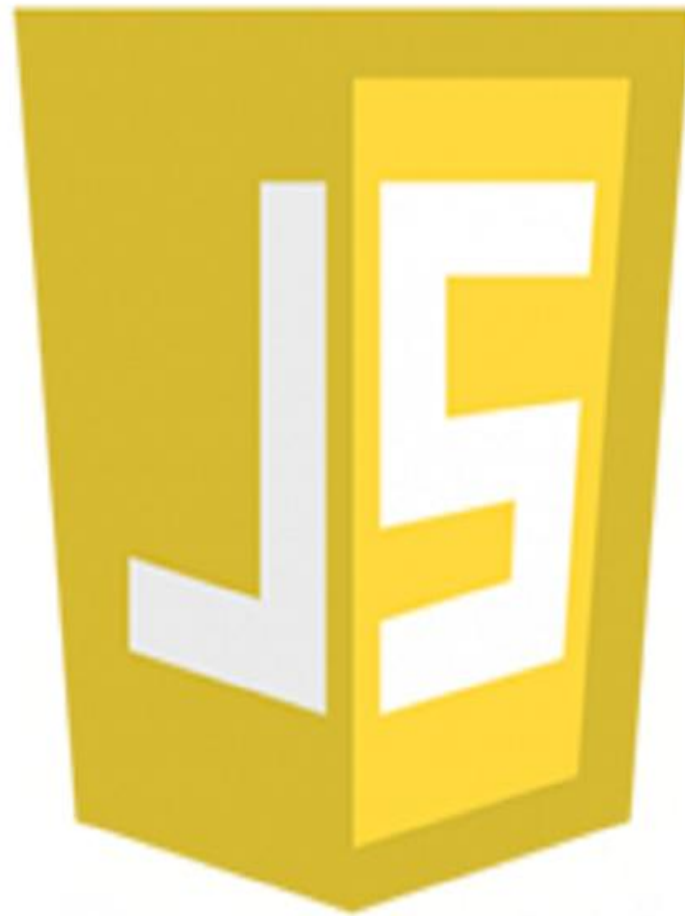


# Programmation Web Avancée

habib.aissaoua@univ-setif.dz

2021-2022



JavaScript

# JavaScript

- Le **JavaScript** est un langage de programmation:
  - Principalement « client-side ».
  - Interprété (le code est analysé et exécuté au fur et à mesure par le navigateur)
  - JavaScript est standardisé par Ecma International (ECMA étant historiquement un acronyme pour European Computer Manufacturers Association) qui délivre un langage de programmation standardisé, international appelé **ECMAScript**
- Supporté par les principaux navigateurs,
- C'est un langage de script léger, multi-plateforme
- Exécution de code du côté client (temps de réponse plus court)

Exemples : test d'un formulaire avant envoi, ...

# Variables et leurs portées

- Pour déclarer des variables locales ou globales on utilise le mot-clé **var** .

```
var person = "John Doe",  
    carName = "Volvo",  
    price = 200;
```

- Les variables sont typées dynamiquement, ce qui veut dire que l'on peut y mettre du texte en premier lieu puis l'effacer et y mettre un nombre.
- Lorsqu'une variable est déclarée avec **var** en dehors des fonctions, elle est appelée variable **globale** car elle est disponible pour tout le code contenu dans le document.
- Lorsqu'une variable est déclarée dans une fonction, elle est appelée variable **locale** car elle n'est disponible qu'au sein de cette fonction.

- la variable **name** est globale. Elle est visible aussi dans la fonction **showname()**. La deuxième déclaration de **name** est locale, elle est visible dans la fonction **showname()** mais elle n'est pas visible dans le contexte global.
- JavaScript ne prend pas en charge la portée de bloc, sauf dans le cas particulier de variables ayant une portée de bloc.

```
var name = "Richard";

function showName () {
  var name = "Jack"; // local variable; only accessible
  in this showName function
  console.log (name); // Jack
}

console.log (name); // Richard: the global variable
```

```
var a = 10;

if (a > 5) {
  var b = 5;
}

var c = a + b; // c is 15
```

Le mot-clé **let** permet de définir des variables au sein d'un bloc et des blocs qu'il contient. D'une certaine façon **let** fonctionne comme **var**, la seule différence dans cette analogie est que **let** fonctionne avec les portées de bloc et **var** avec les portées des fonctions.

```
function varTest() {  
  var x = 31;  
  if (true) {  
    var x = 71; // c'est la même variable !  
    console.log(x); // 71  
  }  
  console.log(x); // 71  
}
```

```
function letTest() {  
  let x = 31;  
  if (true) {  
    let x = 71; // c'est une variable différente  
    console.log(x); // 71  
  }  
  console.log(x); // 31  
}
```

- **const** permet de créer une constante qui peut être globale ou locale pour la fonction dans laquelle elle a été déclarée. Les constantes font partie de la portée du bloc (comme les variables définies avec `let`). Il est nécessaire d'initialiser une constante lors de sa déclaration. Au sein d'une même portée, il est impossible d'avoir une constante qui partage le même nom qu'une variable ou qu'une fonction.

```
// On définit ma_fav comme une constante
// et on lui affecte la valeur 7
// Généralement, par convention, les
// constantes sont en majuscules
const MA_FAV = 7;

// Cette réaffectation lèvera une exception TypeError
MA_FAV = 20;

// affichera 7
console.log("mon nombre favori est : " + MA_FAV);
```

```
// toute tentative de redéclaration renvoie une erreur
// SyntaxError: Identifier 'MY_FAV' has already been declared
const MA_FAV = 20;

// le nom ma_fav est réservé par la constante ci-dessus
// cette déclaration échouera donc également
var MA_FAV = 20;

// cela renvoie également une erreur
let MA_FAV = 20;
```



# Les objets:

- JavaScript possède des objets natifs, comme **String**, **Math**, etc., mais nous permet aussi de créer nos propres objets.

- **A partir de l'objet Object:**

```
var maVoiture = new Object();
```

```
maVoiture["fabricant"] = "Ford";
```

```
maVoiture.année = 1969;
```

```
maVoiture["demarrer"] = function(pos, vitesse) { alert("parametres: " + pos + ", " + vitesse); }
```

- **Avec la notation JSON :**

```
var obj = {
```

```
  attribut: "valeur",
```

```
  methode: function(parametre1, parametre2) { alert("parametres: " + parametre1 + ", " + parametre2);}
```

```
}
```

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
  </head>
  <body>
    <p id='p1'></p>
    <p id='p2'></p>
    <p id='p3'></p>
    <p id='p4'></p>
    <script>
      let dadi = {
        //nom, age et mail sont des propriétés de l'objet "dadi"
        nom : ['Dadi', 'samir'],
        age : 29,
        mail : 'dadi.samir@gmail.com',
        //Bonjour est une méthode de l'objet dadi
        bonjour: function(){
          return 'Bonjour, je suis ' + this.nom[0] +
            ', j\'ai ' + this.age + ' ans';
        }
      };
      document.getElementById('p1').innerHTML = 'Nom : ' + dadi.nom;
      document.getElementById('p2').innerHTML = 'Age : ' + dadi.age;
      document.getElementById('p3').innerHTML = dadi.bonjour();
    </script>
  </body>
</html>
```

# LES FONCTIONS

- Il y a deux moyens pour créer une fonction: les **expressions de fonction** et les **déclarations de fonction**.
  - Dans le cas d'une fonction déclarée, toute la fonction est chargée dans la mémoire du navigateur même si elle n'est pas utilisée immédiatement.
  - À la différence, les expressions de fonction (ou function expression) sont appelées quand l'interpréteur atteint cette ligne de code.

## - Déclaration de fonction:

```
function Identifiant ( FormalParameterListopt ){ FunctionBody }
```

## - Expression de fonction:

```
function Identifiantopt ( FormalParameterListopt ){ FunctionBody }
```

# DÉCLARATION DE FONCTION

- Le code suivant, par exemple, définit une fonction intitulée carré:

```
function carré(nombre) {  
    return nombre * nombre;  
}
```

- Les fonctions déclarées ne sont pas exécutées immédiatement. Ils sont "enregistrés pour une utilisation ultérieure", et seront exécutés plus tard, quand ils sont invoqués (appelés).
- Impossible de déclarer une fonction à une position attendant une *expression* ou dans une structure de contrôle if, for, etc.).

# EXPRESSION DE FONCTION

- La fonctionnalité principale de ce type de fonction est qu'elle se trouvera toujours dans une expression.

Enfermer le code de notre fonction dans une variable et utiliser la variable comme une fonction :

```
var doSomething = function () {  
    var a = 10;  
    var doSomethingElse = function () {  
        console.log(a); // a is 10  
    };  
    doSomethingElse();  
};  
  
doSomething();
```

- Cet exemple montre comment deux expressions de fonctions anonymes sont affectées aux variables

**doSomething** et **doSomethingElse**.

- Il est possible également pour les expressions de fonctions de posséder un nom mais celui-ci est **facultatif**.

## EXPRESSION DE FONCTION

```
var factorielle = function fac(n) { return n < 2 ? 1 : n * fac(n - 1) };  
  
console.log(factorielle(3));
```

## EXPRESSION DE FONCTION

Exo:

```
function map(f, a) {  
    var resultat = [], // Créer un nouveau tableau Array  
    i;  
    for (i = 0; i != a.length; i++)  
        resultat[i] = f(a[i]);  
    return resultat;  
}
```

```
var cube = function(x) { return x * x * x}; // Une expression de fonction  
map(cube, [0, 1, 2, 5, 10]);
```

Donner la valeur retournée par la fonction **map**?

# LES EXPRESSIONS DE FONCTION IMMÉDIATEMENT INVOQUÉES

## Immediately Invoked Function Expression (IIFE)

- Une autre façon d'exécuter une expression fonction est de créer une fonction anonyme qui va s'auto-invoquer c'est-à-dire qui va s'invoquer (ou s'appeler ou encore s'exécuter) elle-même dès sa création.
- Cette expression peut être anonyme ou nominative.
- Seront exécutés automatiquement si l'expression est suivie par () .
- Il faut ajouter l'opérateur de groupement ( ) pour indiquer qu'elle est une expression de fonction:

```
(function () {  
    var x = "Hello!!";           // I will invoke myself  
})();
```



# LES EXPRESSIONS DE FONCTION IMMÉDIATEMENT INVOQUÉES

## Immediately Invoked Function Expression (IIFE)

```
var a = "coucou";
```

```
var b = "monde";
```

```
// IIFE
```

```
(function(x, y) {  
    console.log(x + " " + y);
```

```
})(a, b);
```

```
// coucou monde
```

# LES EXPRESSIONS DE FONCTION IMMÉDIATEMENT INVOQUÉES

## Immediately Invoked Function Expression (IIFE)

// Ways to create IIFE

```
(function() {  
    alert("Brackets around the function");  
})();
```

```
(function() {  
    alert("Brackets around the whole thing");  
})();
```

```
!function() {  
    alert("Bitwise NOT operator starts the expression");  
}();
```

```
+function() {  
    alert("Unary plus starts the expression");  
}();
```

# Closures (Fermetures)

- En JavaScript, les fonctions imbriquées ont ainsi accès aux variables déclarées dans les portées parentes.

```
function init() {  
    var nom = "Mozilla"; // nom est une variable locale de init  
    function afficheNom() { // afficheNom est une fonction interne de init  
        console.log(nom); // ici nom est une variable libre (définie dans init)  
    }  
    afficheNom();  
};  
init();
```

- Une fermeture est une fonction interne qui permet d'accéder et manipuler des variables se trouvant en dehors de la portée de cette fonction (englobante)
- Donc, avec une fermeture, les variables de la fonction englobante sont toujours accessibles, même après la finalisation du contexte créé par cette dernière.

```
function creerFonction() {  
    var nom = "Mozilla";  
    function afficheNom() {  
        console.log(nom);  
    }  
    return afficheNom;  
}  
  
var maFonction = creerFonction();  
maFonction();
```

- La valeur de retour est une **fonction**
- **afficheNom** continue d'exister une fois que l'exécution de sa fonction englobante est terminée
- Ici **maFonction** est une fermeture qui contient la fonction **afficheNom** et une référence à la variable `var nom = "Mozilla"`.

# Closures (Fermetures)

- **ajout5** et **ajout10** partagent la même fonction, mais des environnements différents.

```
function faireAddition(x) {  
    return function(y) {  
        return x + y;  
    };  
};  
  
var ajout5 = faireAddition(5);  
var ajout10 = faireAddition(10);  
  
console.log(ajout5(2)); // 7  
console.log(ajout10(2)); // 12
```

## Exercice1:

Utiliser les fermeture et les IIFE pour écrire une fonction **uniqueId** qui renvoie un identifiant unique chaque fois qu'elle est appelée.

```
> console.log(uniqueId()); // "id_1"
```

```
console.log(uniqueId()); // "id_2"
```

```
console.log(uniqueId()); // "id_3"
```

## Exercise1:

```
const uniqueId = (function() {
```

```
  let count = 0;
```

```
  return function() {
```

```
    ++count;
```

```
    return 'id_' + count;
```

```
  };
```

```
})();
```

# Fonctions de rappel (callback)

- Une fonction callback est une fonction qui sera appelée (exécutée) après l'exécution de la fonction qui l'a reçu en tant que paramètre.

```
function test(fct_rappel) {  
    alert('callback reçu comme param');  
    fct_rappel(); // appel de la fonction  
}  
function Myfunction(){  
    alert('callback appelée');  
}  
test(Myfunction);  
/////
```

```
function doHomework(subject, callback) {  
    alert('Starting my ' + subject + ' homework.');
```

```
    callback();  
}
```

```
doHomework('math', function() {  
    alert('Finished my homework');
```

```
});
```



# Fonctions de rappel (callback)

- Quand on passe une fonction en paramètre, on donne juste son nom, sans les parenthèses

- Un autre exemple:

```
function f1(parm, MyCallback){  
    let x=parm;  
    MyCallback(x);  
}  
f1(3,function(x){  
    let y=0;  
    y=x+10;  
    console.log(y);  
});
```

- Dans jQuery, on utilise souvent les callback

```
$( '#MyP' ).hide(2000,  
    function(){  
        alert("callback function");  
    }  
);
```

# Fonctions fléchées (arrow functions)

- Les fonctions **fléchées** sont une manière beaucoup plus concise et courte de définir des fonctions en Javascript. Leur particularité c'est qu'on utilise une flèche => pour définir une fonction et on n'utilise plus le mot clé **function**.

**Syntaxe:** **([param] [, param]) => { instructions }**

// Parenthèses non nécessaires quand il n'y a qu'un seul argument

**param => expression**

// Une fonction sans paramètre peut s'écrire avec un couple de parenthèses

**() => { instructions }**

```
let sum = (a, b) => a + b;

/* Cette fonction fléchée est la forme raccourcie de :

let sum = function(a, b) {
  return a + b;
};
*/

alert( sum(1, 2) ); // 3
```

Comme vous pouvez le voir  $(a, b) \Rightarrow a + b$  représente une fonction qui accepte 2 arguments nommés a et b. Lors de l'exécution, elle évalue l'expression  $a + b$  et retourne le résultat.

```
let double = n => n * 2;  
// Similaire à : let double = function(n) { return n * 2 }
```

```
console.log(double(3)); // 6
```

*// Sans arguments, les parenthèses seront alors vides (mais elles doivent être présentes) :*

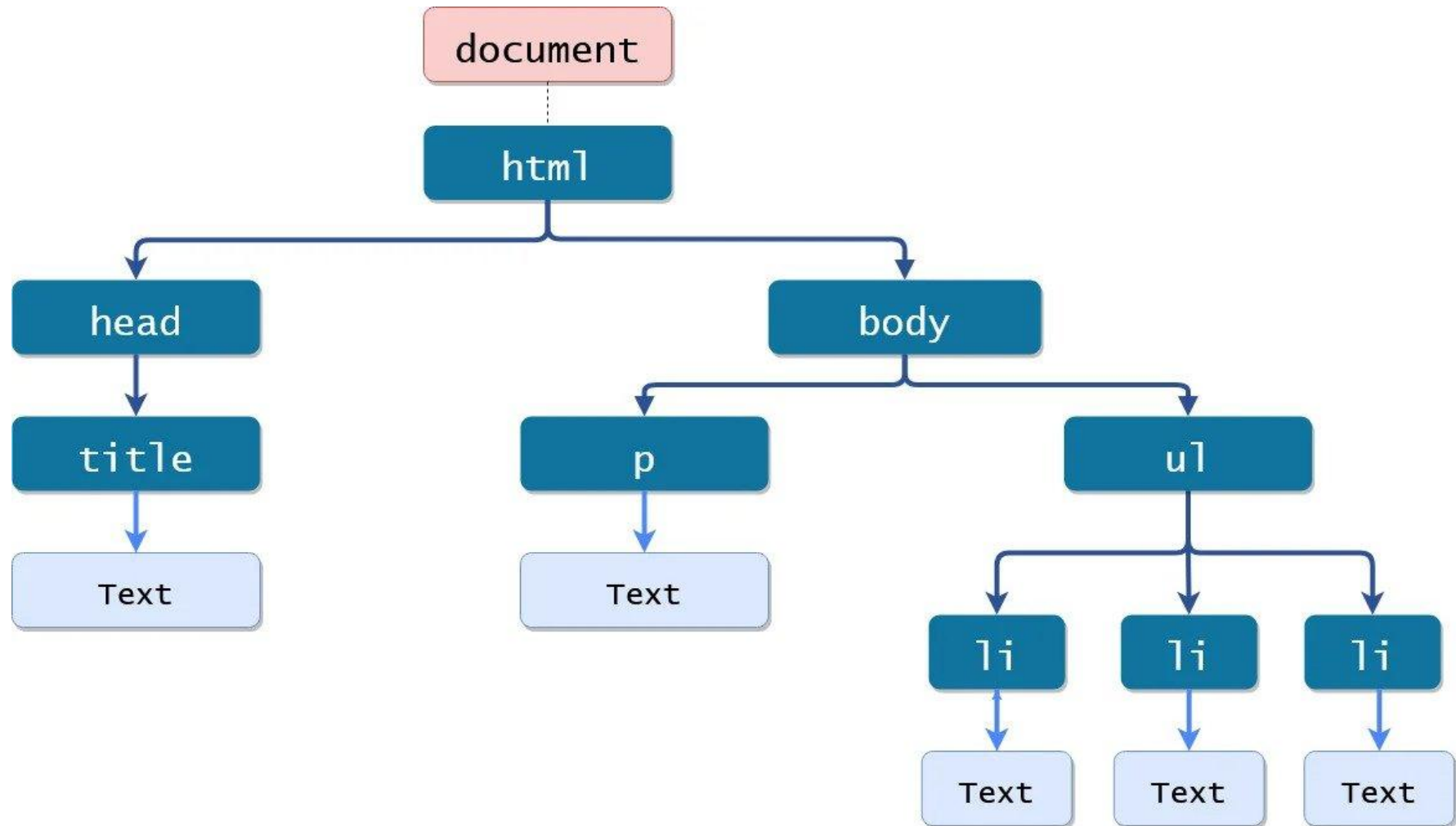
```
let sayHi = () => console.log("Hello!");
```

```
sayHi();
```

# Document Object Model DOM

- Pour manipuler un code HTML dynamiquement, nous allons avoir besoin d'une interface qui va nous permettre d'accéder à ce code HTML.
- L'interface de programmation que nous allons utiliser est **DOM**
- Avec **DOM**, une page Web est schématisée comme une arborescence hiérarchisée. Les différentes composantes d'une telle arborescence sont désignés comme étant des **nœuds**
- Indépendant du langage de programmation et de la plate-forme.

- L'objet central du modèle DOM est l'objet **node** (node = nœud). Il existe trois types de nœud importants : les nœuds: **élément**, **attribut** et **texte**.



Pour manipuler les éléments de la page il faut au préalable les sélectionner. La sélection d'éléments peut se faire par:

- La méthode `getElementById` de l'objet `document` sélectionne l'unique élément du document dont l'`id` est fourni en paramètre, ou `null` si aucun.

```
<div id="myid"> ... </div>
```

```
var element = document.getElementById ("myid");
```

- la méthode `getElementsByTagName` sélectionne les éléments dont la balise est fournie en paramètre

```
                // tous les <div> du document
var divList = document.getElementsByTagName("div");
divList.length;    // -> le nombre d'éléments sélectionnés
                // toutes les <img> descendants de 'sec1'
var sec1 = document.getElementById("section1");
var sec1DivList = sec1.getElementsByTagName("img");
sec1DivList[0];    // -> le premier élément sélectionné
```



# Manipuler le contenu

## ➤ **innerHTML**

la propriété **innerHTML** représente le contenu HTML d'un élément

- En lecture, sa valeur **contient** les balises
- En modification, son contenu **est interprète** par le navigateur

## ➤ **textContent**

la propriété **textContent** représente le contenu textuel d'un élément

- En lecture, sa valeur **ne contient pas** les balises HTML
- En modification, son contenu **n'est pas interprété** par le navigateur

```

...
var element = document.getElementById("exemple"); <div id="exemple">
var htmlText = element.innerHTML;                <p>Ceci est
alert(htmlText); // -> "<p> Ceci est <strong>        <strong>mon
                //   mon</strong> contenu.</p>"      </strong>
                                                    contenu.
var txt = element.textContent;                    </p>
alert(txt); // -> "Ceci est mon contenu"          </div>
...

```

```

// modifie le contenu HTML de l'élément et donc son 'affichage'.
//   La balise <em> est interprétée.
elemt.innerHTML = "un <em>autre</em> contenu";

// modifie le contenu texte de l'élément et donc son 'affichage.'
// Le texte <strong> N'est PAS interprété.
elemt.textContent = "un contenu <strong>texte</strong>";

```

## Naviguer entre les nœuds

- Une fois que l'on a atteint un élément, il est possible de se déplacer de façon un peu plus précise, avec toute une série de méthodes et de propriétés
- La propriété **parentNode** permet d'accéder à l'élément parent d'un élément
- **firstChild** et **lastChild** servent respectivement à accéder au premier et au dernier enfant d'un nœud.
- **nextSibling** et **previousSibling** sont deux propriétés qui permettent d'accéder respectivement au nœud suivant et au nœud précédent
- **childNodes** renvoie la liste des nœuds enfants d'un élément donné. Le premier nœud enfant a l'indice 0.

## Naviguer entre les nœuds

```
var theDiv = document.getElementsByTagName('div')[0];
```

```
<div>
```

```
var p = theDiv.firstChild;
```

```
<p> This is text </p>
```

```
var ul = p.nextSibling;
```

```
<ul>
```

```
<li>Apple</li> ul.childNodes[0]
```

```
<li>Pear</li> ul.childNodes[1]
```

```
<li>Melon</li> ul.childNodes[2]
```

```
</ul>
```

```
</div>
```

## Créer et insérer des éléments

Avec le DOM, l'ajout d'un élément HTML se fait en trois temps :

1. On crée l'élément ;
  2. On lui affecte des attributs ;
  3. On l'insère dans le document, et ce n'est qu'à ce moment-là qu'il sera « ajouté ».
- La création d'un élément se fait avec la méthode **createElement()**, un sous-objet de l'objet racine, c'est-à-dire **document** dans la majorité des cas.

```
let newLink = document.createElement('a');
```

Cet élément est créé, mais n'est pas inséré dans le document, il n'est donc pas visible.

## Créer et insérer des éléments

- On définit les attributs, soit avec **setAttribute()**, soit directement avec les propriétés adéquates.

```
newLink.id= 'MyID' ;
```

```
newLink.setAttribute('href', 'http://www.w3schools.com/ ');
```

- On utilise la méthode **appendChild()** pour insérer l'élément

Donc, pour ajouter notre élément `<a>` dans l'élément `<p>` portant l'ID `myP`, il suffit de récupérer cet élément, et d'ajouter notre élément `<a>` via **appendChild()**:

```
document.getElementById('myP').appendChild(newLink);
```

## Créer et insérer des éléments

**appendChild()** insère toujours l'élément en tant que dernier enfant.

- L'élément a été inséré, seulement il manque quelque chose : le contenu textuel ! La méthode **createTextNode()** sert à créer un nœud textuel

```
let newLinkText = document.createTextNode("Mon site");  
newLink.appendChild(newLinkText);
```

## Naviguer entre les nœuds

```
<body>
  <p id="myP"> </p>

  <script type="text/javascript">
    let newLink = document.createElement('a');
    newLink.id= 'MyID';
    newLink.setAttribute('href', 'http://www.w3schools.com/ ');
    let newLinkText = document.createTextNode("Mon site");
    newLink.appendChild(newLinkText);
    document.getElementById('myP').appendChild(newLink);
  </script>
</body>
```



# ÉVÉNEMENTS

- Les événements correspondent à des actions effectuées soit par un utilisateur, soit par le navigateur lui-même.
- Pour réagir aux événements, nous pouvons assigner un gestionnaire (associer une fonction qui s'exécute lorsque l'événement se déclenche
- La première méthode consiste à utiliser des **attributs** HTML de « type » événement, et ensuite il faut créer le code correspondant à l'action que l'on souhaite attacher à notre événement

Nom de l'événement	Action pour le déclencher
<code>click</code>	Cliquer (appuyer puis relâcher) sur l'élément
<code>dblclick</code>	Double-cliquer sur l'élément
<code>mouseover</code>	Faire entrer le curseur sur l'élément
<code>mouseout</code>	Faire sortir le curseur de l'élément
<code>mousedown</code>	Appuyer (sans relâcher) sur le bouton gauche de la souris sur l'élément
<code>mouseup</code>	Relâcher le bouton gauche de la souris sur l'élément
<code>mousemove</code>	Faire déplacer le curseur sur l'élément

Le script suivant utilise l'attribut **onclick** dans une balise HTML.

```
1 | <input type="button" value="Cliquez ici" onclick="alert('toto')">
```

L'inconvénient de cette méthode est que le javascript se retrouve mélangé avec le HTML


2. La deuxième méthode consiste à utiliser **DOM** HTML. Nous avons deux manières de réagir aux évènements:

- DOM-0: On utilise des propriétés qui vont assigner un gestionnaire d'événement à un élément spécifique en HTML,
- DOM-2: utiliser la méthode **addEventListener()**.

## 1. DOM-0:

Nous pouvons assigner un gestionnaire en utilisant une propriété DOM avec: **on<event>**

```
1 <input id="elem" type="button" value="Click me">
2 <script>
3   elem.onclick = function() {
4     alert('Thank you');
5   };
6 </script>
```



Nous pouvons aussi assigner directement une fonction en tant que gestionnaire:

```
1 function sayThanks() {
2   alert('Thanks!');
3 }
4
5 elem.onclick = sayThanks;
```

Attention: - La fonction doit être assignée comme : **sayThanks** n'est pas **sayThanks()**

- Utiliser **elem.onclick** n'est pas **elem.ONCLICK**

Le **DOM-0** ne permet pas d'assigner plusieurs fois le même gestionnaire d'événement à un élément HTML. Seul l'action du dernier événement sera déclenchée, comme dans l'exemple ci-dessous

```
1 input.onclick = function() { alert(1); }  
2 // ...  
3 input.onclick = function() { alert(2); } // replaces the previous handler
```

1. **DOM-2**: Le modèle d'événement DOM niveau 2 définit deux méthodes de l'objet **Element**, **addEventListener** et **removeEventListener**, qui permettent d'attacher ou de détacher un gestionnaire d'événement.

Syntaxe:

```
1 element.addEventListener(event, handler[, phase]);
```

```
1 // exactly the same arguments as addEventListener  
2 element.removeEventListener(event, handler[, phase]);
```

Les deux méthodes peuvent contenir trois paramètres :

- **event**: une chaîne de caractères désignant le type d'événement sans le "on"  
→ "click", "load", "change", "mouseover", "keypress" etc.
- **handler**: la fonction qui sera appelée lorsque l'événement se produit
- **phase**: un booléen pour gérer la propagation. **false** par défaut.

**Exemple:**

```
element.addEventListener("click", myFunction);
```

```
function myFunction() {  
    alert ("Hello World!");  
}
```

**Ou:**

```
element.addEventListener("click", function(){ alert("Hello World!"); });
```

Pour détacher un gestionnaire d'événement, il faut passer à la méthode `removeEventListener` les mêmes paramètres passés à `addEventListener`. Par exemple

```
1 function handler() {  
2   alert( 'Thanks!' );  
3 }  
4  
5 input.addEventListener("click", handler);  
6 // ....  
7 input.removeEventListener("click", handler);
```

Dans l'exemple ci-dessous, le gestionnaire sera supprimé ou non?

```
1 elem.addEventListener( "click" , () => alert('Thanks!'));  
2 // ....  
3 elem.removeEventListener( "click", () => alert('Thanks!'));
```

Ici, le gestionnaire ne sera pas supprimé, car `removeEventListener` est lié à une autre fonction (même code, mais cela n'a pas d'importance.)

L'un des grands avantages de la méthode `addEventListener()` est de pouvoir lier plusieurs gestionnaires d'évènements de même type sur un élément HTML.

```
1 <input id="elem" type="button" value="Click me"/>
2
3 <script>
4   function handler1() {
5     alert('Thanks!');
6   };
7
8   function handler2() {
9     alert('Thanks again!');
10  }
11
12  elem.onclick = () => alert("Hello");
13  elem.addEventListener("click", handler1); // Thanks!
14  elem.addEventListener("click", handler2); // Thanks again!
15 </script>
```