Manipuler le code HTML

Le Document Object Model

Le **Document Object Model** (abrégé **DOM**) est une interface de programmation pour les documents XML et HTML.

Une interface de programmation, qu'on appelle aussi une **API** (pour **Application Programming Interface**), est un ensemble d'outils qui permettent de faire communiquer entre eux plusieurs programmes ou, dans le cas présent, différents langages. Le DOM est donc une API qui s'utilise avec les documents XML et HTML, et qui va nous permettre, via le JavaScript, d'accéder au code XML et/ou HTML d'un document. C'est grâce au DOM que nous allons pouvoir modifier des éléments HTML (afficher ou masquer un par exemple), en ajouter, en déplacer ou même en supprimer.

Petit historique

À l'origine, quand le JavaScript a été intégré dans les premiers navigateurs (Internet Explorer et Netscape Navigator), le DOM n'était pas unifié, c'est-à-dire que les deux navigateurs possédaient un DOM différent. Et donc, pour accéder à un élément HTML, la manière de faire différait d'un navigateur à l'autre, ce qui obligeait les développeurs Web à coder différemment en fonction du navigateur. En bref, c'était un peu la jungle.

Le W3C a mis de l'ordre dans tout ça, et a publié une nouvelle spécification que nous appellerons « DOM-1 » (pour DOM Level 1). Cette nouvelle spécification définit clairement ce qu'est le DOM, et surtout comment un document HTML ou XML est schématisé. Depuis lors, un document HTML ou XML est représenté sous la forme d'un arbre, ou plutôt hiérarchiquement. Ainsi, l'élément <html> contient deux éléments enfants : <head> et<body>, qui à leur tour contiennent d'autres éléments enfants.

Ensuite, la spécification DOM-2 a été publiée. La grande nouveauté de cette version 2 est l'introduction de la méthode getElementById() qui permet de récupérer un élément HTML ou XML en connaissant son ID.

L'objet window

Avant de véritablement parler du document, c'est-à-dire de la page Web, nous allons parler de l'objet window. L'objet window est ce qu'on appelle un objet global qui représente *la fenêtre du navigateur*. C'est à partir de cet objet que le JavaScript est exécuté.

Si nous reprenons notre « Hello World! », nous avons :

```
<!DOCTYPE html>
<html>
<head>
    <title>Hello World!</title>
</head>

<body>

    <script>

        alert('Hello world!');

    </script>

</body>
</html>
```

Contrairement à ce qui a été dit, alert() n'est pas vraiment une fonction. Il s'agit en réalité d'une méthode appartenant à l'objet window. Mais l'objet window est dit *implicite*, c'est-à-dire qu'il n'y a généralement pas besoin de le spécifier. Ainsi, ces deux instructions produisent le même effet, à savoir ouvrir une boîte de dialogue :

```
window.alert('Hello world!');
alert('Hello world!');
```

Puisqu'il n'est pas nécessaire de spécifier l'objet window, on ne le fait généralement pas sauf si cela est nécessaire, par exemple si on manipule des *frames*.

Si alert() est une méthode de l'objet window, toutes les fonctions ne font pas nécessairement partie de l'objet window. Ainsi, les fonctions comme isNaN(), parseInt() ou encore parseFloat() ne dépendent pas d'un objet. Ce sont des *fonctions globales*. Ces dernières sont, cependant, extrêmement rares. De même, lorsque vous déclarez une variable dans le contexte global de votre script, cette variable deviendra en vérité une propriété de l'objet window :

```
var text = 'Variable globale !';

(function() { // On utilise une IIFE pour « isoler » du code

    var text = 'Variable locale !';

    alert(text); // Forcément, la variable locale prend le dessus

    alert(window.text); // Mais il est toujours possible d'accéder à la variable globale grâce à l'objet « window »

})();
```

Toute variable non déclarée (donc utilisée sans jamais écrire le mot-clé var) deviendra immédiatement une propriété de l'objet window, et ce, quel que soit l'endroit où vous utilisez cette variable ! Prenons un exemple simple :

```
(function() { // On utilise une IIFE pour « isoler » du code

    text = 'Variable accessible !'; // Cette variable n'a jamais été déclarée et pourtant on lui attribue une valeur

})();
```

```
alert(text); // Affiche : « Variable accessible ! »
```

Notre variable a été utilisée pour la première fois dans une IIFE et pourtant nous y avons accès depuis l'espace global ! Alors pourquoi cela fonctionne-t-il de cette manière ? Tout simplement parce que le JavaScript va chercher à résoudre le problème que nous lui avons donné : on lui demande d'attribuer une valeur à la variable text, il va donc chercher cette variable mais ne la trouve pas, la seule solution pour résoudre le problème qui lui est donné est alors d'utiliser l'objet window. Ce qui veut dire qu'en écrivant :

```
text = 'Variable accessible !';
```

le code sera alors interprété de cette manière si aucune variable accessible n'existe avec ce nom :

```
window.text = 'Variable accessible !';
```

Il est conseillé de toujours utiliser le mot-clé var pour ne pas arriver arriver à de grandes confusions dans le code (et à de nombreux bugs). Si vous souhaitez déclarer une variable dans l'espace global alors que vous vous trouvez actuellement dans un autre espace (une IIFE, par exemple), spécifiez donc explicitement l'objet window. Le reste du temps, pensez bien à écrire le mot-clé var.

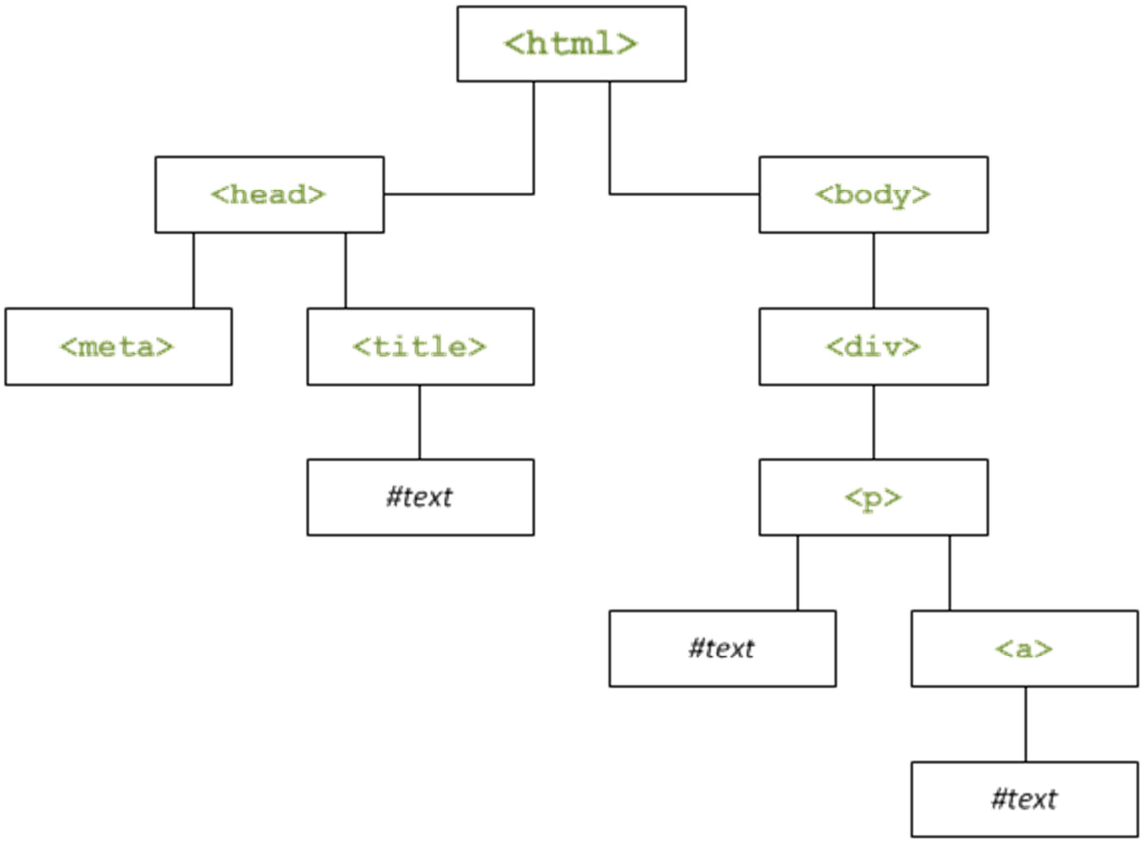
Le document

L'objet document est un sous-objet de window, l'un des plus utilisés. Il représente la *page Web* et plus précisément la balise <html>. C'est grâce à cet élément-là que nous allons pouvoir accéder aux éléments HTML et les modifier.

Naviguer dans le document

La structure DOM

Comme il a été dit précédemment, le DOM pose comme concept que la page Web est vue comme un arbre, comme une hiérarchie d'éléments. On peut donc schématiser une page Web simple comme ceci :



Voici le code source de la page :

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Le titre de la page</title>
</head>

<body>
    <div>
        <p>Un peu de texte <a>et un lien</a></p>
    </div>
</body>
</html>
```

Le schéma est plutôt simple : l'élément <html> contient deux éléments, appelés **enfants** : <head> et <body>. Pour ces deux enfants, <html> est l'élément **parent**. Chaque élément est appelé **nœud** (*node* en anglais). L'élément <head> contient lui aussi deux enfants : <meta> et <title>. <meta> ne contient pas d'enfant tandis que <title> en contient un, qui s'appelle #text. Comme son nom l'indique, #text est un élément qui contient du texte.

Il est important de bien saisir cette notion : le texte présent dans une page Web est vu par le DOM comme un nœud de type #text. Dans le schéma précédent, l'exemple du paragraphe qui contient du texte et un lien illustre bien cela :

```
<p>
    Un peu de texte
    <a>et un lien</a>
</p>
```

Si on va à la ligne après chaque nœud, on remarque que l'élément <p> contient deux enfants : #text qui contient « Un peu de texte » et <a>, qui lui-même contient un enfant #text représentant « et un lien ».

Accéder aux éléments

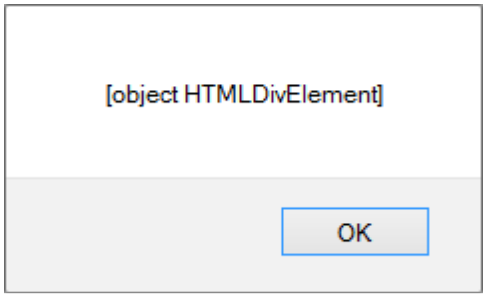
L'accès aux éléments HTML via le DOM est assez simple mais demeure actuellement plutôt limité. L'objet document possède trois méthodes
principales : getElementById(), getElementsByTagName() et getElementsByName().

getElementById()

Cette méthode permet d'accéder à un élément en connaissant son ID qui est simplement l'attribut id de l'élément. Cela fonctionne de cette manière :

```
<div id="myDiv">
    <p>Un peu de texte <a>et un lien</a></p>
</div>

<script>
    var div = document.getElementById('myDiv');

    alert(div);
</script>
```

En exécutant ce code, le navigateur affiche ceci :



Il nous dit que div est un objet de type HTMLDivElement. En clair, c'est un élément HTML qui se trouve être un , ce qui nous montre que le script fonctionne correctement.

getElementsByTagName()

Attention dans le nom de cette méthode : il y a un « s » à Elements. C'est une source fréquente d'erreurs.
Cette méthode permet de récupérer, sous la forme d'un tableau, tous les éléments de la famille. Si, dans une page, on veut récupérer tous les , il suffit de faire comme ceci :

```
var divs = document.getElementsByTagName('div');

for (var i = 0, c = divs.length ; i < c ; i++) {
```

```
    alert('Element n° ' + (i + 1) + ' : ' + divs[i]);
}
```

La méthode retourne une collection d'éléments (utilisable de la même manière qu'un tableau). Pour accéder à chaque élément, il est nécessaire de parcourir le tableau avec une petite boucle.

Deux petites astuces :

1. Cette méthode est accessible sur n'importe quel élément HTML et pas seulement sur l'objet document.
2. En paramètre de cette méthode vous pouvez mettre une chaîne de caractères contenant un astérisque * qui récupérera *tous* les éléments HTML contenus dans l'élément ciblé.

getElementsByName()

Cette méthode est semblable à getElementsByTagName() et permet de ne récupérer que les éléments qui possèdent un attribut name que vous spécifiez. L'attribut name n'est utilisé qu'au sein des formulaires, et est déprécié depuis la spécification HTML5 dans tout autre élément que celui d'un formulaire. Par exemple, vous pouvez vous en servir pour un élément <input> mais pas pour un élément <map>.

Sachez aussi que cette méthode est dépréciée en XHTML mais est standardisée en HTML5.

Accéder aux éléments grâce aux technologies récentes

Ces dernières années, le JavaScript a beaucoup évolué pour faciliter le développement Web. Les deux méthodes que nous allons étudier sont récentes et ne sont pas supportées par les très vieilles versions des navigateurs, leur support commence à partir de la version 8 d'Internet Explorer, pour les autres navigateurs vous n'avez normalement pas de soucis.

Ces deux méthodes sont querySelector() et querySelectorAll() et ont pour particularité de grandement simplifier la sélection d'éléments dans l'arbre DOM grâce à leur mode de fonctionnement. Ces deux méthodes prennent pour paramètre un seul argument : une chaîne de caractères !

Cette chaîne de caractères doit être un sélecteur CSS comme ceux que vous utilisez dans vos feuilles de style. Exemple :

```
#menu .item span
```

Ce sélecteur CSS stipule que l'on souhaite sélectionner les balises de type contenues dans les classes.item elles-mêmes contenues dans un élément dont l'identifiant est #menu.

Internet Explorer 8 ne supporte pas l'usage des sélecteurs CSS 3 avec ces méthodes !
La première, querySelector(), renvoie le premier élément trouvé correspondant au sélecteur CSS, tandis que querySelectorAll() va renvoyer *tous* les éléments (sous forme de tableau) correspondant au sélecteur CSS fourni. Prenons un exemple simple :

```
<div id="menu">

    <div class="item">
        <span>Élément 1</span>
        <span>Élément 2</span>
    </div>

    <div class="publicite">
        <span>Élément 3</span>
        <span>Élément 4</span>
    </div>

</div>

<div id="contenu">
    <span>Introduction au contenu de la page...</span>
</div>
```

Maintenant, essayons le sélecteur CSS présenté plus haut : #menu .item span

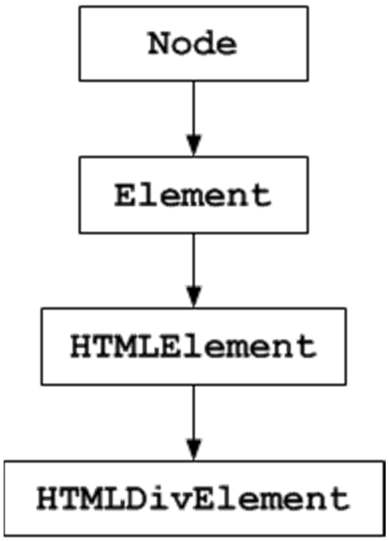
```
var query = document.querySelector('#menu .item span'),
    queryAll = document.querySelectorAll('#menu .item span');

alert(query.innerHTML); // Affiche : "Élément 1"

alert(queryAll.length); // Affiche : "2"
alert(queryAll[0].innerHTML + ' - ' + queryAll[1].innerHTML); // Affiche : "Élément 1 -
Élément 2"
```

L'héritage des propriétés et des méthodes

Le JavaScript voit les éléments HTML comme étant des objets, cela veut donc dire que chaque élément HTML possède des propriétés et des méthodes. Cependant tous ne possèdent pas les mêmes propriétés et méthodes. Certaines sont néanmoins communes à tous les éléments HTML, car tous les éléments HTML sont d'un même type : le type Node, qui signifie « nœud » en anglais.

Notion d'héritage

Nous avons vu qu'un élément est un objet HTMLDivElement, mais un objet, en JavaScript, peut appartenir à différents groupes. Ainsi, notre est un HTMLDivElement, qui est un sous-objet d'HTMLElement qui est lui-même un sous-objet d'Element. Element est enfin un sous-objet de Node. Ce schéma est plus parlant :



En JavaScript, un objet peut appartenir à plusieurs groupes
L'objet Node apporte un certain nombre de propriétés et de méthodes qui pourront être utilisées depuis un de ses sous-objets. En clair, les sous-objets *héritent* des propriétés et méthodes de leurs objets parents.

Éditer les éléments HTML

Les éléments HTML sont souvent composés d'attributs (l'attribut href d'un <a> par exemple), et d'un contenu, qui est de type #text. Le contenu peut aussi être un autre élément HTML.

Comme dit précédemment, un élément HTML est un objet qui appartient à plusieurs objets, et de ce fait, qui hérite des propriétés et méthodes de ses objets parents.

Les attributs

Via l'objet Element

Pour interagir avec les attributs, l'objet Element nous fournit deux méthodes, getAttribute() et setAttribute() permettant respectivement de récupérer et d'éditer un attribut. Le premier paramètre est le nom de l'attribut, et le deuxième, dans le cas de setAttribute() uniquement, est la nouvelle valeur à donner à l'attribut. Petit exemple :

```
<body>
    <a id="myLink" href="http://www.google.com">Un lien modifié dynamiquement</a>

    <script>
        var link = document.getElementById('myLink');
        var href = link.getAttribute('href'); // On récupère l'attribut « href »

        alert(href);

        link.setAttribute('href', 'http://www.yahoo.com'); // On édite l'attribut « href »
```

```
    </script>
</body>
```

On commence par récupérer l'élément myLink, et on lit son attribut href via getAttribute(). Ensuite on modifie la valeur de l'attribut href avec setAttribute(). Le lien pointe maintenant vers http://www.yahoo.com.

Les attributs accessibles

Pour la plupart des éléments courants comme <a>, il est possible d'accéder à un attribut via une propriété. Ainsi, si on veut modifier la destination d'un lien, on peut utiliser la propriété href, comme ceci :

```
<body>
    <a id="myLink" href="http://www.google.com">Un lien modifié dynamiquement</a>

    <script>
        var link = document.getElementById('myLink');
        var href = link.href;

        alert(href);

        link.href = 'http://www.yahoo.com';
    </script>
</body>
```

C'est cette façon de faire que qui est utilisée majoritairement pour les formulaires : pour récupérer ou modifier la valeur d'un champ, on utilisera la propriété value.

Attention cependant ! Un attribut auquel on accède par le biais de la méthode getAttribute() renverra la valeur exacte de ce qui est écrit dans le code HTML (sauf après une éventuelle modification) tandis que l'accès par le biais de sa propriété peut entraîner quelques changements. Prenons l'exemple suivant :

```
<a href="/">Retour à l'accueil</a>
```

L'accès à l'attribut href avec la méthode getAttribute() retournera bien un simple slash tandis que l'accès à la propriété retournera une URL absolue. Si votre nom de domaine est « mon_site.com » vous obtiendrez alors « http://mon_site.com/ ».

La classe

On peut penser que pour modifier l'attribut class d'un élément HTML, il suffit d'utiliser element.class. Ce n'est pas possible, car le mot-clé class est réservé en JavaScript, bien qu'il n'ait aucune utilité. À la place de class, il faudra utiliser className.

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8" />
```

```
    <title>Le titre de la page</title>
    <style>
        .blue {
            background: blue;
            color: white;
        }
    </style>
</head>

<body>
    <div id="myColoredDiv">
        <p>Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
        document.getElementById('myColoredDiv').className = 'blue';
    </script>
</body>
</html>
```

Dans cet exemple, on définit la classe CSS .blue à l'élément myColoredDiv, ce qui fait que cet élément sera affiché avec un arrière-plan bleu et un texte blanc.

Toujours dans le même cas, le nom for est réservé lui aussi en JavaScript (pour les boucles). Vous ne pouvez donc pas modifier l'attribut HTML for d'un <label> en écrivant element.for, il faudra utiliser element.htmlFor à la place.

Attention : si votre élément comporte plusieurs classes (exemple : <a class="external red u">) et que vous récupérez la classe avec className, cette propriété ne retournera pas un tableau avec les différentes classes, mais bien la chaîne « external red u », ce qui n'est pas vraiment le comportement souhaité. Il vous faudra alors couper cette chaîne avec la méthode split() pour obtenir un tableau, comme ceci :

```
var classes = document.getElementById('myLink').className;
var classesNew = [];
classes = classes.split(' ');

for (var i = 0, c = classes.length; i < c; i++) {
    if (classes[i]) {
        classesNew.push(classes[i]);
    }
}

alert(classesNew);
```

On récupère les classes, on découpe la chaîne, mais comme il se peut que plusieurs espaces soient présents entre chaque nom de classe, on vérifie chaque élément pour voir s'il contient quelque chose (s'il n'est pas vide). On en profite pour créer un nouveau tableau, classesNew, qui contiendra les noms des classes, sans « parasites ».

Si le support d'Internet Explorer avant sa version 10 vous importe peu, vous pouvez aussi vous tourner vers la propriété classList qui permet de consulter les classes sous forme d'un tableau et de les manipuler aisément :

```
var div = document.querySelector('div');

// Ajoute une nouvelle classe
div.classList.add('new-class');

// Retire une classe
div.classList.remove('new-class');

// Retire une classe si elle est présente ou bien l'ajoute si elle est absente
div.classList.toggle('toggled-class');

// Indique si une classe est présente ou non
if (div.classList.contains('old-class')) {
    alert('La classe .old-class est présente !');
}

// Parcourt et affiche les classes CSS
var result = '';

for (var i = 0; i < div.classList.length; i++) {
    result += '.' + div.classList[i] + '\n';
}

alert(result);
```

Le contenu : innerHTML

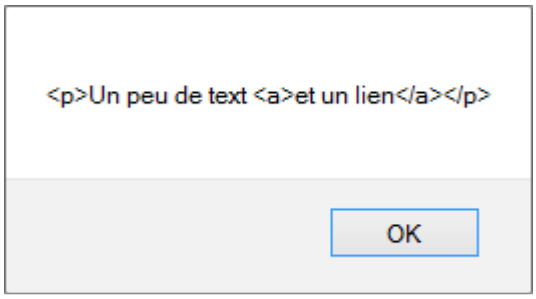
La propriété innerHTML est spéciale et demande une petite introduction. Elle a été créée par Microsoft pour les besoins d'Internet Explorer et a été normalisée au sein du HTML5. Bien que non normalisée pendant des années, elle est devenue un standard parce que tous les navigateurs la supportaient déjà, et non l'inverse comme c'est généralement le cas.

Récupérer du HTML

innerHTML permet de récupérer le code HTML enfant d'un élément sous forme de texte. Ainsi, si des balises sont présentes, innerHTML les retournera sous forme de texte :

```
<body>
    <div id="myDiv">
        <p>Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
        var div = document.getElementById('myDiv');

        alert(div.innerHTML);
    </script>
</body>
```

Nous avons donc bien une boîte de dialogue qui affiche le contenu de myDiv, sous forme de texte :



Ajouter ou éditer du HTML

Pour éditer ou ajouter du contenu HTML, il suffit de faire l'inverse, c'est-à-dire de définir un nouveau contenu :

```
document.getElementById('myDiv').innerHTML = '<blockquote>Je mets une citation à la place du paragraphe</blockquote>';
```

Si vous voulez ajouter du contenu, et ne pas modifier le contenu déjà en place, il suffit d'utiliser += à la place de l'opérateur d'affectation :

```
document.getElementById('myDiv').innerHTML += ' et <strong>une portion mise en emphase</strong>.';
```

Attention, ll ne faut pas utiliser le **+=** dans une boucle ! En effet, innerHTML ralentit considérablement l'exécution du code si l'on opère de cette manière, il vaut donc mieux concaténer son texte dans une variable pour ensuite ajouter le tout via innerHTML. Exemple :

```
var text = '';

while ( /* condition */ ) {
    text += 'votre_texte'; // On concatène dans la variable « text »
}

element.innerHTML = text; // Une fois la concaténation terminée, on ajoute le tout à « element » via innerHTML
```

Attention, si vous souhaitez ajouter une balise <script> à votre page par le biais de la propriété innerHTML, sachez que ceci ne fonctionne pas ! Il est toutefois possible de créer cette balise par le biais de la méthode createElement().

innerText et textContent

Penchons-nous maintenant sur deux propriétés analogues à innerHTML : innerText pour Internet Explorer et textContent pour les autres navigateurs.

innerText

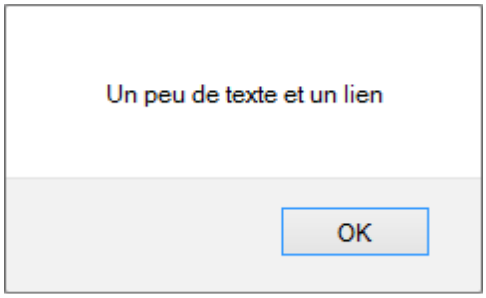
La propriété innerText a aussi été introduite dans Intedrnet Explorer, mais à la différence de sa propriété sœur innerHTML, elle n'a jamais été standardisée et n'est pas supportée par tous les

navigateurs. Internet Explorer (pour toute version antérieure à la neuvième) ne supporte que cette propriété et non pas la version standardisée que nous verrons par la suite.

Le fonctionnement d'innerText est le même qu'innerHTML excepté le fait que seul le texte est récupéré, et non les balises. C'est pratique pour récupérer du contenu sans le balisage, petit exemple :

```
<body>
    <div id="myDiv">
        <p>Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
        var div = document.getElementById('myDiv');

        alert(div.innerText);
    </script>
</body>
```

Ce qui nous donne bien « Un peu de texte et un lien », sans les balises :



textContent

La propriété textContent est la version standardisée d'innerText ; elle est reconnue par tous les navigateurs à l'exception des versions d'Internet Explorer antérieures à la 9. Le fonctionnement est le même. Maintenant une question se pose : comment faire un script qui fonctionne à la fois pour Internet Explorer et les autres navigateurs ? C'est ce que nous allons voir !

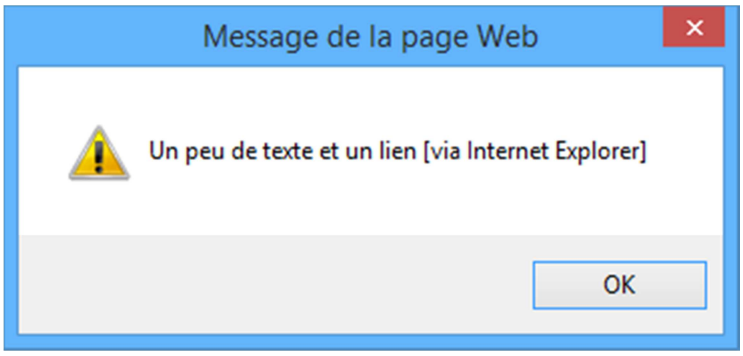
Tester le navigateur

Il est possible via une simple condition de tester si le navigateur prend en charge telle ou telle méthode ou propriété.

```
<body>
    <div id="myDiv">
        <p>Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
        var div = document.getElementById('myDiv');
        var txt = '';

        if (div.textContent) { // « textContent » existe ? Alors on s'en sert !
            txt = div.textContent;
        } else if (div.innerText) { // « innerText » existe ? Alors on doit être sous IE.
            txt = div.innerText + ' [via Internet Explorer]';
```

```
        } else { // Si aucun des deux n'existe, cela est sûrement dû au fait qu'il n'y a
pas de texte
            txt = ''; // On met une chaîne de caractères vide
        }

        alert(txt);
    </script>
</body>
```

Il suffit donc de tester par le biais d'une condition si l'instruction fonctionne. Si textContent ne fonctionne pas, pas de soucis, on prend innerText :



Ce code est quand même très long et redondant. Il est possible de le raccourcir de manière considérable :

```
txt = div.textContent || div.innerText || '';
```

Naviguer entre les nœuds

Nous avons vu qu'on utilisait les méthodes getElementById() et getElementsByTagName()pour accéder aux éléments HTML. Une fois que l'on a atteint un élément, il est possible de se déplacer de façon un peu plus précise, avec toute une série de méthodes et de propriétés.

La propriété parentNode

La propriété parentNode permet d'accéder à l'élément parent d'un élément :

```
<blockquote>
    <p id="myP">Ceci est un paragraphe !</p>
</blockquote>
```

Admettons qu'on doive accéder à myP, et que pour une autre raison on doive accéder à l'élément<blockquote>, qui est le parent de myP. Il suffit d'accéder à myP puis à son parent, avec parentNode :

```
var paragraph = document.getElementById('myP');
var blockquote = paragraph.parentNode;
```

nodeType et nodeName

nodeType et nodeName servent respectivement à vérifier le *type* d'un nœud et le *nom* d'un nœud. nodeType retourne un nombre, qui correspond à un type de nœud. Voici un tableau qui liste les types possibles, ainsi que leurs numéros (les types courants sont mis en gras) :

Numéro	Type de nœud
1	**Nœud élément**
2	**Nœud attribut**
3	**Nœud texte**
4	Nœud pour passage CDATA (relatif au XML)
5	Nœud pour référence d'entité
6	Nœud pour entité
7	Nœud pour instruction de traitement
8	**Nœud pour commentaire**
9	Nœud document

Numéro	Type de nœud
10	Nœud type de document
11	Nœud de fragment de document
12	Nœud pour notation

nodeName, quant à lui, retourne simplement le nom de l'élément, en majuscule. Il est toutefois conseillé d'utiliser toLowerCase() (ou toUpperCase()) pour forcer un format de casse et ainsi éviter les mauvaises surprises.

```
var paragraph = document.getElementById('myP');
alert(paragraph.nodeType + '\n\n' + paragraph.nodeName.toLowerCase());
```

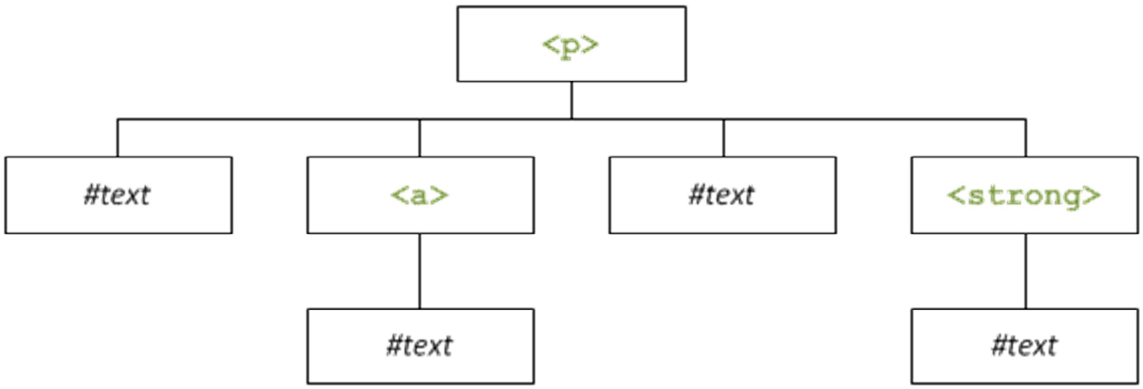
Lister et parcourir des nœuds enfants

firstChild et lastChild

Comme leur nom le laisse présager, firstChild et lastChild servent respectivement à accéder au premier et au dernier enfant d'un nœud.

```
<!doctype html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Le titre de la page</title>
</head>

<body>
    <div>
        <p id="myP">Un peu de texte, <a>un lien</a> et <strong>une portion en
emphase</strong></p>
    </div>

    <script>
        var paragraph = document.getElementById('myP');
        var first = paragraph.firstChild;
        var last = paragraph.lastChild;

        alert(first.nodeName.toLowerCase());
        alert(last.nodeName.toLowerCase());
    </script>
</body>
</html>
```

En schématisant l'élément myP précédent, on obtient ceci :



Le premier enfant de <p> est un nœud textuel, alors que le dernier enfant est un élément .

Dans le cas où vous ne souhaiteriez récupérer que les enfants qui sont considérés comme des éléments HTML (et donc éviter les nœuds **#text** par exemple), sachez qu'il existe les propriétés firstElementChild et lastElementChild. Ainsi, dans l'exemple précédent, la propriété firstElementChild renverrait l'élément <a>.
Malheureusement, ces deux propriétés ne sont supportées qu'à partir de la version 9 concernant Internet Explorer.

nodeValue et data

Changeons de problème : il faut récupérer le texte du premier enfant, et le texte contenu dans l'élément, mais comment faire ?

Il faut soit utiliser la propriété nodeValue soit la propriété data. Si on recode le script précédent, nous obtenons ceci :

```
var paragraph = document.getElementById('myP');
var first = paragraph.firstChild;
var last = paragraph.lastChild;

alert(first.nodeValue);
alert(last.firstChild.data);
```

first contient le premier nœud, un nœud textuel. Il suffit de lui appliquer la propriété nodeValue (ou data) pour récupérer son contenu ; pas de difficulté ici. En revanche, il y a une petite différence avec notre élément : vu que les propriétés nodeValue et data ne s'appliquent *que* sur des nœuds textuels, il nous faut d'abord accéder au nœud textuel que contient notre élément, c'est-à-dire son nœud enfant. Pour cela, on utilise firstChild (et non pas firstElementChild), et ensuite on récupère le contenu avec nodeValue ou data.

childNodes

La propriété childNodes retourne un tableau contenant la liste des enfants d'un élément. L'exemple suivant illustre le fonctionnement de cette propriété, de manière à récupérer le contenu des éléments enfants :

```
<body>
    <div>
        <p id="myP">Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
        var paragraph = document.getElementById('myP');
        var children = paragraph.childNodes;

        for (var i = 0, c = children.length; i < c; i++) {

            if (children[i].nodeType === Node.ELEMENT_NODE) { // C'est un élément HTML
                alert(children[i].firstChild.data);
            } else { // C'est certainement un nœud textuel
                alert(children[i].data);
            }
        }
    </script>
</body>
```

Vous remarquerez que nous n'avons pas comparé la propriété nodeType à la valeur 1 mais à Node.ELEMENT_NODE, il s'agit en fait d'une constante qui contient la valeur 1, ce qui est plus facile à lire que le chiffre seul. Il existe une constante pour chaque type de nœud, vous pouvez les retrouver sur le MDN.

nextSibling et previousSibling

nextSibling et previousSibling sont deux propriétés qui permettent d'accéder respectivement au nœud suivant et au nœud précédent.

```
<body>
    <div>
        <p id="myP">Un peu de texte, <a>un lien</a> et <strong>une portion en
emphase</strong></p>
    </div>

    <script>
        var paragraph = document.getElementById('myP');
        var first = paragraph.firstChild;
        var next = first.nextSibling;

        alert(next.firstChild.data); // Affiche « un lien »
    </script>
</body>
```

Dans cet exemple, on récupère le premier enfant de myP, et sur ce premier enfant on utilise nextSibling, qui permet de récupérer l'élément <a>. Avec ça, il est même possible de parcourir les enfants d'un élément, en utilisant une boucle while :

```
<body>
    <div>
        <p id="myP">Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
        var paragraph = document.getElementById('myP');
        var child = paragraph.lastChild; // On prend le dernier enfant

        while (child) {

            if (child.nodeType === Node.ELEMENT_NODE) { // C'est un élément HTML
                alert(child.firstChild.data);
            } else { // C'est certainement un nœud textuel
                alert(child.data);
            }

            child = child.previousSibling; // À chaque tour de boucle, on prend l'enfant
précédent

        }
    </script>
</body>
```

La boucle tourne « à l'envers », car on commence par récupérer le dernier enfant et on chemine à reculons.

Tout comme pour firstChild et lastChild, sachez qu'il existe les propriétés nextElementSibling et previousElementSibling qui permettent, elles aussi, de ne récupérer que les éléments HTML. Ces deux propriétés ont les mêmes problèmes de compatibilité que firstElementChild et lastElementChild.

Attention aux nœuds vides

En considérant le code HTML suivant, on peut penser que l'élément ne contient que trois enfants <p>:

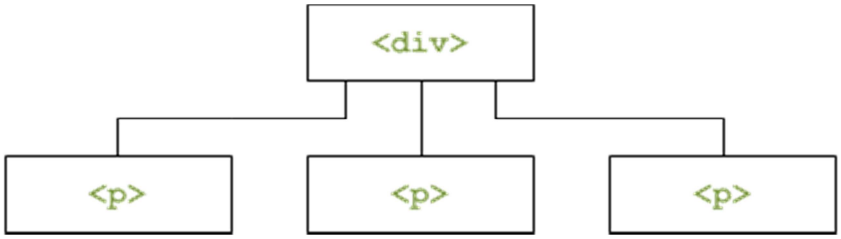
```
<div>
    <p>Paragraphe 1</p>
    <p>Paragraphe 2</p>
    <p>Paragraphe 3</p>
</div>
```

Mais attention, car ce code est radicalement différent de celui-ci :

```
<div><p>Paragraphe 1</p><p>Paragraphe 2</p><p>Paragraphe 3</p></div>
```

En fait, les espaces entre les éléments tout comme les retours à la ligne sont considérés comme des nœuds textuels (enfin, cela dépend des navigateurs) ! Ainsi donc, si l'on schématise le premier code, on obtient ceci :

Alors que le deuxième code peut être schématisé comme ça :



Heureusement, il existe une solution à ce problème ! Les attributs firstElementChild, lastElementChild, nextElementSibling et previousElementSibling ne retournent que des éléments HTML et permettent donc d'ignorer les nœuds textuels. Ils s'utilisent exactement de la même manière que les attributs de base (firstChild, lastChild, etc.). Attention, ces attributs ne sont pas supportés par les versions d'Internet Explorer antérieures à la 9.

Créer et insérer des éléments

Ajouter des éléments HTML

Avec le DOM, l'ajout d'un élément HTML se fait en trois temps :

1. On crée l'élément ;
2. On lui affecte des attributs ;
3. On l'insère dans le document, et ce n'est qu'à ce moment-là qu'il sera « ajouté ».

Création de l'élément

La création d'un élément se fait avec la méthode createElement(), un sous-objet de l'objet racine, c'est-à-dire document dans la majorité des cas :

```
var newLink = document.createElement('a');
```

On crée ici un nouvel élément <a>. Cet élément est créé, mais n'est *pas* inséré dans le document, il n'est donc pas visible. Cela dit, on peut déjà travailler dessus, en lui ajoutant des attributs ou même des événements (que nous verrons dans le chapitre suivant).

Si vous travaillez dans une page Web, l'élément racine sera toujours document, sauf dans le cas des frames.

Affectation des attributs

Ici, c'est comme nous avons vu précédemment : on définit les attributs, soit avec setAttribute(), soit directement avec les propriétés adéquates.

```
newLink.id    = 'new_link';
newLink.href  = 'http://www.google.com';
newLink.title = 'Google';
newLink.setAttribute('tabindex', '10');
```

Insertion de l'élément

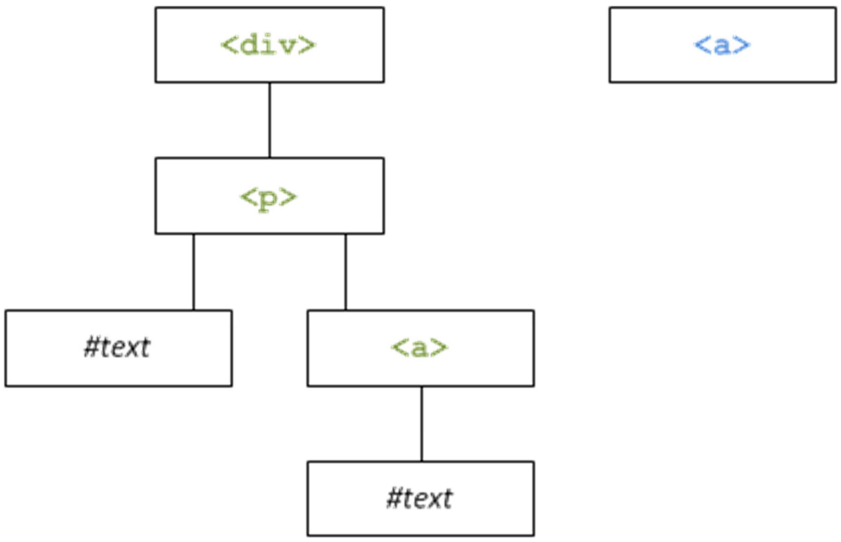
On utilise la méthode appendChild() pour insérer l'élément. *Append child* signifie « ajouter un enfant », ce qui signifie qu'il nous faut connaître l'élément auquel on va ajouter l'élément créé. Considérons donc le code suivant :

```
<!doctype html>
<html>
 <head>
    <meta charset="utf-8" />
    <title>Le titre de la page</title>
</head>

<body>
    <div>
        <p id="myP">Un peu de texte <a>et un lien</a></p>
    </div>
</body>
</html>
```

On va ajouter notre élément <a> dans l'élément <p> portant l'ID myP. Pour ce faire, il suffit de récupérer cet élément, et d'ajouter notre élément <a> via appendChild() :

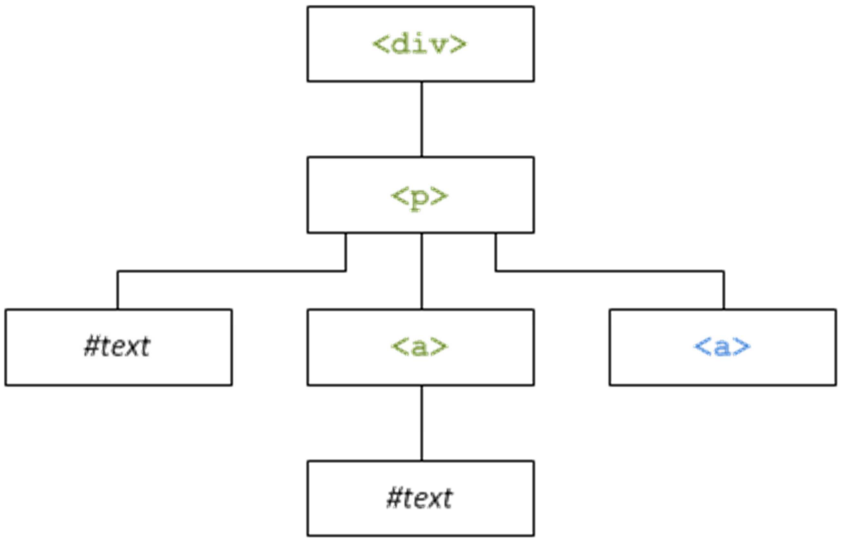
```
document.getElementById('myP').appendChild(newLink);
```

Avant d'insérer notre élément <a>, la structure DOM du document ressemble à ceci :



Avant d'insérer notre élément <a>, la structure DOM du document ressemble à cette figure

On voit que l'élément <a> existe, mais n'est pas lié. Un peu comme s'il était libre dans le document : il n'est pas encore placé. Le but est de le placer comme enfant de l'élément myP. La méthode appendChild() va alors déplacer notre <a> pour le placer en tant que dernier enfant de myP :



Cela veut dire qu'appendChild() insérera toujours l'élément en tant que dernier enfant, ce qui n'est pas toujours très pratique.

Ajouter des nœuds textuels

L'élément a été inséré, seulement il manque quelque chose : le contenu textuel ! La méthode createTextNode() sert à créer un nœud textuel (de type #text) qu'il nous suffira d'ajouter à notre élément fraîchement inséré, comme ceci :

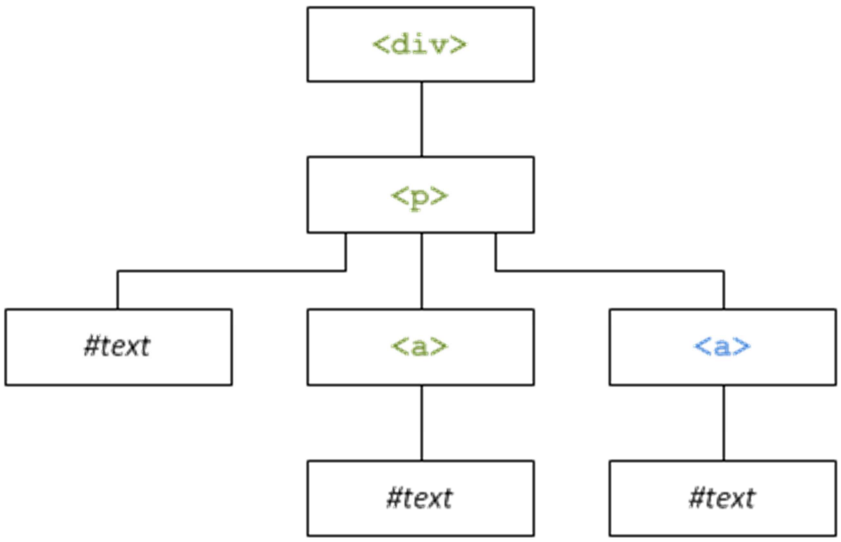
```
var newLinkText = document.createTextNode("Exemple de texte");

newLink.appendChild(newLinkText);
```

L'insertion se fait ici aussi avec appendChild(), sur l'élément newLink. Afin d'y voir plus clair, résumons le code :

```
<body>
    <div>
        <p id="myP">Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
        var newLink = document.createElement('a');

        newLink.id = 'new_link';
        newLink.href = 'http://www.google.com';
        newLink.title = 'Google';
        newLink.setAttribute('tabindex', '10');

        document.getElementById('myP').appendChild(newLink);

        var newLinkText = document.createTextNode("Exemple de texte");
```

```
        newLink.appendChild(newLinkText);
    </script>
</body>
```

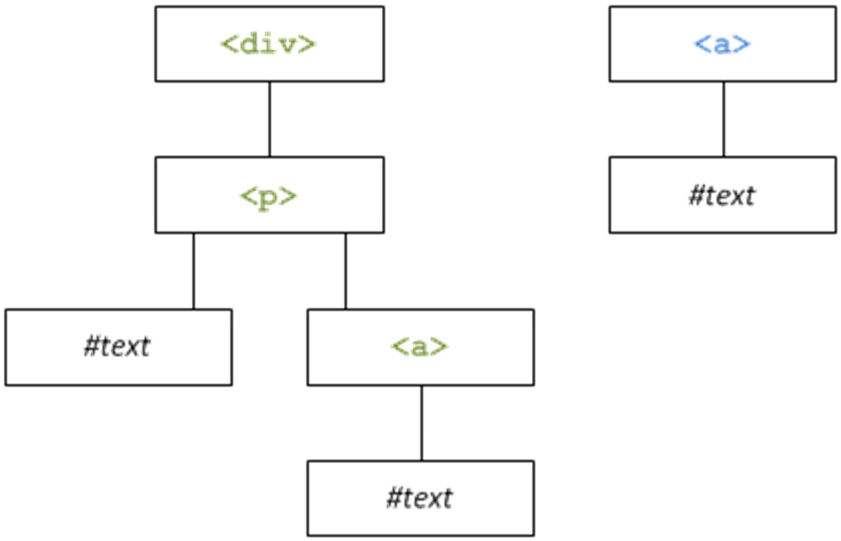
Voici donc ce qu'on obtient, sous forme schématisée :



Le fait d'insérer via appendChild() n'a aucune incidence sur l'ordre d'exécution des instructions. Cela veut donc dire que l'on peut travailler sur les éléments HTML et les nœuds textuels sans qu'ils soient au préalable insérés dans le document. Par exemple, on pourrait ordonner le code comme ceci :

```
var newLink = document.createElement('a');
var newLinkText = document.createTextNode("Exemple de texte");

newLink.id = 'new_link';
newLink.href = 'http://www.google.com';
newLink.title = 'Google';
newLink.setAttribute('tabindex', '10');

newLink.appendChild(newLinkText);

document.getElementById('myP').appendChild(newLink);
```

Ici, on commence par créer les deux éléments (le lien et le nœud de texte), puis on affecte les variables au lien et on lui ajoute le nœud textuel. À ce stade-ci, l'élément HTML contient le nœud textuel, mais cet élément n'est pas encore inséré dans le document :



L'élément HTML contient le nœud textuel, mais cet élément n'est pas encore inséré dans le document

La dernière instruction insère alors le tout.

Nous vous conseillons d'organiser votre code comme le dernier exemple, c'est-à-dire avec la création de tous les éléments au début, puis les différentes opérations d'affection. Enfin, l'insertion des éléments les uns dans les autres et, pour terminer, l'insertion dans le document. Au moins comme ça c'est structuré, clair et surtout plus performant !
appendChild() retourne une *référence* pointant sur l'objet qui vient d'être inséré. Cela peut servir dans le cas où vous n'avez pas déclaré de variable intermédiaire lors du processus de création de votre élément. Par exemple, le code suivant ne pose pas de problème :

```
var span = document.createElement('span');
document.body.appendChild(span);

span.innerHTML = 'Du texte en plus !';
```

En revanche, si vous retirez l'étape intermédiaire (la première ligne) pour gagner une ligne de code alors vous allez être embêté pour modifier le contenu :

```
document.body.appendChild(document.createElement('span'));

span.innerHTML = 'Du texte en plus !'; // La variable « span » n'existe plus… Le code plante.
```

La solution à ce problème est d'utiliser la référence retournée par appendChild() de la façon suivante :

```
var span = document.body.appendChild(document.createElement('span'));

span.innerHTML = 'Du texte en plus !'; // Là, tout fonctionne !
```

Notions sur les références

En JavaScript et comme dans beaucoup de langages, le contenu des variables est « passé par valeur ». Cela veut donc dire que si une variable nick1 contient le prénom « Clarisse » et qu'on affecte cette valeur à une autre variable, la valeur est *copiée* dans la nouvelle. On obtient alors deux variables distinctes, contenant la même valeur :

```
var nick1 = 'Clarisse';
var nick2 = nick1;
```

Si on modifie la valeur de nick2, la valeur de nick1 reste inchangée : normal, les deux variables sont bien distinctes.

Les références

Outre le « passage par valeur », le JavaScript possède un « passage par référence ». En fait, quand une variable est créée, sa valeur est mise en mémoire par l'ordinateur. Pour pouvoir retrouver cette valeur, elle est associée à une adresse que seul l'ordinateur connaît et manipule.

Quand on passe une valeur par référence, on transmet l'adresse de la valeur, ce qui va permettre d'avoir deux variables qui *pointent* sur une même valeur.

Les références avec le DOM

Schématiser le concept de référence avec le DOM est assez simple : deux variables peuvent accéder au même élément. Regardez cet exemple :

```
var newLink = document.createElement('a');
var newLinkText = document.createTextNode('Exemple de texte');

newLink.id = 'new_link';
newLink.href = 'http://www.google.com';

newLink.appendChild(newLinkText);

document.getElementById('myP').appendChild(newLink);

// On récupère, via son ID, l'élément fraîchement inséré
var newLinkref = document.getElementById('new_link');

newLinkRef.href = 'http://www.yahoo.com';

// newLink.href affiche bien la nouvelle URL :
alert(newLink.href);
```

La variable newLink contient en réalité une référence vers l'élément <a> qui a été créé. newLink ne contient pas l'élément, il contient une adresse qui pointe vers ce fameux <a>. Une fois que l'élément HTML est inséré dans la page, on peut y accéder avec de nombreuses autres façons, comme avec getElementById(). Quand on accède à un élément via getElementById(), on le fait aussi au moyen d'une référence.

Ce qu'il faut retenir de tout ça, c'est que les objets du DOM sont *toujours* accessibles par référence, et c'est la raison pour laquelle ce code ne fonctionne pas :

```
var newDiv1 = document.createElement('div');
var newDiv2 = newDiv1; // On tente de copier le <div>
```

newDiv2 contient une référence qui pointe vers le même que newDiv1. Mais comment dupliquer un élément alors ? Il faut le cloner !

Cloner, remplacer, supprimer…

Cloner un élément

Pour cloner un élément, rien de plus simple : cloneNode(). Cette méthode requiert un paramètre booléen (true ou false) : si vous désirez cloner le nœud avec (true) ou sans (false) ses enfants et ses différents attributs.

Petit exemple très simple : on crée un élément <hr />, et on en veut un deuxième, donc on clone le premier :

```
// On va cloner un élément créé :
var hr1 = document.createElement('hr');
var hr2 = hr1.cloneNode(false); // Il n'a pas d'enfants…

// Ici, on clone un élément existant :
var paragraph1 = document.getElementById('myP');
var paragraph2 = paragraph1.cloneNode(true);

// Et attention, l'élément est cloné, mais pas « inséré » tant que l'on n'a pas appelé
appendChild() :
paragraph1.parentNode.appendChild(paragraph2);
```

Une chose très importante à retenir, la méthode cloneNode() ne copie pas les événements associés et créés avec le DOM (avec addEventListener()), même avec un paramètre à true.

Remplacer un élément par un autre

Pour remplacer un élément par un autre, rien de plus simple, il y a replaceChild(). Cette méthode accepte deux paramètres : le premier est le nouvel élément, et le deuxième est l'élément à remplacer. Tout comme cloneNode(), cette méthode s'utilise sur tous les types de nœuds (éléments, nœuds textuels, etc.).

Dans l'exemple suivant, le contenu textuel (pour rappel, il s'agit du premier enfant de <a>) du lien va être remplacé par un autre. La méthode replaceChild() est exécutée sur l'élément <a>, c'est-à-dire le nœud parent du nœud à remplacer.

```
<body>
    <div>
        <p id="myP">Un peu de texte <a>et un lien</a></p>
    </div>

    <script>
```

```
        var link = document.querySelector('a');
        var newLabel = document.createTextNode('et un hyperlien');

        link.replaceChild(newLabel, link.firstChild);
    </script>
</body>
```

Supprimer un élément

Pour insérer un élément, on utilise appendChild(), et pour en supprimer un, on utilise removeChild(). Cette méthode prend en paramètre le nœud enfant à retirer. Si on se calque sur le code HTML de l'exemple précédent, le script ressemble à ceci :

```
var link = document.querySelector('a');

link.parentNode.removeChild(link);
```

Il n'y a pas besoin de récupérer myP (l'élément parent) avec getElementById(), autant le faire directement avec parentNode.
À noter que la méthode removeChild() retourne l'élément supprimé, ce qui veut dire qu'il est parfaitement possible de supprimer un élément HTML pour ensuite le réintégrer où on le souhaite dans le DOM :

```
var link = document.querySelector('a');

var oldLink = link.parentNode.removeChild(link); // On supprime l'élément et on le garde en stock

document.body.appendChild(oldLink); // On réintègre ensuite l'élément supprimé où on veut et quand on veut
```

Autres actions

Vérifier la présence d'éléments enfants

Pour vérifier la présence d'éléments enfants : hasChildNodes(). Il suffit d'utiliser cette méthode sur l'élément de votre choix ; si cet élément possède au moins un enfant, la méthode renverra true:

```
<div>
      <p id="myP">Un peu de texte <a>et un lien</a></p>
</div>

<script>
    var paragraph = document.querySelector('p');

    alert(paragraph.hasChildNodes()); // Affiche true
</script>
```

Insérer à la bonne place : insertBefore()

La méthode insertBefore() permet d'insérer un élément avant un autre. Elle reçoit deux paramètres : le premier est l'élément à insérer, tandis que le deuxième est l'élément avant lequel l'élément va être inséré. Exemple :

```
<p id="myP">Un peu de texte <a>et un lien</a></p>

<script>
    var paragraph = document.querySelector('p');
    var emphasis = document.createElement('em'),
        emphasisText = document.createTextNode(' en emphase légère ');

    emphasis.appendChild(emphasisText);

    paragraph.insertBefore(emphasis, paragraph.lastChild);
</script>
```