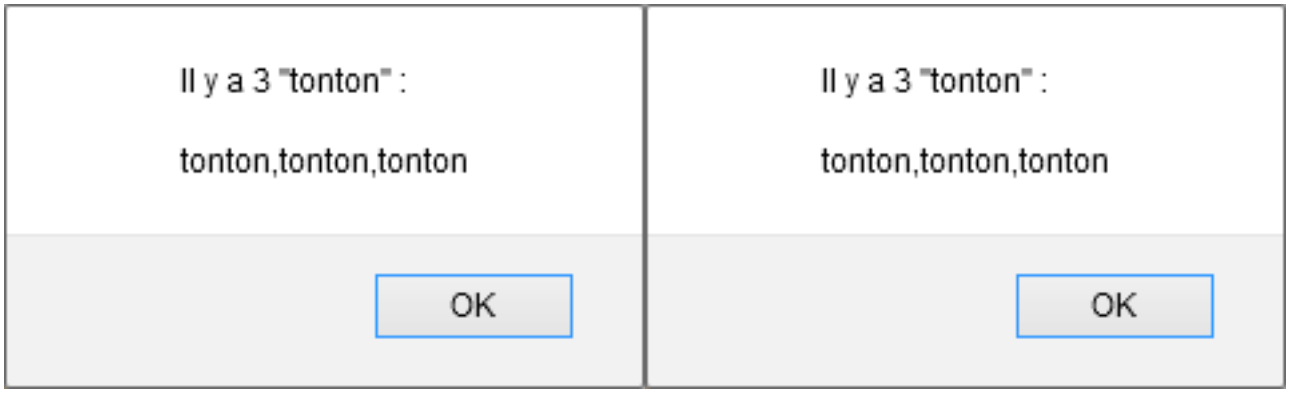
La méthode `search()`, toujours de l'objet `String`, ressemble à `indexOf()` mis à part le fait que le paramètre est une expression régulière .`search()`retourne la position de la première occurrence trouvée. Si aucune occurrence n'est trouvée, -1 est retourné. Exactement comme `indexOf()`:

```
const sentence = 'Si ton tonton';
const result = sentence.search(/\bton\b/);
if (result > -1) { // On vérifie que quelque chose a été trouvé
    alert('La position est ' + result); // 3
}
```

Récupérer toutes les occurrences

La méthode `match()` de l'objet `String` fonctionne comme `search()`, à la différence qu'elle retourne un tableau de toutes les occurrences trouvées. C'est pratique pour compter le nombre de fois qu'une portion de texte est présente par exemple :

```
const sentence = 'Si ton tonton tond ton tonton, ton tonton tondu sera tondu';
const result = sentence.match(/\btonton\b/g);
alert('Il y a ' + result.length + ' "tonton" :\n\n' + result);
```



Il y a trois occurrences de « tonton »

Couper avec une regex

Nous avions vu que la méthode `split()` recevait une chaîne de caractères en paramètre. Mais il est également possible de transmettre une regex. C'est très pratique pour découper une chaîne à l'aide, par exemple, de plusieurs caractères distincts :

```
const family = 'Guillaume,Pauline;Clarisse:Arnaud;Benoît;Maxime';
const result = family.split(/[,:;]/);
alert(result);
```

L'`alert()` affiche donc un tableau contenant tous les prénoms, car il a été demandé à `split()` de couper la chaîne dès qu'une virgule, un deux-points ou un point-virgule est rencontré.

En résumé

- Construire une regex se fait rarement du premier coup. Il faut y aller par étapes, morceau par morceau, car la syntaxe devient vite compliquée.
- En combinant les parenthèses capturantes et la méthode `exec()`, il est possible d'extraire des informations.
- Les recherches doivent se faire en mode non-greedy. C'est plus rapide et correspond plus au comportement que l'on attend.
- L'option `g` indique qu'il faut effectuer plusieurs remplacements, et non pas un seul.
- Il est possible d'utiliser une fonction pour la réalisation d'un remplacement. Ce n'est utile que quand il est nécessaire de faire des opérations en même temps que le remplacement.
- La méthode `search()` s'utilise comme la méthode `indexOf()`, sauf que le paramètre est une regex.

[source - https://openclassrooms.com/fr/courses/1916641-dynamisez-vos-sites-web-avec-javascript/1920284-les-expressions-regulieres-partie-2-2]