JS

Navigateur : Document, Évènements, Interfaces

2

Attributs et propriétés

Lorsque le navigateur charge la page, il "lit" (un autre mot: "analyse") le code HTML et en génère des objets DOM. Pour les nœuds éléments, la plupart des attributs HTML standards deviennent automatiquement des propriétés des objets DOM.

Par exemple, si la balise est <body id="page">, alors l'objet DOM a body.id="page".

Mais le mapping des propriétés d'attribut ne se fait pas un à un ! Dans ce chapitre, nous ferons attention à séparer ces deux notions, pour voir comment travailler avec elles, quand elles sont identiques et quand elles sont différentes.

Propriétés DOM

Nous avons déjà vu des propriétés DOM intégrées. Il y en a beaucoup. Mais techniquement, personne ne nous limite et s'il n'y en a pas assez, nous pouvons ajouter les nôtres.

Les nœuds DOM sont des objets JavaScript normaux.

Nous pouvons les modifier.

Par exemple, créons une nouvelle propriété dans document.body :

```
document.body.myData = {
  name: 'César', title: 'Empereur'
};
alert(document.body.myData.title); // Empereur
```

Attributs et propriétés

Nous pouvons également ajouter une méthode :

```
document.body.sayTagName = function() {
  alert(this.tagName);
};
document.body.sayTagName(); // BODY (la valeur de "this"
dans la méthode est document.body)
```

Nous pouvons également modifier des prototypes intégrés comme Element.prototype et ajouter de nouvelles méthodes à tous les éléments :

```
Element.prototype.sayHi = function() {
  alert(`Hello, je suis ${this.tagName}`);
};
document.documentElement.sayHi(); // Hello,  je suis HTML
document.body.sayHi(); // Hello,  je suis BODY
```

Ils peuvent avoir n'importe quelle valeur.

Ils sont sensibles à la casse (écrivez elem.nodeType, pas elem.NoDeTyPe).

Attributs HTML

En HTML, les balises peuvent avoir des attributs. Lorsque le navigateur analyse le code HTML pour créer des objets DOM pour les balises, il reconnaît les attributs * standard * et crée des propriétés DOM à partir d'elles.

Attributs et propriétés

Ainsi, lorsqu'un élément a id ou un autre attribut standard, la propriété correspondante est créée. Mais cela ne se produit pas si l'attribut n'est pas standard. Par exemple :

```
<body id="test" something="non-standard">
  <script>
    alert(document.body.id); // test
    // l'attribut non standard ne donne pas de propriété
    alert(document.body.something); // undefined
  </script>
</body>
```

Veuillez noter qu'un attribut standard pour un élément peut être inconnu pour un autre. Par exemple, "type" est standard pour <input> (HTMLInputElement), mais pas pour <body> (HTMLBodyElement). Les attributs standard sont décrits dans la spécification de la classe d'élément correspondante. Ici, nous pouvons le voir :

```
<body id="body" type="...">
  <input id="input" type="text">
  <script>
    alert(input.type); // text
    alert(body.type); // undefined: Propriété DOM non créée, car non standard
  </script>
```

Attributs et propriétés

Donc, si un attribut n'est pas standard, il n'y aura pas de propriété DOM pour lui. Existe-t-il un moyen d'accéder à ces attributs ?
Bien sûr, tous les attributs sont accessibles en utilisant les méthodes suivantes :

- elem.**hasAttribute**(name) – vérifie l'existence.
- elem.**getAttribute**(name) – obtient la valeur.
- elem.**setAttribute**(name, value) – définit la valeur.
- elem.**removeAttribute**(name) – supprime l'attribut.

Ces méthodes fonctionnent exactement avec ce qui est écrit en HTML.

On peut aussi lire tous les attributs en utilisant elem.attributes : une collection d'objets qui appartiennent à une classe intégrée Attr,avec name et la propriété value.

Voici une démonstration de la lecture d'une propriété non standard :

Attributs et propriétés

Les attributs HTML ont les fonctionnalités suivantes :

- Leur nom est insensible à la casse (id est identique à ID).
- Leurs valeurs sont toujours des chaînes de caractères.

```
<body something="non-standard">
  <script>
    alert(document.body.getAttribute('something')); // non-standard
  </script>
</body>
```

Voici une démonstration détaillée de l'utilisation des attributs :

```
<body>
  <div id="elem" about="Elephant"></div>
  <script>
    alert( elem.getAttribute('About') ); // (1) 'Elephant', lecture
    elem.setAttribute('Test', 123); // (2), écriture
    alert( elem.outerHTML ); // (3), voir si l'attribut est en HTML (oui)
    for (let attr of elem.attributes) { // (4) lister tout
      alert( `${attr.name} = ${attr.value}` );
    }
  </script>
</body>
```

Attributs et propriétés

Veuillez noter :

- getAttribute('About') – la première lettre est en majuscules ici, et en HTML tout est en minuscules. Mais cela n'a pas d'importance: les noms d'attribut ne sont pas sensibles à la casse.
- Nous pouvons assigner n'importe quoi à un attribut, mais il devient une chaîne. Nous avons donc ici "123" comme valeur.
- Tous les attributs, y compris ceux que nous définissons, sont visibles dans outerHTML.
- La collection attributes est itérable et possède tous les attributs de l'élément (standard et non standard) en tant qu'objets avec les propriétés name et value.

Synchronisation des attributs de propriété

Lorsqu'un attribut standard change, la propriété correspondante est automatiquement mise à jour et (à quelques exceptions près) vice versa.

Dans l'exemple ci-dessous, id est modifié en tant qu'attribut, et nous pouvons également voir la propriété modifiée. Et puis la même chose à l'envers :

Attributs et propriétés

```
<input>
<script>
  let input = document.querySelector('input');
  // attribute => property
  input.setAttribute('id', 'id');
  alert(input.id); // id (mis à jour)
  // property => attribute
  input.id = 'newId';
  alert(input.getAttribute('id')); // newId mis à jour
</script>
```

Mais il y a des exclusions, par exemple input.value se synchronise uniquement de l'attribut → vers la propriété, mais pas dans l'autre sens :

```
<input>
<script>
  let input = document.querySelector('input');
  // attribute => property
  input.setAttribute('value', 'texte');
  alert(input.value); // texte
  // NOT property => attribute
  input.value = 'newValue';
  alert(input.getAttribute('value')); // value non mis à jour
</script>
```

Attributs et propriétés

Dans l'exemple ci-dessus :

- La modification de l'attribut value met à jour la propriété.
- Mais le changement de propriété n'affecte pas l'attribut.

Cette "fonctionnalité" peut en fait être utile, car les actions de l'utilisateur peuvent entraîner des changements de value, puis après cela, si nous voulons récupérer la valeur "d'origine" du HTML, c'est dans l'attribut.

Les propriétés DOM sont typées

Les propriétés DOM ne sont pas toujours des chaînes de caractères. Par exemple, la propriété input.checked (pour les cases à cocher) est un booléen :

```
<input id="input" type="checkbox" checked> Cas à cocher
<script>
alert(input.getAttribute('checked')); // la valeur d'attribut est une chaîne de caractères vide
alert(input.checked); // la valeur de la propriété est : true
</script>
```

Attributs et propriétés

Il y a d'autres exemples. L'attribut style est une chaîne de caractères, mais la propriété style est un objet :

```
<div id="div" style="color:red;font-size:120%">Hello</div>
<script>
  // string
  alert(div.getAttribute('style')); // color:red;font-size:120%
  // object
  alert(div.style); // [object CSSStyleDeclaration]
  alert(div.style.color); // red
</script>
```

Attributs non standard, dataset

Lors de l'écriture HTML, nous utilisons beaucoup d'attributs standard. Mais qu'en est-il des modèles non standard et personnalisés ? Voyons d'abord s'ils sont utiles ou non? Pourquoi ?

Parfois, des attributs non standard sont utilisés pour transmettre des données personnalisées de HTML à JavaScript ou pour "marquer" des éléments HTML pour JavaScript.

Attributs et propriétés

```
<div show-info="name"></div>
<div show-info="age"></div>
<script>
  // le code trouve un élément avec la marque et montre ce qui est demandé
  let user = { name: "Pete", age: 25 };
  for(let div of document.querySelectorAll('[show-info]')) {
    // insert l'info dans le champ correspondant
    let field = div.getAttribute('show-info');
    div.innerHTML = user[field]; // d'abord Pete dans "name", puis 25 dans "age"
  }
</script>
```

Attributs et propriétés

Ils peuvent également être utilisés pour styliser un élément. Par exemple, ici pour la commande de l'état de l'attribut, order-state est utilisé.

Pourquoi l'utilisation d'un attribut serait-elle préférable à des classes comme .order-state-new, .order-state-pending, order-state-canceled ?

Parce qu'un attribut est plus pratique à gérer. L'état peut être changé aussi facilement que :

```
// un peu plus simple que de supprimer l'ancienne / ajouter une nouvelle classe
div.setAttribute('order-state', 'canceled');
```

```
<style>
  /* les styles reposent sur l'attribut
      personnalisé "order-state"        */
  .order[order-state="new"] {
    color: green;
  }
  .order[order-state="pending"] {
    color: blue;
  }
  .order[order-state="canceled"] {
    color: red;
  }
</style>
<div class="order" order-state="new">
  Nouvel ordre
</div>
<div class="order" order-state="pending">
  Ordre en suspend
</div>
<div class="order" order-state="canceled">
  Ordre annulé
</div>
```

Attributs et propriétés

Mais il peut y avoir un problème possible avec les attributs personnalisés. Que se passe-t-il si nous utilisons un attribut non standard selon nos besoins et que plus tard le standard l'introduit et lui fait faire quelque chose ? Le langage HTML est vivant, il grandit et de plus en plus d'attributs semblent répondre aux besoins des développeurs. Il peut y avoir des effets inattendus dans ce cas.

Pour éviter les conflits, il existe les attributs **data-*.**
Tous les attributs commençant par "data-" sont réservés à l'usage des programmeurs.
Ils sont disponibles dans la propriété dataset.

Par exemple, si un elem a un attribut nommé "data-about", il est disponible en tant que elem.dataset.about.

```
<body data-about="Elephants">
<script>
  alert(document.body.dataset.about); // Elephants
</script>
```

Attributs et propriétés

Les attributs de plusieurs mots comme *data-order-state* deviennent camel-cased : **dataset**.*orderState*.

Voici un exemple d'"order state" réécrit ->

L'utilisation des attributs data- * est un moyen valide et sûr de transmettre des données personnalisées.

Veuillez noter que nous pouvons non seulement lire, mais également modifier les attributs de données. Ensuite, CSS met à jour la vue en conséquence : dans l'exemple ci-dessus, la dernière ligne (*) change la couleur en bleu.

```
<!-- Début -->
<style>
  /* les styles reposent sur l'attribut
      personnalisé "order-state"        */
  .order[data-order-state="new"] {
    color: green;
  }
  .order[data-order-state="pending"] {
    color: blue;
  }
  .order[data-order-state="canceled"] {
    color: red;
  }
</style>
<div  id="order" class="order" data-order-state="new">
  Nouvel ordre
</div>
<script>
 // Lecture
 alert(order.dataset.orderState); // new
 // Modification
 order.dataset.orderState = "pending"; // (*)
</script>
```

JS

Exercices

Attributs et propriétés

Obtenez l'attribut - Exercice 1

Écrivez le code pour sélectionner l'élément avec l'attribut **data-widget-name** dans le document et pour lire sa valeur.

```
<!DOCTYPE html>
<html>
<body>
  <div data-widget-name="menu">Choisir le genre</div>
  <script>
    /* Votre code */
  </script>
</body>
</html>
```

Attributs et propriétés

Obtenez l'attribut - Réponse Exercice 1

```
<!DOCTYPE html>
<html>
<body>
  <div data-widget-name="menu">Choisir le genre</div>
  <script>
    // Récupérer l'élément
    let elem = document.querySelector('[data-widget-name]');

    // reading the value
    alert(elem.dataset.widgetName);
    // ou
    alert(elem.getAttribute('data-widget-name'));

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

Attributs et propriétés

Rendre les liens externes orange - Exercice 2

Mettez tous les liens externes en orange en modifiant leur propriété style.
Un lien est externe si : Son href contient :// mais ne commence pas par http://internal.com.

```
<h1>La liste</h1>
<ul>
  <li><a href="http://google.com">http://google.com</a></li>
  <li><a href="/tutorial">/tutorial.html</a></li>
  <li><a href="local/path">local/path</a></li>
  <li><a href="ftp://ftp.com/my.zip">ftp://ftp.com/my.zip</a></li>
  <li><a href="http://nodejs.org">http://nodejs.org</a></li>
  <li><a href="http://internal.com/test">http://internal.com/test</a></li>
</ul>
<script>
  // Style pour un lien
  let link = document.querySelector('a');
  link.style.color = 'orange';
</script>
```

Attributs et propriétés

Rendre les liens externes orange - Réponse Exercice 2

Tout d'abord, nous devons trouver toutes les références externes.Il y a deux façons.
La première consiste à trouver tous les liens à l'aide de document.querySelectorAll('a') puis à filtrer ce dont nous avons besoin :

```
let links = document.querySelectorAll('a');
for (let link of links) {
 let href = link.getAttribute('href');
 if (!href) continue; // pas d'attribut
 if (!href.includes('://')) continue; // pas de protocol
 if (href.startsWith('http://internal.com')) continue; // interne
  link.style.color = 'orange';
}
```

Veuillez noter: nous utilisons link.getAttribute('href'). Pas link.href, car nous avons besoin de la valeur HTML.

... Un autre moyen plus simple serait d'ajouter les contrôles au sélecteur CSS :

```
// recherchez tous les liens qui ont :// dans href
// mais href ne commence pas par http://internal.com
let selector = 'a[href*="://"]:not([href^="http://internal.com"])';
let links = document.querySelectorAll(selector);
links.forEach(link => link.style.color = 'orange');
```

Modification du document

La modification DOM est la clé pour créer des pages "live".
Ici, nous verrons comment créer de nouveaux éléments "à la volée" et modifier le contenu de la page existante.

Exemple : afficher un message

Démontrons en utilisant un exemple. Nous allons ajouter un message sur la page qui est plus joli que alert.
Voici à quoi cela ressemblera :

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>
<div class="alert">
  <strong>Attention !</strong> Vous avez lu un
message important.
</div>
```

Attention ! Vous avez lu un message important.

C'était un exemple HTML. Créons maintenant la même div avec JavaScript (en supposant que les styles sont déjà dans le HTML ou un fichier CSS externe).

Modification du document

Création d'un élément

Pour créer des nœuds DOM, il existe deux méthodes :

document.createElement(tag)

Crée un nouveau noeud élément avec la balise donnée :

```
let div = document.createElement('div');
```

document.createTextNode(text)

Crée un nouveau nœud texte avec le texte donné :

```
let textNode = document.createTextNode('Bonjour');
```

La plupart du temps, nous devons créer des nœuds d'élément, tels qu'une div pour le message.

Création du message

La création de la div message prend 3 étapes :

Modification du document

```
// 1. Créer la <div>
let div = document.createElement('div');

// 2. Ajouter la classe "alert"
div.className = "alert";

// 3. Ajouter du contenu
div.innerHTML = "<strong>Attention !</strong> Vous avez lu un message important.";
```

Nous avons créé l'élément. Mais pour le moment, ce n'est que dans une variable nommée div, pas encore dans la page. Nous ne pouvons donc pas le voir.

Méthodes d'insertion

Pour faire apparaître la div, nous devons l'insérer quelque part dans document.

Par exemple, dans l'élément <body>, référencé par document.body.

Il existe une méthode spéciale append pour cela : **document.body.append(div).**

Voici le code complet :

Modification du document

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>
<script>
let div = document.createElement('div');
div.className = "alert";
div.innerHTML = "<strong>Attention !</strong> Vous avez lu un message important.";
document.body.append(div);
</script>
```

Ici, nous avons appelé **append** sur document.body, mais nous pouvons appeler la méthode append sur n'importe quel autre élément, pour y mettre un autre élément. Par exemple, nous pouvons ajouter quelque chose à en appelant div.append(anotherElement).

Modification du document

Voici plus de méthodes d'insertion, elles spécifient différents endroits où insérer :

- node.**append**(...nodes or strings) – ajouter des nœuds ou des chaînes de caractères à la fin de node,
- node.**prepend**(...nodes or strings) – insérer des nœuds ou des chaînes de caractères au début de node,
- node.**before**(...nodes or strings) -- insérer des nœuds ou des chaînes de caractères avant node,
- node.**after**(...nodes or strings) -- insérer des nœuds ou des chaînes de caractères après node,
- node.**replaceWith**(...nodes or strings) -- remplace node avec les nœuds ou chaînes de caractères donnés.

Les arguments de ces méthodes sont une liste arbitraire de nœuds DOM à insérer ou des chaînes de texte (qui deviennent automatiquement des nœuds de texte).

Voici un exemple d'utilisation de ces méthodes pour ajouter des éléments à une liste et le texte avant/après :

Modification du document

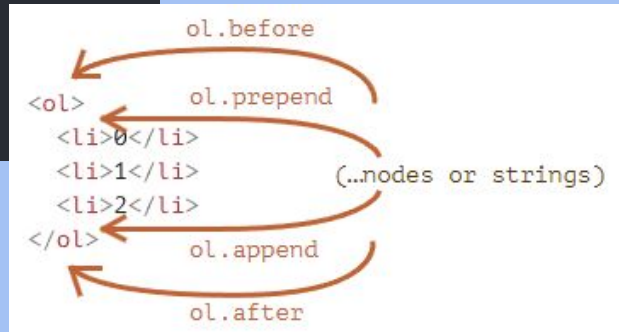
```
<ol id="ol">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>
<script>
  ol.before('before'); // insère la chaîne de caractères "before" avant <ol>
  ol.after('after'); // insère la chaîne de caractères "after" après <ol>

  let liFirst = document.createElement('li');
  liFirst.innerHTML = 'prepend';
  ol.prepend(liFirst); // insère liFirst au début de <ol>

  let liLast = document.createElement('li');
  liLast.innerHTML = 'append';
  ol.append(liLast); // insère liLast à la fin de <ol>
</script>
```

before

1. prepend
2. 0
3. 1
4. 2
5. append

after

Voici une image visuelle de ce que font les méthodes :



Modification du document

La liste finale sera donc :

```
before
<ol id="ol">
  <li>prepend</li>
  <li>0</li>
  <li>1</li>
  <li>2</li>
  <li>append</li>
</ol>
after
```

Comme indiqué, ces méthodes peuvent insérer plusieurs nœuds et morceaux de texte en un seul appel.

Par exemple, ici une chaîne de caractères et un élément sont insérés :

```
<div id="div"></div>
<script>
div.before('<p>Hello</p>', document.createElement('hr'));
</script>
```

Remarque: le texte est inséré “en tant que texte”, pas “en tant que HTML”, avec un échappement approprié des caractères tels que <, >. :

Modification du document

Le HTML final est donc :

```
&lt;p&gt;Hello&lt;/p&gt;
<hr>
<div id="div"></div>
```

En d'autres termes, les chaînes de caractères sont insérées de manière sûre, comme le fait elem.textContent. Ainsi, ces méthodes ne peuvent être utilisées que pour insérer des nœuds DOM ou des morceaux de texte. Mais que se passe-t-il si nous voulons insérer du HTML "en tant que html", avec toutes les balises et les trucs qui fonctionnent, comme elem.innerHTML le fait ?

insertAdjacentHTML/Text/Element

Pour cela, nous pouvons utiliser une autre méthode assez polyvalente : elem.insertAdjacentHTML(where, html). Le premier paramètre est un mot de code, spécifiant où insérer par rapport à elem. Doit être l'un des suivants :

- "**beforebegin**" – insère html immédiatement avant elem,
- "**afterbegin**" – insère html dans elem, au début,
- "**beforeend**" – insère html dans elem, à la fin,
- "**afterend**" – insère html immédiatement après elem.

Modification du document

Le second paramètre est une chaîne HTML insérée "au format HTML".

Par exemple :

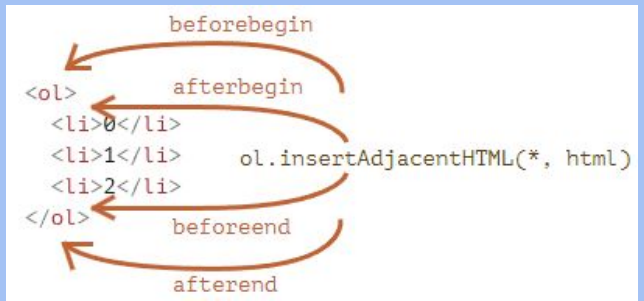
```
<div id="div"></div>
<script>
  div.insertAdjacentHTML('beforebegin', '<p>Hello</p>');
  div.insertAdjacentHTML('afterend', '<p>Au revoir</p>');
</script>
```

...Conduirait à :

```
<p>Hello</p>
<div id="div"></div>
<p>Au revoir</p>
```

Voilà comment nous pouvons ajouter du code HTML arbitraire à la page.

Voici une illustration des variantes d'insertion :



Modification du document

Nous pouvons facilement remarquer des similitudes entre cette image et l'image précédente. Les points d'insertion sont en fait les mêmes, mais cette méthode insère du HTML.

La méthode a deux sœurs :

- elem.**insertAdjacentText(where, text)** – la même syntaxe, mais une chaîne de caractères text est insérée en tant que texte au lieu de HTML,
- elem.**insertAdjacentElement(where, elem)** – la même syntaxe, mais insère un élément.

Elles existent principalement pour rendre la syntaxe "uniforme".

En pratique, seule **insertAdjacentHTML** est utilisée la plupart du temps. Parce que pour les éléments et le texte, nous avons des méthodes append/prepend/before/after – elles sont plus courtes à écrire et peuvent insérer des nœuds/morceaux de texte.

Modification du document

Voici donc une variante alternative pour afficher un message :

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>
<script>
document.body.insertAdjacentHTML("afterbegin",`<strong>Attention !</strong> Vous avez lu un message important.`);
</script>
```

Suppression de noeuds

Pour supprimer un nœud, il existe une méthode **node.remove()**.

Modification du document

Faisons disparaître notre message après une seconde :

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>
<script>
let div = document.createElement('div');
div.className = "alert";
div.innerHTML = "<strong>Attention !</strong> Vous avez lu un message important.";
document.body.append(div);
setTimeout(() => div.remove(), 1000);
</script>
```

Veuillez noter : si nous voulons déplacer un élément vers un autre endroit – il n'est pas nécessaire de le supprimer de l'ancien.

Modification du document

Toutes les méthodes d'insertion suppriment automatiquement le nœud de l'ancien emplacement.

Par exemple, permutons les éléments :

```
<div id="first">Premier/div>
<div id="second">Second</div>
<script>
  // pas besoin d'appeler remove
  second.after(first); // prend #second et après insère #first
</script>
```

Clonage de Noeuds : cloneNode

Comment insérer un autre message similaire ?

Nous pourrions créer une fonction et y mettre le code. Mais l'alternative serait de cloner la div existant et de modifier le texte à l'intérieur (si nécessaire).

Parfois, lorsque nous avons un gros élément, cela peut être plus rapide et plus simple.

Modification du document

L'appel **elem.cloneNode(*true*)** crée un clone "*profond*" de l'élément – avec tous les attributs et sous-éléments.
Si nous appelons **elem.cloneNode(*false*)**, alors le clone est fait sans éléments enfants.

Un exemple de copie du message :

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>
<div class="alert" id="div">
  <strong>Attention !</strong> Vous avez lu un message important.
</div>
<script>
let div2 = div.cloneNode(true); // clone le message
  div2.querySelector('strong').innerHTML = 'Au revoir !';
  // change le clone
  div.after(div2); // affiche le clone après le div existant
</script>
```

Modification du document

Méthodes d'insertion/suppression à l'ancienne

Il existe également des méthodes de manipulation du DOM "à l'ancienne", qui existent pour des raisons historiques.

Ces méthodes viennent d'une époque très ancienne. De nos jours, il n'y a aucune raison de les utiliser, depuis qu'il existe des méthodes modernes, telles que append, prepend, before, after, remove, replaceWith, qui sont plus flexibles.

La seule raison pour laquelle nous listons ces méthodes ici est que vous pouvez les trouver dans de nombreux anciens scripts :

- **parentElem.appendChild(node)** : Appends node as the last child of parentElem.
- **parentElem.insertBefore(node, nextSibling)** : Insère node avant nextSibling dans parentElem.
- **parentElem.replaceChild(node, oldChild)** : Remplace oldChild avec node chez les enfants de parentElem.
- **parentElem.removeChild(node)** : Supprime node de parentElem (en supposant que node est son enfant).

Toutes ces méthodes renvoient le nœud inséré/supprimé. En d'autres termes, parentElem.appendChild(node) renvoie node. Mais généralement, la valeur retournée n'est pas utilisée, nous exécutons simplement la méthode.

Modification du document

Un mot sur “document.write”

Il existe une autre méthode très ancienne pour ajouter quelque chose à une page Web : document.write.

La syntaxe :

```
<p>Quelque part dans la page...</p>
<script>
  document.write('<b>Bonjour JS</b>');
</script>
<p>Fin</p>
```

L’appel à document.write(html) écrit le html dans la page “ici et maintenant”. La chaîne de caractères html peut être générée dynamiquement, donc c’est un peu flexible. Nous pouvons utiliser JavaScript pour créer une page Web à part entière et l’écrire.

La méthode vient de l’époque où il n’y avait pas de DOM, pas de standards … Des temps vraiment anciens. Il vit toujours, car il existe des scripts qui l’utilisent.

Dans les scripts modernes, nous le voyons rarement, en raison de la limitation importante suivante :

L’appel à document.write ne fonctionne que pendant le chargement de la page.

Si nous l’appelons ensuite, le contenu du document existant est effacé.

Modification du document

```
<p>Après une seconde le contenu de cette page sera remplacé...</p>
<script>
  // document.write après 1 seconde
  // cela après que la page soit chargée, donc il efface le contenu existant
  setTimeout(() => document.write('<b>...Par cela.</b>'), 1000);
</script>
```

C'est donc un peu inutilisable au stade "post chargement", contrairement aux autres méthodes DOM que nous avons couvertes ci-dessus.

Voilà l'inconvénient.

Il y a aussi un avantage. Techniquement, lorsque document.write est appelé pendant que le navigateur lit ("analyse") le HTML entrant, et qu'il écrit quelque chose, le navigateur le consomme comme s'il était initialement là, dans le texte HTML.

Cela fonctionne donc très rapidement, car il n'y a aucune modification du DOM impliquée. Il écrit directement dans le texte de la page, tandis que le DOM n'est pas encore construit.

Modification du document

Donc, si nous devons ajouter beaucoup de texte en HTML de manière dynamique, et que nous sommes en phase de chargement de page, et que la vitesse compte, cela peut aider. Mais dans la pratique, ces exigences se rencontrent rarement. Et généralement, nous pouvons voir cette méthode dans les scripts simplement parce qu'ils sont anciens.

JS

Exercices

Modification du document

Effacer l'élément - Exercice 1

Créez une fonction clear(elem) qui supprime tout de l'élément "elem".

```
<ol id="elem">
  <li>Hello</li>
  <li>World</li>
</ol>

<script>
  function clear(elem) {
     /* votre code */
  }

  clear(elem); // efface la liste
</script>
```

Modification du document

Effacer l'élément - Réponse Exercice 1

Une première façon de faire.....de même :

```
<ol id="elem">
  <li>Hello</li>
  <li>World</li>
</ol>

<script>
  function clear(elem) {
   while (elem.firstChild) {
     elem.firstChild.remove();
   }
  }

  clear(elem); // efface la liste
</script>
```

Et il existe également un moyen plus simple de faire de même :

```
<ol id="elem">
  <li>Hello</li>
  <li>World</li>
</ol>

<script>
  function clear(elem) {
   while (elem.firstChild) {
     elem.innerHTML = '';
   }
  }

  clear(elem); // efface la liste
</script>
```

Modification du document

Créer une liste - Exercice 2

Écrivez une interface pour créer une liste à partir des entrées utilisateur.
Pour chaque élément de la liste :

- Interrogez un utilisateur sur son contenu en utilisant prompt.
- Créez le <li> avec et ajoutez-le à <ul>.
- Continuez jusqu'à ce que l'utilisateur annule l'entrée (en appuyant sur la touche Esc ou une entrée vide).

Tous les éléments doivent être créés dynamiquement.

Si un utilisateur tape des balises HTML, elles doivent être traitées comme un texte.

Modification du document

Créer une liste - Réponse Exercice 2

```
<!DOCTYPE HTML>
<html>
<body>
  <h1>Créer une liste</h1>
  <script>
    let ul = document.createElement('ul');
    document.body.append(ul);
    while (true) {
      let data = prompt("Ajouter un élément de liste texte", "");
      if (!data) {
        break;
      }
      let li = document.createElement('li');
      li.textContent = data;
      ul.append(li);
    }
  </script>
</body>
</html>
```

Styles et classes

Avant d'entrer dans les méthodes que JavaScript utilise pour traiter les styles et classes, voici une règle importante. Elle devrait être assez évidente, mais nous devons encore la mentionner.

Il y a, en général, deux façons de styliser un élément:

1. Créer une classe dans CSS et l'ajouter:
2. Écrire les propriétés directement dans style: .

JavaScript peut modifier les classes et les propriétés de style.

Nous devons toujours favoriser l'utilisation des classes CSS plutôt que style. Ce dernier devrait seulement être utilisé si les classes sont incapables d'effectuer la tâche requise.

Par exemple, style est acceptable si nous calculons les coordonnées d'un élément dynamiquement et souhaitons les définir à partir de JavaScript, comme ceci:

Styles et classes

```
let top = /* calculs complexes */;
let left = /* calculs complexes */;

elem.style.left = left; // par ex. '123px', calculé lors de l'exécution
elem.style.top = top; // par ex. '456px'
```

Pour les autres cas, comme rendre le texte rouge, ajouter une icône d'arrière-plan – décrivez cela dans CSS et ensuite ajoutez la classe (JavaScript peut effectuer ceci). C'est plus flexible et plus facile à gérer.

className et classList

Changer une classe est l'une des actions les plus utilisées dans les scripts.

Autrefois, il existait une limitation dans JavaScript: un mot réservé comme "class" ne pouvait pas être une propriété d'un object.

Cette limitation n'existe plus maintenant, mais à l'époque, il était impossible d'avoir une propriété de "class", comme elem.class.

Styles et classes

```
<body class="page d'accueil">
  <script>
    alert(document.body.className); // page d'accueil
  </script>
</body>
```

Si nous attribuons quelque chose à elem.**className**, elle remplace la chaîne entière de classes. Parfois c'est ce que nous avons besoin, mais souvent, nous voulons seulement ajouter ou enlever une classe.

Il y a une autre propriété pour ce besoin: elem.**classList**.

elem.**classList** est un objet spécial avec des méthodes pour **add/remove/toggle** une seule classe.

Prenons, par exemple:

```
<body class="page d'accueil">
  <script>
    // ajouter une classe
    document.body.classList.add( 'article' );
    alert(document.body.className); // page d'accueil
  </script>
</body>
```

Styles et classes

Alors nous pouvons opérer avec la chaîne de toutes les classes en utilisant **className** ou avec les classes individuelles en utilisant **classList.** Ce que nous choisissons dépend de nos besoins.

Méthodes de classList:

- elem.**classList.add/remove**("class") – ajoute ou enlève la classe.
- elem.**classList.toggle**("class") – ajoute la classe si elle n'existe pas, sinon enlève-la.
- elem.**classList.contains**("class") – vérifie pour la classe donnée, renvoie true/false.

En outre, **classList** est itérable, alors nous pouvons lister toutes les classes avec for..of, comme ceci:

```
<body class="page d'accueil">
  <script>
    for ( let name of document.body.classList ) {
      alert(name); // page d'accueil, ensuite page
    }
  </script>
</body>
```

Styles et classes

Style de l'élément

La propriété elem.style est un objet qui correspond à ce qui est écrit dans l'attribut "style". Attribuant elem.style.width="100px" fonctionne de la même façon qu'un attribut style ayant une chaîne width:100px.

Pour une propriété ayant plusieurs mots, camelCase est utilisé:

```
background-color  => elem.style.backgroundColor
z-index           => elem.style.zIndex
border-left-width => elem.style.borderLeftWidth
```

Prenons, par exemple:

```
document.body.style.backgroundColor = prompt('background color?', 'green');
```

Réinitialiser la propriété de style

Parfois nous voulons attribuer une propriété de style, et ensuite la retirer.

Par exemple, pour cacher un élément, nous pouvons définir elem.style.display = "none".

Plus tard, nous voulons peut-être enlever style.display comme si cette propriété n'était définie. Au lieu de delete elem.style.display, nous devons attribuer une chaîne vide à la propriété de style: elem.style.display = "".

Styles et classes

```
// si nous exécutons cette code, <body> clignotera
document.body.style.display = "none"; // cacher
setTimeout(() => document.body.style.display = "", 1000); // retour à la normale
```

Si nous attribuons style.display à une chaîne vide, le navigateur applique les classes CSS et ses styles intégrés normalement, comme s'il n'y avait pas de propriété style.display.

Il existe également une méthode spéciale pour cela, elem.style.removeProperty('style property'). Ainsi, nous pouvons supprimer une propriété comme celle-ci :

```
document.body.style.background = 'red'; //configure le background à rouge
setTimeout(() => document.body.style.removeProperty('background'), 1000);
// supprime l'arrière-plan après 1 seconde
```

Réécriture complète avec style.cssText

Normalement, nous utilisons style.* pour attribuer des propriétés de style individuelles. Nous ne pouvons pas attribuer le style complet comme div.style="color: red; width: 100px", parce que div.style est un object, et il est en lecture seulement.

Styles et classes

Pour définir un style complet comme une chaîne, il y a une propriété spéciale style.cssText:

```
<div id="div">Bouton</div>
<script>
  // nous pouvons attribuer des drapeaux de style spéciaux comme "important" ici
  div.style.cssText=`color: red !important;
    background-color: yellow;
    width: 100px;
    text-align: center;
  `;
  alert(div.style.cssText);
</script>
```

Cette propriété est rarement utilisée parce qu'une telle affectation enlève tous les styles pré-existants: au lieu d'être ajoutée, elle les remplace. Peut occasionnellement effacer quelque chose de nécessaire. Par contre, nous pouvons l'utiliser sans risque pour des nouveaux éléments – nous savons que nous n'effaceront pas un style pré-existant.

La même chose peut être accomplie en définissant un attribut: div.**setAttribute**('style', 'color: red...').

Styles et classes

Faites attention aux unités

N'oubliez pas d'ajouter des unités de CSS aux valeurs.

Par exemple, nous ne devrions pas attribuer elem.style.top à 10, mais plutôt à 10px. Sinon ça ne fonctionnera pas:

```
<body>
  <script>
    // ne fonctionne pas!
    document.body.style.margin = 20;
    alert( document.body.style.margin ); // '' (chaîne vide, l'affectation est ignorée)
    // maintenant ajoutez l'unité de CSS (px) - et ça fonctionne
    document.body.style.margin = '20px';
    alert( document.body.style.margin ); // 20px
    alert( document.body.style.marginTop ); // 20px
    alert( document.body.style.marginLeft ); // 20px
  </script>
</body>
```

Il est à noter: le navigateur "décortique" la propriété style.margin dans les dernières lignes et déduit style.marginLeft et style.marginTop à partir de ceci.

Styles et classes

Styles calculés: getComputedStyle

Alors, modifier un style est facile. Mais comment pouvons-nous le lire?

Par exemple, nous voulons savoir la taille, les marges et la couleur d'un élément. Comment faire?

La propriété style opère seulement sur la valeur de l'attribut "style", sans aucune cascade CSS.

Alors nous ne pouvons rien lire des classes CSS en utilisant elem.style.

Par exemple, ici, style ne reconnaît pas la marge:

```
<head>
  <style> body { color: red; margin: 5px } </style>
</head>
<body>
  Le texte rouge
  <script>
    alert( document.body.style.color ); // vide
    alert( document.body.style.marginTop ); // vide
  </script>
</body>
```

...Mais, si nous voulons, par exemple, augmenter la marge par 20px? Nous en voudrions la valeur actuelle. Il y a une autre méthode pour cela: **getComputedStyle**.

La syntaxe est:

```
getComputedStyle(element, [pseudo])
```

Styles et classes

element

Élément pour lire la valeur de.

pseudo

Un pseudo-élément si nécessaire, par exemple ::before. Une chaîne vide ou aucun argument signifie l'élément lui-même.

Le résultat est un objet avec des styles, comme elem.style, mais maintenant par rapport à toutes les classes CSS.

Prenons, par exemple:

```
<head>
  <style> body { color: red; margin: 5px } </style>
</head>
<body>
  <script>
    let computedStyle = getComputedStyle( document.body );
    // maintenant nous pouvons en lire la marge et la couleur
    alert( computedStyle.marginTop ); // 5px
    alert( computedStyle.color ); // rgb(255, 0, 0)
  </script>
</body>
```

Taille des éléments et défilement

Il existe de nombreuses propriétés JavaScript qui nous permettent de lire des informations sur la largeur, la hauteur des éléments et d'autres caractéristiques géométriques.
Nous en avons souvent besoin lors du déplacement ou du positionnement d'éléments en JavaScript.

Exemple d'élément

Comme exemple d'élément pour démontrer les propriétés, nous utiliserons celui donné ci-dessous :

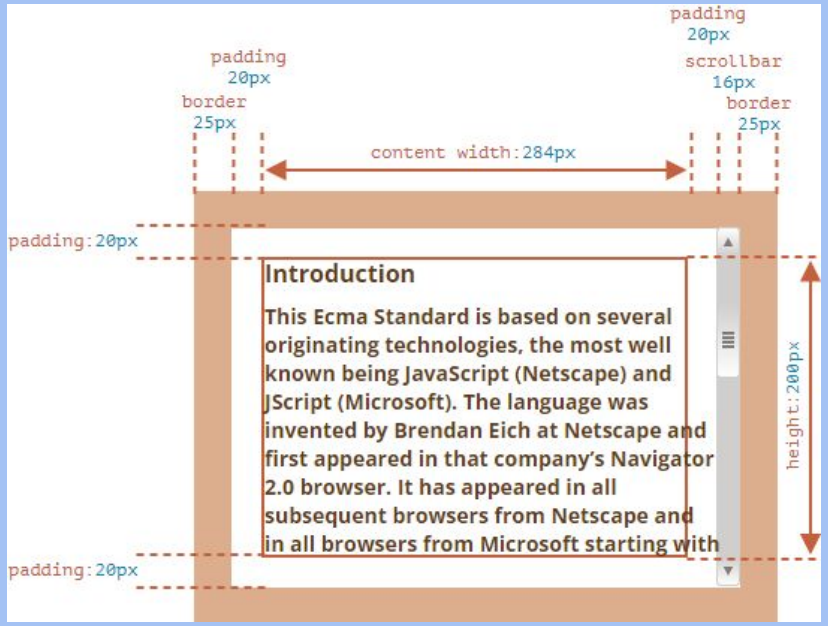
```
<div id="exemple"> Texte </div>
<style>
#exemple{
  width: 300px;
  height: 200px;
  border: 25px solid #E8C48F;
  padding: 20px;
  overflow: auto;
}
</style>
</head>
```

Taille des éléments et défilement

Il a la bordure, le padding et la barre de défilement.

Il n'y a pas de marges, car elles ne font pas partie de l'élément lui-même et il n'y a pas de propriétés spéciales pour elles.

L'élément ressemble à ceci :



Taille des éléments et défilement

Attention à la barre de défilement

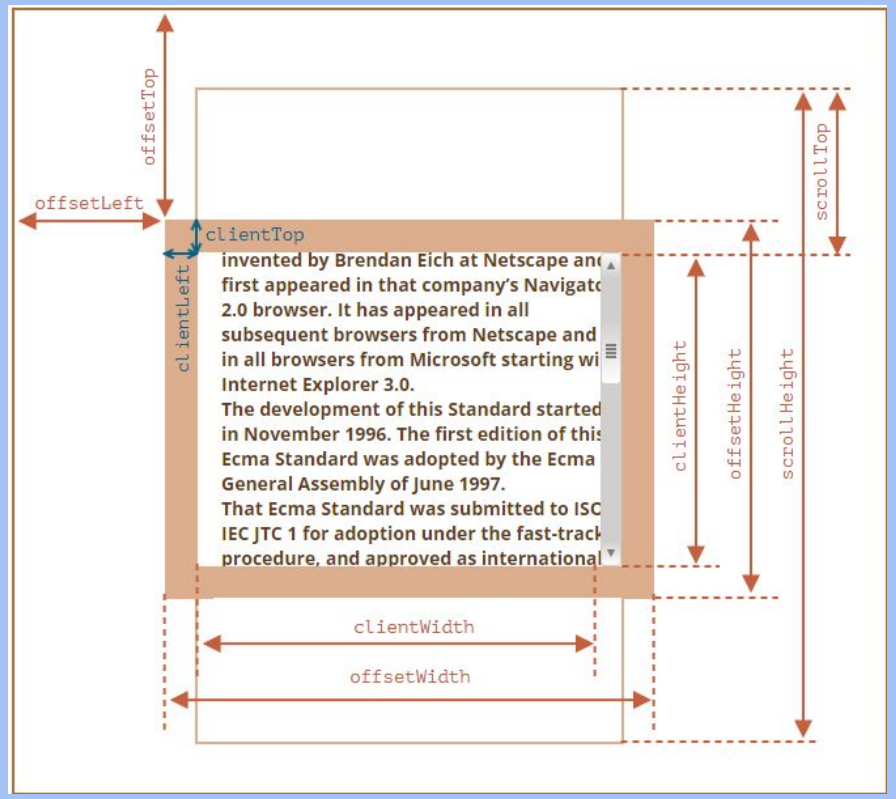
L'image ci-dessus illustre le cas le plus complexe lorsque l'élément a une barre de défilement. Certains navigateurs (pas tous) lui réservent de l'espace en le prenant dans le contenu (étiqueté comme "largeur de contenu" ci-dessus).

Ainsi, sans barre de défilement, la largeur du contenu serait de 300px, mais si la barre de défilement est large de 16 pixels (la largeur peut varier entre les appareils et les navigateurs), il ne reste que 300 - 16 = 284 pixels, et nous devons en tenir compte. . C'est pourquoi les exemples de ce chapitre supposent qu'il y a une barre de défilement. Sans cela, certains calculs sont plus simples.

Taille des éléments et défilement

Géométrie

Voici l'image globale avec les propriétés de la géométrie :



Les valeurs de ces propriétés sont techniquement des nombres, mais ces nombres sont "de pixels", donc ce sont des mesures de pixels.

Commençons à explorer les propriétés à partir de l'extérieur de l'élément.

Taille des éléments et défilement

offsetParent, offsetLeft/Top

Ces propriétés sont rarement nécessaires, mais ce sont toujours les propriétés de géométrie "les plus extérieures", nous allons donc commencer par elles.

Le **offsetParent** est l'ancêtre le plus proche que le navigateur utilise pour calculer les coordonnées pendant le rendu. C'est l'ancêtre le plus proche qui est l'un des suivants :

1. **CSS positionné** (position absolute, relative, fixed ou sticky), ou
2. **<td>, <th>**, ou **<table>**, ou
3. **<body>**.

Les propriétés **offsetLeft/offsetTop** fournissent des coordonnées x/y par rapport au coin supérieur gauche de **offsetParent**.

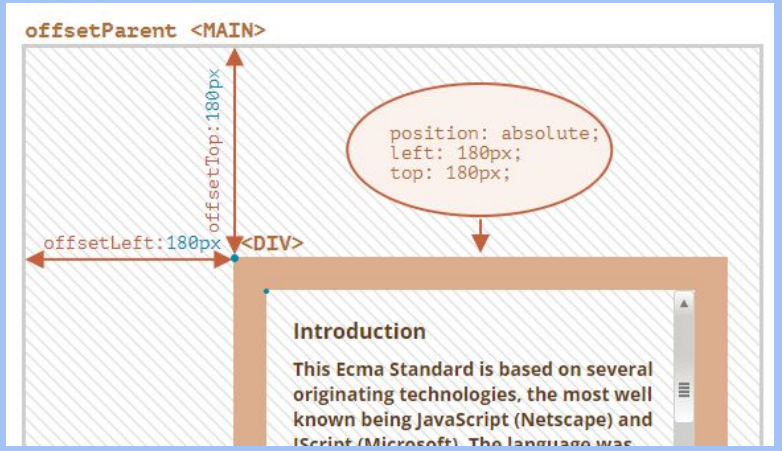
Dans l'exemple ci-dessous, la intérieure a <main> comme offsetParent et offsetLeft/offsetTop décale de son coin supérieur gauche (180) :

Taille des éléments et défilement

```
<main style="position: relative" id="main">
  <article>
    <div id="exemple" style="position:absolute;left:180px;top:180px">
    ...
    </div>
  </article>
</main>
<script>
  alert(example.offsetParent.id); // main
  alert(example.offsetLeft); // 180 (note : un nombre )
  alert(example.offsetTop); // 180
</script>
```

Il y a plusieurs occasions où offsetParent est null :

1. Pour les éléments non affichés (display:none ou pas dans le document).
2. Pour <body> et <html>.
3. Pour les éléments avec position:fixed.

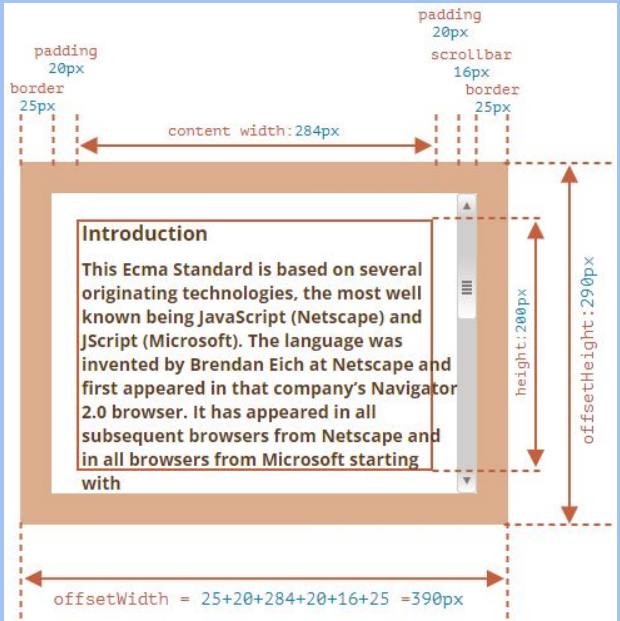


Taille des éléments et défilement

offsetWidth/Height

Passons maintenant à l'élément lui-même.

Ces deux propriétés sont les plus simples. Elles fournissent la largeur/hauteur "extérieure" de l'élément. Ou, en d'autres termes, sa taille complète, y compris les bordures.



Pour notre exemple d'élément :

- offsetWidth = 390 – la largeur extérieure, peut être calculée comme la largeur CSS intérieure (300px) plus les paddings (2 * 20px) et les bordures (2 * 25px).
- offsetHeight = 290 – la hauteur extérieure.

Les propriétés de géométrie sont zéro/nulles pour les éléments qui ne sont pas affichés. Les propriétés de géométrie sont calculées uniquement pour les éléments affichés.Si un élément (ou l'un de ses ancêtres) a display:none ou n'est pas dans le document, alors toutes les propriétés géométriques sont zéro (ou *null* pour offsetParent).

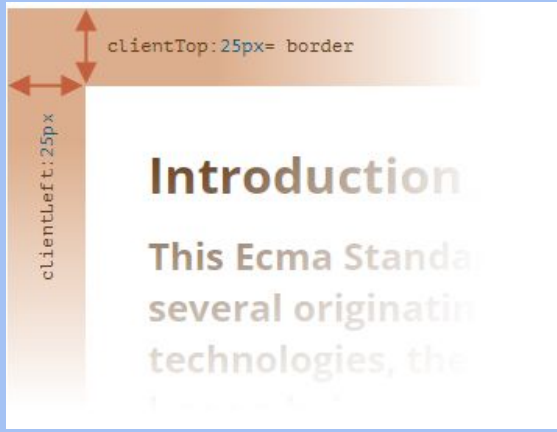
Taille des éléments et défilement

clientTop/Left

À l'intérieur de l'élément, nous avons les bordures. Pour les mesurer, il existe des propriétés clientTop et clientLeft.

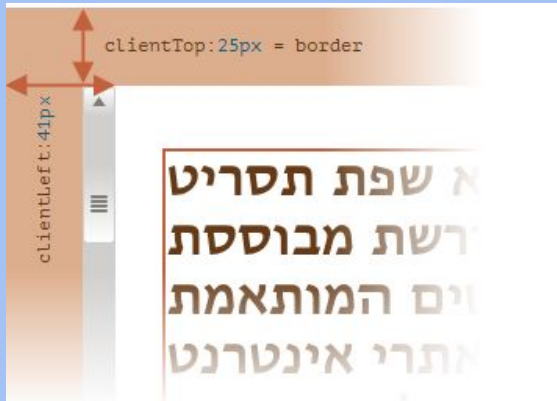
Dans notre exemple :

- clientLeft = 25 – largeur de bordure gauche
- clientTop = 25 – largeur de bordure supérieure



... Mais pour être précis – ces propriétés ne sont pas la largeur/hauteur de la bordure, mais plutôt les coordonnées relatives du côté intérieur par rapport au côté extérieur. Quelle est la différence ?

Il devient visible lorsque le document est de droite à gauche (le système d'exploitation est en arabe ou en hébreu). La barre de défilement n'est alors pas à droite, mais à gauche, puis clientLeft inclut également la largeur de la barre de défilement. Dans ce cas, clientLeft ne serait pas 25, mais avec la largeur de la barre de défilement 25 + 16 = 41. i contre l'exemple en hébreu :

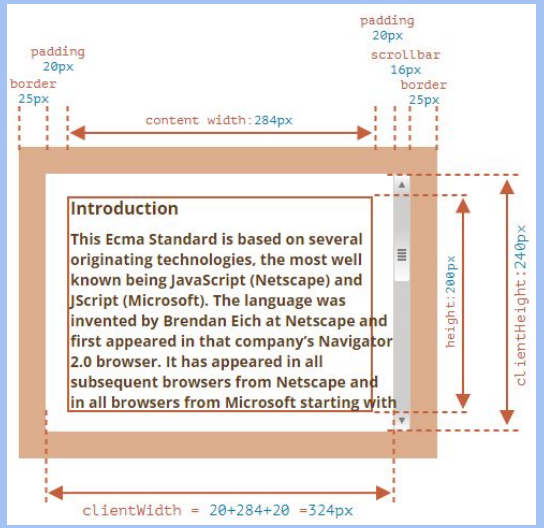


Taille des éléments et défilement

clientWidth/Height

Ces propriétés fournissent la taille de la zone à l'intérieur des bordures des éléments.

Ils incluent la largeur du contenu ainsi que les paddings, mais sans la barre de défilement.



Sur l'image ci-contre, considérons d'abord *clientHeight*.

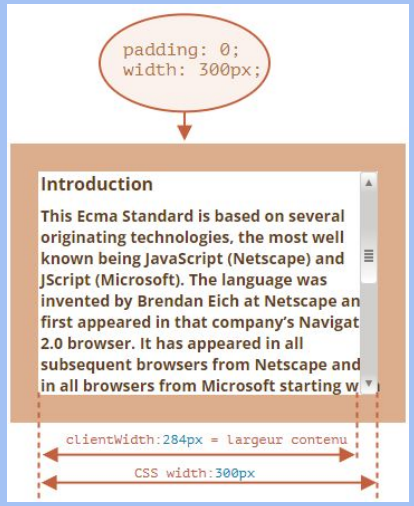
Il n'y a pas de barre de défilement horizontale, c'est donc exactement la somme de ce qui se trouve à l'intérieur des bordures :

hauteur CSS 200px plus paddings supérieur et inférieur (2 * 20px) total 240px.

Maintenant clientWidth – ici la largeur du contenu n'est pas 300px, mais 284px, car 16px sont occupés par la barre de défilement. Ainsi, la somme est 284px plus les paddings gauche et droit, total 324px.

S'il n'y a pas de paddings, alors clientWidth/Height est exactement la zone de contenu, à l'intérieur des bordures et de la barre de défilement (le cas échéant).

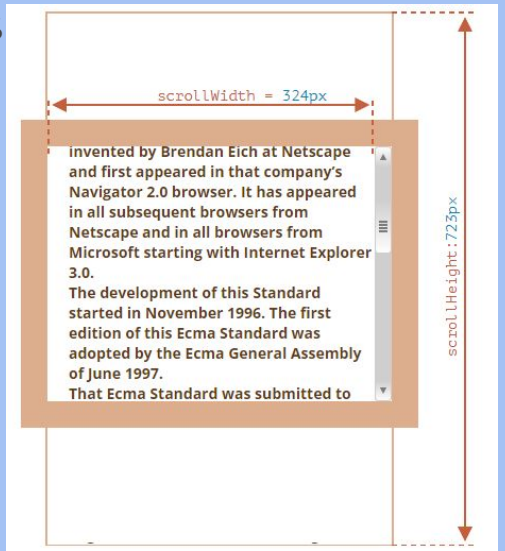
Taille des éléments et défilement



Donc, quand il n'y a pas de padding, nous pouvons utiliser clientWidth/clientHeight pour obtenir la taille de la zone de contenu.

scrollWidth/Height

Ces propriétés sont comme clientWidth/clientHeight, mais elles incluent également les parties déroulantes (cachées) :



Sur l'image ci-contre :

scrollHeight = 723 – est la pleine hauteur intérieure de la zone de contenu, y compris des parties déroulantes.

scrollWidth = 324 – est la largeur intérieure complète, ici nous n'avons pas de défilement horizontal, donc il est égal à clientWidth.

Taille des éléments et défilement

Nous pouvons utiliser ces propriétés pour agrandir l'élément à sa pleine largeur/hauteur.

Comme ceci :

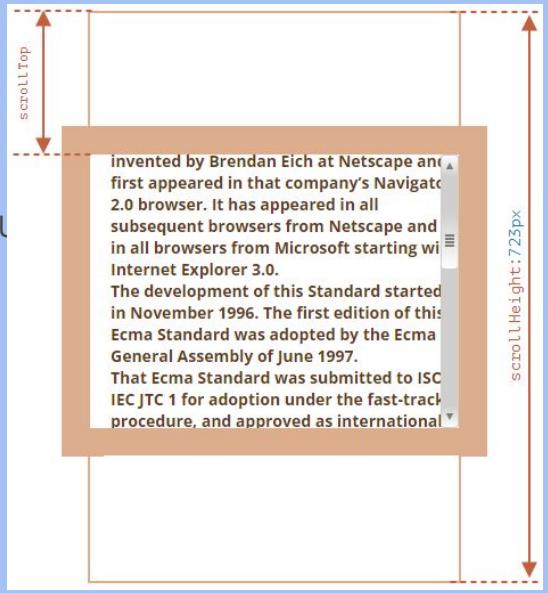
```
// expand the element to the full content height
element.style.height = `${element.scrollHeight}px`;
```

scrollLeft/scrollTop

Les propriétés scrollLeft/scrollTop sont la largeur/hauteur de la partie cachée et déroulée de l'élément.

Sur l'image ci-contre, nous pouvons voir scrollHeight et scrollTop pour un bloc avec u défilement vertical.

En d'autres termes, scrollTop est "combien est déroulé".



scrollLeft/scrollTop peut être modifié

La plupart des propriétés de géométrie ici sont en lecture seule, mais scrollLeft/scrollTop peut être modifié, et le navigateur fera défiler l'élément.

Si vous cliquez sur l'élément ci-dessous, le code elem.scrollTop + = 10 s'exécute.

Cela fait défiler le contenu de l'élément 10px vers le bas.

Taille des éléments et défilement

Ne prenez pas la largeur/hauteur du CSS

Nous venons de couvrir les propriétés géométriques des éléments DOM, qui peuvent être utilisées pour obtenir des largeurs, des hauteurs et calculer des distances.

Mais comme nous le savons du chapitre Styles et classes, nous pouvons lire hauteur et largeur CSS en utilisant getComputedStyle.

Alors pourquoi ne pas lire la largeur d'un élément avec getComputedStyle, comme ceci ?

```
let elem = document.body;
alert( getComputedStyle(elem).width ); // affiche la largeur CSS pour elem
```

Pourquoi devrions-nous plutôt utiliser des propriétés géométriques ? Il y a deux raisons :

- Tout d'abord, la largeur/hauteur en CSS dépend d'une autre propriété : le box-sizing qui définit "ce qui est" la largeur et la hauteur en CSS. Un changement de box-sizing à des fins CSS peut casser un tel JavaScript.
- Deuxièmement, la largeur/hauteur en CSS peut être auto, par exemple pour un élément en ligne :

```
<span id="elem">Hello!</span>
<script>alert( getComputedStyle(elem).width ); // auto</script>
```

Taille des éléments et défilement

Et il y a une autre raison : une barre de défilement. Parfois, le code qui fonctionne correctement sans barre de défilement devient bogué, car une barre de défilement prend de l'espace dans le contenu de certains navigateurs. La largeur réelle disponible pour le contenu est donc inférieure à la largeur CSS.

Et clientWidth/clientHeight en tient compte.

...Mais avec **getComputedStyle**(elem).**width** la situation est différente. Certains navigateurs (par exemple Chrome) renvoient la largeur intérieure réelle, moins la barre de défilement, et certains d'entre eux (par exemple Firefox) – largeur CSS (ignorent la barre de défilement). De telles différences entre navigateurs sont la raison *de ne pas utiliser getComputedStyle*, mais plutôt de s'appuyer sur les propriétés géométriques.

Si votre navigateur réserve l'espace pour une barre de défilement (la plupart des navigateurs pour Windows le font), vous pouvez le tester ci-dessous.

```
<div id="elem" style="overflow-y:scroll;width:300px;height:200px;border:1px solid black">
text text text </div>
<p>L'élément à pour <code>style="width:300px"</code></p>
<button onclick="alert( getComputedStyle(elem).width )">alert( getComputedStyle(elem).width )</button>
```

Taille des éléments et défilement

L'élément avec du texte a comme CSS width:300px.

Sur un OS de bureau Windows, Firefox, Chrome, Edge réservent tous l'espace pour la barre de défilement.
Mais Firefox affiche 300 pixels, tandis que Chrome et Edge affichent moins.
En effet, Firefox renvoie la largeur CSS et les autres navigateurs renvoient la largeur "réelle".

Veuillez noter que la différence décrite concerne uniquement la lecture de ***getComputedStyle****(...).width à partir de JavaScript, visuellement tout est correct.*

JS

Exercices

Taille des éléments et défilement

Quelle est la largeur de la barre de défilement ?

Écrivez le code qui renvoie la largeur d'une barre de défilement standard.

Pour Windows, il varie généralement entre 12px et 20px.
Si le navigateur ne lui réserve pas d'espace (la barre de défilement est à moitié translucide sur le texte, cela arrive également), alors il peut s'agir de 0px.

N.B. : Le code devrait fonctionner pour tout document HTML, ne dépend pas de son contenu.

Taille des éléments et défilement

Quelle est la largeur de la barre de défilement ?

Pour obtenir la largeur de la barre de défilement, nous pouvons créer un élément avec le défilement, mais sans bordures ni paddings.

Ensuite, la différence entre sa largeur totale offsetWidth et la largeur de la zone de contenu interne clientWidth sera exactement la barre de défilement :

```
<script>
// créer une div avec le défilement
let div = document.createElement('div');
div.style.overflowY = 'scroll';
div.style.width = '50px';
div.style.height = '50px';
// doit le mettre dans le document, sinon les tailles seront 0
document.body.append( div );
let scrollWidth = div.offsetWidth - div.clientWidth;
div.remove();
alert( scrollWidth );
</script>
```

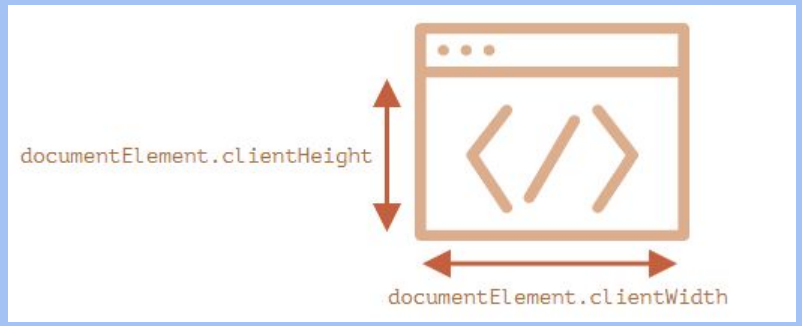
Tailles des fenêtres et défilement

Comment trouver la largeur et la hauteur de la fenêtre du navigateur ? Comment obtenir la largeur et la hauteur complètes du document, y compris la partie déroulante ? Comment faire défiler la page en utilisant JavaScript ?

Pour ce type d'informations, nous pouvons utiliser l'élément de document racine document.documentElement, qui correspond à la balise <html>. Mais il existe des méthodes et des particularités supplémentaires suffisamment importantes pour être prises en compte.

Largeur/hauteur de la fenêtre

Pour obtenir la largeur et la hauteur de la fenêtre, nous pouvons utiliser clientWidth/clientHeight de document.documentElement :



Tailles des fenêtres et défilement

Pas window.innerWidth/Height

Les navigateurs prennent également en charge les propriétés window.innerWidth/innerHeight. Ils ressemblent à ce que nous voulons, alors pourquoi ne pas les utiliser à la place ?

S'il existe une barre de défilement et qu'elle occupe de l'espace, clientWidth/clientHeight fournit la largeur/hauteur sans elle (cela la soustrait). En d'autres termes, elles renvoient la largeur/hauteur de la partie visible du document, disponible pour le contenu.

... Et window.innerWidth/innerHeight inclut la barre de défilement.

S'il y a une barre de défilement et qu'elle occupe de l'espace, ces deux lignes affichent des valeurs différentes :

```
alert( window.innerWidth ); // pleine largeur de fenêtre
alert( document.documentElement.clientWidth ); // largeur de la fenêtre moins la barre de défilement
```

Dans la plupart des cas, nous avons besoin de la largeur de fenêtre disponible : pour dessiner ou positionner quelque chose. C'est-à-dire : à l'intérieur des barres de défilement s'il y en a. Nous devons donc utiliser documentElement.clientHeight/Width.

Taille des éléments et défilement

Largeur/hauteur du document

Théoriquement, comme l'élément de document racine est document.documentElement et qu'il contient tout le contenu, nous pourrions mesurer le document en taille réelle comme document.documentElement.scrollWidth/scrollHeight.

Mais sur cet élément, pour la page entière, ces propriétés ne fonctionnent pas comme prévu. Dans Chrome/Safari/Opera s'il n'y a pas de défilement, alors documentElement.scrollHeight peut être encore moins que documentElement.clientHeight ! Cela ressemble à un non-sens, bizarre, non ?

Pour obtenir de manière fiable la pleine hauteur du document, nous devons prendre le maximum de ces propriétés :

```
let scrollHeight = Math.max(
  document.body.scrollHeight, document.documentElement.scrollHeight,
  document.body.offsetHeight, document.documentElement.offsetHeight,
  document.body.clientHeight, document.documentElement.clientHeight
);
alert('Hauteur totale, avec partie déroulante: ' + scrollHeight);
```

Taille des éléments et défilement

Pourquoi ? Mieux vaut ne pas demander. Ces incohérences viennent des temps anciens ! C''est tout

Obtenez le défilement actuel

Les éléments DOM ont leur état de défilement actuel dans leurs propriétés **scrollLeft/scrollTop**.
Pour le défilement de document, **document.documentElement.scrollLeft/Top** fonctionne dans la plupart des navigateurs, à l'exception des plus anciens basés sur WebKit, comme Safari (bug 5991), où nous devrions utiliser **document.body** au lieu de **document.documentElement.**

Heureusement, nous n'avons pas du tout à nous souvenir de ces particularités, car le défilement est disponible dans les propriétés spéciales window.pageXOffset/pageYOffset :

```
alert( 'Défilement actuel à partir du haut: ' + window.pageYOffset );
alert( 'Défilement actuel à partir de la gauche: ' + window.pageXOffset );
```

Taille des éléments et défilement

Ces propriétés sont en lecture seule.

Également disponible en tant que propriétés window scrollX et scrollY

Pour des raisons historiques, les deux propriétés existent, mais elles sont identiques :

window.pageXOffset est un alias de **window.scrollX**

window.pageYOffset est un alias de **window.scrollY**

Défilement : scrollTo, scrollBy, scrollIntoView

Important :

Pour faire défiler la page avec JavaScript, son DOM doit être entièrement construit.

Par exemple, si nous essayons de faire défiler la page à partir du script dans <head>, cela ne fonctionnera pas.

Taille des éléments et défilement

Les éléments réguliers peuvent défiler en changeant **scrollTop/scrollLeft**.
Nous pouvons faire de même pour la page utilisant document.documentElement.scrollTop/Left (sauf Safari, où **document.body.scrollTop/Left** devrait être utilisé à la place).

Alternativement, il existe une solution plus simple et universelle: des méthodes spéciales **window.scrollBy(x,y)** et **window.scrollTo(pageX,pageY)**.

- La méthode **scrollBy(x, y)** fait défiler la page par rapport à sa position actuelle. Par exemple, scrollBy(0,10) fait défiler la page 10px vers le bas.

```
<button onclick="window.scrollBy(0,10)">window.scrollBy(0,10)</button>
```

- La méthode **scrollTo(pageX,pageY)** fait défiler la page jusqu'aux coordonnées absolues, de sorte que le coin supérieur gauche de la partie visible ait les coordonnées (pageX, pageY) par rapport au coin supérieur gauche du document. C'est comme définir **scrollLeft/scrollTop**.

```
<button onclick="window.scrollTo(0,0)">window.scrollTo(0,0)</button>
```

Ces méthodes fonctionnent de la même manière pour tous les navigateurs.

Taille des éléments et défilement

scrollIntoView

Pour être complet, couvrons une autre méthode : **elem.scrollIntoView(top).**

L'appel à elem.scrollIntoView(top) fait défiler la page pour rendre elem visible. Il a un argument :

Si top=true (c'est la valeur par défaut), alors la page défilera pour faire apparaître elem en haut de la fenêtre. Le bord supérieur de l'élément est aligné avec le haut de la fenêtre.

Si top=false, alors la page défile pour faire apparaître elem en bas. Le bord inférieur de l'élément est aligné avec le bas de la fenêtre.

Le bouton ci-dessous fait défiler la page pour aligner l'élément en haut de la fenêtre :

```
<button onclick="this.scrollIntoView()">this.scrollIntoView()</button>
```

Et ce bouton fait défiler la page pour l'aligner en bas :

```
<button onclick="this.scrollIntoView(false)">this.scrollIntoView()</button>
```

Taille des éléments et défilement

Interdire le défilement

Parfois, nous devons rendre le document "non-défilable". Par exemple, lorsque nous devons le couvrir d'un gros message nécessitant une attention immédiate et que nous voulons que le visiteur interagisse avec ce message, pas avec le document.

Pour rendre le document impossible à faire défiler, il suffit de définir **document.body.style.overflow = "hidden"**.

La page se fige sur son défilement actuel.

Essayez-le :

```
<button onclick="document.body.style.overflow='hidden'">Pas de barre !</button>
```

Nous pouvons utiliser la même technique pour "figer" le défilement pour d'autres éléments, pas seulement pour **document.body**.

L'inconvénient de la méthode est que la barre de défilement disparaît. S'il occupait de l'espace, cet espace est désormais libre et le contenu "saute" pour le remplir.

Cela semble un peu étrange, mais peut être contourné si nous comparons **clientWidth** avant et après le gel, et s'il a augmenté (la barre de défilement a disparu), puis ajoutez un padding à **document.body** à la place de la barre de défilement, pour conservez la même largeur de contenu.