JS

Navigateur : Document, Évènements, Interfaces

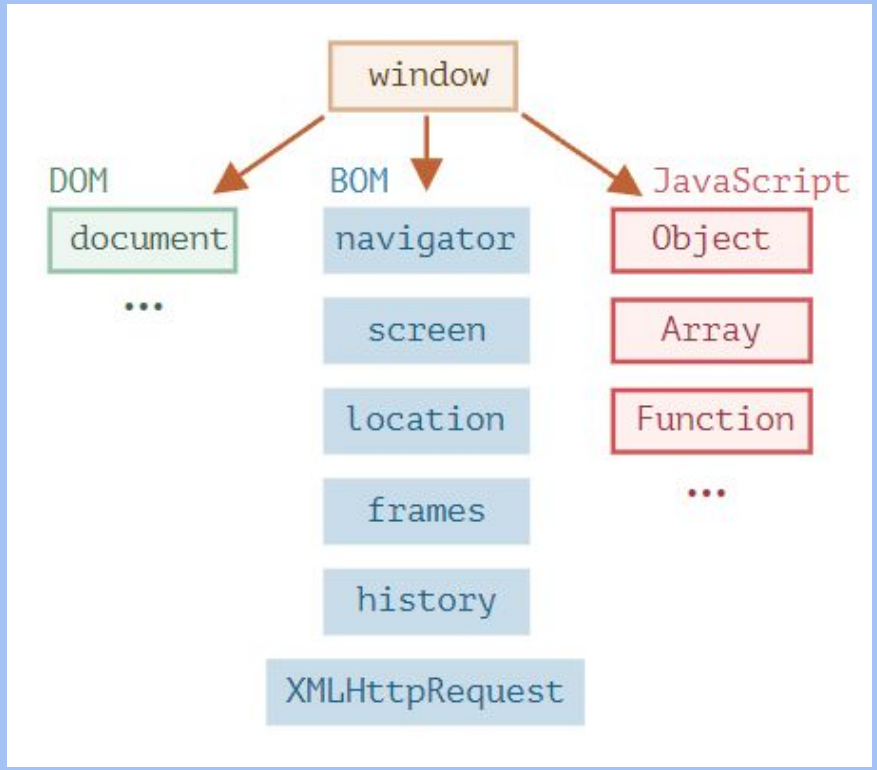
L'environnement du navigateur, spécifications

Le langage JavaScript a été initialement créé pour les navigateurs web. Dès lors, il a évolué et est devenu un langage aux multiples utilisations et plateformes.

Une plate-forme peut être un navigateur, ou un serveur Web ou un autre hôte, ou même une machine à café "intelligente" si elle peut exécuter JavaScript. Chacun d'entre eux fournit des fonctionnalités spécifiques à la plate-forme. La spécification JavaScript appelle cela un environnement hôte.

Un environnement hôte fournit ses propres objets et fonctions en plus du noyau du langage. Les navigateurs Web permettent de contrôler les pages Web. Node.js fournit des fonctionnalités côté serveur, etc.

Voici une vue globale de ce que nous avons lorsque JavaScript s'exécute dans un navigateur Web :



L'environnement du navigateur, spécifications

Il existe un objet "racine" appelé **window**. Il a 2 rôles :

- Premièrement, c'est un objet global pour le code JavaScript, comme décrit dans le chapitre L'objet global.
- Deuxièmement, il représente la "fenêtre du navigateur" et fournit des méthodes pour la contrôler.

Par exemple, nous l'utilisons ici comme un objet global :

```
function sayHi() {
  alert("Hello");
}

// les fonctions globales sont des méthodes de l'objet global :
window.sayHi();
```

Et nous l'utilisons ici comme une fenêtre du navigateur pour voir la hauteur de la fenêtre :

```
alert(window.innerHeight); // hauteur de la fenêtre intérieure
```

Il y a d'autres méthodes et propriétés spécifiques à la fenêtre, nous les étudierons plus tard.

DOM (Document Object Model)

Document Object Model, ou **DOM** en abrégé, représente tout le contenu de la page sous forme d'objets pouvant être modifiés.

L'objet **document** est le "point d'entrée" principal de la page. Nous pouvons changer ou créer n'importe quoi sur la page en l'utilisant.

Par exemple :

```
// change la couleur de fond en rouge
document.body.style.background = "red";

// réinitialisation après 1 seconde
setTimeout(() => document.body.style.background = "", 1000);
```

CSSOM pour le style

Il existe également une spécification distincte, **Modèle d'objet CSS** (**CSSOM**) pour les règles CSS et les feuilles de style, qui explique comment elles sont représentées en tant qu'objets et comment les lire et les écrire.

CSSOM est utilisé avec DOM lorsque nous modifions les règles de style du document. En pratique cependant, CSSOM est rarement nécessaire, car nous avons rarement besoin de modifier les règles CSS à partir de JavaScript (généralement, nous ajoutons / supprimons simplement des classes CSS, pas de modifier leurs règles CSS), mais c'est également possible.

BOM (Browser Object Model)

Le **modèle d'objet du navigateur** (**BOM** en anglais) contient des objets supplémentaires fournis par le navigateur (l'environnement hôte) pour travailler avec tout à l'exception du document.

Par exemple :
L'objet **navigator** fournit des informations contextuelles à propos du navigateur et du système d'exploitation. Il y a beaucoup de propriétés mais les deux plus connues sont : **navigator.userAgent** – qui donne des informations sur le navigateur actuel, et **navigator.platform** sur la plateforme (peut permettre de faire la différence entre Windows/Linux/Mac etc).
L'objet **location** nous permet de lire l'URL courante et peut rediriger le navigateur vers une nouvelle adresse.
Voici comment l'on peut utiliser l'objet location :

```
alert( location.href ); // affiche l'URL actuelle
if ( confirm("Go to Wikipedia?") ) {
  location.href = "https://wikipedia.org"; // rediriger le navigateur vers une autre URL
}
```

Les fonctions **alert/confirm/prompt** font aussi partie du BOM : elles ne sont pas directement liées au document, mais représentent des méthodes du navigateur de communication pure avec l'utilisateur.

L'arbre DOM

L'épine dorsale d'un document HTML est constituée de balises.
Selon le modèle d'objets de document (**DOM**), chaque balise **HTML** est un objet. Les balises imbriquées sont des "enfants" de celle qui les entoure. Le texte à l'intérieur d'une balise est également un objet.
Tous ces objets sont accessibles via **JavaScript**, et nous pouvons les utiliser pour modifier la page.
Par exemple, **document.body** est l'objet représentant la balise **<body>**.
L'exécution de ce code rendra le **<body>** rouge pendant 3 secondes :

```
// change la couleur de fond en rouge
document.body.style.background = "red";

// réinitialisation après 1 seconde
setTimeout(() => document.body.style.background = "", 3000);
```

Ici, nous avons utilisé **style.background** pour changer la couleur d'arrière-plan de **document.body**, mais il existe de nombreuses autres propriétés, telles que :

- **innerHTML** – Contenu HTML du nœud.
- **offsetWidth** – la largeur du nœud (en pixels)
- ... etc.

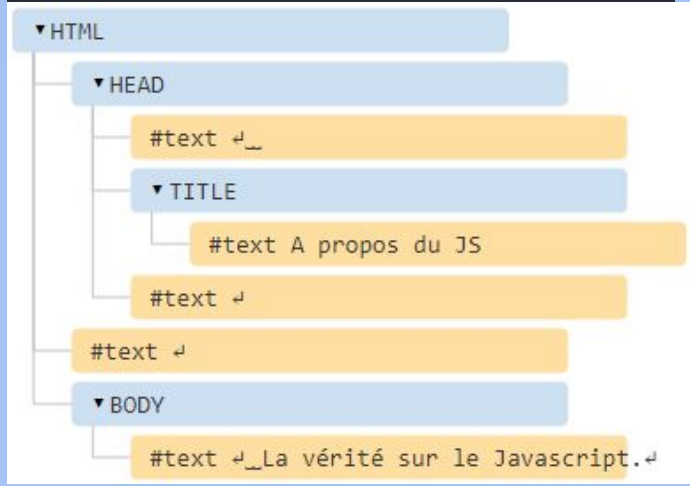
Bientôt, nous apprendrons plus de façons de manipuler le DOM, mais nous devons d'abord connaître sa structure.

L'arbre DOM

Un exemple du DOM
Commençons par le simple document suivant :

```
<!DOCTYPE HTML>
<html>
<head>
  <title>A propos du JS</title>
</head>
<body>
 La vérité sur le Javascript.
</body>
</html>
```

Le DOM représente le HTML comme une structure arborescente de balises.
Voici à quoi ça ressemble :



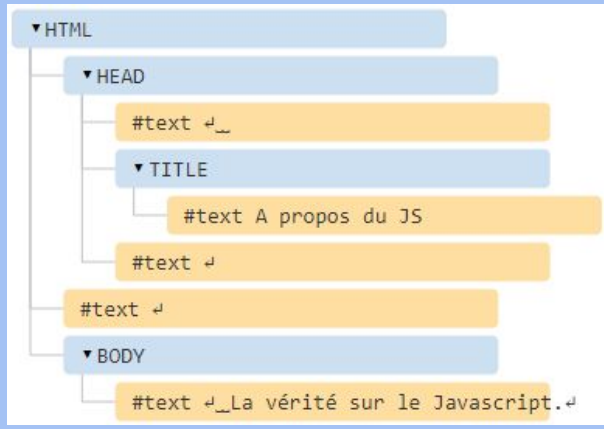
Sur l'image ci-contre, vous pouvez cliquer sur les nœuds des éléments et leurs enfants s'ouvriront/se réduiront.
Chaque nœud de l'arbre est un objet.

Les balises sont des nœuds d'élément (ou simplement des éléments) et forment la structure arborescente : <html> est à la racine, puis <head> et <body> sont ses enfants, etc.

Le texte à l'intérieur des éléments forme des nœuds texte, étiquetés comme #text. Un nœud texte ne contient qu'une chaîne de caractères. Il peut ne pas avoir d'enfants et est toujours une feuille de l'arbre.

L'arbre DOM

Par exemple, la balise **<title>** a le texte "À propos du JS".



Veuillez noter les caractères spéciaux dans les nœuds texte :

- une nouvelle ligne : ↵ (en JavaScript appelé \n)
- un espace : ␣

Les espaces et les nouvelles lignes sont des caractères totalement valides, comme les lettres et les chiffres. Ils forment des nœuds de texte et deviennent une partie du DOM. Ainsi, par exemple, dans l'exemple ci-dessus, la balise **<head>** contient des espaces avant **<title>**, et ce texte devient un nœud **#text** (il contient une nouvelle ligne et quelques espaces uniquement).

Il n'y a que deux exclusions de haut niveau :

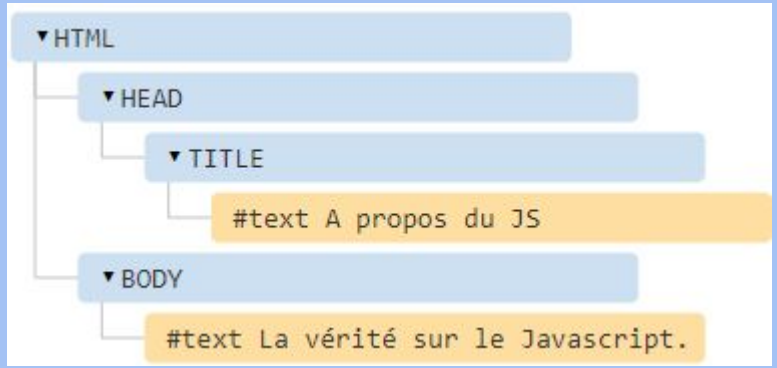
1. Les espaces et les nouvelles lignes avant **<head>** sont ignorés pour des raisons historiques.
2. Si nous mettons quelque chose après **</body>**, alors cela est automatiquement déplacé à l'intérieur du body, à la fin, car la spécification HTML exige que tout le contenu soit à l'intérieur de <body>. Il ne peut donc pas y avoir d'espace après </body>.

L'arbre DOM

Dans d'autres cas, tout est simple – s'il y a des espaces (comme n'importe quel caractère) dans le document, alors ils deviennent des nœuds texte dans le DOM, et si nous les supprimons, il n'y en aura pas.

Voici des nœuds de texte sans espace :

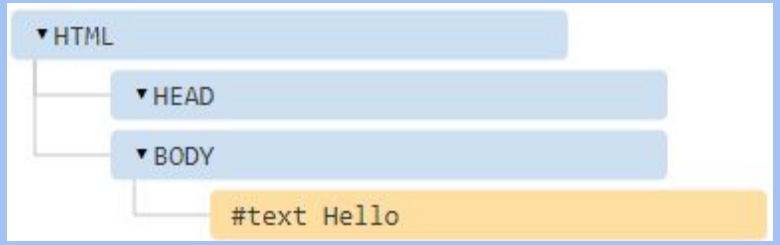
```
<!DOCTYPE HTML>
<html><head><title>A propos du JS</title></head><body>La vérité sur le Javascript.</body></html>
```



L'arbre DOM

Auto-correction

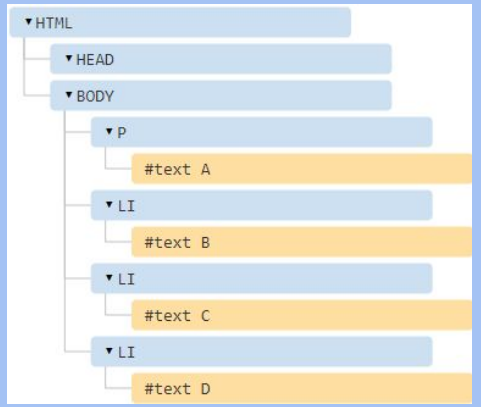
Si le navigateur rencontre du HTML mal formé, il le corrige automatiquement lors de la création du DOM. Par exemple, la balise la plus haute est toujours <html>. Même si elle n'existe pas dans le document, elle existera dans le DOM, car le navigateur la créera. Il en va de même pour <body>.
Par exemple, si le fichier HTML est le seul mot "Hello", le navigateur l'enroulera dans <html> et <body>, et ajoutera le <head> requis, et le DOM sera :



Lors de la génération du DOM, les navigateurs traitent automatiquement les erreurs dans le document, ferment les balises, etc.

Un document avec des balises non fermées :

```
<li>A
<li>B
<li>C
<li>D
```



L'arbre DOM

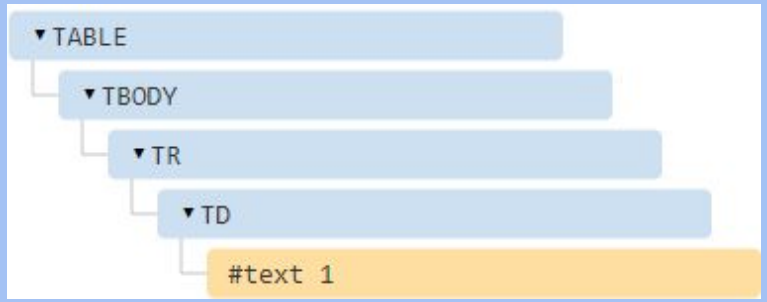
Les tableaux ont toujours <tbody>

Un "cas spécial" intéressant est celui des tableaux. Selon la spécification DOM, ils doivent avoir un <tbody>, mais le texte HTML peut (officiellement) l'omettre. Ensuite, le navigateur crée automatiquement le <tbody> dans le DOM.

Pour le HTML :

```
<table id="table"><tr><td>1</td></tr></table>
```

La structure du DOM sera :



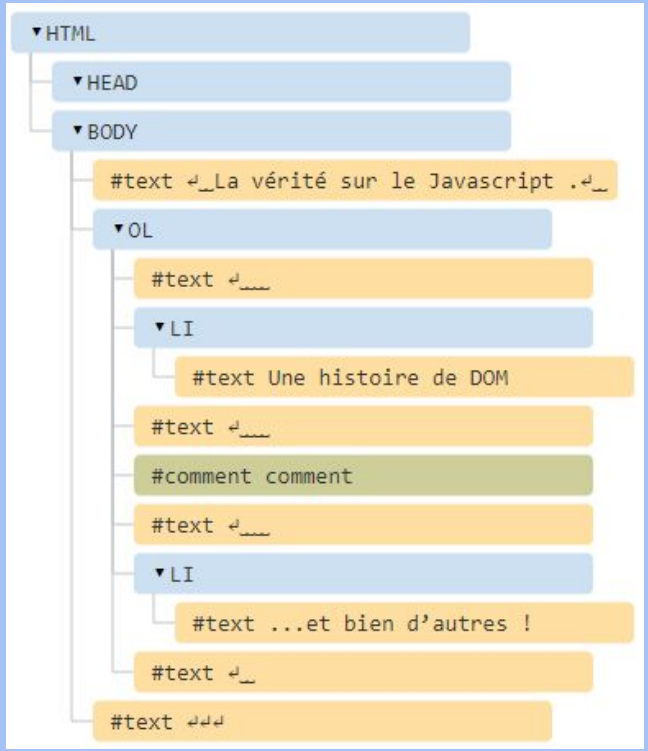
Vous voyez ? Le **<body>** est sorti de nulle part. Vous devez garder cela à l'esprit lorsque vous travaillez avec des tableaux pour éviter les surprises.

L'arbre DOM

Autres types de nœuds

Il existe d'autres types de nœuds en plus des éléments et des nœuds de texte.
Par exemple, les commentaires :

```
<!DOCTYPE HTML>
<html>
<body>
 La vérité sur le Javascript.
 <ol>
    <li>Une histoire de DOM</li>
    <!-- comment -->
    <li>...et bien d'autres !</li>
  </ol>
</body>
</html>
```



L'arbre DOM

Nous pouvons voir ici un nouveau type de nœud de l'arbre – nœud commentaire, étiqueté comme **#comment**, entre deux nœuds texte.
Nous pouvons penser – pourquoi un commentaire est-il ajouté au DOM ? Cela n'affecte en rien la représentation visuelle. Mais il y a une règle – **si quelque chose est en HTML, alors il doit aussi être dans l'arborescence DOM**.

Tout en HTML, même les commentaires, devient une partie du DOM.

Même la directive <!DOCTYPE...> au tout début du html est également un noeud du dom. C'est dans l'arborescence du DOM juste avant <html>. Peu de gens le savent. Nous n'allons pas toucher ce nœud, nous ne le dessinons même pas sur les diagrammes pour cette raison, mais il est là.
L'objet **document** qui représente l'ensemble du document est également, formellement, un nœud dom.
Il existe 12 types de nœuds. En pratique, nous travaillons généralement avec 4 d'entre eux :

1. le document – le "point d'entrée" dans le dom.
2. les nœuds éléments – les balises HTML, les blocs de construction de l'arborescence.
3. les nœuds texte – contient du texte.
4. les commentaires – parfois, nous pouvons y mettre des informations, elles ne seront pas affichées, mais js peut les lire depuis le dom.

L'arbre DOM

Voyez par vous-même

Pour voir la structure dom en temps réel, essayez le live **DOM viewer**.

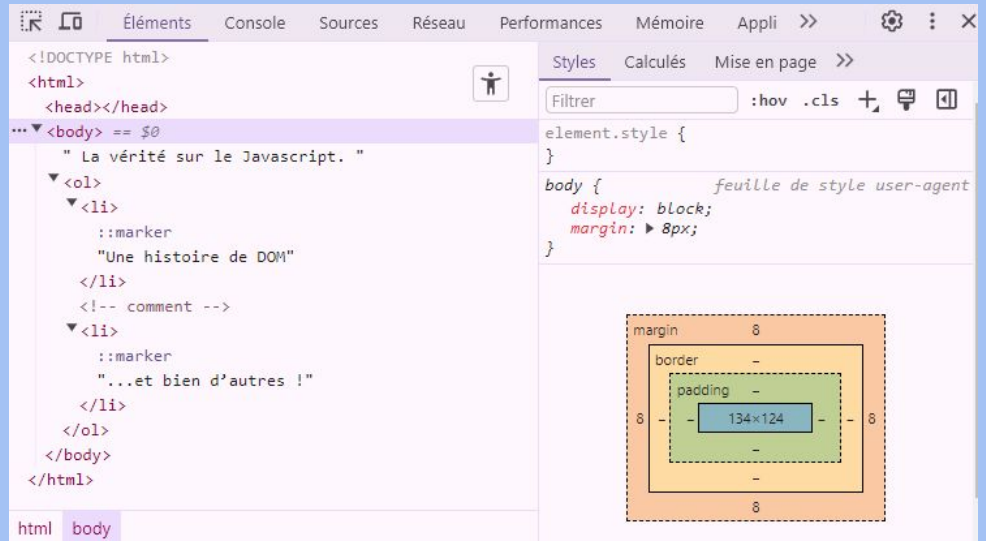
https://software.hixie.ch/utilities/js/live-dom-viewer/

Tapez simplement le document, et il apparaîtra comme un dom en un instant.

Une autre façon d'explorer le dom est d'utiliser les outils de développement du navigateur. en fait, c'est ce que nous utilisons lors du développement.

Pour ce faire, ouvrez les outils de développement du navigateur et passez à l'onglet éléments.

Cela devrait ressembler à ça :



L'arbre DOM

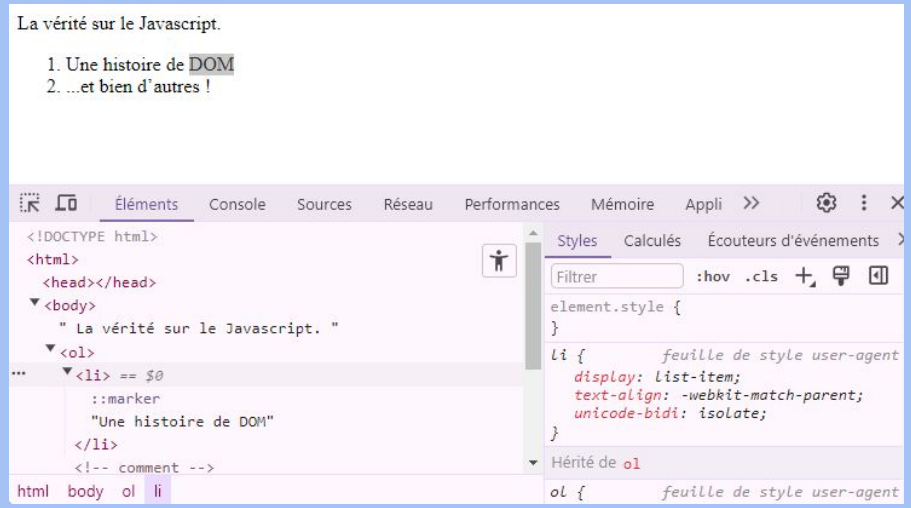
Vous pouvez voir le dom, cliquer sur les éléments, voir leurs détails et ainsi de suite.

Veuillez noter que **la structure du dom dans les outils de développement est simplifiée**. Les nœuds texte sont affichés comme du texte. Et il n'y a aucun nœud texte "vide" (espace uniquement). C'est très bien, car la plupart du temps nous nous intéressons aux nœuds éléments.

En cliquant sur le bouton dans le coin supérieur gauche cela nous permet de choisir un nœud à partir de la page Web à l'aide d'une souris (ou d'autres périphériques de pointeur) et de "l'inspecter" (faites défiler jusqu'à l'onglet Éléments). cela fonctionne très bien lorsque nous avons une énorme page html (et un énorme dom correspondant) et que nous aimerions voir la place d'un élément particulier.

Une autre façon de le faire serait simplement de cliquer avec le bouton droit sur une page Web et de sélectionner "inspecter" dans le menu contextuel.

L'arbre DOM



Dans la partie droite des outils se trouvent les sous-onglets suivants :

Styles – nous pouvons voir le CSS appliqué à l'élément en cours, règle par règle, y compris les règles intégrées (gris). Presque tout peut être modifié sur place, y compris les dimensions/margins/paddings de la boîte ci-dessous.

Calculés – pour voir le CSS appliqué à l'élément par propriété : pour chaque propriété, nous pouvons voir une règle qui la lui donne (y compris l'héritage CSS et autres).

Ecouteurs d'événements – pour voir les écouteurs d'événements attachés aux éléments du DOM (nous les couvrirons dans la prochaine partie du tutoriel).

... etc.

La meilleure façon de les étudier est de cliquer dessus. La plupart des valeurs sont modifiables sur place.

L'arbre DOM

Interaction avec la console
Comme nous travaillons le DOM, nous pouvons également vouloir lui appliquer du JavaScript.

Comme : obtenir un nœud et exécuter du code pour le modifier, pour voir le résultat.
Voici quelques conseils pour voyager entre l'onglet **Elements** et la console.

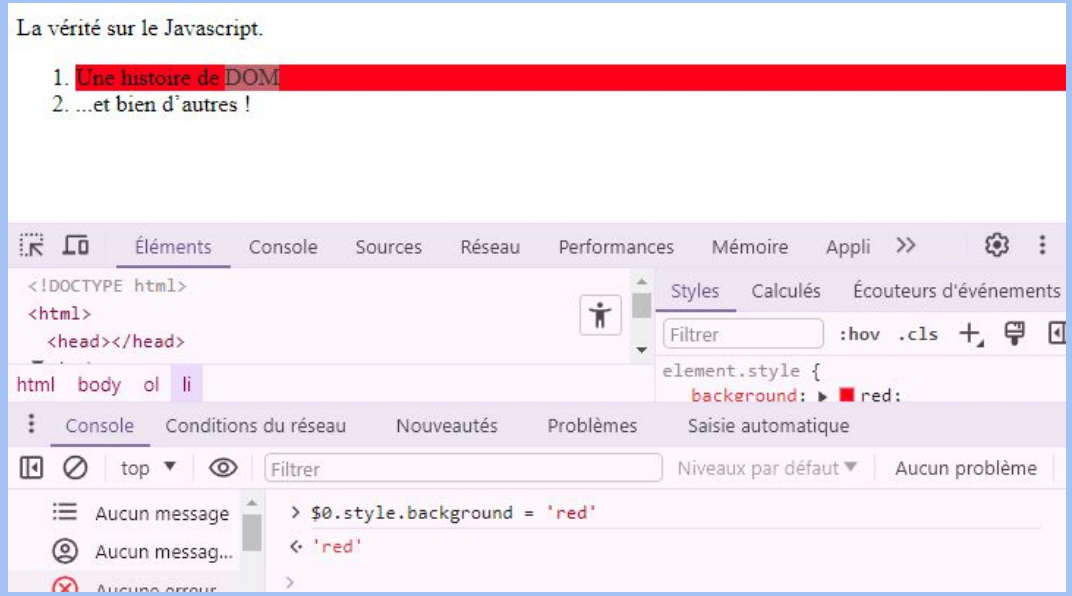
Pour commencer :

1. Sélectionnez le premier **<li>** dans l'onglet Éléments.
2. Appuyez sur la touche **Esc** – cela ouvrira la console juste en dessous de l'onglet Éléments.

Maintenant, le dernier élément sélectionné est disponible en tant que **$0**, le précédent sélectionné est **$1**, etc.

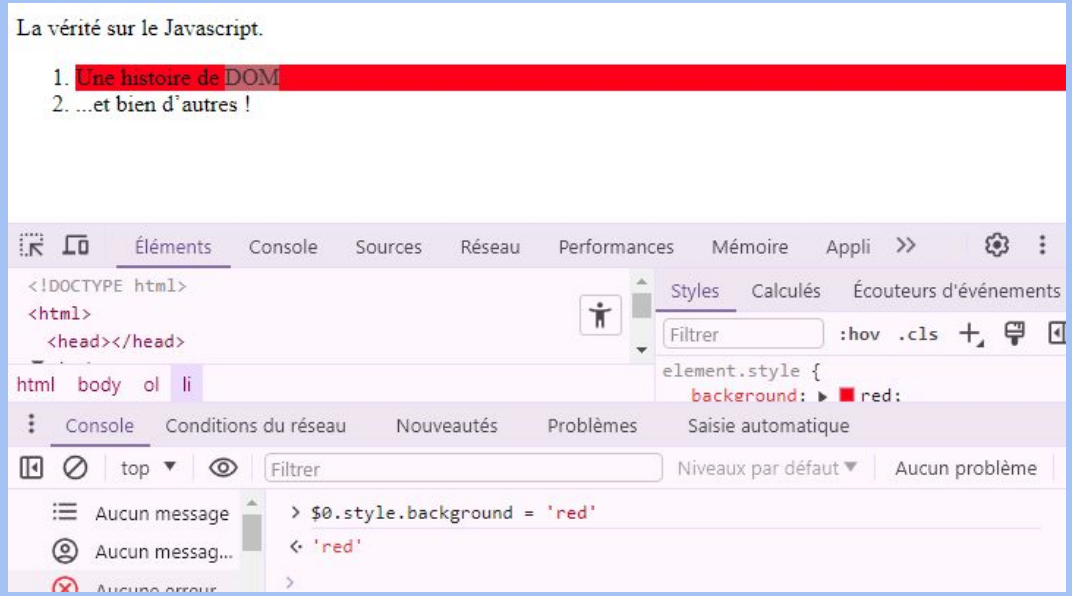
Nous pouvons exécuter des commandes sur eux.
Par exemple, **$0.style.background = 'red'** rend l'élément de la liste sélectionné rouge, comme ceci :

L'arbre DOM



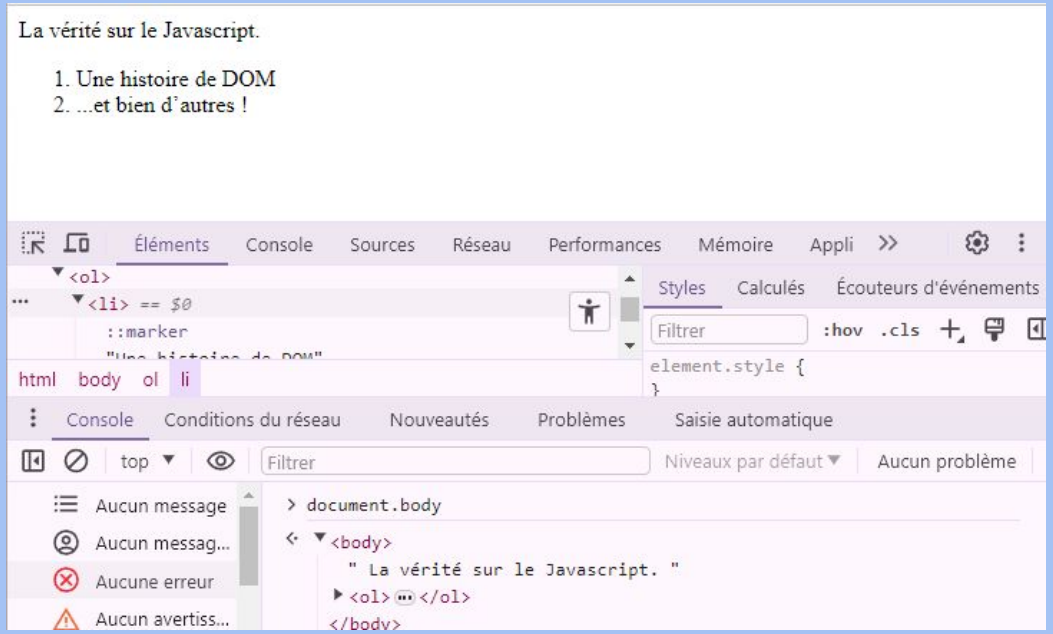
Voilà comment obtenir un nœud à partir d'Elements dans la console.
Il y a aussi un chemin de retour. S'il y a une variable référençant un nœud DOM, alors nous pouvons utiliser la commande inspect(node) dans la console pour la voir dans le volet Éléments.
Ou nous pouvons simplement sortir le nœud DOM dans la console et explorer "sur place", comme document.body ci-dessous :

L'arbre DOM



Voilà comment obtenir un nœud à partir d'Elements dans la console.
Il y a aussi un chemin de retour. S'il y a une variable référençant un nœud DOM, alors nous pouvons utiliser la commande inspect(node) dans la console pour la voir dans le volet Éléments.
Ou nous pouvons simplement sortir le nœud DOM dans la console et explorer "sur place", comme document.body ci-dessous :

L'arbre DOM



C'est à des fins de débogage bien sûr. À partir du chapitre suivant, nous accéderons et modifierons le DOM en utilisant JavaScript.

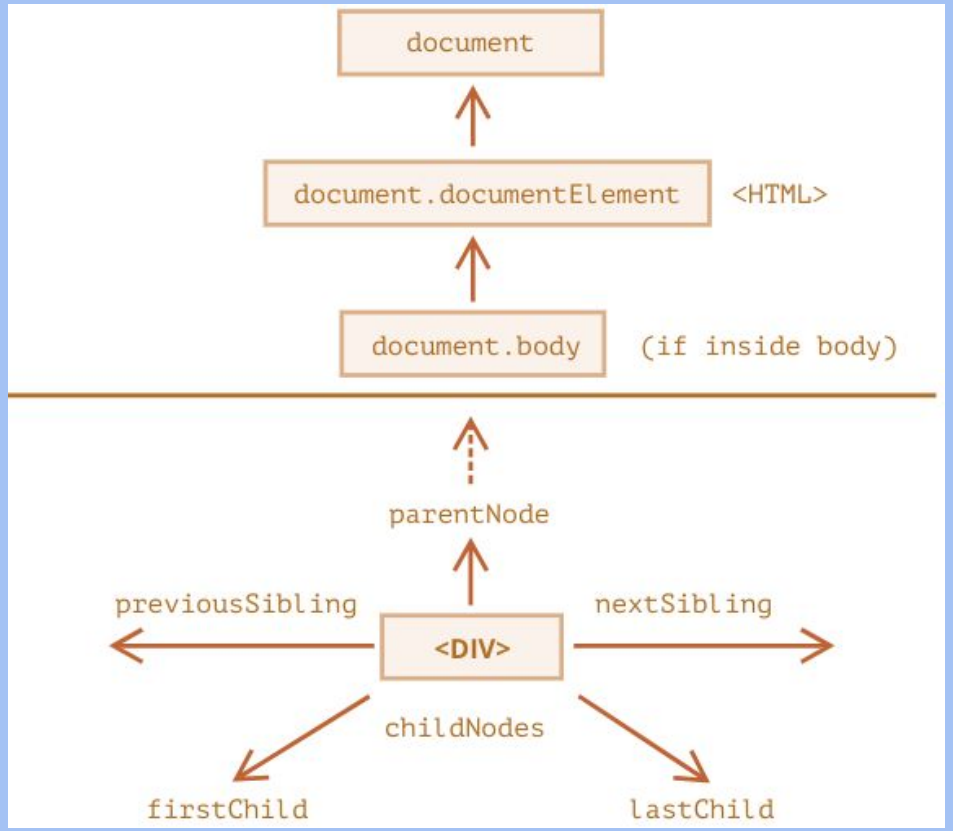
Les outils de développement du navigateur sont d'une grande aide au développement : nous pouvons explorer le DOM, essayer des choses et voir ce qui ne va pas.

JS

Parcourir le DOM

Parcourir le DOM

Le DOM nous permet de faire n'importe quoi avec les éléments et leur contenu, mais nous devons d'abord atteindre l'objet DOM correspondant.
Toutes les opérations sur le DOM commencent par l'objet document. C'est le "point d'entrée" principal du DOM. De là, nous pouvons accéder à n'importe quel nœud.
Ci contre une image des liens qui permettent de voyager entre les nœuds DOM.



En haut : **documentElement** et **body**
Les nœuds supérieurs de l'arbre sont disponibles directement en tant que propriétés de document :

<html> = document.documentElement
Le nœud de document le plus haut est **document.documentElement**. C'est le noeud DOM de la balise <html>.

Parcourir le DOM

<body> = document.body

Un autre nœud DOM largement utilisé est l'élément <body> – document.body.

<head> = document.head

La balise <head> est disponible en tant que document.head.

document.body peut être null

Un script ne peut pas accéder à un élément qui n'existe pas au moment de l'exécution.

En particulier, si un script se trouve dans <head>, alors document.body n'est pas disponible, car le navigateur ne l'a pas encore lu.

Ainsi, dans l'exemple ci-dessous, la première alert affiche null :

```
<html>
<head>
  <script>
    alert( "Dans HEAD: " + document.body ); // null, il n'y a pas encore de <body>
  </script>
</head>
<body>...
```

Parcourir le DOM

```
<html>
<head>
  <script>
    alert( "Dans HEAD: " + document.body ); // null, il n'y a pas encore de
<body>
  </script>
</head>
<body>
 <script>
    alert( "Dans BODY: " + document.body ); // HTMLBodyElement maintenant
existe
  </script>
</body>
</html>
```

Dans le monde du DOM, null signifie "n'existe pas "

Dans le DOM, la valeur null signifie "n'existe pas" ou "pas ce genre de nœud".

Parcourir le DOM

Enfants : childNodes, firstChild, lastChild

Nous utiliserons désormais deux termes :

- Noeuds enfants (ou enfants) – éléments qui sont des enfants directs. En d'autres termes, ils sont imbriqués dans celui donné. Par exemple, <head> et <body> sont des enfants de l'élément <html>.
- Descendants – tous les éléments imbriqués dans l'élément donné, y compris les enfants, leurs enfants, etc.
  Par exemple, ici <body> a des enfants et <ul> (et quelques nœuds texte vides) .

  ... Et les descendants de <body> ne sont pas seulement des enfants directs , <ul> mais aussi des éléments plus profondément imbriqués, tels que <li> (un enfant de <ul>) et <b> (un enfant de <li>) – le sous-arbre entier.

  La collection childNodes répertorie tous les nœuds enfants, y compris les nœuds texte.

```
<html>
<body>
  <div>Début</div>
  <ul>
    <li>
    <b>Information</b>
    </li>
  </ul>
</body>
</html>
```

Parcourir le DOM

L'exemple ci-dessous montre des enfants de document.body :

```
<html>
<body>
  <div>Début</div>
  <ul>
    <li>
      <b>Information</b>
    </li>
  </ul>
  <script>
    for (let i = 0; i < document.body.childNodes.length; i++) {
      alert( document.body.childNodes[i] ); // Text, DIV, Text, UL, ...,
SCRIPT
    }
  </script>
  </body>
</html>
```

Parcourir le DOM

```
<html>
<head>
  <script>
    alert( "Dans HEAD: " + document.body ); // null, il n'y a pas encore de <body>
  </script>
</head>
<body>
 <script>
    alert( "Dans BODY: " + document.body ); // HTMLBodyElement maintenant existe
  </script>
</body>
</html>
```

Dans le monde du DOM, null signifie "n'existe pas "

Dans le DOM, la valeur null signifie "n'existe pas" ou "pas ce genre de nœud".

Parcourir le DOM

<body> = document.body

Un autre nœud DOM largement utilisé est l'élément <body> – document.body.

<head> = document.head

La balise <head> est disponible en tant que document.head.

```
// change la couleur de fond en rouge
document.body.style.background = "red";

// réinitialisation après 1 seconde
setTimeout(() => document.body.style.background = "",
3000);
```

Ici, nous avons utilisé **style.background** pour changer la couleur d'arrière-plan de **document.body**, mais il existe de nombreuses autres propriétés, telles que :

- innerHTML – Contenu HTML du nœud.
- offsetWidth – la largeur du nœud (en pixels)
- ... etc.

Bientôt, nous apprendrons plus de façons de manipuler le DOM, mais nous devons d'abord connaître sa structure.

Parcourir le DOM

Les propriétés firstChild et lastChild donnent un accès rapide aux premier et dernier enfants.

Ce ne sont que des raccourcis. S'il existe des nœuds enfants, ce qui suit est toujours vrai :

```
elem.childNodes[0] === elem.firstChild
elem.childNodes[elem.childNodes.length - 1] === elem.lastChild
```

Il y a aussi une fonction spéciale elem.hasChildNodes() pour vérifier s'il y a des nœuds enfants.

Collections DOM

Comme nous pouvons le voir, childNodes ressemble à un tableau. Mais en réalité ce n'est pas un tableau, mais plutôt une * collection * – un objet itérable spécial semblable à un tableau.

Parcourir le DOM

Il y a deux conséquences importantes :

1. Nous pouvons utiliser **for..of** pour itérer dessus :

```
for (let node of document.body.childNodes) {
  alert(node); // Affiche tous les  noeuds de la collection
}
```

2. Les méthodes de tableau ne fonctionneront pas, car ce n'est pas un tableau :

```
alert(document.body.childNodes.filter); // undefined (Pas de méthode filter !)
```

La première chose est sympa. La seconde est tolérable, car nous pouvons utiliser Array.from pour créer un "vrai" tableau à partir de la collection, si nous voulons des méthodes de tableau :

```
alert( Array.from(document.body.childNodes).filter); // Function
```

Les collections DOM sont en lecture seule

Les collections DOM, et plus encore – toutes les propriétés de navigation répertoriées dans ce chapitre sont en lecture seule. Nous ne pouvons pas remplacer un enfant par autre chose en attribuant childNodes[i] = ... Changer le DOM nécessite d'autres méthodes. Nous les verrons dans le prochain chapitre.

Parcourir le DOM

Les collections DOM sont “live”

Presque toutes les collections DOM avec des exceptions mineures sont live. En d’autres termes, elles reflètent l’état actuel du DOM. Si nous gardons une référence à element.childNodes, et ajoutons/supprimons des nœuds dans le DOM, alors ils apparaissent automatiquement dans la collection

Frères, sœurs et parent

Les frères et sœurs sont des nœuds qui sont les enfants du même parent.

Par exemple, ici <head> et <body> sont des frères et sœurs :

- <body> est dit être le frère “suivant” ou “droit” de <head>,
- <head> est dit être le frère “précédent” ou “gauche” de <body>.

Le frère suivant est dans la propriété **nextSibling**, et le précédent – dans **previousSibling**.

Le parent est disponible en tant que parentNode.

```
<html>
<head>...</head><body>...</body>
</html>
```

Parcourir le DOM

Par exemple :

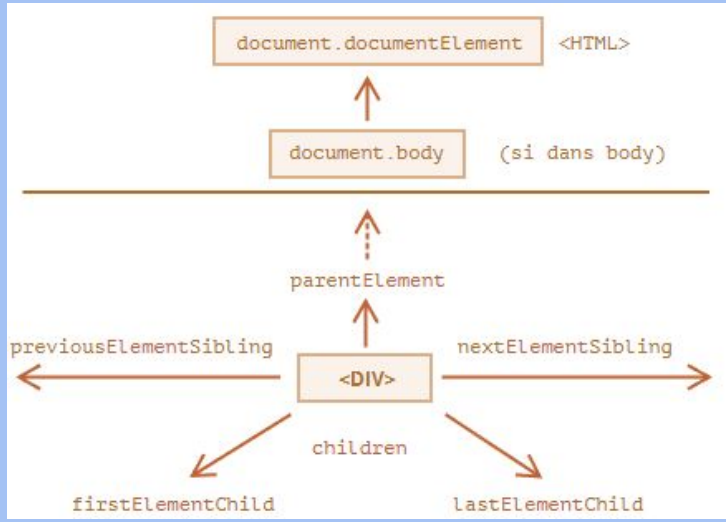
```
// le parent de <body> est <html>
alert( document.body.parentNode === document.documentElement ); // true
// après <head> vient <body>
alert( document.head.nextSibling ); // HTMLBodyElement
// avant <body> vient <head>
alert( document.body.previousSibling ); // HTMLHeadElement
```

Navigation par élément uniquement

Les propriétés de navigation répertoriées ci-dessus font référence à tous les nœuds. Par exemple, dans childNodes, nous pouvons voir à la fois les nœuds texte, les nœuds élément et même les nœuds commentaire s'il en existe.

Mais pour de nombreuses tâches, nous ne voulons pas de nœuds texte ou commentaire. Nous voulons manipuler des nœuds élément qui représentent des balises et forment la structure de la page.

Voyons donc plus de liens de navigation qui ne prennent en compte que les nœuds élément :



Parcourir le DOM

Les liens sont similaires à ceux donnés ci-dessus, juste avec le mot Element à l'intérieur :

- children – seuls les enfants qui sont des nœuds élément.
- firstElementChild, lastElementChild – enfants du premier et du dernier élément.
- previousElementSibling, nextElementSibling – éléments voisins.
- parentElement – élément parent.

Pourquoi parentElement ? Le parent peut-il ne pas être un élément ?

La propriété parentElement renvoie l'élément parent, tandis que parentNode retourne le parent "peu importe le nœud". Ces propriétés sont généralement les mêmes : elles obtiennent toutes deux le parent.

À la seule exception de document.documentElement

```
// le parent de <body> est <html>
alert( document.documentElement.parentNode ) document
alert( document.documentElement.parentElement ); // null
```

La raison en est que le nœud racine document.documentElement (<html>) a document comme parent. Mais document n'est pas un nœud élément, donc parentNode le renvoie et pas parentElement.

Parcourir le DOM

Ce détail peut être utile lorsque nous voulons passer d'un élément arbitraire elem à <html>, mais pas au document :

```
while(elem = elem.parentElement) {
// remonter jusqu'à <html>
  alert( elem );
}
```

Modifions l'un des exemples ci-dessus : remplaçons childNodes par children.

Maintenant, il ne montre que des éléments :

```
<html>
<body>
  <div>Begin</div>
  <ul>
    <li>Information</li>
  </ul>
  <div>End</div>
  <script>
    for (let elem of document.body.children) {
      alert(elem); // DIV, UL, DIV, SCRIPT
    }
  </script>
  ...
</body>
</html>
```

Parcourir le DOM

Les tableaux

Jusqu'à présent, nous avons décrit les propriétés de navigation de base.

Certains types d'éléments DOM peuvent fournir des propriétés supplémentaires, spécifiques à leur type, pour plus de commodité.

Les tableaux en sont un excellent exemple et représentent un cas particulièrement important :

L'élément <table> supporte (en plus de ce qui précède) ces propriétés :

- **table.rows** – la collection d'éléments <tr> du tableau.
- **table.caption/tHead/tFoot** – références aux éléments <caption>, <thead>, <tfoot>.
- **table.tBodies** – la collection d'éléments <tbody> (peut être multiple selon la norme, mais il y en aura toujours au moins une – même si elle n'est pas dans le HTML source, le navigateur la mettra dans le DOM).

<thead>, <tfoot>, <tbody> les éléments fournissent la propriété **rows** :

- tbody.rows – la collection de <tr> à l'intérieur.

<tr>

- tr.cells – la collection de cellules <td> et <th> à l'intérieur du <tr> donné.
- tr.sectionRowIndex – la position (index) du <tr> donné à l'intérieur du <thead>/<tbody>/<tfoot>.
- tr.rowIndex – le nombre de <tr> dans le tableau dans son ensemble (y compris toutes les lignes du tableau).

Parcourir le DOM

<td> et <th>

- **td.cellIndex** – le numéro de la cellule à l'intérieur du <tr> qui l'entoure.

Un exemple d'utilisation :

```
<table id="table1">
  <tr>
    <td>one</td><td>two</td>
  </tr>
  <tr>
    <td>three</td><td>four</td>
  </tr>
</table>
<script>
 // obtenir td avec "two" (première ligne, deuxième colonne)
 let td = table1.rows[0].cells[1];
 td.style.backgroundColor = "red"; // le mettre en valeur
</script>
```

JS

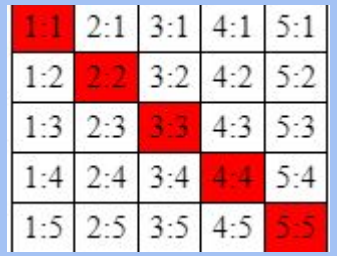
Exercices

Parcourir le DOM

Sélectionner toutes les cellules diagonales - 5

Écrivez le code pour colorer toutes les cellules du tableau diagonal en rouge.
Vous devrez obtenir toutes les diagonales <td> de la <table> et les colorer en utilisant le code :

```
// td doit être la référence à la cellule du tableau
td.style.backgroundColor = 'red';
```



Parcourir le DOM

Réponse

```
<!DOCTYPE HTML>
<html>
<head>
  <style>
    table {
      border-collapse: collapse;
    }
    td {
      border: 1px solid black;
      padding: 3px 5px;
    }
  </style>
</head>
<body>
  <table>
    <tr>
      <td>1:1</td>
      <td>2:1</td>
      <td>3:1</td>
      <td>4:1</td>
      <td>5:1</td>
    </tr>
    ...
    ...
    <tr> ... </tr>
    <tr> ... </tr>
    <tr> ... </tr>
    <tr>
      <td>1:5</td>
      <td>2:5</td>
      <td>3:5</td>
      <td>4:5</td>
      <td>5:5</td>
    </tr>
  </table>
  <script>
    let table = document.body.firstElementChild;
    for (let i = 0; i < table.rows.length; i++) {
      let row = table.rows[i];
      row.cells[i].style.backgroundColor = 'red';
    }
  </script>
</body>
</html>
```

Recherches: getElement*, querySelector*

Les outils de navigation du DOM sont très pratiques quand les éléments sont proches les uns des autres. Mais s'ils ne le sont pas ? Comment atteindre un élément arbitraire de la page ?

Il existe d'autres méthodes de recherche pour cela.

document.getElementById ou juste id

Si un élément a l'attribut id, on peut atteindre cet élément en utilisant la méthode document.getElementById(id), peu importe où elle se trouve.

Par exemple :

```
<div id="elem">
  <div id="elem-content">Elément</div>
</div>
<script>
  // récupération de l'élément :
  let elem = document.getElementById('elem');
  // on met son arrière-plan rouge :
  elem.style.background = 'red';
</script>
```

Recherches: getElement*, querySelector*

querySelectorAll
De loin, la méthode la plus polyvalente, elem.**querySelectorAll**(css) renvoie tous les éléments à l'intérieur de elem correspondant au sélecteur CSS donné en paramètre
Ici, on recherche tous les éléments <li> qui sont les derniers enfants :

Cette méthode est très puissante, car **tous les sélecteurs CSS peuvent être utilisés.**

```
<ul>
  <li>Le</li>
  <li>test</li>
</ul>
<ul>
  <li>a</li>
  <li>réussi</li>
</ul>
<script>
let elements=document.querySelectorAll('ul > li:last-child');
 for (let elem of elements) {
    alert(elem.innerHTML); // "test", "réussi"
}
</script>
```

On peut aussi utiliser des pseudo-classes
Les pseudo-classes dans le sélecteur CSS comme :hover et :active sont aussi acceptés. Par exemple, document.querySelectorAll(':hover') renverra l'ensemble des éléments dont le curseur est au-dessus en ce moment (dans l'ordre d'imbrication : du plus extérieur <html> au plus imbriqué).

Recherches: getElement*, querySelector*

querySelector

Un appel à elem.**querySelector**(css) renverra le premier élément d'un sélecteur CSS donné.
En d'autres termes, le résultat sera le même que elem.**querySelectorAll**(css)**[0]**, mais celui-ci cherchera tous les éléments et en choisira un seul, alors que elem.**querySelector** n'en cherchera qu'un.
C'est donc plus rapide, et plus court à écrire.

matches

Les méthodes précédentes recherchaient dans le DOM.
La commande elem.**matches**(css) ne recherche rien, elle vérifie simplement que elem correspond au sélecteur CSS donné.
Cette méthode devient utile quand on itère sur des éléments (comme dans un array par exemple) et qu'on veut filtrer ceux qui nous intéressent.

```
<a href="http://example.com/file.zip">...</a>
<a href="http://paris.fr">...</a>
<script>
for (let elem of document.body.children) {
 if (elem.matches('a[href$="zip"]')) {
  alert("le lien de l'archive : " + elem.href);
 }
}
</script>
```

closest

Les ancêtres d'un élément sont : le parent, le parent du parent, son propre parent etc... Les ancêtres forment une chaîne de parents depuis l'élément jusqu'au sommet.

Recherches: getElement*, querySelector*

La méthode elem.closest(css) cherche l'ancêtre le plus proche qui correspond au sélecteur CSS. L'élément elem est lui-même inclus dans la recherche.

En d'autres mots, la méthode closest part de l'élément et remonte en regardant chacun des parents. S'il correspond au sélecteur, la recherche s'arrête et l'ancêtre est renvoyé.

getElementsBy*
Il y a aussi d'autres méthodes pour rechercher des balises par tag, classe, etc...

Aujourd'hui, elles sont principalement de l'histoire ancienne, car querySelector est plus puissante et plus courte à écrire.

```
<h1>Contenu</h1>
<div class="contents">
  <ul class="book">
    <li class="chapter">Chapître 1</li>
    <li class="chapter">Chapître 2</li>
  </ul>
</div>
<script>
let chapter=document.querySelector('.chapter');
// LI
  alert(chapter.closest('.book')); // UL
  alert(chapter.closest('.contents')); // DIV
  alert(chapter.closest('h1')); // null (car h1
n'est pas un ancêtre)
</script>
```

Recherches: getElement*, querySelector*

Donc ici, on va surtout en parler dans le souci d'être complet, comme elles peuvent encore se retrouver dans des vieux scripts.

- elem.**getElementsByTagName**(tag) cherche les éléments avec le tag donné et renvoie l'ensemble de ces éléments. Le paramètre tag peut aussi être une étoile "*" pour signifier n'importe quel tag.
- elem.**getElementsByClassName**(className) renvoie les éléments qui ont la classe CSS donnée.
- document.**getElementsByName**(name) renvoie les éléments qui ont l'attribut name, dans tout le document. Très rarement utilisé.

```
// récupérer tous les divs du document.
let divs = document.getElementsByTagName('div');
```

Ensembles courants

Toutes les méthodes "**getElementsBy***" renvoient l'ensemble courant. De tels ensembles montrent toujours l'état courant du document et se mettent à jour automatiquement quand celui-ci change.

Dans l'exemple ci-dessous, il y a deux scripts :

- Le premier crée une référence à l'ensemble des . Maintenant, sa longueur est 1.
- Le second se lance après que le navigateur aie rencontré un autre , donc sa longueur est 2.

Recherches: getElement*, querySelector*

```
<div>Première DIV</div>
<script>
let divs=document.getElementsByTagName('div');
alert(divs.length ); // 1
</script>
<div>Deuxième DIV</div>
<script>
alert(divs.length ); // 2
</script>
```

A l'opposé, **querySelectorAll** renvoie un ensemble statique. C'est comme un tableau fixe d'éléments
Si on l'utilise, alors les deux scripts ci-dessus renvoient 1.

JS

Exercices

Parcourir le DOM

Recherche d'éléments

Voici le document avec le tableau et formulaire. Comment trouver ?...

- Le tableau avec id="age-table".
- Tous les éléments label dans ce tableau (il devrait y en avoir 3).
- Le premier td dans ce tableau (avec le mot "Age").
- Le form avec name="search".
- Le premier input dans ce formulaire.
- Le dernier input dans ce formulaire.

Ouvrez la page **table.html** dans un onglet à part et utilisez les outils du navigateur pour cela.

Parcourir le DOM

Recherche d'éléments

```
// 1. Le tableau avec `id="age-table"`.
let table = document.getElementById('age-table')

// 2. Tous les éléments 'label' dans le tableau
table.getElementsByTagName('label')
// ou
document.querySelectorAll('#age-table label')

// 3. Le premier td dans ce tableau (avec le mot "Age")
table.rows[0].cells[0]
// ou
table.getElementsByTagName('td')[0]
// ou
table.querySelector('td')
```

Parcourir le DOM

Recherche d'éléments

```
// 4. Le formulaire avec le nom "search"
// en supposant qu'il n'y ait qu'un élément avec name="search" dans le document.
let form = document.getElementsByName('search')[0]
// ou spécifiquement un formulaire
document.querySelector('form[name="search"]')

// 5. Le premier input dans ce formulaire
form.getElementsByTagName('input')[0]
// ou
form.querySelector('input')

// 6. Le dernier input dans ce formulaire
let inputs = form.querySelectorAll('input') // trouve tous les inputs
inputs[inputs.length-1] // prend le dernier
```

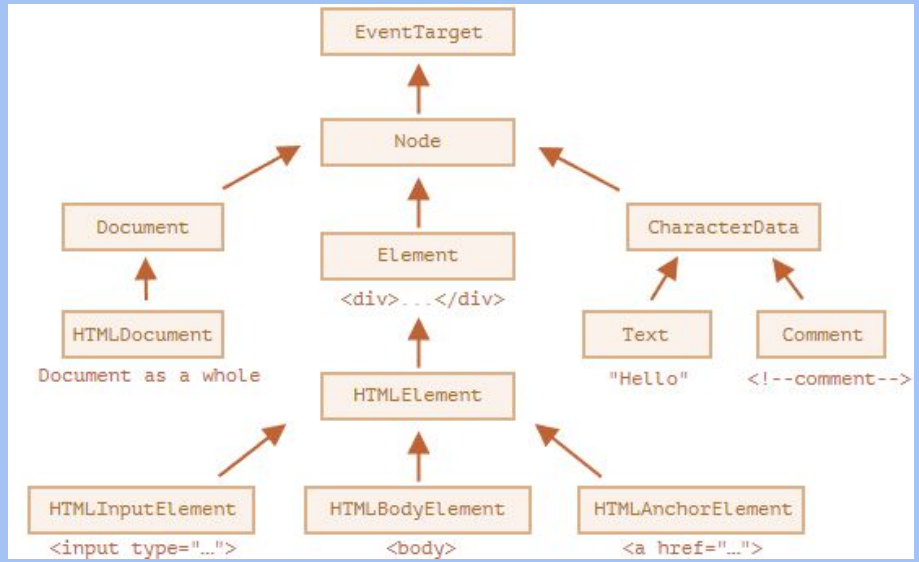
Propriétés de nœud : type, balise et contenu

Voyons plus en détail les nœuds DOM.

Dans ce chapitre, nous verrons plus en détail ce qu'ils sont et découvrirons leurs propriétés les plus utilisées.

Classes de nœud DOM

Différents nœuds DOM peuvent avoir des propriétés différentes. Par exemple, un nœud élément correspondant à la balise <a> a des propriétés liées aux liens, et celui correspondant à <input> a des propriétés liées aux entrées, etc. Les nœuds texte ne sont pas identiques aux nœuds élément. Mais il existe également des propriétés et des méthodes communes à chacun d'entre eux, car toutes les classes de nœuds DOM forment une hiérarchie unique. Chaque nœud DOM appartient à la classe intégrée correspondante.
La racine de la hiérarchie est EventTarget, hérité par Node, et d'autres nœuds DOM en héritent.



Propriétés de nœud : type, balise et contenu

Balise : nodeName et tagName

Étant donné un nœud DOM, nous pouvons lire son nom de balise dans les propriétés **nodeName** ou **tagName**

```
// récupérer tous les divs du document.
alert( document.body.nodeName ); // BODY
alert( document.body.tagName ); // BODY
```

Y a-t-il une différence entre **tagName** et **nodeName** ?

Bien sûr, la différence se reflète dans leurs noms, mais c'est en effet un peu subtile.

- La propriété **tagName** existe uniquement pour les nœuds **Element**.
- Le **nodeName** est défini pour tout **Node** :
  - pour les éléments, cela signifie la même chose que **tagName**.
  - pour les autres types de nœuds (texte, commentaire, etc.), il a une chaîne de caractères avec le type de nœud.

En d'autres termes, tagName est uniquement pris en charge par les nœuds élément (car il provient de la classe Element), tandis que nodeName peut dire quelque chose sur d'autres types de nœuds.

Propriétés de nœud : type, balise et contenu

Par exemple, comparons tagName et nodeName pour le document et un nœud de commentaire :

```
<body><!-- commentaire -->
  <script>
    // pour le commentaire
    alert( document.body.firstChild.tagName ); // undefined (pas un élément)
    alert( document.body.firstChild.nodeName ); // #comment
    // pour le document
    alert( document.tagName ); // undefined (pas un élément)
    alert( document.nodeName ); // #document
  </script>
</body>
```

Si nous ne traitons que des éléments, nous pouvons utiliser à la fois **tagName** et **nodeName** – il n'y a pas de différence.

Le nom de la balise est toujours en majuscule

*En mode HTML, tagName/nodeName est toujours en majuscule : c'est **BODY** pour <body> ou <BoDy>.*

Propriétés de nœud : type, balise et contenu

console.dir(elem) versus console.log(elem)

La plupart des navigateurs prennent en charge deux commandes dans leurs outils de développement : console.log et console.dir. Elles sortent leurs arguments dans la console. Pour les objets JavaScript, ces commandes font généralement la même chose. Mais pour les éléments DOM, elles sont différents :
console.log(elem) affiche l'arborescence DOM de l'élément.
console.dir(elem) affiche l'élément en tant qu'objet DOM, bon pour explorer ses propriétés.
Essayez les sur document.body.

innerHTML: les contenus

La propriété **innerHTML** permet d'obtenir le HTML à l'intérieur de l'élément sous forme de chaîne de caractères.
Nous pouvons également le modifier. C'est donc l'un des moyens les plus puissants de modifier la page.
L'exemple montre le contenu de document.body puis le remplace complètement :

```
<body>
  <p>Un paragraph</p>
  <div>Une div</div>
  <script>
    alert( document.body.innerHTML );
  // lit le contenu actuel
  document.body.innerHTML  = 'Le nouveau BODY!';
  // le remplace
  </script>
</body>
```

Propriétés de nœud : type, balise et contenu

Nous pouvons essayer d'insérer du code HTML invalide, le navigateur corrigera nos erreurs :

```
<body>
  <script>
  document.body.innerHTML  = '<strong>Test';
  alert( document.body.innerHTML );
  </script>
</body>
```

Les scripts ne s'exécutent pas

Si innerHTML insère une balise <script>dans le document – elle devient une partie du HTML, mais ne s'exécute pas.

outerHTML : HTML complet de l'élément

La propriété outerHTML contient le code HTML complet de l'élément. C'est comme innerHTML plus l'élément lui-même.

```
<div id="elem">Hello <b>World</b></div>
<script>
  alert( elem.outerHTML ); // <div id="elem">Hello <b>World</b></div>
</script>
```

Attention : contrairement à innerHTML, l'écriture dans outerHTML ne modifie pas l'élément. Au lieu de cela, il le remplace dans le DOM. Oui, cela semble étrange, et c'est étrange, c'est pourquoi nous en faisons une note séparée ici. Jetez-y un oeil.

Propriétés de nœud : type, balise et contenu

```
<div>Hello, World</div>
<script>
let div = document.querySelector('div');
  // remplace div.outerHTML avec <p>...</p>
  div.outerHTML = '<p>Un nouvel element</p>'; // (*)
  // 'div' est toujours la même !
  alert(div.outerHTML); // <div>Hello, world!</div> (**)
</script>
```

Ça a l'air vraiment bizarre, non ?

Dans la ligne (*) nous avons remplacé div par Un nouvel element . Dans le document externe (le DOM), nous pouvons voir le nouveau contenu au lieu de . Mais, comme nous pouvons le voir dans la ligne (**), la valeur de l'ancienne variable div n'a pas changé !

L'affectation **outerHTML** ne modifie pas l'élément DOM (l'objet référencé, dans ce cas, la variable *'div'*), mais le supprime du DOM et insère le nouveau HTML à sa place.

Donc, ce qui s'est passé dans *div.outerHTML=...* est :

- div a été supprimé du document.
- Un autre morceau du HTML Un nouvel element a été inséré à sa place.
- div a toujours son ancienne valeur. *Le nouveau HTML n'a été enregistré dans aucune variable.*

Propriétés de nœud : type, balise et contenu

nodeValue/data : contenu du nœud texte

La propriété **innerHTML** n'est valide que pour les nœuds élément.

D'autres types de nœuds, tels que les nœuds texte, ont leur contrepartie : les propriétés **nodeValue** et **data**. Ces deux sont presque les mêmes pour une utilisation pratique, il n'y a que des différences de spécifications mineures. Nous allons donc utiliser data, car il est plus court.

Un exemple de lecture du contenu d'un nœud texte et d'un commentaire :

```
<body>
  Hello
  <!-- Commentaire -->
  <script>
    let text = document.body.firstChild;
    alert(text.data); // Hello

    let comment = text.nextSibling;
    alert(comment.data); // Commentaire
  </script>
</body>
```

Propriétés de nœud : type, balise et contenu

Pour les nœuds texte, nous pouvons imaginer une raison de les lire ou de les modifier, mais pourquoi des commentaires ?
Parfois, les développeurs incorporent des informations ou des instructions de modèle dans HTML, comme ceci :

```
<!-- if isAdmin -->
<div>Bonjour, Admin!</div>
<!-- /if -->
```

...Ensuite, JavaScript peut le lire à partir de la propriété data et traiter les instructions intégrées.

textContent: texte pur

Le **textContent** donne accès au texte à l'intérieur de l'élément : seulement le texte, moins tous les <tags>.

```
<div id="news">
  <h1>Info !</h1><p>La planète brûle !</p>
</div>
<script>
  alert(news.textContent); // Info ! La planète brûle !
</script>
```

Propriétés de nœud : type, balise et contenu

Comme nous pouvons le voir, seul le texte est renvoyé, comme si tous les <tags> étaient supprimés, mais le texte qu'ils contenaient est resté.
En pratique, la lecture d'un tel texte est rarement nécessaire.
Ecrire dans **textContent** est beaucoup plus utile, car il permet d'écrire du texte de "manière sûre".
Disons que nous avons une chaîne de caractères arbitraire, par exemple entrée par un utilisateur, et que nous voulons l'afficher.
Avec **innerHTML** nous allons l'insérer "**au format HTML**", avec toutes les balises HTML.
Avec **textContent** nous allons l'insérer "**en tant que texte**", tous les symboles sont traités littéralement.
Comparez les deux :

```
<div id="elem1"></div>
<div id="elem2"></div>
<script>
  let name = prompt("Quel est ton nom ?", "<b>Mon nom est ... !</b>");
  elem1.innerHTML = name;
  elem2.textContent = name;
</script>
```

Propriétés de nœud : type, balise et contenu

- La première obtient le nom "en HTML" : toutes les balises deviennent des balises, nous voyons donc le nom en gras.
- La seconde obtient le nom "sous forme de texte", donc nous voyons littéralement Mon nom est ... !.

Dans la plupart des cas, nous attendons le texte d'un utilisateur et souhaitons le traiter comme du texte. Nous ne voulons pas de HTML inattendu sur notre site. Une affectation à textContent fait exactement cela.

La propriété "cachée"

L'attribut "hidden" (caché) et la propriété DOM spécifient si l'élément est visible ou non.
Nous pouvons l'utiliser dans le HTML ou l'attribuer en utilisant JavaScript, comme ceci :

```
<div>Les deux div ci-dessous ne seront pas affichées </div>
<div hidden>Div avec l'attribut "hidden"</div>
<div id="elem">Div avec l'id "elem"</div>
<script>
elem.hidden = true;
</script>
```

Techniquement, hidden fonctionne de la même manière que style="display:none". Mais c'est plus court à écrire.

Propriétés de nœud : type, balise et contenu

Voici un élément clignotant :

```
<div id="elem">Elément clignotant</div>
<script>
 setInterval(() => elem.hidden = !elem.hidden, 1000);
</script>
```

Plus de propriétés

Les éléments DOM ont également des propriétés supplémentaires, en particulier celles qui dépendent de la classe :

- **value** – la valeur pour <input>, <select> et <textarea> (HTMLInputElement, HTMLSelectElement...).
- **href** – le "href" pour <a href="..."> (HTMLAnchorElement).
- **id** – la valeur de l'attribut "id", pour tous les éléments (HTMLElement).
- ...et beaucoup plus...

```
<input type="text" id="elem" value="valeur">
<script>
  alert(elem.type); // "text"
  alert(elem.id); // "elem"
  alert(elem.value); // valeur
</script>
```

JS

Exercices

Propriétés de nœud : type, balise et contenu

Compter les descendants

Il y a un arbre structuré comme un ul/li imbriqué.

Écrivez le code qui pour que chaque <li> affiche :

- Quel est le texte à l'intérieur (sans le sous-arbre)
- Le nombre de <li> imbriqués – tous les descendants, y compris ceux profondément imbriqués.

Propriétés de nœud : type, balise et contenu

```
<body>
<ul>
 <li>...<li>
 ...
</ul>
</body>
<script>
for (let  li of document.querySelectorAll('li')) {
      // Récupère le texte du noeud -text node-
      let title = li.firstChild.data;

      title = title.trim(); // Supprime les espaces restant à la fin

      // Compte le nombre “d’enfants descendant”
      let count = li.getElementsByTagName('li').length;

      alert( title + ': ' + count );
    }
</script>
```