

Javascript module 4

Le DOM et la dynamisation des pages web

Lier du Javascript à une page web	3
L'objet window	3
Les méthodes et propriétés de window	4
Les boîtes de dialogue	4
Les timers	4
setInterval()	4
setTimeout()	4
Arrêter les timers	4
La page courante	5
L'historique, le défilement, ...	5
Le DOM : interagir avec le contenu d'une page	5
Accéder aux éléments de la page	6
Cibler des éléments par rapport à un autre	7
Créer un élément	7
Placer ou déplacer un élément	8
Supprimer un élément	8
Le contenu d'un élément	8
Modifier les propriétés d'un éléments	9
Attribuer un identifiant	9
Modifier le style d'un élément	9
Manipuler les classes	10
Editer les attributs d'un élément	10
Les attributs data-	11
La gestion des évènements	12
Les écouteurs d'événements	12
Les différents évènements	13
Arrêter un écouteur	13
La propagation des événements	13
Manipuler les événements	14



Lier du Javascript à une page web

Le code javascript est déclaré dans une page web au travers d'une balise HTML spécifique, la balise **<script>**.

Cette balise peut directement contenir du code javascript comme ceci :

```
<script type="text/javascript">
  console.log("this is my code");
</script>
```

Mais il est toujours préférable de séparer le code javascript du code HTML en mettant le javascript dans un fichier *.js spécifique et en le liant au fichier HTML avec l'attribut src d'une balise **<script>**, comme ceci :

```
<script type="text/javascript" src="js/script.js"></script>
```

Cette balise **script** peut être déclarée à 2 endroits dans la page :

- Dans la balise **<head>** de la page, tout comme le CSS : Le javascript sera alors exécuté avant l'interprétation de la page.
- A la toute fin de la balise **<body>** de la page : Le javascript sera alors exécuté après le chargement et l'affichage du contenu de la page. Lorsque l'on souhaite interagir avec le contenu de cette page, c'est à cet emplacement que l'on va privilégier l'inclusion de notre javascript.

L'objet window

L'objet **window** représente la fenêtre du navigateur en elle-même. Cet objet représente le contexte le plus haut

Toutes les fonctions que vous avez déclarées dans le contexte global, sont déclarées comme méthode de cet objet.

```
function letMeGo() {
  console.log("Bye");
}

window.letMeGo();
```

Les méthodes et propriétés de window

Les boîtes de dialogue

Les fonctions **prompt()** et **alert()** que nous avons déjà vu sont des méthodes de **window**.

Il existe aussi la méthode **confirm()** dans **window** qui ouvre une boîte de dialogue contenant 2 boutons OK et Annuler. La méthode retourne alors le booléen correspondant au choix de l'utilisateur.

Les timers

L'objet window a des méthodes qui permettent de gérer le temps.

setInterval()

La méthode **setInterval()** permet de déclarer du code qui sera exécuté plusieurs fois à intervalle de temps régulier.

Son appel prend 2 paramètres :

- une fonction de callback avec le code à exécuter ;
- l'intervalle de temps entre 2 exécutions, exprimé en millisecondes.

```
const timer = setInterval(() => {  
    const now = new Date();  
    console.log(now.getHours(), now.getMinutes(), now.getSeconds());  
}, 1000);
```

Le code ci-dessus va donc afficher toutes les secondes dans la console l'heure précise en heures, minutes et secondes.

setTimeout()

La méthode **setTimeout()** fonctionne sur le même principe que **setInterval()**, mais exécute le code de la fonction de callback une seule fois au bout du temps défini.

Arrêter les timers

Pour arrêter les timers, il existe 2 méthodes associées aux méthodes précédentes :

- **clearInterval(timer)**
- **clearTimeout(timer)**

Elles prennent en paramètre la valeur retournée par la déclaration du timer.

Pour l'exemple ci-dessus, cela donne ceci :

```
clearInterval(timer);
```

La page courante

Bien sûr **window** connaît l'adresse de la page où vous vous trouvez, puisqu'il s'agit de l'adresse dans la barre de votre navigateur. Elle est disponible comme ceci :

```
window.location.href
```

L'historique, le défilement, ...

L'objet **window** nous donne également la possibilité de manipuler l'historique (**window.history**), de connaître la valeur de défilement dans la page (**window.scrollX** et **window.scrollY**) et bien d'autres informations sur la fenêtre du navigateur.

Voici la documentation complète de **window** qui vous donnera les propriétés et les méthodes que cet objet vous offre :

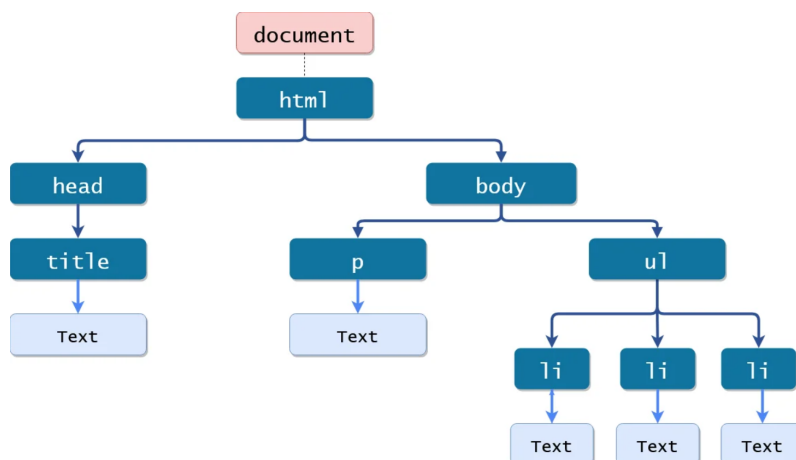
- <https://developer.mozilla.org/fr/docs/Web/API/Window>

Le DOM : interagir avec le contenu d'une page

Quand le navigateur interprète le code HTML d'une page web, il crée une copie de l'arborescence de la page qu'il vient de lire : c'est le DOM pour Document Object Model.

Dans le DOM, chaque nœud de l'arborescence est un élément du document. Les textes sont des nœuds au même titre que ce qui a été déclaré par une balise HTML.

C'est ce DOM qui est accessible via l'objet **document** qui va nous permettre d'interagir avec le contenu de nos pages en JavaScript.



Accéder aux éléments de la page

Chaque élément d'une page Web est donc accessible dans le DOM sous la forme d'un objet JavaScript via l'objet **document**.

Il serait possible de manipuler directement l'arborescence de l'objet **document** en passant par tous les **childNodes**, qui sont en fait des array contenant la liste des éléments enfants.

Voici à quoi ressemblerait notre code pour accéder à un simple élément de notre page :

```
document.documentElement.childNodes[2].childNodes[4].childNodes[0];
```

Heureusement l'objet **document** a toute une collection de méthodes fournies par le navigateur qui nous permettent de pointer sur des éléments précis de notre page :

Méthode	Résultat
<pre>document.getElementById("main");</pre>	Retourne l'élément de la page correspondant à l'id passé en paramètre.
<pre>document.getElementsByClassName("nav");</pre>	Retourne la liste des éléments de la page ayant la classe en paramètre sous la forme d'un Object (HTMLCollection).
<pre>document.getElementsByTagName("p");</pre>	Retourne la liste des éléments de la page correspondant au nom de balise en paramètre sous la forme d'un Object (HTMLCollection).
<pre>document.querySelector("#main .nav");</pre>	Retourne le premier élément de la page qui correspond au sélecteur passé en paramètre.
<pre>document.querySelectorAll(".nav > .nav-list");</pre>	Retourne la liste des éléments de la page qui correspondent au sélecteur CSS passé en paramètre sous la forme d'un Array (NodeList).

Les méthodes à privilégier pour pointer sur des éléments de la page sont **.querySelector()** et **.querySelectorAll()**, en fonction de si l'on cherche à avoir un seul élément ou une liste d'éléments.

Ces méthodes permettent à la fois de sélectionner des id, des noms de classes et les noms de balise, mais aussi de faire une sélection précise en utilisant la hiérarchie du document, tout comme nous le faisons en CSS.

Exemple :

```
document.querySelectorAll("#main .nav > .nav-list");
```

Sélectionne les éléments qui ont la classe **nav-list** qui sont des enfants direct d'un élément portant la classe **nav**. Cet élément devant lui être enfant de l'élément portant l'id **main**.

Pour construire les sélecteurs, référez-vous au document **Annexe - Les sélecteurs CSS**.

Cibler des éléments par rapport à un autre

Lorsque vous avez atteint un élément de votre page, vous pouvez sélectionner les éléments qui l'entourent avec différentes propriétés.

Vous avez sélectionné un élément et l'avez conservé dans la variable **nav** comme ceci :

```
const nav = document.querySelector("#main .nav");
```

Vous pouvez alors sélectionner les éléments qui l'entourent comme ceci :

	Uniquement les éléments	Tous les noeuds dont les textes
Retourne les enfants direct	<code>nav.children;</code>	<code>nav.childNodes;</code>
Cible le premier enfant	<code>nav.firstElementChild;</code>	<code>nav.firstChild;</code>
Cible le dernier enfant	<code>nav.lastElementChild;</code>	<code>nav.lastChild;</code>
Cible le noeud juxtaposé qui le précède	<code>nav.previousElementSibling;</code>	<code>nav.previousSibling;</code>
Cible le noeud juxtaposé qui le succède	<code>nav.nextElementSibling;</code>	<code>nav.nextSibling;</code>
Cible le noeud parent	<code>nav.parentElement;</code>	<code>nav.parentNode;</code>

Note: Dans la pratique, nous allons utiliser uniquement les méthodes permettant de pointer les éléments.

Créer un élément

Pour créer un élément il faut appeler la méthode **document.createElement()** en donnant en paramètre le type de balise à créer.

```
const newSection = document.createElement("section");
```

En l'état le nouvel élément est créé mais n'est pas encore dans la page.

Placer ou déplacer un élément

La technique la plus utilisée pour placer un élément dans une page est d'utiliser la méthode **.appendChild()**. Elle s'applique au futur élément parent et prend en paramètre l'élément à ajouter. L'élément sera alors ajouté comme dernier enfant.

Voici comment ajouter la section que l'on a créé ci-dessus à la fin du body.

```
document.body.appendChild(newSection);
```

Il est aussi possible d'utiliser la méthode **.insertBefore()** de la même manière, mais cette fois pour positionner un élément juste avant un élément dans le même parent. La méthode s'applique aussi sur le parent, mais prend un second paramètre qui est l'élément avant lequel on place le nouvel élément.

```
const newDiv = document.createElement("div");  
document.body.insertBefore(newDiv, newSection);
```

Attention: Si vous appliquez ces méthodes à des éléments qui sont déjà présents dans le DOM vous allez les déplacer dans la page et non les copier. Pour copier un élément, vous pouvez lui appliquer la méthode **.cloneNode(true)**.

Supprimer un élément

Pour supprimer un élément du DOM, vous pouvez lui appliquer la méthode **.remove()**. Mais celle-ci n'est pas encore supportée par tous les navigateurs (<https://caniuse.com/?search=remove>).

```
document.querySelector("#main .button").remove();
```

Une alternative est de sélectionner l'élément parent et d'y appliquer la méthode **.removeChild()** avec en paramètre l'élément à supprimer.

```
const button = document.querySelector("#main .button");  
button.parentElement.removeChild(button);
```

Le contenu d'un élément

Il existe 2 propriétés principales sur les éléments qui permettent d'accéder et de modifier le contenu d'un élément :

- **.innerHTML** permet de récupérer ou de modifier le contenu en HTML dans un élément.

```
const section = document.querySelector("section");
// Retourne le contenu HTML
console.log(section.innerHTML);
// Remplace le contenu actuel par du HTML
section.innerHTML = "<h2>A new title</h2>";
// Ajoute ce HTML à la fin du contenu actuel
section.innerHTML += "<article>Lorem ipsum dolor sit amet</article>";
```

- **.innerText** (ou **.textContent**) permet de récupérer ou de modifier le contenu textuel uniquement d'un élément. Il fonctionne de la même manière que **.innerHTML** excepté qu'il ne retourne pas et n'interprète pas le HTML.

Modifier les propriétés d'un éléments

Attribuer un identifiant

Il est possible d'attribuer un identifiant à un élément en éditant directement sa propriété **id** comme ceci :

```
const newButton = document.createElement("button");
newButton.id = "buttonNext";
```

Modifier le style d'un élément

Sur chaque élément vous pouvez accéder à **.style** qui contient toute une liste de propriétés permettant d'éditer l'ensemble du style de l'élément, comme nous le ferions en CSS.

Par exemple, pour passer le texte de ce bouton en rouge et changer sa taille, je peux effectuer ceci :

```
const button = document.querySelector("#main .button");
button.style.color = "#ff0000";
button.style.fontSize = "26px";
```

En inspectant l'élément dans la page, vous verrez que le Javascript s'est contenté d'ajouter un attribut **style=""** en y ajoutant les propriétés CSS correspondant à notre code.

```
<button style="color: #ff0000; font-size: 26px;">Click me</section>
```

Attention: Si vous souhaitez changer dynamiquement le style d'un élément, il est préférable de se contenter d'utiliser le Javascript pour lui appliquer une classe - classe qui affecte un style défini dans votre fichier CSS : c'est son job !

Manipuler les classes

Chaque élément possède la propriété **.className** qui permet d'accéder et d'éditer la chaîne de caractère de l'ensemble des classes de l'élément, comme elles apparaissent dans le HTML.

Chaque élément possède aussi **.classList** qui permet de gérer les classes de l'élément avec des méthodes dédiées.

Méthode	Description	Exemple
.add()	Ajoute la classe en paramètre à l'élément.	<pre>button.classList.add("blue");</pre>
.remove()	Retire la classe en paramètre à l'élément.	<pre>button.classList.remove("blue");</pre>
.toggle()	Ajoute la classe en paramètre si elle est absente et la retire si elle existe.	<pre>button.classList.toggle("blue");</pre>
.contains()	Retourne un booléen pour savoir si l'élément a la classe en paramètre.	<pre>button.classList.contains("blue");</pre>

C'est la façon la plus propre d'appliquer et d'agir sur le style des éléments en Javascript, en s'appuyant sur des classes définies en CSS.

Editer les attributs d'un élément

Les éléments du DOM ont également une série de méthodes qui permettent d'éditer leurs attributs. Si par exemple vous voulez définir l'attribut ou connaître le **href** d'un lien ou le type d'un champ de formulaire, ce sont ces méthodes qui vont vous être utiles.

Nous allons sélectionner un lien de notre page et le placer dans une variable pour les exemples qui suivent:

```
const link = document.querySelector(".link");
```

Méthode	Description	Exemple
.getAttribute()	Retourne la valeur de l'attribut en paramètre.	<pre>link.getAttribute("href")</pre>

Méthode	Description	Exemple
.setAttribute()	Définit l'attribut passé en premier paramètre avec la valeur en second paramètre.	<code>link.setAttribute("href", "page2.html")</code>
.hasAttribute()	Retourne un booléen pour savoir si l'attribut en paramètre est défini.	<code>link.hasAttribute("id")</code>
.removeAttribute()	Retire l'attribut passé en paramètre.	<code>link.removeAttribute("href")</code>

Il y a des exceptions, comme la source d'une image qui se récupère et se modifie directement via la propriété **.src**.

Les attributs data-

Il est possible de créer sur les éléments HTML des attributs personnalisés dont le nom commence par "data-". Cela permet d'ajouter des données riches aux éléments d'une page HTML et d'y accéder facilement en Javascript. Cela se fait grâce à la propriété **.dataset** d'un élément.

Un attribut défini avec des tirets comme **data-my-attribute**, sera accessible en javascript sous la forme **element.dataset.myAttribute**. C'est-à-dire le nom de l'attribut sans le préfixe data et transformé en camelCase.

Voici un exemple de HTML :

```
<p id="my-paragraph">This is my text and nobody will see it before you
click on the link below.</p>

<a href="#" data-label-for="my-paragraph">Click here to view my text</a>
```

Et un exemple de code Javascript correspondant :

```
document.querySelector("[data-label-for]").addEventListener("click", function(event) {

    event.preventDefault();

    document.getElementById(this.dataset.labelFor).classList.toggle("active");

});
```

La gestion des évènements

Nous avons désormais les outils de base pour atteindre et modifier les éléments d'une page web. Mais modifier une page dès son chargement a un intérêt très limité.

Ce que l'on souhaite faire la plupart du temps en Javascript, c'est modifier notre page lorsqu'un certain événement se produit, comme dans le pseudo code suivant :

Au clic sur ce bouton, une image apparaît.

Au défilement de la page, une fenêtre apparaît.

Au clic sur cet élément, la page change de couleur.

Lorsqu'un événement survient, je change quelque chose.

Événement, programme.

Nous savons donc comment faire la partie **programme** de ces phrases. Voyons donc comment les exécuter en fonction d'un **événement**.

Les écouteurs d'événements

Les éléments du DOM ont tous une méthode permettant de manipuler et de capturer les événements qui leur arrive. Cette méthode **.addEventListener()** est appelée écouteur d'évènement - ou event handler.

Elle prend 2 paramètres :

- le nom de l'événement que l'on souhaite capturer sous la forme d'un string ;
- une fonction de callback qui correspond aux actions à effectuer lors de la capture de l'événement :
 - L'événement sera passé en paramètre de cette fonction.
 - **this** dans le contexte de la fonction sera l'élément sur lequel l'événement a été capturé.

Voici un exemple : **Au clic sur le bouton, le bouton devient rouge ou revient à son style d'origine.**

```
function toggleRed(event) {  
    this.classList.toggle("button-red");  
}
```

```
document.querySelector(".button").addEventListener('click', toggleRed);
```

Dans cet exemple, on définit un écouteur sur l'événement **"click"**.

Lorsque cet événement se produira, la fonction de callback **toggleRed** sera appelée.

Cette fonction manipule **this** qui correspond à l'élément sur lequel l'événement a été capturé : ici notre bouton. Grâce à la méthode **.classList.toggle()** la classe **"button-red"** est ajoutée ou retirée.

Les différents événements

Il existe un grand nombre d'événements qui ont chacun un nom (string) qui sert à le déclarer. Ils peuvent permettre de détecter les clics, les défilements de la page, la fin du chargement de celle-ci ou encore les actions sur les touches du clavier. Certains sont valables sur tous les types d'éléments d'une page. Certains sont spécifiques à des balises précises comme les formulaires par exemple. Certains sont spécifiques à l'objet **document** ou à l'objet **window**.

Vous trouverez l'ensemble des événements que vous pouvez écouter sur cette documentation :

- https://developer.mozilla.org/fr/docs/Web/Events#listing_des%C3%A9v%C3%A9nements

Arrêter un écouteur

Si on veut arrêter un écouteur et donc arrêter de capturer l'événement, il faut utiliser la méthode antagoniste **.removeEventListener()**. Celle-ci prend exactement les mêmes paramètres que la première. Dans notre exemple précédent du bouton on peut donc arrêter ce comportement au clic avec cette ligne de code :

```
document.querySelector(".button").removeEventListener('click', toggleRed);
```

La propagation des événements

Il est important de comprendre la notion de propagation d'un événement.

Lorsque qu'un événement se produit sur un élément, comme par exemple le clic sur notre bouton, l'événement remontera toute l'arborescence du DOM, élément parent après élément parent, jusqu'au **body**, puis jusqu'à la racine **document** et même jusqu'à l'objet **window**.

L'événement peut alors être capturé par n'importe quel écouteur défini sur un élément se trouvant sur le passage de sa propagation.

Il y a donc une différence entre l'élément sur lequel l'événement s'est produit et l'élément qui a capturé l'événement.

Pour voir la différence, nous allons reprendre l'exemple précédent du clic sur notre bouton, mais cette fois en ajoutant un autre écouteur au clic également sur le body.

Info: Notez que cette fois les fonctions de callback sont directement des fonctions anonymes définies en paramètre.

```
document.querySelector(".button").addEventListener('click',
function(event) {
    console.log("button", this);
    console.log("button", event.target);
});
document.body.addEventListener('click', function(event) {
    console.log("body", this);
    console.log("body", event.target);
});
```

Avec cet exemple, on se rend compte que lorsque l'on clic sur notre bouton, l'événement est capturé par l'écouteur défini sur le bouton, puis par celui défini sur le body de la page : C'est la propagation.

Il est intéressant de voir ici la différence de valeur entre **this** et **event.target** dans la console lors de la propagation de cet événement :

- Dans le cas de l'écoute de notre bouton, le **this** et le **event.target** ont bien la même valeur : C'est l'élément bouton.
- Dans le cas de l'écoute du body, le **this** a pour valeur le body, alors que le **event.target** a pour valeur le bouton.


En fonction du contexte ces 2 variables peuvent pointer sur le même élément, ou pointer sur des éléments différents.

Il faut donc retenir que dans la fonction de callback d'un écouteur d'événement :

- **this** a pour valeur l'élément sur lequel l'écouteur a été déclaré ;
- **event.target** a pour valeur l'élément qui est à l'origine de l'événement.

Manipuler les événements

Cette valeur **event** que nous avons récupérée contient des propriétés et des méthodes qui permettent d'effectuer certaines choses intéressantes :

- 
- **`event.stopPropagation()`** permet de stopper la propagation de l'événement et ainsi d'éviter qu'il soit capturé plus haut dans le DOM.
 - **`event.preventDefault()`** permet d'annuler le comportement par défaut de l'événement, comme par exemple le changement de page suite au clic sur un lien ou l'envoi d'un formulaire.

Pour plus d'information sur la manipulation des événements, voici une documentation complète :

- <https://developer.mozilla.org/fr/docs/Web/API/Event>