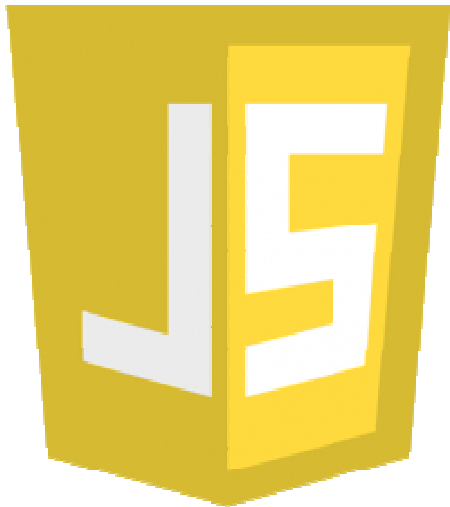


# Technologies du web



# Partie 3 : JavaScript



JavaScript



# Plan de la présentation

**Introduction à JavaScript**

**Core JavaScript**

**Les Objets dans JavaScript**

**Client-side JavaScript**

# Introduction

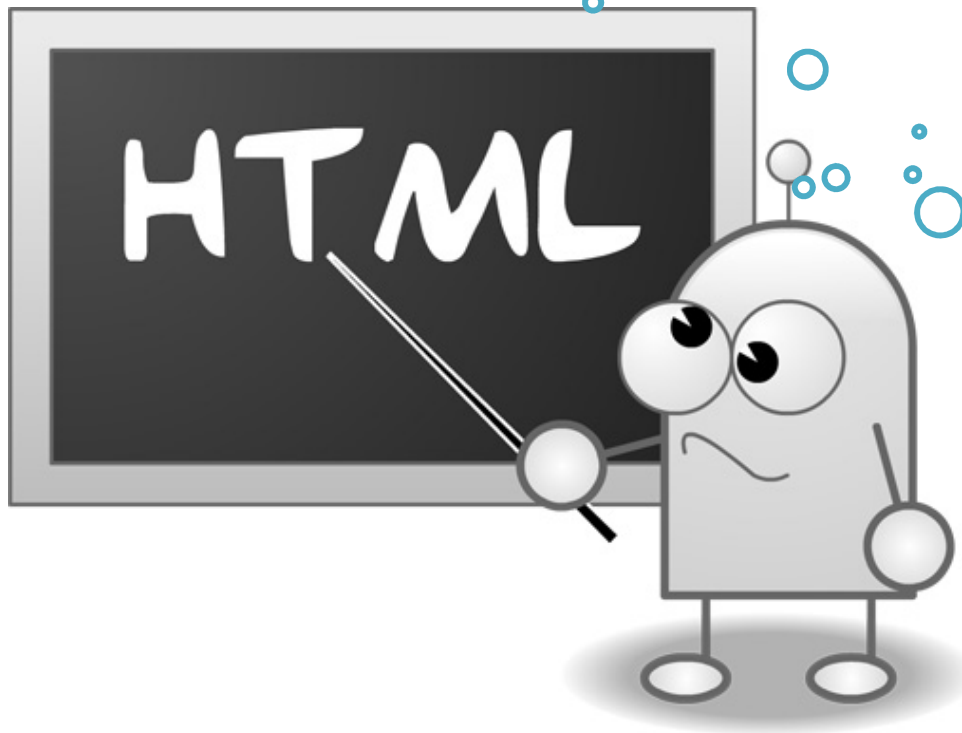
Manipuler les  
événements

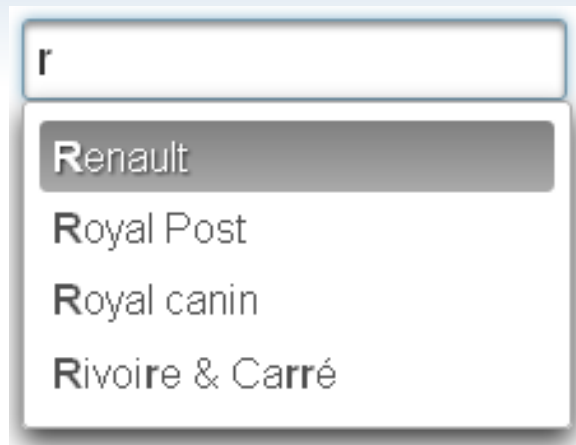
Interagir  
avec les  
éléments  
HTML

Valider les formulaires  
web côté client

*Drag & Drop*

Faire un peu de dessin  
et des animations, et  
bien d'autres choses !





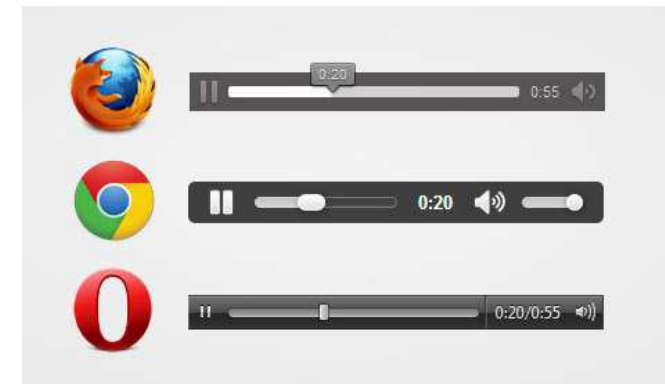
Une suggestion lors de la frappe dans un champ de texte, comme lors d'une recherche avec Google ;



Un système de chat, comme celui de Facebook



Des jeux exploitants la balise <canvas> :  
Torus; Tetris en 3D ; modélisation 3D d'une Lamborghini affichée grâce à l'API WebGL et à la bibliothèque Three.js



Un lecteur de vidéos ou de musiques grâce aux balises <video> et <audio>

- JavaScript C'est quoi ???!!
- Compilé vs Interprété
- L'usage de JavaScript
- Les particularités de JavaScript
- Historique de JavaScript

# Un peu d'histoire...

- Netscape Communications a eu besoin d'un langage coller au HTML afin de rendre les pages un peu plus dynamique
- Ce langage doit être:
  - Facile utiliser par les designers et les programmeurs
  - Capable d'assembler les composants (images, plugins,..)
- Brendan Eich, a développé le premier prototype en 10 jours

- Le premier nom de ce langage a été **Mocha**
- Ensuite **LiveScript**
- Il a été renommé « **JavaScript** » lorsque sa version Beta 3 a été déployé
  - Aucun lien avec JAVA
  - Choix de Marketing
- Normalisé par *Ecma International* en 1996 il a été baptisé **ECMAScript**



# JavaScript C'est quoi ???!!

- JavaScript est un langage de programmation qui sert principalement à dynamiser et à rendre interactifs les pages WEB.
- Exécuté sur vos machine donc côté Client.
- L'un des langages WEB les plus utiles et membre du fameux triplet :
  - HTML pour le contenu
  - CSS pour la présentation
  - JS pour le comportement
- JavaScript est:
  - Haut niveau
  - Dynamique
  - Non typé
  - Interprété

# Compilé vs Interprété

## ➤ Compilation

- Un compilateur va traduire le programme dans un code spécifique à la machine cible (l'exécutable).
- Etapes :
  - Le code source est analysé et parsé : C'est là ou on détecte les erreurs.
  - Un générateur de code produit l'exécutable.
  - Le code est ensuite exécuté

## ➤ Interprétation

- Le code source n'est pas directement exécuté par la machine cible.
- Un autre programme (i.e. l'interpréteur) lit et exécute le code source.
- Dans l'interprétation le code source est traduit en un code intermédiaire qui est ensuite traduit en un code machine.

- Les bases du JavaScript
  - Les variables
  - Les types de données
  - Opérateurs et Expressions
  - Structures conditionnelles et itératives
- Les Fonctions
- Les Tableaux
- Les Objets

- JS est sensible à la casse
  - Les tags html et ces attributs doivent être représenté en minuscule coté JS.
- JS ignore les espaces et les retours à la ligne.
  - Cependant formater votre code c'est UN DEVOIR
- Les commentaires
  - `//` : commente le texte jusqu'à la fin de la ligne
  - `/* text */` : Commente un bloc

## ➤ Identifiants

- Doit commencer par une lettre, par (\_) ou par le signe dollar (\$)
- Le reste des lettres, chiffres, (\_), ou \$

## ➤ Mots clé réservés

- Certain identifiants sont réservés comme mot clé

break	delete	function	return
	typeof		
case	do	if	switch
catch	else	in	this
continue	false	instanceof	throw
debugger	finally	new	true
default	for	null	try
			var
			void
			while
			with

- Certains sont moins utilisés mais peuvent l'être plus dans les futures versions

class	const	enum	export	extends	import	super
-------	-------	------	--------	---------	--------	-------

- Dans JS l'heure et le temps sont gérés par l'objet Date.
- Stocké sous forme de timestamp (nombre de millisecondes écoulées depuis le 1<sup>er</sup> Janvier 1970).

```

var maDate = new Date(2010, 0, 1);           // Premier jour du premier mois
                                              // de 2010

var apres = new Date(2010, 0, 1,            // Même jour, à 5:10:30pm, heure
17, 10, 30);                                locale

var mnt = new Date();                       // Date et heure actuelles
var elapsed = now - then;                   // La valeur de retour représente
                                              la différence en millisecondes

apres.getFullYear()                         // => 2010
apres.getMonth()                           // => 0: (0:11)
apres.getDate()                            // => Jour du mois 1: (1:31)
apres.getDay()                             // => Jour de la semaine (0:6)
                                              5: zero-based jour
                                              0 représente dimanche.

apres.getHours()                           // => Heure (0:23) 17
apres.getMinutes()                         // (0:59)
apres.getSeconds()                         // (0:59)
apres.getMilliseconds()                    // (0:999)
apres.getTime()                            // retourne le timestamp de l'objet
apres.setTime(timestamp)                   // modifie l'objet selon le timestamp

```

# Dates et Times

- **setTimeout**( `execFunction`, `waitingTime`, `param1`,`param2`,...,`paramN`)  
est une fonction qui permet d'exécuter la fonction `execFunction` après `waitingTime` milliseconde de son appel. Les paramètres `param1`, `paramN` sont les paramètres que nous voulons passer à `execFunction` (ne marche pas dans les internet explorer <10). Cette fonction retourne un id qui identifie la fonction pour pouvoir l'annuler avec la fonction `clearTimeout`.
- **setInterval**( `execFunction`, `intervalTime`, `param1`,`param2`,...,`paramN`)  
est une fonction qui permet d'exécuter la fonction `execFunction` tout les `intervalTime` milliseconde à partir de son appel. Les paramètres `param1`, `paramN` sont les paramètres que nous voulons passer à `execFunction` (ne marche pas dans les internet explorer <10).
- Si La version est inférieur à explorer 10 il suffit d'utiliser une **fonction anonyme** et dans laquelle on appellera la fonction `execFunction`. Cette fonction retourne un id qui identifie la fonction pour pouvoir l'annuler avec la fonction `clearInterval`.





```
function testTimeout() {
    setTimeout(function () {
        console.log('Je test le Timeout');
    }, 2000);
}
testTimeout(); var i=0;
testInterval();
function testInterval() {
    var code = setInterval(function () {
        if (i < 22)
        {
            console.log(i);
            i++;
        }
        else {
            clearInterval(code);
        }
    }, 100, code);
}
```



# Chaines

- Les chaines en JS peuvent être mises entre ' ou «
- Il y a deux types de chaines de caractère, le type primitif et les strings
- `var primitifChaine = « bjr »; var varString = new String(« bjr »);`
- En ECMAScript 5, les chaines peuvent être représentées sur plusieurs lignes en terminant chaque ligne par \
- Pour échapper un caractère spécial on utilise \
  - `\n, \' , \t, \\, ...`
- La concatenation se fait avec +
- Pour avoir le nombre de caractère on utilise l'attribut `length`.
- `chaine.length`

# Chaines

<code>var s = "hello, world"</code>	on déclare une chaine
<code>s.charAt(0)</code>	"h": premier caractère
<code>s.charAt(s.length-1)</code>	"d": le dernier caractère
<code>s.substring(1,4)</code>	"ell": les 2ème 3ème et 4ème caractère.
<code>s.slice(1,4)</code>	"ell": même résultat
<code>s.slice(-3)</code>	"rld": le deuxième paramètre peut être négatif ce qui indique qu'on commence par la fin.
<code>s.indexOf("l")</code>	2: position de la première occurrence de l, -1 si la chaine ou le caractère n'existe pas.
<code>s.lastIndexOf("l")</code>	10: position du dernier caractère ou chaine.
<code>s.indexOf("l", 3)</code>	3: position of first "l" at or after 3
<code>s[0]</code>	"h"
<code>s[s.length-1]</code>	"d"
<code>s.trim()</code>	supprime les blanc à gauche et à droite

# Les booléens

- Deux valeurs `true` et `false`
- Les valeurs évaluées à `false` sont :
  - `undefined`, `null`, `0`, `-0`, `NaN`, `""`
  - Tous le reste est évalué à `true`
- Pour convertir un booléen vers un string on peut utiliser le `toString()`
- Les opérations sur les booléens
  - `&&` : et
  - `||` : ou
  - `!` : NOT

# La conversion de type

- Deux types de conversion : implicite et explicite
- Conversion implicite
  - Lorsque JS s'attend à avoir un type particulier, il le convertit automatiquement
  - Pour les numériques, si la conversion n'est pas possible, la variable est convertit en NaN.

<code>Number("3")</code>	<code>// =&gt; 3</code>
<code>String(false)</code>	<code>// =&gt; "false" Or use</code>
<code>false.toString()</code>	
<code>Boolean([])</code>	<code>// =&gt; true</code>
<code>Object(3)</code>	<code>// =&gt; new Number(3)</code>
<code>x + ""</code>	<code>// Même chose que String(x)</code>
<code>+x</code>	<code>// Même chose que Number(x).</code>



- Conversion explicite
  - Utilisation de : `Boolean()`, `Number()`, `String()`, `Object()`
  - Utilisation de +

<code>10 + " objets"</code>	<code>// =&gt; "10 objets". 10 est converti en une chaine</code>
<code>"7" * "4"</code>	<code>// =&gt; 28: Les chaines sont convertis en un entier</code>
<code>var n = 1 - "x";</code>	<code>// =&gt; NaN: x ne peut pas être convertit en un entier</code>
<code>n + " objets"</code>	<code>// =&gt; "NaN objets": NaN est convertit en la chaine"NaN"</code>

# Les variables

- Les variables sont déclarées avec le mot clé **var**.
- Si une variable n'est pas initialisée elle aura la valeur **undefined**.
- Une variable déclarée **sans** utiliser le mot **var** est une **variable globale**.
- Les variables sont **non typées**, le type est associé lors de l'exécution et il **peut changer**.

# La portée des variables

- La portée (scope) d'une variable est l'emplacement dans lequel elle est définie.
- Variables globales : ce sont les variables définies dans tout le script JS.
- Variables locales
  - Déclarées dans le corps d'une fonction
  - Définit dans une fonction ou dans n'importe quelle fonction définie à l'intérieur de celle-ci.
  - Vous devez toujours utiliser **var** pour les variables locales.
- JS (avant la version de ES6) n'a **pas** de **portée de block**.
- Avec var si vous définissez la variable dans une fonction, elle sera visible à l'intérieur de celle-ci.
- Si vous définissez var dans le script, la variable est créée comme une propriété à l'objet global (**window** dans un navigateur, **global** ou **module** dans NodeJs).



# La portée des variables

- La portée (scope) d'une variable est l'emplacement dans lequel elle est définie.
- Variables globales : ce sont les variables définies dans tout le script JS.
- Variables locales
  - Déclarées dans le corps d'une fonction

```
function test(o) {
    var i = 0;
    if (typeof o == "object") {
        var j = 0;
        for (var k=0; k < 10; k++) {
            console.log(k);
        }
        console.log(k);
    }
    console.log(j);
}
```

// i est défini dans la fonction

// j est défini dans la fonction pas que dans la boucle

// Afficher les nombres de 0 à 9

// k est toujours défini

// j est toujours défini mais elle peut ne pas être initialisé





- Les variables locales sont visibles dans la fonction même avant leur déclaration.
- Ce processus est appelé le *hoisting*
  - Toutes les déclarations de variables sont « hoisté » (mis en haut). Ce n'est pas le cas des initialisations.

```
var scope = "globale";
function f() {
    console.log(scope);
    var scope = "locale";

    console.log(scope);
}
```

```
// "undefined"
// Variable initialisée ici, mais
// définie en haut dû au hoisting
// affiche "locale"
```



# La portée des variables

- Les variables locales sont visibles dans la fonction même avant leur déclaration.
- Ce processus est appelé le *hoisting*
  - Toutes les déclarations de variables sont « hoisté » (mis en haut). Ce n'est pas le cas des initialisations.

```
var scope = "globale";
function f() {
    console.log(scope);
    var scope = "locale";

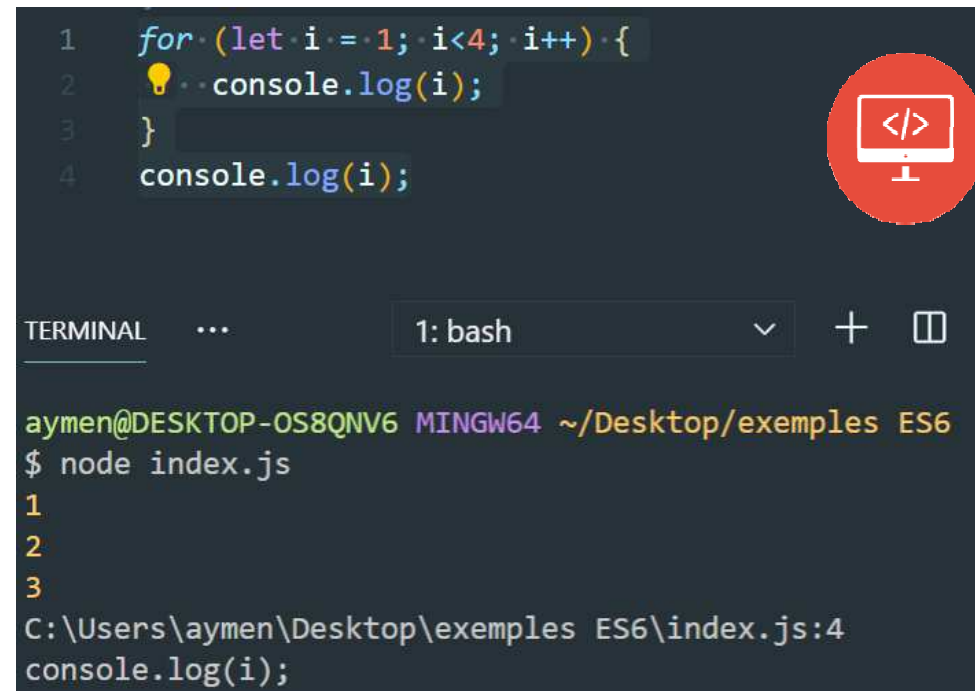
    console.log(scope);
}
```

```
// "undefined"
// Variable initialisée ici, mais
// définie en haut dû au hoisting
// affiche "locale"
```



# La portée des variables let

- A partir de ES6 deux nouvelles portées ont été introduites :
  - `let`
  - `const`
- `let` vous permet de déclarer la variable dans le bloc courant.
- Dans le cadre d'une déclaration hors d'un bloc, c'est-à-dire dans le contexte global, `let` crée une vraie variable globale (là où `var` créait une propriété de l'objet global)
- Avec `let` on ne parle plus de Hoisting.
- Si vous définissez donc une variable dans une boucle elle n'est visible que dans la boucle.



```

1  for (let i = 1; i < 4; i++) {
2    console.log(i);
3  }
4  console.log(i);

```

TERMINAL ... 1: bash

```

aymen@DESKTOP-OS8QNV6 MINGW64 ~/Desktop/exemples ES6
$ node index.js
1
2
3
C:\Users\aymen\Desktop\exemples ES6\index.js:4
console.log(i);

```

# La portée des variables const

- Pour définir une constante, il faut utiliser la nouvelle portée const.
- Une fois déclarée, cette variable n'est accessible qu'en lecture.
- const a les mêmes règles de portée que let
- Lorsque vous définissez une variable en tant qu'objet vous obtenez la référence. Ceci implique que si c'est une constante c'est la référence qui doit rester constante et non l'objet sur laquelle elle réfère.



- Un tableau est une suite de valeurs de différents types.
- Pour initialiser un tableau on a plusieurs méthodes :
  - `monArray = [val1, val2, ,valN];`
  - `monArray = new Array();`

```
monArray = [ ];
monArray = [1+2,3+4];    // 3, 7
var matrice = [[1,2,3], [4,5,6], [7,8,9]];
var eparsedArray = [1,,5]; // eparsedArray[1]==undefined;
var t = new Array(1,2,3) ⇔ var t=[1,2,3]
```



# Les tableaux

- Lire un élément du tableau
  - Avec l'opérateur `[]`
- Un tableau peut être tronqué en modifiant l'attribut `length`
- Ajout et suppression d'éléments
  - `push()`: ajoute une valeur à la fin du tableau
  - `unshift()`: ajoute une valeur au début du tableau
  - `delete`: supprime un élément
- Itérer sur un tableau
  - for ou for/in

```
var a = ["world"];
var value = a[0];
a[1] = 3.14;
i = 2;
a[i] = 3;
a[i + 1] = "hello";
a[a[i]] = a[0];
a.length // => 4
a.length = 2 // a devient ["world",3.14]
// Ajout
a.push("zero"); // a[2]=="zero"
a.push(1,3); // a[3]==1, a[4]==3
// Suppression
a = [1,2,3];
delete a[1]; // a n'a plus d'élément l'indice 1
1 in a // => false: pas d'indice 1
a.length // => 3: delete n'affecte pas la taille du tableau
// Itération
for (var indice in a) {
    var value = a[indice];
    // Faites ce que vous voulez
}
```



# Les tableaux

- Les méthodes sur les tableaux :
  - la méthode `concat` permet de concaténer 2 tableaux et de retourner le tableau résultat.
  - la méthode `forEach` permet de parcourir le tableau. Elle prend en `paramètre` la `fonction` à exécuter à qui elle `fourni` la `valeur actuelle du tableau` son `indice` et le `tableau en question`.
  - Les méthodes de recherche sont les mêmes que les chaines mais d'une façon générique vu que le tableau contient des éléments de différents types. Pour rappel les méthodes sont `indexOf( )` et `lastIndexOf( )`.

```
var tab1=[1,2,3];
var tab2 = [4,5,6];
var tab3 = tab1.concat(tab2);
tab3.forEach(function(val,ind,monTab){
  console.log("tab["+ind+"]="+val);
});
```



# Les tableaux

- Afin de **trier** un tableau nous utilisons la méthode **sort( )** qui transforme les éléments du tableau en **chaîne de caractère** et effectue le tri par **ordre alphabétique**.
- La fonction **sort** prend un **paramètre facultatif** qui est une **fonction** qui spécifie l'ordre de tri. Cette fonction devra retourner un entier (-1 si le 1<sup>er</sup> est inférieur au second, 0 si égaux et 1 sinon).
- Pour extraire une portion d'un tableau on utilise la même fonction que les chaînes **slice**.
- Afin de **modifier une partie du tableau** on utilise la méthode **splice(indice, nbElement, var1, var2,..., varN)** qui extrait les **nbElement** à partir de l'indice **indice** et les remplace par les variables **varn**). Seul les deux premiers paramètres sont obligatoires.





# Les tableaux

*// Ceci marche du à la conversion implicite de JS  
(mais ca ne marche pas en cas de undefined et de  
NaN)*

```
function comparerDesNombres(a, b) {
    return a - b;
}

function comparerDesNombres(a,b) {
    var x=parseInt(a,10);var y=parseInt(b,10);
if (x>y) {
        return 1;
    } else if (y>x) {
        return -1;
    } return 0;
}

var t=[1,2,3,11,22,33];
t.sort(comparerDesNombres);
```



# Les tableaux

Plusieurs méthodes sont offertes avec les tableaux afin de faciliter leur utilisation.

- **concat()** cette méthode renvoie un nouveau tableau constitué de ce tableau concaténé avec un ou plusieurs autre(s) tableau(x) et/ou valeur(s).
- **includes()** cette méthode détermine si le tableau **contient ou non un certain élément à partir d'une position (optionnelle)**. Elle renvoie true ou false selon le cas de figure.
- **indexOf()** cette méthode retourne le premier (plus petit) index d'un élément égal à la valeur passée en paramètre à l'intérieur du tableau, ou -1 si aucun n'a été trouvé.
- **join()** cette méthode concatène tous les éléments d'un tableau en une chaîne de caractères.
- **lastIndexOf()** cette méthode retourne le dernier (plus grand) index d'un élément égal à la valeur passée en paramètre à l'intérieur du tableau, ou -1 si aucun n'a été trouvé.
- **slice(start, end)** cette méthode extrait une portion d'un tableau pour retourner un nouveau tableau constitué de ces éléments.

- La méthode **Array.from()** permet de créer une nouvelle instance d'Array (une copie superficielle) à partir d'un objet itérable ou semblable à un tableau.

```
console.log(Array.from('foo'));
// expected output: Array ["f", "o", "o"]

console.log(Array.from([1, 2, 3], x => x + x));
// expected output: Array [2, 4, 6]
```



- La méthode **every()** permet de tester si tous les éléments d'un tableau vérifient une condition donnée par une fonction en argument. Cette méthode renvoie un booléen pour le résultat du test..

```
const isBelowThreshold = (currentValue) => currentValue < 40;

const array1 = [1, 30, 39, 29, 10, 13];

console.log(array1.every(isBelowThreshold));
// expected output: true
```



- La méthode **filter()** crée et retourne un nouveau tableau contenant tous les éléments du tableau d'origine qui remplissent une condition déterminée par la fonction callback.

```
const words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];

const result = words.filter(word => word.length > 6);

console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```



- La méthode **find()** renvoie la valeur du premier élément trouvé dans le tableau qui respecte la condition donnée par la fonction de test passée en argument. Sinon, la valeur `undefined` est renvoyée.

```
const array1 = [5, 12, 8, 130, 44];

const found = array1.find(element => element > 10);

console.log(found);
// expected output: 12
```



- La méthode **map()** crée un nouveau tableau avec les résultats de l'appel d'une fonction fournie sur chaque élément du tableau appelant.

```
const array1 = [1, 4, 9, 16];

// pass a function to map
const map1 = array1.map(x => x * 2);


console.log(map1);
// expected output: Array [2, 8, 18, 32]
```



- La méthode **reduce()** et **reduceRight()** applique une fonction sur un accumulateur et sur chaque valeur du tableau (de gauche à droite et de droite à gauche) de façon à obtenir une unique valeur à la fin.

```
const array1 = [1, 30, 39, 29, 10, 13];

console.log(
  array1.reduce(
    (accumulateur, currentVal) => accumulateur + currentVal
  )
);
```





# Les objets : Valeurs primitives vs Objets

## ➤ Primitives:

➤ undefined, null, boolean, numériques, et strings

➤ **Immutable** : leur allocation de mémoire initiale ne change jamais

➤ Sont comparées par **valeur**

## ➤ Objets :

➤ Tout le reste: objets, tableaux et fonctions

➤ **Mutable**: leur mémoire peut changer dynamiquement

➤ Sont comparés par **référence**

# Les objets

- Un **objet** est une **valeur composite**
- C'est une collection de propriétés désordonnée, chacune avec un nom et une valeur et qui peuvent être:
  - Des Attributs
  - Des Méthodes
- Les propriétés peuvent être ajoutés **dynamiquement**
- Trois types d'objets :
  - **Objet Native**: défini par la spécification ECMAScript (Arrays, fonctions, dates)
  - **Objet Hôte**: défini par l'environnement hôte (e.g. navigateur) dans lequel l'interpréteur est embarqué, i.e. HTML\_Element dans le JS coté client.
  - **Object défini par l'utilisateur** : objet crée par l'exécution du code JS.

# Le modèle objet JS : Les prototypes

- Un langage basée sur les prototypes ne possède (généralement) que des objets.
- Parmi ces objets il peut y avoir des objets prototypes agissant comme un modèle permettant ainsi s'avoir des propriétés et des méthodes initiales (jouant le rôle de la classe mais sans en être une).
- Tout objet peut définir ces propres propriétés dynamiquement et à n'importe quel moment.
- Si une propriété est ajoutée à un objet utilisé comme prototype, tous les objets qui l'utilisent comme prototype bénéficieront de cette propriété.

# Le modèle objet JS : Les prototypes

Langage de classe (Java)	Langage de prototype (JavaScript)
classes et instances sont deux entités distinctes.	Tous les objets sont des instances.
Une classe est définie avec une définition de classe. On instancie une classe avec des méthodes appelées constructeurs	On définit et on crée un ensemble d'objets avec des fonctions qui sont des constructeurs.
On crée un seul objet grâce à l'opérateur new.	Même chose
hiérarchie d'objets construite en utilisant les définitions des classes.	Hiérarchie d'objets en assignant un prototype à un objet dans le constructeur.
Les objets héritent des propriétés appartenant à la chaîne des classes de la hiérarchie.	Les objets héritent des propriétés appartenant à la chaîne des prototypes de la hiérarchie.
La définition de la classe définit exactement toutes les propriétés de toutes les instances d'une classe. Il est impossible d'ajouter des propriétés dynamiquement pendant l'exécution.	Le constructeur ou le prototype définit un ensemble de propriétés initiales. Possibilité de retirer des propriétés dynamiquement, pour certains objets en particuliers ou bien pour l'ensemble des objets.

# Les objets : Création

## ➤ Avec les objets littéraux

- Une **paire de clef valeur** séparée par des **“;”**
- Le nom d'une propriété doit être défini en tant que chaîne si :
  - Il inclut un espace
  - C'est un mot réservé

## ➤ Avec l'opérateur new

- **new** doit être suivi du constructeur

## ➤ Avec Object.create()

- Crée un nouvel objet en utilisant le premier argument comme prototype de l'objet

```
// Avec les objets littéraux
var book = {
    "main title": "JavaScript",
    'sub-title': "The Guide",
    "for": "all audiences",
    author: {
        firstname: "David",
        surname: "Flanagan"
    }
};

// Avec l'opérateur new
var o = new Object();
var a = new Array();
var d = new Date();
var r = new RegExp("js");

// Avec Object.create()
var o1 = Object.create({x:1, y:2});
var o2 = Object.create(null);
var o3 = Object.create(Object.prototype);
var o4 = Object.create(o1);

//Exemple constructeur
function Personne(nom, prenom) {
    this.nom=nom;
    this.prenom=prenom;
    this.affiche=function() {
        console.log(this.nom+this.prenom);
    }
}
```



```
// ES5
const monObjetES5 = {
  maMethode1: function () {
    console.log("hello");
  },
  maMethode2: function (arg1, arg2) {
    console.log(arg2);
  },
};


// Equivalent ES6
const monObjet = {
  maMethode1() {
    console.log("hello");
  },
  maMethode2(arg1, arg2) {
    console.log(arg2);
  },
};
monObjet.maMethode1();
// => "hello"
monObjet.maMethode2("banana", true);
```



# Les objets : Création

## ➤ Les valeurs implicites par homonyme

- Lorsque le nom de la variable est le même que le nom de la propriété de votre objet, il n'est pas nécessaire de spécifier le nom et la valeur de la propriété.



```

23  const firstname = 'aymen';
24  const name = 'aymen';
25
26  const user = {
27    firstname,
28    name
29  }
30  console.log(user);

```

TERMINAL ... 2: bash + ▢

```

aymen@DESKTOP-OS8QNV6 MINGW64 ~/Desktop/exemples ES6
$ node script.js
{ firstname: 'aymen', name: 'aymen' }

```

## ➤ Nom de propriété dynamique.

- Avec ES6 vous pouvez dorénavant définir des noms dynamique de vos variables lors de la définition.

```

const nom = 'aymen';
function getAge() {
  return 38;
}
const monObjet = {
  [nom + getAge()]: "Computer Science",
};
console.log(monObjet);
//{ aymen38: 'Computer Science' }

```

- Ajout et modification des propriétés
  - Avec `'.'` ou `[]`
- Suppression des propriétés
  - Avec la fonction `delete` suivie de la propriété à supprimer
- Tester et énumérer des propriétés
  - Avec l'opérateur `in`
    - Pour accéder à la valeur, on utilise `o[indice]`, et non `o.indice`

```
// Ajout et modification des propriétés
var author = book.author;
var name = author.surname;
var title = book["main title"];

// Suppression des propriétés
delete book.author;
delete book["main title"];
delete Object.prototype; //on ne peut
pas la supprimer
//Tester et énumérer des propriétés
var o = { x: 10}
"x" in o           // true
"y" in o           // false

for (p in o) {
    console.log(p);
    //=> x
    console.log(o[p]);
    //=> 10
    console.log(o.p);
    //=> undefined
}
```





## ➤ S rialisation d'Object

- C'est le processus de **transformation d'un objet** en une **cha ne** de caract re et qui peut  tre r cup r e ult rieurement.
- 2 fonctions :
  - **JSON.stringify(objet)** : retourne une cha ne
  - **JSON.parse(objet)** : restore l'objet partir de la cha ne JSON



```
o = {x:1, y:{z:[false,null,""]}};
s = JSON.stringify(o);
p = JSON.parse(s);
```

```
// s : '{"x":1,"y":{"z":[false,null,""]}}'
// p est une copie de o
```

# Les Fonctions

- C'est un objet « spécial »
  - Vous pouvez lui ajouter des propriétés et invoquer des méthodes.
- C'est un bloc JS défini une fois mais qui peut être exécuté à volonté.
- Paramétrable :
  - Inclut un ensemble d'identifiant, paramètres travaillant comme des variables locales.
  - L'invocation de la fonction nécessite de lui passer les arguments.
- Peut retourner une valeur
- Possède un contexte d'invocation
  - Le mot clé this
- Peut être assigné à la propriété d'un objet qui devient une méthode
- Quand une fonction est invoqué à travers un objet, cet objet devient son contexte d'invocation.
- Peut être utilisé pour initialiser un objet. C'est le constructeur de l'objet.

# Les Fonctions

➤ Une fonction est déclarée en utilisant le mot clé **function**

➤ Une fonction littérale est déclarée

```
var carre=function(x) {return x*x;}
```

➤ Dans ce cas, seule la déclaration va être hoistée.

➤ Déclaration standard

```
function square(x) { return x*x; }
```

➤ Dans ce cas c'est toute la fonction qui va être hoistée.

# Les Fonctions

```
function foo(){
  function bar() {
    return 3;
  }
  return bar();
  function bar() {
    return 8;
  }
}
alert(foo());
```

```
// return 8
```

```
/******
```

```
function foo(){
  var bar = function() {
    return 3;
  };
  return bar();
  var bar = function() {
    return 8;
  };
}
alert(foo());
```

```
// return 3
```

```
alert(foo());
function foo(){
  var bar = function() {
    return 3;
  };
  return bar();
  var bar = function() {
    return 8;
  };
}
```

```
// return 3
```

```
/******
```

```
function foo(){
  return bar();
  var bar = function() {
    return 3;
  };
  var bar = function() {
    return 8;
  };
}
alert(foo());
```

```
//[Type Error: bar is not a function]
```



# Les Fonctions

- Comme toute fonction, elle ne sera évoqué que si on fait appel à elle.
- Les fonctions JavaScript peuvent être invoqué de plusieurs façons :
  - En tant que fonction
  - En tant que méthode
  - En tant que constructeur
  - Indirectement avec les méthodes `call()` et `apply()`

## ➤ Invocation d'une méthode

➤ `o["m"](x,y);`

➤ `a[0](z)` // En supposant que `a[0]` soit une fonction.

➤ `f().m();` // Invoque la méthode `m()` de la valeur de retour de `f()`

```
var calculator = {                                // un objet littéral
  operand1: 1,
  operand2: 1,
  add: function() {
    // Ici on utilise le this pour faire référence à l'objet
    // (le context d'exécution)
    this.result = this.operand1 + this.operand2;
  }
};
calculator.add();
calculator.result                                // => 2
```



# Les Fonctions

## ➤ Invocation indirecte

- Les fonctions JS sont des objets, ils peuvent donc avoir des méthodes.
- Deux méthodes prédéfinies permettent d'invoquer indirectement la fonction : `call()` et `apply()`
- Elles permettent toutes les deux d'invoquer une méthode temporaire d'un objet sans l'ajouter à son prototype.
- Le premier arguments des deux méthodes est l'objet auquel on applique la méthode.
  - `call()`: utilise sa liste d'arguments comme arguments de la méthode invoquer.
  - `apply()`: Ne connaissant pas le nombre des arguments on passe un tableau.



# Les Fonctions

## ➤ Invocation indirecte

- Les fonctions JS sont des objets, ils peuvent donc avoir des méthodes.
- Deux méthodes prédéfinies permettent d'invoquer indirectement la fonction : `call()` et `apply()`
- Elles permettent toutes les deux d'invoquer une méthode temporaire d'un objet sans l'ajouter à son prototype.
- Le premier arguments des deux méthodes est l'objet

```
f.call(o,1,2); // Associe l'objet o à la méthode f avec les arguments 1 et 2
f.apply(o,[1,2]); // Associe l'objet o la méthode f avec le tableau d'arguments
```

```
// Les deux sont équivalentes à :
```

```
o.m = f;           // Crée une méthode temporaire de o.
o.m(1,2);          // L'invoquer en lui passant 2 paramètres.
delete o.m;        // Supprimer la méthode.
```





# Les Fonctions

- S'il y a plus de paramètres, il n'y a aucune façon d'accéder directement à ces derniers
- On peut utiliser l'objet `arguments`
  - Tableau contenant l'ensemble des paramètres transmises
  - Permet de définir des fonctions avec un nombre quelconque de paramètres



```
function max(/* ... */) {
    // On initialise à - l'infini :D
    var max = Number.NEGATIVE_INFINITY;
    // On boucle sur les arguments pour chercher le plus grand.
    for (var i = 0; i < arguments.length; i++)
        if (arguments[i] > max) max = arguments[i];
    return max;
}

//Appeler avec autant de paramètres que vous voulez
var largest = max(1, 10, 2, 20, 3, 30, 4, 400, 500, 6);    // => 500
```

# Les Fonctions auto exécutées

- Vous pouvez déclencher l'appel de votre fonction en même temps que sa définition.
- Syntaxe:  

```
(function(param1, ..., paramN) {  
    // traitement  
})(p1, ..., pN);
```
- Synthétise la déclaration et l'appel en une seule syntaxe

# ES6 : Les Fonctions fléchées

- La notation des fonctions fléchées (arrow functions) est l'exemple typique de syntaxe visant à optimiser la compacité du code.
- Cette notation astucieuse, utilisant les caractères `=>`, est même finalement assez lisible.
- Si la fonction attend un seul paramètre, la syntaxe générique est :
- Si la fonction attend plusieurs paramètres, les parenthèses sont obligatoires :

```
p => { /* Instructions */};
```

```
(p1, ..., pN) => { /* Instructions */};
```

- Nous pouvons réécrire notre fonction de mise à jour du titre avec :

```
const saySomething = message => { console.log(message); };
```

# ES6 : Les Fonctions fléchées

Si la fonction n'a pas de paramètre, on utilise les parenthèses **vides** :

```
() => { /* Instructions */};
```

Si l'instruction de la fonction retourne une expression, on **écrit** :

```
(p1, ..., pN) => { return expression };
```

La syntaxe peut être encore réduite **à** :

```
(p1, ..., pN) => expression;
```

# ES6 : Les Fonctions fléchées Avantages

- L'un des avantages majeurs de l'utilisation des fonctions fléchées est la notion de `this` lexical et non local.

```
function Voiture() {
  this.kilometres = 100;
  console.log(this.kilometres);
  setTimeout(function () {
    console.log(this);
    this.kilometres += 10;
    console.log(this.kilometres);
  }, 1000);
}
new Voiture();
```

```
$ node fonctionArrow.js
100
Timeout {
  _idleTimeout: 1000,
  _idlePrev: null,
  _idleNext: null,
  _idleStart: 46,
  _onTimeout: [Function (anonymous)],
  _timerArgs: undefined,
  _repeat: null,
  _destroyed: false,
  [Symbol(refed)]: true,
  [Symbol(kHasPrimitive)]: false,
  [Symbol(refed)]: true,
  [Symbol(kHasPrimitive)]: false,
  [Symbol(asyncId)]: 5,
  [Symbol(triggerId)]: 1
}
NaN
```

# ES6 : Les Fonctions fléchées Avantages

- L'un des avantages majeurs de l'utilisation des fonctions fléchées est la notion de `this` lexical et non local.

```
function Voiture() {
  var self = this;
  self.kilometres = 100;
  console.log(self.kilometres);
  setTimeout(function() {
    console.log("this", self);
    self.kilometres += 10;
    console.log(self.kilometres);
  }, 2000);
}
new Voiture();
```

```
function Voiture() {
  this.kilometres = 100;
  console.log(this.kilometres);
  setTimeout(
    function () {
      console.log("this", this);
      this.kilometres += 10;
      console.log(this.kilometres);
    }.bind(this),
    2000
  );
}
new Voiture();
```

```
$ node fonctionArrow.js
100
this : Voiture { kilometres: 100 }
110
```

# ES6 : Les Fonctions fléchées Avantages

- Les fonctions fléchées en es6 utilise un this lexical (celui du contexte ou elles sont définies).

```
function Voiture() {  
  this.kilometres = 100;  
  console.log(this.kilometres);  
  setTimeout(() => {  
    console.log("this", this);  
    this.kilometres += 10;  
    console.log(this.kilometres);  
  }, 1000);  
}  
new Voiture();
```

```
100  
this Voiture { kilometres: 100 }  
110
```

# Les Fonctions (callback function)

- Une fonction de rappel **callback function** est une fonction passée en paramètre d'une autre fonction. Pour JavaScript, c'est le cas par exemple de *setTimeout()*, *setInterval()* ou encore des fonctions appelées par un gestionnaire d'événement (client side).
- Permettent la généricité de traitement.





# Les Fonctions (callback function)

- Une fonction de rappel **callback function** est une fonction passée en paramètre d'une autre fonction. Pour JavaScript, c'est le cas par exemple de *setTimeout()*, *setInterval()* ou encore des fonctions appelées par un gestionnaire d'événement (client

```
// Retourne une fonction qui calcule f(g)
// La fonction retournée g passe tous ces arguments à f, puis retourne la valeur
// de retour de f. (c'est le principe de f°g).
```



```
// f et g sont appelées avec le même this.
```

```
function compose(f,g) {
    return function() {
        // Nous avons utilisé call pour f parce qu'on lui transmet une seule
        // valeur
        // Nous avons utilisé apply pour g parce qu'on lui transmet un tableau
        // de paramètres
        return f.call(this, g.apply(this, arguments));
    };
}
```

```
var square = function(x) { return x*x; };
var sum = function(x,y) { return x+y; };
var squareofsum = compose(square, sum);
squareofsum(2,3)
```

// => 25



# Les Fonctions

```
function add(x,y) { return x + y; }
function subtract(x,y) { return x - y; }
function multiply(x,y) { return x * y; }
function divide(x,y) { return x / y; }

// Voici une fonction qui prend en
// paramètre la fonction ainsi que les deux
// opérandes.
function operate(operator, operand1,
operand2) {
    return operator(operand1,
operand2);
}
// Que représente cet exemple
var i = operate(add, operate(add, 2, 3),
                operate(multiply, 4,
5));
// On regroupe tout ca dans un objet
var operators = {
    add: function(x,y) { return x+y; },
    subtract: function(x,y) { return x-y;
},
    multiply: function(x,y) { return x*y;
},
    divide: function(x,y) { return x/y; },
    pow: Math.pow
};
```

// Cette fonction prend le nom de l'opération, vérifie que c'est une fonction de l'objet operators et ensuite l'exécute

```
function operate2(operation, operand1,
operand2) {
    if (typeof operators[operation] ===
        "function")
        return operators[operation](operand1,
operand2);
    else throw "unknown operator";
}
```

// Permet de retourner la chaine "hello world "

```
var j = operate2("add", "hello",
operate2("add",
        " ", "world"));
```


```
// utilise pow:
var k = operate2("pow", 10, 2);
```



# Les Classes (ES6)

- En JS vanilla la définition de classes se faisait à travers des fonctions.

```
function creerPersonne(nom, age) {
  // propriétés
  this.nom = nom;
  this.age = age;
  // méthode
  this.log = function () {
    console.log(this.nom + " a " + this.age + " ans.");
  };
}
var aymenSellaouti = new creerPersonne("Aymen Sellaouti", 38);
aymenSellaouti.log();
```



- Depuis ES6, vous pouvez définir des classes.

## ➤ Syntaxe

```
class Personne {
  constructor(nom, age) {
    // propriétés
    this.nom = nom;
    this.age = age;
    // méthode
    this.log = function () {
      console.log(this.nom + " a " + this.age + " ans.");
    };
  }
}
const aymen = new Personne("aymen", 38);
aymen.log();
```




# Les Classes : Héritage (ES6)

- Pour faire de l'héritage, utiliser le mot clé **extends**.
- Pour référencer le parent utiliser le mot clé **super**

```
class Admin extends Personne {
  constructor(nom, age, email, password) {
    super(nom, age);
    this.email = email;
    this.password = password;
  }
  log() {
    super.log();
    console.log("cc");
    console.log(`mon email est : ${this.email}`);
  }
}

const admin = new Admin(
  "aymen sellauti",
  38,
  "aymen.sellauti@gmail.com",
  "123456"
);
admin.log();
```



# Les Classes : getter et setter (ES6)

- Pour les getter et setter ajouter les mot clé **get** et **set** devant la méthode au nom de la propriété.
- **Attention**, si vous écrivez `set email(email){this.email = email }`, vous allez déclencher une boucle infinie
- En effet, puisque vous déclenchez un nouveau set. Il faut donc modifier le nom de la variable interne pour qu'il soit différent du nom de la méthode set.
- La convention est d'utiliser un tiret bas pour différencier la variable de classe de son setter/getter (`this._email` et `set email`).

```
class Admin extends Personne {
  constructor(nom, age, email, password) {
    super(nom, age);
    this._email = email;
    this.password = password;
  }
  get email() {
    return `L'email est : ${this._email}`;
  }
  set email(email) {
    if (email.includes("@")) {
      this._email = email;
    } else {
      console.log("email must contain @");
    }
  }
  log() {
    super.log();
    console.log("cc");
    console.log(`mon email est : ${this._email}`);
  }
}

const admin = new Admin(
  "aymen sellaouti",
  38,
  "aymen.sellaouti@gmail.com",
  "123456"
);
admin.log();
admin.email = "newEmail";
admin.log();
```




# Les Classes : méthodes static (ES6)

- Comme dans tout les langages, une méthode statique est une méthode de classe indépendante de toute instance.
- Pour accéder à une propriété static, vous devez passer par la classe.

```
class Admin extends Personne {
  constructor(nom, age, email, password) {
    super(nom, age);
    this._email = email;
    this.password = password;
    Admin.status++;
  }
  static status = 1;
  static wholAm()
    console.log("I am an admin");
}

const admin = new Admin(
  "aymen sellaouti",
  38,
  "aymen.sellaouti@gmail.com",
  "123456"
);
admin.log();
admin.email = "newEmail";
admin.log();
Admin.wholAm();
console.log(Admin.status);
```



# ES6 déstructuration (destructuring)

- La déstructuration permet de définir des variables depuis un tableau ou un objet en utilisant de la mise en correspondance (matching) et plus particulièrement de la correspondance de motifs ( pattern matching)

```
var array = ["un", "deux", "trois", "quatre"];
var [a, b, c, d] = array;
```

```
console.log(a, b, c, d);
// => un, deux, trois, quatre
```

```
// => Il est possible d'ignorer un élément
var [un, , trois, quatre] = ["α", "β", "γ", "δ"];
```

```
console.log(un, trois, quatre);
// => α, γ, δ
```





# ES6 déstructuration (destructuring)

- Pour les objets on accède via les noms des propriétés.
- Ces noms peuvent être renommés.

```
//Pour les objets on accède avec les noms de propriétés
```

```
var newObject = { a: 1, b: 2, c: 3};
```

```
var { a, b, c } = newObject;
```

```
console.log(a, b, c);
```

```
// On peut nommer les paramètres récupérés
```

```
var newObject = { a: 1, b: 2, c: 3};
```

```
var { a: un, b: deux, c: trois } = newObject;
```

```
console.log(un, deux, trois);
```

- En cas d'erreur la variable aura comme valeur *undefined*

```
var [a, b, c] = [1, 2];
```

```
console.log(a, b, c);
```

```
// => 1, 2, undefined
```

```
var [a] = [];
```

```
console.log(a);
```

```
// => undefined
```

# ES6 décomposition (spread)

- La **syntaxe de décomposition** permet d'étendre un itérable (tableau, chaîne de caractères ou objet) en lieu et place de plusieurs arguments (pour les appels de fonctions) ou de plusieurs éléments (pour les littéraux de tableaux) ou de paires clés-valeurs (pour les littéraux d'objets).
- Pour l'utilisation de la décomposition dans les appels de fonction :  
`f(...objetIterable);`
- Pour les littéraux de tableaux :  
`[...objetIterable, 4, 5, 6]`
- Pour les littéraux objets (nouvelle fonctionnalité pour ECMAScript, actuellement en proposition de niveau 4, finalisée) :  
`let objClone = { ...obj };`

# ES6 décomposition (spread)

- La **syntaxe de décomposition** permet d'étendre un itérable (tableau, chaîne de caractères ou objet) en lieu et place de plusieurs arguments (pour les appels de fonctions) ou de plusieurs éléments (pour les littéraux de tableaux) ou de paires clés-valeurs (pour les littéraux d'objets).
- Pour l'utilisation de la décomposition dans les appels de fonction :  
`f(...objetIterable);`
- Pour les littéraux de tableaux :  
`[...objetIterable, 4, 5, 6]`
- Pour les littéraux objets (nouvelle fonctionnalité pour ECMAScript, actuellement en proposition de niveau 4, finalisée) :  
`let objClone = { ...obj };`

# ES6 fonctions : les valeurs par défaut

- Vous pouvez dorénavant associer des valeurs par défaut à vos paramètres de fonctions.

```

1  function testDefaultParam(message = "message par défaut") {
2    console.log(message);
3  }
4  testDefaultParam();
5  testDefaultParam("message du user");
6

```

TERMINAL   PROBLEMS   OUTPUT   ...   1: bash   +   [ ]   [X]

```

aymen@DESKTOP-OS8QNV6 MINGW64 ~/Desktop/exemples ES6 (master)
$ node fonctions.js
message par défaut
message du user

```

# ES6 fonctions : paramètres de nombre variables

- Vous pouvez avoir un nombre variables de paramètres.
- Vous pouvez aussi récupérer ces paramètres comme vous le souhaitez.

```

JS fonctions.js > ...
5   testDefaultParam("message·du·user"); */
6
7   function genericSomme(val1, val2, ...resteDesParams) {
8       console.log(val1);
9       console.log(val2);
10      console.log(resteDesParams);
11  }
12  genericSomme(1,2,3,4,5,6,7,8);

```

TERMINAL   PROBLEMS   ...   1: bash   +   □

```

aymen@DESKTOP-OS8QNV6 MINGW64 ~/Desktop/exemples ES6 (master)
$ node fonctions.js
1
2
[ 3, 4, 5, 6, 7, 8 ]

```

# ES6 fonctions : paramètres de nombre variables

- Vous pouvez aussi utiliser le spread operator afin de décomposer les paramètres que vous passez à votre fonction.

```
JS fonctions.js > genericSomme
6
7 function genericSomme(val1, val2, ...resteDesParams) {
8     console.log(val1);
9     console.log(val2);
10    console.log(resteDesParams);
11 }
12 tableau = [1,2,3,4,5,6,7,8];
13 genericSomme(...tableau);
```

TERMINAL PROBLEMS ... 1: bash

```
aymen@DESKTOP-OS8QNV6 MINGW64 ~/Desktop/exemples ES6 (master)
$ node fonctions.js
1
2
[ 3, 4, 5, 6, 7, 8 ]
```

# ES6 Les collections et les dictionnaires

- Avant pour gérer une collection on utilisait un tableau et pour gérer un dictionnaire, on utilisait les objets.
- Afin de gérer ses deux structures d'une façon plus fluide deux nouvelles structures ont été ajoutées :
  - Set
  - Map

# ES6 Les collections (Set)

- Un Set est un objet permettant de stocker un ensemble de valeurs, quelle qu'elle soit comme un tableau.
- La différence est que le Set va garantir l'unicité des éléments.
- Si vous ajouter un élément déjà existant dans le Set, celle-ci ne sera stockée qu'une seule fois.
- Set conserve les valeurs dans l'ordre ou elles ont été insérées.
- Pour définir un Set **instancier un objet de la Set**;
- Pour **ajouter un élément** utiliser la méthode **add**.
- Pour **supprimer un élément** utiliser la méthode **delete**.
- Pour vérifier **l'existence d'un élément** utiliser la méthode **has**.
- Pour récupérer la **taille** d'un Set utiliser **l'attribut size**.



# ES6 Les collections (Set)

```
const maCollection = new Set();
console.log('Taille initiale de ma collection',maCollection.size);
maCollection.add('aymen');
maCollection.add(38);
console.log('Taille de ma collection',maCollection.size);
maCollection.add(38);
console.log('Taille de ma collection',maCollection.size);
maCollection.delete(38);
console.log('Taille de ma collection',maCollection.size);
console.log('est ce que maCollection a la valeur 38', maCollection.has(38));
```

```
$ node collection.js
Taille initiale de ma collection 0
Taille de ma collection 2
Taille de ma collection 2
Taille de ma collection 1
est ce que maCollection a la valeur 38 false
```

# ES6 Les dictionnaires (Map)

- Le paramètre valeurs passé peut être tout objet itérable.
- Un Map est un objet permettant de stocker des valeurs indexées par un clé unique, on peut le considérer comme un dictionnaire.
- Contrairement à un Objet où les noms de propriétés doivent être des chaînes, les clé et les valeurs d'un Map peuvent être de n'importe quel type.
- Pour créer un map instancier l'objet Map.
- Pour **ajouter un élément** utiliser la méthode **set**.
- Pour **recupérer un élément** utiliser la méthode **get**.
- Pour **supprimer un élément** utiliser la méthode **delete**.
- Pour vérifier **l'existence d'un élément** utiliser la méthode **has**.
- Pour récupérer la **taille** d'un Set utiliser **l'attribut size**.

# ES6 Les dictionnaires (Map)

```

1  const map = new Map();
2  // Ajouter et mettre à jour un élément
3
4  map.set('name', 'selaouti');
5  console.log(map);
6  map.set('name', 'sellaouti');
7  console.log(map);
8  map.set('prenom', 'aymen');
9  console.log(map);
10 console.log('As tu une propriété name ?? ',map.has('name'));
11 console.log('Quel est ta taille ?? ',map.size);
12 map.forEach((valeur, cle)=> {
13     console.log(`Je suis le champ ${cle} et ma valeur est ${valeur}`);
14 })
15 map.delete('name');
16 console.log(map);
17 |

```

TERMINAL

PROBLEMS

OUTPUT

DEBUG CONSOLE

1: bash

```

Map(1) { 'name' => 'selaouti' }
Map(1) { 'name' => 'sellaouti' }
Map(2) { 'name' => 'sellaouti', 'prenom' => 'aymen' }
As tu une propriété name ?? true
Quel est ta taille ?? 2
Je suis le champ name et ma valeur est sellaouti
Je suis le champ prenom et ma valeur est aymen
Map(1) { 'prenom' => 'aymen' }

```

# ES6 Les promesses (Promise)

- Une promesse est un objet (Promise) qui représente la complétion ou l'échec d'une opération asynchrone.
- Le fonctionnement des promesses est le suivant :
  - On crée une promesse en instanciant un objet de Promise.
  - Le constructeur prend en paramètre deux callbacks qui sont resolve et reject.
  - La promesse va toujours retourner deux résultats :
    - resolve en cas de succès
    - reject en cas d'erreur
  - Vous devrez donc gérer les deux cas afin de créer votre traitement

# ES6 Les promesses (Promise)

- Pour consommer la promesse, vous avez deux méthodes à gérer :
  - **then** en cas de succès et qui prend en paramètre les données passées à la méthode **resolve**.
  - **catch** en cas d'erreur et qui récupère un objet **error**.

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(3);
  }, 2000);
});
promise
  .then((x) => {
    console.log("resolved with value :", x);
  })
  .catch((error) => console.log("caught error", error));
```

# ES6 Les promesses (Promise)

- La méthode `then` retourne à son tour une Promise.
- Ceci nous permet de chainer les fonctions dépendantes les une des autres.

```
faireQqc().then(function(result) {  
    return faireAutreChose(result);  
})  
.then(function(newResult) {  
    return faireUnTroisiemeTruc(newResult);  
})  
.then(function(finalResult) {  
    console.log('Résultat final : ' + finalResult);  
})  
.catch(failureCallback);
```

# ES6 await async

- Afin d'éviter les successions de **then** des promesses pour gérer les fonctions **asynchrones**, vous pouvez utiliser les mots clés **async** et **await**.
- En précédant une fonction avec le mot clé **async**, vous spécifier que votre méthode est asynchrone.
- Si vous souhaitez que dans cette fonction il y a un traitement asynchrone et que vous souhaitez attendre sa complétion pour continuer le traitement, il vous suffit de le précéder de **await**.

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved');
    }, 2000);
  });
}

async function asyncCall() {
  console.log('calling');
  const result = await resolveAfter2Seconds();
  console.log(result);
  // expected output: "resolved"
}

asyncCall();
```

# Intégrer le Js avec votre page HTML

- Afin d'intégrer votre code JS avec votre page HTML, deux méthodes peuvent être appliquées :
  - L'intégrer directement à travers une balise `<script></script>` au sein de votre page HTML et y mettre votre code JS.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>
  <script>
    console.log('Bonjour je suis un
script non portable je ne sert qu\'ici
:(');
  </script>
</head>
<body>

</body>
</html>
```

- Ça assassine le concept de portabilité du code.
- Anti séparation des rôles.



# Intégrer le Js avec votre page HTML

- insertion d'un fichier externe (usuellement '.js')

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Title</title>

</head>
<body>
<script src='app.js'></script>
</body>
</html>
```

first.html

```
console.log('Bonjour je suis un script
libre liiiiiiibre vous avez besoin de
moi il suffit de m'appeler :D');
```

app.js

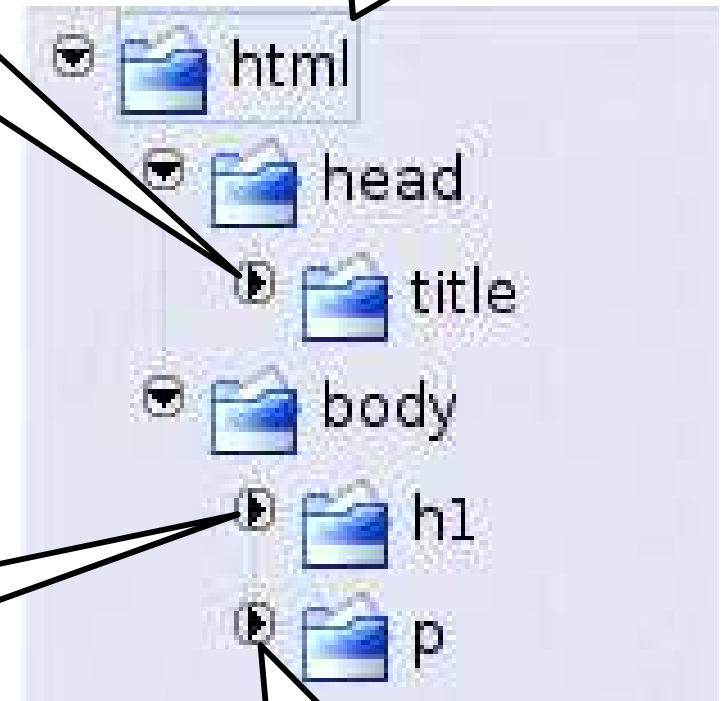
## Remarque

Traitement séquentiel des pages par un navigateur ⇒ Placer les scripts dans le footer permet de charger les éléments visuels avant les scripts.

# Le DOM

- Le Document Object Model (**DOM**) représente une API pour les documents HTML et XML permettant leur manipulation.
- En tant que recommandation du W3C, l'objectif du DOM est de fournir une interface de programmation standard pour être utilisée par tous (applications, OS)
- Permet de construire une **arborescence** de la structure d'un document et de ses éléments.
- Modélisé sous forme **d'arbre**.
- Chaque élément du DOM est représenté par un objet en JavaScript.
- La tête de l'arbre est l'objet **window** qui représente la fenêtre du navigateur.
- L'objet **document** est un sous-objet de window c'est son fils. Il représente la page Web et permet de pointer ou d'accéder à la balise HTML.

```
<html>
<head>
  <title>Titre du document</title>
</head>
<body>
  <h1>Titre</h1>
  <p>Un peu de texte</p>
</body>
</html>
```



**title** est l'unique enfant de **head**

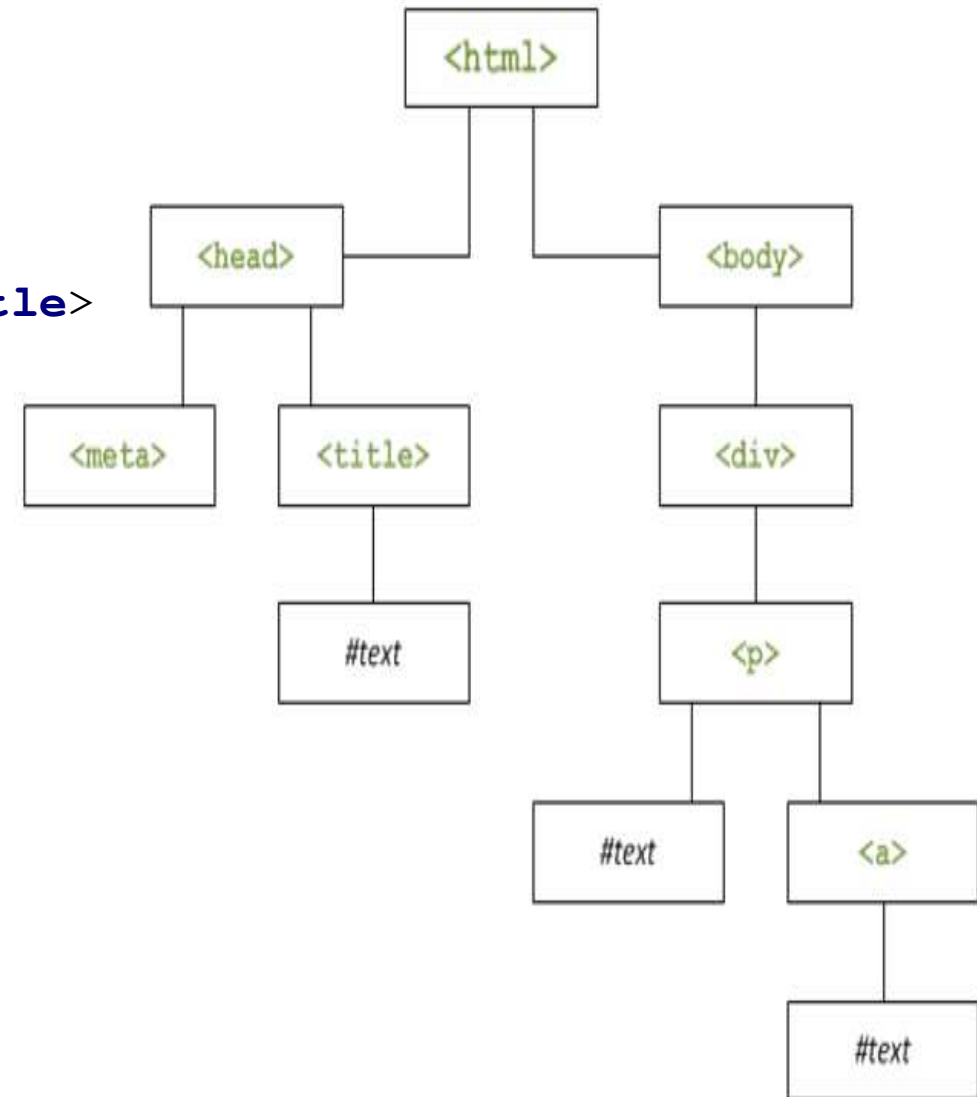
**html** est le parent direct de **head** et de **body**, ses deux enfants

**h1** est le premier enfant de **body**

**p** est le deuxième, il est frère de **h1**

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Le titre de la page</title>
</head>

<body>
<div>
  <p>Un peu de texte
    <a>et un lien</a>
  </p>
</div>
</body>
</html>
```



# Les différents types des objets du DOM

- **Node** : C'est le type le plus générique, tous les éléments du DOM sont des Nodes.
- **Element** : Représente un élément HTML ou XML
- **HTMLElement** : Dans le DOM HTML, un Element est un HTMLElement. Ça représente les éléments HTML du documents , e.g : les balises.
- **NodeAttribute** : Les attributs des balises
- **TextAttribute** : Les attributs textuelles
- Les nœuds commentaires

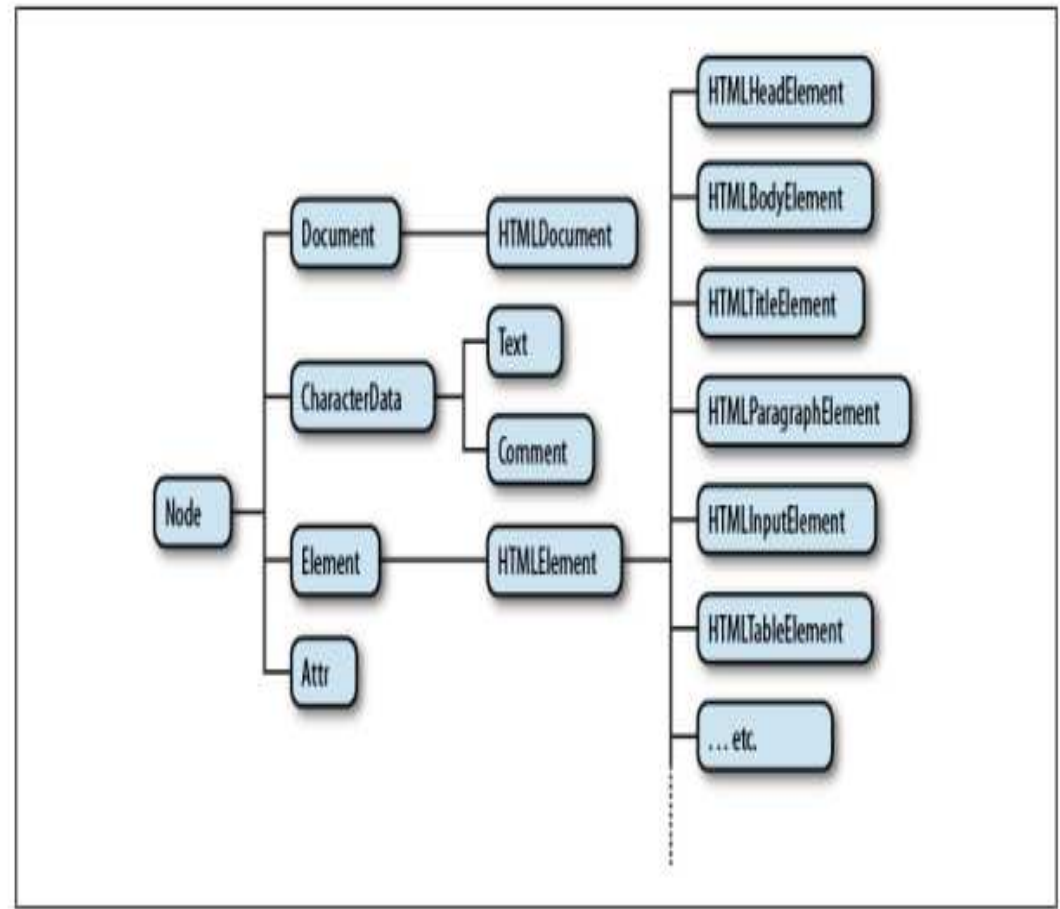
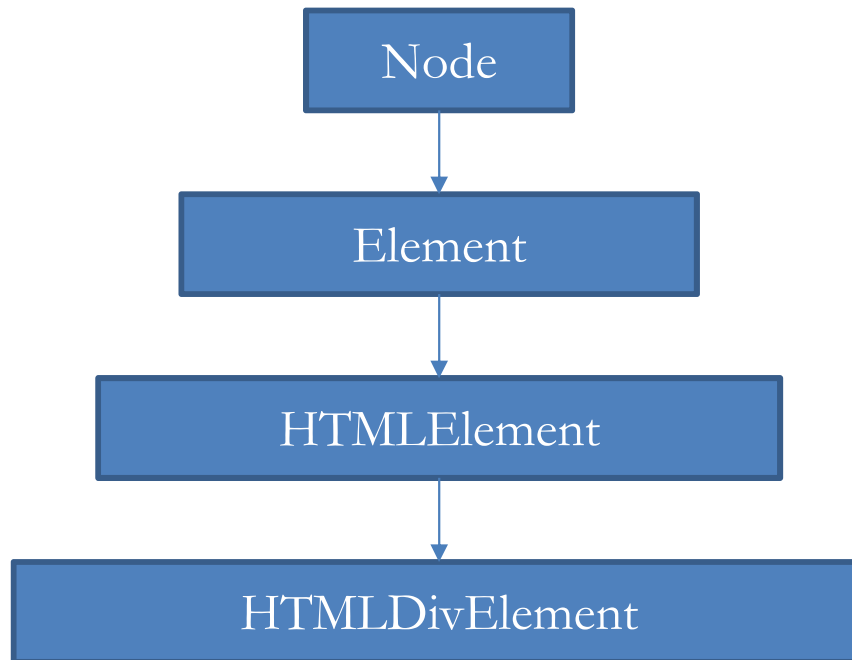
Interface (type de nœud)	Nom de la constante	Valeur retournée par nodeType
Element	Node.ELEMENT_NODE	1
Attribut	Node.ATTRIBUTE_NODE	2
Noeud texte	Node.TEXT_NODE	3
	Node.CDATA_SECTION_NODE	4
	Node.ENTITY_REFERENCE_NODE	5
	Node.ENTITY_NODE	6
Instruction de traitement	Node.PROCESSING_INSTRUCTION_NODE	7
Commentaire	Node.COMMENT_NODE	8
	Node.DOCUMENT_NODE	9
	Node.DOCUMENT_TYPE_NODE	10
Fragment XML	Node.DOCUMENT_FRAGMENT_NODE	11
	Node.NOTATION_NODE	12

# Quelques fonctions de Window

- **alert()** : affiche une alerte et bloque le script lors de son appel.
- **prompt()** : permet d'avoir un champ input permettant de récupérer une valeur de l'utilisateur.
- **Confirm()** : affiche un message contenant deux boutons afin de confirmer ou non.

# Héritage dans le DOM

- Tous les éléments du DOM sont des **objets** ce qui implique qu'ils contiennent tous leurs propres **méthodes** et des **attributs**.
- Ils existent des méthodes et attributs communs à l'ensemble des objets. Ceci est dû au fait que l'ensemble des éléments sont **tous du même type** : les nœuds (**Node**).
- L'aspect d'héritage entre les différents types d'éléments permet d'avoir les éléments en commun.



# Manipulation des éléments

- Pour accéder aux éléments HTML en utilisant le DOM, l'objet **document** offre plusieurs méthodes :
  - **getElementById()** qui, comme son nom l'indique, permet d'accéder à un élément par son id.
  - **getElementsByTagName()** permet de récupérer sous forme d'un **tableau** d'objets tous les éléments du **tag** passé en **paramètre**.
  - **getElementsByName()** permet de récupérer sous forme d'un **tableau** d'objets tous les éléments dont l'**attribut name** est égale au **nom** passé en **paramètre**.
  - **querySelector()** cette fonction prend en **paramètre** une chaîne de caractère représentant un **sélecteur CSS**. Elle retourne le premier élément qui y correspond.
  - **querySelectorAll()** cette fonction prend en **paramètre** une chaîne de caractère représentant un **sélecteur CSS**. Elle retourne un **tableau d'objets** contenant **TOUS** les éléments qui correspondent à ce **sélecteur**.




# Manipulation des éléments : Exemple

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Ma page de Test </title>
</head>
<body>
<div id="monMenu">
  <div class="item">
    <span>menu 1</span>
    <span>menu 2</span>
  </div>
  <div class="Pub">
    <span>Pub 1</span>
    <span>Pub 2</span>
  </div>
</div>
<div id="content">
<span>Ici je vais mettre mon
contenu</span>
</div>
Un formulaire bizarre pour tester
getElementsByName() :
<form name="monNom">
  <input type="password" name="monNom">
  <input>
  <button name="monNom"></button>
</form>
<script src="test.js"></script>
</body>
</html>
```

test.html

menu 1 menu 2  
 Pub 1 Pub 2  
 Ici je vais mettre mon contenu  
 Un formulaire bizarre pour tester getElementByName() :



```
var getByid=
document.getElementById('monMenu');
var getElementByTagName =
document.getElementsByTagName("div");
var getElementByName =
document.getElementsByName("monNom");
console.log('ce que me donne
getElementById : ' + getByid.textContent);
console.log('Ce que donne le
getElementByTagName : ');

for(var
i=0;i<getElementByTagName.length;i++){
console.log(getElementByTagName[i]);
}

console.log('Ce que donne le
getElementByName : ');
for(var
i=0;i<getElementByName.length;i++){
console.log(getElementByName[i]);
}
```

test.js

# Manipulation des éléments : Exemple

ce que me donne getElementById :

[test.js:8](#)

menu 1  
menu 2

Pub 1  
Pub 2

Ce que donne le getElementByTagName :

[test.js:9](#)

```
<div id="monMenu">
  <div class="item">...</div>
  <div class="Pub">...</div>
</div>
```

[test.js:11](#)

```
<div class="item">
  <span>menu 1</span>
  <span>menu 2</span>
</div>
```

[test.js:11](#)

```
<div class="Pub">
  <span>Pub 1</span>
  <span>Pub 2</span>
</div>
```

[test.js:11](#)

```
<div id="content">
  <span>Ici je vais mettre mon contenu</span>
</div>
```

[test.js:11](#)

Ce que donne le getElementByName :

[test.js:13](#)

```
<form name="monNom">...</form>
```

[test.js:15](#)

```
<input type="password" name="monNom">
```

[test.js:15](#)

```
<button name="monNom"></button>
```

[test.js:15](#)

> |

Résultat dans la console

# Manipulation des éléments : Exemple

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Ma page de Test </title>
</head>
<body>
<div id="monMenu">
  <div class="item">
    <span>menu 1</span>
    <span>menu 2</span>
  </div>
  <div class="Pub">
    <span>Pub 1</span>
    <span>Pub 2</span>
  </div>
</div>
<div id="content">
<span>Ici je vais mettre mon
contenu</span>
</div>
Un formulaire bizarre pour tester
getElementsByName() :
<form name="monNom">
  <input type="password" name="monNom">
  <input>
  <button name="monNom"></button>
</form>
<script src="test1.js"></script>
</body>
</html>
```

test.html

menu 1 menu 2  
 Pub 1 Pub 2  
 Ici je vais mettre mon contenu  
 Un formulaire bizarre pour tester getElementByName() :

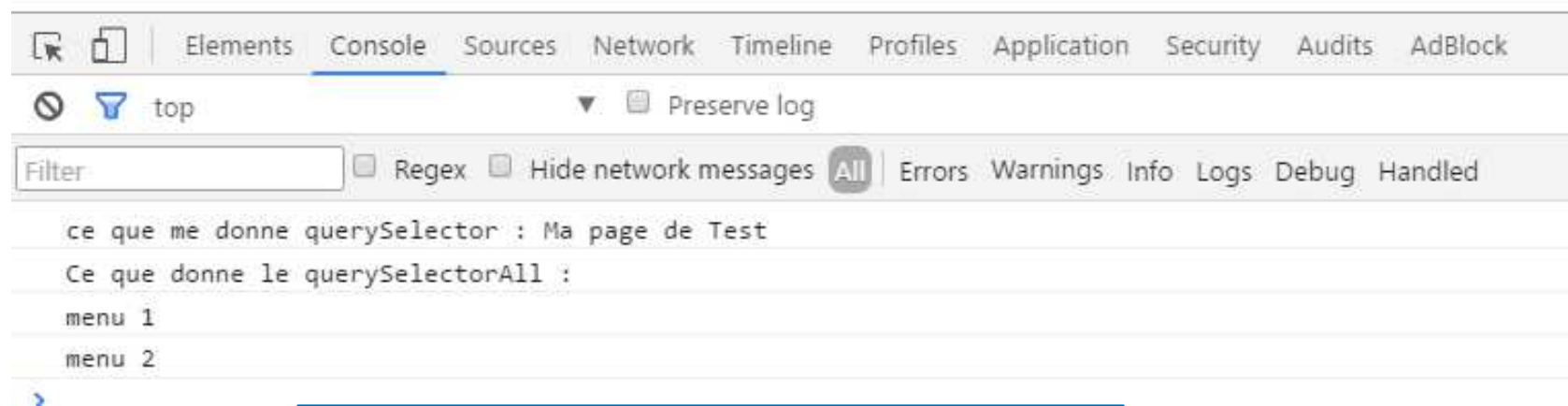
```
var queryselector=
document.querySelector("head title");
var queryselectorAll=
document.querySelectorAll("#monMenu .item
span");

console.log('ce que me donne querySelector
: ' + queryselector.textContent);
console.log('Ce que donne le
querySelectorAll : ');
for(var
i=0;i<queryselectorAll.length;i++){

console.log(queryselectorAll[i].textConten
t);
}
```

test1.js

# Manipulation des éléments : Exemple



Résultat dans la console

- Pour accéder à la liste des classes CSS définies dans l'attribut class d'un élément nous utilisons **className** qui retourne la liste des classes associées à l'élément en question dans une chaîne de caractère où les classes sont séparées par des espaces.
- La méthode **classList** quant à elle retourne un tableau contenant les classes associées à un élément. Elle offre aussi les méthodes suivantes :
  - **add()** qui ajoute une classe au tableau.
  - **remove()** qui supprime une classe.
  - **contains()** qui vérifie si une classe existe ou non.
  - **toggle()** ajoute la classe si elle n'existe pas et l'enlève si elle existe

# Manipulation des éléments

- Afin de lire, de créer ou de modifier un attribut d'un « Element » du DOM, l'objet Element met votre disposition deux méthodes
  - **getAttribute()** qui comme son nom l'indique permet de récupérer la valeur d'un attribut.
  - **setAttribute()** qui permet de la modifier

Exemple :

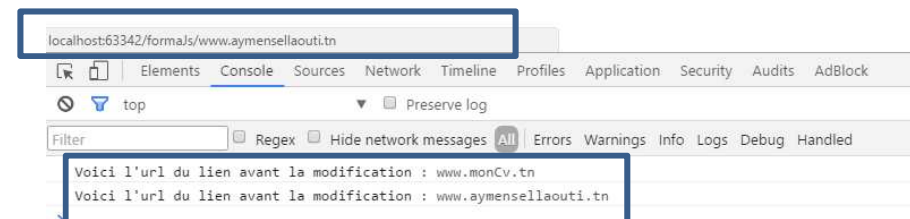
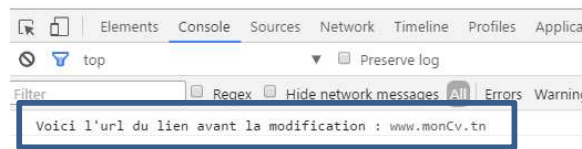
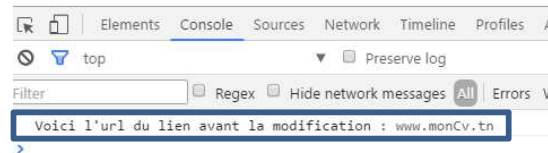
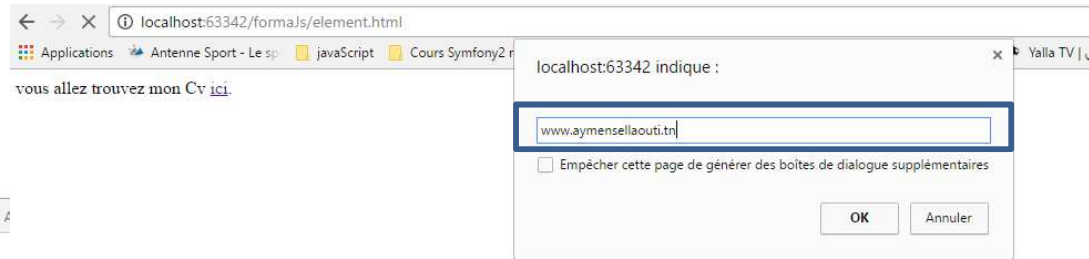
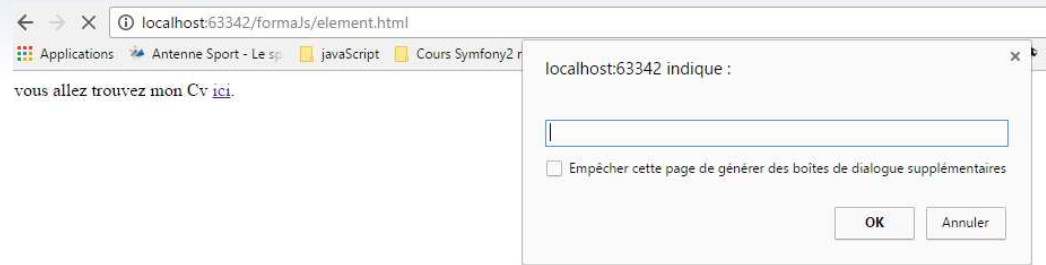
```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Ma page de Test </title>
</head>
<body>
vous allez trouvez mon Cv <a id="lelien"
href="www.monCv.tn">ici</a>.
<script src="test2.js"></script>
</body>
</html>
```

Element.html

```
var
leLien=document.getElementById("lelien");
var dest=lelien.getAttribute("href");
console.log("Voici l'url du lien avant la
modification : "+dest);
var nouveauCvLien=prompt();
leLien.setAttribute("href",nouveauCvLien)
;
dest=lelien.getAttribute("href");
console.log("Voici l'url du lien après la
modification : "+dest);
```

test2.js

# Manipulation des éléments



# Manipulation des éléments

- La deuxième méthode mais qui n'est pas appliquée sur la totalité des navigateurs est d'accéder directement en utilisant le nom de l'attribut
- Dans notre exemple le `getAttribut(«href»)` est remplacé par `.href`. Le `setAttribut` est lui remplacé par `lautreLien.href="newLien.tn"`

```
var autreLien=document.getElementById("laurelien");

alert(autreLien.href);

autreLien.href="newLien.tn"
```

test2.js



# Manipulation le texte des balises

- **innerHTML** : permet de récupérer du **code html** enfant d'une balise.
- Elle permet aussi d'ajouter directement du contenu HTML
- **textContent** (dans sa version standard et **innerText** qui n'est pas supporté par tous les navigateurs) permet de faire le même travail que **innerHTML** sauf qu'elle supprime les balises HTML)

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Ma page de Test </title>
</head>

<body>
<div id="div">
  <p>Je suis dans la div d'id div</p>
</div>
<div id="div1">
  <p>Je suis dans la div d'id div1</p>
</div>
<script src="test4.js"></script>
</body>

</html>
```

Element.html

```
var monDiv=document.getElementById("div");
var monDiv1=document.getElementById("div1");
console.log("Je suis innerHTML voila ce que
j'affiche "+monDiv.innerHTML);

console.log("Je suis textContentt voila ce
que j'affiche "+monDiv1.textContent);

monDiv1.textContent="I delete every thing in
my road <b>i'm a monster 3:</b> ";

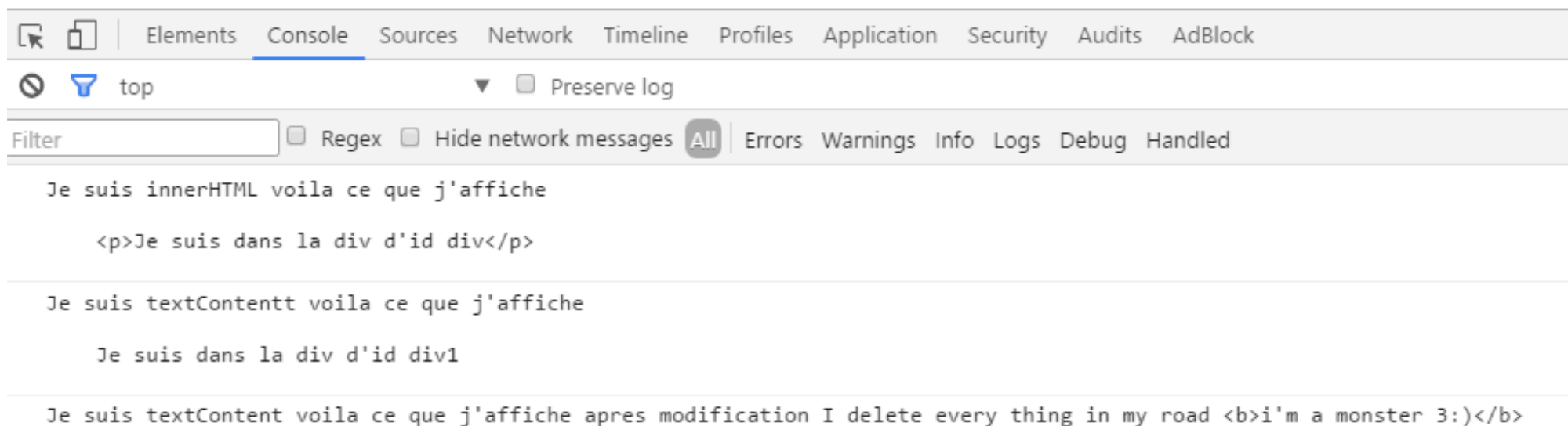
monDiv.innerHTML="I delete every thing in my
road <b>i'm a monster too 3:</b> ";

console.log("Je suis innerHTML voila ce que
j'affiche apres modification
"+monDiv.innerHTML);

console.log("Je suis textContent voila ce
que j'affiche apres modification
"+monDiv1.textContent);
```

test4.js

# Manipulation le texte des balises



console

# Quelques méthodes utiles

- Il existe quelques méthodes très utiles qui renseignent sur les éléments du DOM
- **nodeType** permet de récupérer le type d'un nœud du DOM. Elle retourne un numéro qui correspond au type du nœud. Les types les plus courants sont :
  - 1 Element
  - 2 attribut
  - 3 texte
  - 8 commentaire
- **nodeName** permet de récupérer le nom du nœud en MAJISCULE.

# Se déplacer dans le DOM

- **childNodes** permet de retourner un tableau contenant l'ensemble des nœuds fils.
- **nextSibling** permet d'accéder au nœud suivant donc au frère suivant d'un nœud
- **previousSibling** permet d'accéder au nœud précédent donc au frère précédent d'un nœud
- **nextElementSibling** permet d'accéder à l'élément HTML suivant donc au frère suivant d'un nœud
- **previousElementSibling** permet d'accéder à l'élément HTML précédent donc au frère précédent d'un nœud

# Se déplacer dans le DOM

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Ma page de Test </title>
</head>
<body>

<p id="para">vous allez trouvez mon Cv <a
id="lelien" href="www.monCv.tn">ici</a>.
Veuillez le consulter et appeler moi au
<i>222222</i> pour plus de détails</p>

<div id="monMenu">

  <div id="item">
    <span>menu 1</span>
    <span>menu 2</span>
  </div>

  <div class="Pub">
    <span>Pub 1</span>
    <span>Pub 2</span>
  </div>

</div>
<script src="test6.js">
</script>
</body>
</html>
```

Element.html

```
var nod=document.getElementById("para");
console.log("Bonjour je suis le noeud
"+nod.nodeName) ;
console.log("Pour visualiser la différence
entre firstChild et firstElementChild ");
console.log("bonjour je suis le first child
: "+nod.firstChild);
console.log("bonjour je suis le first
Elementchild : "+nod.firstElementChild);
```

test6.js

vous allez trouvez mon Cv [ici](#). Veuillez le consulter et appeler moi au 222222 pour plus de détails

menu 1 menu 2  
Pub 1 Pub 2



console

# Se déplacer dans le DOM

- Plusieurs propriétés sont offertes permettant le déplacement dans le DOM.
- Ces propriétés permettent à partir d'un nœud d'accéder à son père, ou à ses enfants.
- **parentNode** permet à partir d'un élément d'accéder à son élément père.
- **firstChild** permet, comme son nom l'indique, à partir d'un élément, d'accéder au premier enfant d'un noeud.
- **lastChild** permet, comme son nom l'indique, à partir d'un élément, d'accéder au dernier enfant d'un noeud.
- **firstElementChild** permet, comme son nom l'indique, à partir d'un élément, d'accéder au premier **Element** enfant d'un noeud.
- **lastElementChild** permet, comme son nom l'indique, à partir d'un élément, d'accéder au dernier **Element** enfant d'un noeud.
- **childNodes** permet de retourner un tableau contenant l'ensemble des nœuds fils.

# Se déplacer dans le DOM

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Ma page de Test </title>
</head>
<body>

<p id="para">vous allez trouvez mon Cv <a
id="lelien" href="www.monCv.tn">ici</a>.
Veuillez le consulter et appeler moi au
<i>222222</i> pour plus de détails</p>

<div id="monMenu">

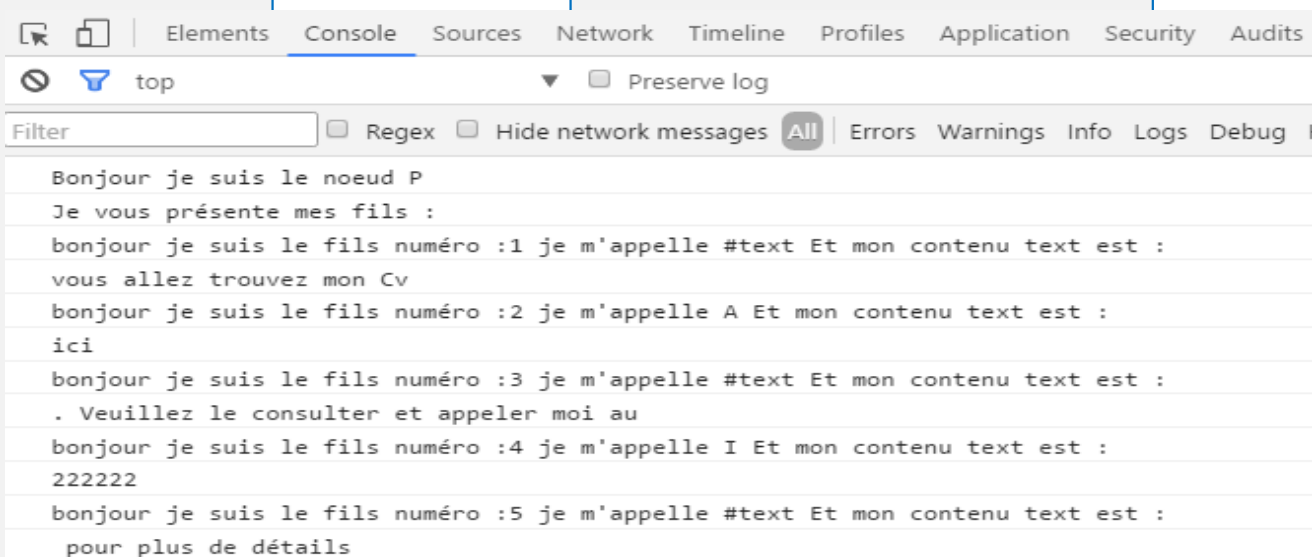
  <div id="item">
    <span>menu 1</span>
    <span>menu 2</span>
  </div>

  <div class="Pub">
    <span>Pub 1</span>
    <span>Pub 2</span>
  </div>

</div>
<script src="test7.js">
</script>
</body>
</html>
```

```
var nod=document.getElementById("para");
console.log("Bonjour je suis le noeud
"+nod.nodeName) ;
console.log("Je vous présente mes fils : ");
var fils=nod.childNodes;
for (var i=0;i<fils.length;i++){
  console.log("bonjour je suis le fils numéro
:"+ (i+1)+" je m'appelle"
+fils[i].nodeName+" Et mon contenu text est :");
  if(fils[i].nodeType==Node.ELEMENT_NODE){
    console.log(fils[i].firstChild.data);
  }else{
    console.log(fils[i].data);
  }
}
```

test7.js



Element.html

console

# Mise à jour du DOM : Ajout d'un élément

- Afin d'ajouter un nœud dans le DOM il faut généralement suivre les étapes suivantes :
  - Récupérer ou créer l'élément à ajouter. Pour le créer, on utilise la méthode de l'objet document **createElement()** qui prend en paramètre une chaîne contenant le **nom de l'élément à créer**.
  - Décorer l'élément avec les attributs nécessaires en utilisant l'une des deux méthodes précédemment mentionnées.
    - En utilisant la méthode **setAttribute(nomAttrib,val)**
    - En accédant directement à la propriété de l'objet : **objet.attribut=valeur**
  - Insérer l'élément dans le DOM en utilisant la méthode **appendChild()** qui prend en paramètre l'élément à insérer et elle ajoute cet élément comme **dernier fils** de l'objet qui a appelé la méthode.
- Pour les **TextNode** on utilise la méthode **createTextNode()** qui prend en paramètre le **texte en question**.
- Il existe aussi la méthode qui permet d'insérer un nœud avant un autre c'est la méthode **insertBefore()** qui prend en paramètres le **nœud à insérer** et le **nœud fils avant lequel on va insérer**. Si ce nœud **n'est pas spécifié** le nouveau nœud sera insérer en **dernier**.



# Mise à jour du DOM : exemple ajout

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Ma page de Test </title>
</head>
<body>

<p id="para">vous allez trouvez mon Cv <a
id="lelien" href="www.monCv.tn">ici</a>.
Veuillez le consulter et appeler
<i>222222</i> pour plus de détail

<div id="monMenu">

  <div id="item">
    <span>menu 1</span>
    <span>menu 2</span>
  </div>

  <div class="Pub">
    <span>Pub 1</span>
    <span>Pub 2</span>
  </div>

</div>
<script src="test8.js">
</script>
</body>
</html>
```

Element.html

```
var nod=document.getElementById("para");
var newNode=document.createElement("img");
newNode.id="asimg";
newNode.src="as.jpg";
newNode.alt="Je suis l'image de aymen";
newNode.width="50";
newNode.height="50";
nod.appendChild(newNode);
```

test8.js

vous allez trouvez mon Cv [ici](#). Veuillez le consulter et appeler moi au 222222 pour plus de détails

menu 1 menu 2  
Pub 1 Pub 2



The screenshot shows a web browser window with the rendered page. The page content matches the HTML code provided. Below the browser window, the Chrome DevTools console is open, showing the DOM tree. The element selected is the newly added image element, which is an tag with id="asimg", src="as.jpg", alt="Je suis l'image de aymen", width="50", and height="50". The console also shows the styles applied to the element, including a width of 50px and a height of 50px.

console

# Mise à jour du DOM : Cloner un élément

- Afin de cloner un nœud dans le DOM il suffit d'appeler la méthode **cloneNode()**.
- Cette méthode prend en paramètre un booléen.
  - True : Le nœud sera cloner avec ces fils et ses attributs
  - False : Le nœud sera cloner sans ces fils et ses attributs

# Mise à jour du DOM : exemple clone

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Ma page de Test </title>
</head>
<body>

<p id="para">vous allez trouvez mon Cv <a
id="lelien" href="www.monCv.tn">ici</a>.
Veuillez le consulter et appeler moi au
<i>222222</i> pour plus de détails</p>

<div id="monMenu">

  <div id="item">
    <span>menu 1</span>
    <span>menu 2</span>
  </div>

  <div class="Pub">
    <span>Pub 1</span>
    <span>Pub 2</span>
  </div>

</div>
<script src="test9.js">
</script>
</body>
</html>
```

Element.html

```
var nod=document.getElementById("para");
var newNode=document.createElement("img");
newNode.id="asimg";
newNode.src="as.jpg";
newNode.alt="Je suis l'image de aymen";
newNode.width="50";
newNode.height="50";
nod.appendChild(newNode);
var trueClonedNode = nod.cloneNode(true);
var falseClonedNode = nod.cloneNode(false);
document.body.appendChild(trueClonedNode);
document.body.appendChild(falseClonedNode);
```

test9.js

# Mise à jour du DOM : exemple clone

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Ma page de Test </title>
</head>
<body>

<p id="para">vous allez trouvez mon Cv <a
id="lelien" href="www.monCv.tn">ici</a>.
Veuillez le consulter et appeler moi au
<i>222222</i> pour plus de détails</p>

<div id="monMenu">

  <div id="item">
    <span>menu 1</span>
    <span>menu 2</span>
  </div>

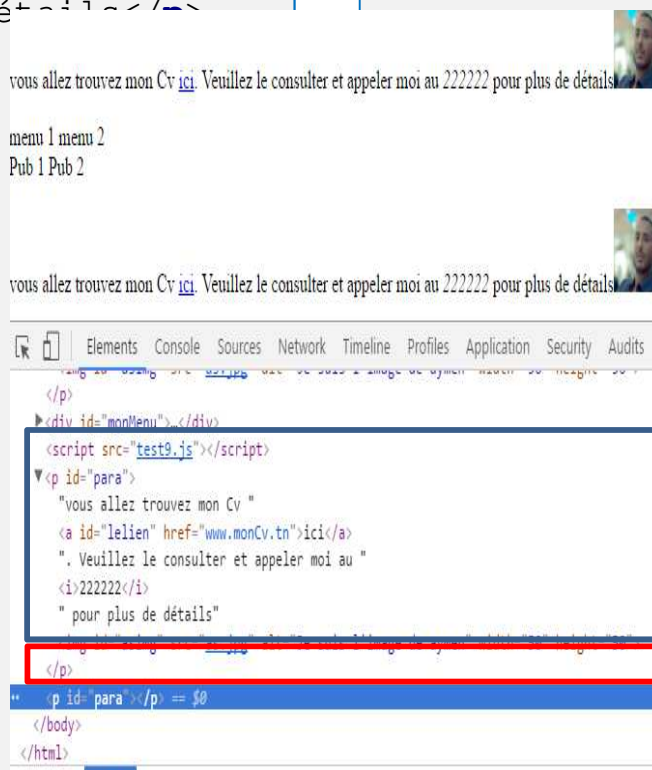
  <div class="Pub">
    <span>Pub 1</span>
    <span>Pub 2</span>
  </div>

</div>
<script src="test9.js">
</script>
</body>
</html>
```

Element.html

```
var nod=document.getElementById("para");
var newNode=document.createElement("img");
newNode.id="asimg";
newNode.src="as.jpg";
newNode.alt="Je suis l'image de aymen";
newNode.width="50";
newNode.height="50";
nod.appendChild(newNode);
var trueClonedNode = nod.cloneNode(true);
var falseClonedNode = nod.cloneNode(false);
document.body.appendChild(trueClonedNode);
document.body.appendChild(falseClonedNode);
```

test9.js



console

# Mise à jour du DOM : Supprimer un élément

- Afin de supprimer un nœud dans le DOM il suffit d'appeler la méthode **removeChild()** à partir du nœud père.
- Cette méthode prend en paramètre le nœud à supprimer.
- La valeur de retour est une référence sur le nœud supprimé.
- Comment faire pour supprimer le premier span d'un menu d'un div d'id item ?

# Mise à jour du DOM : exemple remove

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Ma page de Test
</title>
</head>
<body>

<p id="para">vous allez trouvez mon Cv
<a id="lelien"
href="www.monCv.tn">ici</a>. Veuillez le
consulter et appeler moi au
<i>222222</i> pour plus de détails</p>

<div id="monMenu">

  <div id="item">
    <span>menu 1</span>
    <span>menu 2</span>
  </div>

  <div class="Pub">
    <span>Pub 1</span>
    <span>Pub 2</span>
  </div>

</div>
<script src="test10.js">
</script>
</body>
</html>
```

Element.html

```
var nod=document.getElementById("item");
nod.removeChild(nod.firstChild);
```

test10.js

vous allez trouvez mon Cv [ici](#). Veuillez le consulter et appeler moi au 222222 pour plus de détails

menu 2  
Pub 1 Pub 2



console

# Mise à jour du DOM : remplacer un élément

- Afin de remplacer un nœud dans le DOM par un autre il suffit d'appeler la méthode **replaceChild()** partir du nœud père.
- Cette méthode prend en paramètre le nouveau nœud suivi de l'ancien nœud.
- Comment faire pour remplacer le second span par la div pub?

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Ma page de Test </title>
</head>
<body>
<p id="para">vous allez trouvez mon Cv <a id="lelien" href="www.monCv.tn">ici</a>. Veuillez le consulter
et appeler moi au <i>222222</i> pour plus de détails</p>
<div id="monMenu">
  <div id="item">
    <span>menu 1</span>
    <span>menu 2</span>
  </div>
  <div class="Pub">
    <span>Pub 1</span>
    <span>Pub 2</span>
  </div>
</div>
<script src="test10.js">
</script>
</body>
</html>
```

# Mise à jour du DOM : exemple replace

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Ma page de Test
</title>
</head>
<body>

<p id="para">vous allez trouvez mon Cv
<a id="lelien"
href="www.monCv.tn">ici</a>. Veuillez le
consulter et appeler moi au
<i>222222</i> pour plus de détails</p>

<div id="monMenu">

  <div id="item">
    <span>menu 1</span>
    <span>menu 2</span>
  </div>

  <div class="Pub">
    <span>Pub 1</span>
    <span>Pub 2</span>
  </div>

</div>
<script src="test11.js">
</script>
</body>
</html>
```

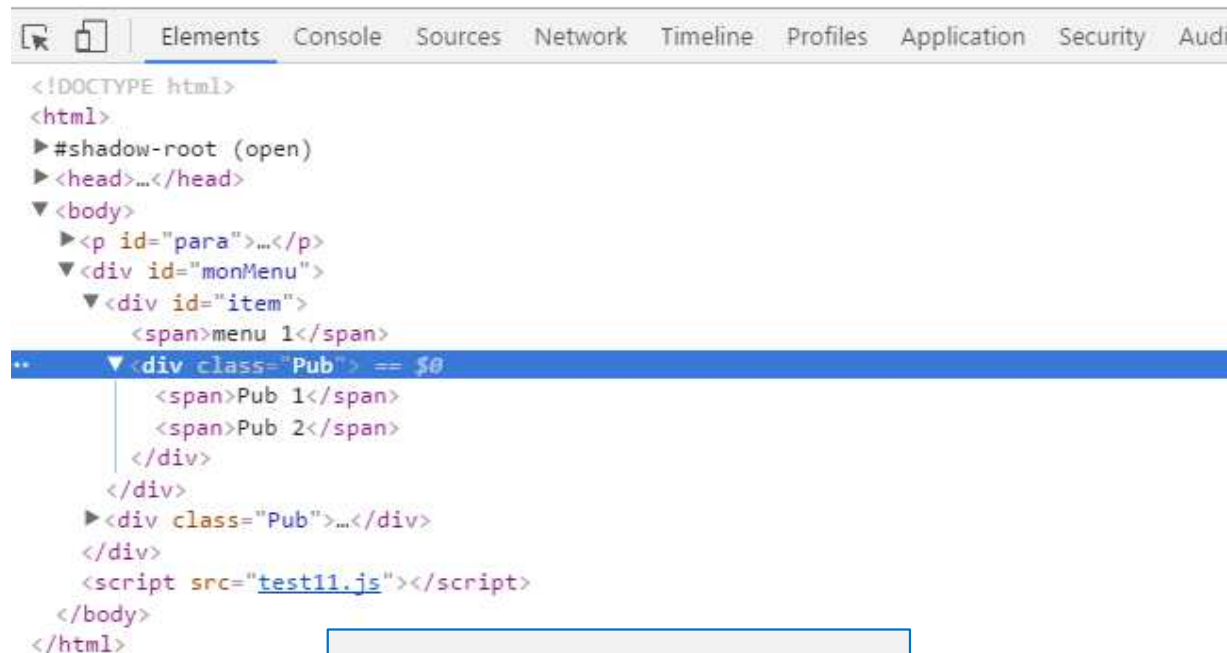
Element.html

```
var nod=document.getElementById("item");
var
newnod=document.getElementsByClassName("Pub")[0].cloneNode(true);
nod.replaceChild(newnod,nod.lastElementChild);
```

test11.js

vous allez trouvez mon Cv [ici](#). Veuillez le consulter et appeler moi au 222222 pour plus de détails

menu 1  
Pub 1 Pub 2  
Pub 1 Pub 2



console



# Manipulation du CSS

- JavaScript permet aussi de manipuler et de gérer l'apparence des éléments HTML.
- Pour ce faire, il offre une panoplie d'outils permettant de modifier le CSS.
- La base de la manipulation des CSS par JavaScript est l'aspect **CASCADE** qui indique que les **règles de styles** qui sont appliquées à un élément du document HTML peuvent venir d'une **cascade de différentes sources** dont la **plus prioritaire** est l'attribut style d'un élément HTML individuel.
- Style étant la propriété la plus prioritaire, c'est elle qui sera la plus utilisée.

# Manipulation du CSS : La propriété style

- Pour accéder à la propriété style :
  - NotreElement.**style**
- Pour ajouter ou modifier une des propriétés de style
  - NotreElement.**style.propriété**

```
...
<div id="monMenu">
  <div id="item">
    <span>
      menu 1
    </span>
    <span>
      menu 2
    </span>
  </div>
  <div class="Pub">
    <span>Pub 1</span>
    <span>Pub 2</span>
  </div>
</div>
<script src="test12.js">
</script>
</body>
</html>
```

Element.html

```
var nod=document.getElementById("item");
nod.style.backgroundColor="blue";
```

test12.js

vous allez trouvez mon Cv [ici](#). Veuillez le consulter et appeler moi au 222222 pour plus de détails

menu 1 menu 2  
Pub 1 Pub 2

[Un lien bidon](#)

Elements Console Sources Network Timeline Profiles Application Security Audits AdBlock

```
<!DOCTYPE html>
<html>
  <#shadow-root (open)>
    <head>...</head>
    <body>
      <p id="para">...</p>
      <div id="monMenu">
        <div id="item" style="background-color: blue;">...</div> == $0
        <div class="Pub">...</div>
      </div>
      <input id="input" type="text" size="50" value="Cliquez ici !" onfocus="this.value='Appuyez maintenant sur votre touche de
```

- Pour afficher une propriété ?

# Manipulation du CSS : `getComputedStyle`

- Pour accéder à une des propriétés des feuilles de styles il faut utiliser la méthode **`getComputedStyle()`**
- Cette méthode récupère les style CSS associés à un Element HTML qu'elle soient dans l'attribut style, dans un bloc style ou dans une feuille de style.
- La valeur de retour est un objet contenant l'ensemble des styles, donc pour accéder à un style en particulier il faut utiliser le nom de cette propriété.

# Manipulation du CSS : getComputedStyle

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8" />
  <title>Ma page de Test </title>
  <link href="css/test.css"
rel="stylesheet" type="text/css">
</head>
<body>
<p id="para">vous allez trouvez mon Cv
<a id="lelien"
href="www.monCv.tn">ici</a>. Veuillez le
consulter et appeler moi au
<i>222222</i> pour plus de détails</p>
<div id="monMenu">
  <div id="item">
    <span>menu 1</span>
    <span>menu 2</span>
  </div>
  <div class="Pub">
    <span>Pub 1</span>
    <span>Pub 2</span>
  </div>
</div>
<input id="input" type="text" size="50"
value="Cliquez ici !"
onfocus="this.value='Appuyez maintenant
sur votre touche de tabulation.';"
onblur="this.value='Cliquez ici !';"/>
<br /><br />
<a href="#"
onfocus="document.getElementById('input'
).value = 'Vous avez maintenant le focus
sur le lien, bravo !';">Un lien
bidon</a>
<script src="test13.js"></script>
</body>
</html>
```

Element.html

```
var nod=document.getElementById("item");
console.log("je suis l'objet " + nod.nodeName + "voila
mes attributs : ");
var mesStyles=getComputedStyle(nod);
console.log("ma taille est : "+mesStyles.width+" et "+
mesStyles.height);
console.log("La couleur de mon arrière plan est :
"+mesStyles.backgroundColor+" et la couleur de
l'écriture est "+ mesStyles.color);
for(myStyle in mesStyles){
  console.log(myStyle);
  console.log(mesStyles[myStyle]);
}
```

test13.js

```
#item{
  background-color: yellow;
  color: red;
  width: 200px;
  height: 200px;
}
```

Test.css

# Manipulation du CSS : getComputedStyle

menu 1 menu 2

Pub 1 Pub 2

Cliquez ici !

[Un lien bidon](#)

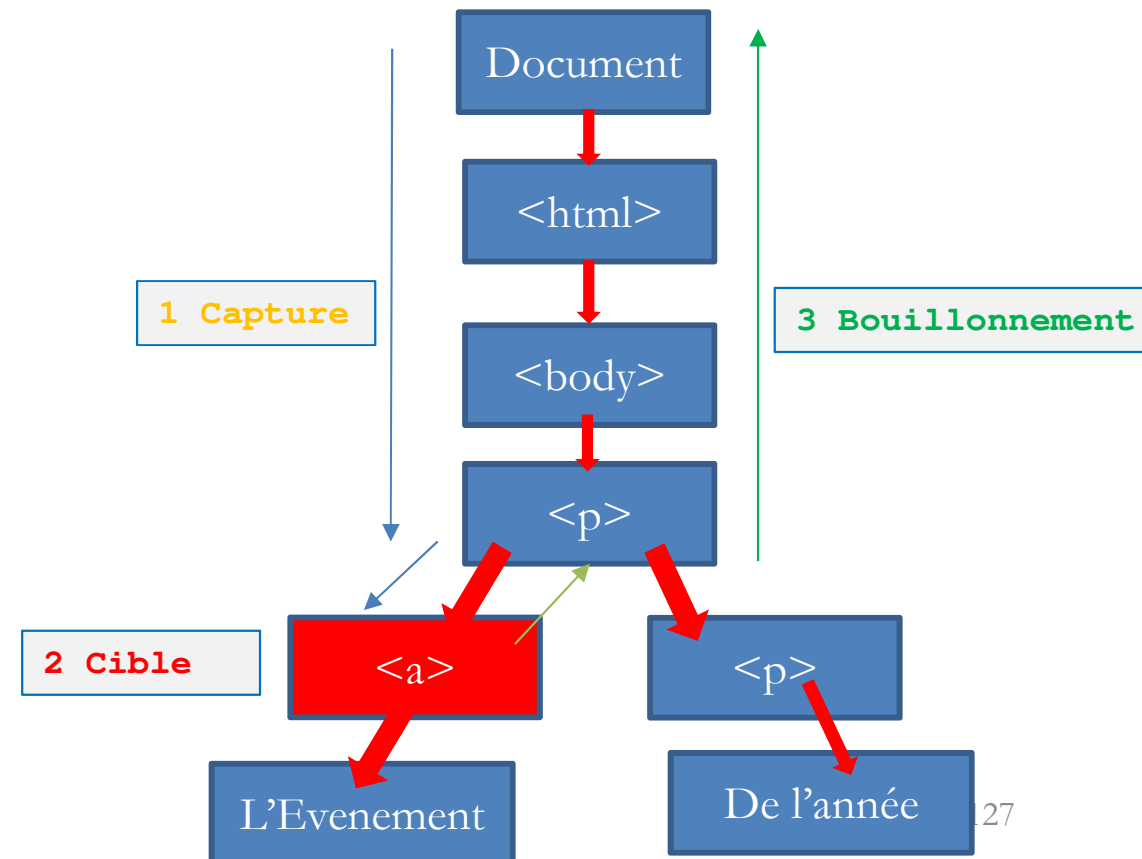
Filter	Regex	Hide network messages	All	Errors	Warnings	Info	Logs	Debug	Handled
je suis l'objet DIVvoilà mes attributs :									
ma taille est : 200px et 200px									
La couleur de mon arrière plan est : rgb(255, 255, 0) et la couleur de l'écriture est rgb(255, 0, 0)									
0									
animation-delay									
1									
animation-direction									
2									
animation-duration									
3									
animation-fill-mode									
4									
animation-iteration-count									
5									
animation-name									
6									
animation-play-state									
7									
animation-timing-function									
8									
background-attachment									

- Un événement dans notre contexte est un changement d'état d'un des éléments du DOM :
  - Lorsqu'on click sur un élément, qu'on le survole ou qu'on écrit quelque chose dedans
  - Lorsqu'on on submit un formulaire
  - ...
- L'utilisation des événements consiste à déclencher un traitement particulier lors de la détection d'un événement.
- Le navigateur intercepte les événements (interruptions) et agit en conséquence
- Une fois l'événement crée, il se propage dans le DOM (l'arbre) en se dirigeant de la racine vers le nœud cible puis en effectuant le sens inverse. Les trois étape par lequel passe l'événement sont :
  - Phase de capture
  - Phase de ciblage (l'événement atteint la cible)
  - Phase de bouillonnement (bubbling)

# Gestion des événements

- Lors de l'activation du lien, un événement **click** va être créé.
- L'événement entame sa **propagation (phase de capture)** du **document** vers le nœud qui précède la cible qui est ici le nœud **a**.
- L'événement **atteint sa cible** (le nœud **a**)
- L'événement entame sa propagation vers le document (**phase de bouillonnement**)

```
<html>
<body>
  <p>
    <a href="http://www.JS.com/">
      L' Evenement
    </a> de l'année
  </p>
</body>
</html>
```



# Gestion des événements

Liste **non exhaustive** des événements les plus utilisées :

- **click** : un clic du bouton gauche de la souris sur une cible
- **dblclick** : undouble clic du bouton gauche de la souris sur une cible
- **mouseover** : passage du pointeur de la souris sur une cible
- **mouseout** : sortie du pointeur de la souris d'une cible
- **focus** : une activation d'une cible
- **blur** : une perte de focus d'une cible
- **select** : sélection d'une cible
- **change** : une modification du contenu d'une cible
- **Input** : Saisir du texte dans un champ texte
- **submit** : une soumission d'un formulaire
- **reset** : réinitialiser les éléments du formulaire
- **load** : à la fin du chargement d'un élément



# Gestion des évènements : Utilisation

L'interception des évènements et l'actionnement d'une action suite à cette interception se fait de deux manières :

- Associer à un élément html un attribut **on**event et y affecter le code JavaScript à exécuter.
  - Exemple **onclick**, **onblur**, **onchange**.
  - `<span onclick = "alert('Salut, tu as cliqué sur ce span et moi j ai intercepté ton click :D');">Cliquez-moi !</span>`
- En utilisant le DOM
  - En utilisant l'ancienne méthode (DOM-0), ceci est fait en affectant à l'attribut **on**event de l'élément à traiter la fonction à exécuter.
  - En utilisant le DOM-2 et la méthode **addEventListener()**

- La détection des événement sera réalisée à travers la méthode **addEventListener()**
- Cette méthode prend **2 paramètres obligatoires** et un **optionnel**.
  - Nom de l'événement (click, blur, ...)
  - La fonction qui s'exécutera
  - Un booléen (False par défaut) qui permet de spécifier quelle phase utiliser
    - False : Phase de bouillonnement y inclut la cible
    - True : Phase de capture
- Dans le cas d'une Phase de bouillonnement l'élément le plus profond est le premier à s'exécuter. C'est la phase la plus utilisé.
- Dans le cas d'une Phase de capture l'élément le plus profond est le dernier à s'exécuter.

# Gestion des évènements : L'objet Event

- L'objet Event permet de récupérer des information sur l'événement déclenché
- N'est récupérable que dans une fonction associé à un événement
  - `Nod.addEventListener('click',f(e){alert( e );})`
- Tester ce code et vérifier le contenu de l'objet event dans la console

```
var nod=document.getElementById("item");

nod.addEventListener('click',function (e) {
    console.log(e);
    console.log('Bonjour je suis l\'objet'+
    this.nodeName+'et mon contenu est : '+this.innerHTML);
});
```

- L'une des méthodes la plus utilisée est **preventDefault()** qui permet d'annuler l'événement par exemple bloquer l'envoi d'un formulaire
- Maintenant et selon votre besoin utiliser et étudier ces propriétés.

# Introduction

- Sortie en 2015
- Noms : ES6 – ECMAScript6 – **ECMAScript2015** – Harmony
- Plusieurs nouveautés :
  - Arrow Functions

# Les fonctions fléchées ( Arrow Functions)

- Sortie en 2015
- Noms : ES6 – ECMAScript6 – **ECMAScript2015** – Harmony
- Plusieurs nouveautés :
  - Arrow Functions

- M. Haverbeke, Eloquent JavaScript: A modern introduction to programming, 2014
- D. Flanagan, *JavaScript: The Definitive Guide*, 6<sup>th</sup> edition, O'reilly, 2011
- Julien CROUZET, EcmaScript 6
- Olivier Hondermarck, Tout JavaScript