4

Programmation DOM

Avant la standardisation du DOM, chaque navigateur Web implémentait ses propres méthodes afin de manipuler les éléments des pages HTML. Cela se traduisait par autant d'API qu'il y avait de navigateurs. Le code JavaScript était dès lors très difficilement portable sur l'ensemble des navigateurs.

Afin d'harmoniser ces différences, un groupe de travail du W3C s'est donné pour mission de spécifier une API de manipulation de l'arbre DOM.

L'API du DOM (Document Object Model) permet d'accéder à une page Web et de manipuler son contenu, sa structure ainsi que ses styles. Le DOM fournit pour cela une représentation objet normalisée des documents, dont le contenu est arborescent. Ces derniers peuvent être des pages HTML ou des documents XML.

Les spécifications du DOM sont indépendantes de tout langage de programmation. Elles comportent différentes versions, appelées niveaux. Dans le cadre de notre ouvrage, nous détaillons son utilisation avec le langage JavaScript.

Spécifications du DOM

À ce jour, trois spécifications du DOM ont été définies, correspondant à différents niveaux d'évolution, appelés 1, 2 et 3. Le niveau 0 fait référence à des fonctionnalités non spécifiées formellement et utilisées initialement par Internet Explorer 3.0 et Netscape Navigator 3.0.

Apparu en 1998, le DOM niveau 1 est en partie implémenté dans les versions 5 d'Internet Explorer et 6 de Netscape Navigator. Ce niveau permet la manipulation d'un document HTML ou XML.

Le DOM niveau 2, publié en 2000, apporte de nouvelles fonctionnalités, notamment l'ajout de vues filtrées, la gestion des événements et des feuilles de style et de nouvelles méthodes de parcours de l'arbre. Ce niveau correspond à la dernière version finalisée de la spécification.

Le DOM niveau 3 est en cours de spécification. Il ajoute une interface permettant les opérations de chargement et de sauvegarde du document sous forme XML. Il offre également la possibilité d'utiliser des expressions XPath afin d'exécuter des requêtes de recherche dans l'arbre. Il supporte en outre les espaces de nommage de XML.

Support par les navigateurs Web

Les navigateurs Web sont apparus bien avant la normalisation du DOM. De ce fait, leur support du DOM prend plus ou moins en compte les niveaux des spécifications et comporte généralement des extensions propriétaires non normalisées.

Pour des raisons de portabilité des applications, l'appel des méthodes des extensions propriétaires est déconseillé. Il est cependant parfois inévitable lors du développement de certaines fonctionnalités. Dans ce cas, il est préférable d'utiliser une couche d'abstraction afin de garantir le portage et de ne plus avoir à se soucier des problèmes d'incompatibilité entre navigateurs.

Le tableau 4.1 récapitule le support du DOM par les principaux navigateurs HTML.

Tableau 4.1 Support des spécifications du DOM par les navigateurs

Navigateur	Description
Internet Explorer 5+	Support partiel du niveau 1
Mozilla/Firefox	Support des niveaux 1 et 2 et partiel du niveau 3
Opera 7+	Support des niveaux 1 et 2 et partiel du niveau 3
Safari	Support des niveaux 1 et 2

Structure du DOM

Pour détailler la structure d'un arbre DOM ainsi que ses entités de base, nous nous appuyons sur les fonctionnalités présentes dans les niveaux 1 et 2 du DOM.

Tout langage de balises possède une structure logique arborescente dont la représentation objet correspond au DOM. Cette dernière fournit des méthodes permettant de parcourir cet arbre et éventuellement de le modifier.

Afin de détailler la structure du DOM, nous recourons tout au long des sections suivantes à un exemple simple de fragment HTML, qui comprend une balise racine `div` contenant deux blocs HTML, l'un délimité par la balise `span` et l'autre par une autre balise `div` :

```
<div id="maZone">
    <span><b>Un texte</b></span>
    <div id="monAutreZone">
         Un autre texte
    </div>
</div>
```

Ce fragment de code HTML correspond à la structure hiérarchique illustrée à la figure 4.1. L'objet `document` en est la racine. Les balises HTML sont reliées entre elles par des relations parent ou enfant.

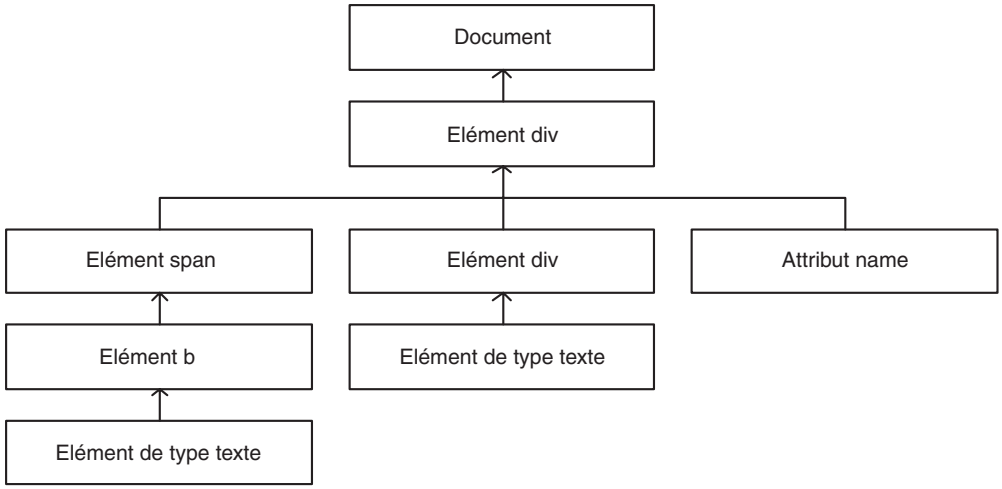

Figure 4.1
Structure de l'arbre DOM du code HTML

Avec le DOM, chacune des balises HTML possède une représentation objet correspondant à son nom. Dans l'exemple ci-dessus, la balise `div` est représentée par un objet de type `HTMLDivElement` et la balise `span` par `HTMLSpanElement`.

Nous pouvons remarquer que les nœuds peuvent être de différents types. Dans notre exemple, le deuxième nœud représente une balise et le cinquième un attribut de cette dernière.

La classe Node

La classe centrale représentant un nœud DOM est `Node`. Elle définit différentes propriétés afin de donner accès aux informations relatives au nœud, telles que son type et son nom.

Le tableau 4.2 récapitule les principales propriétés de la classe `Node`.

Tableau 4.2 Propriétés de la classe *Node*

Propriété	Description
`attributes`	Attributs du nœud
`nodeName`	Nom de la balise du nœud
`nodeType`	Type du nœud *(voir la section suivante)*
`nodeValue`	Valeur de la balise. Cette valeur peut être nulle lorsque aucune valeur n'est associée à la balise.
`ownerDocument`	Référence le document représentant l'arbre DOM dans lequel se trouve le nœud.

Dans notre exemple de fragment HTML, la balise identifiée par `maZone` est une `div`. La valeur de sa propriété `nodeType` est `1`.

Cette classe va de pair avec les classes `NodeList` et `NamedNodeMap`, qui définissent des collections de nœuds. Ces dernières sont utilisées lorsque des méthodes de manipulation de l'arbre DOM renvoient plusieurs nœuds ou attributs.

Différentes syntaxes permettent de parcourir une liste de nœuds, dont la plus simple consiste à utiliser la liste à la manière d'un tableau, comme dans l'exemple suivant :

```
var elements = (...)
for( var cpt=0; cpt<elements.length; cpt++ ){
    var element = elements[cpt];
    (...)
}
```

Une autre solution, plus orientée objet, consiste à utiliser la méthode `item` sur l'objet `NodeList` de la manière suivante :

```
var elements = (...)
for( var cpt=0; cpt<elements.length; cpt++ ){
    var element = elements.item(cpt);
    (...)
}
```

La classe `NamedNodeMap` est similaire à `NodeList` en ce qu'elle représente une liste de nœuds. Les éléments de cette dernière sont toutefois stockés par clé. Le code suivant en donne un exemple d'utilisation avec des attributs :

```
var attributs = (...)
for( var cpt=0; cpt<attributs.length; cpt++ ){
    var attribut = attributs.item(cpt);
    var nomAttribut = attribut.name;
    var valeurAttribut = attribut.value;
    (...)
}
```

Nous détaillons dans la suite du chapitre les différentes techniques permettant de récupérer une liste de nœuds et d'attributs.

La classe `Node` définit en outre des propriétés référençant d'autres nœuds de l'arbre. Nous verrons que ces dernières peuvent se révéler très utiles pour implémenter une navigation relative au nœud.

Types des nœuds

Un arbre DOM regroupe un ensemble de nœuds représentant différentes entités. Comme la classe `Node` est générique, elle contient une propriété `nodeType` dont la valeur est un entier.

Afin de faciliter l'utilisation de la propriété `nodeType`, la classe `Node` définit les constantes récapitulées au tableau 4.3.

Tableau 4.3 Constantes de la classe *Node* définissant les types de noeuds

Constante	Valeur	Description
ATTRIBUT_NODE	2	Attribut d'une balise de l'arbre
COMMENT_NODE	8	Balise de commentaire
DOCUMENT_FRAGMENT_NODE	11	Nœud racine d'un fragment d'arbre
DOCUMENT_NODE	9	Nœud racine de l'arbre. Il s'agit du type de l'objet document.
ELEMENT_NODE	1	Balise de l'arbre
TEXT_NODE	3	Nœud texte de l'arbre

DOCUMENT_NODE est le type du nœud racine de l'arbre DOM d'une page HTML. Cet élément est représenté par l'objet document de la page.

Dans notre exemple, la balise d'identifiant maZone est de type ELEMENT_NODE, et son attribut id de type ATTRIBUT_NODE. Le texte « un autre texte » est de type TEXT_NODE.

Manipulation des éléments

Nous allons à présent détailler les différentes API permettant de manipuler les éléments d'un arbre DOM.

Toutes les techniques décrites dans cette section correspondent au niveau 1 de la spécification.

Accès direct aux éléments

Pour accéder au nœud d'un document afin de pouvoir le manipuler par la suite, une méthode simple consiste à identifier ce nœud par un attribut id. La méthode getElementById de l'objet document peut ensuite se fonder sur ce dernier pour récupérer l'instance correspondante.

La balise div de notre bloc HTML exemple peut être référencée de la façon suivante :

```
var zone = document.getElementById("maZone");
```

Puisque l'identifiant d'une balise doit être unique dans une page HTML, la méthode getElementById renvoie un unique élément.

D'autres méthodes permettent de récupérer un ensemble de références sur des nœuds en se fondant soit sur l'objet document, soit sur un objet représentant un nœud.

La méthode getElementsByTagName, permet de récupérer une liste de nœuds en se fondant sur le nom des balises, lequel correspond à la valeur de la propriété nodeName de la classe Node décrite précédemment.

La méthode getElementsByTagName peut s'appliquer sur l'objet document ou sur un objet nœud, comme dans le code suivant :

```
var balisesDiv = document.getElementsByTagName("div");
```

Ce code permet de récupérer l'ensemble des balises `div` d'un document sous la forme d'une liste de nœuds (`NodeList`). La valeur `*` peut être utilisée comme argument afin de récupérer l'ensemble des nœuds du document, comme dans le code suivant :

```
var elements = document.getElementsBytagName("*");
```

La méthode `getElementsByName` retourne une liste de nœuds dont l'attribut `name` est égal à la valeur spécifiée en paramètre.

Cette méthode est particulièrement appropriée lors de la manipulation de cases d'option *(radio button)* d'un formulaire HTML, comme dans l'exemple suivant :

```
<form method="post" id="monFormulaire">
    <input type="radio" name="couleur" value="rouge"/>Rouge<br/>← ❶
    <input type="radio" name="couleur" value="bleu"/>Bleu<br/>← ❶
    <input type="radio" name="couleur" value="jaune"/>Jaune<br/>← ❶
</form>
```

Le code JavaScript suivant permet de référencer directement les trois balises `input` (repères ❶ dans le code précédent) avec le support de cette méthode :

```
var elementsInput = document.getElementsByName("couleur");
```

Les méthodes `getElementsByTagName` et `getElementsByName` permettent de récupérer une liste de nœuds dans une hiérarchie complète. Ces méthodes parcourent l'ensemble de l'arbre à partir d'un nœud de manière récursive.

Accès aux éléments à partir d'un nœud

La classe `Node` permet de parcourir un arbre DOM relativement à un nœud précis.

Elle possède à cet effet des propriétés spécifiques qui permettent de référencer les nœuds autour du nœud courant.

Le tableau 4.4 récapitule ces propriétés.

Tableau 4.4 Propriétés de la classe *Node* relatives aux nœuds dépendants

Propriété	Description
`childNodes`	Représente la liste des nœuds enfants sous forme d'objet `NodeList`. La méthode `hasChildNodes` permet de déterminer si le nœud a des nœuds enfants. La propriété `childNodes` ne contient pas les attributs du nœud.
`firstChild`	Référence le premier nœud enfant, correspondant au premier nœud de la liste `childNodes`.
`lastChild`	Référence le dernier nœud enfant, correspondant au dernier nœud de la liste `childNodes`.
`nextSibling`	Pointe sur l'enfant suivant dont le parent est identique. Elle contient `null` si le nœud est le dernier enfant.
`parentNode`	Référence le nœud parent.
`previousSibling`	Pointe sur l'enfant précédent dont le parent est identique. Elle contient `null` si le nœud est le premier enfant.

Jusqu'à présent, nous avons vu que l'accès aux nœuds consistait en la navigation au sein d'un objet `NodeList`. Les propriétés `firstChild`, `lastChild`, `previousSibling` et `nextSibling` permettent de s'abstraire de cette navigation en référençant directement certains nœuds.

Le code suivant donne un exemple de la façon d'accéder à ces nœuds :

```
var zone = document.getElementById("maZone");
var span = zone.firstChild;←❶
var div = zone.lastChild;←❷
```

La variable du repère ❶ référence la balise `span` et celle du repère ❷ la balise `div`.

L'utilisation des propriétés `previousSibling` et `nextSibling` offre la possibilité de parcourir un arbre DOM, comme dans le code suivant :

```
var zone = document.getElementById("maZone");
var element = zone.firstChild;
while( element!=null ){
    (...)
     element = element.nextSibling;
}
```

La propriété `childNodes` permet de référencer les différents enfants directs d'un nœud. Le code suivant indique comment récupérer l'ensemble des enfants du nœud d'identifiant `maZone` de notre fragment HTML :

```
var zone = document.getElementById("maZone");
var enfants = zone.childNodes;
```

Dans cet exemple, la variable `enfants` contient les nœuds relatifs aux balises `span` et `div` contenues dans la balise `div` d'identifiant `maZone`.

Spécificité de l'utilisation de la propriété *childNodes* avec Firefox

Dans le navigateur Firefox, la propriété `childNodes` renvoie l'ensemble des espaces comprises entre les balises HTML sous la forme de nœuds de type texte (`nodeType=TEXT_NODE`). Il faut donc veiller à prendre en compte ces nœuds texte lors des traitements.

Manipulation des nœuds

DOM définit des méthodes permettant de créer, d'ajouter, de remplacer ou de supprimer des nœuds de tout type. Ces méthodes sont fournies par l'objet `document`, qui décline les méthodes de création pour tous les types de nœuds.

Le code suivant fournit un exemple de création d'une balise, d'un nœud de type texte et d'un commentaire HTML :

```
var element = document.createElement("label");
var elementTexte = document.createTextNode("mon texte");
var commentaire = document.createComment("mon commentaire");
```

Une fois le nœud créé, il ne reste plus qu'à l'attacher à un nœud existant en utilisant les méthodes `appendChild` ou `insertBefore`, qui ajoutent un nœud enfant sous une balise.

Le code suivant se fonde sur notre code HTML exemple pour ajouter dynamiquement une balise `label` une fois la page chargée :

```
var zone = document.getElementById("maZone");
var label = document.createElement("label");
var texte = document.createTextNode("mon label");
label.appendChild(texte);
zone.appendChild(label);
```

Ce code permet de modifier l'arbre DOM en mémoire de la page afin d'obtenir le résultat suivant :

```
<div id="maZone">
    <span><b>Un texte</b></span>
    <div id="monAutreZone">
        Un autre texte
    </div>
    <label>mon label</label>← ❶
</div>
```

Le repère ❶ met en valeur le bloc ajouté. Notons que la méthode `appendChild` ne permet pas de spécifier à quel endroit le nœud enfant est ajouté. Ce dernier est toujours ajouté à la fin de la liste des enfants.

Lorsque nous souhaitons insérer dans un tableau une nouvelle ligne entre deux lignes existantes, l'utilisation de `insertBefore` est requise. Dans l'exemple précédent, cette méthode permet d'insérer la balise `label` directement en dessous de la balise `div` d'identifiant `maZone`, comme dans le code suivant :

```
var zone = document.getElementById("maZone");
var autreZone = document.getElementById("monAutreZone");
var label = document.createElement("label");
var texte = document.createTextNode("mon label");
label.appendChild(texte);
zone.insertBefore(autreZone, label);
```

Cet exemple permet d'obtenir l'arbre DOM suivant :

```
<div id="maZone">
    <span><b>Un texte</b></span>
    <label>mon label</label>
    <div id="monAutreZone">
        Un autre texte
    </div>
</div>
```

Tous les nouveaux nœuds que nous avons créés jusqu'à présent utilisaient les différentes méthodes `createXXX` de la classe `Document`. Il est aussi possible d'obtenir un nouveau nœud par clonage d'un nœud existant, en utilisant la méthode `cloneNode`. Le premier argument de cette méthode est un booléen, qui permet d'indiquer si seul le nœud doit être cloné ou si le nœud ainsi que tous ses enfants doivent l'être.

Le code suivant donne un exemple d'utilisation de la méthode `cloneNode` :

```
var zone = document.getElementById("maZone");
var zoneClone = zone.cloneNode(true) ; // clonage complet
var zonePartielle = zone.cloneNode(false); // clonage uniquement de la balise div
```

La méthode `removeChild` permet de supprimer un nœud. Le code suivant donne un exemple de suppression de la balise `div` d'identifiant `monAutreZone` :

```
var zone = document.getElementById("maZone");
var autreZone = document.getElementById("monAutreZone");
zone.removeChild(autreZone);
```

Utilisation des fragments d'arbre

Les fragments d'arbre permettent de travailler sur une portion de l'arbre. Ils sont représentés par l'objet `DocumentFragment`, qui peut être considéré comme un objet `Document` allégé. Il implémente toutes les méthodes de la classe `Node`.

Le code suivant permet de créer un nouveau fragment d'arbre :

```
var fragment = document.createDocumentFragment();
```

Les fragments d'arbre sont très utiles lorsqu'une nouvelle portion de l'arbre DOM doit être ajoutée.

Supposons que nous souhaitions ajouter sept paragraphes contenant le nom du jour de la semaine à la `div` de notre exemple. Les sept paragraphes peuvent être créés dans un fragment d'arbre. Ce dernier peut ensuite être ajouté sous la balise `div`.

Le code suivant fournit un exemple de création de paragraphes par l'utilisation d'un objet `DocumentFragment` :

```
var zone = document.getElementById("maZone");
var fragment = document.createDocumentFragment();
var jours = ["Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche"];
for (var i=0; i < jours.length; i++){
    var paragraphe = document.createElement("P");
    var texte = document.createTextNode(jours[i]);
    paragraphe.appendChild(texte);
    fragment.appendChild(paragraphe);
}
zone.appendChild(fragment);
```

Ce code permet d'obtenir l'arbre DOM suivant :

```
<div id="maZone">
    <p>Lundi</p>
    <p>Mardi</p>
    <p>Mercredi</p>
    <p>Jeudi</p>
    <p>Vendredi</p>
    <p>Samedi</p>
    <p>Dimanche</p>
</div>
```

Lorsque le fragment de cet exemple est inséré sous la balise `div`, ses nœuds enfants sont insérés, mais pas le fragment lui-même. Le fragment est utilisé uniquement comme container et n'existe plus en tant que tel une fois qu'il a été inséré dans l'arbre DOM.

Le résultat de cet exemple peut aussi être obtenu sans utiliser de fragment d'arbre. Dans ce cas, des balises `p` sont insérées au fur et à mesure de leur création sous la balise `div` par des `appendChild`. Chacune des insertions dans l'arbre DOM provoque le rafraîchissement complet de la page. Dans notre cas, la page est donc rafraîchie sept fois.

En cas d'utilisation d'un fragment d'arbre, la page n'est rafraîchie que lors de l'insertion du fragment dans le DOM. Lors de la construction du fragment, les nœuds sont uniquement présents en mémoire et ne sont pas affichés. Ils ne sont réellement affichés qu'au moment de l'ajout du fragment sur le DOM de la page.

L'utilisation de fragments d'arbre améliore sensiblement les performances lors de l'insertion de plusieurs nœuds puisqu'elle réduit le nombre de rafraîchissement du document HTML affiché.

Manipulation des attributs

Tout élément du type `NODE_ELEMENT` peut contenir des attributs permettant de lui ajouter des informations. Un attribut possède une clé et une valeur, la valeur de la clé devant être unique pour l'élément.

Le code suivant met en œuvre les attributs dans notre fragment HTML exemple :

```
<div id="maZone">
    <span><b>Un texte</b></span>
    <div id="monAutreZone">
        Un autre texte
    </div>
</div>
```

`id` est un attribut de la balise `div`. La classe `Node` met à disposition la propriété `attributes` afin de lister ses différents attributs. Cette propriété est du type `NamedNodeMap`, décrit précédemment.

La classe `Node` fournit également l'ensemble de méthodes récapitulées au tableau 4.5.

Tableau 4.5 Méthodes de manipulation des attributs de la classe *Node*

Méthode	Paramètre	Description
`getAttribute`	Identifiant de l'attribut	Référence un attribut d'un élément en utilisant son identifiant.
`hasAttribute`	Identifiant de l'attribut	Détermine si un attribut est présent pour un élément.
`removeAttribute`	Identifiant de l'attribut	Supprime un attribut pour un élément.
`setAttribute`	Identifiant de l'attribut ainsi que sa valeur	Crée un attribut ou remplace un attribut existant d'un élément.

La méthode `setAttribute` offre la possibilité d'ajouter un attribut à une balise. Le code suivant donne un exemple d'utilisation de cette technique afin d'ajouter un attribut d'identifiant `type` pour la première balise `div` de notre fragment HTML :

```
var zone = document.getElementById("maZone");
zone.setAttribute("type", "UnType");
```

La méthode `getAttribute` donne accès à la valeur d'un attribut. Dans le cas où l'attribut n'existe pas, la méthode renvoie `null`.

Le code suivant permet de récupérer la valeur de l'attribut `type` précédemment ajouté :

```
var elt = document.getElementById("d1");
var type = elt.getAttribute("type");
```

Les méthodes `hasAttribute` et `removeAttribute` permettent respectivement de vérifier la présence d'un attribut et de supprimer un attribut, comme dans le code suivant :

```
var zone = document.getElementById("maZone");
if( zone.hasAttribute("type") ) {
    zone.removeAttribute("type");
}
```

Les attributs peuvent être utilisés sur des nœuds afin d'ajouter des données supplémentaires accessibles pour les traitements sans impacter le rendu visuel. Ces attributs sont souvent utilisés pour stocker des éléments techniques sur le nœud.

Parcours de l'arbre DOM

Le DOM niveau 2 introduit de nouvelles fonctionnalités, appelées DOM Traversal et Range, permettant de parcourir un arbre DOM. Elles correspondent essentiellement aux classes `NodeIterator` et `TreeWalker`.

La classe `NodeIterator` est très utile pour parcourir un arbre DOM. Nous allons utiliser le fragment suivant afin d'illustrer ses capacités de navigation :

```
<div id="maZone">
    <span><b>Un texte</b></span>
    <div id="monAutreZone">
        <p>un paragraphe</p>
        <p>deuxième paragraphe</p>
    </div>
    <div id="encoreUneZone">
        <b>texte en gras</b>
    </div>
</div>
```

La classe `NodeIterator` permet de parcourir l'ensemble des nœuds de ce fragment par le biais d'une méthode dite de *parcours de l'arbre en profondeur préfixé*.

Avant d'utiliser la classe `NodeIterator`, il faut la créer. Il suffit pour cela d'utiliser la méthode `createNodeIterator` sur l'objet `Document`. Les arguments de cette méthode de création

permettent de définir les nœuds qui doivent être explorés. Ces arguments sont récapitulés au tableau 4.6.

Tableau 4.6 Arguments de la méthode *createNodeIterator*

Argument	Description
`root`	Un document ou un nœud à parcourir
`whatToShow`	Précise les types de nœuds qui doivent être parcourus. Les valeurs possibles de cet argument sont recensées au tableau 4.7.
`filter`	Classe `NodeFilter` contenant les caractéristiques d'un filtre ou `null` si aucun filtre n'est nécessaire.
`entityReferenceExtension`	Booléen indiquant si les entités référencées doivent être explorées.

Les valeurs les plus usuelles de l'argument `whatToShow` sont récapitulées au tableau 4.7.

Tableau 4.7 Principales valeurs de l'argument *whatToShow*

Valeur	Description
`NodeFilter.SHOW_ALL`	Retourne tous les nœuds.
`NodeFilter.SHOW_COMMENT`	Retourne uniquement les nœuds de type `COMMENT_NODE`.
`NodeFilter.SHOW_ELEMENT`	Retourne uniquement les nœuds de type `ELEMENT_NODE`.
`NodeFilter.SHOW_TEXT`	Retourne uniquement les nœuds de type `TEXT_NODE`.

Le code suivant donne un exemple de création d'un objet `NodeIterator` :

```
var zone = document.getElementById("maZone");
var parcours = document.createNodeIterator(zone, NodeFilter.SHOW_ELEMENT, null, false);
```

Il ne reste plus qu'à se déplacer dans l'objet `NodeIterator` en utilisant les méthodes `nextNode` ou `previousNode`.

L'exemple ci-dessous indique comment utiliser ces deux méthodes :

```
var monSpan = parcours.nextNode();
var maBaliseB = parcours.nextNode();
```

À ce stade, l'appel de la méthode `previousNode` permet de revenir sur le nœud représentant la balise `span` :

```
var encoreMonSpan = parcours.previousNode();
```

Le code suivant donne un exemple de parcours complet de notre fragment HTML de départ :

```
var zone = document.getElementById("maZone") ;
var parcours = document.createNodeIterator(zone, NodeFilter.SHOW_ELEMENT, null, false);
var sortie ;
var nœud = parcours.nextNode();
while (nœud){
    sortie += noeud.nodeName + " ";
    noeud = parcours.nextNode();
}
```

La variable `sortie` contient la chaîne `SPAN B DIV P P DIV B`.

Jusqu'à présent, nous n'avons pas utilisé la classe `NodeFilter`. Cette dernière permet d'ajouter un filtre supplémentaire permettant d'exclure certains nœuds lors du parcours de l'arbre. Cette classe est définie par une unique méthode `acceptNode`. Le code de retour de cette méthode permet de rejeter ou de garder le nœud dans le `NodeIterator`.

La création d'un objet `NodeFilter` est très simple, comme le montre le code suivant :

```
var oFilter = new Object() ;
oFilter.acceptNode = function (oNode){}
```

La fonction `acceptNode` doit retourner la valeur `NodeFilter.FILTER_REJECT` ou `NodeFilter.FILTER_ACCEPT` afin d'indiquer si le nœud passé en argument doit faire partie ou non du `NodeIterator`.

À partir de notre fragment de code HTML, il est possible de ne garder dans un `NodeIterator` que les balises `p`, comme dans le code suivant :

```
var oFilterP = new Object();
oFilterP.acceptNode = function (oNode){
    if (oNode.nodeName == "P"){
        return NodeFilter.FILTER_ACCEPT;
    } else {
        return NodeFilter.FILTER_REJECT;
    }
}
```

Il ne reste plus qu'à créer un `NodeIterator` à partir de ce filtre :

```
var parcours = document.createNodeIterator(zone, NodeFilter.SHOW_ALL, oFilterP, false);
```

Il est également possible de parcourir un arbre DOM en utilisant la classe `TreeWalker`. Cette dernière offre les mêmes fonctionnalités que la classe `NodeIterator` mais comporte des méthodes de déplacement supplémentaires.

La méthode `createTreeWalker` de l'objet `document` permet de créer un `TreeWalker` (cette méthode de création porte la même signature que son homologue `CreateNodeIterator`) :

```
var walker = document.createTreeWalker(zone, NodeFilter.SHOW_ELEMENT, null, false);
```

Les méthodes permettant de naviguer dans un `TreeWalker` sont recensées au tableau 4.8.

Tableau 4.8 Méthodes de navigation de la classe *TreeWalker*

Méthode	Description
`firstChild`	Retourne le premier enfant du nœud courant.
`lastChild`	Retourne le dernier enfant du nœud courant.
`nextNode`	Retourne le nœud suivant (identique à `NodeIterator.nextNode`).
`nextSibling`	Retourne l'enfant suivant dont le parent est identique au nœud courant.
`parentNode`	Retourne le nœud parent du nœud courant.
`previousNode`	Retourne le nœud précédent (identique à `NodeIterator.previousNode`).
`previousSibling`	Retourne l'enfant précédent dont le parent est identique au nœud courant.

Le code suivant fournit un exemple d'utilisation de ces méthodes :

```
var walker = document.createTreeWalker(zone, NodeFilter.SHOW_ELEMENT, null, false);
var monSpan = walker.firstChild();←❶
var divMonAutreZone = walker.nextSibling();←❷
var premierP = walker.firstChild();
```

La ligne identifiée par le repère ❶ permet de récupérer la balise `span`. La balise `div` portant l'identifiant `monAutreZone` est récupérée au repère ❷.

Les classes `NodeIterator` et `TreeWalker` simplifient le parcours d'un arbre DOM en offrant deux niveaux de filtre. Le premier, `whatToShow`, est un filtre sur le type de nœud, et le second, `NodeFilter`, un filtre défini par programmation.

Notons que ces classes ne sont pas supportées par Internet Explorer.

L'attribut *innerHTML*

Bien que l'attribut `innerHTML`, présent sur les nœuds d'un arbre DOM, ne soit pas normalisé par le W3C, cet ouvrage se doit de le présenter. Cet attribut permet en effet de remplacer complètement le contenu d'un élément par celui spécifié dans une chaîne de caractères.

Le code ci-dessous donne un exemple d'utilisation de l'attribut `innerHTML` afin de remplacer le contenu de la balise `span` définie en début de chapitre par un ensemble de balises HTML :

```
var zone = document.getElementById("maZone");
var span = zone.getElementsByTagName("span")[0];
span.innerHTML = "<p><b>mon texte</b></p>";
```

Ce code permet de modifier l'arbre DOM en mémoire de la page afin d'obtenir le résultat suivant :

```
<div id="maZone">
    <span><p><b>Un texte</b></p></span>←❶
    <div id="monAutreZone">
        Un autre texte
    </div>
</div>
```

Le repère ❶ met en valeur le bloc modifié. Nous constatons qu'une balise `p` a été ajoutée autour de la balise `b`.

L'utilisation de l'attribut `innerHTML` sur un élément est particulièrement tentante, puisqu'elle évite de multiples appels à la méthode `appendChild` et autant de créations de nouveaux nœuds. Elle force cependant à construire des chaînes de caractères contenant un ensemble de balises ainsi que leurs propriétés (événements, styles, etc.) sous la forme de multiples concaténations de chaînes. Or ces chaînes nuisent à la maintenabilité du

code JavaScript et peuvent conduire à des syntaxes HTML invalides. De plus, l'utilisation de cette méthode rompt la structuration définie dans les spécifications du DOM.

Au vu de ces considérations, l'utilisation de cette méthode n'est recommandée que dans des cas bien particuliers, et il ne faut surtout pas en abuser.

L'attribut `innerHTML` peut être invoqué pour résoudre des problèmes de performance résultant de l'ajout d'un grand nombre de nœuds. Cette problématique de performance peut également être résolue par l'utilisation de fragments d'arbre. Dans la quasi-totalité des navigateurs Web, l'utilisation de l'attribut `innerHTML` est beaucoup plus performante que celle de la méthode `appendChild` pour ajouter de nouveaux nœuds à l'arbre DOM.

Nous allons illustrer l'utilisation des deux approches.

La première correspond à l'utilisation des méthodes du DOM afin de créer dynamiquement un tableau composé de trois cents lignes :

```
var nbLigne = 3000;
var zone = document.getElementById("maZone");
var elementTableau = document.createElement("table");
var elementTBody = document.createElement("tbody");
elementTableau.appendChild(elementTBody);
var elementTr = document.createElement("tr");
var elementTd = document.createElement("td");
var elementDonnees = document.createTextNode("Mes données");
for( var cpt=0; cpt<nbLigne; cpt++ ){
    var tmpElementTr = elementTBoby.appendChild(elementTr.cloneNode(true));
    var tmpElementTd = tmpElementTr.appendChild(elementTd.cloneNode(true));
    tmpElementTd.appendChild(elementDonnees.cloneNode(true));
    }
zone.appendChild(elementTableau);
```

La seconde correspond à l'utilisation de l'attribut `innerHTML` afin de mettre en œuvre la même fonctionnalité :

```
var nbLigne = 3000;
var zone = document.getElementById("maZone");
var html = new Array();
html.push("<table><tbody>");
for( var cpt=0; cpt< nbLigne;cpt++ ){
    html.push("<tr>");
    html.push("<td>Mes données</td>");
    html.push("</tr>");
}
html.push("</tbody></table>");
zone.innerHTML = html.join("");
```

Les temps de réponse d'affichage du tableau sont nettement supérieurs avec l'attribut `innerHTML`.

Utilisation du DOM niveau 0

Comme indiqué précédemment, le DOM niveau 0 n'est pas normalisé par le W3C. Il correspond à la première implémentation par les navigateurs Web de l'accès aux objets d'un document HTML.

Ce DOM niveau 0 permet d'accéder à différents éléments d'un document HTML par l'intermédiaire de propriétés de l'objet `document`. Par exemple, la propriété `forms` permet de récupérer toutes les balises `form` contenues dans une page Web.

Le code suivant indique comment utiliser la propriété `forms` :

```
var mesFormulaires = document.forms;
for (var i=0 ; i < mesFormulaires.length ; i++){
    var formulaire = mesFormulaires[i];
}
```

De la même manière, toutes les images d'un document sont accessibles par le biais de la propriété `images` :

```
var mesImages = document.images;
for (var i=0 ; i < mesImages.length ; i++){
    var image = mesImages[i];
}
```

La propriété `links` permet de récupérer tous les liens (balise `a`) d'un document.

Bien que le DOM niveau 0 ne soit pas normalisé, les dernières versions des navigateurs implémentent en partie l'accès à ses propriétés. Il est cependant recommandé de ne pas les utiliser et de leur préférer la méthode `getElementsByTagName`.

Les propriétés `document.forms` et `documents.images` peuvent être remplacées par le code suivant :

```
var mesFormulaires = document.getElementsByTagName("form");
var mesImages = document.getElementsByTagName("img");
```

Modification de l'arbre DOM au chargement

Comme nous l'avons vu jusqu'à présent, le DOM permet de manipuler la structure d'une page HTML. Il faut toutefois garder à l'esprit que le DOM n'est accessible que lorsque la page est chargée dans sa globalité. Les traitements sur l'arbre DOM doivent donc être effectués après le déclenchement de l'événement `onload` du document afin de garantir la présence de tous les éléments.

Le DOM peut également être utilisé pour modifier le contenu d'une page HTML après son chargement. Cette fonctionnalité se révèle particulièrement pratique pour la construction de widgets à partir de leur description dans la page.

Pour mieux comprendre la modification de l'arbre DOM au chargement, nous allons l'illustrer par un exemple concret, dans lequel nous souhaitons remplacer les boutons

standards définis par la balise `input` par une balise `div` afin d'améliorer l'esthétique des boutons.

L'exemple suivant indique comment remplacer des boutons au chargement de la page :

```
<html>
<head>
    <script language="text/javaScript">
        function remplaceBouton(){
            var oInputs = document.getElementsByTagName("input");
            for (var i=0; i < oInputs.length; i++){
                var oInput = oInputs.item(i);
                if (oInput.getAttribute("type") == "button"){
                    // construction du nouveau bouton
                    var oDiv = document.createElement("div");
                    oDiv.style.border = "1px solid blue";
                    var t = document.createTextNode(oInput.value);
                    oDiv.appendChild(t);
                    // modification d'une noeud existant
                    oInput.parentNode.replaceChild(oDiv, oInput);
                }
            }
        }
    </script>
</head>
<body onload="remplaceBouton()">
<h1>remplacement des boutons par des div</h1>
<input type="button" value="mon action"/>←❶
</body>
</html>
```

Dans ce code, la méthode `remplaceBouton` est exécutée lorsque la page est complètement chargée. Elle récupère toutes les balises `input` de type `button` afin de les remplacer par une balise `div`.

Après le chargement de la page, la balise du repère ❶ devient :

```
<div style="border : 1px solid blue">mon action</div>
```

Cet exemple de modification du DOM au chargement est des plus simple. Des cas plus complexes peuvent être traités de la même manière. Il est notamment possible de décrire les éléments graphiques constituant l'interface homme machine (IHM) par des balises non interprétées par le navigateur Web. Elles sont ensuite prises en compte après le chargement complet de la page afin d'obtenir le rendu visuel des composants graphiques. C'est ce principe qu'utilise la bibliothèque dojo pour construire ses widgets.

L'exemple de code suivant construit un widget de type tableau au chargement de la page :

```
<div widgetType="grid">
    <column title="Nom" style="width:40%"/>
    <column title="Prénom" style="width:60%"/>
</div>
```

Dans les balises HTML ci-dessus, l'attribut `widgetType` permet de définir la balise `div` comme étant un élément à modifier au chargement. Les balises `column` permettent de définir les colonnes du tableau.

Le code suivant, lancé après chargement de la page, permet d'interpréter l'ensemble des balises avec un attribut `widgetType` :

```
function buildWidget(){
    var nodes = document.getElementsByTagName("*");
    for (var i=0; i < nodes.length; i++){
        var node = nodes[i];
        if (node.nodeType == 3){
            var widgetType = node.getAttribute("widgetType");
            if ("grid" == widgetType){
                constructGrid(node);
            } else if ("onglet" == widgetType){
                (...)
            }
            (...)
        }
    }
}
```

La fonction `buildWidget` parcourt toutes les balises de l'arbre DOM et déclenche un traitement spécifique lorsqu'une balise contient un attribut `widgetType`, en l'occurrence la fonction `constructGrid`, chargée de la construction du tableau :

```
function constructGrid(baseNode){
    // construction d'un fragment
    var grid = document.createDocumentFragment();
    var oTable = document.createElement("table");
    var oTbdy = document.createElement("tbody");
    oTable.appendChild(oTbdy);
    var oEntete = document.createElement("tr");
    oTbdy.appendChild(oEntete);
    // interprétation des balises column
    var columns = baseNode.getElementsByTagName("column");
    for (var i = 0 ; i < columns.length; i++){
        var column = columns[i];
        var titre = column.getAttribute("title");
        var oTd = document.createElement("td");
        oTd.style.cssText = column.style.cssText;
        var txt = document.createTextNode(titre);
        oTd.appendChild(txt) ;
        oEntete.appendChild(oTd);
        (...)
    }
    baseNode.appendChild(grid);
}
```

La fonction `constructGrid` interprète les balises `column` afin de créer les différentes colonnes du tableau. Le code de cet exemple n'est évidemment pas complet, mais il permet de se rendre compte du potentiel de la modification du DOM au chargement.

Conclusion

Le DOM normalise une API permettant de fournir une représentation objet de la structure d'un document hiérarchique, la plupart du temps fondé sur un langage de balises. Il offre également un support afin de le manipuler et de le parcourir.

Dans ce chapitre, nous avons décrit les différents objets de l'arbre logique DOM. Nous les avons mis en œuvre afin d'interagir avec l'arbre DOM d'une page HTML. L'API DOM offre la possibilité de parcourir cet arbre et de le modifier dynamiquement, même après le chargement des pages HTML.

Le DOM s'avère bien adapté pour mettre en œuvre des composants graphiques en travaillant directement sur l'arbre DOM.

Tous les navigateurs Web n'implémentent cependant pas encore l'ensemble des spécifications du DOM. Une couche d'abstraction (bibliothèque, framework, boîte à outils, etc.) est donc souvent nécessaire pour résoudre les problèmes d'implémentation.