

Javascript Premiers pas

Support de cours Version 1.6

Sommaire

Sommaire	2
1 Introduction.....	5
2 Premiers pas avec JS.....	7
2.1 Avec quels outils démarrer ?.....	7
2.2 Emplacement du code Javascript	8
2.2.1 La console JS	8
2.2.2 Inclusion du JS dans la page HTML	12
2.3 Notion de variable	15
2.4 Créer du HTML avec du JS	18
2.4.1 La technique du « rouleau compresseur ».....	19
2.4.2 La technique « de l'amateur éclairé »	24
2.4.3 La technique « full JS »	30
2.4.4 Générer un tableau HTML 4 colonnes.....	34
2.5 Créer du JS avec du HTML	38
2.5.1 la méthode « getElementByTagName »	39
2.5.2 Affinage des méthodes de sélection	43
2.5.3 Le sélecteur « couteau suisse »	45
2.6 Notion de fonctions.....	50
2.6.1 Généralités	50
2.6.1 Le mode « strict » est ton ami	54
2.6.3 Les variables sont musclées	57
2.7 Vidage d'un élément	60
2.8 Un petit tour dans les « events »	62
2.9 Filtre dynamique	67
2.10 Téléportation.....	74
3 Conclusion	82
4 Annexe.....	84
4.1 Téléportation (source complet)	84
4.2 Tableau 4 colonnes version 2	86

4.3 Où est le démarreur ?.....	88
4.4 Filtre dynamique (variante encore plus dynamique)	90
5 Changelog	93

Notes de l'auteur :

Je m'appelle Grégory Jarrige.

Je suis développeur professionnel depuis 1991. Après avoir longtemps travaillé sur des gros systèmes et des langages et technos propriétaires, j'ai fait le pari de me former aux technos et langage open source vers 2005-2006. J'ai commencé à développer des applications webs professionnelles à partir de 2007, avant d'en faire mon activité principale à partir de 2010. L'arrivée du HTML5 dans la même période a été pour moi une véritable bénédiction, et surtout un formidable terrain d'expérimentation (avec des API comme Canvas, WebAudio, etc...).

En plus de mon activité de développeur freelance, je suis également formateur - sur des sujets tels que PHP, HTML5, Javascript, SQL – tantôt en entreprise, tantôt dans le cadre de programmes de reconversion (GRETA notamment).

Ce support est une création réalisée en juin 2017 en vue de proposer une introduction rapide au langage Javascript pour des personnes débutant dans ce langage, mais ayant déjà des bases d'algorithmique, et/ou des bases de programmation dans d'autres langages (PHP ou autres...). On retrouvera dans ce support certaines techniques Javascript que j'avais utilisées lors d'ateliers d'initiation donnés dans le cadre du meetup « Creative Coding Paris » : <https://www.meetup.com/fr-FR/CreativeCodeParis>

Le présent document est publié sous Licence Creative Commons n° 6.

Il est disponible en téléchargement libre sur mon compte Github :

<https://github.com/gregja/JSCorner>

Ce document se situe dans la continuité des supports « HTML5 – Premiers pas » et « CSS3 – Premiers pas » qui sont disponibles dans le même dépôt Github.

1 Introduction

Javascript est un vieux langage, si l'on considère qu'il est apparu au milieu des années 90, à peu près en même temps que PHP et Ruby.

Le vrai nom de Javascript, c'est ECMAScript. C'est en réalité le nom de la norme sur laquelle le langage Javascript est bâti.

Javascript a été créé dans l'urgence par un ingénieur talentueux - Brendan Eich - pour répondre à un besoin très précis qui était de faciliter la communication d'applets Java avec l'écosystème des navigateurs de l'époque (et en particulier du défunt Netscape Navigator). Le nom de Javascript vient de là, de cette lointaine filiation avec Java. Si Javascript emprunte, comme Java, certaines caractéristiques syntaxiques au langage C, Java et Javascript présentent plus de différences que de similitudes :

- Java est un langage compilé, Javascript est un langage interprété,
- Java est un langage fortement typé, Javascript est faiblement typé,
- Dans le discours marketing des concepteurs de Java, ce langage a longtemps été présenté comme le langage à tout faire, capable de s'exécuter partout via le principe de la JVM (Java Virtual Machine). Mais c'est surtout en tant que langage serveur que Java a acquis ses lettres de noblesse. Javascript a été très longtemps cantonné au développement web « front » (côté navigateur), mais il est sorti de son périmètre initial par l'intermédiaire du projet NodeJS, pour devenir l'un des langages les plus polyvalents qui soit. Aujourd'hui, « Javascript is everywhere... » 😊

Bref, Javascript n'est pas Java, qu'on se le dise... Cet amalgame est d'autant plus préjudiciable, que de nombreux recruteurs encore aujourd'hui sont incapables de faire le distinguo entre les 2 langages.

Javascript est une compétence très recherchée sur le marché du travail aujourd'hui, mais c'est d'abord et avant tout – en ce qui me concerne – le langage le plus fun et le plus passionnant que je connaisse. On peut tout lui faire faire, du son (grâce à l'API Webaudio, disponible dans tout bon navigateur), du graphisme 2D et 3D, animé ou non (avec les APIs SVG, Canvas ou WebGL), de la réalité virtuelle (avec la toute nouvelle API WebVR), etc...

En rédigeant cette introduction au Javascript, j'ai décidé de faire l'impasse sur de nombreux sujets, et de me focaliser sur certains aspects du langage qui me semblent essentiels, et qui sont souvent méconnus, ou simplement mal compris par les développeurs. Or ces aspects sont

essentiels pour écrire du code robuste et pérenne. Apprendre à maîtriser ces techniques, c'est la garantie de pouvoir se sortir de toutes les situations, bref d'être un développeur bien dans ses baskets.

J'espère avec ce modeste tuto vous communiquer un peu de la passion que j'éprouve pour cet incroyable langage qu'est Javascript.

Dans la suite de ce tuto, j'emploierai le plus souvent le sigle JS pour désigner le langage Javascript.

2 Premiers pas avec JS

2.1 Avec quels outils démarrer ?

Nous allons utiliser dans un premier temps un navigateur : je vous recommande d'utiliser Firefox ou Chrome, selon votre préférence. Si vous êtes sur Mac, vous pouvez aussi utiliser Safari.

Dans mes supports d'introduction dédiés à HTML5 et à P5.js (disponibles dans le même projet Github), j'ai beaucoup évoqué les plateformes Codepen et Codecircle. Ces deux plateformes sont géniales, et si vous êtes à l'aise avec, vous pouvez tout à fait les utiliser pour suivre ce tuto :

- Codepen : <https://codepen.io/>
- Codecircle : <https://live.codecircle.com/>

Mais j'aimerais vous montrer que vous pouvez faire énormément de choses à l'intérieur même de votre navigateur préféré, sans utiliser le moindre outil extérieur.

Nous aurons cependant besoin, dans la suite de ce cours, d'une page HTML de base, la plus simple possible, histoire de démarrer notre apprentissage sur de bonnes bases. Le plus simple, pour obtenir cette page HTML, c'est d'en créer une. Voici son code source, vous pouvez le copier-coller dans un éditeur quelconque, en prenant soin de le sauvegarder avec l'extension « .html ». Vous pouvez par exemple l'appeler « mapage.html » :

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>ma page de test</title>
</head>
<body>
  <div id="cible">ceci est le contenu d'une balise "div" de test</div>
</body>
</html>
```

Si vous ne comprenez pas certains éléments du code ci-dessus, je vous recommande d'étudier le cours « HTML5 – Premiers pas », qui est disponible dans le même projet Github.

2.2 Emplacement du code Javascript

Pour démarrer cette initiation au Javascript, je vous propose d'utiliser dans un premier temps la console du navigateur, c'est ce que nous allons faire dans le chapitre qui suit.

Mais une question se pose très rapidement à l'étudiant qui étudie le Javascript, une fois qu'il a compris les principes de la console, et qu'il souhaite passer à la vitesse supérieure, c'est : « où placer le code Javascript dans la page HTML ? ». C'est ce que nous allons voir dans le chapitre d'après.

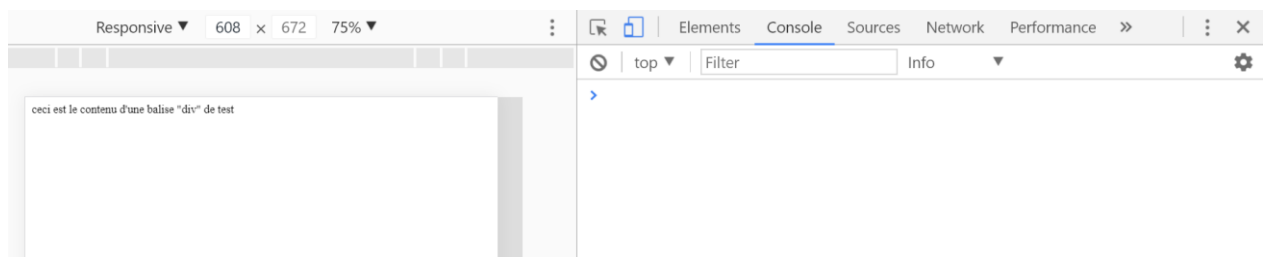
2.2.1 La console JS

Double-cliquez sur votre fichier « mapage.html », vous devriez voir votre navigateur préféré se lancer (Chrome ou Firefox, en fonction de vos préférences système). Et si tout s'est bien passé, vous devriez voir apparaître le message suivant dans votre navigateur préféré :

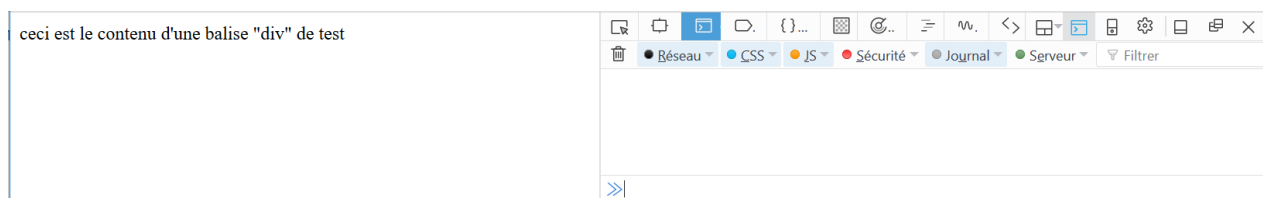
ceci est le contenu d'une balise "div" de test

Pressez maintenant la touche F12 pour accéder aux outils de développement de votre navigateur.

Dans Chrome, vous devriez voir une fenêtre ressemblant à ceci :



Dans Firefox, vous devriez voir une fenêtre ressemblant à cela :



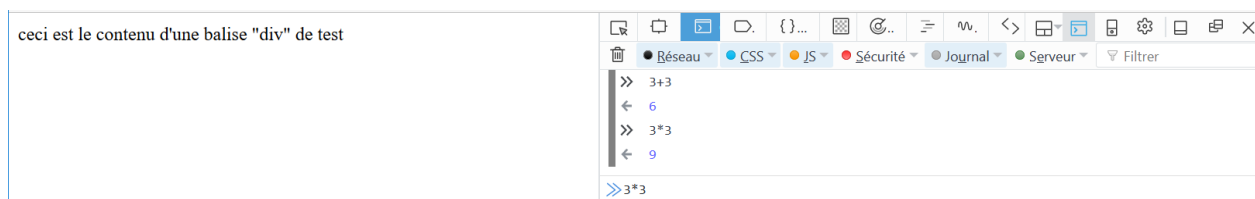
Ce qui nous intéresse ici, c'est la console JS. Cette console va nous permettre de saisir et

d'exécuter des commandes Javascript en « live ».

Vous remarquerez que la console JS n'est pas placée de la même façon dans Chrome et dans Firefox. Pour ma part, je trouve que la disposition de la console est plus pratique dans Chrome, mais c'est une affaire de goût personnel.

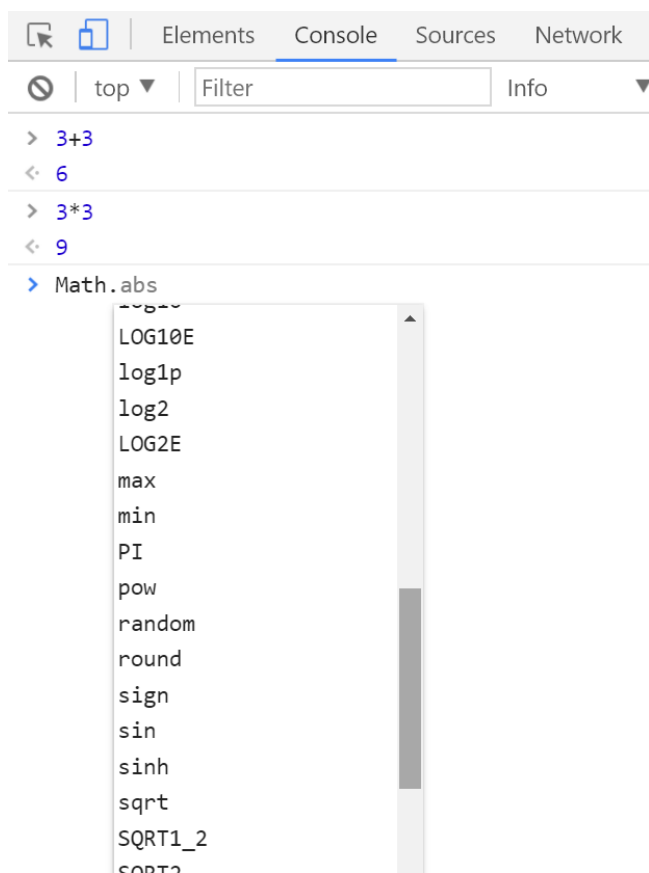
On peut utiliser la console JS comme une simple calculatrice.

Essayez de saisir quelques opérations simples comme dans l'exemple suivant :



Vous pouvez aussi effectuer des calculs scientifiques. Pour ce faire, JS met à votre disposition l'objet Math.

Si vous saisissez « Math » (attention à bien respecter la casse), le navigateur déclenche une autocomplétion et vous propose un certain nombre de propriétés (comme le nombre « PI ») et de méthodes (comme « sin », « cos », etc...) :

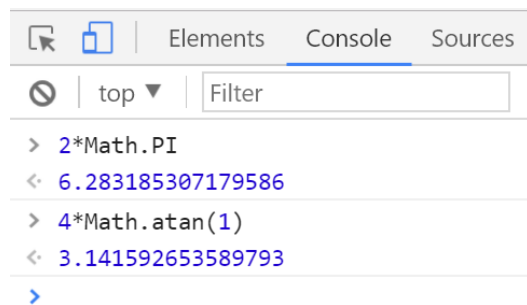


C'est quoi une propriété, c'est quoi une méthode ?

Disons pour simplifier que :

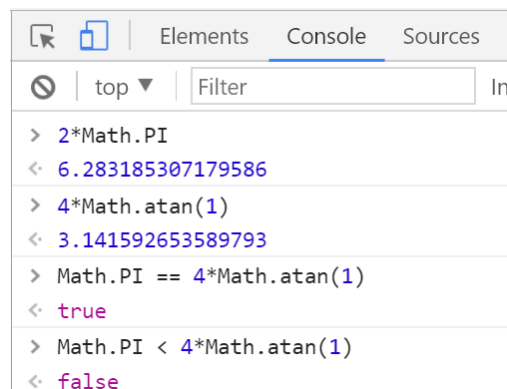
- Une propriété c'est une variable attachée à un objet. Par exemple PI est la propriété ou variable associée à l'objet Math, elle contient la valeur du fameux nombre Pi. Une propriété peut être de type texte, de type nombre, de type booléen, de type tableau, etc... et même de type objet.
- Une méthode c'est une fonction attachée à un objet. Par exemple, `sin()` et `cos()` sont deux méthodes ou fonctions associées à l'objet Math. Vous pourriez très bien avoir une configuration dans laquelle vous disposez d'une fonction `sin()` indépendante (peut être écrite par vous), et de la fonction `Math.sin()` fournie par le langage Javascript. Si vous êtes dans ce cas, la fonction `sin()` et la méthode `sin()` de l'objet Math n'ont strictement rien à voir.

Quelques exemples de calculs exploitant des méthodes et propriétés de l'objet Math :



```
> 2*Math.PI
< 6.283185307179586
> 4*Math.atan(1)
< 3.141592653589793
>
```

On peut aussi utiliser la console pour effectuer des comparaisons :



```
> 2*Math.PI
< 6.283185307179586
> 4*Math.atan(1)
< 3.141592653589793
> Math.PI == 4*Math.atan(1)
< true
> Math.PI < 4*Math.atan(1)
< false
```

En JS, le test d'égalité se fait en utilisant le symbole « `==` » (ou éventuellement « `===` », mais je ne souhaite pas m'étendre sur ces subtilités ici).

2.2.2 Inclusion du JS dans la page HTML

La console, c'est sympa à utiliser, et c'est même un précieux outil pour le développeur au quotidien. Mais quand on ferme la page du navigateur, on perd tout ce que l'on a saisi dans la console. Il convient donc de placer le code JS dans un emplacement permanent, et le meilleur endroit pour le faire, c'est la page HTML.

On a deux options possibles, utilisant toutes deux la balise « script » :

- Option 1 : Inclure le code JS directement à l'intérieur de la page, comme ceci :

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>Inclusion du JS directement dans la page</title>
</head>
<body>
<div id="cible">div de test</div>

<script>
  // emplacement du code JS
  // placer de préférence le bloc de code à la fin du "body"
  var a = 2 * Math.PI;
  console.log(a);
</script>

</body>
</html>
```

- Option 2 : placer le code JS dans un fichier externe (par exemple : le fichier « inclusion_code.js ») et utiliser la balise « script » pour définir un pointeur vers ce fichier externe

Le fichier « inclusion_code.js » :

```
// Code JS placé dans un fichier externe
// placer de préférence le bloc de code à la fin du "body"
var a = 2 * Math.PI;
console.log(a);
```

Le fichier HTML :

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>Inclusion du JS dans un fichier externe</title>
</head>
<body>
<div id="cible">div de test</div>

<script src="inclusion_code.js"></script>

</body>
</html>
```

Cette seconde méthode offre un avantage qui est que le même fichier JS peut être inclus dans plusieurs pages HTML. On place ainsi le code à un seul endroit, et si on est amené à le modifier, cela va se répercuter sur l'ensemble des pages.

On notera qu'il existe aussi une troisième méthode d'inclusion du JS dans le HTML, comme le montre l'exemple ci-dessous, dans lequel on a associé l'appel d'une fonction Javascript (la fonction « doSomething ») à un événement « click » (au moyen de la propriété « onclick ») :

```
<button onclick="doSomething()" id="action-btn">Cliquez moi !</button>
```

Cette solution a été longtemps pratiquée, mais n'est plus considérée à l'heure actuelle comme une bonne pratique : imaginez que l'utilisateur clique sur le bouton avant que le code Javascript ait été chargé dans le DOM. Dans ce cas de figure, la fonction « doSomething » n'ayant pas encore été chargée dans la mémoire du navigateur, le clic prématuré de l'utilisateur va déclencher une erreur grave (puisque la fonction n'existe pas encore), rendant l'ensemble de l'application inopérante (du moins côté navigateur). Il existe des solutions élégantes pour éviter de se retrouver dans cette situation, mais je ne vais pas les développer ici. Nous verrons un cas d'utilisation de la propriété « onclick » dans le chapitre 2.10 (« Téléportation »).

Maintenant que vous avez compris le principe, une question demeure : est-il préférable de placer le code JS à la fin du « body » - comme dans l'exemple ci-dessus – ou dans le « head », comme on le voit quelquefois. Car si vous avez pris le temps de regarder le code source de quelques sites internet, vous aurez sans doute remarqué que les 2 cas de figure se présentent souvent. Alors, qui a raison ?

En fait, pendant très longtemps, pratiquement tous les développeurs plaçaient leur code JS dans le « head » des pages, le plus souvent dans des fichiers externes. Mais une tendance inverse a émergé il y a quelques années, car les fichiers JS sont devenus de plus en plus gros. Or quand un navigateur charge un fichier JS, il se fige jusqu'à ce qu'il ait fini de charger et d'exécuter le code

contenu dans le fichier JS. L'effet est très désagréable pour les utilisateurs. Aussi pour éliminer cet effet désagréable, on recommande aujourd'hui de placer le code JS à la fin du « body ».

Il y a une autre raison pour laquelle on place son JS à la fin du « body ». Supposons que ce code JS modifie certains éléments de la page, et si ce code est placé dans le « head », alors il va être exécuté avant que le code HTML correspondant n'est pas encore en place, il va donc être exécuté trop tôt. Au mieux, ce code sera sans effet, au pire il va déclencher une erreur grave, rendant l'application inopérante.

A chaque problème, il existe une solution, particulièrement en Javascript, et les développeurs qui préfèrent malgré tout placer leur code JS dans le « head », peuvent le faire moyennant certains aménagements dans leur code. Mais si j'explique cela maintenant, je vais employer plein de gros mots, comme « écouteur d'événement », ou encore « DOM », c'est-à-dire des choses que nous n'avons pas encore étudiées, et que nous découvrirons au fil de l'eau. C'est la raison pour laquelle, vous trouverez quelques explications complémentaires en annexe, dans le chapitre 4.3 (« où est le démarreur ? »).

2.3 Notion de variable

Pour créer une variable en JS, on utilise le mot clé « var ».

Dans le chapitre précédent, nous avons effectué un test comparant deux valeurs :

```
> Math.PI == 4*Math.atan(1)
< true
```

Mais nous n'avons pas pris la peine de conserver le résultat de ce test dans une variable. Si nous avons besoin d'obtenir le résultat de ce test plus tard, nous serons obligés de réexécuter ce test. C'est dommage, alors que nous avons la possibilité de sauvegarder le résultat de ce test dans une variable.

Le bon développeur est un développeur paresseux. Il n'aime pas refaire plusieurs fois la même chose. Nous allons donc faire en sorte de conserver le résultat de notre test dans une variable, c'est facile à faire, ça s'écrit comme ceci :

```
> var test_pi = Math.PI == 4*Math.atan(1)
< undefined
```

Ma variable s'appelle « test_pi » et elle contient le résultat du test effectué précédemment. Si je traduis ça d'un point de vue algorithmique, je dirais que :

La variable « test_pi » reçoit le résultat du test de comparaison du nombre « PI » par rapport au calcul « 4 multiplié par l'arc tangente de 1 »

... ou en version plus abrégée :

$\text{test_pi} \leftarrow 3.14159... = 4 * \text{atan}(1) ?$

Comment puis-je vérifier le contenu de la variable « test_pi » ? Facile, en tapant son nom dans la console comme ceci :

```
> test_pi
< true
```

Quand j'ai effectué mon calcul tout à l'heure, j'ai vu apparaître le mot clé « undefined »... Késacko ?

```
> var test_pi = Math.PI == 4*Math.atan(1)
< undefined
```

Il s'agit d'un comportement normal du navigateur. En gros, il nous indique qu'il a bien fait ce qu'on lui demandait de faire, mais que le résultat de cette action (qui consistait à alimenter une variable) n'a produit aucun résultat directement affichable (d'où la valeur « undefined »).

Cela peut surprendre, mais si on met cela en perspective par rapport à ce qui se passait précédemment, quand on effectuait notre test sans le stocker dans une variable, cela devient plus logique :

```
> Math.PI == 4*Math.atan(1)
< true
```

Dans cet exemple ci-dessus, nous effectuons notre test mais nous ne le stockons pas dans une variable, le navigateur se trouve donc obligé de nous renvoyer l'information « en live », d'où la valeur « true ». Quand nous stockons ce résultat dans une variable, le navigateur n'a plus rien à nous afficher, d'où la valeur « undefined ».

En résumé, si le navigateur vous renvoie la valeur « undefined », ce n'est pas anormal, c'est même plutôt bon signe 😊.

Si l'on se replace du point de vue de notre variable « test_pi », il s'agit d'une variable « booléenne ». Elle peut contenir les valeurs « true » ou « false ».

JS met à notre disposition l'instruction « typeof » qui nous permet de connaître le type exact d'une variable, à un instant donné :

```
> typeof test_pi
< "boolean"
```


Tiens, par curiosité, si je demande à JS de me donner le type d'une variable « test_pi2 », que je n'ai pas encore créée, voici ce que j'obtiens :

```
> typeof test_pi2  
< "undefined"
```

L'instruction « typeof » nous renvoie donc « undefined »... OK, ça semble logique.

Si maintenant je crée cette variable « test_pi2 » avec la comparaison utilisant le symbole « inférieur » que nous avons évoquée précédemment ?

```
> var test_pi2 = Math.PI < 4*Math.atan(1);  
< undefined  
> typeof test_pi2;  
< "boolean"  
> test_pi2;  
< false
```

J'ai indiqué dans l'introduction que JS était un langage faiblement typé, cela signifie qu'une variable peut changer de type durant son cycle de vie. Par exemple, si je modifie la valeur de « test_pi2 » en lui donnant la valeur « null » (qui est une valeur reconnue par JS), alors le type de cette variable change :

```
> test_pi2 = null;  
< null  
> typeof test_pi2  
< "object"
```

Si maintenant, j'alimente la variable « test_pi2 » avec une chaîne de caractères, cela donne ceci :

```
> test_pi2 = 'le typage faible, c'est cool';  
< "le typage faible, c'est cool"  
> typeof test_pi2  
< "string"
```

Je rappelle qu'en anglais le type « chaîne de caractères » s'appelle le type « string ». Ce typage faible des données est une caractéristique que JS partage avec d'autres langages interprétés comme PHP. Cela peut apparaître comme une faiblesse, mais pour ma part je trouve

que cela procure beaucoup de souplesse... question de point de vue.

2.4 Créer du HTML avec du JS

Dans le cours « HTML5 – Premiers pas », nous avons étudié comment créer des listes HTML, et en particulier une liste de langages, que voici :

```
<ul>
  <li>java</li>
  <li>javascript</li>
  <li>scala</li>
  <li>haskell</li>
  <li>go</li>
  <li>rust</li>
  <li>julia</li>
  <li>basic</li>
  <li>pascal</li>
  <li>fortran</li>
  <li>c</li>
  <li>c++</li>
  <li>c#</li>
  <li>ruby</li>
  <li>python</li>
</ul>
```

D'un point visuel, le code HTML ci-dessus donne le résultat suivant dans un navigateur :

- java
- javascript
- scala
- haskell
- go
- rust
- julia
- basic
- pascal
- fortran
- c
- c++
- c#
- ruby
- python

Pourrait-on se servir du Javascript pour générer ce genre de liste ?

La réponse est bien évidemment « oui »

Nous allons étudier comme procéder, ce qui nous permettra d'introduire quelques notions nouvelles au fil de l'eau.

2.4.1 La technique du « rouleau compresseur »

Nous allons utiliser ici une méthode hyper basique qu'on n'utilise quasiment plus aujourd'hui, dans les applications professionnelles, mais qui est intéressante d'un point de vue pédagogique.

La méthode utilisée ici consiste à considérer que notre liste HTML est déjà faite, nous l'avons à notre disposition dans une variable de type « chaîne de caractères ». Cette liste pourrait nous avoir été envoyée par un serveur PHP via une technique qu'on appelle « AJAX » sur laquelle je ne veux pas m'étendre car c'est en dehors du sujet qui nous intéresse ici.

Pour les besoins de notre exercice, nous avons besoin de cette variable de type texte contenant notre liste au format HTML, et je ne me vois pas vous expliquer maintenant comment faire ça en AJAX, alors je vous propose de créer artificiellement cette variable avec le code suivant :

```
var lang_list = '<ul>' +  
  '<li>java</li>' +  
  '<li>javascript</li>' +  
  '<li>scala</li>' +  
  '<li>haskell</li>' +  
  '<li>go</li>' +  
  '<li>rust</li>' +  
  '<li>julia</li>' +  
  '<li>basic</li>' +  
  '<li>pascal</li>' +  
  '<li>fortran</li>' +  
  '<li>c</li>' +  
  '<li>c++</li>' +  
  '<li>c#</li>' +  
  '<li>ruby</li>' +  
  '<li>python</li>' +  
  '</ul>';
```

J'ai fait quoi en réalité ? Eh bien, je me suis contenté de reprendre le code HTML et j'ai encapsulé les différentes parties entre apostrophes, et ajouté des « plus » en bout de ligne. Ce « plus » est le symbole JS pour la concaténation, c'est-à-dire l'opération consistant à coller « bout à bout » différentes informations.

Si l'on regarde ce que ça donne concrètement dans la console JS, cela vous semblera peut être plus clair :

```
> var lang_list = '<ul>' +
  '<li>java</li>' +
  '<li>javascript</li>' +
  '<li>scala</li>' +
  '<li>haskell</li>' +
  '<li>go</li>' +
  '<li>rust</li>' +
  '<li>julia</li>' +
  '<li>basic</li>' +
  '<li>pascal</li>' +
  '<li>fortran</li>' +
  '<li>c</li>' +
  '<li>c++</li>' +
  '<li>c#</li>' +
  '<li>ruby</li>' +
  '<li>python</li>' +
  '</ul>';
< undefined
> lang_list
< "<ul><li>java</li><li>javascript</li><li>scala</li><li>haskell</li>
<li>go</li><li>rust</li><li>julia</li><li>basic</li><li>pascal</li>
<li>fortran</li><li>c</li><li>c++</li><li>c#</li><li>ruby</li>
<li>python</li></ul>"
```

On voit que la concaténation de nos petits bouts de code HTML ('', 'java', etc...) aboutit à une seule et même chaîne de caractères, qui est stockée dans la variable « lang_list ».

Croyez-le ou pas, il s'agit de code HTML valide. Je dois donc être en mesure de le placer à n'importe quel endroit de la page, mais comment faire cela ?

Il nous faut un point d'entrée dans la page HTML. Nous avons besoin d'un moyen pour JS d'identifier ce point d'entrée, et d'y placer le code HTML que nous venons de créer avec notre méthode.

Il existe en réalité plusieurs méthodes en JS pour trouver ce point d'entrée, mais nous allons utiliser la plus simple d'entre elle, j'ai nommé « getElementById » : il s'agit d'une méthode associée à l'objet JS « document ». Cet objet « document » est un objet que le navigateur met à

votre disposition, et qui contient beaucoup de méthodes très pratiques.

Comme son nom l'indique, « `getElementById` » nous permet d'accéder à un élément de la page, via un identifiant (Id). OK, ça tombe bien, car si vous vous souvenez bien, je vous ai demandé de créer une page HTML contenant une balise « `div` », et il se trouve que cette balise « `div` » a un identifiant qui s'appelle « `cible` » :

```
<div id="cible">ceci est le contenu d'une balise "div" de test</div>
```

Il est sympa papi « Greg » de vous avoir fait créer cet identifiant, hein ? Ben oui, je l'avais fait exprès, je ne suis pas prof pour rien 😊.

Mais j'arrête les explications, c'est à vous de jouer maintenant, saisissez les instructions ci-dessous et observez ce qui se passe :

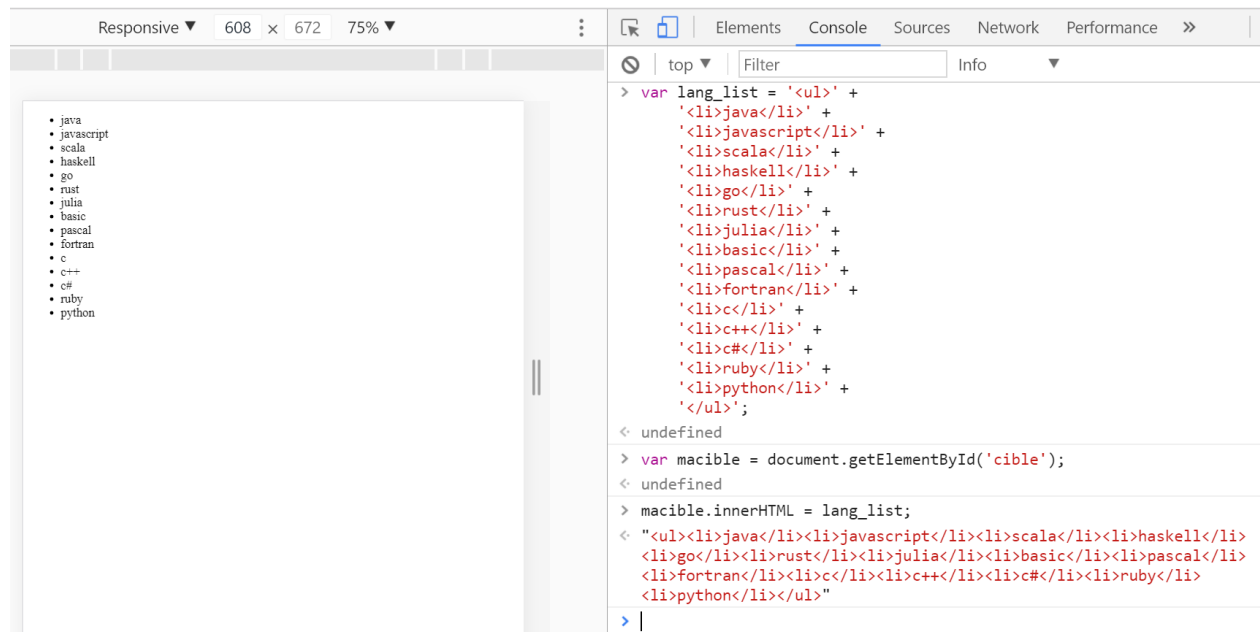
```
> var macible = document.getElementById('cible');  
< undefined  
-----  
> macible.innerHTML = lang_list;  
< "<ul><li>java</li><li>javascript</li><li>scala</li><li>haskell</li>  
    <li>go</li><li>rust</li><li>julia</li><li>basic</li><li>pascal</li>  
    <li>fortran</li><li>c</li><li>c++</li><li>c#</li><li>ruby</li>  
    <li>python</li></ul>"
```

La première instruction a consisté à créer une variable « `macible` ». Cette variable est un objets JS particulier qui nous sert de « pointeur » vers l'élément de la page qui a pour identifiant « `cible` », soit la valeur que l'on a transmis en paramètre à la méthode « `getElementById` ».

Grâce à ce pointeur sur l'élément de la page, nous disposons d'une propriété « `innerHTML` » qui est fournie par le JS pour notre objet JS particulier. Comme son nom le laisse supposer, cette propriété nous permet de taper directement dans le contenu HTML de la « `div` » sur laquelle nous sommes en train de « pointer ».

Donc, en tapant ceci : `macible.innerHTML = lang_list`
... nous disons au navigateur d'aller « coller » dans la « `div` » le code HTML contenu dans la variable « `lang_list` ».

Si vous avez bien travaillé, si vous n'avez pas commis d'erreur de syntaxe, normalement le contenu de la « div » a été remplacé par ce que vous voyez dans le cadre gauche ci-dessous :



Question : et si j'avais souhaité ajouter ma liste HTML à la suite du contenu existant de la « div », c'était possible ? Oui bien sûr. C'est facile à faire :

- Commencer par effacer le contenu de la div en mettant un texte quelconque comme dans l'exemple suivant :

```

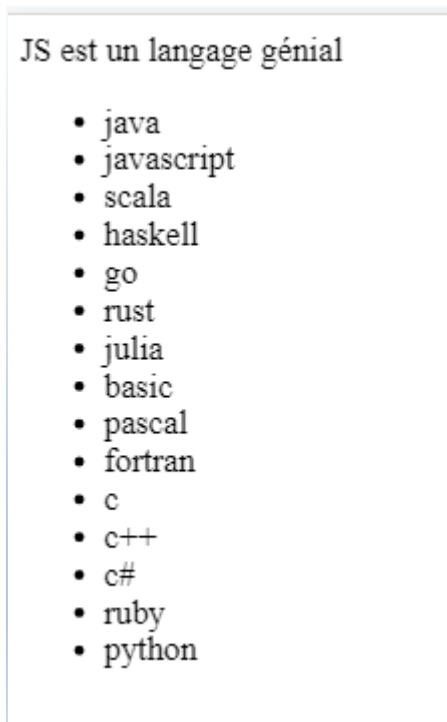
> macible.innerHTML = 'JS est un langage génial';
< "JS est un langage génial"
  
```

- Saisissez maintenant la ligne suivante (attention à bien utiliser « += ») :

```

> macible.innerHTML += lang_list;
< "JS est un langage génial<ul><li>java</li><li>javascript</li>
<li>scala</li><li>haskell</li><li>go</li><li>rust</li><li>julia</li>
<li>basic</li><li>pascal</li><li>fortran</li><li>c</li><li>c++</li>
<li>c#</li><li>ruby</li><li>python</li></ul>"
  
```

Si vous ne vous êtes pas trompé, vous devriez obtenir ceci :



Vous aurez peut être noté que l'on a fait appel à l'objet JS « macible » plusieurs fois, et cela très facilement. A partir du moment où vous avez créé un pointeur sur un élément de la page, cet objet est réutilisable autant de fois que nécessaire. Je crois important de le souligner, car j'ai rencontré à plusieurs reprises des développeurs qui n'avaient pas compris ce principe, et pensaient qu'il était indispensable de refaire un « getElementByld » avant chaque modification du contenu d'un élément de la page, même s'ils avaient déjà utilisé cet élément précédemment.

Nous avons vu beaucoup de notions dans ce chapitre, et en particulier une technique essentielle qui a consisté à interagir avec le DOM (Document Object Model) du navigateur. Ce DOM, c'est une représentation que le navigateur se fait de votre page HTML initiale. Le DOM est dynamique, c'est-à-dire que nous pouvons le modifier en utilisant des techniques JS comme celle que nous avons étudiées dans ce chapitre :

- nous avons utilisé la méthode « getElementByld » pour accéder à un élément du DOM via son identifiant,
- puis nous avons utilisé la propriété « innerHTML » pour modifier le contenu HTML

de l'élément du DOM sélectionné.

Nous allons améliorer notre technique au chapitre suivant, en étudiant une manière plus dynamique de construire notre code HTML.

2.4.2 La technique « de l'amateur éclairé »

Dans le chapitre précédent, nous avons utilisé une liste HTML « prémâchée ».

Dans le contexte d'une vraie application, ce cas de figure pourrait se présenter dans le cas où la construction des éléments HTML est du ressort d'une application PHP. Beaucoup d'applications web écrites entre 2000 et 2010 utilisaient ce paradigme, dans lequel le rôle du Javascript consistait à récupérer des bouts de code HTML transmis par le serveur, et à les « coller » quelque part dans la page, à la demande.

Mais au tournant des années 2010, avec l'arrivée conjointe du HTML5, d'une nouvelle version de Javascript (la norme ECMAScript 5), et la montée en puissance de nouveaux frameworks de développement comme Backbone, Knockout, AngularJS, etc... le rôle de JS a complètement changé. Dans les applications développées aujourd'hui, le rôle de PHP est de transmettre des données au format XML ou JSON, à charge pour le JS de réaliser le travail de construction du HTML. C'est ce que nous allons voir dans ce chapitre.

Reprenons le cas de notre liste de langages. Supposons que cette liste ne se présente plus sous la forme d'un code HTML, mais plutôt d'un tableau Javascript. C'est quoi un tableau Javascript ? c'est une variable de type liste que l'on va pouvoir manipuler via tout un tas de fonctions très pratiques.

Pour construire un tableau JS, on dispose de plusieurs solutions :

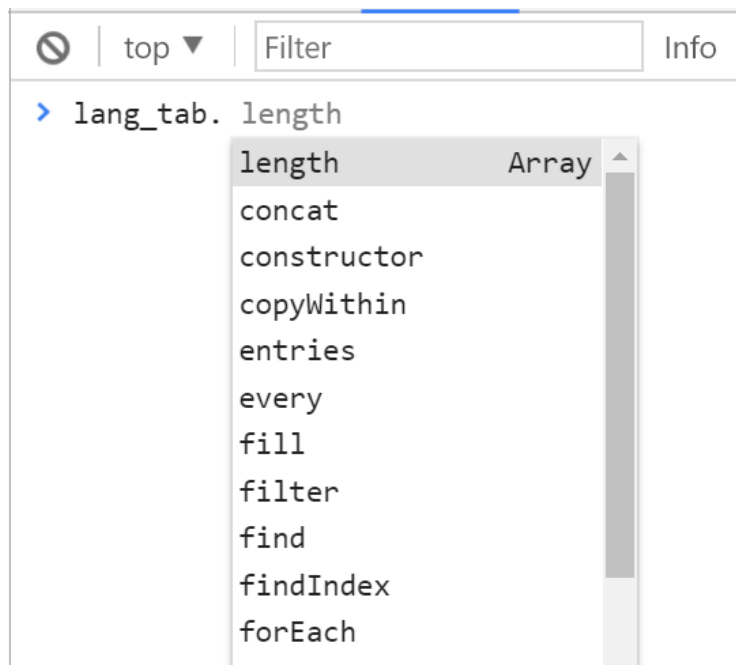
- solution 1 : on initialise le tableau et son contenu en une seule opération

```
var lang_tab = ['java', 'javascript', 'scala', 'haskell',  
               'go', 'rust', 'julia', 'basic', 'pascal',  
               'fortran', 'c', 'c++', 'c#', 'ruby', 'python'];
```


- solution 2 : on initialise le tableau vierge, puis on l'alimente via la méthode « push »

```
var lang_tab = []; // tableau vierge
lang_tab.push('java');
lang_tab.push('javascript');
lang_tab.push('scala');
lang_tab.push('haskell');
lang_tab.push('go');
lang_tab.push('rust');
lang_tab.push('julia');
lang_tab.push('basic');
lang_tab.push('pascal');
lang_tab.push('fortran');
lang_tab.push('c');
lang_tab.push('c++');
lang_tab.push('c#');
lang_tab.push('ruby');
lang_tab.push('python');
```

Quelle que soit la méthode de création utilisée, une fois le tableau créé nous disposons de différentes méthodes associées à l'objet de type « tableau ». Nous avons déjà vu « push », mais il y en a d'autres. On s'en rend vite compte, car en saisissant « lang_tab. » dans la console, l'autocomplétion se met automatiquement en route, qui nous donne un aperçu des possibilités :



Voici un petit échantillon de méthodes et propriétés intéressantes associées à notre objet de type tableau :

- « length » pour obtenir le nombre de postes du tableau

```
> lang_tab.length;  
◀ 15
```

- « join » pour générer une chaîne contenant tous les éléments du tableau reliés par un caractère séparateur (ici la virgule)

```
> lang_tab.join(',');  
◀ "java,javascript,scala,haskell,go,rust,julia,basic,pascal,fortran,c,c++,c#,ruby,python"
```

- « foreach » pour générer une boucle parcourant tous les éléments du tableau

```
> lang_tab.forEach(function(element, index) {  
  console.log(index+ ' ' + element);  
});  
0 java  
1 javascript  
2 scala  
3 haskell  
4 go  
5 rust  
6 julia  
7 basic  
8 pascal  
9 fortran  
10 c  
11 c++  
12 c#  
13 ruby  
14 python  
◀ undefined
```

- Etc...

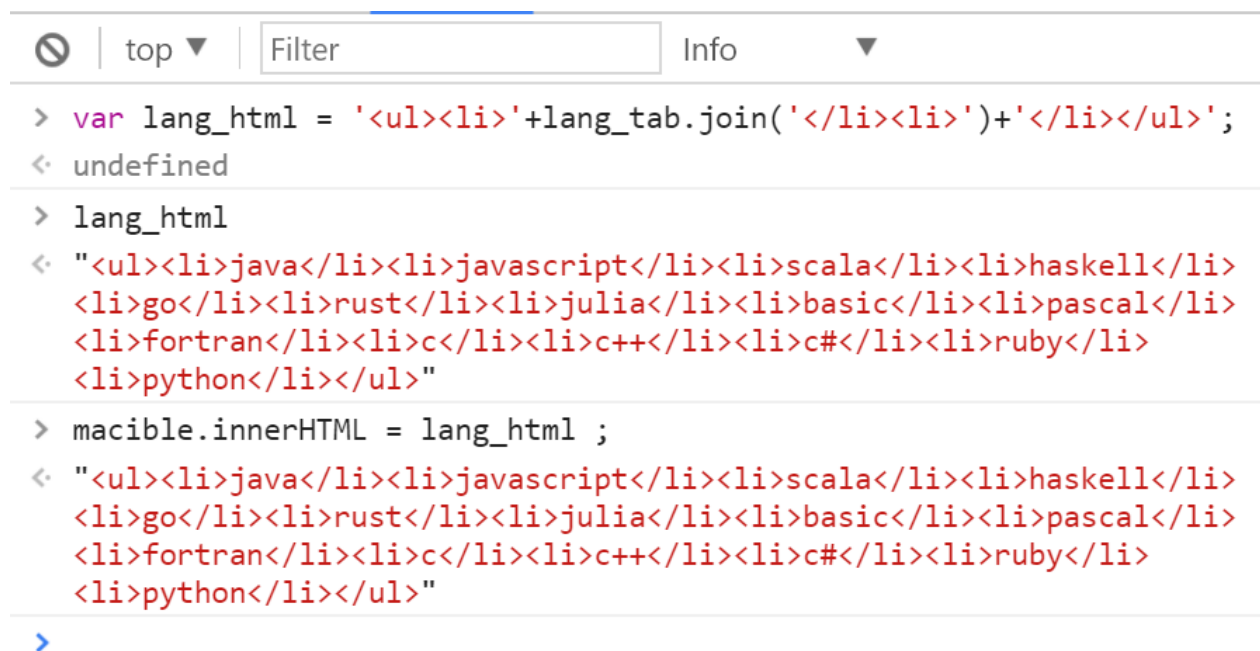
Tiens, sur la page précédente, nous avons vu au moins deux manières de créer notre code HTML, soit avec « join », soit avec « foreach ».

Je vous propose de fermer un moment ce document, de réfléchir à la manière dont vous pourriez procéder pour générer le code HTML de la liste, aussi bien avec « join » qu'avec « foreach ». Si vous trouvez des solutions, testez les dans votre navigateur, puis lisez ce qui suit.

Voyons ce que ça peut donner avec « join » :

```
var lang_html = '<ul><li>'+lang_tab.join('</li><li>')+</li></ul>';
```

Essayez cette ligne dans la console de votre navigateur :



```

> var lang_html = '<ul><li>'+lang_tab.join('</li><li>')+</li></ul>';
< undefined
> lang_html
< "<ul><li>java</li><li>javascript</li><li>scala</li><li>haskell</li>
<li>go</li><li>rust</li><li>julia</li><li>basic</li><li>pascal</li>
<li>fortran</li><li>c</li><li>c++</li><li>c#</li><li>ruby</li>
<li>python</li></ul>"
> macible.innerHTML = lang_html ;
< "<ul><li>java</li><li>javascript</li><li>scala</li><li>haskell</li>
<li>go</li><li>rust</li><li>julia</li><li>basic</li><li>pascal</li>
<li>fortran</li><li>c</li><li>c++</li><li>c#</li><li>ruby</li>
<li>python</li></ul>"
>

```

Et voilà le travail 😊 !!

- java
- javascript
- scala
- haskell
- go
- rust
- julia
- basic
- pascal
- fortran
- c
- c++
- c#
- ruby
- python

Bon, mais avec « foreach », ça donne quoi ?

Voici le code que je vous invite à tester dans la console de votre navigateur :

```
var lang_html = '<ul>';
lang_tab.forEach(function(element, index) {
    lang_html += '<li>' + element + '</li>';
});
lang_html += '</ul>';
```

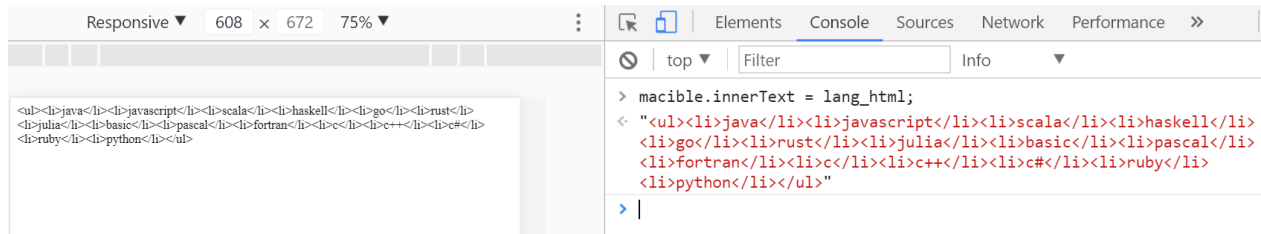
```
> var lang_html = '<ul>';
    lang_tab.forEach(function(element, index) {
        lang_html += '<li>' + element + '</li>';
    });
    lang_html += '</ul>';
< "<ul><li>java</li><li>javascript</li><li>scala</li><li>haskell</li>
    <li>go</li><li>rust</li><li>julia</li><li>basic</li><li>pascal</li>
    <li>fortran</li><li>c</li><li>c++</li><li>c#</li><li>ruby</li>
    <li>python</li></ul>"
> macible.innerHTML = lang_html ;
< "<ul><li>java</li><li>javascript</li><li>scala</li><li>haskell</li>
    <li>go</li><li>rust</li><li>julia</li><li>basic</li><li>pascal</li>
    <li>fortran</li><li>c</li><li>c++</li><li>c#</li><li>ruby</li>
    <li>python</li></ul>"
```

Là encore, si vous n'avez pas commis d'erreur de syntaxe, vous devriez voir apparaître la liste des langages dans la page web.

Vous avez peut être remarqué, quand vous avez commencé à saisir dans la console « macible.inner » que l'autocomplétion de la console vous offrait le choix entre « innerText » et « innerHTML ».

```
> macible.inner|Text
innerText      HTMLElement
innerHTML      Element
>innerHTML = lang_html ;
>innerHTML += lang_list;
>innerHTML = 'JS est un langage génial';
>innerHTML = lang_list;
```

Et si on essayait « innerText », pour voir...



Vous voyez que le code HTML transmis à la variable « macible » par l'intermédiaire de la propriété « innerText » a été injecté dans la page web tel quel, sans être interprété. A contrario, avec la propriété « innerHTML », le code HTML contenu dans la variable « lang_html » est « interprété » par le navigateur. A partir du moment où ce code est valide d'un point de la syntaxe HTML, le navigateur est en mesure de le comprendre, donc de l'interpréter et de modifier le DOM (le Document Object Model) en conséquence.

Bon c'est cool, je sais injecter du code HTML dans une portion de page, et je sais modifier le DOM en temps réel à partir d'un code HTML que j'ai construit en « live », au moment où cela me convient.

Mais y a-t-il d'autres manières de faire ? Oui bien sûr, et c'est ce que nous allons étudier au chapitre suivant.

2.4.3 La technique « full JS »

On peut générer du HTML au moyen de Javascript, sans écrire soi-même une seule ligne de code HTML. Cela ne signifie pas que l'on ne doit pas comprendre le HTML, bien au contraire. Mais plutôt que de créer du HTML en concaténant des bouts de chaînes de caractères (ce que nous avons au chapitre précédent), nous pouvons utiliser des fonctions Javascript qui vont nous permettre de créer tous les éléments HTML dont nous avons besoin pour notre liste de langages de programmation.

Par exemple, pour créer une liste « ul » contenant un seul élément « li », nous pouvons utiliser différentes méthodes du navigateur associées à l'objet « document », telles que :

- `document.createTextNode` : crée un nœud de texte, contenant un libellé, on utilisera cette technique pour créer des textes aussi bien dans des paragraphes, que dans des balises « li », que dans des balises de type « td », et même dans une « div »
- `document.createElement` : crée n'importe quel type de nœud, il suffit de lui transmettre en paramètre le type de nœud à créer ('ul', 'li', etc...)
- `parent.append(enfant)` : la méthode « append » va pouvoir être utilisée sur n'importe quel nœud pour lui faire adopter un enfant
- `element.setAttribute` : la méthode « setAttribute » pourra être utilisée sur n'importe quel élément du DOM pour ajouter des propriétés (ou attributs) HTML, comme par exemple : « id », « class », « required »

Voici un exemple de code dans lequel on crée une liste « ul » par programmation, en utilisant les méthodes présentées ci-dessus, sans utiliser le moindre code HTML (puisque le code HTML est produit par ces méthodes).

```
// création d'un texte node contenant un libellé (javascript)
// Attention : ce noeud texte n'est pour l'instant rattaché à aucun parent
var noeud_texte = document.createTextNode('javascript');

// création d'un noeud de type "li"
// Attention : ce noeud est pour l'instant virtuel,
// il n'est rattaché à aucun parent
var noeud_li = document.createElement('li');
// Résultat => <li></li>

// Rattachement du noeud "li" qui devient parent du noeud "texte"
// Attention : pour l'instant ce noeud "li" n'est rattaché
// à un aucun élément du DOM
noeud_li.append(noeud_texte);
// Résultat => <li>javascript</li>

// Création de la balise "ul" avec un "id" qui a pour valeur "cible_ul"
// Attention : ce noeud n'est pour l'instant rattaché à aucun parent
var noeud_ul = document.createElement('ul');
// Résultat => <ul></ul>
```

```

noeud_ul.setAttribute('id', 'cible_ul');
// Résultat => <ul id="cible_ul"></ul>

// Rattachement de la balise "ul" comme parent de la balise "li"
noeud_ul.append(noeud_li);

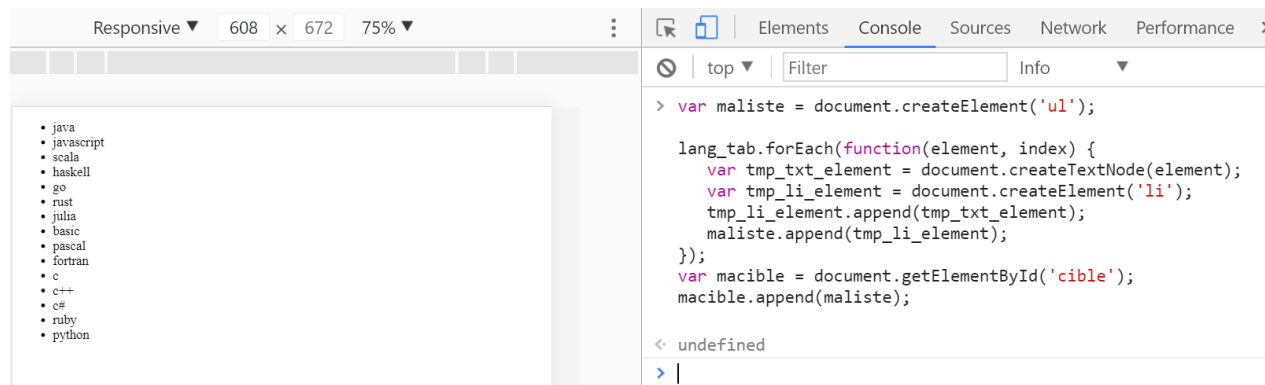
// pointage sur un élément du DOM qui a pour identifiant "cible"
var cible = document.getElementById('cible');

// Rattachement de la balise "ul" (jusqu'ici virtuelle) à un élément du DOM
// c'est seulement à partir de là que la liste "ul" va apparaître
// dans la page du navigateur
cible.append(noeud_ul);

```

Dans la suite de ce chapitre, nous allons utiliser ces techniques pour générer une liste « ul » de langage, à partir du tableau Javascript « lang_tab » vu dans un précédent chapitre.

Voici ce que ça donne, les explications vont suivre tout de suite après :



OK, étudions le code en détail.

La première étape a consisté à créer un élément HTML de type « ul ». Pour cela nous avons utilisé une méthode que nous n'avions pas encore vue, la méthode « createElement » associée à l'objet « document » :

```
var maliste = document.createElement('ul');
```

Vous constatez que nous créons une variable « maliste » pour conserver un « pointeur » sur l'élément créé. Sans cette variable « maliste » notre variable ne serait rattachée à rien, et serait inutilisable. Vous noterez qu'aucune balise n'apparaît sur la page web... c'est normal car à ce stade l'interpréteur Javascript n'est pas en mesure de savoir à quel endroit de la page nous allons insérer cette balise. Nous lui donnerons cette information plus tard, au moment où cela nous arrange.

Poursuivons, notre étude du code. Vous connaissez déjà la syntaxe de la méthode « forEach ».

```
lang_tab.forEach(function(element, item) {  
    ...  
});
```

Ce qui se passe à l'intérieur de la boucle est très intéressant, il y a 4 étapes distinctes :

- Etape 1 : création d'un élément du DOM qu'on appelle un « text node » (un nœud de type texte). Tout élément d'une page contenant du texte contient en réalité un nœud de ce type. C'est le cas des cellules de tableau HTML, des balises paragraphes (p)... et bien sûr des balises « li » :

```
var tmp_txt_element = document.createTextNode(element);
```

- Etape 2 : création d'un élément du DOM de type « li » :

```
var tmp_li_element = document.createElement('li');
```

- Etape 3 : rattachement de l'élément de type « text node » à l'élément de type « li » au moyen de la méthode « append » :

```
tmp_li_element.append(tmp_txt_element);
```

- Etape 4 : rattachement de l'élément de type « li » à l'élément de type « ul » au moyen de la méthode « append »

```
maliste.append(tmp_li_element);
```

Le code complet de notre boucle est le suivant :

```
lang_tab.forEach(function(element, item) {  
    var tmp_txt_element = document.createTextNode(element);  
    var tmp_li_element = document.createElement('li');  
    tmp_li_element.append(tmp_txt_element);  
    maliste.append(tmp_li_element);  
});
```

A la fin de la boucle, nous avons une liste complète, mais elle n'apparaît toujours pas sur la page web. Pour qu'elle apparaisse, nous avons besoin de rattacher la variable « maliste », à un élément de la page, par exemple à la « div » ayant pour identifiant « cible » :

```
var macible = document.getElementById('cible');  
macible.append(maliste);
```

Et voilà, cette fois-ci, la liste apparaît.

En résumé, pour créer un élément, nous utilisons la méthode « createElement », puis nous rattachons cet élément à un élément de niveau supérieur dans la page (par exemple, un « text node » vers une balise « li », ou encore une balise « li » à une balise « ul », etc...).

Cette manière de faire peut sembler plus compliquée, mais en réalité elle est très puissante, et avec un peu de pratique, on s'y fait très bien. On peut se servir de cette technique pour créer tout type d'élément HTML (canvas, audio, script, img, etc...).

2.4.4 Générer un tableau HTML 4 colonnes

Finalement, générer une liste HTML, ce n'est pas si compliqué. Du coup un de mes élèves m'a demandé si on ne pourrait pas créer un tableau HTML – plutôt qu'une simple liste – à partir de notre liste de langages. Voilà une excellente idée d'exercice, et qui fait appel à quelques notions d'algorithmique intéressantes. Alors, allons-y 😊

Mais au fait, quel type de tableau générer. Si on se contente de générer un tableau contenant une seule colonne, et autant de lignes que de langages (soit une cellule par ligne et langage), ce n'est pas très intéressant. En revanche, ce qui serait intéressant, c'est d'essayer de générer 4 colonnes par lignes, et d'y placer à chaque fois 4 langages. Concrètement, je voudrais obtenir le résultat suivant :

java	javascript	scala	haskell
go	rust	julia	basic
pascal	fortran	c	c++
c#	ruby	python	

Bon, je l'ai fait super moche, mais c'est juste pour que voyiez l'idée.

On va avoir besoin d'une « div » réceptrice de notre tableau HTML généré, alors créons la tout de suite, comme ça on n'en parle plus :

```
<div id="cible">contenu qui sera écrasé par le Javascript</div>
```

Rappel sur le contenu de la variable contenant la liste des langues :

```
var lang_tab = ['java', 'javascript', 'scala', 'haskell',  
               'go', 'rust', 'julia', 'basic', 'pascal',  
               'fortran', 'c', 'c++', 'c#', 'ruby', 'python'];
```

Et maintenant voici une première version du code Javascript servant à générer le tableau à 4 colonnes :

```
var liste = '<table>';
var nbcol = 0;
lang_tab.forEach(function(element, index) {
    if (nbcol > 3) {
        nbcol = 0;
        liste += '</tr>';
    }
    if (nbcol == 0) {
        liste += '<tr>';
    }
    liste += '<td>' + element + '</td>';
    nbcol += 1;
});
liste += '</table>';

var cible = document.getElementById('cible');
cible.innerHTML = liste ;
```

Le nœud de l'affaire, c'est que nous avons besoin d'une variable de type compteur, qui va nous servir à déterminer dans quelle colonne nous nous situons à un moment donné. C'est la variable « nbcol » qui va remplir cet office.

A chaque itération dans notre boucle « forEach », nous allons tester la valeur de « nbcol » :

- A la 1^{ère} itération, « nbcol » est à zéro, donc on ne passe pas dans le premier test (nbcol > 3)
- En revanche, on passe dans le second test (car la condition « nbcol égal à zéro » est bien remplie) et on génère la balise « tr » débutant la ligne HTML. Vous noterez la syntaxe du test avec le « double égal ». Si on avait utilisé un « simple égal », cela aurait signifié que l'on affectait la valeur zéro à la variable « nbcol ». L'opération d'affectation s'étant bien déroulée, elle aurait naturellement renvoyé la valeur « true », ce qui aurait complètement faussé l'algorithme.
- Ensuite on génère la balise « td » relative au langage considéré (la première fois, il s'agit du langage « Java »)
- Tout de suite après, on incrémente la variable « nbcol » (qui passe donc à 1), et on repart dans une nouvelle itération de la boucle
- A la seconde itération, le premier test (nbcol > 3) est ignoré, le second aussi (nbcol == 0), on génère simplement une nouvelle balise « td » pour le langage « Javascript », et on incrémente de nouveau « nbcol » (qui passe à 2)
- A la 3^{ème} itération, même combat, on génère la balise « td » pour le langage « Scala », puis on incrémente de nouveau « nbcol », et on repart dans une nouvelle itération
- A la 4^{ème} itération, le premier test est réalisé (nbcol est bien égal à 3 à ce stade), donc on génère la balise « tr » de fermeture de la ligne, et on repasse « nbcol » à zéro
- Toujours pendant cette 4^{ème} itération, le second test est lui aussi réalisé (car la condition

nbcol == 0 est remplie), on génère donc une nouvelle balise « tr » pour démarrer la nouvelle ligne

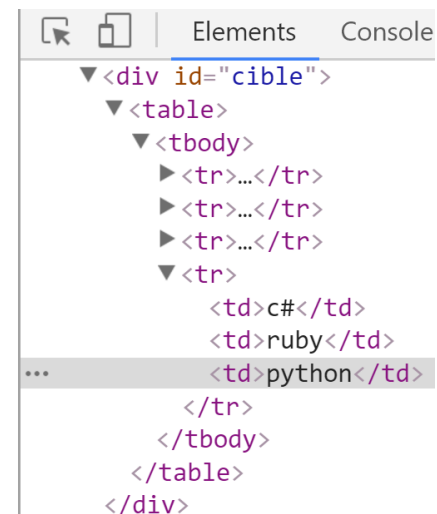
- On génère ensuite une cellule « td » pour le langage « Go », et ainsi de suite...

A l'écran, cela semble bien fonctionner, mais il manque quand même quelque chose. Si on regarde le code HTML généré de près, on s'aperçoit en effet qu'il manque la balise « tr » de fermeture de la dernière ligne.

```
<table><tr><td>java</td><td>javascript</td><td>scala</td><td>haskell</td></tr><tr><td>go</td><td>rust</td><td>julia</td><td>basic</td></tr><tr><td>pascal</td><td>fortran</td><td>c</td><td>c++</td></tr><tr><td>c#</td><td>ruby</td><td>python</td></table>
```

Mais ce n'est pas dramatique, car le navigateur est très tolérant, et il génère automatiquement dans le DOM, la balise « tr » manquante :

java	javascript	scala	haskell
go	rust	julia	basic
pascal	fortran		
c#	ruby	python	



En revanche, ce qui est plus embêtant, c'est que notre dernière ligne de tableau contient 3 colonnes alors que les précédentes lignes contiennent 4 colonnes. C'est normal, notre tableau Javascript contient 15 postes, ce n'est pas un multiple de 4. Mais nous pouvons facilement rattraper le coup en ajoutant une petite boucle complémentaire, après la boucle principale, et juste avant la balise de fermeture du tableau. Voici une manière très simple d'écrire cette boucle de rattrapage :

```
if (nbcol < 4) {
    while(nbcol < 4) {
        liste += '<td>&nbsp;</td>';
        nbcol += 1;
    }
    liste += '</tr>';
}
```

Tiens, voici une technique de boucle qu'on n'avait pas encore vue : « while ». Dans notre cas, il s'agit de boucler « tant que la valeur de nbcol est inférieure à 4 ».

A l'intérieur de la cellule « td » générée, on utilise le symbole HTML « » qui a pour effet de générer un blanc.

Vous remarquerez qu'on en a profité de ce code pour ajouter la balise « tr » manquante. Tant qu'à faire, autant faire les choses proprement 😊.

Pour une meilleure lisibilité de l'exemple, voici le code source complet :

```
<div id="cible">contenu qui sera écrasé par le Javascript</div>
<script>
    var lang_tab = ['java', 'javascript', 'scala', 'haskell',
        'go', 'rust', 'julia', 'basic', 'pascal',
        'fortran', 'c', 'c++', 'c#', 'ruby', 'python'];

    var liste = '<table>'; var nbcol = 0;
    lang_tab.forEach(function(element, index) {
        if (nbcol > 3) {
            nbcol = 0;
            liste += '</tr>';
        }
        if (nbcol == 0) {
            liste += '<tr>';
        }
        liste += '<td>' + element + '</td>';
        nbcol += 1;
    });
    if (nbcol < 4) {
        // Boucle de rattrapage pour les cas où le nombre de postes
        // du tableau n'est pas un multiple de 4
        while(nbcol < 4) {
            liste += '<td>&nbsp;</td>'; // cellule avec un blanc
            nbcol += 1;
        }
        liste += '</tr>';
    }
    liste += '</table>';

    var cible = document.getElementById('cible');
    cible.innerHTML = liste ;
</script>
```

Vous trouverez en annexe (chapitre 4.2) une autre version du code ci-dessus. Dans cette autre version, on a remplacé la génération du code HTML par les méthodes « createElement » et « createTextNode ».

2.5 Créer du JS avec du HTML

Nous avons vu comment générer du HTML à partir de Javascript, mais pourrait-on générer du Javascript à partir de HTML ? Ou plus exactement, serait-on en mesure de générer un tableau Javascript contenant la liste des langages de programmation, à partir d'une liste HTML comme celle que nous avons créée aux chapitres précédents ? Et dans quel cas pourrions-nous avoir besoin de ce genre de technique ?

Commençons par répondre à la dernière question. Si vous n'avez jamais entendu parler de « web scraping », c'est le moment d'aborder le sujet.

Voici un extrait de la définition Wikipédia du terme Web Scraping :

« Le web scraping (parfois appelé harvesting) est une technique d'extraction du contenu de sites Web, via un script ou un programme, dans le but de le transformer pour permettre son utilisation dans un autre contexte, par exemple le référencement. /.../ Cela permet de récupérer le contenu d'une page web en vue d'en réutiliser le contenu. »

Je vous invite à lire la suite de l'article sur Wikipédia :

https://fr.wikipedia.org/wiki/Web_scraping

Il paraît que la technique dite du « web scraping » n'a pas bonne presse... c'est marrant car beaucoup de gens dans le monde de la presse et de l'édition utilisent cette technique qui consiste à prélever des informations dans des pages webs. Et les entreprises ne s'en privent pas, qui essaient de constituer ainsi, des listes de prospects à bon compte. Au-delà de ces considérations mercantiles, il y a un intérêt plus intéressant et légal à l'affaire. En effet, toute personne qui a besoin de collecter des données pour mettre en lumière des faits, et pour rapprocher des données de sources différentes, peut être intéressée par le « web scraping ».

Cette mise au point étant faite, voyons comment extraire la liste des langages d'une liste HTML. Si l'on y regarde de plus près, une liste HTML c'est un mélange de balises HTML avec des données, et ces données se trouvent localisées dans un type d'élément particulier évoqué au chapitre précédent, le « text node ». Voilà qui devrait nous faciliter la tâche, il nous faut donc trouver un moyen d'isoler les « text nodes » et d'en extraire les informations qui nous intéressent.

Tout d'abord, il nous faut trouver un moyen d'accéder aux balises « li » qui nous intéressent. Nous avons plusieurs moyens à notre disposition. Nous allons les passer en revue, et voir les « pour et contre » de chaque technique.

2.5.1 la méthode « `getElementsByName` »

Sachant que nous souhaitons accéder à l'ensemble des balises « `li` » de notre liste de langages, nous pouvons utiliser la méthode « `getElementsByName` » associé à l'objet « `document` » :

```
> document.getElementsByTagName('li')
< ▼ (15) [li, li, li, li, li, li, li, li, li, li, li, li, li, li, li] ⓘ
    length: 15
    ▶ 0: li
    ▶ 1: li
    ▶ 2: li
    ▶ 3: li
    ▶ 4: li
    ▶ 5: li
    ▶ 6: li
    ▶ 7: li
    ▶ 8: li
    ▶ 9: li
    ▶ 10: li
    ▶ 11: li
    ▶ 12: li
    ▶ 13: li
    ▶ 14: li
    ▶ __proto__: HTMLCollection
```

Nous venons de constituer une « `HTMLCollection` ». En apparence, on dirait un tableau Javascript, et ça en a quelques attributs, comme la propriété « `length` » qui nous permet de savoir que notre `HTMLCollection` contient 15 éléments, numérotés de 0 à 14.

```
> document.getElementsByTagName('li').length;
< 15
```

Ok, donc à partir de ces éléments, nous sommes en mesure d'écrire une boucle qui parcourt la `HTMLCollection`. Un petit coup de « `forEach` » et le tour est joué... C'est bien vu, mais non, ça ne va pas fonctionner, enfin... pas aussi simplement. Car je vous ai dit qu'une `HTMLCollection` n'avait pas tous les attributs d'un tableau Javascript. Il lui manque en particulier la méthode « `forEach` ». Heureusement, à chaque problème il y a une solution, et en l'occurrence on a ici au moins deux solutions :

- La boucle « `for` »
- La boucle « `forEach` » par une voie un peu détournée 😊

Bon, on n'avait pas encore vu la boucle « for » dans le cadre de ce support de cours, alors c'est une bonne occasion pour l'étudier. C'est la même boucle « for » que l'on trouve dans de nombreux autres langages (comme PHP par exemple) :

```
var mes_li = document.getElementsByTagName('li');

for (var i = 0 ; i < mes_li.length ; i++) {
}
```

Explication : la boucle « for » se compose de 3 parties :

- L'initialisation (var i = 0)
- La condition vérifiant si la boucle doit se poursuivre (i < mes_li.length)
- L'incrémentation du compteur (« i++ », qu'on aurait pu écrire aussi « i+=1 »)

Bon, je n'insisterai pas trop là-dessus dans ce cours d'introduction au Javascript, mais pour ma part je préfère cette version de la même boucle « for » :

```
for (var i = 0, imax = mes_li.length ; i < imax ; i++) {
}
```

En effet, dans cette seconde version, la partie initialisation initialise 2 variables (i et imax). La variable « imax » reçoit le nombre de postes de la HTMLCollection « mes_li », cela évite de recalculer le nombre de poste à chaque itération (c'est ce qui se passe quand on écrit « i < mes_li.length »). Ce n'est pas grave sur une petite liste, mais ça le devient vite sur des listes de taille plus importantes).

Mais revenons à nos moutons.

Je rappelle que notre objectif est de récupérer les éléments de la HTMLCollection dans un tableau Javascript, donc on commence par initialiser un tableau « mes_datas » vide :

```
mes_datas = [];
```

La boucle ci-dessous va balayer tous les éléments de la HTMLCollection, et pour chaque élément de cette liste, on va extraire le contenu HTML via la propriété « innerHTML ». On utilise la méthode « push » de notre tableau « mes_datas » pour y insérer les données extraites de la HTMLCollection :

```
for (var i = 0 ; i < mes_li.length ; i++) {
    mes_datas.push(mes_li[i].innerHTML);
}
```


Pour vérifier que notre tableau « mes_datas » est bien alimenté, testons le en saisissant son nom :

```
> mes_datas ;
< (15) ["java", "javascript", "scala", "haskell", "go", "rust",
  ► "julia", "basic", "pascal", "fortran", "c", "c++", "c#", "ruby",
    "python"]
```

Vous noterez la présence des crochets et du « i » juste derrière « mes_li ». Si on avait oublié cette information essentielle, le résultat de notre extraction aurait été très différent :

- La bonne version :

```
mes_datas.push(mes_li[i].innerHTML);
```
- La mauvaise version :

```
mes_datas.push(mes_li.innerHTML);
```

Le résultat obtenu avec la mauvaise version :

```
> mes_datas
< (15) [undefined, undefined, undefined, undefined, undefined,
  ► undefined, undefined, undefined, undefined, undefined, undefined,
    undefined, undefined, undefined, undefined]
```

Eh oui, nous n'avons pas extrait l'information au bon niveau, mais le navigateur est « permissif », il ne nous a pas renvoyé d'erreur, il nous a simplement renvoyé « undefined » car il n'a pas été en mesure de nous renvoyer une information plus pertinente, compte tenu du contexte.

C'est une erreur d'inattention assez fréquente, rassurez-vous, il n'y a pas de honte... Que celui qui ne l'a jamais faite me lance le premier octet 😊.

Bon, je vous ai dit que la méthode « forEach » n'était pas disponible avec notre HTMLCollection, mais vous n'êtes pas obligé de me croire, et ça ne coûte rien d'essayer :

```
> mes_li.length
< 15
> mes_li.forEach
< undefined
```

Ouais... « undefined », ça semble « mal barré » non ?

Heureusement, la méthode « `forEach` » a un petit secret, on peut l'utiliser avec un tableau vide, de cette manière :

```
[].forEach.call(mon_tableau , function(element, index) {
  ...
}) ;
```

On l'essaye dans notre contexte ?

Comme je suis sympa avec vous, je vous mets le code en clair, pour faciliter le copier-coller :

```
mes_datas = [];
[].forEach.call(mes_li, function(element, index) {
  mes_datas.push(element.innerHTML);
}) ;
mes_datas;
```

Petite copie d'écran de ce que ça donne dans le navigateur :

```
>   mes_datas = [];
    [].forEach.call(mes_li, function(element, index) {
      mes_datas.push(element.innerHTML);
    }) ;
    mes_datas;
< (15) ["java", "javascript", "scala", "haskell", "go", "rust",
  ► "julia", "basic", "pascal", "fortran", "c", "c++", "c#", "ruby",
    "python"]
```

La vache, c'est quand même cool, le Javascript !! 😊

Bon, mais j'ai dit plus haut que j'allais parler des inconvénients de chaque méthode. Je rappelle qu'on a créé notre HTMLCollection avec cette technique :

```
var mes_li = document.getElementsByTagName('li');
```

C'est bien mignon, ça nous a renvoyé toutes les balises de notre page... mais vous croyez vraiment que dans la « vraie vie » ça va toujours être aussi simple ?

Que se passerait-il si la page dont nous souhaitons « scraper » le contenu contenait plusieurs listes HTML, et que seule l'une de ces listes nous intéressait ? Eh bien on serait dans la m....

2.5.2 Affinage des méthodes de sélection

Certaines personnes travaillent dans l'affinage de fromages, nous nous allons œuvrer dans l'affinage de « sélection HTML ».

Plusieurs cas de figure peuvent se présenter dans la « vraie vie ». Peut être que la liste « ul » qui nous intéresse a un identifiant comme dans cet exemple :

```
▼ <div id="cible">
  ▼ <ul id="myprecious"> == $0
    <li>java</li>
    <li>javascript</li>
    <li>scala</li>
    <li>haskell</li>
```

Dans ce cas de figure, on peut écrire ceci :

```
> var myprecious = document.getElementById('myprecious');
< undefined
> var mes_li = myprecious.getElementsByTagName('li');
< undefined
> mes_li
< ► (15) [li, li, li, li, li, li, li, li, li, li, li, li, li, li, li]
```

On a commencé par créer un pointeur sur la balise « ul » via son identifiant « myprecious ». J'ai donné le même nom à la variable créée pour l'occasion.

A partir de cette variable « myprecious », je dispose de la méthode « getElementsByTagName » que j'avais utilisée précédemment sur l'objet « document ». C'est strictement la même méthode, mais on l'utilise à partir d'un élément particulier du DOM, le nœud qui a pour identifiant « myprecious ».

C'est tout, affaire classée, on peut remballer... non ?

Non, on ne remballer pas encore, car toujours dans la « vraie vie », il se pourrait que notre balise « ul » n'ait pas d'identifiant particulier. Aurait-elle une classe par exemple ? Oui ? Ah tiens, une classe « myprecious », comme c'est étrange :

```
▼ <div id="cible">
  ▼ <ul class="myprecious"> == $0
    <li>java</li>
    <li>javascript</li>
    <li>scala</li>
```

Attention danger, avec un identifiant, c'était facile. On sait que dans une page HTML normalement constituée, un identifiant est forcément unique. Mais pour les classes c'est une autre affaire. Plusieurs listes pourraient avoir cette même classe « myprecious », alors... retour à la case départ ? Pas tout à fait, si dans notre cas on sait que la liste qui nous intéresse se trouve dans une « div » spécifique et qu'on sait l'isoler via son identifiant (qui s'appelle « cible » dans notre exemple), alors on est sauvés 😊

```
> var myprecious =
  document.getElementById('cible').getElementsByClassName('myprecious');
< undefined
> var mes_li = myprecious[0].getElementsByTagName('li');
< undefined
> mes_li
< ► (15) [li, li, li, li, li, li, li, li, li, li, li, li, li, li, li]
```

Et voilà, avec Javascript, à chaque problème il y a une solution !! 😊

Mais je vais vous montrer dans le dernier chapitre de cette série d'introduction au « web scraping », qu'il y a une méthode géniale qui permet de faire tout ça en une seule instruction. Je vois déjà la tête de mes élèves, déçus, se disant que ce prof se paie vraiment leur tête. C'est vrai que ça fait un peu genre... le type qui « se la pète », en sortant une fleur, puis un lapin, puis une tortue de son chapeau...

Mais les techniques que nous étudions sont fondamentales, les maîtriser fera de vous d'excellents développeurs Javascript. Vous serez capables de vous sortir de toutes les situations, sans stress et avec le sourire. Alors courage, car je vais vous montrer un truc vraiment super dans le chapitre qui suit.

2.5.3 Le sélecteur « couteau suisse »

La technique que je vais vous montrer est apparue vers 2009-2010, avec la norme ECMAScript 5, soit à peu près en même temps que l'arrivée de la norme HTML5. Il s'agit d'une API qui se présente sous la forme de deux méthodes :

- `document.querySelector('selecteur')` ; // renvoie une seule occurrence
- `document.querySelectorAll('selecteur')` ; // renvoie x occurrences

Le framework JQuery fournissait ce type de fonctionnalité depuis déjà longtemps, et on peut dire que cette fonctionnalité avait largement contribué au succès de JQuery auprès des web designers et des intégrateurs. Ce qu'on sait moins c'est que cette fonctionnalité était fournie à JQuery par un composant Javascript répondant au doux nom de Sizzle :

<https://sizzlejs.com/>

L'API « `querySelector` » (et sa frangine « `querySelectorAll` ») couvre donc le même besoin, mais de manière native, c'est-à-dire avec de bien meilleures performances que ne pouvait le faire JQuery et Sizzle.

Le sélecteur que l'on transmet à « `querySelector` » et « `querySelectorAll` », c'est un sélecteur CSS identique à ceux que l'on utilise en CSS, et en particulier en CSS3. Nous avons abondamment évoqué ce sujet dans le support de cours « CSS3 – Premiers pas » (si vous l'avez zappé, il est temps de vous y intéresser).

Par exemple, pour sélectionner un élément qui a pour identifiant « cible », on écrira :

```
> document.querySelector('#cible');  
< ▶ <div id="cible">...</div>
```

Et si on souhaite sélectionner toutes les balises « ul » d'une page :

```
> document.querySelectorAll('ul');  
< ▶ [ul.myprecious]
```

Pour sélectionner toutes les balises pointant sur la classe « myprecious » :

```
> document.querySelectorAll('.myprecious');  
< ▶ [ul.myprecious]
```

Mais là où ça devient carrément fun, c'est que l'on peut vraiment utiliser toute la puissance des sélecteurs CSS3. Si vous vous rappelez du dernier exemple relatif au chapitre précédent, nous pouvons écrire ceci :

```
> document.querySelectorAll('#cible > ul.myprecious > li');
< ▶ (15) [li, li, li, li, li, li, li, li, li, li, li, li, li, li, li]
```

Eh oui, nous avons isolé un élément de la page qui a pour identifiant « cible » (il s'agit d'une « div » mais ça c'est secondaire). A l'intérieur de cet élément, nous avons sélectionné toutes les balises « li » qui appartiennent à une balise « ul » rattachée à la classe « myprecious ».

On notera que « `querySelector` », avec le même sélecteur en paramètre, renvoie le premier élément de la liste, en l'occurrence le langage « java » :

```
> document.querySelector('#cible > ul > li');
< <li>java</li>
```

A partir du moment où nous savons obtenir cette liste d'éléments, nous sommes en mesure d'appliquer les mêmes boucles que nous avons utilisées au chapitre précédent, par exemple :

```
> var mes_li = document.querySelectorAll('#cible > ul > li');
  mes_datas = [];
  [].forEach.call(mes_li, function(element, index) {
    mes_datas.push(element.innerHTML);
  });
  mes_datas;
< ▶ (15) ["java", "javascript", "scala", "haskell", "go", "rust", "julia",
  "basic", "pascal", "fortran", "c", "c++", "c#", "ruby", "python"]
```

Je pense utile de rappeler l'exemple de sélecteur que j'avais donné dans le support de cours « CSS3 – Premier pas » car il est intéressant :

```
var test = document.querySelectorAll('input[type=text]:required');
console.log(test);
```

... mais je ne donne pas d'explication ici, je vous invite plutôt à lire (ou relire) le support de cours concerné. Et je vous recommande particulièrement de lire, à la fin du chapitre 2.4.4 de ce support, le parallèle que je fais entre sélecteur CSS et SQL.

Avant de clore ce chapitre, je vous invite à déplier la liste renvoyée par le sélecteur suivant :

```
> document.querySelectorAll('#cible > ul > li');
< ▼ (15) [li, li, li, li, li, li, li, li, li, li, li, li, li, li, li] ⓘ
  length: 15
    ▶ 0: li
    ▶ 1: li
    ▶ 2: li
    ▶ 3: li
    ▶ 4: li
    ▶ 5: li
    ▶ 6: li
    ▶ 7: li
    ▶ 8: li
    ▶ 9: li
    ▶ 10: li
    ▶ 11: li
    ▶ 12: li
    ▶ 13: li
    ▶ 14: li
    ▶ __proto__: NodeList
```

Vous avez peut être remarqué qu'en bas de la liste, le navigateur ne nous indique pas « HTMLCollection » mais « NodeList ». La « NodeList » souffre des mêmes limitations que la « HTMLCollection ». La « NodeList » fournit la propriété « length », mais pas la propriété « forEach », donc on utilise les mêmes techniques d'itération qu'avec la « HTMLCollection ».

Mais pourquoi cette différence ? Car il y en a une.

- La « NodeList », comme son nom l'indique, c'est une liste de nœuds du DOM. Chaque élément à l'intérieur du DOM est un nœud. Une « div » est un nœud, une « ul » en est un autre, etc...
- La « HTMLCollection » est une « NodeList » particulière car elle est vivante (« live »). En effet, toute modification du DOM est automatiquement répercutée dans une HTMLCollection. Je rappelle que les APIs « getElementById », « getElementsByClassName » et « getElementsByTagName » renvoient des « HTMLCollection ».
- A l'inverse, la « NodeList » générée par les APIs « querySelector » et « querySelectorAll » est une liste de nœuds figée, pour ainsi dire « morte ».

Pour bien comprendre, prenons pour exemple notre liste de langages de programmation.

Je crée deux variables :

- La variable « mes_li1 » est une HTMLCollection constituée de toutes les « li » de ma liste, soit 15 éléments, via la méthode « getElementsByTagName »
- La variable « mes_li2 » est une NodeList constituée des mêmes 15 éléments, via la méthode « querySelector »

```

> var mes_li1 = document.getElementsByTagName('li');
< undefined

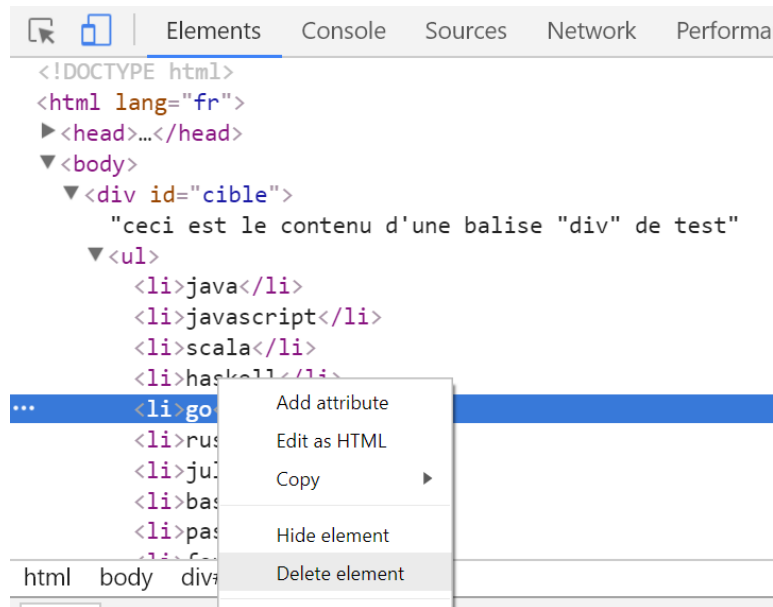
> mes_li1
< ► (15) [li, li, li, li, li, li, li, li, li, li, li, li, li, li, li]

> var mes_li2 = document.querySelectorAll('li');
< undefined

> mes_li2
< ► (15) [li, li, li, li, li, li, li, li, li, li, li, li, li, li, li]

```

Comme j'ai la flemme, je vais vous montrer un raccourci pour supprimer un élément de la liste sans écrire une ligne de code : je fais un « clic droit » sur la ligne correspondant au langage « go » et je sélectionne l'option « Delete element ». Eh hop, plus de langage « go » :



Que nous dit le navigateur, si on lui demande de moucharder concernant les variables « mes_li1 » et « mes_li2 » ?

```

> mes_li1
< ► (14) [li, li, li, li, li, li, li, li, li, li, li, li, li, li]

> mes_li2
< ► (15) [li, li, li, li, li, li, li, li, li, li, li, li, li, li, li]

```


Eh oui, la `HTMLCollection` référencée dans la variable « `mes_li1` » est bien « vivante », tandis que la « `NodeList` » référencée dans la variable « `mes_li2` » est une photographie figée du DOM à un instant donné.

Cette différence de comportement n'est pas problématique dans notre contexte de « web scraping », mais dans des cas plus spécifiques, la connaissance de cette distinction entre « `NodeList` » et « `HTMLCollection` » pourra vous épargner quelques maux de tête.

Si vous avez tenu le coup jusqu'ici, toutes mes félicitations. Vous êtes courageux, et vous méritez mon plus profond respect. Si certains points vous semblent encore nébuleux, dites-vous que c'est normal. Beaucoup des points que nous avons étudiés ici ne sont pas franchement intuitifs. Il m'a fallu pas mal de temps pour assimiler certaines de ces techniques, mais aujourd'hui elles me permettent de réaliser à peu près tout ce que je veux dans une page web. Et je ne doute pas qu'elles vont vous rendre beaucoup de services d'ici peu.

2.6 Notion de fonctions

2.6.1 Généralités

Les fonctions en Javascript constituent un formidable terrain d'expérimentation.

Nous allons nous cantonner dans cette introduction à quelques techniques relativement simples, qui nous seront utiles dans le chapitre suivant.

La création de fonctions en Javascript peut se faire de deux manières différentes :

- Méthode 1 : déclaration avec le mot clé « function » suivi du nom de la fonction

```
function mafonction(param1, param2) {  
    if (typeof param1 == 'number' && typeof param2 == 'number') {  
        return (param1 + param2) * 100 ;  
    } else {  
        return 'message : ' + param1 + ',' + param2 ;  
    }  
}
```

- Méthode 2 : création d'une variable de type « function » référençant le code d'une fonction anonyme

```
var mafonction = function (param1, param2) {  
    if (typeof param1 == 'number' && typeof param2 == 'number') {  
        return (param1 + param2) * 100 ;  
    } else {  
        return 'message : ' + param1 + ',' + param2 ;  
    }  
};
```

La différence entre ces deux manières de déclarer une fonction est subtile, et si la première méthode ne présente généralement pas de difficulté pour les développeurs débutants, la seconde méthode est souvent jugée plus déroutante. Mais avec un peu de pratique, on s'y fait vite.

Un point essentiel à comprendre avec les fonctions anonymes, c'est que le code qui se trouve à l'intérieur des accolades ne sera affecté à la variable « mafonction » qu'au moment où le bloc de code sera exécuté. Tant que ce code n'est pas exécuté, la variable "réceptrice" de la fonction ne contient rien d'autre que "undefined". Je n'aborderai pas ici les impacts liés à cette différence de fonctionnement. Vous trouverez plus d'infos sur ce sujet dans le support de cours « Javascript et HTML5 ».

D'un point de vue de l'utilisation effective, les deux variables de ma fonction renvoient strictement les mêmes résultats :

```
> mafonction(1, 2);
< 300

> mafonction('toto', 2)
< "message : toto,2"
```

On voit que pour exécuter une fonction, il suffit de saisir son nom et de lui transmettre entre parenthèses les paramètres qu'elle attend. Il faut faire attention à transmettre les paramètres dans l'ordre dans lequel ils sont déclarés dans le code source de la fonction.

Comme le Javascript est un langage faiblement typé, les paramètres des fonctions le sont aussi, aussi il est plutôt bien vu d'être parano, et de bien tester la validité des paramètres transmis avant de s'en servir. Dans l'exemple de « mafonction », vous avez vu que j'ai testé le type des paramètres (avec le mot clé « typeof ») pour déterminer de quelle manière j'allais utiliser les paramètres reçus. C'est une manière de faire, assez efficace au demeurant. Mais laissons ça de côté.

Il y a une question qui vous turlupine peut être : dans quel cas faut-il utiliser les fonctions ? Eh bien, à peu près tout le temps. En fait, dès que vous commencez à dupliquer du code, c'est généralement le bon moment pour vous dire que vous feriez mieux de faire un break, de prendre un peu de recul, et de commencer à « refactoriser » le code (comme disent les pros).

Je vous propose de reprendre l'exemple vu au chapitre 2.4.3, dans lequel nous générions une liste HTML à partir d'un tableau Javascript. Voici une transposition sous forme de fonction, du code que nous avons étudié précédemment :

```
var gen_liste = function (lang_liste, cible) {
    var maliste = document.createElement('ul');

    lang_liste.forEach(function(element, item) {
        var tmp_txt_element = document.createTextNode(element);
        var tmp_li_element = document.createElement('li');
        tmp_li_element.append(tmp_txt_element);
        maliste.append(tmp_li_element);
    });

    var macible = document.getElementById(cible);
    macible.append(maliste);
}

gen_liste(lang_tab, 'cible');
```

Vous voyez que j'ai profité de la réécriture du code pour mettre 2 éléments en paramètres :

- Le tableau Javascript contenant la liste des langages
- L'identifiant de l'élément cible dans lequel la liste HTML sera générée

En procédant ainsi, ma fonction « `gen_liste` » devient réutilisable pour de multiples usages. Supposons que nous disposions d'un tableau Javascript référençant une liste d'albums de bande dessinées, et que nous souhaitions générer la liste HTML correspondante dans une « `div` » qui a pour identifiant « `bd` », voilà ce que cela pourrait donner :

```
var albums_bd = ['Garulfo', 'Bone', 'Travis', 'Sillage', 'Nausicaa'];
gen_liste(albums_bd, 'bd');
```

Voilà, pas besoin de réécrire une nouvelle version de la fonction « `gen_liste` », à partir du moment où la structure du tableau Javascript des bandes dessinées est similaire à la structure du tableau des langages de programmation, ça va rouler tout seul 😊.

En revanche, on peut « blinder » le code pour le rendre plus robuste, dans un contexte de production :

```
var gen_liste = function (lang_liste, cible) {
    var macible = document.getElementById(cible);

    if (macible == undefined) {
        throw new Error('cible non trouvée');
    }

    if (typeof lang_liste != 'object' || lang_liste.length == undefined) {
        throw new Error('paramètre 1 pas de type array');
    }

    var maliste = document.createElement('ul');

    lang_liste.forEach(function(element, item) {
        var tmp_txt_element = document.createTextNode(element);
        var tmp_li_element = document.createElement('li');
        tmp_li_element.append(tmp_txt_element);
        maliste.append(tmp_li_element);
    });

    macible.append(maliste);
}
```

Une fois que vous avez créé votre fonction « `gen_liste` », essayez de l'appeler via le code suivant :

```
gen_liste(albums_bd, 'cible');
```

Dans cette nouvelle version, j'ai placé le code de recherche de la « cible » au tout début, et si je m'aperçois que cette cible n'existe pas sur la page, je déclenche une erreur grave (avec le message « cible non trouvée »).

Juste après, je regarde si le paramètre « lang_liste » est bien de type « object » (le type « array » n'existe pas en Javascript, un tableau en JS est forcément de type « object »). Du coup je « double » le contrôle d'un second test vérifiant si le paramètre dispose de la propriété « length ». S'il n'en dispose, c'est que ce n'est définitivement pas un tableau, donc je déclenche une autre erreur grave (avec le message « paramètre 1 pas de type array »).

Bon, on peut trouver d'autres manières de blinder le code, mais ce n'est franchement pas mal en l'état.

Il y a sujet que j'ai passé sous silence jusqu'ici, c'est la notion de variable. Comment fonctionnent les variables qui sont déclarées à l'intérieur d'une fonction comme notre fonction gen_liste ? Si l'on regarde notre code, nous voyons que nous avons 4 variables, qui sont :

- macible, variable locale appartenant à la fonction « gen_liste »
- maliste, variable locale appartenant à la fonction « gen_liste »
- tmp_txt_element, variable locale appartenant à la fonction anonyme déclarée dans la méthode « forEach »
- tmp_li_element, variable locale appartenant à la fonction anonyme déclarée dans la méthode « forEach »

Ces 4 variables sont déclarées à l'intérieur de la fonction avec le mot clé « var ». On dit qu'elles sont « locales », ou pour être plus précis, on dit que leur « portée » est locale à la fonction dans laquelle elles sont déclarées. Certains développeurs préfèrent parler de « périmètre » pour parler de cette notion de « portée ». C'est la même chose. Les anglais emploient généralement le terme de « scope » pour désigner cette notion de « portée ».

Dans le cas de nos 4 variables, on voit qu'elles n'ont pas toutes la même portée, car 2 variables sont dans le « scope » de la fonction « gen_liste », alors que 2 autres variables sont dans le « scope » de la fonction anonyme déclarée à l'intérieur de la fonction « gen_liste ».

On notera plusieurs points qui me semblent importants :

- si une variable « macible » a été déclarée en dehors de la fonction « gen_liste », cette variable est complètement ignorée par la fonction « gen_liste », qui utilise sa propre variable « macible ».
- la variable « macible » peut être utilisée par la fonction anonyme qui est déclarée à l'intérieur de la fonction « gen_liste ». Dans le cas présent, c'est une autre variable de la fonction « gen_liste » qui est utilisée par la fonction anonyme, à savoir la variable « maliste ».

Les fonctions en Javascript constituent un vaste sujet que nous n'avons fait qu'effleurer ici. Pour l'approfondir, je vous recommande la lecture du support « Javascript et HTML5 » qui se trouve dans le même dépôt Github que le présent support.

2.6.1 Le mode « strict » est ton ami

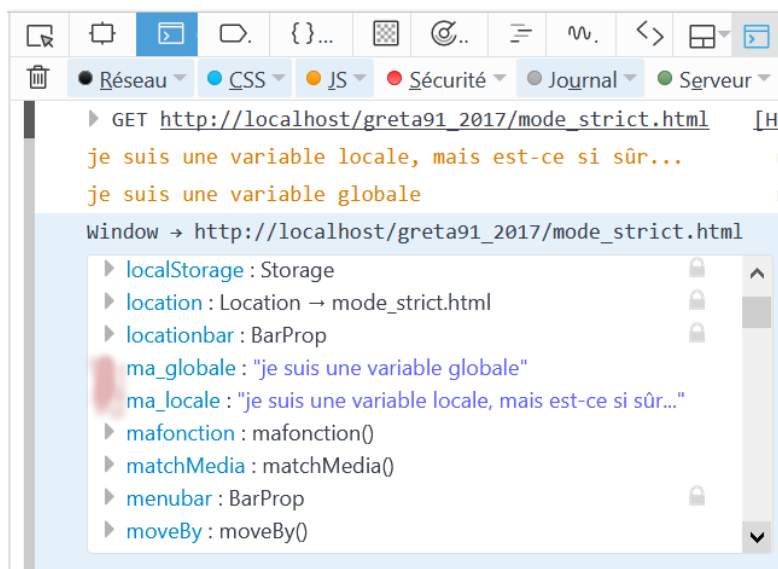
Que se passe-t-il si, à l'intérieur d'une fonction, j'essaie d'utiliser une fonction qui n'est pas locale à la fonction, et que j'ai oublié de déclarer en dehors de la fonction, c'est-à-dire de manière globale. Vous seriez tenté de vous dire que ça va planter, pas vrai ? Eh bien, non !!!

Pour voir ce qui se passe, je vous propose de tester le bout de code suivant, que vous pouvez placer dans la page HTML de votre choix :

```
<script>
    ma_globale = "je suis une variable globale";

    function mafonction() {
        ma_locale = 'je suis une variable locale, mais est-ce si sûr...';
        console.log(ma_locale);
        console.log(ma_globale);
    }
    mafonction();
    console.dir(window);
</script>
```

Exécutez ce code et observez ce que nous dit la console de Firefox :



On s'aperçoit que l'objet « window » contient les 2 variables « ma_globale » et « ma_locale ».

Qu'est ce que c'est que ce bazar ? J'allais même écrire « what the ... », mais je n'ai pas osé, des enfants pourraient lire ce document, sait-on jamais 😊.

Bon, comment se fait-il que l'objet « window » soit pollué par ces 2 variables « ma_globale » et « ma_locale ». A la limite, pour « ma_globale », je peux comprendre... étant donné que j'ai déclaré cette variable sans le mot clé « var », il fallait bien que le navigateur raccroche cette variable à quelque chose, et il se trouve que par défaut, c'est l'objet « window » qui se trouve être le réceptacle de toutes les brebis égarées :

```
ma_globale = "je suis une variable globale";
```

Bon admettons, mais en ce qui concerne la variable « ma_locale », utilisée dans la fonction « mafonction ». Ah oui, j'ai oublié d'utiliser le mot clé « var » lors de la déclaration de cette variable. Du coup, rebelote, le navigateur considère que c'est une variable globale par défaut, et il la rattache à l'objet « window » qui se récupère une brebis égarée de plus.

La vache ! C'est le bazar, non ? Oui et non, car il y a une solution simple et élégante pour éviter de se retrouver dans cette situation. C'est d'utiliser le mode « strict ». Ce mode s'active très simplement en utilisant la syntaxe suivante :

```
"use strict";
```

C'est une sorte de drapeau, qui dit au navigateur : « mon p'tit gars, j't'aime bien, mais si tu veux qu'on continue à bosser ensemble, il va falloir que tu changes ». Et ce changement passe par l'utilisation du mode strict, mode qui est apparu avec la norme ECMAScript 5 (ES5).

Je vous propose de placer ce drapeau au début de notre code et de regarder ce qui se passe :

```
✖ ▶ ReferenceError: assignment to undeclared variable  
ma_globale [En savoir plus]
```

On voit que le navigateur ne tolère plus qu'on utilise la variable « ma_globale », alors qu'elle n'a pas été déclarée explicitement avec le mot clé « var ». OK, alors corrigeons et relançons :

```
"use strict";
```

```
var ma_globale = "je suis une variable globale";
```

```
✖ ▶ ReferenceError: assignment to undeclared variable  
ma_locale [En savoir plus]
```

Le navigateur ne tolère plus que j'essaie d'utiliser la variable « ma_locale » alors que je ne l'ai pas déclarée.

Je ne sais pas pour vous, mais ce mode strict me plaît bien.

Corrigeons, et relançons :

```
function mafonction() {
  var ma_locale = 'je suis une variable locale, c\'est sûr...';
  console.log(ma_locale);
  console.log(ma_globale);
}
```



Ah, c'est mieux !! 😊

On voit que l'objet « window » ne contient plus que la variable « ma_globale ». Donc la variable « ma_locale » est bien locale à la fonction « mafonction », elle ne vient plus polluer l'objet « window ».*

Vous comprenez sans peine pourquoi j'aime le mode strict. Je vous avoue que j'ai pris l'habitude de l'utiliser dans toutes mes fonctions, comme ceci :

```
function mafonction() {
  "use strict";
  var ma_locale = 'je suis une variable locale, maintenant c\'est sûr...';
  console.log(ma_locale);
  console.log(ma_globale);
}
```

En revanche, c'est quelquefois délicat de l'utiliser au niveau global, comme je l'ai fait au début de ce chapitre. Car, si vous faites appel à des bibliothèques de code développées par des tiers, et que ces bibliothèques n'ont pas été développées avec la rigueur imposée par le mode strict, ces bibliothèques

sont susceptibles de déclencher des plantages en pagaille. Si vous êtes dans ce cas, mieux vaut utiliser le mode strict dans vos propres fonctions, ce sera déjà très bien, et surtout cela vous épargnera pas mal d'erreurs, souvent difficiles à déceler.

On notera que l'impact du mode strict ne se cantonne pas uniquement à la déclaration des variables. Cela modifier le comportement de l'interpréteur JS sur d'autres aspects. C'est un sujet intéressant, mais pas prioritaire pour ce cours, aussi vous pourrez lire cet excellent article quand vous aurez le temps :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Strict_mode

2.6.3 Les variables sont musclées

Je vous avoue que j'ai hésité à insérer ce chapitre dans ce support de cours, mais le sujet qui suit est intéressant, et même amusant, alors je ne résiste pas à l'envie de vous le présenter.

Savez-vous que les variables JS sont musclées ? Si, sérieusement, elles passent leur temps à faire de l'escalade.

Pour bien comprendre, je vous propose l'exemple de fonction suivant :

```
function mafonction() {  
    "use strict";  
    var ma_locale = 'je suis une variable locale, maintenant c\'est sûr...';  
  
    var datas = ['java', 'javascript', 'php', 'python'];  
    for (var i=0, imax=datas.length; i<imax ; i++) {  
        console.log(datas[i]);  
    }  
  
    var test = 'je pense donc je suis';  
  
    var test2 = 'qui suis-je ?';  
  
    return test + ', mais ' + test2 ;  
}  
  
console.log(mafonction());
```

Vérifions dans la console que le code fonctionne :

```
java
javascript
php
python
je pense donc je suis, mais qui suis-je ?
```

OK. En lisant le code source de la fonction, vous vous dites peut être que les variables sont analysées et traitées au fil de l'eau par l'interpréteur JS... probablement dans l'ordre où elles sont déclarées dans le code. Eh bien vous avez tout faux.

En fait, les fonctions JS utilisent un mécanisme que nos amis anglophones désignent par le terme de « hoisting », que l'on pourrait traduire par « hissage »... ☹ bon disons plutôt « escalade ».

Donc imaginez que, au moment où la fonction commence son exécution, nos petites variables se hissent - avec leur petits bras musclés - au début de la fonction. Du coup, le code qui est réellement exécuté par l'interpréteur JS ressemble plutôt à ceci :

```
function mafonction() {
    "use strict";
    var ma_locale;
    var datas;
    var i, imax;
    var test, test2;

    ma_locale = 'je suis une variable locale, maintenant c\'est sûr...';

    datas = ['java', 'javascript', 'php', 'python'];
    for (i=0, imax=datas.length; i<imax ; i++) {
        console.log(datas[i]);
    }

    test = 'je pense donc je suis';

    test2 = 'qui suis-je ?';

    return test + ', mais ' + test2 ;
}

console.log(mafonction());
```

On constate que toutes les variables sont déclarées au début du code de la fonction, de manière implicite, sans qu'on ait notre mot à dire :

```
var ma_locale;  
var datas;  
var i, imax;  
var test, test2;
```

Les variables sont déclarées, mais pas typées, le typage viendra plus tard, au moment où le contenu de la variable sera précisé. Par exemple, si on prend la variable « test2 » et qu'on ajoute quelques mouchards devant et derrière :

```
console.log(test2);    // renvoie "undefined"  
console.log(typeof test2); // renvoie "undefined"  
    var test2 = 'qui suis-je ?';  
console.log(test2);    // renvoie "qui suis-je ?"  
console.log(typeof test2); // renvoie "string"
```

On voit que la variable « test2 » existe avant même qu'on ait commencé à l'utiliser, simplement au début elle est de type « undefined » (indéfinie). Après son utilisation effective, son type passe à « string », ce qui est conforme au contenu qu'on lui a assignée.

Cette notion de « hoisting » n'a pas d'incidence stratégique, vous pouvez l'oublier si vous voulez... mais je vous recommande quand même de la garder dans un coin de votre mémoire, car elle pourra vous permettre de comprendre plus facilement certains comportements du langage, et de ne pas partir dans une analyse erronée, s'il vous arrive de buter sur un bug un peu coriace.

2.7 Vidage d'un élément

On a parfois besoin de « vider » une « div » de son contenu. Le truc pour le faire, c'est de créer un pointeur sur cet élément (si l'élément a un identifiant c'est hyper facile), puis d'utiliser la propriété « innerText » ou « innerHTML » et d'y insérer une chaîne de caractères vides.

Exemple :

```
var div_a_vider = document.getElementById('xxx');  
div_a_vider.innerText = '';
```

La méthode ci-dessus est efficace et sans appel.

Mais on peut aussi le faire en pur Javascript, c'est juste un peu plus verbeux :

```
var div_a_vider = document.getElementById('xxx');  
while (div_a_vider.firstChild) {  
    div_a_vider.removeChild(div_a_vider.firstChild);  
}
```

C'est amusant comme technique non ? On utilise une boucle qui « recherche » le premier nœud enfant de la cible (via la propriété « firstChild »). A l'intérieur de la boucle, on utilise la méthode « removeChild » associée à notre cible, méthode à laquelle on passe le premier « nœud enfant » associé à la cible. Si notre cible contient 10 « nœuds enfants », la boucle va « itérer » 10 fois, avant de s'arrêter, faute de combattants (tous les « nœuds enfants » ayant été supprimés).

Si vous ne retenez pas cette seconde méthode, ce n'est pas grave, la première fonctionne très bien. Quand vous aurez acquis plus de pratique dans la manipulation du DOM, la seconde méthode vous semblera plus accessible, et sans doute plus logique.

Tiens, j'en profite pour vous proposer un petit exercice : fort de ce que tout ce que vous savez maintenant, essayez de trouver une méthode pour « vider » le contenu de la page, et le remplacer par le message « GAME OVER ». Prenez le temps de faire des tests, il existe plusieurs solutions possibles à ce problème.

Pour que vous ne soyez pas tenté de regarder trop vite la solution, je la place sur la page suivante.

Ca y est, vous avez trouvé ?

Bon, voici une première solution possible :

```
> var overgame = document.querySelector('body');  
< undefined  
  
> overgame.innerText = 'GAME OVER';  
< "GAME OVER"
```

Autre solution possible :

```
> var overgame = document.getElementsByTagName('body');  
< undefined  
  
> overgame[0].innerHTML = 'GAME OVER';  
< "GAME OVER"
```

Vous vous rappelez que la méthode « `getElementsByTagName` » renvoie 1 ou plusieurs éléments dans une `HTMLCollection`. Comme il existe forcément un et un seul élément « `body` » sur la page, c'est le poste « 0 » de la liste « `overgame` » qui nous intéresse ici.

Bon, une dernière solution pour la route ?

```
> var div_a_vider = document.querySelector('body');  
  while (div_a_vider.firstChild) {  
    div_a_vider.removeChild(div_a_vider.firstChild);  
  }  
  div_a_vider.appendChild(document.createTextNode('GAME OVER'));
```

2.8 Un petit tour dans les « events »

Je vous propose dans ce chapitre une rapide introduction aux « événements » (en anglais « events »), ou plus exactement aux « écouteurs d'événements ».

Nous allons continuer à jouer avec notre liste de langages. Pour découvrir le sujet, je vous propose d'ajouter un écouteur d'événement de type « click » sur la « div » dans laquelle se trouve notre liste :

```
> var macible = document.getElementById('cible');
< undefined
> macible
< ▶<div id="cible">...</div>
> function test_macible(evt) {
    console.dir(evt);
  }
< undefined
> macible.addEventListener('click', test_macible, false);
< undefined
```

Explication :

- on commence par créer une variable « macible » définissant un pointeur sur un élément de la page qui a l'id « cible » (en l'occurrence il s'agit d'une « div »)
- on voit en tapant « macible » dans la console qu'il s'agit effectivement d'une « div »
- on crée une fonction « test_macible » qui reçoit un paramètre « evt » qu'on va afficher dans la console du navigateur
- on ajoute sur la variable « macible » un écouteur d'événement (un « event listener ») de type « click », qui va avoir pour effet d'appeler la fonction « test_macible » dès que le navigateur va détecter que l'événement en question se produit.

Cette méthode « addEventListener » est disponible sur tous les nœuds du DOM, et nous pouvons l'utiliser avec différents types d'événements. Très souvent, on utilisera un événement de type « click » pour un bouton, un événement de type « change » pour détecter un changement de valeur sur un champ de formulaire, etc... On notera qu'il existe une série d'événement dont le nom commence par « drag » (pour détecter des événements de type « drag and drop »), certains événements ont un nom qui commence par « touch » (pour les écrans tactiles), ou par « mouse » (pour les événements liés à la souris), etc...

Dans l'exemple qui précède, j'ai utilisé une fonction créée explicitement (« test_macible »), que j'ai associée à l'écouteur d'événement. J'aurais aussi pu utiliser la syntaxe suivante, utilisant une fonction anonyme :

```
ma_cible.addEventListener('click', function(evt) { console.dir(evt) }, false);
```

Elle n'est pas forcément évidente à lire, peut être préférerez-vous la version ci-dessous, strictement équivalente, mais dont le code est présenté sur plusieurs lignes :

```
ma_cible.addEventListener(  
    'click',  
    function(evt) {  
        console.dir(evt)  
    },  
    false  
);
```

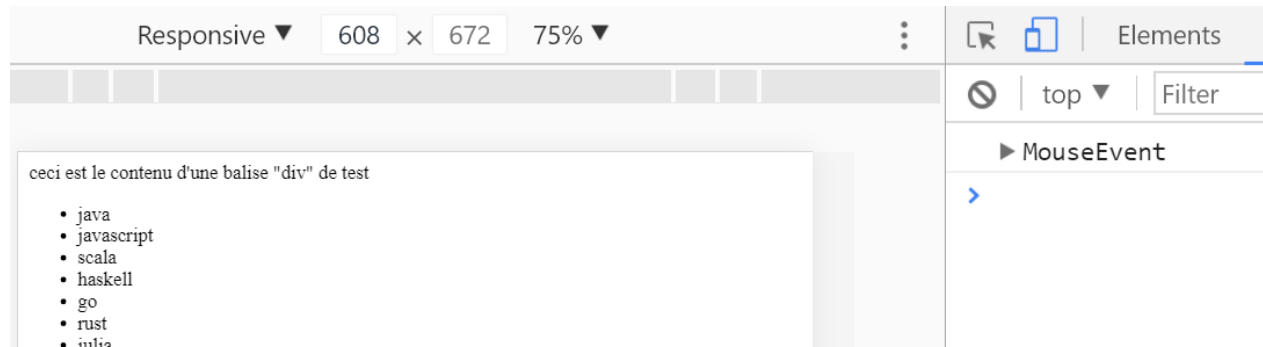
On voit que la méthode « addEventListener » attend trois paramètres qui sont :

- Le type d'événement (ici l'événement de type « click »)
- Le nom d'une fonction (comme dans le premier exemple), ou le code d'une fonction anonyme (comme dans l'exemple ci-dessus).
- Un booléen, qu'on paramètrera toujours à « false » (sauf cas très particulier, et plutôt rare, sur lequel je ne m'étendrai pas ici car cela compliquerait inutilement les choses).

Dans la fonction anonyme ci-dessus, j'ai placé un appel à la fonction Javascript « console.dir », qui va nous être utile, car elle va gentiment « moucharder » sur le contenu de la variable « evt », variable qui est reçue par la fonction anonyme. Ce paramètre attendu par la fonction anonyme utilisé sur un « addEventListener » fait partie des conventions du Javascript. Nous allons voir qu'il contient des informations très utiles, mais notez que vous pouvez lui donner le nom que vous voulez. Ici, je l'ai appelée « evt », mais on l'appelle aussi assez souvent « e », ou encore « event ». A vous de voir, c'est une question de goût...

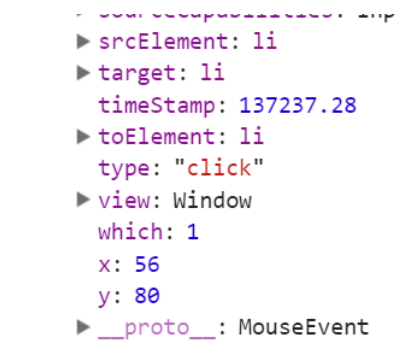
Donc si vous avez bien mis en place votre écouteur d'événement, le fait de cliquer sur n'importe quel élément de la liste doit faire apparaître des informations intéressantes dans la console du navigateur.

Si par exemple je clique sur le langage « Scala » :



... je vois apparaître dans la console à droite, une ligne « MouseEvent ».

Déplions cette ligne. On voit qu'il y a beaucoup d'informations intéressantes, mais c'est surtout la fin de la liste qui m'intéresse :



Avez-vous vu ce que nous indique la propriété « target » ? Il semble que la cible de mon clic soit un élément de type « li ». Je déplie cette propriété « target » et je trouve les informations suivantes :

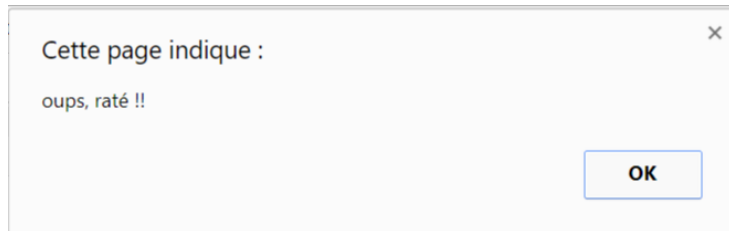
```
▼ target: li
  accessKey: ""
  assignedSlot: null
  ▶ attributes: NamedNodeMap
    baseURI: "file:///C:/Users/grego/Desktop/test.html"
    childElementCount: 0
    ▶ childNodes: NodeList(1)
    ▶ children: HTMLCollection(0)
    ▶ classList: DOMTokenList(0)
      className: ""
      clientHeight: 18
      clientLeft: 0
      clientTop: 0
      clientWidth: 597
      contentEditable: "inherit"
    ▶ dataset: DOMStringMap
      dir: ""
      draggable: false
    ▶ firstChild: text
      firstElementChild: null
      hidden: false
      id: ""
      innerHTML: "scala"
      innerText: "scala"
      isConnected: true
      isContentEditable: false
      lang: ""
```

On voit que la propriété « innerHTML » contient « scala », soit le langage sur lequel j'ai cliqué il y a un instant. Vous voyez que tout se tient, la console Javascript est notre amie, elle nous dit tout, absolument tout : notre page, ses éléments HTML et ses événements, n'ont aucun secret pour nous.

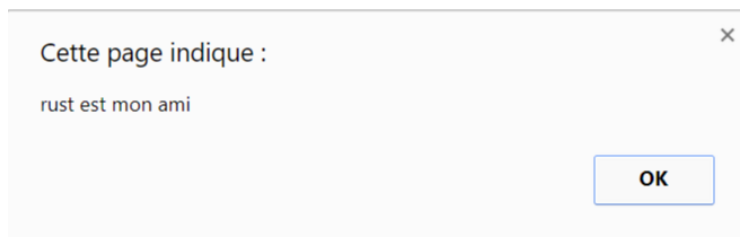
Je vous propose maintenant de modifier notre écouteur d'événement de la façon suivante :

```
ma_cible.addEventListener(
  'click',
  function(e) {
    if (e.target.tagName == 'LI') {
      alert(e.target.innerHTML + ' est mon ami');
    } else {
      alert('oups, raté !!');
    }
  },
  false
);
```

Si maintenant je clique en dehors de la liste HTML, j'obtiens ce message :



... et si je clique sur un élément de la liste, par exemple le langage « rust », j'obtiens ceci :



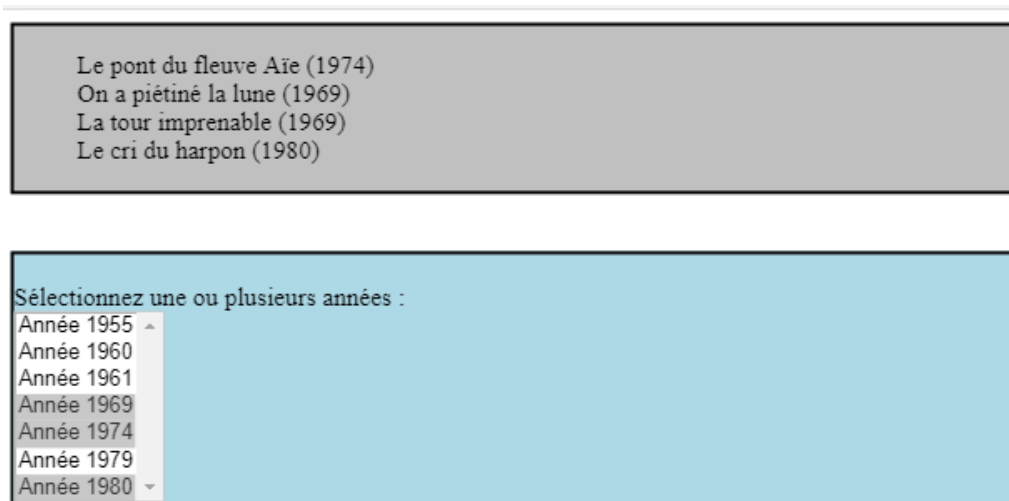
La technique que nous venons d'étudier va nous être utile dans le chapitre qui vient.

2.9 Filtre dynamique

Dans ce chapitre, nous allons développer un petit filtre permettant de sélectionner des films par année de sortie. Cela va nous permettre en pratique de nombreuses techniques vues dans les chapitres précédents, et cela va nous permettre aussi d'aborder quelques techniques complémentaires qu'il est intéressant de connaître.

Je n'ai pas cherché à créer une vraie liste de films, j'ai plutôt imaginé une liste de films bidons, en parodiant des titres de vrais films. Vous avez le droit de vous amuser à retrouver les vrais films cachés derrière ces titres bidons, mais méfiez-vous, les années de sortie sont vraiment bidons, et ne vous aideront en aucune manière.

Concrètement, voici un prototype de ce que j'aimerais obtenir :



Explication :

- Dans le cadre au fond gris, j'ai une liste de films, cette liste est directement impactée par les années de diffusion sélectionnées dans le cadre au fond bleu
- Dans le cadre au fond bleu, j'ai un champ de saisie de type « select » permettant d'effectuer des sélections multiples. Ce champ contient une liste d'années, dès que je modifie la sélection, la liste de films doit être réactualisée en temps réel

Il me faut donc une liste de films, avec pour chaque film, son titre et son année de sortie.

Voici cette liste de films, sous une forme que nous n'avions pas encore vue dans ce support de cours :

```
var films_fantaisie = [  
  {titre: 'Le jour où la terre s\'emballa', annee: 1955},  
  {titre: 'L\'attaque des morts poilants', annee: 1960},  
  {titre: 'Le pont du fleuve Aïe', annee: 1974},  
  {titre: 'On a piétiné la lune', annee: 1969},  
  {titre: 'Le bon, la brute et le mécréant', annee: 1979},  
]
```

```
{titre:'La tour imprenable', annee:1969},  
{titre:'Le cri du harpon', annee:1980}  
];
```

Il s'agit bien d'un tableau (les crochets en témoignent), mais un tableau d'objets. En effet, chaque occurrence du tableau est un objet Javascript, par exemple :

```
{titre:'Le jour où la terre s\'emballa', annee:1955}
```

Il s'agit ici d'un objet ayant 2 propriétés :

- La propriété « titre », qui contient une chaîne de caractères
- La propriété « annee », qui contient une valeur numérique entière

Si j'avais voulu déclarer une variable de type objet contenant un seul film, j'aurais pu écrire ceci :

```
monfilm = {titre:'Le jour où la terre s\'emballa', annee:1955};
```

... ou encore cela, qui est strictement équivalent :

```
monfilm = {};  
monfilm.titre = 'Le jour où la terre s\'emballa';  
monfilm.annee = 1955;
```

Donc notre tableau « films_fantaisie » est un regroupement d'objets ayant tous la même structure (soit les 2 propriétés « titre » et « annee »). C'est un exemple typique de jeu de données structurées, que l'on aurait pu extraire d'une base de données, et envoyer à un navigateur via une requête AJAX. Je ne m'étends pas sur le sujet, ce qui m'intéresse ici, c'est la structure du jeu de données, et la manière dont je peux l'exploiter en Javascript.

Si l'on reprend la fonction `gen_liste()` étudiée dans un précédent chapitre, nous avons très peu de choses à modifier pour qu'elle soit en mesure de générer notre liste de films. J'ai indiqué en rouge les parties que j'ai modifiées :

```
var gen_liste_films = function (lang_liste, cible) {
  var macible = document.getElementById(cible);
  if (macible == undefined) {
    throw new Error('cible non trouvée');
  }
  if (typeof lang_liste != 'object' || lang_liste.length == undefined) {
    throw new Error('paramètre 1 pas de type array');
  }
  var maliste = document.createElement('ul');
  maliste.setAttribute('data-id', 'films');
  lang_liste.forEach(function(element, item) {
    var tmp_txt_element = document.createTextNode(
      element.titre + ' (' + element.annee + ')');
    var tmp_li_element = document.createElement('li');
    tmp_li_element.setAttribute('data-year', element.annee);
    tmp_li_element.append(tmp_txt_element);
    maliste.append(tmp_li_element);
  });

  macible.append(maliste);
}
```

Premier changement, l'ajout d'un attribut « data-id » sur notre balise « ul » :

```
maliste.setAttribute('data-id', 'films');
```

... ce sera en effet plus pratique de manipuler notre liste, si elle dispose d'un identifiant permettant de l'isoler facilement. J'aurais pu écrire ceci :

```
maliste.setAttribute('id', 'films');
```

... mais je ne veux pas utiliser le traditionnel attribut « id » car il y a toujours un risque de prendre par accident un identifiant déjà utilisé (et je rappelle que tout identifiant HTML doit être unique). L'attribut « data-id » est un attribut personnalisé, c'est une nouveauté du HTML5, qui nous autorise à créer autant d'attributs de ce type que l'on souhaite. C'est « open bar », il suffit de préfixer toutes vos données personnalisées par « data- ». C'est surtout très pratique pour stocker les données « métier » dont nous avons besoin dans notre code HTML.

Second changement important, je crée un « text node » contenant le titre du film, suivi de l'année de sortie placée entre parenthèses :

```
var tmp_txt_element = document.createTextNode(
    element.titre + ' (' + element.annee + ')'
);
```

Cela nous permettra d'obtenir des titres formatés de cette manière :

```
Le pont du fleuve Aïe (1974)
On a piétiné la lune (1969)
La tour imprenable (1969)
Le cri du harpon (1980)
```

Troisième changement : j'ajoute à chaque balise « li » un attribut « data-year » qui contiendra l'année de sortie de chaque film :

```
tmp_li_element.setAttribute('data-year', element.annee);
```

Grâce à cet attribut « data-year », il sera facile de faire correspondre les années de sortie sélectionnées dans le champ de saisie, avec les années de sortie des films de la liste.

Pour les besoins de mon prototype, j'ai besoin de 2 « div » distinctes. La première « div » qui a pour identifiant « cible », sera vide. La liste des films y sera « injectée » dynamiquement. La seconde « div » contient le champ de saisie de type « select » définissant la liste des années de sortie de notre filtre de sélection :

```
<div id="cible" style="border-style: solid; background-color: silver;"></div>
<br><br>
<div style="border-style: solid; background-color: lightblue;">
  <br>
  <label>Sélectionnez une ou plusieurs années :<br>
  <select id="filtre-annees" multiple size="7">
    <option value="1955">Année 1955</option>
    <option value="1960">Année 1960</option>
    <option value="1961">Année 1961</option>
    <option value="1969">Année 1969</option>
    <option value="1974">Année 1974</option>
    <option value="1979">Année 1979</option>
    <option value="1980">Année 1980</option>
  </select>
</label>
  <br>
</div>
```

J'avais tout d'abord pensé créer le champ de type « select » via du code Javascript, mais après réflexion je n'ai pas souhaité compliquer ce tuto. Je préfère que nous nous concentrons sur la

logique de sélection des films.

Toutes les pièces du puzzle son en place, nous sommes maintenant en mesure de générer notre liste de films :

```
gen_liste_films(films_fantaisie, 'cible');
```

Vous devriez obtenir une liste se présentant de la façon suivante :

- Le jour où la terre s'emballa (1955)
- L'attaque des morts poilants (1960)
- Le pont du fleuve Aïe (1974)
- On a piétiné la lune (1969)
- Le bon, la brute et le mécréant (1979)
- La tour imprenable (1969)
- Le cri du harpon (1980)

A ce stade, il est intéressant d'utilisateur l'inspecteur d'élément pour étudier la structure HTML obtenue :

```
▼ <div id="cible" style="border-style: solid; background-color: silver;">  
  ▼ <ul data-id="films"> == $0  
    <li data-year="1955">Le jour où la terre s'emballa (1955)</li>  
    <li data-year="1960">L'attaque des morts poilants (1960)</li>  
    <li data-year="1974">Le pont du fleuve Aïe (1974)</li>  
    <li data-year="1969">On a piétiné la lune (1969)</li>  
    <li data-year="1979">Le bon, la brute et le mécréant (1979)</li>  
    <li data-year="1969">La tour imprenable (1969)</li>  
    <li data-year="1980">Le cri du harpon (1980)</li>  
  </ul>  
</div>
```

Vous remarquez la présence de l'attribut « data-id » sur la balise « ul », et de l'attribut « data-year » sur chaque balise « li ». Avec ces informations cachées dans le code HTML, il va être relativement facile de mettre en place la mécanique du filtre de sélection.

Nous devons maintenant définir l'écouteur d'événement qui va nous permettre de détecter toute modification du filtre correspondant au champ de type « select ». Cette balise « select » a pour identifiant « filtre-annees », alors nous pouvons l'identifier via la méthode « getElementById » et appliquer un « addEventListener » :

```
var filtre = document.getElementById('filtre-annees') ;  
filtre.addEventListener('change', filterdyn, false);
```

Pour le second paramètre de la méthode « `addEventListener` », j'ai préféré indiquer le nom d'une fonction (que j'ai appelée « `filterdyn` »), plutôt que de placer ici une fonction anonyme. La raison principale, c'est que le code va être un peu plus complexe que dans nos exemples précédents, du coup je préfère l'isoler dans une fonction spécifique.

Voici le code source de la fonction « `filterdyn` ». J'y ai placé de nombreux commentaires pour que vous puissiez vous approprier cet exemple :

```
function filterdyn(evt) {  
    // récupération des options de la balise "select"  
    var options = evt.target.options ;  
    // détermination des options sélectionnées par l'utilisateur  
    var selected = [] ;  
    for (var iopt = 0, ioptmax = options.length ; iopt < ioptmax ; iopt+=1 ) {  
        if (options[iopt].selected) {  
            selected.push(options[iopt].value) ;  
        }  
    }  
    // identification de la liste des films  
    var table_link = document.querySelectorAll('[data-id=films] > li') ;  
    // pour chaque film de la liste, on détermine si sa date de sortie  
    // correspond à la sélection de l'utilisateur  
    for (var ilink = 0, ilinkmax = table_link.length ;  
        ilink < ilinkmax ; ilink +=1) {  
        var annee = table_link[ilink].dataset.year;  
        // recherche de l'année dans le tableau selected  
        if (selected.indexOf(annee) !== -1) {  
            // l'année du film correspond à la sélection alors on l'affiche  
            table_link[ilink].style.display='list-item' ;  
        } else {  
            // l'année de sortie ne correspond pas, alors on cache le film  
            table_link[ilink].style.display='none' ;  
        }  
    }  
};
```

Vous noterez de quelle manière on affiche ou cache un élément de la liste HTML, cette technique pourra vous resservir dans d'autres contextes.

Vous noterez également l'utilisation de la méthode « `indexOf` » pour effectuer une recherche dans le tableau « `selected` ».

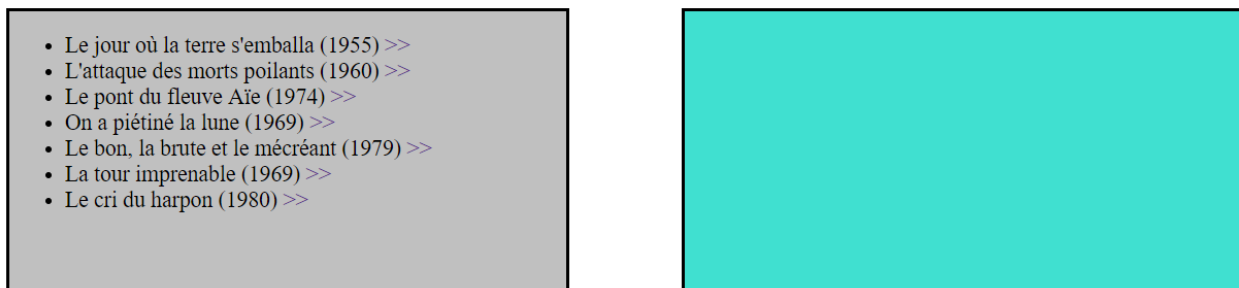
Vous noterez aussi l'utilisation de la méthode « `querySelectorAll` », et le fait qu'on peut l'utiliser pour rechercher un élément ayant un attribut « `data-id` » par exemple.

Dans le chapitre 4.4 en annexe, vous trouverez une variante du code que nous venons étudier. Dans cette variante, les options du champ « select » sont générées dynamiquement en fonction des années de sortie définies dans le tableau des films.

2.10 Téléportation

Dans ce chapitre, nous allons apprendre à déplacer un élément dans le DOM. Vous allez voir, c'est amusant à faire, on dirait quasiment de la téléportation. Comment, personne ne vous avait prévenu du fait que vous alliez étudier de la physique quantique dans ce cours ? 😊

Pour les besoins de l'exercice, on va définir 2 « div », une cadrée à gauche et l'autre cadrée à droite. La « div » de gauche contiendra notre liste de films (celle du chapitre précédent), tandis que la « div » de droite sera vide au départ, et recevra les films « téléportés » de la liste de droite, chaque fois qu'on cliquera sur une flèche se trouvant à droite du titre de chaque film. Concrètement cela doit ressembler à ça :



En cliquant sur les flèches se trouvant à droite du titre d'un film, ce dernier se trouve téléporté dans la liste de droite :



Ok, commençons par définir nos 2 div :

```
<div id="liste1"></div>  
<div id="liste2"></div>
```

Pour que nos « div » ressemblent à quelque chose, nous devons ajouter un peu de CSS dans l'entête de la page :

```
#liste1 {
    border-style: solid;
    background-color: silver;
    width:400px;
    height:200px;
    float:left;
}
#liste2 {
    border-style: solid;
    background-color: turquoise;
    width:400px;
    height:200px;
    float:right;
}
```

Pour la génération de la liste des films dans la « div » de gauche, je vous propose de réutiliser à peu près le même code que dans le chapitre précédent :

```
var gen_liste_films = function (lang_liste, cible) {
    var macible = document.getElementById(cible);

    var maliste = document.createElement('ul');
    maliste.setAttribute('data-id', 'films');

    lang_liste.forEach(function(element, item) {
        // Création du lien cliquable déclencheur de la téléportation
        var tmp_link = document.createElement('a');
        tmp_link.append(document.createTextNode(' >> '));
        tmp_link.setAttribute('href', '#');
        tmp_link.setAttribute('onclick', 'teleportation(this);');
        tmp_link.style['text-decoration'] = 'none'; // pas de soulignement

        // TextNode contenant le titre de chaque film
        var tmp_txt_element = document.createTextNode(
            element.titre + ' (' + element.annee + ') '
        );

        // Création de l'élément "li" correspondant à un film
        var tmp_li_element = document.createElement('li');
        tmp_li_element.setAttribute('data-year', element.annee);
        tmp_li_element.append(tmp_txt_element); // ajout du titre du film
        tmp_li_element.append(tmp_link); // ajout du lien de téléportation
        maliste.append(tmp_li_element);
    });

    macible.append(maliste);
}
```

Attention, j'ai bien dit que c'était « à peu près » le même code. J'ai en effet supprimé les

contrôles que j'avais placés au début de la fonction, dans le chapitre précédent, afin de simplifier un peu l'exemple. Je l'ai fait pour des raisons pédagogiques, pour simplifier le code afin que l'on se concentre sur d'autres aspects qui me semblent plus importants ici.

Si la mécanique de génération de chaque élément « li » est à peu près la même, vous remarquerez que j'ai quand même ajouté un peu de code permettant de générer la balise « a » définissant la « double flèche » (>>) cliquable :

```
// Création du lien cliquable déclencheur de la téléportation
var tmp_link = document.createElement('a');
tmp_link.append(document.createTextNode(' >> '));
tmp_link.setAttribute('href', '#');
tmp_link.setAttribute('onclick', 'teleportation(this);');
tmp_link.style['text-decoration'] = 'none'; // pas de soulignement
```

Cette balise « a », nous devons l'ajouter à notre élément « li », cet ajout est réalisé via la méthode « append », un peu plus loin à l'intérieur de la boucle :

```
tmp_li_element.append(tmp_link); // ajout du lien de téléportation
```

Concernant le lien lui-même, vous remarquerez que j'ai utilisé une astuce permettant de supprimer le soulignement sous les double-flèches :

```
tmp_link.style['text-decoration'] = 'none'; // pas de soulignement
```

... c'est simple et pratique, vous pourrez réutiliser cette astuce dans d'autres circonstances.

Mais le plus important, c'est l'ajout de la propriété « onclick », liée à une fonction Javascript « teleportation », fonction que nous allons créer dans un instant.

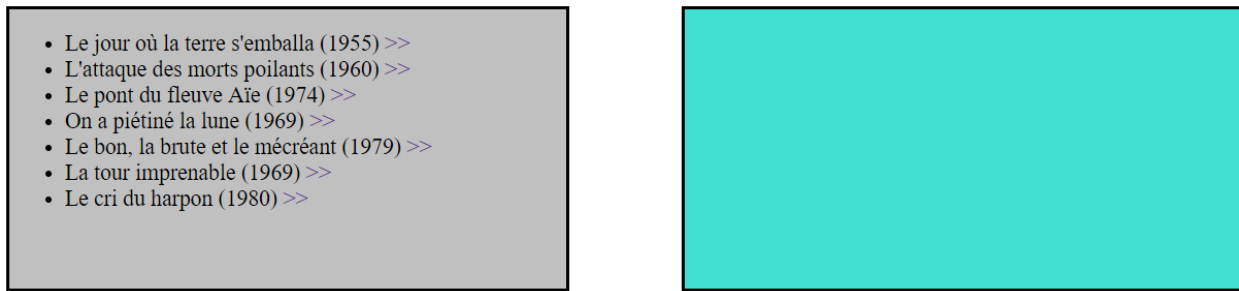
Tout n'est pas encore prêt, mais à ce stade, nous sommes déjà en mesure de faire fonctionner notre page, et de voir notre liste se générer. Ajoutez le code suivant et... action !!!

```
var films_fantaisie = [
  {titre:'Le jour où la terre s\'emballa', annee:1955},
  {titre:'L\'attaque des morts poilants', annee:1960},
  {titre:'Le pont du fleuve Aïe', annee:1974},
  {titre:'On a piétiné la lune', annee:1969},
  {titre:'Le bon, la brute et le mécréant', annee:1979},
  {titre:'La tour imprenable', annee:1969},
  {titre:'Le cri du harpon', annee:1980}
];

var origine = "listel";

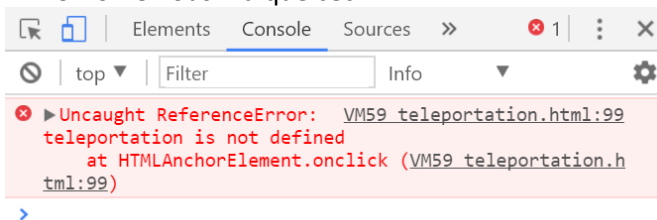
gen_liste_films(films_fantaisie, origine);
```

Si vous n'avez rien oublié, ça devrait déjà donner ceci :

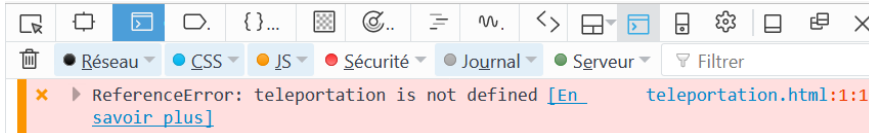


Mais pour l'instant, si vous cliquez sur les double-flèches de l'un des films, vous allez déclencher une erreur que vous pourrez voir dans la console (l'utilisateur lui n'en saura rien) :

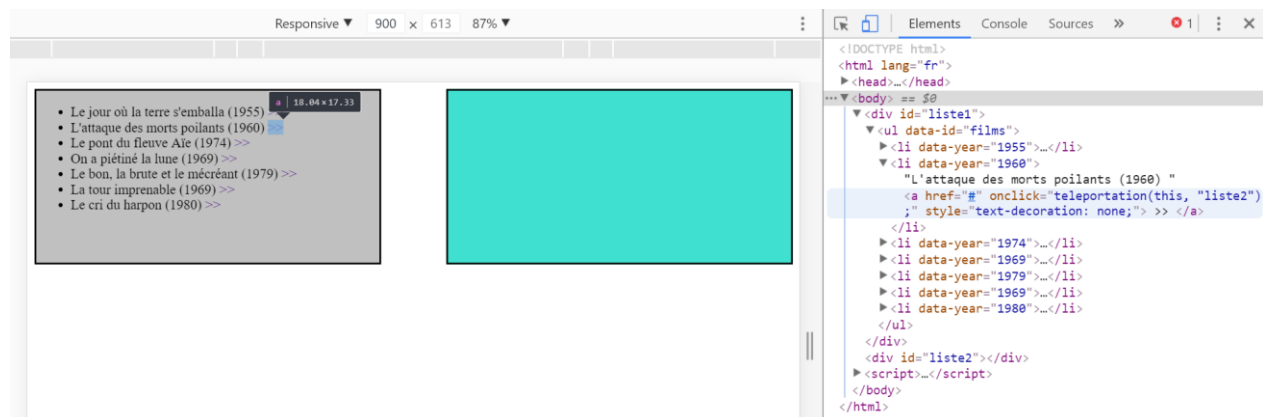
- Chrome nous indique ceci :



- Firefox ne nous dit pas autre chose :



Avant de créer la fonction manquante, prenons un moment pour regarder, avec l'inspecteur d'élément, à quoi ressemble notre liste HTML :



Je vous propose de zoomer sur l'un des films, pour voir de plus près ce que cela donne :

```
▼<li data-year="1960">
  "L'attaque des morts poilants (1960) "
  <a href="#" onclick="teleportation(this, "liste2")
  ;" style="text-decoration: none;"> >> </a>
</li>
```

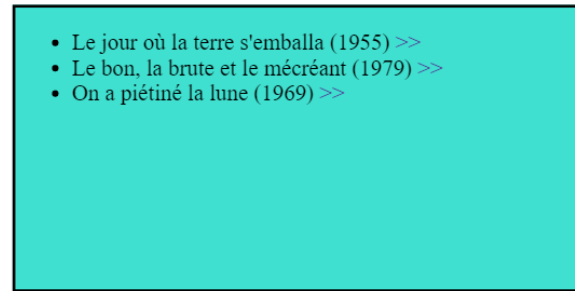
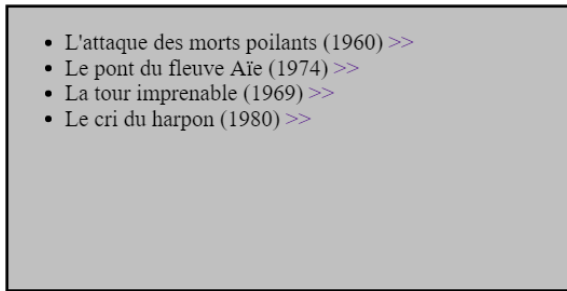
On voit que notre code Javascript a bien créé une balise « a », que cette balise a une propriété « onclick » qui va déclencher l'appel de la fonction « teleportation ». Cette fonction reçoit deux paramètres :

- this : c'est l'objet Javascript référençant la balise « a » sur laquelle l'utilisateur a cliqué. Cette donnée sera précieuse, à l'intérieur de la fonction « teleportation », elle nous permettra de récupérer facilement la balise « li » parente, pour la téléporter vers notre liste cible. Vous remarquerez que this n'est pas encadré par des guillemets, la présence de guillemets en ferait une simple chaîne de caractères, ce ne serait plus un objet Javascript et il serait dans ce cas inutilisable
- « liste2 » : là on a bien des guillemets, car il s'agit d'une simple chaîne de caractères, indiquant le nom de la « div » cible, qui sera réceptrice des films téléportés.

Le code de la fonction n'est pas très compliqué, mais il contient pas mal de petits détails. J'ai mis des commentaires sur les points importants :

```
function teleportation(that, cible) {
  var target = document.getElementById(cible);
  // on regarde s'il existe déjà une liste "ul" dans la "div" de destination
  var target2 = target.getElementsByTagName('ul');
  var target_list = null;
  // s'il n'existe pas de liste "ul", alors on en crée une
  if (target2.length == 0) {
    target_list = document.createElement('ul');
    target.append(target_list);
  } else {
    // s'il existe déjà une liste, on la sélectionne
    target_list = target2[0];
  }
  // on récupère le lien du noeud parent (soit la balise "li")
  // et on l'affecte à la liste "ul" cible
  target_list.append(that.parentNode);
};
```

Bon, ben il n'y a plus qu'à tester :



C'est pas mal, mais vous avez peut être remarqué qu'il y a un souci. En effet, les éléments « li » sont bien transférés dans la « div » de droite, mais ils contiennent un détail gênant : la double-flèche est toujours là. D'ailleurs, si vous cliquez sur les double-flèches dans la « div » de droite, vous constatez que l'ordre des éléments change. C'est un phénomène normal, la fonction « teleportation », toujours en action, prend l'élément considéré, et le replace à la fin de la liste. Bref, on tourne un peu en rond 😊.

Bon, il faut qu'on les « dégage », ces doubles-flèches. Mais comment faire ? Il y a certainement plusieurs solutions, plus ou moins élégantes. On ne va pas y passer la journée, alors je vous propose une solution qui me semble assez propre. Nous allons modifier légèrement la manière dont les données sont générées sur les balises « a » de la liste d'origine, en ajoutant un attribut « data-text ». Cet attribut nous permettra de conserver sur chaque balise « a » le titre du film, dans un format « propre ». L'objectif, c'est d'obtenir ceci :

```
▼ <li data-year="1955" data-text="Le jour où la terre s'emballa (1955)" >
  "Le jour où la terre s'emballa (1955) "
  <a href="#" onclick="teleportation(this, "liste2");" style="text-
  decoration: none;"> >> </a>
</li>
```

On voit dans cet exemple que le titre du film est toujours affiché à l'intérieur de la balise « a », mais qu'il est aussi stocké dans la propriété « data-text ». Il est « en clair », pas pollué par la balise « a » relative aux doubles flèches, ça va être très pratique pour corriger notre problème dans la phase de téléportation.

Pour que cela fonctionne, il nous faut modifier légèrement la boucle de génération des balises « li » qui se trouve dans la fonction « gen_liste_films ». Voici ce que cela donne (j'ai mis en « gris » les éléments modifiés) :

```
lang_liste.forEach(function(element, item) {
    // Création du lien cliquable déclencheur de la téléportation
    var tmp_link = document.createElement('a');
    tmp_link.append(document.createTextNode(' >> '));
    tmp_link.setAttribute('href', '#');
    tmp_link.setAttribute('onclick', 'teleportation(this, "liste2");');
    tmp_link.style['text-decoration'] = 'none'; // pas de soulignement

    // texte préformaté qui sera affiché et stocké dans un attribut data-text
    var texte = element.titre + ' (' + element.annee + ') ';

    // TextNode contenant le titre de chaque film
    var tmp_txt_element = document.createTextNode(texte);

    // Création de l'élément "li" correspondant à un film
    var tmp_li_element = document.createElement('li');
    tmp_li_element.setAttribute('data-year', element.annee);
    tmp_li_element.append(tmp_txt_element); // ajout du titre du film
    tmp_li_element.append(tmp_link); // ajout du lien de téléportation
    tmp_li_element.setAttribute('data-text', texte);
    maliste.append(tmp_li_element);
});
```

Dans la fonction « teleportation », on retouche légèrement le code qui se trouve à la fin, là encore j'ai mis en « gris » les quelques modifications :

```
function teleportation(that, cible) {
    var target = document.getElementById(cible);
    // on regarde s'il existe déjà une liste "ul" dans la "div" de destination
    var target2 = target.getElementsByTagName('ul');
    var target_list = null;
    // s'il n'existe pas de liste "ul", alors on en crée une
    if (target2.length == 0) {
        target_list = document.createElement('ul');
        target.append(target_list);
    } else {
        // s'il existe déjà une liste, on la sélectionne
        target_list = target2[0];
    }
    // on récupère le lien du noeud parent (soit la balise "li")
    // et on l'affecte à la liste "ul" cible
    var li_parente = that.parentNode;
    // à l'intérieur de la balise "li", on remplace le texte par celui stocké
    // dans l'attribut "data-text"
    li_parente.innerHTML = li_parente.getAttribute('data-text');

    target_list.append(li_parente);
};
```


Et voilà !!!

- L'attaque des morts poilants (1960) >>
- On a piétiné la lune (1969) >>
- Le bon, la brute et le mécréant (1979) >>
- Le cri du harpon (1980) >>

- Le jour où la terre s'emballa (1955)
- Le pont du fleuve Aïe (1974)
- La tour imprenable (1969)

Vous trouverez en annexe le code source complet de cet exercice.

Vous pouvez essayer de le modifier de manière à générer, dans la liste des films de la « div » de droite, des doubles-flèches pointant dans l'autre direction (<<), et permettant ainsi de « retéléporter » les films vers la « div » de droite.

3 Conclusion

Dans cette introduction au Javascript, j'ai bien évidemment fait l'impasse sur beaucoup de choses, pour me concentrer sur certaines notions qui me semblent essentielles. D'autant plus essentielles qu'elles sont trop souvent passées sous silence dans les cours Javascript que l'on peut trouver un peu partout sur le web.

Si vous souhaitez approfondir certains aspects du langage Javascript, je vous recommande la lecture du hors-série n° 24 du magazine « Linux Pratique » :



LINUX PRATIQUE HS 24

Initiation à Javascript

Je choisis mon support



Version Numérique (PDF)

Quantité

1



8,00 € TTC



AJOUTER AU PANIER

S'ABONNER AU MAGAZINE

VOIR LES ANCIENS NUMÉROS

https://boutique.ed-diamond.com/anciens-numeros/424-lphs24.html?search_query=javascript&results=43#/38-format_du_magazine-version_numerique_pdf

Quoiqu'un peu ancien, le contenu de ce hors-série demeure très pertinent, et ses 2 chapitres d'introduction sont excellents :

- Initiation au JavaScript pour les (vrais) débutants (page 4)
- Les aventures de Bob le blob dans le DOM (page 16)

Pour une étude ludique du Javascript, au travers du graphisme, et en particulier du framework P5.js, vous pouvez essayer le tutoriel que j'ai créé en mai 2017 pour les membres du meetup « Creative Coding Paris ». Ce tutoriel se trouve dans le projet Github suivant (le PDF s'intitule « Tuto P5 – premiers pas ») :

<https://github.com/gregja/JSCorner>

Enfin, pour une étude plus détaillée sur la plupart des sujets abordés ici, je vous recommande la lecture du support qui s'intitule « Cours Javascript HTML5 » qui se trouve dans le même projet

Github. Ce dernier support reprend tous les points de manière beaucoup plus détaillée, avec en prime un chapitre très complet sur les sélecteurs CSS3 (incluant de nombreux exercices pratiques). Vous trouverez également dans ce support une bibliographie détaillée.

Un autre moyen intéressant d'explorer et d'approfondir votre connaissance du Javascript, c'est de le faire au travers du développement de jeux, ou au travers de l'étude des APIs Webaudio et Canvas, qui sont passionnantes.

Voici quelques pistes à explorer :

https://developer.mozilla.org/fr/docs/Web/API/Web_Audio_API/Using_Web_Audio_API

<https://developer.mozilla.org/en-US/docs/Games/Anatomy>

<http://www.codeincomplete.com/games/>

<http://youtube.com/codingmath>

Vous trouverez d'autres liens intéressants dans le support de cours « Javascript et HTML5 » qui se trouve dans le même dépôt Github que le présent support :

<https://github.com/gregja/JSCorner>

4 Annexe

4.1 Téléportation (source complet)

Code source complet de l'exemple présenté au chapitre 2.10.

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>ma page de test</title>
  <style>
    #liste1 {
      border-style: solid;
      background-color: silver;
      width:400px;
      height:200px;
      float:left;
    }
    #liste2 {
      border-style: solid;
      background-color: turquoise;
      width:400px;
      height:200px;
      float:right;
    }
  </style>
</head>
<body>
<div id="liste1"></div>
<div id="liste2"></div>

<script>
  var gen_liste_films = function (lang_liste, cible) {
    var macible = document.getElementById(cible);

    var maliste = document.createElement('ul');
    maliste.setAttribute('data-id', 'films');

    lang_liste.forEach(function(element, item) {

      // Création du lien cliquable déclencheur de la téléportation
      var tmp_link = document.createElement('a');
      tmp_link.append(document.createTextNode(' >> '));
      tmp_link.setAttribute('href', '#');
      tmp_link.setAttribute('onclick', 'teleportation(this, "liste2");');
      tmp_link.style['text-decoration'] = 'none'; // pas de soulignement

      // texte préformaté qui sera affiché et stocké dans un attribut data-text
      var texte = element.titre + ' (' + element.annee + ') ';

      // TextNode contenant le titre de chaque film
      var tmp_txt_element = document.createTextNode(texte);

      // Création de l'élément "li" correspondant à un film
```

```

        var tmp_li_element = document.createElement('li');
        tmp_li_element.setAttribute('data-year', element.annee);
        tmp_li_element.append(tmp_txt_element); // ajout du titre du film
        tmp_li_element.append(tmp_link); // ajout du lien de téléportation

        tmp_li_element.setAttribute('data-text', texte);

        maliste.append(tmp_li_element);
    });

    macible.append(maliste);
}

function teleportation(that, cible) {
    var target = document.getElementById(cible);
    // on regarde s'il existe déjà une liste "ul" dans la "div" de
    // destination
    var target2 = target.getElementsByTagName('ul');
    var target_list = null;
    // s'il n'existe pas de liste "ul", alors on en crée une
    if (target2.length == 0) {
        target_list = document.createElement('ul');
        target.append(target_list);
    } else {
        // s'il existe déjà une liste, on la sélectionne
        target_list = target2[0];
    }
    // on récupère le lien du noeud parent (soit la balise "li")
    // et on l'affecte à la liste "ul" cible
    var li_parente = that.parentNode;
    // à l'intérieur de la balise "li", on remplace le texte par
    // celui stocké dans l'attribut "data-text"
    li_parente.innerHTML = li_parente.getAttribute('data-text');

    target_list.append(li_parente);
}

var films_fantaisie = [
    {titre:'Le jour où la terre s\'emballa', annee:1955},
    {titre:'L\'attaque des morts poilants', annee:1960},
    {titre:'Le pont du fleuve Aie', annee:1974},
    {titre:'On a piétiné la lune', annee:1969},
    {titre:'Le bon, la brute et le mécréant', annee:1979},
    {titre:'La tour imprenable', annee:1969},
    {titre:'Le cri du harpon', annee:1980}
];

var origine = "liste1";
var destination = "liste2";

gen_liste_films(films_fantaisie, origine);
</script>
</body>
</html>

```

4.2 Tableau 4 colonnes version 2

Variante de l'exemple du chapitre 2.4.4 : dans cette variante, le code HTML est produit par les méthodes « createElement », « createTextNode », « append » :

```
<div id="cible">contenu qui sera écrasé par le Javascript</div>
<script>
    var lang_tab = ['java', 'javascript', 'scala', 'haskell',
        'go', 'rust', 'julia', 'basic', 'pascal',
        'fortran', 'c', 'c++', 'c#', 'ruby', 'python'];

    var liste = document.createElement('table');
    var ligne = null;
    var nbcol = 0;
    lang_tab.forEach(function(element, index) {
        if (nbcol > 3) {
            nbcol = 0;
            //liste += '</tr>';
        }
        if (nbcol == 0) {
            ligne = document.createElement('tr');
            liste.append(ligne);
            //liste += '<tr>';
        }
        var texte = document.createTextNode(element);
        var cellule = document.createElement('td');
        cellule.append(texte);
        ligne.append(cellule);
        //liste += '<td>'+element+'</td>';
        nbcol += 1;
    });
    if (nbcol < 4) {
        // Boucle de rattrapage pour les cas où le nombre de postes
        // du tableau n'est pas un multiple de 4
        while(nbcol < 4) {
            //liste += '<td>&nbsp;</td>'; // cellule avec un blanc
            ligne.append(document.createElement('td')
                .append(document.createTextNode('&nbsp;')));
            nbcol += 1;
        }
        //liste += '</tr>';
    }
    //liste += '</table>';

    var cible = document.getElementById('cible');
    cible.append(liste) ;
</script>
```

Explication : J'ai laissé en commentaire les portions de l'ancien code qui ont été remplacées par les méthodes « createElement » et « createTextNode ». Ceci afin de vous permettre d'identifier les portions de code impactées par les modifications. Vous remarquerez que dans cette

nouvelle version, nous n'avons pas à nous préoccuper de la génération des balises fermantes. Les balises fermantes sont en effet générées automatiquement pour nous par la fonction « createElement ». C'est plutôt cool, non ?

Dans l'exemple qui précède, la ligne suivante mérite une explication :

```
ligne.append(document.createElement('td')  
            .append(document.createTextNode(' &nbsp;')));
```

C'est du « tout en un », il faut le lire en partant de la fonction la plus à droite :

- On commence par créer un TextNode contenant un blanc (« »)
- Ce TextNode est créé à la volée et associé directement à un parent (de type « td »)
- Le parent de type « td » est lui-même créé à la volée et associé à un parent de type « tr » (qui a été créé dans la partie précédente du code)

C'est vrai que cela peut donner un peu mal à la tête la première fois qu'on lit ça, mais avec un peu de pratique, on s'aperçoit que c'est très pratique, et on y prend vite goût.

4.3 Où est le démarreur ?

Ce chapitre est le prolongement du chapitre 2.2.2 dans lequel nous avons étudié de quelle manière nous pouvions placer notre code JS dans une page HTML.

Nous avons indiqué dans le chapitre 2.2.2 qu'une bonne pratique consistait à placer son code en fin de page, avant la balise « /body ». Mais si on se contente de cette solution, ça fait un peu « bricolage », ça ne sent pas très bon. Ça fait un peu, genre... « ouf, le DOM est chargé, mon code JS peut maintenant démarrer, ça devrait bien se passer ».

Bon, il y a quand même solution robuste et élégante pour s'assurer que le code JS peut être chargé et exécuté, c'est de placer un écouteur d'événement sur la page, sur le type d'événement « load ». On peut l'écrire de plusieurs manières, en voici deux qui sont strictement équivalentes et qui fonctionnent bien :

- 1^{ère} version :

```
<script>
  function mon_code_a_moi () {
    // je place ici le code de mon application
  }

  document.body.onload = mon_code_a_moi ;
</script>
```

- 2^{ème} version :

```
<script>
  function mon_code_a_moi () {
    // je place ici le code de mon application
  }

  document.body.addEventListener ( 'load', mon_code_a_moi, false );
</script>
```

Explication : dans un cas comme dans l'autre, la fonction « mon_code_a_moi » sera appelée dès que le DOM du navigateur sera chargé, c'est-à-dire que l'événement « load » sera déclenché. Dans le premier exemple, on utilise la propriété « onload » qui est en fait un raccourci vers l'événement « load ». Les 2 solutions ci-dessus sont réellement équivalentes.

On peut souligner que l'événement « load » se déclenche dès que le navigateur a fini de charger le code HTML, les images (s'il y en a), les fichiers CSS et Javascript, bref, dès que le DOM est prêt à fonctionner.

Vous remarquerez que – dans les 2 cas ci-dessus - on fait référence à la fonction sans les parenthèses. Si vous faites l'erreur d'utiliser les parenthèses comme dans l'exemple ci-dessous, alors vous lancez l'exécution de la fonction, au lieu de laisser à la méthode « `addEventListener` » le soin de s'en charger (bref, c'est « bad ») :

```
document.body.addEventListener ( 'load', mon_code_a_moi (), false ); // BAD 😞
```

Ce problème lié à la détection de l'événement « load » a fait couler beaucoup d'encre virtuelle sur le net, et on trouve beaucoup d'articles sur le sujet. J'ai essayé d'en faire une synthèse plus approfondie (que ce que vous venez de lire), dans le chapitre 4.5.7 (« événement Load, les bonnes pratiques ») du support « Javascript et HTML5 » qui se trouve dans le même dépôt Github que le présent support.

4.4 Filtre dynamique (variante encore plus dynamique)

Dans le chapitre 2.9, nous avons étudié un exemple dans lequel un champ de saisie de type « select » était défini « en dur ». J'avais fait ce choix pour ne pas alourdir la démonstration. Mais maintenant que vous êtes en train de passer au niveau « pro », je pense que vous êtes en mesure d'étudier et de comprendre le code qui suit.

Ce code est fonctionnellement équivalent au code présenté dans le chapitre 2.9, à la différence que la liste des options du champ « select » est générée dynamiquement en fonction des années de sorties définies dans le tableau JS des films. L'effet le plus apparent est que seules les années pour lesquelles il existe des films sont affichées dans les options du champ « select ».

Parmi les points importants :

- Vous noterez dans le code de la fonction « gen_liste_annees » la manière dont les doublons sont éliminés lors de la génération du tableau JS des années de sortie.
- Vous remarquerez dans le code de la fonction « gen_options_select » de quelle manière on a forcé les attributs « multiple » et « size » du champ « select ».

Voici le code source complet :

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <title>Filtres dynamiques version 2</title>
</head>
<body>
<div id="cible" style="border-style: solid; background-color: silver;"></div>
<br><br>
<div style="border-style: solid; background-color: lightblue;">
  <br>
  <label>Sélectionnez une ou plusieurs années :<br>
    <select id="filtre-annees">
      </select>
  </label>
  <br>
</div>
<script>
  "use strict";

  var films_fantaisie = [
    {titre:'Le jour où la terre s\'emballa', annee:1955},
    {titre:'L\'attaque des morts poilants', annee:1960},
    {titre:'Le pont du fleuve Aïe', annee:1974},
    {titre:'On a piétiné la lune', annee:1969},
    {titre:'Le bon, la brute et le mécréant', annee:1979},
    {titre:'La tour imprenable', annee:1969},
    {titre:'Le cri du harpon', annee:1980}
```

```

];

/**
 * Génération dynamique de la liste des films dans la liste "ul"
 * @param Array
 * @param string
 */
function gen_liste_films (lang_liste, cible) {
    var macible = document.getElementById(cible);
    var maliste = document.createElement('ul');
    maliste.setAttribute('data-id', 'films');

    lang_liste.forEach(function(element, item) {
        var tmp_txt_element = document.createTextNode(
            element.titre + ' (' + element.annee + ')'
        );
        var tmp_li_element = document.createElement('li');
        tmp_li_element.setAttribute('data-year', element.annee);
        tmp_li_element.append(tmp_txt_element);
        maliste.append(tmp_li_element);
    });

    macible.append(maliste);
}

/**
 * Génération d'un tableau JS contenant la liste des années
 * @param Array
 * @returns {Array}
 */
function gen_liste_annees (films) {
    var anneess = [];

    // Le test à l'intérieur de la boucle permet d'éliminer les doublons éventuels
    for (var i=0, imax=films.length ; i < imax ; i+=1 ) {
        if (anneess.indexOf(films[i].annee) === -1) {
            // l'élément n'existe pas dans le tableau alors on l'ajoute
            anneess.push(films[i].annee) ;
        }
    }
    // on renvoie le tableau des années trié en ordre croissant
    return anneess.sort();
}

/**
 * Fonction de génération des options du champ select
 * dont l'id est transmis en paramètre
 * @param string
 */
function gen_options(select) {
    var cible = document.getElementById(select);
    var liste_annees = gen_liste_annees(films_fantaisie);

    if (liste_annees.length>0) {
        // on ajoute l'attribut "multiple" et l'attribut "size"
        // sur le champ "select"
        cible.setAttribute('multiple', 'true');
        cible.setAttribute('size', liste_annees.length);

        liste_annees.forEach(function (element, index) {

```

```

        var txtnode = document.createTextNode('Année ' + element);
        var option = document.createElement('option');
        option.append(txtnode);
        option.setAttribute('value', element);
        cible.append(option);
    });
}

/**
 * Filtrage dynamique des films en fonction des années sélectionnées
 * @param evt
 */
function filterdyn(evt) {
    // récupération des options de la balise "select"
    var options = evt.target.options ;
    // détermination des options sélectionnées par l'utilisateur
    var selected = [] ;
    for (var iopt = 0, ioptmax = options.length ; iopt < ioptmax ; iopt+=1 ) {
        if (options[iopt].selected) {
            selected.push(options[iopt].value) ;
        }
    }
    // identification de la liste des films
    var table_link = document.querySelectorAll('[data-id=films] > li') ;
    // pour chaque film de la liste, on détermine si sa date de sortie
    // correspond à la sélection de l'utilisateur
    for (var ilink = 0, ilinkmax = table_link.length ; ilink < ilinkmax ;
        ilink +=1) {
        var annee = table_link[ilink].dataset.year;
        if (selected.indexOf(annee) !== -1) {
            // l'année du film correspond à la sélection alors on l'affiche
            table_link[ilink].style.display='list-item' ;
        } else {
            // l'année de sortie ne correspond pas, alors on cache le film
            table_link[ilink].style.display='none' ;
        }
    }
}

// Génération de la liste des films dans la liste "ul"
gen_liste_films(films_fantaisie, 'cible');

// Génération des options du champs "select"
gen_options('filtre-annees');

// Mise en place de l'écouteur d'événements de type "change" sur le champ "select"
var filtre = document.getElementById('filtre-annees') ;
filtre.addEventListener('change', filterdyn, false);
</script>
</body>
</html>

```

5 Changelog

Version 1.2 publiée le 01/07/2017 :

- ajout de précisions dans le chapitre 2.5.3 sur les sélecteur CSS
- ajout d'un chapitre 2.10 présentant une méthode de téléportation ☺

Version 1.3 publiée le 03/07/2017 :

- ajout du chapitre 2.4.4 (génération d'un tableau HTML 4 colonnes)

Version 1.4 publiée le 4/07/2017 :

- complément sur le chapitre 2.4.3
- complément sur les boucles « for » dans le chapitre 2.5.1
- ajout d'un chapitre 4.2 (annexe)
- présentation de la notion de variable locale dans le chapitre 2.6

Version 1.5 publiée le 5/07/2017 :

- chapitre 2.2 scindé en 2 chapitre, avec création d'un chapitre 2.2.2 expliquant de quelle manière inclure du JS dans une page
- correction de quelques coquilles (notamment dans le chapitre 2.6)
- explication plus détaillée dans le chapitre 2.8 (événements)

Version 1.6 publiée le 5/07/2017 :

- une partie du chapitre 2.2.2 est transférée en annexe, dans le chapitre 4.3
- Ajout des chapitre 2.6.2 (mode strict) et 2.6.3 (variables musclées, et phénomène de « hoisting »)
- Chapitre 2.9 : correction (utilisation de « display : list-item » au lieu de « display : block », afin de pouvoir conserver le rendu des « puces » lors de l'affichage des films sélectionnés)
- Ajout du chapitre 4.4