

Cursus SALESFORCE

M2I - Formation - 2023

Donjon Audrey

Javascript - Fondamentaux

- 1. Bases du langage, fonctions, arrays et objets**
- 2. Le Dom et les écouteurs d'évènements**
- 3. Gérer les CSS depuis JS / Ajax**



1. Bases du langage, fonctions, arrays et objets

1.1) Introduction à Javascript

- A) IDE
- B) ECMAScript
- C) Où placer JS

1.2) Les bases du langage

- A) Variables
- B) Conditions
- C) Boucles

1.3) Les fonctions js

- A) Déclaration
- B) Fonctions natives
- C) Fonctions anonymes
- D) Fonctions auto-invoqués

1.4) Les arrays

- A) Rappel et syntaxe
- B) Fonctions utiles pour les arrays
- D) Récupérer et parcourir un array

1.5) Les objets

- A) Les objets Js
- B) L'objet this
- C) L'objet Window
- D) L'objet String et Number
- E) Introduction à la POO

1.1 Introduction à Javascript

A) IDE

B) ECMAScript

C) Où placer Js

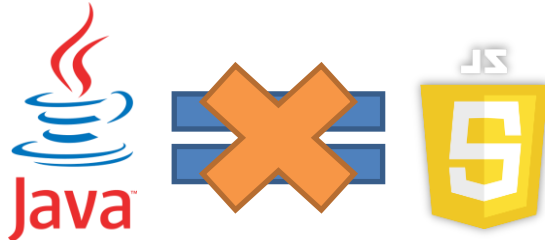
Rappel :

Le langage **HTML** sert à **structurer** les contenus d'une page Web.

Le langage **CSS** à **habiller** ces contenus.

Et le langage **Javascript** lui est un **langage** de **script** qui sert à **modifier dynamiquement** les éléments HTML/CSS de la page Web sans la recharger, en amenant une dimension d'interactivité avec l'utilisateur et le navigateur.

Attention : **JAVA** n'est pas **Javascript** (dit **JS** et **non Java** pour diminutif) :



Ide pour Integrated **D**evelopment **E**nvironment (**E**nvironnement de **D**éveloppement **I**ntégré), il s'agit d'un logiciel qui permet **d'écrire du code** avec tout un tas **d'outils embarqués** pour aider le dev à être plus **rapide** et **efficace** (comme la vérification de la syntaxe du langage, la détection des erreurs, l'auto-complétion, un compilateur, un serveur etc).

Attention à la différence d'un **éditeur de code/texte avancé** qui lui sera plus **léger**, les outils seront à rajouter en fonction des besoins grâce aux nombreux **plugins** disponible donc pour être efficace il faudra passer du temps à le **configurer** avant de se lancer.

Toutefois vous entendrez souvent le terme **IDE** employé pour parler d'un éditeur de code ou d'un environnement de développement intégré car les éditeur de code sont de plus en plus amélioré même ci ceux-ci ne sont pas aussi développer qu'un véritable IDE.

Environnements de développement et éditeurs de codes avancés

Éditeurs de code avancé

[Visual Studio Code](#)



[Sublime Text](#)



[Brackets](#)



[Notepad++](#)



IDE

[PhpStorm](#)



[WebStorm](#)



[Komodo IDE](#)



[NetBeans](#)

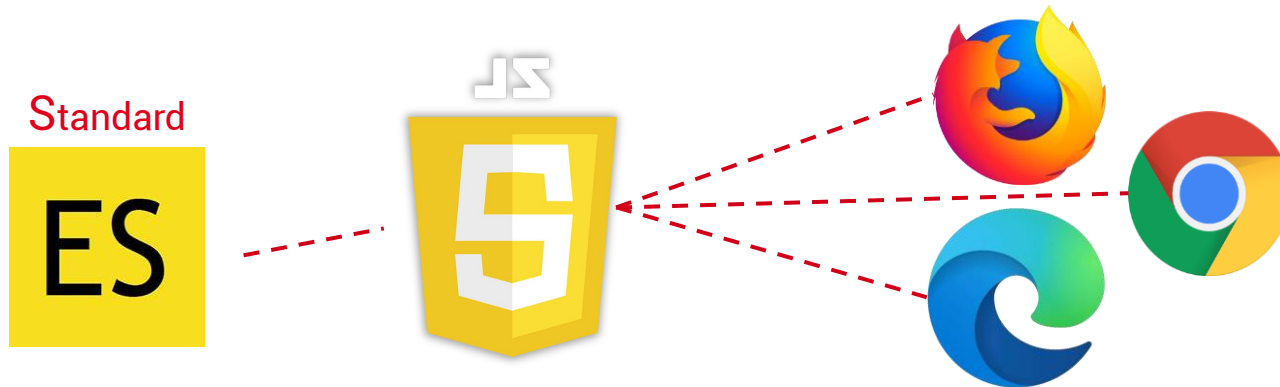


ECMAScript

Il s'agit d'un **ensemble** de **normes** concernant les langages de programmation de type script et standardisé par **Ecma International**.

Il s'agit d'un **standard**, dont les spécifications sont mises en œuvre dans **différents langages de script**, notamment **Javascript**.


C'est **ECMAScript** qui définit donc comment **JS** doit fonctionner et tous les **navigateurs Web** doivent respecter ses spécifications afin que le langage fonctionne de partout pareil.



Où placer le Javascript

Écrit dans 3 endroits possibles :

1 / Dans un fichier JS (script.js) inclut dans le code HTML (le plus utilisé car mis en cache par le navigateur)



JS	<pre><!-- script.js--> alert('Fichier JS chargé !');</pre>
----	--

HTML	<pre><!-- index.html --> <!DOCTYPE html> <html lang="fr"> <head> <meta charset="UTF-8"> </head> <body> <!-- contenu de ma page--> <script src="js/script.js"></script> </body> </html></pre>
------	--

2 / Dans une balise script directement (à éviter car pas mis en cache par le navigateur)

html	<pre><!-- index.html --> <body> <script> alert('Code JS chargé !'); </script> </body></pre>
------	---

3 / Dans la console JS du navigateur (Ctrl + Maj + K) : Utilisé pour tester du code directement dans le navigateur

1.2 Les bases du langage

- A) Variables**
- B) Conditions**
- C) Boucles**
- D) Exercice**

Les variables

Pour rappel une **variable** est composée de :

- Un **nom** (par exemple "age")
- Un **type** (par exemple "number")
- Une **valeur** (par exemple "36")



Syntaxe

Pour créer une nouvelle variable, on doit utiliser le mot "**let**" :

```
JS // Crée une nouvelle variable et lui donne une valeur  
let age = 36;
```

Le type de la variable est implicite : 36 est un type "number"

Modification

```
JS let age = 36;  
// Change la valeur  
age = 25;
```

Nom d'une variable

Le choix **du nom d'une variable** est **libre** mais doit **respecter** les **règles** suivantes :

- Le nom d'une variable ne peut être composé que des symboles suivants : **a-z A-Z 0-9 _ \$**
- Le premier caractère ne doit **pas être un chiffre**.
- Le nom doit être toujours **en anglais** (convention)
- Le nom doit toujours **décrire le plus explicitement son contenu** (par exemple : `userCity`)
- les **mots-clés JS sont interdits** (comme "let" par exemple)
- les variables doivent être **écrites en lower camel case** (c'est-à-dire une **majuscule à chaque mot sauf le premier**. Exemple : `jeSuisUneVariableDeTest`)

Attention: le nom des variables est sensible à la casse (aux majuscules). Par exemple, "username" et "userName" sont deux variables différentes.

Les variables : rappel sur Les types

Types de contenus

Le type d'une variable définit la nature de l'information stockée dans une variable. Liste de quelques types :

<p>string : chaîne de caractères (avec des guillemets simples, doubles ou obliques autour de la valeur).</p>	<pre>Js let city = 'Paris'; let firstname = "Alice"; let lastname = `Durand`; // Attention si vous avez besoin de stocker un guillemet dans la // chaîne par exemple à bien l'échapper avec un antislash devant : let sentence1 = 'Le chat grimpe dans l\'arbre'; let sentence2 = "Ceci est une \"citation\"";</pre>
<p>number : nombres (sans rien autour de la valeur).</p> <ul style="list-style-type: none"> ➤ La valeur "Infinity" est une valeur particulière de type "number" qui veut dire l'infinie. ➤ La valeur "NaN" est une valeur particulière de type "number" qui veut dire "pas un nombre". 	<pre>js let age = 36; let money = 125.56;</pre>
<p>boolean : valeur booléenne "vrai" ou "faux" (soit true, soit false, sans rien autour de la valeur).</p>	<pre>js let isAdmin = true; let isModerator = false;</pre>

<pre>JS // On peut tester le type d'une variable avec typeof let name = 'Alice'; // Affichera "string" dans la console console.log(typeof name);</pre>	
---	--

Opérateurs arithmétiques et concaténation

Opérateurs arithmétiques

Comme tous les langages de programmation, JS gère les calculs mathématiques. On peut ainsi utiliser les opérateurs arithmétiques classiques :

```
JS    let number = 5;

      // Addition
      console.log( number + 5 ); // Affichera 10

      // Soustraction
      console.log( number - 20 ); // Affichera -15

      // Multiplication
      console.log( number * 10 ); // Affichera 50

      // Division
      console.log( number / 2 ); // Affichera 2.5
```

Concaténation

La concaténation est l'action de "coller" deux chaînes de texte ensemble :

```
JS    let name = 'Alice';

      // Affichera "Bonjour Alice !"
      console.log('Bonjour ' + name + ' !');

      // Possible de faire la même chose différemment UNIQUEMENT avec les guillemets obliques `` autour :
      console.log(`Bonjour ${name} !`);
```

Opérateurs d'affectation

Opérateurs d'affectation

Les opérateurs d'affectation permettent d'assigner une nouvelle valeur à une variable. Le plus simple étant le signe "=". Il en existe d'autres permettant de combiner plusieurs opérateurs :

JS

```
let number;

// Affectation simple de valeur
number = 25;

// Double action : calcul de la valeur actuelle de la variable + 5 ET affectation de la nouvelle valeur dans
la variable (number vaud maintenant 30, résultat de 25 + 5)
number += 5;    // 30

// Double action : calcul de la valeur actuelle de la variable - 10 ET affectation de la nouvelle valeur dans
la variable (number vaud maintenant 20, résultat de 30 - 10)
number -= 10;   // 20

let number;

// Double action : calcul de la valeur actuelle de la variable * 3 ET affectation de la nouvelle valeur dans
la variable (number vaud maintenant 60, résultat de 20 * 3)
number *= 3;    // 60

// Double action : calcul de la valeur actuelle de la variable / 4 ET affectation de la nouvelle valeur dans
la variable (number vaud maintenant 15, résultat de 60 / 4)
number /= 4;    // 15

// Double action : calcul de la valeur actuelle de la variable % 4 ET affectation de la nouvelle valeur dans
la variable (number vaud maintenant 3, résultat du modulo de 15 par 4)
number %= 4;    // 3
```

Incrémentation et décrémentation

L'incrémentation est l'action de **changer la valeur d'une variable pour lui ajouter 1** (passer de 22 à 23 par exemple). Il existe la même chose dans l'autre sens, c'est-à-dire **enlever 1 : la décrémentation**.

L'incrémentation

JS

```
let number = 20;  
// Incrémentation  
number++;  
  
// Affichera "21"  
console.log( number );
```

la décrémentation

JS

```
let number = 10;  
// Décrémentation  
number--;  
  
// Affichera "9"  
console.log( number );
```

Changer le type d'une variable

Parfois une variable peut être d'un type qui ne nous convient pas. Il est possible grâce à certaines fonctions de "transtyper" une **variable**, c'est-à-dire **changer son type tout en préservant autant que possible sa valeur**.

Dans l'exemple suivant, on récupère l'âge d'un utilisateur via la fonction "prompt" pour effectuer dessus un petit calcul ->

Le problème ici c'est que **prompt renvoi toujours la valeur récupérée sous la forme d'une "string"**. Quand on essaie donc d'ajouter 5 à la valeur "25", il en résulte une **concaténation** ("25" + 5 = 255) au lieu d'une **addition**, car pour JS "25" est une chaîne de texte.

JS

```
let userAge = prompt('Quel est votre âge ?');

userAge += 5;

// Ici si l'utilisateur entre "25", le résultat sera : "Dans 5 ans, vous aurez 255 ans !"
alert('Dans 5 ans, vous aurez ' + userAge + ' ans !');
```

Pour résoudre le problème, il faudrait donc que l'âge récupéré soit **"converti"** en **type "number"** au lieu d'être une **"string"** ->

JS

```
let userAge = prompt('Quel est votre âge ?');

// On "force" la variable à être un entier de type "number"
userAge = parseInt( userAge );

userAge += 5;

// Ici si l'utilisateur entre "25", le résultat sera : "Dans 5 ans, vous aurez 30 ans !"
alert('Dans 5 ans, vous aurez ' + userAge + ' ans !');
```

Quelques méthodes pour changer de type :

JS

```
// Force à être de type "number" et entier
console.log( parseInt("25.36") ); // Affichera 25

// Force à être de type "number"
console.log( parseFloat("25.36") ); // Affichera 25.36

// Attention, si parseInt et parseFloat ne contiennent pas de nombre valide, le résultat sera NaN
console.log( parseInt("Je suis pas un nombre !") ); // Affichera NaN

// Force à être de type "string"
console.log( String(56) ); // Affichera "56" (en chaîne de texte)
```


Les constantes

Les **constantes** sont comme des variables (**informations stockées** dans la mémoire **avec un type et une valeur**) mais contrairement à ces dernières, **une fois créées elles ne peuvent plus changer de valeur** :

JS

```
// Création d'une constante
const name = 'Alice';

// Affiche "Alice"
console.log( name );

// Erreur, une constante ne peut pas être modifiée
name = 'Bob';
```

L'utilité des **constantes** réside dans le fait que leur valeur est inaltérable : c'est **une garantie supplémentaire** que **cette valeur ne pourra pas être changée dans le script**.

Les conditions

Syntaxe :

```
JS // l'expression 5 plus petit que 10 est vrai, donc la condition sera bien lue
if(5 < 10){
    // Sera bien lu
    alert('Bonjour !');
}

let age = 5;
// l'expression "age" plus grand ou égal que 18 est fausse car la variable contient "5",
// donc tout le bloc de la condition sera ignoré
if(age >= 18){
    // Ne sera pas lu
    alert('Bienvenue sur notre site !');
}
```

Une **condition** est une **structure de code** et non une instruction à proprement parler, il ne faut donc **pas mettre de point-virgule** à la fin d'un if ou d'une accolade.

Opérateurs de comparaison

Pour construire les conditions, il faut utiliser des **opérateurs de comparaison** :

Opérateur « valeur plus petite que » :	<pre> JS let test = 50; // Condition vraie car 50 est bien plus petit que 60 if(test < 60){ // ... } </pre>
Opérateur « valeur plus petite ou égale à » :	<pre> JS // Condition vraie car 50 est bien plus petit ou égal à 80 if(test <= 80){ // ... } </pre>
Opérateur « valeur plus grande que » :	<pre> JS // Condition vraie car 50 est bien plus grand que 30 if(test > 30){ // ... } </pre>
Opérateur « valeur plus grande ou égale à » :	<pre> JS let test = 50; // Condition vraie car 50 est bien plus grand ou égal à 45 if(test >= 45){ // ... } </pre>
Opérateur « valeur égale à » (vérifie uniquement la valeur, pas le type !)	<pre> JS // Condition vraie car 50 est bien égal à 50 if(test == 50){ // ... } // Condition vraie aussi car 50 et "50" ont bien la même valeur (même si le type est différent !) if(test == "50"){ // ... } </pre>
Opérateur « valeur différente de » :	<pre> JS // Condition vraie car 50 est bien différent de 0 if(test != 0){ // ... } </pre>

Comparaison valeur et/ou type

Opérateur "**valeur ET type égaux à**"

(vérifie la valeur mais aussi le type cette fois !):

JS

```
let test = 50;
// Condition vraie car 50 est bien égal à 50 et de même type (number en l'occurrence)
if( test === 50 ){
    // ...
}

// Condition fausse car 50 et "50" ont bien la même valeur mais ils sont d'un type différent ! ("50" = string alors que 50 = number)
if( test === "50" ){
    // ...
}
```

Opérateur "**valeur OU type différent de**"

(Il suffit que la valeur ou le type soit différent pour que la condition soit vraie):

JS

```
let test = 50;
// Condition vraie car 50 et "50" ont un type différent
if( test !== "50" ){
    // ...
}

// Condition vraie car 50 et 52 ont une valeur différente
if( test !== 52 ){
    // ...
}
```

Opérateurs logiques

Les opérateurs **logiques** servent à combiner **plusieurs "tests" dans une condition** (sauf pour "NO") :

<p>Opérateur logique "&&" (AND) : Pour que la condition soit vraie, il faut que les deux tests soient vrais.</p>	<pre>JS let age = 25; // Condition vrai car les 2 tests sont satisfaits : 25 est bien plus grand ou égal à 20 et il est également plus petit que 30 if(age >= 20 && age < 30){ alert('Bienvenue sur ce site réservé aux personnes ayant la vingtaine !'); }</pre>
<p>Opérateur logique " " (OU) : Pour que la condition soit vraie, il faut qu'au moins un des deux tests soit vrai.</p>	<pre>JS let temperature = 60; // Condition vrai car "temperature" est plus grande que 40 if(temperature < 0 temperature > 40){ alert('La température est dangereuse !'); }</pre>
<p>Opérateur logique "!" (NO) : Inverse le sens d'un booléen. Cet opérateur est très utilisé pour inverser le sens d'une fonction ou d'une variable.</p>	<pre>JS // Utilisateur non autorisé (c'est un exemple purement démonstratif) let authorizedUser = false; // Dans cette situation, on souhaite entrer dans la condition si la variable contient false. On peut donc inverser le sens avec l'opérateur "!" if(!authorizedUser){ alert('Vous n\'êtes pas un utilisateur autorisé !'); }</pre>



Structures conditionnelles

Structure else

Dans le cas où une **condition** est **fausse**

JS

```
let temperature = 30;

// Condition fausse car "temperature" n'est pas plus petite que 0, ni plus grande que 40 : c'est donc le bloc "else" qui sera exécuté
if(temperature < 0 || temperature > 40){
  alert('La température est dangereuse !');
} else {
  alert('La température est bonne !');
}
```

Structure else if

Pour **imbriquer** d'autres conditions dans la **même structure** conditionnelle

JS

```
let city = 'Lyon';
if(city == 'Paris'){
  // Faux, passe au if suivant
  alert('Bonjour à toi le Parisien !');
} else if(city == 'Lyon'){
  // vrai
  alert('Bonjour à toi le Lyonnais !');
} else if(city == 'Marseille'){
  // Pas lu
  alert('Bonjour à toi le Marseillais !');
} else {
  // Pas lu
  alert('Désolé je ne connais pas ta ville');
}
```

Conditions ternaires

Syntaxe : **condition** ? **exprSiVrai** : **exprSiFaux**

Une **condition ternaire** : **manière très raccourcie** d'écrire une **condition** pour affecter une valeur.

JS

```
let temperature = 30;
// Code classique
if(temperature < 15){
  alert('Il fait froid !');
} else {
  alert('Il fait chaud !');
}

// Même chose avec une ternaire
alert( (temperature < 15) ? 'Il fait froid !' : 'Il fait chaud !' );
```

Boucle While et For

La boucle **while** est la boucle la plus **classique**, qui **s'exécutera tant que sa condition sera vraie**.

Pour éviter de faire une boucle infinie, on utilise souvent une variable qui servira à compter combien de fois la boucle s'est déjà exécutée. Par convention cette variable s'appelle "i" (car c'est un itérateur qui s'incrémente de 1 à chaque tour de boucle)

JS

```
// Itérateur à 0 au début
let i = 0;

// La boucle s'exécutera tant que i sera plus petit que 10
while(i < 10){
    alert('Je suis une boucle !');

    // Très important pour augmenter le compteur, sinon boucle infinie !
    i++;
}
```

La boucle **for** est **une boucle avec un itérateur intégré** directement dans sa structure (même chose que while sinon)

JS

```
// La boucle s'exécutera tant que i sera plus petit que 10
for(let i = 0; i < 10; i++){

    alert('Je suis une boucle !');
}
```

Si dans une boucle pour une raison ou une autre vous souhaitez **arrêter la boucle avant la fin**, vous pouvez avec **"break"**, ce qui permettra de **sortir immédiatement de la boucle**

JS

```
// Boucle infinie (mais contrôlée car prompt met le code en pause à chaque tour, donc pas de risque de plantage !)
while( true ){

    // On demande le mot de passe à l'utilisateur
    let userAttempt = prompt('Quel est le mot de passe ?');

    // Si le mot de passe entré est "azerty", c'est bon
    if(userAttempt == 'azerty'){

        alert('C\'est le bon mot de passe !');

        // Stop la boucle pour qu'elle ne continue pas
        break;
    }
    alert('Mauvais mot de passe, veuillez ré-essayer.');
```

Exercice bases

Énoncé :

Définissez une **variable** contenant **une adresse mail**, une **variable** contenant **un mot de passe**.

Demander à l'utilisateur de rentrer une première valeur qui sera l'adresse mail, puis **demandeur** lui le mot de passe.

Si le mail et le mot de passe **ne corresponde pas** à ce que vous aviez défini dans vos variables **continuez à demander** le **mail** et le **mot de passe** à l'utilisateur sinon afficher un message d'alert « Bienvenue dans votre espace »



1.3 Fonctions

A) Déclaration

B) Fonctions natives

C) Fonctions anonymes

**D) Fonctions auto-
invoquées**

E) Exercice

Les Fonctions

Rappel

Une fonction est un **regroupement d'instructions** qui **produit un résultat**. Elle permet de faire appel à ces instructions sans être obligé de les retaper à chaque fois.

Pour **appeler une fonction**, on utilise son **nom** suivi d'une **paire de parenthèse** :

```
JS
```

```
// Invocation de la fonction alert  
alert();
```

➤ Il existe déjà de base dans JS plein de **fonctions natives** comme "**alert()**", "**confirm()**" ou "**parseInt()**" par exemple.

Certaines fonctions peuvent accepter des **valeurs entre leurs parenthèses** : on appelle ces valeurs des **arguments** :

```
JS
```

```
// Invocation de la fonction alert avec une chaîne de texte en argument  
alert('Bonjour !');
```

Quelques fonctions natives

Fonction console.log

La fonction "**console.log**" permet d'afficher du texte dans la console du navigateur (utilisé pour déboguer le code, faire des tests) :

```
js console.log('Cette instruction fonctionne !');
```

Fonction alert

La fonction "**alert**" permet d'afficher du texte dans la page web, dans une fenêtre modale basique du navigateur (sert à afficher un message à l'utilisateur) :

```
js alert('Cette instruction fonctionne !');
```

Fonction confirm

La fonction "**confirm**" permet de demander une confirmation à l'utilisateur (elle renverra "true" si l'utilisateur clique sur "oui", sinon "false") :

```
js let result = confirm('Êtes-vous sûr de vouloir faire cette action ?');
```

Fonction prompt

La fonction "**prompt**" permet de demander à l'utilisateur de rentrer du texte dans une boîte de dialogue (le résultat récupéré sera forcément du type "string") :

```
js let answer = prompt('Quel est votre nom ?');
```

Créer une fonction

Un développeur aura toujours besoin de créer des fonctions sur mesure pour remplir des tâches que les fonctions natives de Javascript ne savent pas faire. **Pour créer une fonction il faut la déclarer :**

```
JS // Déclaration de la nouvelle fonction
function sayHello(){

    // Ici le corps de la fonction, c'est-à-dire le code qui la compose
    alert('Bonjour !');

}

// On peut maintenant utiliser cette nouvelle fonction en l'invoquant
sayHello();
```

- Le nom d'une fonction doit être écrit **en lower camel case** (c'est-à-dire **une majuscule à chaque mot sauf le premier**). Exemple : encore**U**ne**F**onction())

Paramètres de fonction

Les paramètres de fonction vont permettre de **faire rentrer des valeurs dans la fonction** au moment où cette dernière est appelée.

```
JS // La fonction pourra récupérer un prénom en argument grâce au paramètre "name"
function sayHello(name){

    // On peut utiliser la variable "name" dans la fonction comme on le souhaite
    alert('Bonjour ' + name + ' !');

}

// Dire bonjour à Alice
sayHello('Alice');

// Dire bonjour à Bob
sayHello('Bob');
```

Valeur par défaut à un paramètre

Un **paramètre** de fonction peut avoir une **valeur par défaut** si vous le souhaitez :

JS

```
// Le paramètre "name" contiendra "John Doe" si jamais ce dernier n'est pas  
rempli lors de l'appel de la fonction  
function sayHello(name = 'John Doe'){  
  
    alert('Bonjour ' + name + ' !');  
  
}  
  
// Dire bonjour à Alice  
sayHello('Alice');  
  
// Dire bonjour à John Doe (car il n'y a pas de paramètre donc ce dernier  
aura sa valeur par défaut)  
sayHello();
```

Retourner un résultat

Si une fonction produit un résultat qui doit être utilisé dans le code, il faut que cette dernière **retourne ce résultat avec le mot-clé 'return'**

```
JS // Fonction qui doit calculer et retourner le résultat du triple du nombre
    // entré en paramètre
    function triple(number){
        return number * 3;
    }

    // Grâce au return de la fonction, le résultat (120) pourra être récupéré
    // et stocké dans la variable result
    let result = triple(40);
```

- En général une fonction va soit **faire une action** et ne **rien retourner** (fonction alert() par exemple), soit **calculer quelque chose** et **retourner le résultat** (comme parseInt() par exemple)
- L'instruction "return" stop le code de la fonction. Si du code est placé après il ne sera jamais lu (sauf si le return est dans une condition).

Portée des variables

La **portée des variables** est un **concept important** qui définit où sont accessibles les variables que l'on crée dans notre code. Par rapport aux fonctions, il existe **deux sortes de variables** :

Les **variables locales** : ce sont les variables **créées directement dans une fonction** et qui **n'existent que dans cette fonction**. Les variables locales d'une fonction n'existent que pour elle :

JS

```
function test(){
    let firstname = 'Alice';

    // Fonctionne car la variable "firstname" existe dans le corps de la
    // fonction (elle est locale à la fonction)
    alert(firstname);
}

// Erreur variable introuvable : ayant été créée dans une fonction, elle
// n'existe que dans cette fonction
alert(firstname);
```

Les **variables globales** : ce sont les variables **créées normalement, en dehors d'une fonction** et qui **existent de partout**.

ATTENTION : Utiliser une variable globale directement dans une fonction est possible (comme vu ici) mais doit être évité le plus possible ! En effet le principe de base d'une fonction est d'être un outil fonctionnel en étant le plus autonome possible. Une fonction qui utilise des variables globales est dépendante de son contexte d'utilisation (c'est-à-dire que la fonction sera dépendante de l'extérieur pour fonctionner correctement).

JS

```
let firstname = 'Alice';

function test(){
    // Fonctionne car la variable "firstname" a été créée en dehors
    // de la fonction et est accessible aussi dans les fonctions: c'est une
    // variable globale
    alert(firstname);
}

// Fonctionne normalement
alert(firstname);
```


Syntaxe des fonctions anonymes

Une fonction **anonyme** est une **fonction sans nom** de fonction. Généralement **utilisées comme arguments** pour d'autres fonctions ou comme **expressions de fonction**. les fonctions anonymes peuvent être utiles lorsque vous avez **besoin d'une fonction simple** pour une **utilisation spécifique, sans avoir à la nommer ou à la déclarer séparément**.

JS

```
// Fonction anonyme créée comme expression de fonction
(function () {});

// Fonction anonyme créée comme fonction fléchée
() => {};
```

Exemple de fonction anonyme

JS

```
// exemple d'une fonction ayant besoin de 2 arguments (1er argument : fonction qui s'applique au
nombre donné, 2ème argument : nombre donné)
function appliquerFonction(fonction, nombre) {
    return fonction(nombre);
}

// version avec fonction nommée dans le cas où on voudrait doubler le chiffre donnée
function doubler(x) {
    return x * 2;
}
const resultat = appliquerFonction(doubler, 5);
console.log(resultat); // Affiche 10

// version avec fonction anonyme dans le cas où on voudrait doubler le chiffre donnée
const resultat = appliquerFonction(function(x) {
    return x * 2;
}, 5);
console.log(resultat); // Affiche 10
```

Syntaxe des fonctions auto invoquées

Une fonction **auto-invoquée**, également connue sous le nom de **fonction immédiatement invoquée** (Immediately Invoked Function Expression, **IIFE**), est une fonction en JavaScript qui est **exécutée dès qu'elle est définie**.

Les IIFEs sont définies entre parenthèses (), et sont généralement **utilisées pour créer une nouvelle portée de variable**, évitant ainsi **la pollution de l'espace global de noms** (global namespace).

JS

```
// Fonction auto-invoquée
(function (){
    let message = "Bonjour tous le monde !"
    console.log(message); // Affiche le message dans la console immédiatement
})();

// Fonction nommée
function hello(){
    let message = "Bonjour à toi !"
    console.log(message); // Affichera le message dans la console lorsqu'elle sera invoquée
}

// variable et utilisation de celle-ci
let message = « Hi»;
console.log(message); // Affiche le message dans la console immédiatement sans rentrée en conflit avec la
variable de la fonction auto-invoquée qui porte le même nom

//Invocation de la fonction hello()
hello();
```

Exemple de fonction auto- invoquée (IIFE)

Supposons que nous voulions **calculer le carré d'un nombre** et **stocker** le résultat dans une **variable, sans polluer l'espace de noms global** avec **des variables temporaires**.

Nous pouvons utiliser une **IIFE** pour atteindre cet objectif :

```
const carre = (function(){  
    const nombre = 5; // nous avons défini une variable nombre à laquelle  
    nous avons attribué la valeur 5  
  
    const resultat = nombre * nombre; // nous avons calculé le carré de ce  
    nombre en multipliant nombre par lui-même  
  
    return resultat; // nous avons ensuite stocké le résultat dans la  
    variable  
  
})();  
  
console.log(carre); // Affiche 25
```

La variable "**carre**" est affectée avec la valeur retournée par l'IIFE (25 dans cet exemple)

Exercice fonction

Énoncé :

Écrire une **fonction Javascript** dont le but sera de **retourner le résultat** d'une **division** entre **2 chiffres** qu'auront rentré l'utilisateur **à notre demande** (le **premier** correspondant au nombre de **kilomètre parcourue** et le **deuxième** le temps mis à parcourir cette distance) et ensuite afficher dans un message d'alert avec le texte suivant en utilisant la fonction :
Votre vitesse est de m/s'.



1.4 Les arrays

A) Rappel et syntaxe

**B) Fonctions utiles
pour les arrays**

**D) Récupérer et
parcourir un array**

E) Exercice

Les arrays

Qu'est ce que c'est ?

Un **array** (tableau de données en français) est **un objet permettant de stocker plusieurs valeurs dans une seule variable** :

```
JS // Création d'un array contenant des animaux
let animals = ['chat', 'chien', 'loutre', 'hérisson'];
```

Pour **accéder à un élément du tableau** (pour l'afficher par exemple), il faut **l'appeler via son index**, c'est-à-dire **sa position dans l'array en partant du zéro pour le premier** :

```
JS // Création d'un array contenant des animaux
let animals = ['chat', 'chien', 'loutre', 'hérisson'];

// Affichera "loutre" (chat = 0, chien = 1, loutre = 2, hérisson = 3)
alert( animals[2] );
```

Attention : le compte des éléments dans un array commence à partir de 0 !

Quelques fonctions utiles sur les arrays

unshift : Ajoute un élément au **début** du tableau

JS

```
let animals = ['chat', 'chien', 'loutre', 'hérisson'];  
  
animals.unshift('lapin');  
  
// Affichera "lapin" (lapin = 0, chat = 1, chien = 2,  
loutre = 3, hérisson = 4)  
alert( animals[0] );
```

push : Ajoute un élément à la **fin** du tableau

JS

```
let animals = ['chat', 'chien', 'loutre', 'hérisson'];  
  
animals.push('lapin');  
  
// Affichera "lapin" (chat = 0, chien = 1, loutre = 2,  
hérisson = 3, lapin = 4)  
alert( animals[4] );
```

shift : Supprime le **premier** élément du tableau

JS

```
let animals = ['chat', 'chien', 'loutre', 'hérisson'];  
// Supprime chat  
animals.shift();  
  
// Affichera "chien" (chien = 0, loutre = 1, hérisson  
= 2)  
alert( animals[0] );
```

pop : Supprime le **dernier** élément du tableau

JS

```
let animals = ['chat', 'chien', 'loutre', 'hérisson'];  
// Supprime hérisson  
animals.pop();  
  
// Affichera "undefined" car il n'existe plus d'élément  
avec l'index 3 (chat = 0, chien = 1, loutre = 2)  
alert( animals[3] );
```


Récupérer et parcourir un array

Pour **récupérer la taille d'un array** (nombre d'éléments dedans), on utilise "**length**" :

```
JS let animals = ['chat', 'chien', 'loutre', 'hérisson'];  
  
// Affichera "4"  
alert(animals.length);
```

Il est très courant d'avoir besoin de **parcourir chaque élément d'un tableau**. Pour **automatiser ça** on peut utiliser les **boucles**, dont une qui a été créée **spécialement à cet effet** : la **boucle forEach**

```
JS let animals = ['chat', 'chien', 'loutre', 'hérisson'];  
// Boucle forEach pour parcourir tous les éléments du tableau "animals", un à un (à chaque tour de  
// boucle) :  
animals.forEach((animal) => {  
    // le paramètre "animal" est une variable qui contient un animal différent à chaque tour de boucle  
    alert (animal);  
});
```

Pour résumer, la **boucle forEach** permet de faire des actions sur **tous les éléments d'un tableau**. Chacun de ces éléments est extrait puis mis à disposition dans le paramètre de la fonction de retour entre les parenthèses de la boucle ("animal" dans l'exemple, le nom est au choix du développeur !).

Exercice array, fonction et boucle

Énoncé :

Écrire un programme Javascript qui :

1. Va **demander confirmation** à l'utilisateur si il souhaite **ajouter un nouveau prénom** puis va **demander** « Quel prénom souhaitez-vous ajouter à la liste ? »
2. **Après avoir** rentrer le prénom, il lui sera re demandé si il souhaite **ajouter un nouveau prénom**
3. **Tant que** l'utilisateur entrera **un prénom à ajouter**, **rajouter** ce prénom dans un **array** et **l'afficher** dans la **console**



1.5 L'objet en JS

A) Les objets Js

B) L'objet this

C) L'objet Window

**D) L'objet String et
Number**

**E) Introduction à la
POO**

F) Exercice

Les objets

Qu'est ce que c'est ?

Les objets sont des **éléments de programmation** qui **possède des variables** (appelées des **attributs**) et des **fonctions** (appelées des **méthodes**). Contrairement aux variables et aux fonctions habituelles, ces dernières n'existent qu'à **l'intérieur de leur objet**.

JS

```
// Un objet
let car = {

  // Un attribut de l'objet (variable)
  color: 'red',

  // Une méthode de l'objet (fonction)
  start: function(){
    alert('La voiture démarre');
  },

};
```

Objet : Voiture



Une **méthode** de l'objet :
La voiture **démarre**

Un **attribut** de l'objet :
couleur : rouge

Les objets servent aussi à créer des **listes d'éléments** (comme les arrays) mais avec des **index personnalisés** (au lieu de 0, 1, 2, etc...) :

JS

```
// Objet matérialisant un bonbon
let candy = {
  type: 'nounours',
  color: 'red',
};

// Deux moyens pour accéder aux éléments d'un objet :
// Affichera "nounours"
alert( candy.type );

// Affichera "nounours" aussi
alert( candy['type'] );
```

Les objets : syntaxe

Les objets permettent de **créer** des **structures autonomes** qui **centralisent tous les éléments nécessaires à leur fonctionnement**. On pourrait par exemple imaginer un objet pour créer un lecteur vidéo et y intégrer dedans toutes ses propriétés (dimensions, fichier lu, etc...) ainsi que toutes ses fonctionnalités (mettre en pause, monter le son, etc...)

JS

```
// Création d'un objet pour gérer un lecteur vidéo (exemple simple, pas un vrai !)  
const videoPlayer = {  
  
  // Attributs (variables) de l'objet  
  height: 200,  
  width: 500,  
  
  // Méthodes (fonctions) de l'objet  
  play: function(){  
    alert('Vidéo lancée !');  
  },  
  pause: function(){  
    alert('Vidéo en pause !');  
  },  
  
};  
  
// Accès aux éléments de l'objet :  
  
// Affichera "200"  
alert( videoPlayer.height );  
  
// Exécutera la fonction play dans l'objet "videoPlayer"  
videoPlayer.play();
```

Les objets : This

Si une **méthode** (fonction) dans un objet **a besoin d'accéder à une autre méthode** ou un **attribut** (variable) de ce même objet, il **peut y avoir accès** facilement grâce à **"this"** :

JS

```
let car = {  
    speed: 50,  
    readSpeed: function(){  
        // "this" correspond en fait à l'objet dans lequel on est actuellement. this.speed  
        // veut donc dire "va chercher la valeur de l'attribut "speed" dans l'objet "car" "  
        alert('La vitesse actuelle de la voiture est de ' + this.speed + ' km/h');  
    }  
};  
  
// Affichera la phrase avec 50 dedans  
car.readSpeed();
```

L'objet window

L'objet **Window** représente la **fenêtre de navigation elle-même**, c'est **l'objet global** en Javascript !

Voici quelques exemples de propriétés et méthodes de cet objet :

```
JS // Propriétés :  
  
// document : Représente le document actuellement chargé dans la fenêtre. Vous pouvez utiliser l'objet  
// document pour manipuler le DOM.  
  
window.document;  
  
// innerWidth et innerHeight : Permettent d'obtenir la largeur et la hauteur de la fenêtre du navigateur.  
  
let widthWindow = window.innerWidth;  
console.log(widthWindow);  
  
let heightWindow = window.innerHeight;  
console.log(heightWindow);  
  
// Méthodes :  
  
// window.confirm() / window.alert() / window.prompt()
```

Note : Lorsque l'on utilise les propriétés et méthodes de l'objet global window, il n'est pas nécessaire d'écrire explicitement window

Voir plus de propriétés et méthodes de l'objet Window : <https://developer.mozilla.org/fr/docs/Web/API/Window>

Exemple d'autres objets : String et Number

L'objet **String** est un objet permettant de **créer et représenter toutes les chaînes de texte** utilisées en **Javascript**.

Quelques **méthodes** et **attributs** accessibles sur les chaînes de texte :

```
JS // Retourne la taille de la chaîne
   'chat'.length // "4"

// Retourne un caractère de la chaîne (premier caractère = 0)
'chat'[2] // "a"

// Permet de tester si la chaîne contient un mot (en réalité beaucoup plus que ça, match permet de
// tester des expressions régulières)
'le chat aime les arbres'.match(/chat/) // true
```

L'objet **Number** est un objet permettant **de créer et représenter tous les nombres** utilisés en **Javascript**.

Quelques **méthodes** et **attributs** accessibles sur les nombres :

```
JS let test = 50;

// Retourne le nombre arrondi à x nombre derrière la virgule (x = paramètre passé entre
// parenthèses)
test.toFixed(2) // "50.00"
```


Introduction à la POO

La **P**rogrammation **O**rientée **O**bjets (**POO**) est un paradigme de programmation (une autre manière de programmer). La manière "**habituelle**" de programmer est la **programmation procédurale**, qui consiste simplement à **écrire à la suite des instructions qui seront lues les unes après les autres**.

Pourquoi apprendre une **nouvelle manière de programmer** ?

Parce que la **programmation procédurale** (programmation faite de manière classique) devient vite **limitée** quand **l'application/site web grossit**.

L'**avantage** de la **programmation procédurale**, c'est qu'elle est **plus simple à comprendre** et à **apprendre**. Mais **pour réaliser des sites web complexes et professionnels, ça se complique**.

La **POO** est **plus complexe à apprendre**, par contre elle **apporte beaucoup d'avantages** :

- ☐ Code **modulaire** (plus facile à réutiliser)
- ☐ Le fonctionnement est **plus intuitif** (les objets se comparent facilement avec ceux de la vie réelle)
- ☐ Le **code** est **beaucoup mieux organisé** et donc **plus évolutif**
- ☐ **Facilite le travail à plusieurs** sur un projet (grâce à la compartimentation du code en sections distinctes et indépendantes)
- ☐ La **maintenabilité du code est plus simple**
- ☐ Le code est **mieux factorisé** (limitation des répétitions de code)

Tous les **sites web avancés, applications et logiciels complexes** utilisent la **programmation orientée objet**. La **POO** n'est pas réservée qu'à **Javascript**, la plupart des gros langages permettent de programmer avec des objets (PHP, C++, Java, etc...).

```
// Définir une classe
class Voiture {
  constructor(modele, couleur) {
    this.modele = modele;
    this.couleur = couleur;
  }

  presentationVoiture() {
    console.log(`Voici une voiture, du modèle ${this.modele} et de couleur ${this.couleur}.`);
  }
}
```

JS

```
// Créer une instance de la classe
let renault2023 = new Voiture ("2023 Renault Clio", "Bleu Iron");
renault2023.presentationVoiture(); // Affiche dans la console :
"Voici une voiture, du modèle 2023 Renault Clio et de couleur Bleu Iron."
```

```
// Créer une autre instance de la classe
let ferrari2023 = new Voiture ("2023 Ferrari Roma Spider", "Rosso mugello");
ferrari2023.presentationVoiture(); // Affiche dans la console :
"Voici une voiture, du modèle 2023 Ferrari Roma Spider et de couleur Rosso mugello."
```

Exercice objet

Énoncé :

Créer un Objet en Javascript qui sera :

Un **personnage**

avec **2 attributs** :

- un **attribut** sur le **nom** du personnage (ex : Alice, Charlotte, Jean...),
- un **attribut** sur l'**âge** du personnage (ex : 28, 40, 18...),

avec **1 méthode** :

- Une **méthode sePresenter** qui produira un message dans la console (« Bonjour, je suis **nom** et j'ai **age** ans.»)

Bonus :

Transformer l'objet créer en une classe pour utiliser la POO



2. Le Dom et les écouteurs d'évènements

2.1) Le DOM

- A) Le DOM
- B) Les sélecteurs
- C) Parcourir le DOM
- D) Manipuler les contenus et les attributs html
- E) Appliquer du CSS et supprimer un élément
- F) Déplacer un élément
- G) Créer et dupliquer un élément
- H) Agir sur un array d'élément

2.2) Les Écouteurs d'évènement

- A) Les écouteurs d'évènement
- B) Les types d'écouteurs
- C) This
- D) Supprimer un écouteur
- E) Attendre le chargement complet du DOM

2.3) Les formulaires

- A) Agir sur des champs de formulaire
- B) Empêcher le comportement par défaut d'un élément
- C) Empêcher l'envoi d'un formulaire

2.4) Les datasets

2.1 Le DOM

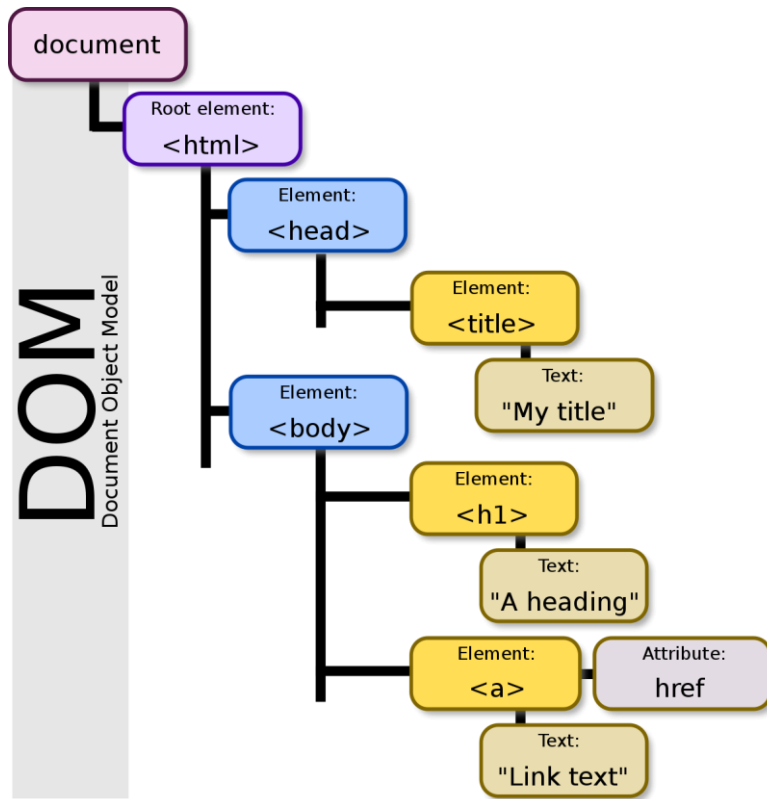
- A) Le DOM**
- B) Les sélecteurs**
- C) Parcourir le DOM**
- D) Manipuler les contenus et les attributs html**
- E) Appliquer du CSS et supprimer un élément**
- F) Déplacer un élément**
- G) Créer et dupliquer un élément**
- H) Agir sur un array d'élément**
- I) Exercice**

Qu'est ce que c'est ?

Le **DOM** (pour **D**ocument **O**bject **M**odel) est un **objet Javascript créé** par le navigateur qui **représente l'intégralité de la page web actuelle**. Il sert de **point d'entrée** pour que le développeur **puisse agir sur la page web via Javascript**.

Dans cet objet, **toutes les balises HTML** ainsi que tous les **textes** de la **page web** sont accessibles sous la forme de "**nœuds**". Javascript, à travers le DOM, nous permet de réaliser **des actions directement avec ces nœuds** :

- **Changer** le texte et/ou HTML d'un nœud
- **Modifier** les attributs HTML d'un nœud
- **Appliquer** du CSS sur un nœud
- **Dupliquer** un nœud
- **Supprimer** un nœud
- **Déplacer** un nœud
- **Créer** un nouveau nœud



Le DOM : Les sélecteurs

Pour pouvoir **agir sur un nœud du DOM**, il faut déjà le **sélectionner**. Il existe **5 sélecteurs** permettant de sélectionner **un ou plusieurs nœuds**.

Les **3 premiers sélecteurs** sont historiquement les **plus anciens** et aussi les **moins utilisés**, car remplacés par les **deux derniers** qui sont **plus simples d'utilisation** !

Avec **querySelector** on pourra sélectionner **un seul élément**, avec **querySelectorAll** plusieurs éléments (array). Les 3 autres peuvent être oubliés.

JS

```
// querySelector
// Sélection avec un sélecteur CSS (un seul élément peut être sélectionné au maximum)

// Ce sélecteur fonctionne exactement comme les sélecteurs CSS (on peut donc construire des sélecteurs très avancés comme en CSS)

let test1 = document.querySelector('#exemple'); // Sélection par ID
let test2 = document.querySelector('.red'); // Sélection par classe (le premier élément de la page ayant cette classe)
let test3 = document.querySelector('strong'); // Sélection par balise (le premier strong de la page)
let test4 = document.querySelector('.main-navbar h1 a'); // Sélection plus avancée (le lien dans un titre h1 dans un élément ayant la classe ".main-navbar")

// querySelectorAll
// Sélection avec un sélecteur CSS (plusieurs éléments peuvent être sélectionnés : le résultat sera sous forme d'un array dans tous les cas)

// Ce sélecteur fonctionne exactement comme les sélecteurs CSS (on peut donc construire des sélecteurs très avancés comme en CSS)

let test1 = document.querySelectorAll('.red'); // Sélection par classe (tous les éléments ayant cette classe dans la page)
let test2 = document.querySelectorAll('strong'); // Sélection par balise (tous les strong de la page)
let test3 = document.querySelectorAll('.main-navbar ul li a'); // Sélection plus avancée (tous les liens "a" dans un li dans un ul dans l'élément ayant la classe ".main-navbar")
```

Le DOM : Les sélecteurs

Les **sélecteurs** vus précédemment permettent aussi de **sélectionner des éléments directement dans un autre** :

HTML

```
<div class="block">  
  <h1>Lorem ipsum dolor.</h1>  
  
</div>
```

JS

```
// Sélection du bloc dans le DOM  
let block = document.querySelector('.block');  
  
// Selection du titre h1 directement dans le block  
let title = block.querySelector('h1');
```

Parcourir l'arborescence du DOM 1/3

Une fois qu'un élément est **sélectionné**, on peut **se déplacer dans l'arborescence à partir de cet élément** :

HTML

```
<div class="test">
  <h2>Lorem ipsum dolor.</h2>
  <h2>Architecto, dignissimos eius!</h2>
  <h2>Doloribus hic, laboriosam.</h2>
  <h2>Lorem consectetur adipisicing.</h2>
</div>
```

Sélectionner tous les enfants de l'élément :

JS

```
// Sélection de l'élément
let testElement = document.querySelector('.test');

// Sélection de tous les enfants de l'élément (tous les h2)
let testElementChildren = testElement.children;
```

Sélectionner un seul enfant de l'élément :

JS

```
// Sélection de l'élément
let testElement = document.querySelector('.test');

// Sélection du premier enfant de l'élément
let firstChild = testElement.firstChild; // Premier h2

// Sélection du dernier enfant de l'élément
let lastChild = testElement.lastChild; // Dernier h2

// Sélection du xème enfant de l'élément (celui que vous voulez, le premier = index 0)
let oneChild = testElement.children[2]; // 3ème h2
```


Parcourir l'arborescence du DOM 2/3

Sélectionner un **élément frère de l'élément**

HTML

```
<ul>
  <li>Pomme</li>
  <li>Poire</li>
  <li class="target">Cerise</li>
  <li>Citron</li>
  <li>Kiwi</li>
</ul>
```

JS

```
// Sélection de l'élément
let target = document.querySelector('.target');

// Sélection de l'élément frère situé juste avant l'élément
let previous = target.previousElementSibling; // Sélectionne le li "Poire"

// Sélection de l'élément frère situé juste après l'élément
let next = target.nextElementSibling; // Sélectionne le li "Citron"
```

Parcourir l'arborescence du DOM 3/3

Sélection du **parent** de l'élément

HTML

```
<div>  
  <p class="target">Exemple</p>  
</div>
```

JS

```
// Sélection de l'élément  
let target = document.querySelector('.target');  
  
// Sélection du parent de l'élément  
let parent = target.parentElement; // Sélectionne la div
```

Manipuler le contenu textuel/HTML d'un élément 1/2

Récupérer ou modifier le contenu textuel d'un élément :

HTML

```
<p class="target">Pomme</p>
```

JS

```
// Sélection de l'élément
let target = document.querySelector('.target');

// Récupère le texte actuel contenu dans l'élément
let fruitName = target.textContent; // Récupère "Pomme"

// Modifie le texte contenu dans l'élément
target.textContent = 'Poire'; // Met "Poire" à la place de "Pomme"
```

Manipuler le contenu textuel/HTML d'un élément 2/2

Récupérer ou modifier le contenu HTML d'un élément :

HTML

```
<p class="target">  
  <strong>Pomme</strong>  
</p>
```

JS

```
// Sélection de l'élément  
let target = document.querySelector('.target');  
  
// Récupère le contenu HTML actuel dans l'élément  
let elementContent = target.innerHTML; // Récupère "<strong>Pomme</strong>"  
  
// Modifie le contenu HTML actuel dans l'élément  
target.innerHTML = '<i>Poire</i>'; // Met "<i>Poire</i>" à la place de  
"<strong>Pomme</strong>"
```

Modifier les attributs HTML d'un élément 1/2

Javascript permet aussi de **manipuler n'importe quel attribut HTML** :

L'**id** HTML :

JS

```
// Sélection d'un élément
let target = document.querySelector('.target');

// Récupère l'ID HTML actuel de l'élément
let elementID = target.id;

// Change l'ID HTML actuel de l'élément
target.id = 'new-id';
```

Les **classes** HTML :

JS

```
// Sélection d'un élément
let target = document.querySelector('.target');

// Récupère la valeur entière de l'attribut "class" de l'élément
let elementClasses = target.className;

// Change l'attribut "class" de l'élément
target.className = 'red main';
```

Modifier les attributs HTML d'un élément 2/2

Les classes HTML :

JS

```
// Retire une classe CSS sans toucher les autres
target.classList.remove('red');

// Ajoute une classe CSS sans toucher les autres
target.classList.add('blue');

// Remplace une classe CSS par une autre sans toucher les autres
target.classList.replace('blue', 'red');

// Ajoute une classe CSS si l'élément ne l'a pas, sinon retire la classe s'il l'a déjà
target.classList.toggle('green');

// Test si l'élément contient une classe CSS ou pas
if( target.classList.contains('green') ){

    alert("L'élément contient bien la classe CSS green");
} else {

    alert("L'élément ne contient pas la classe CSS green");
}
```

Les autres attributs HTML :

JS

```
// Sélection d'un élément
let target = document.querySelector('.target');

// Récupère le contenu d'un attribut de l'élément par son nom (href, alt, src, title, type, value, etc...)
let attributeValue = target.getAttribute('src');

// Modifie un attribut de l'élément par son nom
target.setAttribute('alt', 'Photo de chat');
```

Appliquer du CSS sur un élément et supprimer un élément

S'il est possible de changer les propriétés d'un élément HTML en jouant avec ses classes, on peut aussi **changer directement des attributs CSS** dessus :

```
JS // Sélection d'un élément
let target = document.querySelector('.target');

// Change la propriété CSS "color" de l'élément
target.style.color = 'red';

// Attention, les propriétés CSS composées de plusieurs mots doivent être écrites en lower camel case
target.style.fontSize = '4rem';
```

Pour **supprimer un élément**, la technique est un peu déroutante au départ. Il faut **sélectionner le parent de l'élément** pour **ensuite supprimer son enfant** :

```
JS // Sélection de l'élément à supprimer
let targetToDelete = document.querySelector('.target');

// Suppression de l'élément en passant par son parent
targetToDelete.parentElement.removeChild( targetToDelete );
```

Déplacer un élément

Il existe 4 déplacements d'éléments HTML fondamentaux avec JS :

- **before** : Déplacer un élément avant un autre
- **after** : Déplacer un élément après un autre
- **prepend** : Déplacer un élément dans un autre au début avant ses autres enfants
- **append** : Déplacer un élément dans un autre à la fin après ses autres enfants

HTML

```
<div class="block1">
  <p class="target">Lorem ipsum
  dolor.</p>
</div>

<div class="block2">

</div>
```

JS

```
// Sélection de l'élément à déplacer
let targetToMove = document.querySelector('.target');

// Sélection de l'élément qui servira de référence au déplacement (on va déplacer .target dans
.block2)
let destination = document.querySelector('.block2');

// Déplacement de "target" dans "destination"
destination.append( targetToMove );
```


Créer et dupliquer un élément

Pour **créer un élément**, il faut d'abord **le former** puis ensuite **l'insérer dans le DOM** avec **after** / **before** / **append** / **prepend** :

HTML

```
<div class="block1">  
</div>
```

JS

```
// Création d'un nouvel élément de type paragraphe (p)  
let newElement = document.createElement('p');  
  
// On donne un contenu textuel à notre élément  
newElement.textContent = 'lorem ipsum dolor';  
  
// Pour le moment cet élément n'existe pas dans la page web, on l'insère donc dans l'élément .block1  
document.querySelector('.block1').append( newElement );
```

Dupliquer un élément se fait assez facilement :

JS

```
// Sélection de l'élément à dupliquer  
let target = document.querySelector('.target');  
  
// Création d'une copie de l'élément qui sera stockée dans la variable (true = copier aussi les enfants de l'élément)  
let copy = target.cloneNode(true);  
  
// Maintenant ont fait ce qu'on veut de la copie (modifications, insertion dans le DOM, etc...)
```

Agir sur un array d'éléments

Quand on utilise un **sélecteur qui retourne un array de plusieurs éléments** (comme `querySelectorAll`), on **ne peut pas directement manipuler tous les éléments**, il faut les **parcourir** avec une **boucle** :

HTML

```
<ul>
  <li>Lorem ipsum dolor.</li>
  <li>Libero, quae, quia!</li>
  <li>Facere, quos, vitae!</li>
  <li>Reprehenderit similique, voluptatum.</li>
  <li>Officiis, repellat soluta!</li>
</ul>
```

JS

```
// Sélection de tous les "li" dans le "ul"
let elements = document.querySelectorAll('ul>li');

// Si on veut mettre tous les éléments de l'array en rouge, il faut d'abord parcourir tous
le tableau pour ensuite appliquer la transformation sur chaque élément un par un
elements.forEach((element) => {

  // mise en rouge de chaque élément de l'array (un par tour)
  element.style.color= 'red';

});
```

Exercice DOM

Énoncé : Création d'une partie d'une page HTML en manipulant le DOM

Dans votre fichier JS :

- Créer un Nœud qui se situera **dans le body** et qui sera **l'élément html 'footer'**
- **Dans ce nœud** on devra y trouver les éléments suivant :
 - Un premier élément **nav** avec la classe css **social_nav** dans le footer
 - Un élément **ul** dans cette nav avec **3 li** à l'intérieur et pour chaque li :
 - **Le premier li** avec à l'intérieur un élément **a** qui contient **2 attributs href="#"** et **title="Facebook"**
 - Dans l'élément **a**, il y a un élément **i** avec **2 class "fa-brands"** et **"fa-facebook"**
 - **Le deuxième li** avec à l'intérieur un élément **a** qui contient **2 attributs href="#"** et **title="Instagram"**
 - Dans l'élément **a**, il y a un élément **i** avec **2 class "fa-brands"** et **"fa-intagram"**
 - **Le troisième li** avec à l'intérieur un élément **a** qui contient **2 attributs href="#"** et **title="Pinterest"**
 - Dans l'élément **a**, il y a un élément **i** avec **2 class "fa-brands"** et **"fa-pinterest"**
 - Un Deuxième élément **nav** avec la classe css **other_nav** dans le footer après la première nav
 - Un élément **ul** dans cette nav avec **3 li** à l'intérieur et pour chaque li :
 - **Le premier li** qui contient **1 class "copyright"**
 - Dans l'élément **li**, il y a un élément **texte** : **@copyright HedghogInLove**
 - **Le deuxième li** avec à l'intérieur un élément **a** qui contient **1 attribut href="#"**
 - Dans l'élément **a**, il y a un **texte** : **C.G.U**
 - **Le troisième li** avec à l'intérieur un élément **a** qui contient **1 attribut href="#"**
 - Dans l'élément **a**, il y a un **texte** : **Mentions légales**
 - Le fichier HTML sera donné et à ne pas modifier
 - Le fichier CSS sera donné et également et à ne pas modifier

Si vous respectez le plan décrits avant pour construire le Footer depuis votre fichier JS, vous aurez le même résultat visuel que le screen qui vous sera fournit.

2.2 Écouteurs d'évènements

- A) Les écouteurs d'évènement**
- B) Les types d'écouteurs**
- C) This**
- D) Supprimer un écouteur**
- E) Attendre le chargement complet du DOM**

Les écouteurs d'évènements

Qu'est ce que c'est ?

Un écouteur d'évènement est une **fonction rattachée à un élément HTML** dans le **DOM**. Cette fonction sera **exécutée à chaque fois qu'un évènement sera déclenché depuis l'élément HTML** (click, passage de souris, etc...).

HTML

```
<h1>Lorem ipsum dolor sit.</h1>  
<h2>Libero quibusdam recusandae repudiandae.</h2>
```

JS

```
// Mise en place d'un écouteur d'évènement au "click" de souris qui sera rattaché au  
titre h1  
document.querySelector('h1').addEventListener('click', function(){  
  
    // Le titre h2 change de couleur en rouge  
    document.querySelector('h2').style.color = 'red';  
  
});
```

Types d'écouteurs d'évènements

Liste non exhaustive d'écouteurs d'évènements JS :

- ❑ **click** : se déclenche au clique gauche de souris
- ❑ **dblclick** : se déclenche au double clique gauche de souris
- ❑ **mouseenter** : se déclenche quand la souris entre sur l'élément
- ❑ **mouseleave** : se déclenche quand la souris quitte l'élément
- ❑ **mousemove** : se déclenche quand la souris se déplace sur l'élément
- ❑ **keydown** : se déclenche quand une touche du clavier s'enfonce
- ❑ **keyup** : se déclenche quand une touche du clavier remonte
- ❑ **focus** : se déclenche quand un élément récupère le focus clavier
- ❑ **change** : se déclenche quand un champ de formulaire change de valeur
- ❑ **submit** : se déclenche quand un formulaire est envoyé
- ❑ **reset** : se déclenche quand un formulaire est reset

Liste complète des évènements ici : https://developer.mozilla.org/fr/docs/Web/Events#listing_des_événements

Si depuis l'intérieur d'un **écouteur d'évènement** on souhaite utiliser l'élément qui a déclenché l'évènement, on peut y accéder facilement avec le mot-clé **"this"**.

HTML

```
<button>Clique-moi pour me changer de couleur !</button>
```

JS

```
// Mise en place d'un écouteur d'évènement au "click" de souris qui sera  
rattaché au bouton  
document.querySelector('button').addEventListener('click', function(){  
    // this = l'élément qui à déclenché l'écouteur d'évènement, donc le  
    bouton dans cet exemple  
    this.style.color = 'red';  
});
```

Supprimer un écouteur d'évènement déjà mis en place

Il existe 2 méthodes pour supprimer un écouteur d'évènement déjà mis en place sur un élément HTML :

- Soit on supprime directement l'élément HTML sur lequel est positionné l'écouteur (radical, ça supprime tout)
- Soit on supprime juste l'écouteur d'évènement (mais pour ça il faut que la fonction de l'écouteur d'évènement soit non anonyme) :

```
JS // Fonction de l'écouteur d'évènement
function test(){
    alert();
}

// Mise en place de l'écouteur d'évènement
document.querySelector('.exemple').addEventListener('click', test);

// Suppression de l'écouteur d'évènement
document.querySelector('.exemple').removeEventListener('click', test);
```


Attendre le chargement complet de DOM

Si on essaye de faire des actions sur des éléments HTML qui ne sont pas encore chargés par le navigateur (si par exemple le **fichier JS est inclus dans le head** par exemple), ça ne fonctionnera pas. Il faut **retarder l'exécution du code JS pour attendre que tout le DOM ait fini de charger.**

JS

```
// Écouteur d'évènement un peu particulier, directement rattaché au DOM
document.addEventListener('DOMContentLoaded', function(){
    // Code JS qui sera exécuté après chargement complet du DOM...
});
```

2.3 Les formulaires

- A) Agir sur des champs de formulaire**
- B) Empêcher le comportement par défaut d'un élément**
- C) Empêcher l'envoi d'un formulaire**
- D) Exercice**

Agir sur des champs de formulaire

Pour récupérer la valeur d'un champ de formulaire :

HTML

```
<input type="text" class="target">
```

JS

```
// Sélection du champ de formulaire
let field = document.querySelectorAll('.target');

// Récupération du texte qui est actuellement dans le champ
let fieldValue = field.value;
```

Il est possible de s'en servir pour faire des vérifications et ainsi avertir l'utilisateur si le champ n'est pas correct :

JS

```
// Vérification du champ
// On vérifie si la taille de la valeur correspond à l'intervall que nous souhaitons sinon
erreur
if(fieldValue.length < 2 || fieldValue.length > 25){

    // Appel d'une fonction 'setFieldError()' qu'on aura créé permettant de mettre le champ
    en rouge avec un message d'erreur dessus
    field.setFieldError(firstnameField, 'Le prénom doit contenir entre 2 et 25 caractères
    !');
}
```

Empêcher le comportement par défaut d'un élément HTML

Depuis un écouteur d'évènement, il est possible de **stopper le comportement par défaut d'un élément HTML** (comme les liens ou les formulaires).

HTML

```
<a href="https://www.google.fr">Lien vers Google</a>
```

JS

```
// Dans cet exemple, le lien ne fonctionnera jamais  
document.querySelector('a').addEventListener('click', function(e){  
    e.preventDefault();  
});
```

Attention à ne pas oublier de déclarer le paramètre "e" pour avoir accès à la méthode preventDefault dans la fonction.

- En général **bloquer le comportement par défaut** d'un élément HTML est utile quand on souhaite par exemple faire des **vérifications sur les champs d'un formulaire sans que ce dernier ne recharge la page**.

Empêcher le comportement par défaut d'un formulaire

Pour **empêcher l'envoi du formulaire** afin de ne l'envoyer que quand les champs seront valides par exemple, on peut alors utiliser **e.preventDefault()** en plaçant un **écouteur à l'envoi du formulaire sélectionné** :

HTML

```
<form action="" method="POST" id="register-form">
  <div>
    <label for="firstname">Prénom</label>
    <input type="text" id="firstname" name="firstname">
  </div>
  <div>
    <label for="lastname">Nom</label>
    <input type="text" id="lastname" name="lastname">
  </div>
  <div>
    <input type="submit" value="Envoyer">
  </div>
</form>
```

JS

```
// Dans cet exemple, l'envoi du formulaire ne fonctionnera jamais
document.querySelector('#register-form').addEventListener('submit', function(e){
  e.preventDefault();
});
```

Exercice formulaire

Énoncé : Suivre les instructions suivantes pour faire la **vérification de 2 champs de formulaire** et **faire afficher l'erreur qui correspond**.

Pour cet exercice:

- Le **fichier HTML** vous sera **fourni** et ne devra **pas être modifié**.
- Le **fichier CSS** vous sera également **fourni** et ne **devra pas être modifié**.
- Dans **votre fichier JS** vous devrez :
 - Créer une **fonction** avec **deux paramètres** (le premier fera référence au **champ demandé** et le deuxième sera le **message d'erreur à afficher**).
 - Dans cette fonction, vous devrez :
 - Ajouter la classe « **field-invalid** » sur le champ demandé.
 - Créer une div pour le message d'erreur.
 - Ajouter la classe « **error-text** » à cette div.
 - Donner du **texte** à cette div qui sera le **message d'erreur à afficher**.
 - Insérer cette div après le champ demandé.

Exercice formulaire suite

- Attendre le chargement de la page et à l'intérieur :
 - **Sélectionner** le formulaire à vérifier.
 - Créer **2 variables** qui correspondent au **2 champs de formulaire à vérifier**
 - Mettre un **écouteur d'évènement** sur notre **formulaire sélectionné** à l'envoi de celui-ci
 - Dans notre écouteur il faudra :
 - Empêcher le comportement par défaut de notre formulaire.
 - Nettoyer les messages d'erreurs qui seront déjà présent pour éviter l'accumulation de ceux-ci (class field-invalid et div avec class error-text).
 - Vérifier si la valeur récupérée du premier champ est inférieur à 2 ou supérieur à 25, si c'est le cas on appelle la fonction créée précédemment en indiquant **en premier paramètre le champ choisi** et **en second le texte** suivant : **'Le prénom doit contenir entre 2 et 25 caractères !'**
 - Vérifier si la valeur récupéré du deuxième champ est inférieur à 2 ou supérieur à 25, si c'est le cas on appelle la fonction créée précédemment en indiquant **en premier paramètre le champ choisi** et **en second le texte** suivant : **'Le nom doit contenir entre 2 et 25 caractères !'**

Exercice formulaire screen

Formulaire JS

Prénom

Le prénom doit contenir entre 2 et 25 caractères !

Nom

Le nom doit contenir entre 2 et 25 caractères !

Envoyer

A thin vertical black line is positioned to the left of the section header.

2.4 Les datasets

Les datasets

Les **datasets** sont des **attributs HTML spéciaux** (qui commencent par "**data-**") qui permettent de **stocker des informations directement dans le code HTML** (un peu comme des variables dans le code HTML)

Ces attributs spéciaux sont **valides au W3C** et peuvent avoir n'importe quel nom du moment qu'ils commencent par "data-" :

HTML

```
<!-- Cette balise contient un dataset contenant un nom de ville -->  
<h2 data-city="paris">Jean</h2>
```

JS

```
// Récupère "paris"  
let cityName = document.querySelector('h2').dataset.city;
```

3. Gérer le CSS depuis JS / Ajax

3.1) Gérer le CSS depuis JS

- A) Rappel succinct de CSS
- B) Les objets et propriétés importantes pour manipuler CSS en JavaScript
- C) Accéder en écriture et lecture à CSS depuis JavaScript

3.2) Manipulation des media queries pour un design responsive en JS

- A) Les media queries en JS
- B) Construction dynamique d'interface selon le type d'écran

3.3) Ajax

- A) Javascript et Ajax
- B) Données JSON
- C) Créer une requête AJAX
- D) Diagramme
- E) Finally
- F) API

3.4) Nouveautés d'ECMAScript

3.1 Gérer le CSS depuis JS

CSS



JS



A) Rappel succinct de CSS

B) Les objets et propriétés importantes pour manipuler CSS en JavaScript

C) Accéder en écriture et lecture à CSS depuis JavaScript

Rappel succinct de CSS

- ❑ **Introduction au CSS** : C'est **un langage de feuille de style** qui définit le **style des documents HTML**, comme la **couleur**, la **taille de la police**, la **disposition**, etc.
- ❑ **Sélecteurs CSS** : Ils permettent de **cibler des éléments spécifiques sur une page HTML** pour leur appliquer des styles.
- ❑ **Propriétés CSS** : Elles **définissent les styles à appliquer** aux éléments sélectionnés.
- ❑ **Classes et ID** : Ces sélecteurs permettent de **cibler des éléments spécifiques** sur une page.

HTML

```
<!-- élément html (balise de titre) avec une class et un id-->
<h1 class="title" id="main-title">Titre de ma page</h1>
```

CSS

```
/*Syntaxe*/
selecteurCSS{
  Propriété: valeur;
  Propriété: valeur;
  Propriété: valeur;
}
```

CSS

```
/*Sélection de l'élément html h1 par sa classe et application d'une couleur,
d'une taille de police et de son alignement*/
.title{
  color: red;
  font-size: 1.5rem;
  text-align: center;
}
```



Les objets et propriétés importantes pour manipuler CSS en JavaScript

- ❑ **L'objet document** : Permet d'accéder au DOM (Document Object Model) et d'interagir avec lui.
- ❑ **L'objet element.style** : Permet d'accéder et de modifier les styles CSS d'un élément.
- ❑ **La méthode document.querySelector() et document.querySelectorAll()** : Permet de sélectionner des éléments en utilisant des sélecteurs CSS.
- ❑ **La méthode element.classList** : Permet de manipuler les classes CSS d'un élément.

HTML

```
<!-- élément html (balise de titre) avec une class -->  
<h1 class="title">Titre de ma page</h1>
```

JS

```
// Sélection du titre  
let title = document.querySelector('.title');  
  
// Change la propriété CSS "color" de l'élément  
title.style.color = 'black';  
  
// Attention, les propriétés CSS composées de plusieurs mots doivent être écrites en  
lower camel case  
title.style.fontSize = '4rem';
```

Accéder en écriture et lecture à CSS depuis JavaScript

- ❑ **Lire les styles CSS** : On peut lire les styles CSS en ligne d'un élément en utilisant **element.style.propertyName**. Attention cependant cela ne fonctionne pas pour les styles définis dans une feuille de style externe ou interne. Si on veut pouvoir lire les styles CSS autrement on peut utiliser : **getComputedStyle(element).propertyName**
- ❑ **Modifier les styles CSS** : On peut modifier les styles CSS en ligne d'un élément en utilisant **element.style.propertyName = newValue**.
- ❑ **Utiliser element.classList** : Les méthodes **add()**, **remove()**, **toggle()** et **contains()** permettent respectivement d'ajouter, de supprimer, de basculer et de vérifier si une classe est présente.

HTML

```
<!-- élément html (balise de paragraphe) avec une classe -->
<p class="text-red" style="color: pink;">Mon texte est de couleur rose</p>
```

JS

```
let element = document.querySelector(".text-red");

// Lire une propriété CSS
console.log(element.style.color); // affiche dans la console la valeur de la propriété color de notre élément, ici « pink »

// Modifier une propriété CSS
element.style.color = "blue"; // Modifie ou ajoute le style de couleur bleu à notre élément

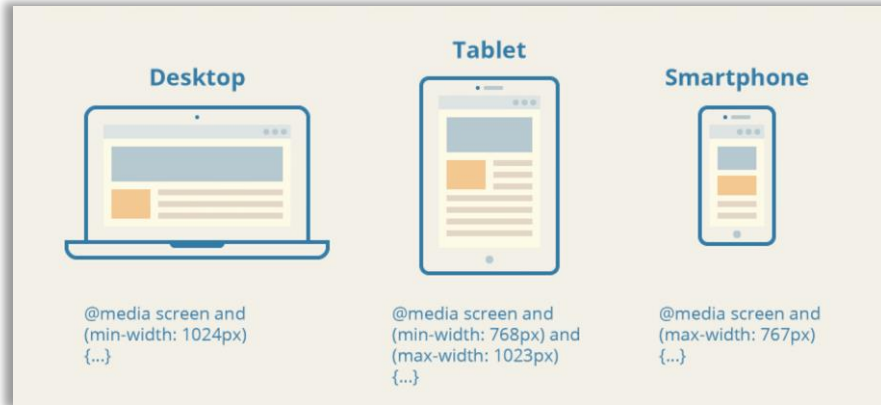
// Ajouter une classe
element.classList.add("maClasse"); // Ajoute la classe maClasse à notre l'élément

// Supprimer une classe
element.classList.remove("uneAutreClasse"); // Supprime la classe uneAutreClasse de notre élément

// Vérifier si une classe est présente
console.log(element.classList.contains("maClasse")); // Affiche true dans la console car la classe est présente

// Basculer une classe
element.classList.toggle("maClasse");
```

3.2 Manipulation des media queries pour un design responsive en JS



A) Les media queries en JS

B) Construction dynamique d'interface selon le type d'écran

Les media queries en JS

Les **media queries CSS** sont généralement utilisées dans les feuilles de style pour rendre le design d'un site web responsive.

Cependant, on peut également les utiliser en JavaScript à l'aide de **window.matchMedia()**. Cette méthode retourne un objet **MediaQueryList** qui contient plusieurs propriétés et méthodes qui nous permettent de travailler avec la media query.

Matches est une propriété de l'objet **MediaQueryList** et permet de **retourner une valeur booléenne** qui renverra true si le document actuel correspond à la media query

JS

```
let mediaQuery = window.matchMedia("(min-width: 600px)");

if (mediaQuery.matches) {
  // La largeur de la fenêtre est de plus de 600px
  alert('la largeur de la fenêtre est de plus de 600px');
} else {
  // La largeur de la fenêtre est de 600px ou moins
  alert('la largeur de la fenêtre est de 600px ou moins');
}
```

On peut également écouter les changements de media queries à l'aide de l'**évènement « change »** de notre écouteur d'évènement **addEventListener**, cela peut permettre d'ajuster dynamiquement l'interface utilisateur quand la taille de la fenêtre change.

JS

```
let mediaQuery = window.matchMedia("(min-width: 600px)");
mediaQuery.addEventListener('change',function(e) {
  if (e.matches) {
    // La media query correspond maintenant
    valueStyle.style.color = "purple"; // Changera la couleur du texte en violet
  } else {
    // La media query ne correspond plus
    valueStyle.style.color = "grey"; // Changera la couleur du texte en gris
  }
});
```

Construction dynamique d'interface selon le type d'écran

Exemple de ce que l'on pourrait faire pour ajuster dynamiquement l'interface de l'utilisateur en fonction de la taille de l'écran.

HTML

```
<!--Menu principal en ligne-->
<div class="menu">
  <ul>
    <li><a href="#">Accueil</a></li>
    <li><a href="#">À propos</a></li>
    <li><a href="#">Contact</a></li>
  </ul>
</div>

<!--bouton menu burger-->
<div id="menu-burger">
  <div class="menu">
    <a href="#"><i class="fa-solid fa-bars"></i></a>
  </div>
</div>
```

JS

```
let menu = document.querySelector(".menu");
let menuBurger = document.querySelector("#menu-burger");

let mediaQuery = window.matchMedia("(min-width: 600px)");
mediaQuery.addEventListener("change", ajusterMenu);

function ajusterMenu(e) {
  if (e.matches) {
    // Grand écran : afficher le menu et cacher le bouton de menu
    burger
    menu.style.display = "block";
    menuBurger.style.display = "none";
  } else {
    // Petit écran : cacher le menu et afficher le bouton de menu
    burger
    menu.style.display = "none";
    menuBurger.style.display = "block";
  }
}

// Appeler la fonction une fois au chargement de la page pour initialiser
l'interface
ajusterMenu(mediaQuery);
```

Exercice manipuler le CSS depuis le JS

Énoncé : Suivre les instructions suivantes pour faire le script qui permettra l'ajout ou la suppression de la classe « **dark-theme** » selon l'écoute du clic sur nos **deux boutons** qui servent à changer le thème de couleur de notre page.

Le **fichier HTML** sera donné et à ne **pas modifier**.

Le **fichier CSS** sera donné également et à ne **pas modifier**.

Dans votre **fichier JS** vous devrez :

- ☐ Faire une **fonction** avec **un paramètre** qui définira si il s'agit du **thème dark ou light**, avec une condition à l'intérieur pour ajouter la classe au body ou la supprimer.
- ☐ Faire **deux écouteurs d'évènement** qui définiront **selon le bouton** qui sera cliqué la fonction a utilisé avec le **thème light** ou **dark**.



3.3 AJAX

- A) Javascript et Ajax**
- B) Données JSON**
- C) Créer une requête AJAX**
- D) Diagramme**
- E) Finally**
- F) API**
- G) Exercice**

Javascript et Ajax

AJAX est une **technique en Javascript** permettant d'effectuer des requêtes HTTP à un serveur web, et ce, **sans avoir à recharger la page actuelle**.

AJAX est l'acronyme de **Asynchronous Javascript And XML**.

AJAX n'est pas un langage de programmation ! C'est simplement une **fonctionnalité de Javascript**.

Le rôle d'AJAX est de **permettre la récupération et/ou l'envoi de données vers une autre page web**, **sans quitter la page actuelle**. Voici quelques systèmes qui peuvent utiliser AJAX :

- ❖ **Chargements infinis au scroll type Facebook** : les actus sont chargées en temps réel sans recharger la page
- ❖ **Tchat de discussion instantané** : les messages sont envoyés et récupérés sans recharger la page (Les développeurs utilisent plutôt la technologie des Websockets, qui est plus adaptée pour les tchats)
- ❖ **Calendrier interactif** : les évènements du calendrier sont chargés et enregistrés en temps réel
- ❖ **Formulaire avec validation sans rechargement de page** : les données sont envoyées au serveur via AJAX, sans bouger de la page
- ❖ **Vérification de la disponibilité d'un email/pseudonyme en temps réel** : la vérification peut se faire en direct, avant même de cliquer sur un bouton par exemple
- ❖ **Système d'autocomplétion** : AJAX permet en temps réel de créer des suggestions dans un champ, comme dans une recherche Google par exemple

Données JSON

AJAX peut utiliser **XML** pour **transférer les données**, mais il peut également utiliser **d'autres formats** comme **JSON**.

JSON (**J**ava**S**cript **O**bject **N**otation) est un **format léger d'échange de données**, facile à lire et à écrire pour les humains, et facile à analyser et à générer pour les machines. Il est **couramment utilisé pour transmettre des données entre un serveur et un client web**, comme une **alternative aux formats XML ou HTML**.. En **JavaScript**, les **données JSON** sont **représentées sous forme de chaînes de caractères**.

Syntaxe **JSON** : délimités par des accolades `{}` et contiennent **des paires clé-valeur** séparées par des virgules.

Les **clés** sont des **chaînes de caractères** entourées de guillemets doubles "

les **valeurs** peuvent être **des chaînes de caractères**, des **nombres**, des **objets JSON**, des **tableaux** ou des valeurs littérales **true**, **false** et **null**.

JS

```
{  
  "prenom": "Jean",  
  "age": 30,  
  "actif": true,  
  "competences": ["HTML", "CSS", "JavaScript"],  
  "adresse": {  
    "rue": "123 rue des Fleurs",  
    "ville": "Paris"  
  }  
}
```

Créer une requête AJAX

Pour créer une requête AJAX, il existe 2 manières différentes :

Soit utiliser l'objet XMLHttpRequest (**ancienne méthode**), soit la **fonction fetch()** (**plus récente, plus souple et plus puissante**). Ce cours se basera donc sur la fonction fetch() .

Voici un exemple d'une requête AJAX avec fetch() pour récupérer le contenu d'une page HTML :

Cet exemple montre comment gérer la récupération d'un contenu textuel ou HTML. Pour récupérer un contenu plus spécifique comme JSON ou XML, il faudra configurer différemment la requête (voir suite du cours).

JS

```
// Requête AJAX pour récupérer le contenu de la page "test.html"
fetch('test.html')

    .then((response) => {

        // Si la requête est OK (c'est-à-dire pas un code 404, 500 ou autre du genre)
        if(response.ok){

            // Permet d'envoyer le contenu textuel de la requête AJAX dans la variable
            "data" du prochain ".then()"
            return response.text();
        }

        // On indique qu'une erreur a eu lieu
        alert('Le contenu n\'a pas pu être récupéré');

    })
```

Suite après ->

Créer une requête AJAX

il y a **plusieurs situations possibles** quand on fait de l'**AJAX** : la requête **peut fonctionner normalement** ou bien **échouer si le serveur ne répond pas**, ou si la page n'est pas trouvée par exemple.

La **fonction "fetch()"** renvoi une **promesse** Javascript qui est suivie de méthodes pour **traiter les différentes situations possibles** de la requête.

1 : La fonction **"fetch()"** crée la requête AJAX et renvoi une **promesse** Javascript

2 : Si la requête AJAX a échoué, la promesse échoue et c'est la méthode **".catch()"** qui sera lue et exécutée

3 : Si la requête AJAX a réussi, la promesse réussit et c'est la première méthode **".then()"** qui sera lue et exécutée

4 : La première méthode **".then()"** récupère la promesse dans le paramètre **"response"** et vérifie que la page récupérée n'est pas une erreur (404, 403, etc...). Cette dernière retourne le contenu de la page ciblée par AJAX.

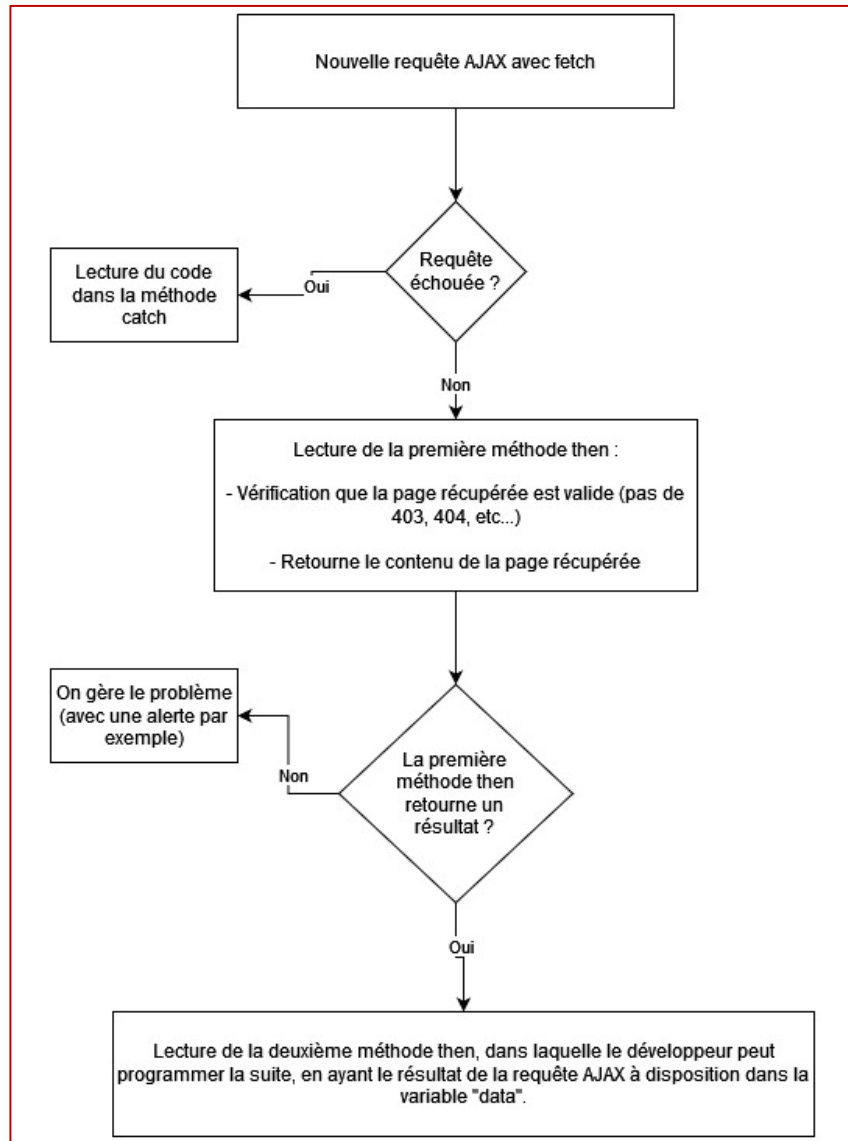
5 : La deuxième méthode **".then()"** récupère le contenu de la page ciblée par AJAX dans son paramètre **"data"** et s'exécute.

6 : Le développeur peut coder ce qu'il veut dans la deuxième méthode **".then()"**, avec le contenu récupéré dans **"data"**

Suite :

```
.then((data) => {  
  
    // Dans cette fonction, la variable "data"  
    // contient le contenu de la page requêtée par AJAX.  
    // On peut faire ce que l'on veut maintenant de  
    // ce contenu comme l'afficher dans la console ou l'afficher  
    // dans un endroit de la page par exemple  
    console.log(data);  
  
    // ...  
  
})  
  
.catch(() => {  
  
    // Si on rentre ici dans le catch, ça veut dire  
    // que la requête AJAX n'a pas réussi à joindre le serveur  
    // web de la page cible  
    alert('Le contenu n\'a pas pu être récupéré');  
  
});
```


Diagramme du fonctionnement d'une requête Ajax avec fetch



Finally

Optionnellement, on peut ajouter une méthode **".finally()"** permettant d'exécuter du code supplémentaire quand la **requête AJAX** est terminée.

La particularité de **finally**, c'est que cette méthode **sera exécutée dans tous les cas de figure**, que la **requête échoue ou non**. On l'utilise beaucoup pour **supprimer un overlay avec une icône de chargement par exemple** :

JS

```
// Imaginons que nous avons une fonction permettant de créer un overlay sur toute la page, qui restera en place le temps que la requête AJAX se termine
// Mise en place de l'overlay, juste avant d'envoyer la requête AJAX
setOverlay();

fetch('test.html')

  .then((response) => {
    // Code de traitement...
  })

  .then((data) => {
    // Code en cas de réussite...
  })

  .catch(() => {
    // Code en cas de problème...
  })

  .finally(() => {

    // On rentrera ici dans tous les cas, que la requête AJAX ait réussi ou échoué
    // Suppression de l'overlay
    removeOverlay();
  });
```

Une **API** web (**A**pplication **P**rogramming **I**nterface) est une **interface permettant à un service** (comme un site web) **de mettre à disposition des fonctionnalités** pour d'autres sites web.

Les **API** servent beaucoup à mettre **à disposition des données** pour que **d'autres sites web puissent les récupérer et les utiliser**.

Attention : beaucoup de services d'API **peuvent être payants** en fonction du **volume de données** que vous allez demander au service !
Très souvent il est **nécessaire de s'inscrire au préalable** et d'avoir une **"clé d'API"**, servant à vous authentifier auprès du service d'API.

Une **API** s'utilise avec ses **endpoints** (points de terminaison en français). Les **endpoints** sont en fait les **URL qui permettent d'utiliser l'API avec un verbe HTTP (GET, POST, etc...)** :

- **GET https://monapi.fr/users/** -> Cet **endpoint** signifie que faire une **requête "GET"** sur cette **adresse URL** permettra de **récupérer tous les utilisateurs**
- **POST https://monapi.fr/users/** -> Cet **endpoint** signifie que faire une **requête "POST"** sur cette **adresse URL** permettra de **créer un nouvel utilisateur** à partir des données POST envoyées avec la requête
- **GET https://monapi.fr/articles/** -> Cet **endpoint** signifie que faire une **requête "GET"** sur cette **adresse URL** permettra de **récupérer tous les articles**
- etc...

Exercice Ajax

Énoncé : Le but sera de **chargé la bonne vue selon le bouton cliqué avec Ajax** ce qui évitera le chargement de la page à chaque clic effectué. Le lien **vers la démo en ligne** et les **ressources** vous seront fournis.

Si un des 3 boutons est cliqué, utiliser une requête AJAX pour récupérer le contenu du fichier de l'animal correspondant dans "content" et l'insérer dans la div.

Ex : le bouton "Chat" est cliqué, il faut faire une requête AJAX sur le fichier "content/chat.php", puis intégrer le contenu de ce fichier dans la div.

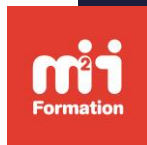
Bonus : Essayez de ne pas dupliquer 3 fois le même code pour les 3 boutons !



3.3 Nouveautés ECMAScript 6 à 11

Résumé des nouvelles fonctionnalités majeures introduites dans les versions ECMAScript 6 à 11

ECMAScript 6 (ES6 / ES2015)	<ul style="list-style-type: none"> ❑ Let et Const : Deux nouvelles déclarations de variables qui viennent en complément du mot-clé "var". "let" est pour les variables qui peuvent être réassignées, et "const" est pour les constantes. ❑ Classes : Une syntaxe plus claire pour créer des objets et gérer l'héritage. ❑ Fonctions fléchées : Elles permettent une syntaxe plus courte pour écrire des fonctions, et ne créent pas leur propre contexte (this). ❑ Promesses : Pour gérer les opérations asynchrones. ❑ Modules : Pour importer et exporter des modules.
ECMAScript 7 (ES7 / ES2016)	<ul style="list-style-type: none"> ❑ Opérateur exponentiel : ** pour représenter une puissance. ❑ Array.prototype.includes : Une méthode pour vérifier si un tableau contient une certaine valeur.
ECMAScript 8 (ES8 / ES2017)	<ul style="list-style-type: none"> ❑ Async/Await : Une syntaxe plus lisible et facile à gérer pour les opérations asynchrones. ❑ Object.values() / Object.entries() : Ces méthodes récupèrent les valeurs/les paires clé-valeur d'un objet.
ECMAScript 9 (ES9 / ES2018)	<ul style="list-style-type: none"> ❑ Rest/Spread Properties : Extension de l'opérateur Rest/Spread aux objets. ❑ Asynchronous Iteration : Pour itérer sur des données générées de manière asynchrone. ❑ Promise.prototype.finally() : Méthode appelée quand la Promesse est résolue, qu'elle ait été tenue ou rejetée.
ECMAScript 10 (ES10 / ES2019)	<ul style="list-style-type: none"> ❑ Array.prototype.{flat,flatMap} : Des méthodes pour aplatir des tableaux. ❑ Object.fromEntries() : La méthode transforme une liste de paires clé-valeur en un objet. ❑ String.prototype.{trimStart,trimEnd} : Des méthodes pour enlever les espaces en début ou fin de chaîne.
ECMAScript 11 (ES11 / ES2020)	<ul style="list-style-type: none"> ❑ Optional Chaining : Permet d'accéder en lecture à la valeur d'une sous-propriété d'un objet, même si cette dernière ou ses intermédiaires sont undefined ou null. ❑ Nullish Coalescing : ?? est un opérateur logique qui renvoie son opérande de droite lorsque son opérande de gauche est null ou undefined. ❑ BigInt : Pour représenter des entiers de taille arbitraire. ❑ Promise.allSettled : Renvoie une promesse qui est tenue quand toutes les promesses dans l'itérable passé en argument ont été tenues ou rejetées.



Fin du module

par Audrey Donjon



m2iinformation.fr