**Javascript Dom Avancé**

Dans les chapitres précédents, nous avons :

* Utilisé l’API fournie par le navigateur pour accéder aux éléments du DOM d’une page web/HTML,
* Modifié les classes et le rendu d’éléments,
* Défini une réaction aux évènements déclenchés par l’utilisateur,
* Et introduit les concepts de Programmation Orientée Objet, et de classes.

Dans ce chapitre, nous allons voir :

* Comment en savoir plus sur un évènement qui a été déclanché par l’utilisateur,
* Comment naviguer dans le DOM d’une page web/HTML, de noeud en noeud,
* Et comment en modifier la structure.

**1. Utilisation de la classe Event**

Reprenons un exemple de page web avec gestion d’évènement *click* sur un élément HTML:

<button id="mon-bouton">Mon Beau Bouton</button>

Pour afficher un alert à chaque fois que l’utilisateur cliquera sur ce bouton, nous avons vu qu’il fallait affecter une fonction à la propriété onclick de l’objet JavaScript représentant ce bouton:

document.getElementById('mon-bouton').onclick = function() {

alert('bonjour !');

};

En effet, les propriétés d’évènements (onclick, onchange, ou autre) d’un élément permettent d’indiquer au navigateur quelle fonction appeler lorsque l’évènement correspondant est déclenché par l’utilisateur.

Ce que nous n’avons pas encore vu, en revanche, c’est que le navigateur passe systématiquement un paramètre lorsqu’il appelle cette fonction: une instance de la [classe Event](https://developer.mozilla.org/fr/docs/Web/API/Event).

Affichons la valeur de ce paramètre event (objet-instance de la classe Event) dans la console, afin d’en découvrir les propriétés:

document.getElementById('mon-bouton').onclick = function(event) {

console.log(event);

};

La propriété la plus couramment utilisée est event.currentTarget. En effet, elle a pour valeur l’élément HTML sur lequel l’évènement a été intercepté par notre fonction.

Cette propriété est particulièrement utile dans le cas où nous voulons affecter une même fonction de gestion d’évènement sur plusieurs éléments, mais que cette fonction a besoin de savoir sur lequel de ces éléments l’évènement a été déclenché.

Par exemple:

<button>Mon 1er Bouton</button>

<button>Mon 2ème Bouton</button>

<button>Mon 3ème Bouton</button>

var boutons = document.getElementsByTagName('button');

for (var i = 0; i < boutons.length; i++) {

boutons[i].onclick = function(event) {

alert('bouton clické: ' + event.currentTarget.innerHTML);

// => le contenu du bouton clické va être affiché dans l'alert

};

}

À noter que, selon le type d’évènement auquel est associée une fonction, le paramètre event peut être une instance de sous-classes contenant des propriétés et méthodes supplémentaires.

Par exemple:

* le paramètre event d’une fonction liée à l’évènement mousemouse (appelée à chaque mouvement de la souris) est une instance de la classe [MouseEvent](https://developer.mozilla.org/fr/docs/Web/API/MouseEvent) qui possède les propriétés clientX et clientY, afin de connaître la position de la souris sur l’écran.
* le paramètre event d’une fonction liée à l’évènement keydown (appelée quand l’utilisateur tape un caractère dans un champ) est une instance de la classe [KeyboardEvent](https://developer.mozilla.org/fr/docs/Web/API/KeyboardEvent) qui possède les propriétés key et keyCode, permettant de connaître quelle touche a été tapée par l’utilisateur.

Par ailleurs, la classe Event et ses sous-classes définissent une méthode preventDefault() qui permet d’annuler le traitement habituel d’un évènement qui a été intercepté par une fonction. Cette méthode est par exemple utile pour empêcher l’ouverture d’un hyperlien <a>, la soumission d’un formulaire, ou l’ajout de caractères dans un champ.

**2. Navigation dans le DOM**

Toute page HTML est structurée sous forme hiérarchique (arbre): chaque élément HTML a un élément parent, et peut avoir plusieurs éléments enfants.

Reprenons par exemple notre page web ne contenant qu’un bouton:

<body>

<button id="mon-bouton">Mon Beau Bouton</button>

</body>

Dans cette page, l’élément <button> a l’élément <body> comme *parent*, et un *noeud texte* (le contenu textuel de l’élément) comme *enfant*. On peut aussi dire que <button> est un *noeud enfant* de <body>.

Ici, le seul *enfant* de notre élément <button> est un noeud textuel, mais il serait possible que ce même élément ait d’autres éléments comme *enfants*.

Supposons que notre bouton soit référencé par la variable element:

var element = document.getElementById('mon-bouton');

Comme tout élément HTML représenté en JavaScript (et donc instance de la classe [Element](https://developer.mozilla.org/fr/docs/Web/API/Element)), notre variable element possède trois propriétés utiles pour nous aider à naviguer dans ses noeuds parent et enfants:

* parentNode est le noeud/élément HTML parent de l’élément,
* children est un tableau de d’éléments, enfants directs de l’élément, et
* [childNodes](https://developer.mozilla.org/en-US/docs/Web/API/Node/childNodes) est un tableau de noeuds HTML (éléments et/ou noeuds textuels), enfants directs de l’élément.

Dans notre exemple:

* element.parentNode est l’équivalent de document.body; (l’élément <body> de notre page)
* element.children est un tableau vide, car il n’y a pas d’éléments à l’intérieur du bouton, et
* element.childNodes est un tableau qui ne contient que le noeud textuel ayant pour valeur “Mon Beau Bouton”.

**Exercice 11 Changer la couleur du parent au clic**

## 3. Modification de la structure du DOM

À ce stade, nous savons accéder à des éléments du DOM, modifier leur contenu et rendu, et accéder à leurs parent et enfants.

Dans cette partie, nous allons voir comment créer, rattacher et supprimer des noeuds du DOM.

### Créer et rattacher un noeud dans le DOM

Pour ajouter un élément HTML dans le DOM d’une page web, le navigateur fournit les méthodes suivantes:

* document.createElement(nom) pour créer un élément,
* document.createTextNode(texte) pour créer un noeud textuel, et
* conteneur.appendChild(element) pour ajouter un élément comme enfant d’un autre élément.

Donc, pour ajouter un bouton au <body> d’une page HTML, il faudra exécuter les instructions JavaScript suivantes ;

// 1. créer l'élément

var bouton = document.createElement('button');

// 2. créer le contenu du bouton (noeud textuel)

var texteDuBouton = document.createTextNode('Mon beau bouton');

// 3. ajouter le contenu au bouton

bouton.appendChild(texteDuBouton);

// 4. ajouter le bouton au body de la page

document.body.appendChild(bouton);

### Supprimer un noeud du DOM

Le retrait d’un noeud du DOM est l’opération inverse de celle d’ajout (appendChild()): elle consiste à appeler la méthode conteneur.removeChild(noeud), où conteneur est le noeud parent duquel on souhaîte retirer noeud.

Ainsi, pour supprimer le bouton que nous avons ajouté à la page de l’exemple précédent, il faudra exécuter:

document.body.removeChild(bouton);

… ou, de manière plus générale :

bouton.parentNode.removeChild(bouton);

**Exercice 11 construire une page web en Javascript**

**Les components WEB**

Sur le Web, un composant est un programme qui permet de fournir une interface intégrable, conçue pour un usage précis, mais personnalisable par chaque développeur.

Par exemple, si je développe et diffuse un composant de galerie d’images, d’autre développeurs pourront intégrer simplement mon composant sur leur site, et y ajouter les images de leur choix.

Les composants permettent donc d’enrichir le contenu, l’esthétique, et/ou les interactions proposées par un site Web, en ré-utilisant du code qui n’a pas été écrit spécifiquement pour ce site.

Il existe de très nombreux composants publiés sur Internet et utilisables gratuitement. Pour la plupart, ils sont basés sur la librairie “jQuery” (dont nous parlerons plus tard dans ce cours). Mais nombreux ont été conçus en JavaScript/DOM natif (aussi appelé “Vanilla JavaScript”), et peuvent donc fonctionner sans jQuery.

**Note: Dans le cadre de ce cours, l’usage (direct ou pas) de jQuery ne sera pas accepté.**

Voici quelques exemples de composants natifs intégrables librement:

* [Sweet Alert](http://t4t5.github.io/sweetalert/)
* [Dialog Modal](https://frend.co/components/dialogmodal/)
* [Animate on scroll](https://github.com/michalsnik/aos)
* [Tranglify](http://qrohlf.com/trianglify/)

Vous pourrez trouver d’autres composants natifs sur le site [plainjs.com](https://plainjs.com/javascript/plugins/).

À ce stade, nous allons apprendre à réaliser un composant *simple*, dans le sens où celui-ci ne sera pas suffisamment modulaire pour être intégré plusieurs fois sur une même page. Nous verrons plus tard comment faire cela.

### Exercice 1: intégrer un composant simple

Pour comprendre le principe d’usage des composants web:

1. Intégrer le composant Sweet Alert (cf liste ci-dessus) sur une page web,
2. Puis personnaliser son aspect à l’aide de sa documentation.

### Généricité et instructions d’intégration

Afin que notre composant puisse être intégré de manière personnalisée par chaque développeur, quel que soit le site Web en question et son contenu, il est important de définir quelques règles et abstractions.

D’abord, un composant doit être simple à intégrer, et un utilisateur de composant ne devrait jamais avoir à consulter ni modifier le code source du composant. Il va falloir donc que le composant soit suffisamment générique et configurable depuis le site Web qui l’intégrera.

Par exemple, un composant de galerie d’images doit s’adapter à une liste d’images fournie par un développeur désirant l’intégrer, et l’intégrateur ne devrait en aucun cas avoir à modifier la liste d’images dans le code source du composant.

Pour cela, un composant:

* Ne doit pas contenir de valeurs et références littérales; (ex: nombre d’images et/ou URLs des images stockés “en dur” dans le code du composant)
* Doit définir et documenter les règles et contraintes éventuelles que devront respecter les intégrateurs du composant.

Exemples d’instructions d’intégration fournies par la documentation d’un composant:

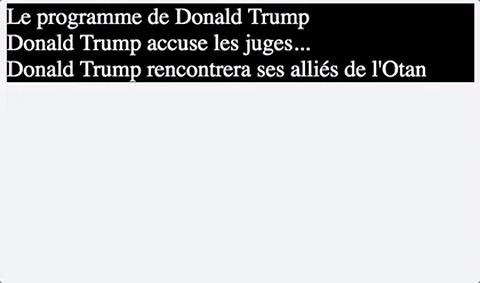
* *Toutes les images à afficher dans la galerie doivent être des balises <img> portant la classe carousel-img*;
* Ou encore : *appeler la fonction creerGallerie() (définie par le composant) en passant en paramètres l’identifiant du DIV devant contenir la galerie, et un tableau d’URLS d’images*.

Prenez le temps d’analyser les instructions d’intégration des composants listés plus haut.

**Exercice 12 Le caroussel**

#### Composant: Accordéon

Un “accordéon” est un composant proposant plusieurs rubriques à l’utilisateur, et lui permettant d’afficher le contenu d’une rubrique à la fois, en cliquant sur son titre.

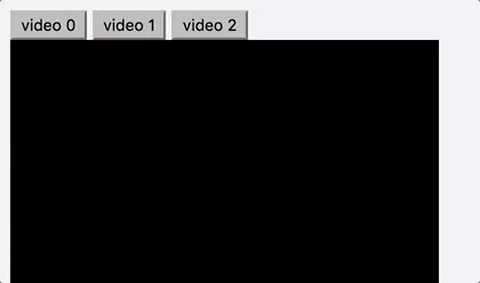


Pour développer un accordéon, il faut maîtriser :

* les sélecteurs DOM (getElementById(), getElementsByClassName() et/ou getElementsByTagName()),
* la manipulation de styles et/ou classes CSS en JavaScript,
* la capture d’événements onclick,
* et la gestion du scope.

#### Composant: Galerie vidéo

Une galerie vidéo permet à l’utilisateur de visualiser une vidéo, en cliquant parmi une sélection fournie.



Pour développer une galerie vidéo, il faut maîtriser:

* les sélecteurs DOM (getElementById(), getElementsByClassName() et/ou getElementsByTagName()),
* la capture d’événements onclick,
* l’intégration d’<iframe> (ex: à l’aide de innerHTML),
* et la gestion du scope.

**Exercice 12** **Développer et documenter un composant réutilisable**

**La gestion des erreurs en Javascript**

## Gestion d’erreurs en JavaScript

Dans un programme informatique comme dans la vie, tout ne se passe pas toujours comme prévu !

Dans ce chapitre, nous allons découvrir les différents types d’erreurs qui peuvent survenir dans un programme JavaScript, et comment les gérer dans notre code.

### Erreurs synchrones et asynchrones

Avant de catégoriser les différents types d’erreurs, définissons la différence entre du code synchrone et asynchrone.

Une instruction synchrone est dite “bloquante”. C’est à dire que l’instruction suivante sera exécutée seulement une fois que l’opération invoquée par cette instruction aura fini de s’exécuter.

Exemple:

// alert() est une fonction synchrone car:

alert('cliquez sur OK'); // exécution du programme bloquée jusqu'au clic de l'utilisateur

console.log('bonjour !');

// bonjour sera affiché dans la console seulement une fois que l'utilisateur aura cliqué

Une instruction asynchrone est dite “non bloquante”. C’est à dire qu’elle d’éclanche une opération qui va se dérouler en arrière plan, pendant que les instructions suivantes du programme vont être exécutées.

Conséquence importante: le résultat de l’exécution d’une instruction asynchrone n’était pas disponible immédiatement, il va falloir définir et fournir un fonction qui sera appelée à la fin de l’opération asynchrone correspondante.

Exemple:

function direBonjour() {

console.log('bonjour !');

}

setTimeout(direBonjour, 1000);

console.log('salut !');

// => salut va s'afficher avant bonjour, car setTimeout() est une fonction asynchrone.

Généralement, une instruction asynchrone consiste à appeler une fonction, en passant une autre fonction de *callback* en paramètre. Cette fonction de *callback* sera appelée à la fin de l’opération asynchrone, et un résultat sera passé en paramètre de cette fonction.

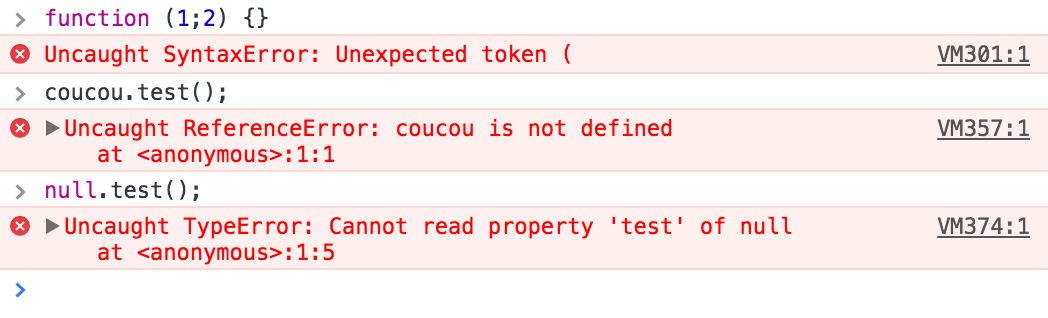
Le résultat passé en paramètre de la fonction de *callback* peut être une valeur résultante de l’opération (équivalent à l’usage de l’instruction return, dans le cas d’une fonction synchrone) ou une description de l’erreur qui serait éventuellement survenue pendant cette opération. (équivalent à l’usage de l’instruction throw, dans le cas d’une fonction synchrone)

On peut alors discerner quatre types d’erreurs:

| **Synchrone** | **Asynchrone** |
| --- | --- |
| Erreur de programmation immédiate | Erreur de programmation tardive |
| Erreur opérationnelle gérable par try-catch | Erreur opérationnelle gérable par fonction de callback |

### Erreurs de programmation

Avez-vu déjà vu ce type d’erreurs dans la console JavaScript de votre navigateur ?



Les erreurs de programmation sont causées par le développeur du programme. Souvent par inattention, ou cas non prévus.

Exemples d’erreurs de programmation:

* erreur de syntaxe. ex: il manque une accolade fermante
* usage d’un symbole non défini: variable ou instruction mal orthographiée
* usage d’une propriété ou méthode d’un objet non défini (undefined)
* non respect du type d’un paramètre de fonction
* fonction de callback non passée en paramètre de l’appel d’une fonction asynchrone

Quand une erreur survient dans un programme JavaScript s’exécutant dans un navigateur web, ce dernier arrête l’exécution de ce programme et affiche l’erreur dans la console, afin que le développeur puisse corriger son programme.

Les erreurs de ce type peuvent être corrigées en relisant attentivement le code (ex: localiser les paires de parenthèses et accolades), et en exécutant des tests couvrant un maximum de cas limites (ex: que se passe-t-il si l’utilisateur clique sur le bouton “annuler” au lieu de fournir la valeur demandée ?).

### Erreurs opérationnelles

Une erreur opérationnelle n’est pas directement causée par le développeur du programme. Elle peut survenir dans des cas en dehors de son contrôle.



Exemples d’erreurs opérationnelles:

* le serveur auquel une requête AJAX a été envoyée met trop de temps à répondre
* le paramètre passé à JSON.parse() n’est pas un objet JSON valide
* le navigateur de l’utilisateur a empêché le pop-up de s’ouvrir
* l’utilisateur a désactivé le stockage de cookies, ou la localisation
* l’utilisateur n’est plus connecté à internet
* le système est à cours de mémoire

Ce type d’erreurs arrive généralement de manière imprévue, et peut nuire gravement à l’expérience de l’utilisateur si votre programme ne les gère pas correctement. En effet, comme pour les erreurs de programmation, le navigateur interrompra l’exécution de votre programme si une de ces erreurs survient et que votre programme ne décrit pas comment les gérer.

L’interception d’une erreur opérationnelle dépend de la nature de l’instruction qui a l’a indirectement causée: synchrone ou asynchrone.

#### Cas synchrone

Dans le cas d’une opération synchrone, il suffit de contenir l’instruction pouvant causer une erreur à l’intérieur d’un bloc try-catch.

Par exemple, la fonction JSON.parse() est synchrone, et déclanchera une erreur si la chaîne de caractères passée en paramètre n’est pas une chaine de caractère ou si elle ne contient pas du JSON valide. Pour gérer cette erreur dans notre programme:

try {

var json = JSON.parse('{ JSON invalide }'); // cause une erreur interceptée par le catch()

console.log('résultat:', json); // => cette instruction ne sera pas exécutée

} catch (error) {

console.log('une erreur est survenue:', error.message);

}

console.log('cette ligne s\'affichera dans tous les cas.');

#### Cas asynchrone

Dans la plupart des cas:

* une erreur asynchrone n’interrompra pas directement l’exécution du programme dans le navigateur.
* mettre l’appel de fonction asynchrone dans un try-catch est inutile car l’erreur n’est pas directement causée par le déclenchement de l’opération, mais par le traitement qui est indirectement occasionné par lui.

Par contre, il est important de gérer ce type d’erreurs quand même afin d’en informer l’utilisateur, et éventuellement l’aider à trouver une solution afin de ne pas rester bloqué.

Pour rappel, quand on déclenche une opération asynchrone, il faut généralement fournir une fonction de *callback* en paramètre de la fonction correspondante. La gestion d’erreur se fait alors en analysant la valeur d’un des paramètres passés à l’appel de cette fonction de *callback*.

Dans les programmes Node.js, c’est le premier paramètre d’une fonction de *callback* qui est conventionnellement réservé à cet usage, et contient soit une instance de la classe Error, soit la valeur null ou undefined (dans le cas où aucune erreur n’aurait eu lieu).

uneFonctionAsynchrone(function(err, res) {

if (err) {

alert('une erreur est survenue: ' + err.message);

} else {

alert('resultat de la fonction: ' + res);

}

});

Mais les conventions dépendent de la fonction qui est appelée. Par exemple, la fonction getCurrentPosition() de l’API Geolocation prend deux fonctions de *callback*: une première qui sera appelée quand la position de l’utilisateur sera déterminée, et une deuxième qui ne sera appelée qu’en cas d’erreur.

function traiterPos(pos) {

alert('Coordonnées GPS: ' + pos.coords.latitude + ', ' + pos.coords.longitude);

}

function traiterErr(err) {

alert('Une erreur nous empêche de déterminer votre position: ' + err.message);

// exemple d'erreur: l'utilisateur n'a pas accepté de partager sa position

}

navigator.geolocation.getCurrentPosition(traiterPos, traiterErr);

À noter que certaines opérations asynchrones proposent une manière spécifique de gérer les erreurs. Par exemple, la classe XMLHttpRequest demande au développeur d’affecter une fonction à la propriété onerror de l’instance correspondante à une requête AJAX.

Il faut donc toujours penser à lire la documentation des fonctions et classes qu’on utilise dans son programme, afin de gérer les erreurs éventuelles.

**Nous arrivons a la fin de ce programme Javascript, soyez fier de vous, vous êtes désormais des développeurs et non plus des intégrateurs.**