



JAVASCRIPT

Coder en JavaScript..



Introduction

Créé en 1995 par Brendan Eich pour la Netscape Communication Corporation.

Version actuelle : **ECMAScript 2025**
https://www.w3schools.com/Js/js_2025.asp

Rappel

Attention : Java et Javascript sont radicalement différents.

Script

Le JavaScript est un langage de script basé sur la norme ECMAScript.

Extension .js

Intégré

Il s'insère dans le code HTML d'une page web, et permet d'en augmenter le spectre des possibilités (interactivité et dynamisme).

POO

Ce langage de POO, faiblement typé, est exécuté côté client.
Mais également côté serveur avec Node.js

Normalisé

Normalisé par ECMAScript
<https://262.ecma-international.org/>

Où se place le code JavaScript ?

Intégration de Javascript

1. Directement dans les balises HTML

- Utilisation du gestionnaire d'évènement :

Un évènement qui doit déclencher le script.

<nom eventHandler="script" /nom>

2. Entre les balises <script> </script>

- Une nouvelle balise
soit dans le head (exécuté plus tard).
soit dans le body (à l'affichage de la page).

Attention aux anciens navigateurs astuce

<!-- code -->

3. Placer le code dans un fichier séparé

- Tout comme le CSS, déclaration d'un fichier contenant le script.

```
<!DOCTYPE html>
<html lang="fr">

  <head>

    <!-- ENCODAGE -->
    <meta charset="UTF-8">
    <!-- Comptabilité navigateur selon version -->
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <!-- description du site -->
    <meta name="description" content="Cours HTML">
    <!-- prise en charge du contexte mobile -->
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <!-- titre de la page -->
    <title>Document</title>
    <link rel="stylesheet" type="text/css" href="ressources/style.css">

    <!-- Meilleur méthode -->
    <script type="text/javascript" src="ressources/script/script.js"></script>
  </head>

  <body>

    <!-- chargement de la page -->
    <script type="text/javascript">
      <!--
        | alert('Début du chargement de la page');
        //-->
    </script>

    <!-- directement dans la balise -->
    <a href="#" onclick="alert('Bonjour !');">lien</a>
    <a href="javascript:alert('Coucou');> Cliquez ici </a>

  </body>

</html>
```

Le débogage

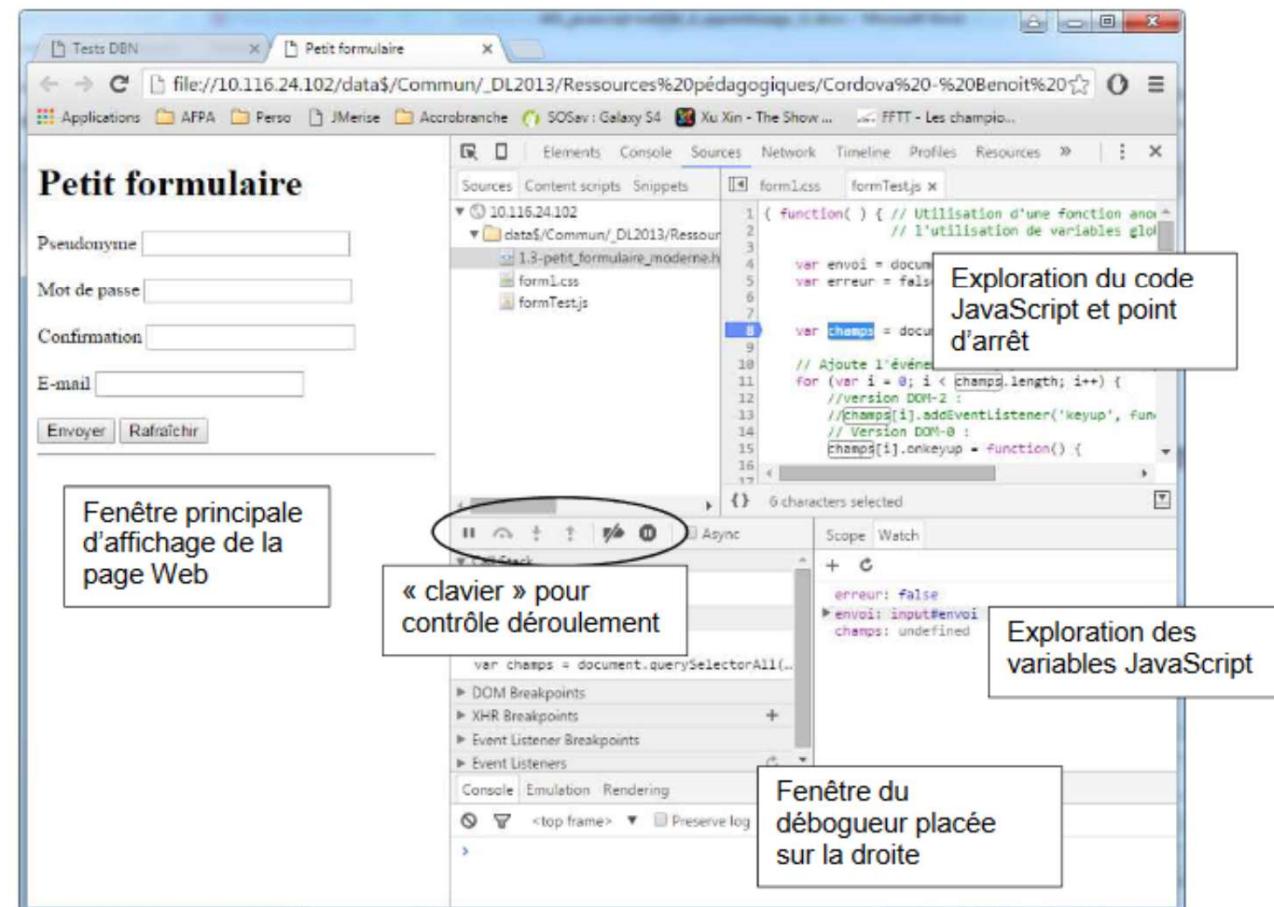
Debugger un script Javascript

En cas d'anomalie :

- **Une erreur de syntaxe Javascript, c'est tout le chargement du bloc qui est annulé !**
- L'utilisation `alert()` pour afficher des boites de dialogue supplémentaires permettant de tracer le déroulement d'un script.
- `console.log()` pour contrôler un ensemble de valeurs et les afficher dans la console du navigateur.

La console du navigateur est accessible en lançant l'outil de développement du navigateur.

- C'est également dans cette console que l'on pourra récupérer les informations sur les erreurs de syntaxe, effectuer des points d'arrêt, du pas à pas, ...



Pour avancer pas à pas après une pause sur un point d'arrêt, utiliser les touches de fonction F10 et F11.



Découverte du langage

<https://www.w3schools.com/js/>

<https://fr.javascript.info/>

<https://developer.mozilla.org/fr/docs/Web/JavaScript>

Découverte du langage

Faisons connaissance...

```
<!-- chargement de la page -->
<script type="text/javascript">
  <!--
    | alert('Debut du chargement de la page');
    //-->
  </script>
```



Base

- Sensible à la case : alert() et non Alert()
`// commentaire`
`bloc de commentaire /* ... */`



Convention

`;` pour terminer une instruction.
Même si c'est possible sans, cela permet d'éviter les erreurs.



Les variables

- Typée dynamiquement et à typage faible.
- Mutabilité des variables.
- Une variable non déclarée est `undefined`



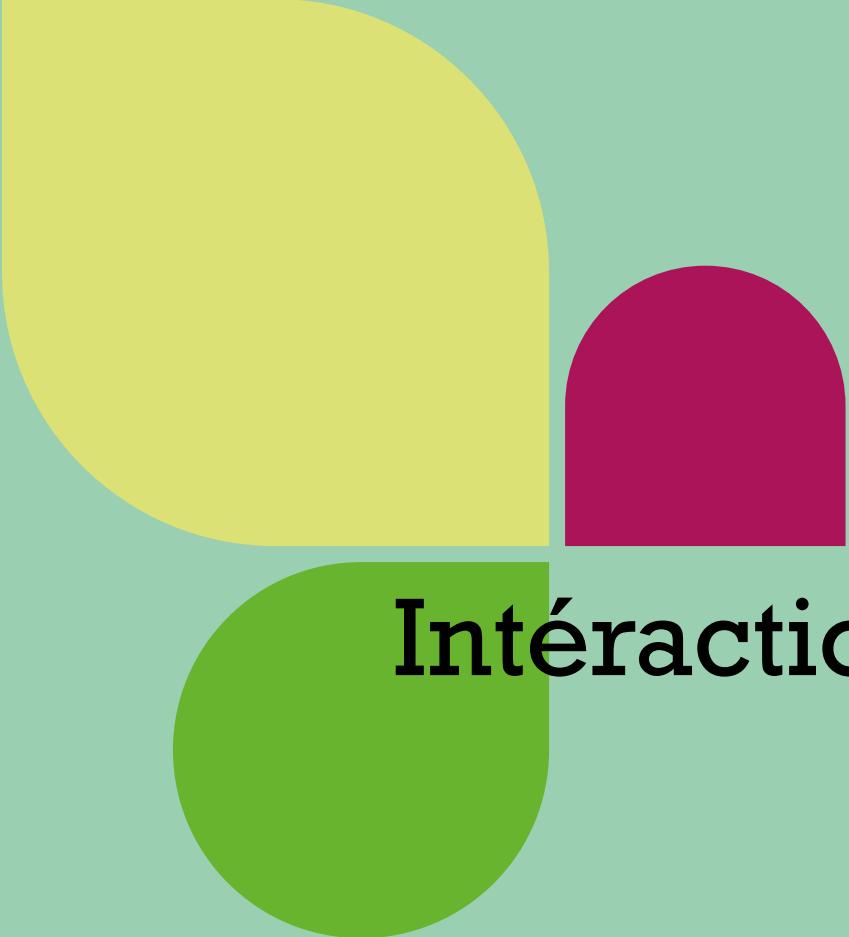
var ou let

*Attention : vous pourrez croiser le mot clé var plutôt que let.
Pour l'instant considérer var comme l'ancienne version de let
let nom = "Jérôme"*



tips

Pour les anciens navigateurs, on doit encapsuler notre code entre les balises de commentaires HTML (voir capture)



A large, semi-transparent graphic is overlaid on the slide. It features three overlapping circles: a yellow circle at the top left, a red circle at the top right, and a green circle at the bottom left. The circles overlap to create a sense of depth and interaction.

Intéractions

Interaction

voyons quelques fonctions pour interagir avec l'utilisateur : **alert**, **prompt** et **confirm**.



alert : affiche un message model et attend que l'utilisateur appuie sur ok



prompt : affiche une fenêtre modale avec un message texte, un champ de saisie pour le visiteur et les boutons OK ou Annuler.



confirm : affiche une fenêtre modale avec une question et deux boutons : OK et Annuler. Le résultat est true si vous appuyez sur OK et false dans le cas contraire.



let, var et const déclaration de variables

Let, var et const

Javascript a été créé à l'origine avec un seul mot-clé pour définir une variable, ce mot réservé est : `var`.

Mais depuis la ES6 : ECMAScript 2015 (*la plus grande mise à jour de JS*) , deux nouveaux mots-clés sont apparus : `let` et `const`.

1

Stocké en global ?

- `var` : Oui ✓ Une variable déclarée avec `var` en dehors d'une fonction est ajoutée à l'objet global (`window` dans les navigateurs).
- `let` : Non ✗ Une variable déclarée avec `let` en dehors d'un bloc ou d'une fonction n'est pas ajoutée à l'objet global, mais elle est accessible globalement dans le script.
- `const` : Non ✗ Comme `let`, `const` n'est pas ajouté à l'objet global, mais est accessible globalement dans le script.

Let, var et const

Précision :

La phrase "La portée (ou scope) d'une fonction est la seule à mettre toutes les variables sur un même pied d'égalité" est un peu trompeuse.

En réalité, var se comporte différemment de let et const même dans une fonction, car var n'est pas limité aux blocs (comme les boucles for, if, etc.).

(2)

Se limite à la portée d'une fonction ?

- **var** : Oui ✓ var est limité à la portée de la fonction où il est déclaré. Si déclaré en dehors de toute fonction, il est global.
- **let** : Oui ✓ let est limité à la portée du bloc (ou de la fonction) où il est déclaré.
- **const** : Oui ✓ const est aussi limité à la portée du bloc (ou de la fonction) où il est déclaré.

Let, var et const

Attention, une référence constante ne veut pas dire que la valeur derrière la référence est "immutable", Cela ne veut PAS dire que la valeur stockée est immuable.

(3)

Un bloc d'instruction

- `var` : Non ❌ Une variable déclarée avec var dans un bloc (comme un for, if, etc.) n'est pas limitée à ce bloc, elle est accessible dans toute la fonction ou globalement si elle est déclarée en dehors d'une fonction.
- `let` : Oui ✓ let est limité au bloc où il est déclaré.
- `const` : Oui ✓ const est aussi limité au bloc où il est déclaré.

Let, var et const

Réassigné et redéclaré ?

(4)

Peut être réassigné ?

- `var` : Oui ✓
- `let` : Oui ✓
- `const` : Non ✗ Une variable déclarée avec `const` ne peut pas être réassignée après son initialisation.

Peut être redéclaré ?

- `var` : Oui ✓ `var` peut être redéclaré dans le même scope.
- `let` : Non ✗ `let` ne peut pas être redéclaré dans le même scope.
- `const` : Non ✗ `const` ne peut pas être redéclaré dans le même scope.

Hoisting

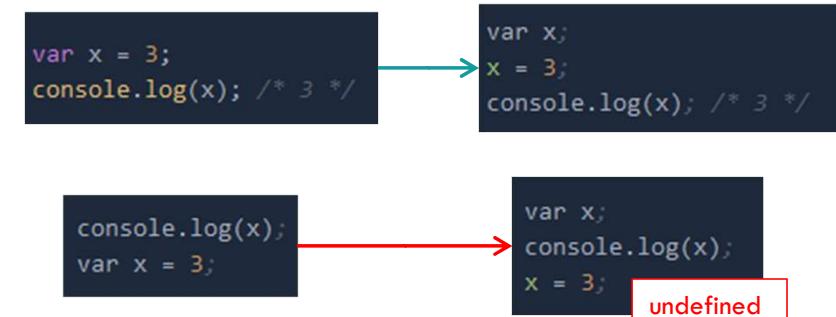
Cette mécanique consiste donc à faire virtuellement remonter la déclaration d'une variable (ou d'une fonction) tout en haut de son scope lors de l'analyse du code par le moteur d'interprétation Javascript.

- C'est une mécanique automatique et obligatoire qui fait partie de la spécification ECMAScript même si le terme hoisting n'y apparait pas en tant que tel.

5

Est affecté par le hoisting ?

- var : Oui ✓ Les variables déclarées avec var sont "hoistées" (remontées) en haut de leur scope et initialisées avec undefined.
- let : Non ✗ Les variables déclarées avec let sont "hoistées" mais ne sont pas initialisées, ce qui crée une "Temporal Dead Zone" (TDZ) jusqu'à leur déclaration.
- const : Non ✗ Comme let, const est "hoisté" mais n'est pas initialisé, ce qui crée aussi une TDZ.



Conclusion

Règles simples pour éviter les problèmes

①

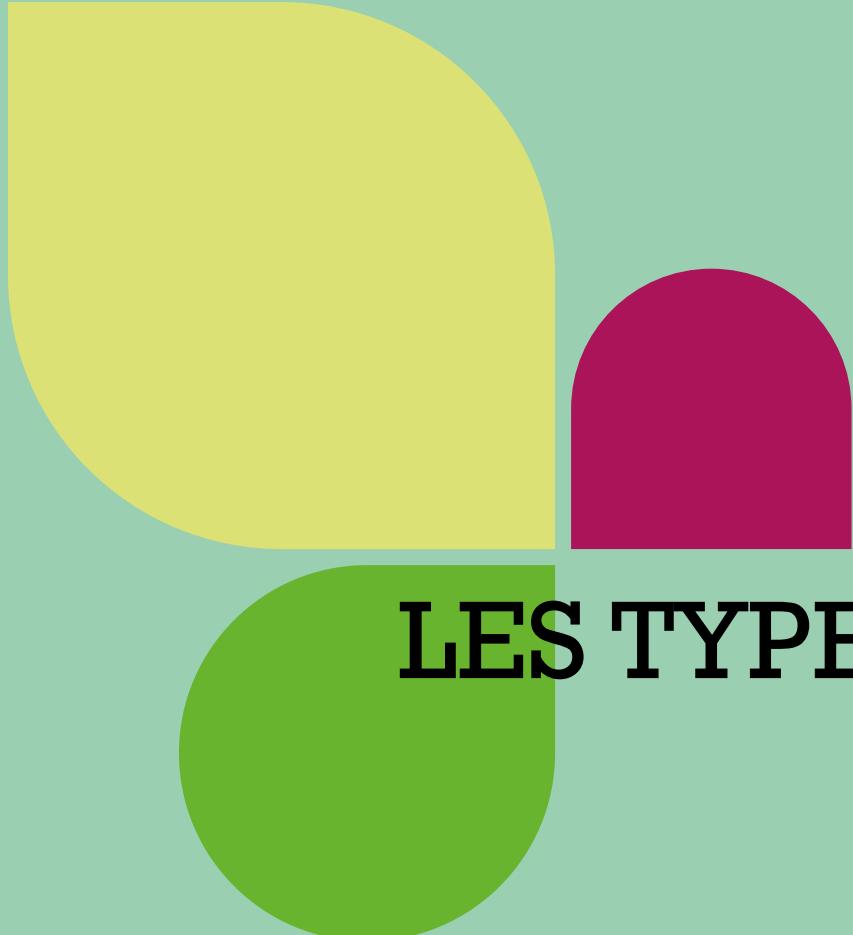
Déclare toujours tes variables avant de les utiliser.

②

Utilise const quand la valeur ne change pas.

③

Utilise let pour les valeurs qui évoluent. Évite var sauf cas très particulier (*code legacy = ancien code*).



Les types

Primitif

En javascript, on appelle ces types, des valeurs primitives, fondamentaux qui ne sont pas des objets.

- Cependant, il est important de noter que JavaScript traite parfois les primitives comme des objets temporaires pour permettre l'accès aux méthodes et propriétés

Chaînes de caractères.	<code>String</code>
Nombres (entiers, décimaux, NaN, Infinity) et les grands entiers	<code>Number</code> <code>bignint</code>
Valeur vide volontaire.	<code>null</code>
Vrai ou faux.	<code>boolean</code>
Valeur d'une variable non initialisée.	<code>undefined</code>
Valeur unique et immuable, souvent utilisées comme clés d'objet. <u>https://fr.javascript.info/symbol</u>	<code>symbol</code>

Les types

Non primitif



Objet

Structure clé/valeur ou instances plus complexes.



Les objets incluent :

- Array
- Function
- Date
- RegExp
- Map / Set
- WeakMap / WeakSet
- Error

`typeof()` permet de vérifier le type en cours.

... et tous les objets personnalisés.

⚠ En JavaScript, tout ce qui n'est pas un type primitif est un objet.



Les opérateurs :
*On retrouve la plupart des opérateurs
que nous connaissons dans le
développement.
Mais il existe quelques particularités !!*

L'opérateur de coalescence des nuls ??

L'opérateur de coalescence des nuls est écrit sous la forme de deux points d'interrogation ??

Le résultat de a ?? b est :

- si a est défini, alors a,
- si a n'est pas défini, alors b.

Nous pouvons également utiliser une séquence de ?? pour sélectionner la première valeur dans une liste qui n'est pas null/undefined.

Disons que nous avons les données d'un utilisateur dans les variables firstName, lastName ou nickName. Tous peuvent être indéfinis, si l'utilisateur décide de ne pas entrer de valeurs correspondantes.

Nous aimerais afficher le nom d'utilisateur à l'aide de l'une de ces variables, ou afficher "Anonyme" si toutes sont null/undefined.

Utilisons l'opérateur ?? pour cela :

```
let firstName = null;
let lastName = null;
let nickName = "Supercoder";

// affiche la première valeur définie :
alert(firstName ?? lastName ?? nickName ?? "Anonymous"); // Supercoder

// ancienne manière de l'écrire
alert(firstName || lastName || nickName || "Anonymous"); // Supercoder
```

Historiquement, l'opérateur OR || était là en premier. Il existe depuis le début de JavaScript, donc les développeurs l'utilisaient à de telles fins depuis longtemps.

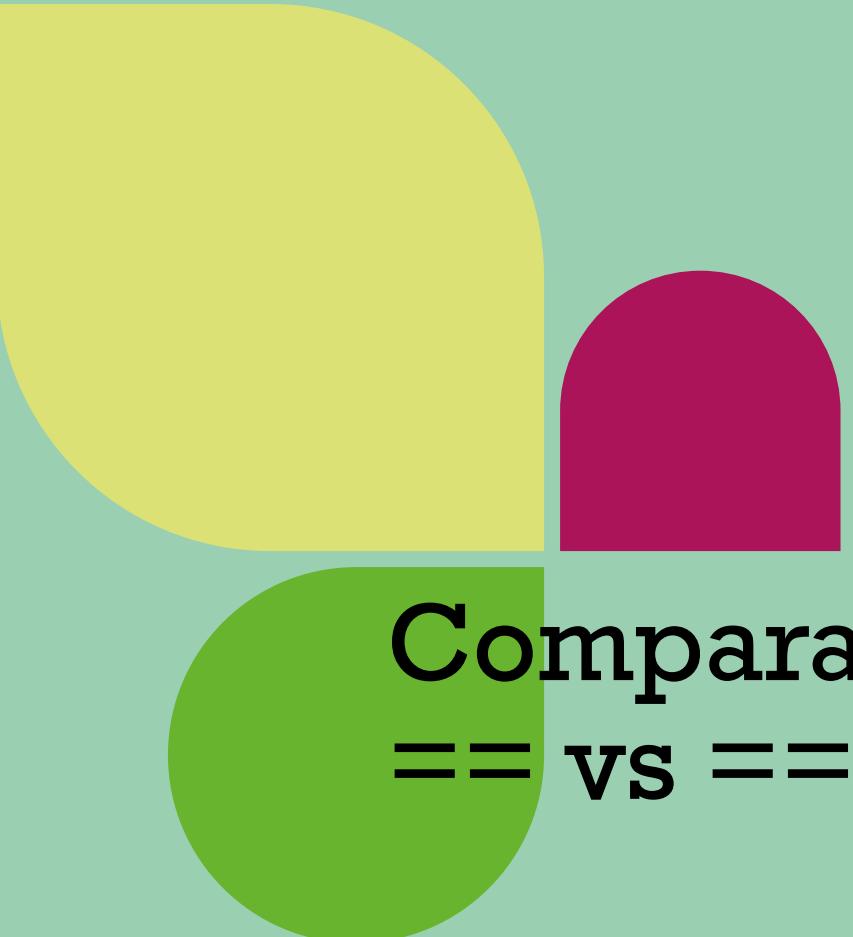
D'un autre côté, l'opérateur de coalescence des nuls ?? n'a été ajouté à JavaScript que récemment, et la raison en était que les gens n'étaient pas tout à fait satisfaits de || .

La différence importante entre eux est que :

- || renvoie la première valeur vraie.
- ?? renvoie la première valeur définie.

```
let height = 0;

alert(height || 100); // 100
alert(height ?? 100); // 0
```



Comparaison

== vs ==

== VS ===

Comparaison

`==` correspond à une comparaison d'égalité abstraite

`====` correspond à une comparaison d'égalité stricte

Il vaut mieux privilégier l'utilisation de l'égalité stricte

On considère que ce n'est jamais une bonne idée d'utiliser l'égalité faible.

- Le résultat d'une comparaison utilisant l'égalité stricte est plus simple à appréhender et à prédire, de plus il n'y a aucune conversion implicite ce qui rend le test plus rapide.

https://developer.mozilla.org/fr/docs/Web/JavaScript/Equality_comparisons_and_sameness

- L'égalité faible (`==`) effectuera une conversion des deux éléments à comparer avant d'effectuer la comparaison
- L'égalité stricte (`====`) effectuera la même comparaison mais sans conversion préalable (elle renverra toujours false si les types des deux valeurs comparées sont différents)

```
var num = 0;
var obj = new String("0");
var str = "0";

console.log(num === num); // true
console.log(obj === obj); // true
console.log(str === str); // true

console.log(num === obj); // false
console.log(num === str); // false
console.log(obj === str); // false
console.log(null === undefined); // false
console.log(obj === null); // false
console.log(obj === undefined); // false
```

```
var num = 0;
var obj = new String("0");
var str = "0";

console.log(num == num); // true
console.log(obj == obj); // true
console.log(str == str); // true

console.log(num == obj); // true
console.log(num == str); // true
console.log(obj == str); // true
console.log(null == undefined); // true

// Les deux assertions qui suivent sont fausses
// sauf dans certains cas exceptionnels
console.log(obj == null);
console.log(obj == undefined);
```

L'égalité faible avec ==

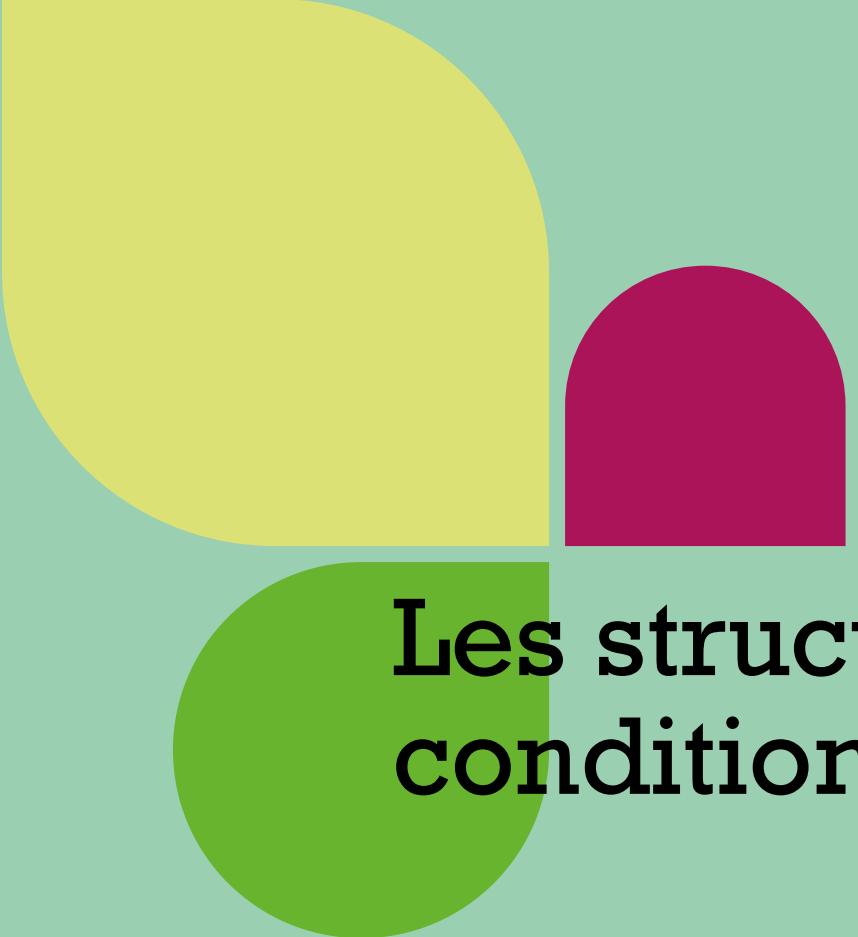
Conversions

Le test d'égalité faible compare deux valeurs après les avoir converties en valeurs d'un même type.

- Une fois converties (la conversion peut s'effectuer pour l'une ou les deux valeurs), la comparaison finale est la même que celle effectuée par ===.
- L'égalité faible est symétrique : A == B aura toujours la même signification que B == A pour toute valeur de A et B.
- ToNumber(A) correspond à une tentative de convertir l'argument en un nombre avant la comparaison.
- ToPrimitive(A) correspond à une tentative de convertir l'argument en une valeur primitive grâce à plusieurs méthodes comme A.toString et A.valueOf.

Tableau des conversions

		Opérande B					
		Undefined	Null	Number	String	Boolean	Object
Opérande A	Undefined	true	true	false	false	false	false
	Null	true	true	false	false	false	false
	Number	false	false	A === B	A === ToNumber(B)	A === ToNumber(B)	A == ToPrimitive(B)
	String	false	false	ToNumber(A) === B	A === B	ToNumber(A) === ToNumber(B)	A == ToPrimitive(B)
	Boolean	false	false	ToNumber(A) === B	ToNumber(A) === ToNumber(B)	A === B	false
	Object	false	false	ToPrimitive(A) == B	ToPrimitive(A) == B	ToPrimitive(A) == ToNumber(B)	A === B



Les structures conditionnelles

DÉCOUVERTE DU LANGAGE

Les conditions

```
// l'expression if  
if (condition) une_instruction;  
  
if (condition) {  
    instruction1;  
} else if (autre_condition) {  
    instruction2;  
} else {  
    instruction3;  
}  
  
// ternaire  
(test_condition) ? valeur_vrai : valeur_faux;
```

Switch

```
// Switch  
var animal = "oiseau";  
switch(animal) {  
    case "chien": ...  
    case "oiseau" : ...  
    case "poisson": ...  
    case "vache" :  
        console.log("C'est un vertébré");  
    break;  
    case "mouche" : ...  
    default :  
        console.log("C'est un invertébré");  
}
```

DÉCOUVERTE DU LANGAGE

Les répétitions

```
// Les répétitions
for (var i=0; i<100; i++) {
    console.log("Préfère la boucle for si tu connais le nombre !");
}

var i;
while (!i) {
    i = confirm("As-tu compris ?");
}

do { // l'instruction suivante sera exécutée au moins 1 fois !
    i = prompt("laisse vide ou annule");
} while (i);

break;      // pour arrêter une boucle for ou while
continue;   // pour sauter une instruction ou passer à l'itération suivante
```

Parcours de tableaux

```
const passengers = [
    "Will Alexander",
    "Sarah Kate",
    "Audrey Simon",
    "Tao Perkington"
];

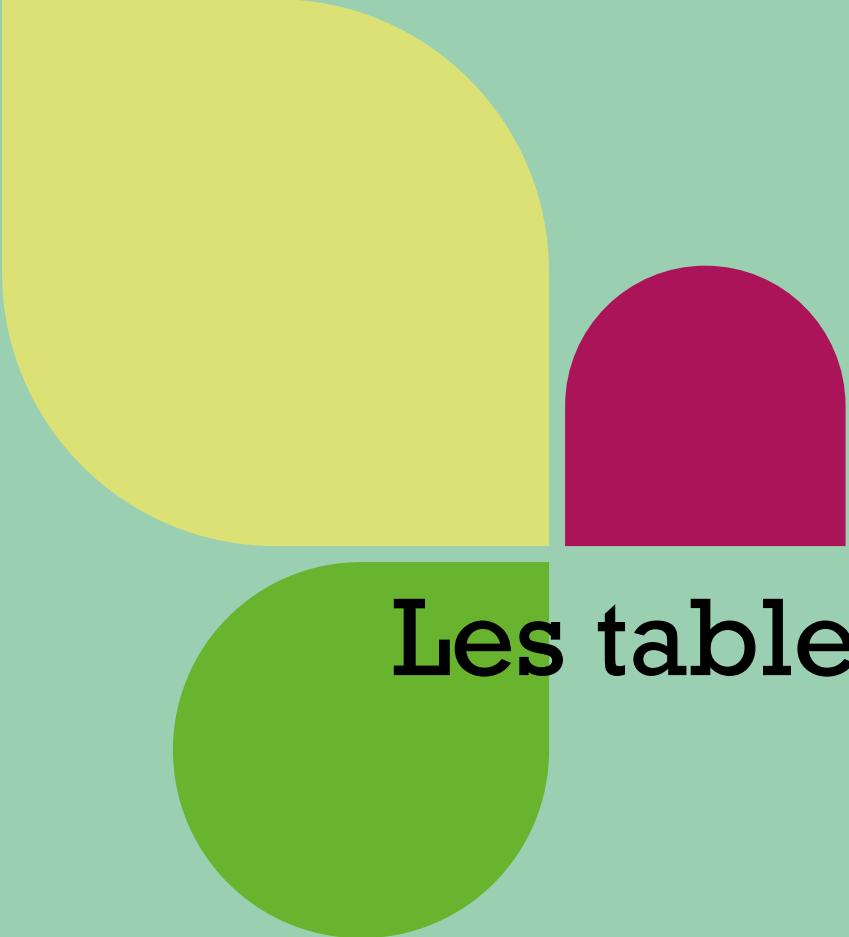
for (let i in passengers) {
    console.log("Embarquement du passager " + passengers[i]);
}

const list = [
    "Will Alexander",
    "Sarah Kate",
    "Audrey Simon",
    "Tao Perkington"
];

for (let passenger of passengers) {
    console.log("Embarquement du passager " + passenger);
}
```

avec
l'indice

avec les
éléments du
tableau



Les tableaux

Les tableaux

En Javascript, nous avons la possibilité de créer des tableaux

https://developer.mozilla.org/fr/docs/Learn_web_development/Core/Scripting/Arrays



création

```
let tab=[valeur1, ..., valeurN];  
let tab=new Array();
```

Indice de départ : 0



recherche

```
find()
```



suppression

```
pop()
```



ajout

```
push()
```



Parcourir

```
tab.forEach(num -> console.log(num));
```



Glossaire

Méthodes utiles à retenir pour les tableaux



Les tableaux

objets très puissants avec plusieurs méthodes pratiques pour manipuler et travailler avec leurs éléments

push()

Ajoute un ou plusieurs éléments à la fin d'un tableau et retourne la nouvelle longueur du tableau.

`let arr = [1, 2]; arr.push(3); arr devient [1, 2, 3]`

pop()

Supprime le dernier élément d'un tableau et le retourne.

`let arr = [1, 2, 3]; arr.pop(); arr devient [1, 2] et donne 3`

shift()

Supprime le premier élément d'un tableau et le retourne.

`let arr = [1, 2, 3]; arr.shift(); arr devient [2, 3] et donne 1`

unshift()

Ajoute un ou plusieurs éléments au début d'un tableau et retourne la nouvelle longueur du tableau.

`let arr = [2, 3]; arr.unshift(1); arr devient [1, 2, 3]`

concat()

Fusionne 2 ou + tableaux et retourne un nouveau tableau.

`let arr1 = [1, 2]; let arr2 = [3, 4];
let arr3 = arr1.concat(arr2); arr3 devient [1, 2, 3, 4]`

Les tableaux

objets très puissants avec plusieurs méthodes pratiques pour manipuler et travailler avec leurs éléments

join()

Retourne une chaîne de caractères formée par la concaténation des éléments du tableau, séparés par un délimiteur spécifié.

```
let arr = ['a', 'b', 'c']; let str = arr.join(','); "a, b, c"
```

slice()

Retourne une copie superficielle d'une portion du tableau dans un nouveau tableau.

```
let arr = [1, 2, 3, 4]; let sliced = arr.slice(1, 3); [2, 3]
```

splice()

Permet de modifier un tableau en ajoutant, supprimant ou remplaçant des éléments à partir d'une position donnée.

```
let arr = [1, 2, 3, 4]; arr.splice(2, 1, 5);  
arr devient [1, 2, 5, 4] (remplace 3 par 5)
```

forEach()

Exécute une fonction donnée sur chaque élément du tableau.

```
let arr = [1, 2, 3]; arr.forEach(num => console.log(num));  
Affiche 1, 2, 3
```

map()

Crée un nouveau tableau avec les résultats de l'appel d'une fonction sur chaque élément du tableau d'origine.

```
let arr = [1, 2, 3]; let doubled = arr.map(num => num * 2);  
[2, 4, 6]
```

Les tableaux

objets très puissants avec plusieurs méthodes pratiques pour manipuler et travailler avec leurs éléments

filter()

Crée un nouveau tableau contenant uniquement les éléments qui passent un test (défini par une fonction).

```
let arr = [1, 2, 3, 4]; let evenNumbers = arr.filter(num => num % 2 === 0); donne [2, 4]
```

reduce()

Applique une fonction sur chaque élément du tableau (de gauche à droite) et retourne une seule valeur qui est le résultat de la réduction.

```
let arr = [1, 2, 3]; let sum = arr.reduce((acc, num) => acc + num, 0); donne 6
```

find()

Retourne le premier élément du tableau qui satisfait une condition donnée. let arr = [1, 2, 3, 4];

```
let found = arr.find(num => num > 2); donne 3
```

indexOf()

Retourne l'index du premier élément trouvé qui correspond à la valeur donnée, ou -1 si l'élément n'existe pas

includes()

Vérifie si un élément est présent dans le tableau et retourne true ou false.

Trie décroissant : $(a,b) \Rightarrow b - a$
 Trie croissant : $(a,b) \Rightarrow a - b$



Explication : la fonction $(a, b) \Rightarrow a - b$ doit **retourner** :
un nombre négatif → a doit venir **avant** b
un nombre positif → a doit venir **après** b
0 → l'ordre ne change pas

Ensuite sort() se charge de l'inversion ou pas

Les tableaux

objets très puissants avec plusieurs méthodes pratiques pour manipuler et travailler avec leurs éléments

sort()



trie **par ordre lexicographique** (comme du texte).
 Pour les entiers, il faut donc donner une fonction de comparaison : `nombres.sort((a, b) => a - b);`



reverse()



Inverse l'ordre des éléments dans le tableau.

some()



Vérifie si au moins un élément du tableau passe un test (renvoie true ou false). `let arr = [1, 2, 3]; let hasEven = arr.some(num => num % 2 === 0); // true`

every()



Vérifie si tous les éléments du tableau passent un test. `let arr = [2, 4, 6]; let allEven = arr.every(num => num % 2 === 0); // true`

findIndex()



Retourne l'index du premier élément qui satisfait une condition. `let arr = [1, 2, 3, 4]; let index = arr.findIndex(num => num > 2); // 2`

Glossaire

Méthodes utiles de la classe Math



Classe Math

Elle offre plusieurs méthodes très utiles pour effectuer des calculs mathématiques.

`abs(x)`

Retourne la valeur absolue de x (c'est-à-dire la distance de x à zéro, indépendamment de son signe).

`round(x)`

Retourne l'entier le plus proche de x . Si x est à égalité avec deux entiers, le plus proche vers zéro est retourné.

`floor(x)`

Retourne l'entier inférieur à x (arrondi à l'entier le plus bas).

`ceil(x)`

Retourne l'entier supérieur à x (arrondi à l'entier le plus haut).

`max(x, y, ...)`

Retourne le plus grand des nombres passés en arguments. Ça marche pour les tableaux :
`Math.max(...arr);` ... opérateur spread

Classe Math

Elle offre plusieurs méthodes très utiles pour effectuer des calculs mathématiques.

`min(x, y, ...)`

Retourne le plus petit des nombres passés en arguments.

`random()`

Retourne un nombre pseudo-aléatoire entre 0 (inclus) et 1 (exclus).

`pow(base, exponent)`

Retourne base élevée à la puissance de exponent.

`sqrt(x)`

Retourne la racine carrée de x.

`sin(x)`
`cos(x)`
`tan(x)`

Ces méthodes retournent respectivement le sinus, le cosinus et la tangente de l'angle x exprimé en radians.

Classe Math

Elle offre plusieurs méthodes très utiles pour effectuer des calculs mathématiques.

`log(x)`

Retourne le logarithme naturel (base e) de x. Si vous voulez un logarithme à une autre base, vous pouvez utiliser une formule comme `Math.log(x) / Math.log(base)` pour la base que vous souhaitez.

`exp(x)`

Retourne e élevé à la puissance de x (soit `Math.E^x`).

`trunc(x)`

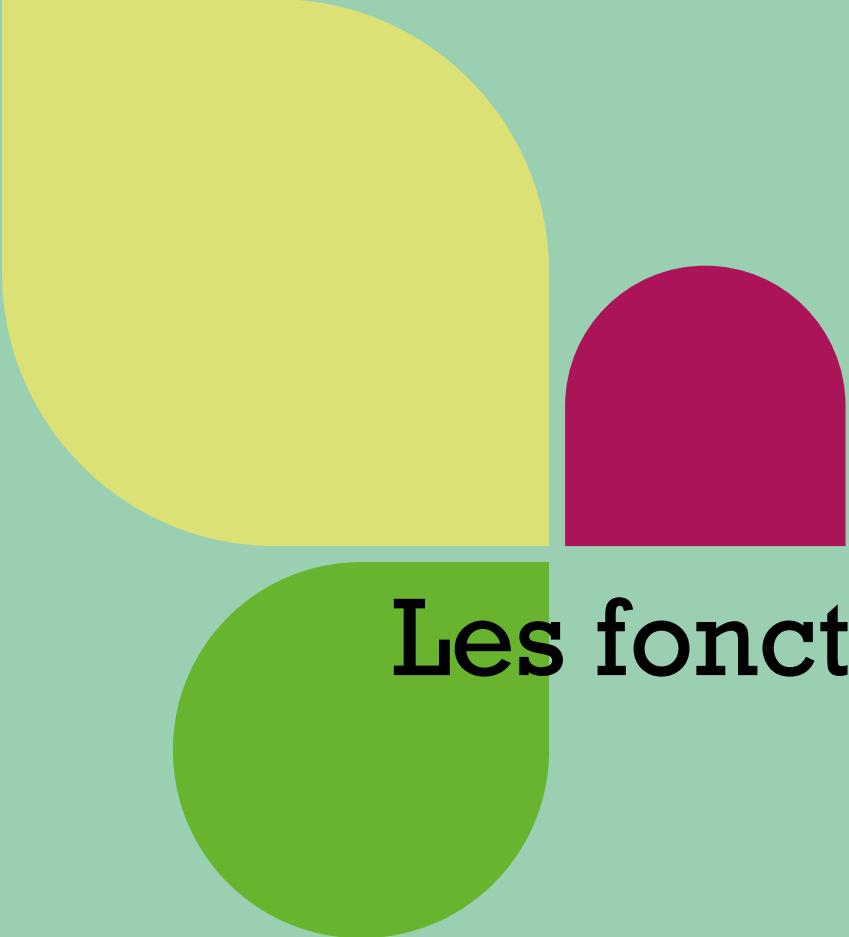
Retourne la partie entière de x en retirant sa partie décimale.

`PI`

Représente la constante π (pi) en JavaScript.

`E`

Représente la constante e (la base des logarithmes naturels).



Les fonctions

LES FONCTIONS

Définie dans un bloc d'instruction {} à un seul endroit du script, réutilisable et exécutable par un simple appel depuis le script ou depuis une autre fonction.

Un mot-clé **function** permettant de définir ses propres fonctions.

Deux types de fonctions :

- Avec résultat → utilisation d'un **return**
- Sans résultat → la fonction devient une procédure.

```
function nom([param[, param[, ... param]]]) {  
    instructions  
}
```

JavaScript regorge de fonctions natives :

<https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Functions>

Par exemple :

- isNaN(x) true si le paramètre x n'est pas un nombre.
- parseFloat(string) convertit la chaîne en nombre à virgule flottante.
- parseInt(string) convertit la chaîne en entier.

Il existe plusieurs sortent de fonctions :

- Les instructions de fonctions (les plus courantes),
- Les expressions de fonctions,
- Les fonctions anonymes (qui servent à isoler une partie du code),
- Les fonctions « Callback »,
- Les fonctions auto-exécutables,
- Les fonctions issues d'un objet (les méthodes et les constructeurs).

LES FONCTIONS

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Functions#objet_arguments

Instructions de fonctions

Syntaxe :

```
Function myName (paramètres) { instructions };
```

Appel :

```
myName(paramètres);
```

Exemples :

```
function coucou() { alert("coucou"); }

function calculeSurface(largeur, hauteur) {
    return largeur * hauteur;
}
```

Les paramètres des fonctions

Plus souple, JavaScript n'impose pas de respect strict des paramètres attendus.

A chaque appel d'une fonction, l'objet arguments stocke tous les paramètres envoyés.

C'est au développeur de faire en sorte de traiter la surcharge de la méthode

```
function perimetre(largeur, Longueur) {
    var res = 0;
    // test si au moins un paramètre reçu
    if (!largeur) res = 0;
    // 1 param reçu : carré
    else if (!longueur) { res = 4*largeur; }
    // 2 param : rectangle
    else if (arguments.length == 2) { res = (largeur + longueur)*2; }
    // polygone
    else { for (i in arguments) res += arguments[i]; }
    console.log(resultat);
}
```

PARAMÈTRES DES FONCTIONS

valeur par défaut

On a la possibilité d'indiquer une valeur par défaut pour nos paramètres.

```
function multiply(a, b = 1) {
  return a * b;
}

console.log(multiply(5, 2));
// Expected output: 10

console.log(multiply(5));
// Expected output: 5
```

Rest Parameters

Cela permet de représenter un nombre indéfini d'arguments sous forme d'un tableau.

```
function sum(...theArgs) {
  let total = 0;
  for (const arg of theArgs) {
    total += arg;
  }
  return total;
}

console.log(sum(1, 2, 3));
// Expected output: 6

console.log(sum(1, 2, 3, 4));
// Expected output: 10
```

LES FONCTIONS

Les expressions de fonctions

Les expressions de fonctions passent par la création d'une variable affectée par la définition d'une fonction.

- peuvent être écrite n'importe où dans le code
- Ne peuvent pas être appelées avant d'avoir été déclarées.

```
let surface = function calculeSurface(Largeur, hauteur) {  
    return Largeur * hauteur;  
}
```

```
// utilisation de la variable  
surface(4,6);
```

Fonctions anonymes

La fonction ne porte pas de nom. On dit qu'elle est anonyme.

- Très utilisées notamment dans la gestion d'évènements, les objets, les closures et les callback...
- Surcharge le code au détriment de sa lisibilité

```
// se lance toutes les secondes  
var decomppte = setInterval(function() {  
    // décompte de 10 à 1  
    console.log(i--);  
}, 1000);
```



Les fonctions callback, auto-exectucables, flechées et imbriquées

CALLBACKS

Un callback est une fonction qui est passée comme argument à une autre fonction et qui est exécutée après qu'une certaine opération a été complétée.

- Les callbacks sont couramment utilisés dans les opérations asynchrones, comme les requêtes réseau ou les opérations de fichiers.

Dans cet exemple simple, `sayGoodbye` est une fonction de callback qui est exécutée après que `greet` a terminé son exécution.

```
function greet(name, callback) {
    console.log('Hello ' + name);
    callback();
}

function sayGoodbye() {
    console.log('Goodbye!');
}

greet('Alice', sayGoodbye);
```

AUTO- EXÉCUTABLES

Une fonction auto-exécutable (IIFE, Immediately Invoked Function Expression) est une fonction qui est définie et exécutée immédiatement après sa définition.

- Les IIFE sont souvent utilisées pour créer une portée privée et éviter la pollution de l'espace global.

La syntaxe impose simplement de faire suivre la définition de la fonction d'une paire de parenthèses afin de provoquer son exécution.

- Pour provoquer l'exécution immédiate d'une fonction, on peut encore l'englober dans une autre paire de parenthèses sans oublier de la faire suivre par sa paire de parenthèses

```
var test = function() {
  console.log('hello world');
}();

(function() {
  console.log('hello world');
}());

(function() {
  console.log('hello world');
})();
```

FONCTIONS FLÉCHÉES

- Syntaxe très compacte, rapide à écrire, utilisant le signe =>

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Functions/Arrow_functions

```
([param] [, param]) => {
  /* instructions */
}

(param1, param2, paramx) => expression;

// équivalent à
(param1, param2, paramx) => {
  return expression;
}

// Parenthèses non nécessaires quand il n'y a qu'un seul argument
param => expression;

// Une fonction sans paramètre peut s'écrire avec un couple
// de parenthèses
() => {
  /* instructions */
}

(param1 = valeurDefaut1, param2, paramN = valeurDefautN) => {
  /* instructions */
}
```

FONCTIONS IMBRIQUÉES

- Une fonction peut avoir une ou plusieurs fonctions internes.
- Ces fonctions imbriquées sont dans la portée de la fonction externe.
- La fonction interne peut accéder aux variables et aux paramètres de la fonction externe.
 - Cependant, la fonction externe ne peut pas accéder aux variables définies dans les fonctions internes.

```
function afficheMessage(prenom)
{
    function disBonjour() {
        alert("Bonjour " + prenom);
    }
    return disBonjour();
}

afficheMessage("Michel"); // Affiche Bonjour Michel
```

FONCTIONS IMBRIQUÉES AVEC CLOSURES

Les [closures](#) permettent à une fonction imbriquée d'accéder aux variables de la portée englobante, même après que la fonction externe a terminé son exécution.

```
// Définition de la fonction createCounter
function createCounter() {
    // Initialisation de la variable count à 0
    let count = 0;

    // Retourne une fonction anonyme
    return function() {
        // Incrémente count de 1
        count++;
        // Retourne la nouvelle valeur de count
        return count;
    };
}

// Appel de createCounter et assignation de la fonction retournée à counter
const counter = createCounter();

// Appel de la fonction retournée (counter)
console.log(counter()); // 1 (count est incrémenté de 0 à 1)
console.log(counter()); // 2 (count est incrémenté de 1 à 2)
console.log(counter()); // 3 (count est incrémenté de 2 à 3)
```

```
// Définition de la fonction externe
function outerFunction(outerVariable) {
    // Définition de la fonction interne
    return function innerFunction(innerVariable) {
        // Affichage des variables
        console.log('Outer Variable:', outerVariable);
        console.log('Inner Variable:', innerVariable);
    };
}

// Appel de la fonction externe avec 'outside'
const newFunction = outerFunction('outside');
// newFunction est maintenant une référence à innerFunction
// avec outerVariable fixé à 'outside'

// Appel de la fonction interne avec 'inside'
newFunction('inside');
// innerFunction affiche 'Outer Variable: outside'
// et 'Inner Variable: inside'
```

- outerFunction retourne une fonction imbriquée (innerFunction) qui capture la variable outerVariable grâce à la closure.
- newFunction est une référence à innerFunction avec outerVariable fixé à 'outside'.
- Lorsque vous appelez newFunction('inside'), innerFunction affiche les deux variables : outerVariable (qui est 'outside') et innerVariable (qui est 'inside').

La fonction anonyme capture la variable count grâce à la closure, ce qui permet de maintenir l'état de count entre les appels.



Les objets

Les Objets

Les objets Javascripts sont écrits en JSON par des séries de paires clés-valeurs séparées par des virgules, entre des accolades.

```
let myBook = {
  title: 'The Story of Tau',
  author: 'Will Alexander',
  numberOfPages: 250,
  isAvailable: true
};
```

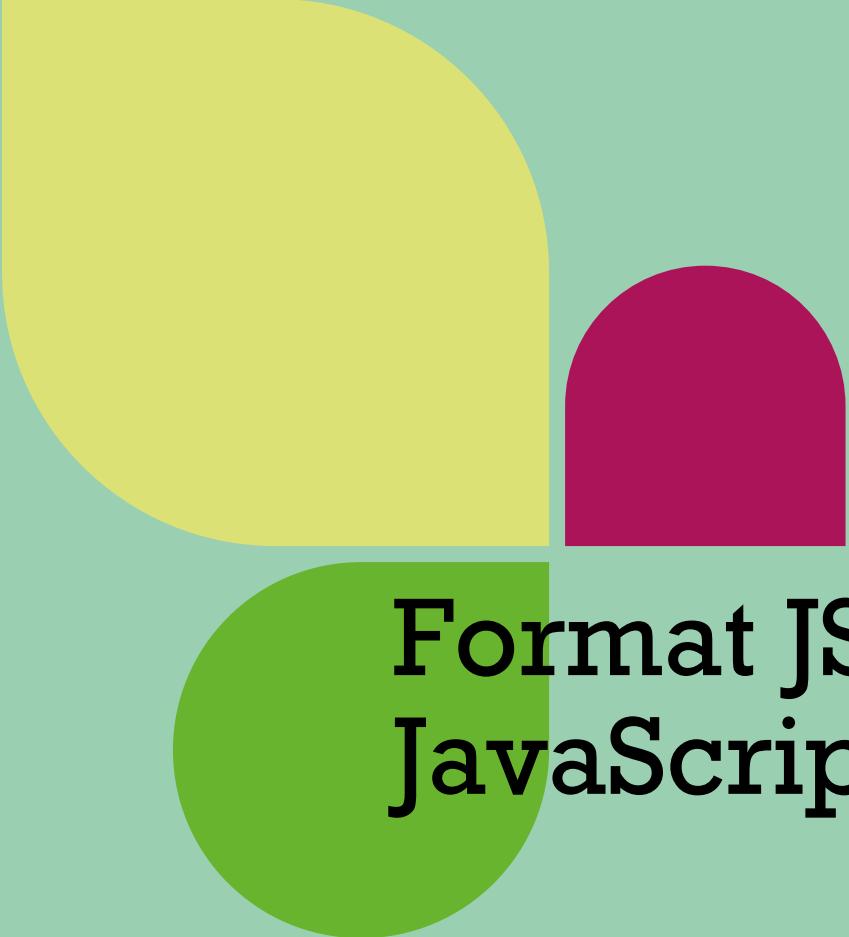


Accès

```
let title = myBook.title;
let author = myBook["author"];
```

Freeze et Seal

```
1  /*
2   * Object.freeze
3   * - L'objet est complètement bloqué.
4   * - Impossible de modifier, ajouter ou supprimer des propriétés.
5   */
6 const hero = { name: 'Superman', power: 'Flight' };
7 Object.freeze(hero);
8
9 hero.power = 'Invisibility'; // ✗ Pas possible
10 hero.city = 'Metropolis'; // ✗ Pas possible
11 delete hero.name; // ✗ Pas possible
12
13 console.log(hero); // { name: 'Superman', power: 'Flight' }
14
15
16 /*
17  * Object.seal
18  * - L'objet est scellé.
19  * - Impossible d'ajouter ou de supprimer des propriétés.
20  * - Les propriétés existantes peuvent être modifiées.
21  */
22 const hero2 = { name: 'Thor', power: 'Lightning' };
23 Object.seal(hero2);
24
25 hero2.power = 'Stormbreaker'; // ✓ Possible
26 hero2.weapon = 'Mjolnir'; // ✗ Pas possible (ajout impossible)
27 delete hero2.name; // ✗ Pas possible
28
29 console.log(hero2); // { name: 'Thor', power: 'Stormbreaker' }
30
```



Format JSON JavaScript Object Notation

JSON

Format du web

JSON (JavaScript Object Notation) est un format de fichier textuel conçu pour l'échange de données.

Il représente des données structurées basées sur la syntaxe des objets du langage de programmation JavaScript.

De ce fait, un programme JavaScript peut convertir des données JSON en objets JavaScript natifs sans analyser ou sérialiser les données.

JSON est populaire.

- en raison de son style autodescriptif,
- facile à comprendre,
- léger et compact,
- compatible avec de nombreux langages de programmation, environnements et bibliothèques.

La notation d'objets JavaScript (JSON) est un format textuel lisible par l'homme, conçu pour l'échange de données.

Il est pris en charge par de nombreux langages de programmation, environnements et bibliothèques.

JSON est remarquable car il permet aux utilisateurs de demander des données à travers les domaines en utilisant la fonction JSONP*. De plus, il est plus simple et plus léger que XML.

Définition

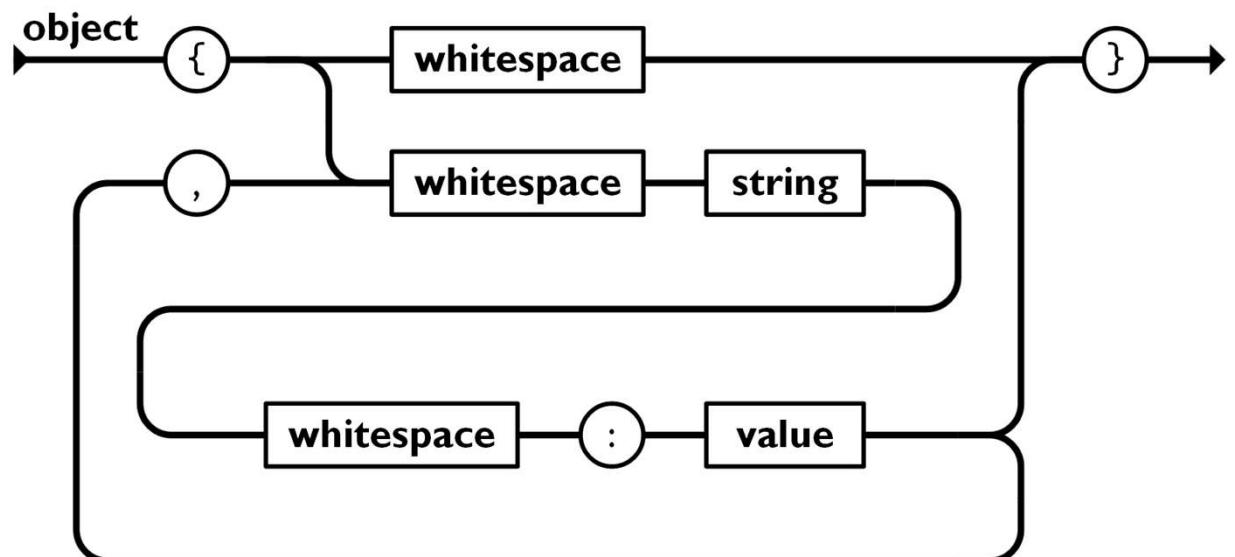
la syntaxe Json

Voici les principaux éléments de la syntaxe JSON :

- Les données sont présentées sous forme de paires clé/valeur.
- Les éléments de données sont séparés par des virgules.
- Les crochets {} désignent les objets.
- Les crochets [] désignent des tableaux.
- Par conséquent, la syntaxe des littéraux d'objets JSON ressemble à ceci :

```
{"key": "value", "key": "value", "key": "value".}
```

Json



Les types de valeurs

Les différents types possibles

Les chaînes de caractères :

- { "firstName" : "Tom" }

Nombre

- { "age" : 30 }

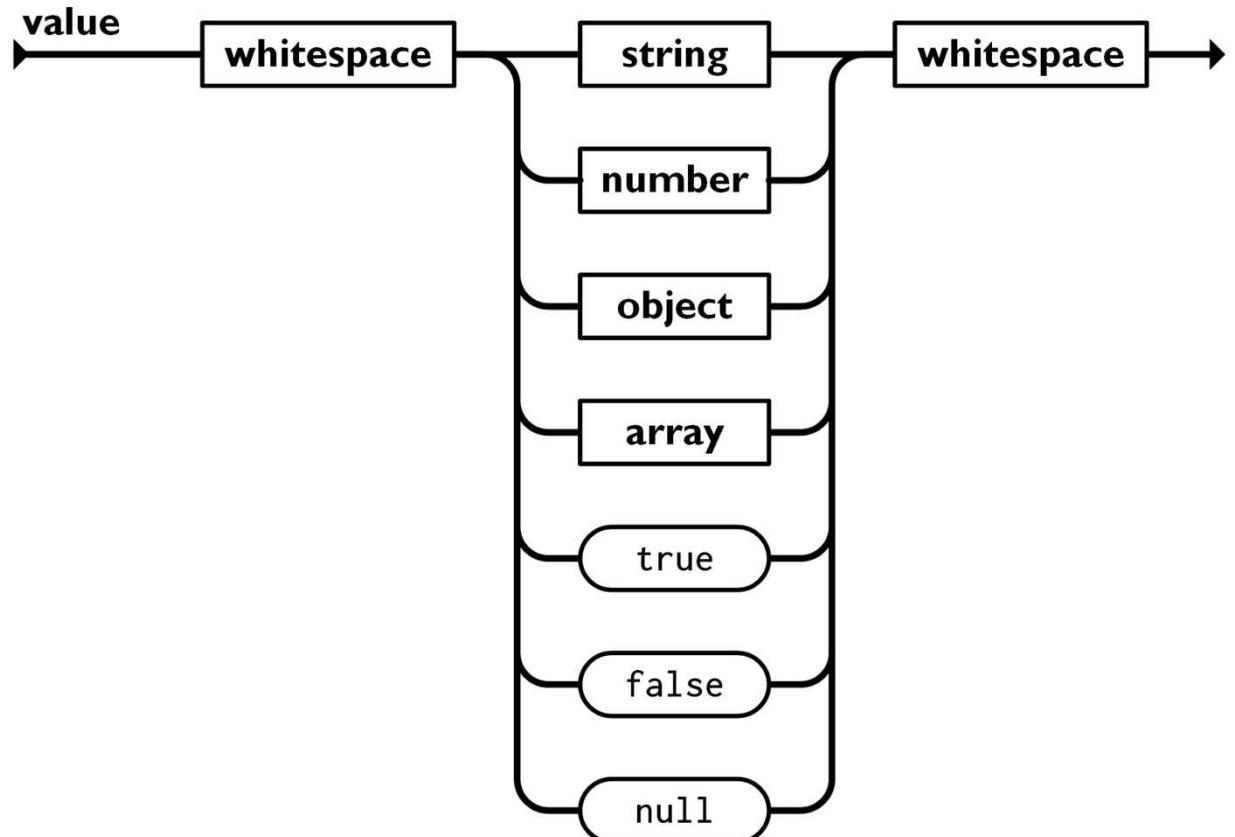
Booléen

- { "married" : false }

Null : Null est une valeur vide.

- { "bloodType" : null }

Les différents types



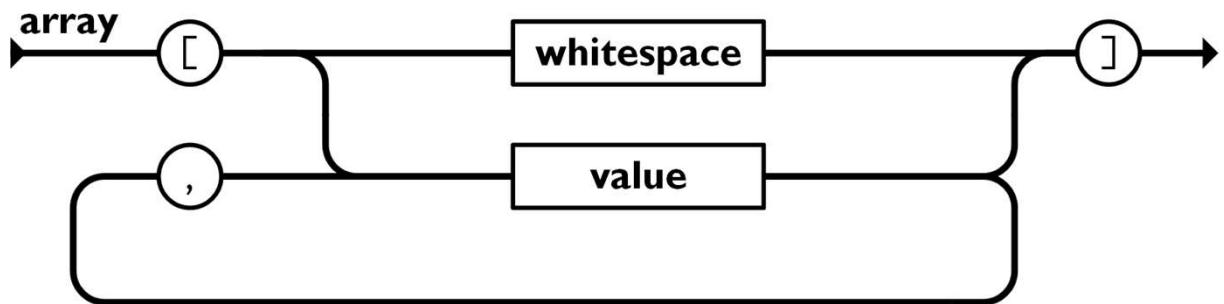
Les tableaux

Utilisation

Un tableau est une collection ordonnée de valeurs.

- Une valeur de tableau peut contenir des objets JSON, ce qui signifie qu'elle utilise le même concept de paire clé/valeur.

déclaration



```
{  
  "students": [  
    {"firstName": "Tom", "lastName": "Jackson"},  
    {"firstName": "Linda", "lastName": "Garner"},  
    {"firstName": "Adam", "lastName": "Cooper"}  
]
```

Les objets JSON

Les objets JSON sont constitués de paires de deux composants constitués d'une clé et de sa valeur

- Les clés sont des chaînes de caractères – des séquences de caractères entourées de guillemets.
- Les valeurs sont des types de données JSON valides. Elles peuvent se présenter sous la forme d'un tableau, d'un objet, d'une chaîne de caractères, d'un booléen, d'un nombre ou de null.
- Un deux-points est placé entre chaque clé et chaque valeur, et une virgule sépare les paires. Les deux éléments sont placés entre guillemets.

Les différents types



Il existe deux façons de stocker des données JSON : [les objets](#) et [les tableaux](#).

- Utiliser les tableaux :

Les valeurs sont placées entre crochets, et des virgules séparent chaque ligne.

Chaque valeur des tableaux JSON peut être d'un type différent.

```
{
  "firstName": "Tom",
  "lastName": "Jackson",
  "gender": "male",
  "hobby": [
    "football",
    "reading",
    "swimming"
  ]
}
```

Exemple

Un objet JSON

Voici un exemple simple d'utilisation de JSON, Voici ce que chaque paire indique :

- La première ligne `className` : `Class 2B` est une chaîne de caractères.
- La deuxième paire `year` : `2022` a une valeur numérique.
- La troisième paire `phoneNumber` : `null` représente un null – il n'y a pas de valeur.
- La quatrième paire `active` :`true` est une expression booléenne.
- La cinquième ligne `homeroomTeacher` : `{ firstName : Richard , lastName : Roe }` représente un objet littéral.
- Enfin, un tableau de membres.

Un petit exemple

```
{  
    "className": "Class 2B",  
    "year": 2022,  
    "phoneNumber": null,  
    "active": true,  
    "homeroomTeacher": {  
        "firstName": "Richard", "lastName": "Roe"  
    },  
    "members": [  
        { "firstName": "Jane", "lastName": "Doe" },  
        {"firstName": "Jinny", "lastName": "Roe"},  
        {"firstName": "Johnny", "lastName": "Roe"}  
    ]  
}
```



Les classes en Javascript

Classe en ES6

Classe d'objets

Avec ES6, JavaScript a introduit les classes, qui sont une syntaxe plus simple pour travailler avec les prototypes :

- Une classe en JavaScript est définie à l'aide du mot-clé `class`.
- Pour créer une instance de la classe `Person`, utilisez le mot-clé `new`
- Les classes en JavaScript supportent l'héritage via le mot-clé `extends`.

Création d'une classe

```
class Person {  
    constructor(firstName, lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    getFullName() {  
        return this.firstName + " " + this.lastName;  
    }  
}  
  
class Employee extends Person {  
    constructor(firstName, lastName, jobTitle) {  
        super(firstName, lastName);  
        this.jobTitle = jobTitle;  
    }  
}  
  
let jim = new Employee("Jim", "Brown", "Manager");  
console.log(jim.getFullName()); // "Jim Brown"  
console.log(jim.jobTitle); // "Manager"
```

Utilisation de _ et # dans les classes

Dans vos diverses recherches sur Javascript et l'objet, il se peut que vous trouviez du code avec _ ou # devant les attributs.

Ils sont chacun une signification différente que nous allons détailler



- # permet de désigner une variable comme **private**. Ce qui la rends inaccessible en dehors de la classe. Ceci correspond donc à l'encapsulation.
- Pour avoir accès à l'attribut, il faudra passer par un **getter et un setter**.



- _ est une convention plus ancienne et moins stricte. Elle peut être utilisée dans des contextes où tu souhaites indiquer que certains attributs ne devraient pas être directement accessibles ou modifiés (pas de getter et setter), mais où tu n'as pas besoin d'une encapsulation stricte.
- Cependant, avec l'introduction des champs privés, il est généralement préférable d'utiliser # pour une meilleure encapsulation.
- La variable peut être accessible en dehors de la classe mais par convention, cela indique une utilisation interne à la classe.

Classe en ES6

Static, constructeur, getter et setter

static

```
class MathUtils {  
    static add(a, b) {  
        return a + b;  
    }  
  
    console.log(MathUtils.add(2, 3)); // 5
```

Les méthodes statiques sont définies avec le mot-clé **static** et peuvent être appelées directement sur la classe sans créer d'instance.

constructeur

```
class Rectangle {  
    constructor(width, height) {  
        this._width = width;  
        this._height = height;  
    }  
  
    get area() {  
        return this._width * this._height;  
    }  
  
    set width(value) {  
        this._width = value;  
    }  
  
    set height(value) {  
        this._height = value;  
    }  
  
    let rect = new Rectangle(10, 20);  
    console.log(rect.area); // 200  
    rect.width = 15;  
    console.log(rect.area); // 300
```

getter

setter

Exporter une classe

Utilisation d'une classe dans un script

Supposons que vous avez une classe Person dans un fichier nommé Person.js. Il faut la déclarer en export

```
// Person.js
export class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  getFullName() {
    return this.firstName + " " + this.lastName;
  }
}
```

Importation

Vous pouvez ensuite importer cette classe dans un autre fichier, par exemple main.js :

```
// main.js
import { Person } from './Person.js';

let john = new Person("John", "Doe");
console.log(john.getFullName()); // "John Doe"
```

Dans ce cas, lors de la déclaration du script dans HTML, il faut le déclarer en module :

```
<script type="module" src="./script/script.js"></script>
```

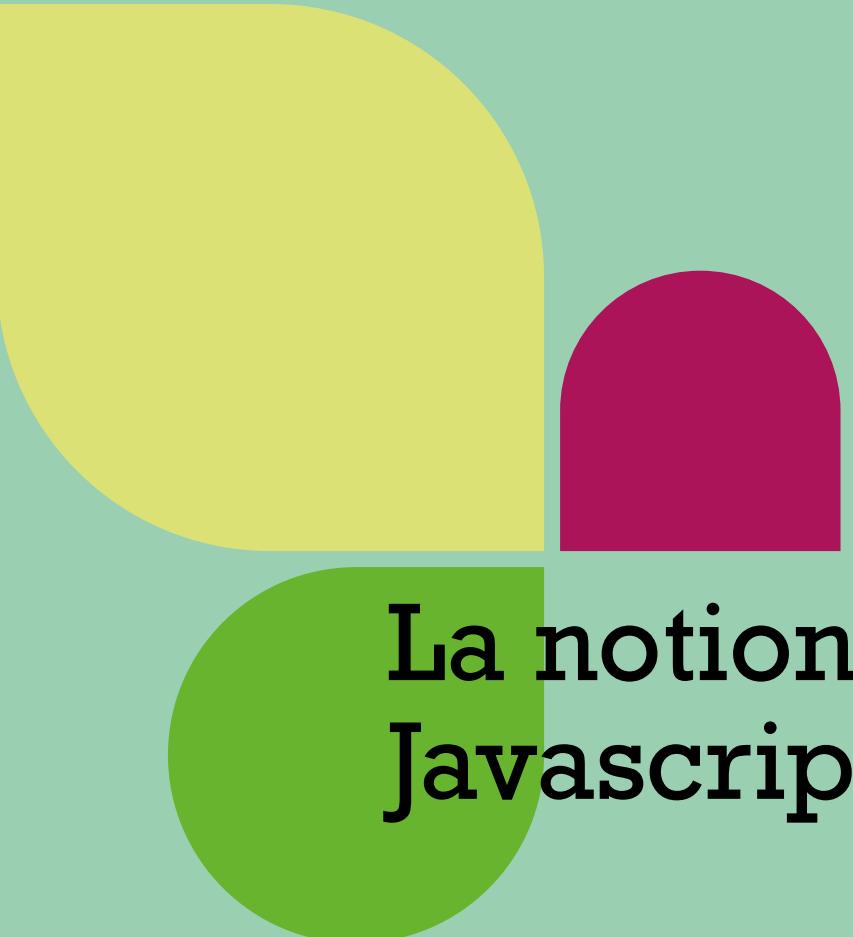
Utiliser des exports par défaut : Si vous souhaitez exporter une seule classe ou fonction par défaut, vous pouvez utiliser `export default` et ensuite importer cette classe sans accolades :

```
// Person.js
export default class Person { // main.js
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  getFullName() {
    return this.firstName + " " + this.lastName;
  }
}

import Person from './Person.js';

let john = new Person("John", "Doe");
console.log(john.getFullName()); // "John Doe"
```



La notion de prototype en Javascript

Les Prototypes

En JavaScript, tout est **objet** (sauf les types primitifs comme number, string, etc.).

Contrairement à d'autres langages (comme Java ou C++), JavaScript n'utilise pas de **classes** pour l'héritage, mais un mécanisme appelé **prototype**.

- C'est ce qu'on appelle l'héritage **prototypique** et diffère de l'héritage de classe (Java par exemple).

Qu'est-ce qu'un prototype ?

- Chaque objet en JavaScript a une propriété cachée appelée `__proto__` (ou accessible via `Object.getPrototypeOf()`).
- Cette propriété pointe vers un autre objet : son **prototype**.
- Le prototype est un objet qui contient des propriétés et méthodes **partagées** par tous les objets qui en héritent.

```
const animal = { mange: true };
const chat = { miaule: true };
chat.__proto__ = animal; // chat hérite de animal

console.log(chat.mange); // true (hérité)
console.log(chat.miaule); // true (propre)
```

La chaîne de prototypes

Si une propriété/méthode n'est pas trouvée dans un objet, JavaScript la cherche dans son prototype, puis dans le prototype du prototype, etc.

Cela forme une **chaîne de prototypes**.

- La chaîne se termine à `Object.prototype` (le prototype par défaut de tous les objets), dont le prototype est null.



```
console.log(chat.__proto__ === animal); // true
console.log(animal.__proto__ === Object.prototype); // true
console.log(Object.prototype.__proto__); // null
```

Constructeurs et prototypes

- Les fonctions constructeurs (utilisées avec `new`) ont une propriété `prototype`.
- Quand un objet est créé avec `new`, son `__proto__` pointe vers `constructeur.prototype`.



**Tester le prototype d'un
objet**

TESTER UN PROTOTYPE

instanceof

La méthode instanceof permet de vérifier si un objet est une instance d'un constructeur particulier. Cela fonctionne en vérifiant la chaîne de prototypes.

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

let john = new Person("John", "Doe");

console.log(john instanceof Person); // true
console.log(john instanceof Object); // true
```

isPrototypeOf

La méthode isPrototypeOf permet de vérifier si un objet est dans la chaîne de prototypes d'un autre objet.

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

let john = new Person("John", "Doe");

console.log(Person.prototype.isPrototypeOf(john)); // true
console.log(Object.prototype.isPrototypeOf(john)); // true
```

TESTER UN PROTOTYPE

Object.getPrototypeOf

La méthode `Object.getPrototypeOf` permet d'obtenir le prototype d'un objet. Vous pouvez ensuite comparer ce prototype avec un autre objet.

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

let john = new Person("John", "Doe");

console.log(Object.getPrototypeOf(john) === Person.prototype); // true
console.log(Object.getPrototypeOf(john) === Object.prototype); // false
```

`__proto__`

Bien que l'utilisation de `__proto__` soit **déconseillée** en faveur de `Object.getPrototypeOf`, elle est toujours disponible pour obtenir le prototype d'un objet.

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

let john = new Person("John", "Doe");

console.log(john.__proto__ === Person.prototype); // true
console.log(john.__proto__ === Object.prototype); // false
```

TESTER UN PROTOTYPE

Pour vérifier le type d'un objet, vous pouvez utiliser `Object.prototype.toString.call`.

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

let john = new Person("John", "Doe");

console.log(Object.prototype.toString.call(john)); // "[object Object]"
console.log(Object.prototype.toString.call(Person.prototype)); // "[object Object]"
```

```
function Person(firstName, lastName) {
  this.firstName = firstName;
  this.lastName = lastName;
}

let john = new Person("John", "Doe");

// Utiliser instanceof
console.log(john instanceof Person); // true
console.log(john instanceof Object); // true

// Utiliser isPrototypeOf
console.log(Person.prototype.isPrototypeOf(john)); // true
console.log(Object.prototype.isPrototypeOf(john)); // true

// Utiliser Object.getPrototypeOf
console.log(Object.getPrototypeOf(john) === Person.prototype); // true
console.log(Object.getPrototypeOf(john) === Object.prototype); // false

// Utiliser __proto__
console.log(john.__proto__ === Person.prototype); // true
console.log(john.__proto__ === Object.prototype); // false

// Utiliser Object.prototype.toString.call
console.log(Object.prototype.toString.call(john)); // "[object Object]"
console.log(Object.prototype.toString.call(Person.prototype)); // "[object Object]"
```



DOM

Manipulation de nos pages HTML

AFPA CENTRE DE POMPEY

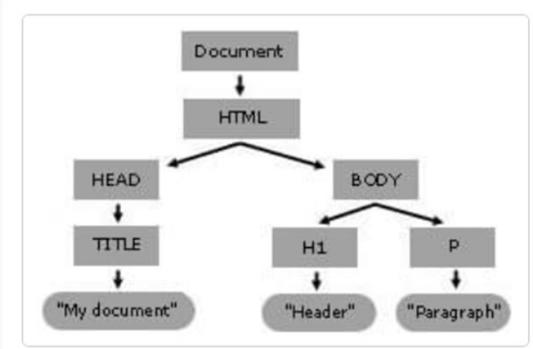


DÉFINITION

Le **DOM** ou **Document Object Model** est une interface de programmation (ou API) pour les documents HTML.

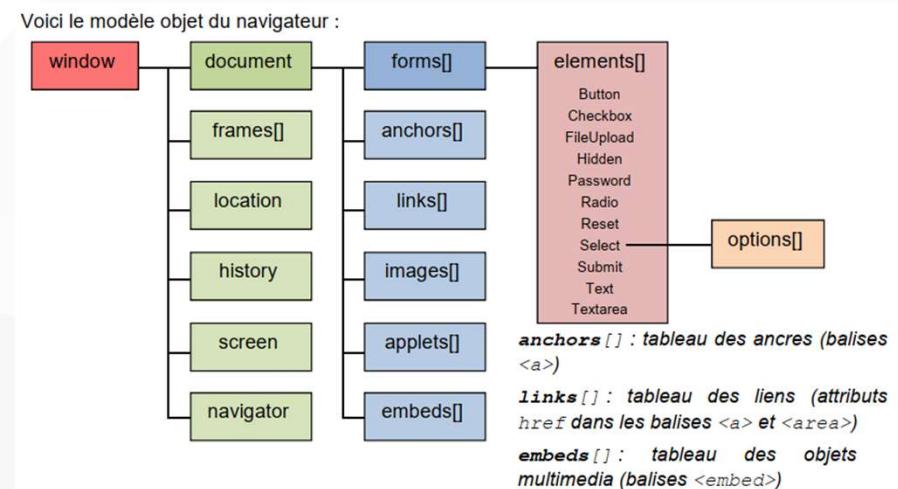
C'est donc un ensemble d'outils qui permettent de faire communiquer entre eux, dans le cas présent, les langages HTML et JavaScript.

- Standard établit par W3C



A la racine, l'objet **window** représente l'**instance du navigateur** :

- l'objet **location** qui symbolise la barre d'adresse,
- l'objet **history** qui représente l'historique des pages visitées par l'utilisateur
- l'objet **document** qui représente la page Web en cours, le contenu du <body> Html, lui-même référençant tous ses éléments.



window

Le 1^{er} objet

L'objet `window` qui représente une fenêtre contenant un document DOM

```
> console.log(window.location);
```

[VM162:1](#)

```
Location {ancestorOrigins: DOMStringList, href: 'http://tpform/t  
emplate.html?lastName=b&firstName=b...DateStart=&periodDateEnd=&n  
umberDay=&code01=01_01', origin: 'http://tpform', protocol: 'htt  
p:', host: 'tpform', ...}
```

```
< undefined
```

```
> console.log(window.location.search);
```

[VM284:1](#)

```
?  
lastName=b&firstName=b&study=CDA&option=optionOne&awayDate=2022-  
03-  
07&awayTimeStart=10&awayTimeEnd=11&periodDateStart=&periodDateEnd=  
&numberDay=&code01=01_01
```

```
< undefined
```

```
> console.log(window.location.search.substring(1));
```

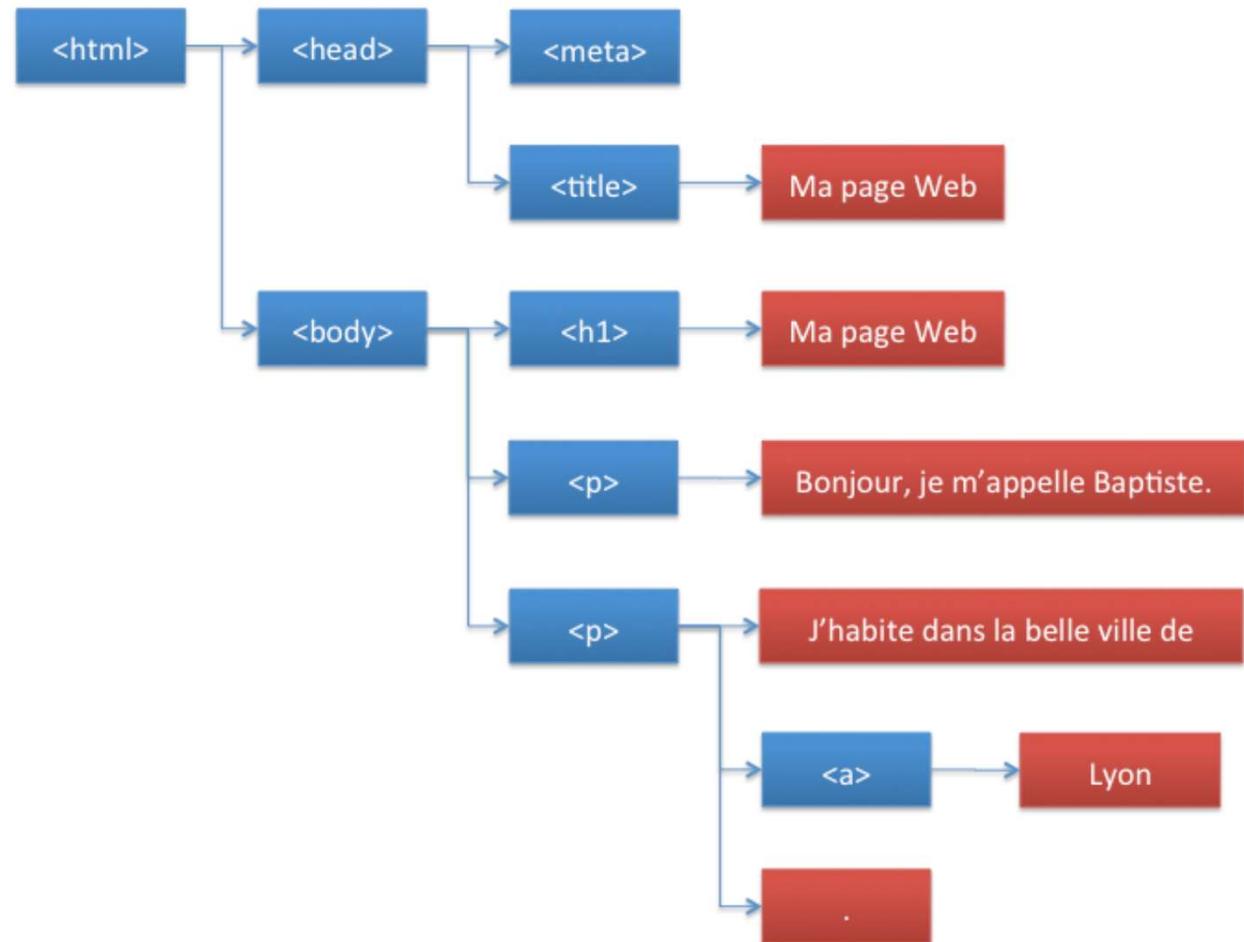
```
lastName=b&firstName=b&study=CDA&option=optionOne&awayDate VM367:1  
=2022-03-  
07&awayTimeStart=10&awayTimeEnd=11&periodDateStart=&periodDateEnd=  
&numberDay=&code01=01_01
```

Type de noeud

document étant l'élément <html>

Dans le DOM, 2 types de nœuds :

- Ceux en bleu sont des éléments HTML comme body, p, h1 etc...
 - Ces nœuds peuvent avoir des sous-nœuds, appelés fils ou enfants (children) (`nodeType = ELEMENT_NODE`)
- Ceux en rouge, sont au contenu textuel de la page.
 - Ces nœuds ne peuvent avoir de fils. (`nodeType = TEXT_NODE`)



PARCOURIR LES NŒUDS

```

<body>
    <h1>Les sept merveilles du monde</h1>
    <p>Connaissez-vous les merveilles du monde ?</p>
    <div id="contenu">
        <h2>Merveilles du monde antique</h2>
        <p>Cette liste nous vient de l'Antiquité.</p>
        <ul class="merveilles" id="antiques">
            <li class="existe">La pyramide de Khéops</li>
            <li>Les jardins suspendus de Babylone</li>
            <li>La statue de Zeus</li>
            <li>Le temple d'Artémis</li>
            <li>Le mausolée d'Halicarnasse</li>
            <li>Le Colosse de Rhodes</li>
            <li>Le phare d'Alexandrie</li>
        </ul>
        <h2>Nouvelles merveilles du monde</h2>
        <p>Cette liste a été établie en 2009 à la suite d'un vote par Internet.</p>
        <ul class="merveilles" id="nouvelles">
            <li class="existe">La Grande Muraille de Chine</li>
            <li class="existe">Pétra</li>
            <li class="existe">Le Christ du Corcovado</li>
            <li class="existe">Machu Picchu</li>
            <li class="existe">Chichén Itzá</li>
            <li class="existe">Le Colisée</li>
            <li class="existe">Le Taj Mahal</li>
        </ul>
        <h2>Références</h2>
        <ul>
            <li><a href="https://fr.wikipedia.org/wiki/Sept_merveilles_du_monde">Merveilles antiques</a></li>
            <li><a href="https://fr.wikipedia.org/wiki/Sept_nouvelles_merveilles_du_monde">Nouvelles merveilles</a></li>
        </ul>
    </div>
</body>

```

```

// etape 1
var h = document.head; // La variable h contient l'objet head du DOM
console.log(h);
// etape 2
var b = document.body; // La variable b contient l'objet body du DOM
console.log(b);
// etape 3
// type de noeud
if (document.body.nodeType === document.ELEMENT_NODE) {
    console.log("Body est un noeud élément");
} else {
    console.log("Body est un noeud textuel");
}
// etape 4
// Accéder aux enfants d'un noeud élément
// Accès au premier enfant du noeud body ???
console.log(document.body.childNodes[0]);
/*
les espaces entre les balises ainsi que les retours à la ligne dans le code
HTML sont considérés par le navigateur comme des nœuds textuels. Ici, le noeud
h1 n'est donc que le deuxième enfant du noeud body.
*/
console.log(document.body.childNodes[1]);
// parcours des noeuds enfants
// Affiche les noeuds enfant du noeud body
for (let i = 0; i < document.body.childNodes.length; i++) {
    console.log(document.body.childNodes[i]);
}
// Accéder au parent d'un noeud
var h1 = document.body.childNodes[1];
console.log(h1.parentNode); // Affiche le noeud body

console.log(document.parentNode); // Affiche null : document n'a aucun noeud
parent

```

```
let element = document.getElementById(id);

let elements = document.getElementsByClassName(names); // ou:
elements = rootElement.getElementsByClassName(names);

elements = document.getElementsByTagName(nom);

elements = document.getElementsByName(nom)

let el = document.querySelector(".maclasse");

let matches = myBox.querySelectorAll("p");
```

Accéder au document

Chaque élément du DOM est un objet Javascript avec ses propriétés et ses fonctions pour le manipuler.

A partir `document` qui représente la page entière, on peut retrouver chaque élément(s)

[getElementById\(\)](#)

Retrouver un élément précis par son id

[getElementsByClassName\(\)](#)

Retrouver tous les éléments par la classe

[getElementsByTagName\(\)](#)

Retrouver tous les éléments avec un nom de balise

[querySelector\(\)](#)

Retrouver le 1er élément correspondant au sélecteur CSS ou groupe de sélecteurs CSS

[querySelectorAll\(\)](#)

Retrouver tous les éléments correspondant au sélecteur CSS ou groupe de sélecteurs CSS

Recherche depuis un élément

Il n'y a pas qu'avec document que vous pouvez rechercher des éléments.

- Comme nous l'avons vu, chaque élément est un objet JavaScript avec ses propriétés et ses fonctions.

Et parmi ces dernières, il en existe pour parcourir les enfants et le parent de chaque élément !

```
<div id="parent">
  <div id="previous">Précédent</div>
  <div id="main">
    <p>Paragraphe 1</p>
    <p>Paragraphe 2</p>
  </div>
  <div id="next">Suivant</div>
</div>
```

```
const elt = document.getElementById('main');
```

elt.children = les éléments de type p enfant de #main
elt.parentElement = div qui à l'id parent
elt.nextElementSibling = next
elt.previousElementSibling = previous

element.children

Retourne une liste d'enfant de l'élément

element.parentElement

Retourne l'élément parent

element.nextElementSibling et element.previousElementSibling

Permet de naviguer vers l'élément suivant / précédent de même niveau

MODIFIER LE DOM

Deux propriétés principales :

- [innerHTML](#)
 - Récupère ou définit la contenu HTML d'un élément du DOM
- [textContent](#)
 - Récupère son contenu textuel sans le balisage HTML du DOM

- Les attributs et les classes :
 - Modifier des classes d'un élément :
 - Possible d'accéder à la liste des classes d'un élément avec la propriété [classList](#)
 - [add\(string\)](#) : ajoute une classe
 - [remove\(string\)](#) : supprime la classe
 - [contains\(string\)](#) : vérifie si la classe existe
 - [replace\(old, new\)](#) : remplace l'ancienne classe par la nouvelle
 - Changer les styles d'un élément :
 - Possible d'accéder au style avec la propriété [style](#)
 - [element.style.backgroundColor = '#000';](#)
 - Modifier les attributs d'un élément :
 - Définir ou remplacer les attributs avec la fonction [setAttribute](#), [getAttribute](#), [removeAttribute](#)

DIFFÉRENTES MÉTHODES

Modifier un attribut avec setAttribute

La méthode setAttribute permet d'ajouter ou de modifier un attribut HTML (comme class, id, data-*, etc.).

```
const element = document.getElementById("monElement");
// Ajouter/modifier un attribut
element.setAttribute("class", "ma-classe");
element.setAttribute("id", "monId");
element.setAttribute("data-info", "valeur");
```

Cas d'usage :

- Ajouter une classe, un ID, ou un attribut personnalisé (data-*).

Limites :

- Pour les styles, il est préférable d'utiliser style ou classList

Modifier le style directement avec style

Pour modifier le style CSS inline d'un élément, utilise la propriété style :

```
const element = document.getElementById("monElement");
element.style.backgroundColor = "red";
element.style.fontSize = "20px";
element.style.border = "1px solid black";
```

Cas d'usage :

- Appliquer des styles dynamiques et uniques à un élément.

Note :

- Les noms des propriétés CSS sont en camelCase
(backgroundColor au lieu de background-color).

DIFFÉRENTES MÉTHODES

Ajouter/supprimer une classe avec classList

Pour ajouter, supprimer ou basculer une classe CSS (sans écraser les autres classes) :

```
const element = document.getElementById("monElement");
// Ajouter une classe
element.classList.add("ma-classe");
// Supprimer une classe
element.classList.remove("ancienne-classe");
// Basculer une classe (ajoute si absente, supprime si présente)
element.classList.toggle("ma-classe");
// Vérifier si une classe existe
if (element.classList.contains("ma-classe")) {
  console.log("La classe est présente !");
}
```

Cas d'usage :

- Appliquer des styles définis dans une feuille CSS, de manière modulaire.

Modifier plusieurs styles avec cssText

Pour appliquer plusieurs styles en une fois via une chaîne de caractères :

```
const element = document.getElementById("monElement");
element.style.cssText = "background-color: red; font-size: 20px; border: 1px solid black;"
```

Cas d'usage :

- Réinitialiser ou appliquer plusieurs styles d'un coup.

Attention : Écrase tous les styles inline existants.

DIFFÉRENTES MÉTHODES

Modifier un attribut de style avec setAttribute (peu recommandé)

Techniquement, on peut utiliser setAttribute pour modifier le style, mais ce n'est pas recommandé car cela écrase tous les styles inline existants :

```
element.setAttribute("style", "background-color: red; font-size: 20px;");
```

Problème : Écrase tous les styles inline précédents.

Utiliser getAttribute et setAttribute pour les attributs standards

Pour lire ou modifier des attributs comme class, id, title, etc. :

```
const element = document.getElementById("monElement");
// Lire un attribut
const classeActuelle = element.getAttribute("class");
// Modifier un attribut
element.setAttribute("class", "nouvelle-classe");
```

- Cas d'usage :
 - Manipuler des attributs autres que style (comme class, id, src, etc.).

MODIFIER LE DOM

On peut également :

1. Créer de nouveaux éléments.
2. Ajouter des enfants.
3. Supprimer et remplacer des éléments.

- La fonction [createElement](#) permet de créer de nouveaux éléments.
- Plusieurs façons d'ajouter un élément :
 - [appendChild](#)
- Les fonctions [removeChild](#) et [replaceChild](#) permettent de supprimer ou remplacer un élément

```
const newEl = document.createElement("div");
let elt = document.getElementById("main");
elt.appendChild(newEl);

elt.removeChild(newEl); // Supprime l'élément newEl de l'élément elt
elt.replaceChild(document.createElement("article"), newEl); // Remplace l'élément newEl par un nouvel élément de type article
```

```
const newEl = document.createElement("div");
let elt = document.getElementById("main");

elt.appendChild(newEl);
```

```
const newEl = document.createElement("div");
```

DIFFÉRENTES MÉTHODES

Utilisation de innerHTML (Approche basique)

```
const container = document.getElementById("container");
container.innerHTML += "<div class='nouvel-element'>Contenu</div>";
```

Avantages :

- Simple et rapide à écrire.

Inconvénients :

- Risque de sécurité (injection XSS si le contenu n'est pas contrôlé).
- Performance médiocre : À chaque modification, le navigateur doit reparsier tout le contenu HTML de l'élément.
- Écrase les écouteurs d'événements existants sur les éléments enfants.

Utilisation de document.write() (À éviter)

```
document.write("<div class='nouvel-element'>Contenu</div>");
```

Avantages :

- Très simple.

Inconvénients :

- À éviter absolument : Écrase tout le document si appelé après le chargement de la page.
- Non recommandé pour les applications modernes.

DIFFÉRENTES MÉTHODES

Utilisation de `document.createElement()` et `appendChild()` (Approche standard)

```
const container = document.getElementById("container");
const nouvelElement = document.createElement("div");
nouvelElement.className = "nouvel-element";
nouvelElement.textContent = "Contenu";
container.appendChild(nouvelElement);
```

Avantages :

- Sécurisé (pas de risque XSS si le contenu est contrôlé).
- Préserve les écouteurs d'événements et la structure DOM.

Inconvénients :

- Un peu verbeux pour des structures complexes.

Utilisation de `insertAdjacentHTML()` (Approche flexible)

```
const container = document.getElementById("container");
container.insertAdjacentHTML("beforeend", "<div class='nouvel-element'>Contenu</div>");
```

Avantages :

- Plus performant que `innerHTML` car ne reparsé pas tout le contenu.
- Permet d'insérer du HTML à des positions précises (`beforebegin`, `afterbegin`, `beforeend`, `afterend`).

- Inconvénients :

DIFFÉRENTES MÉTHODES

Utilisation de DocumentFragment (Approche optimisée pour plusieurs éléments)

```
const fragment = document.createDocumentFragment();
const nouvelElement1 = document.createElement("div");
nouvelElement1.className = "nouvel-element";
nouvelElement1.textContent = "Contenu 1";
fragment.appendChild(nouvelElement1);

const nouvelElement2 = document.createElement("div");
nouvelElement2.className = "nouvel-element";
nouvelElement2.textContent = "Contenu 2";
fragment.appendChild(nouvelElement2);

const container = document.getElementById("container");
container.appendChild(fragment);
```

Avantages :

- Optimisé pour l'ajout de plusieurs éléments : Un seul reflow/repaint du DOM.
- Pas de risque XSS si le contenu est contrôlé.

Inconvénients :

- Un peu plus complexe à mettre en place.

Utilisation de cloneNode() (Pour dupliquer des éléments existants)

```
const elementExistant = document.querySelector(".element-existant");
const clone = elementExistant.cloneNode(true); // Clone avec tous les enfants
document.getElementById("container").appendChild(clone);
```

Avantages :

- Rapide pour dupliquer des éléments existants.
- Préserve les écouteurs d'événements si cloneNode(true) est utilisé (mais pas les écouteurs ajoutés via addEventListener).

Inconvénients :

- Ne convient pas pour créer des éléments entièrement nouveaux.

DIFFÉRENTES MÉTHODES

Utilisation de templates HTML et content.cloneNode() (Approche moderne)

```
//HTML
<template id="mon-template">
  <div class="nouvel-element">Contenu</div>
</template>

// JS
const template = document.getElementById("mon-template");
const clone = template.content.cloneNode(true);
document.getElementById("container").appendChild(clone);
```

Avantages :

- Très performant : Le template est parsé une seule fois.
- Sécurisé : Le contenu du template est inerte jusqu'à son utilisation.
- Modularité : Idéal pour des structures HTML complexes ou répétitives.

Inconvénients :

- Nécessite de définir un template dans le HTML.

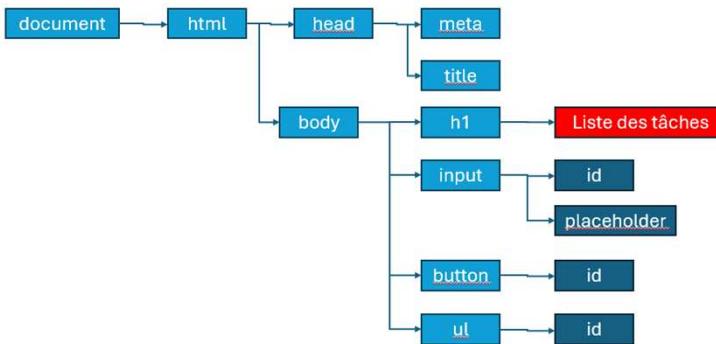
Recommandation

- Pour des ajouts simples, utilise createElement() + appendChild().
- Pour des ajouts multiples, utilise DocumentFragment.
- Pour des structures complexes, utilise des templates HTML.
- Pour des applications dynamiques, envisage une bibliothèque comme React ou Vue.

Exemple

Ci contre, un fichier HTML pour lister des tâches que l'utilisateur saisie. On souhaite ajouter une tâche dynamiquement

Sa décomposition en arbre donnera :



```
dom.html > ...
1  <!DOCTYPE html>
2  <html lang="fr">
3  <head>
4      <meta charset="UTF-8">
5      <title>Liste de Tâches</title>
6      <!-- CSS -->
7      <link rel="stylesheet" href="./CSS/swapiometeo.css">
8  </head>
9  <body>
0      <h1>Liste de Tâches</h1>
1      <input type="text" id="nouvelleTache" placeholder="Ajouter une tâche">
2      <button id="ajouter">Ajouter</button>
3      <ul id="listeTaches"></ul>
4
5      <script type="text/javascript" src="./scriptJS/dom.js"></script>
6  </body>
7  </html>
8
9 |
```

Exemple

Le script associé au fichier précédemment va construire le DOM en fonction de la saisie utilisateur en créant les nœuds **li** et les ajouter au nœud **ul**

Liste de Tâches

- apprendre les bases de JS
- faire les exercices
- ...

```
// script JS de manipulation du DOM
function ajouterTache() {

    // recuperation de la valeur de l'element dont l'id est nouvelleTache dans le DOM
    // utilisation de trim pour enlever les espaces vides
    let tache = document.getElementById("nouvelleTache").value.trim();
    // test si aucune saisie
    if (tache !== "") {
        // recuperation de l'element dont l'id est listeTaches dans le DOM
        let liste = document.getElementById("listeTaches");
        // creation d'un element li
        let element = document.createElement("li");
        // ajout du texte du li
        element.textContent = tache;

        // ajout d'un evenement sur l'element li
        element.onclick = function () {
            // suppression de l'element
            liste.removeChild(element);
        };

        // ajout de l'element li au parent
        liste.appendChild(element);
        // reset du champs input
        document.getElementById("nouvelleTache").value = "";
    } else {
        // affichage d'une alerte de saisie
        alert("Merci de saisir une tâche !!");
    }
}

// au chargement de la page HTML
document.addEventListener('DOMContentLoaded', function () {
    // ajout d'un evenement sur le bouton sur click
    document.getElementById("ajouter").addEventListener('click', ajouterTache);
});
```



LES ÉVÈNEMENTS

TYPE D'ÉVÉNEMENTS

Les événements que les éléments du DOM peuvent déclencher sont très nombreux.

Le tableau ci-contre présente les principales catégories d'événements.

Quel que soit le type d'événement, son déclenchement s'accompagne de la création d'un objet [Event](#)

```
// Ajout d'un gestionnaire qui affiche le type et la cible de l'événement
document.getElementById("bouton").addEventListener("click", function (e) {
    console.log("Evènement : " + e.type +
        ", texte de la cible : " + e.target.textContent);
});
```



keyup - keydown - keypress



clic - mousemove

Catégorie	Exemples
Événements clavier	Appui ou relâchement d'une touche du clavier
Événements souris	Clic avec les différents boutons, appui ou relâchement d'un bouton de la souris, survol d'une zone avec la souris
Événements fenêtre	Chargement ou fermeture de la page, redimensionnement, défilement (<i>scrolling</i>)
Événements formulaire	Changement de cible de saisie (focus), envoi d'un formulaire



submit - reset



load - beforeunload

LES ÉVÈNEMENTS

La souris

Avec `mousemove`, on peut détecter le mouvement de la souris.

Cet évènement fournit un objet `MouseEvent` qui permet de récupérer

- `clientX / clientY` : position de la souris
- `offsetX / offsetY` : position par rapport à l'élément
- `pageX / pageY` position par rapport au document
- `screenX / screenY` position par rapport à la fenêtre
- `movementX / movementY` position par rapport à la position lors du dernier évènement `mousemove`

Lire un champ texte

L'évènement `change` pour les éléments `input`, `select`, `textarea`, `checkbox`, `radio` est déclenché lorsque le champ perd le focus

L'évènement `input` permet de connaître les modifications d'un élément en cours par l'utilisateur.

```
<label>Choose an ice cream flavor:  
  <select id="ice-cream" name="ice-cream">  
    <option value="">Select One ...</option>  
    <option value="chocolate">Chocolate</option>  
    <option value="strawberry">Strawberry</option>  
    <option value="vanilla">Vanilla</option>  
  </select>  
  document.querySelector('select[name="ice-cream"]').onchange=changeEventHandler;  
,false);  
  
function changeEventHandler(event) {  
  // You can use "this" to refer to the selected element.  
  if(!event.target.value) alert('Please Select One');  
  else alert('You like ' + event.target.value + ' ice cream.');//
```

DIRECTEMENT SUR LA BALISE

```
<!DOCTYPE html>
<html lang="fr">

  <head>
    <!-- ENCODAGE -->
    <meta charset="UTF-8">
    <!-- description du site -->
    <meta name="description" content="Cours HTML">
    <!-- prise en charge du contexte mobile -->
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <!-- titre de la page -->
    <title>Document</title>
    <link rel="stylesheet" type="text/css" href="ressources/style/style.css">

  <body>
    <h1>TEST JS</h1>
    <input type="button" value="Change ce document." onclick="change()">
    <h2>H2</h2>
    <p>Paragraphe</p>

    <script StyleSheet="text/javascript" src="/ressources/script/script.js"></script>
  </body>

</html>
```

On peut appliquer l'événement directement dans le html

```
function change() {
  // document.getElementsByTagName ("H2") renvoie un NodeList du <h2>
  // éléments dans le document, et le premier est le nombre 0:
  let header = document.getElementsByTagName("H2").item(0);
  // le firstChild de l'en-tête est un noeud texte::
  header.firstChild.data = "A dynamic document";
  // maintenant l'en-tête est "Un document dynamique".

  let para = document.getElementsByTagName("P").item(0);
  para.firstChild.data = "This is the first paragraph.";

  // crée un nouveau noeud texte pour le second paragraphe
  let newText = document.createTextNode("This is the second paragraph.");
  // crée un nouvel Element pour le second paragraphe
  let newElement = document.createElement("P");
  // pose le texte dans le paragraphe
  newElement.appendChild(newText);
  // et pose le paragraphe à la fin du document en l'ajoutant
  // au BODY (qui est le parent de para)
  para.parentNode.appendChild(newElement);
}
```

Écouter les évènements

On va pouvoir ajouter, retirer des événements à notre document ou à nos éléments avec Javascript.

Écouter ?

Réaction à une action émise par l'utilisateur comme le clic, la saisie etc...

Évènement ?

Un événement en JavaScript est représenté par un nom (click , mousemove ...) et une fonction que l'on nomme un callback

Propagation

Par défaut, un événement est propagé, transmis à l'élément parent jusqu'à la racine du document tant qu'on ne l'a pas traité.

Callback

La fonction de callback est appelée à chaque fois que l'action est désirée

AddEventListener removeEventListener

- permet d'ajouter tous types d'événements.
- Permet de retirer l'événement.

LA PROPAGATION DES ÉVÉNEMENTS

Le DOM représente une page web sous la forme d'une hiérarchie de nœuds.

- Les événements déclenchés sur un nœud enfant vont se déclencher ensuite sur son nœud parent, puis sur le parent de celui-ci, et ce jusqu'à la racine du DOM (la variable document).

C'est ce qu'on appelle la propagation des événements.

`PreventDefault()` est une option permettant de supprimer le comportement par défaut de l'événement

`stopPropagation()` permet d'empêcher la propagation de l'événement au parent

```
<body>
    <button id="bouton">Cliquez-moi !</button>

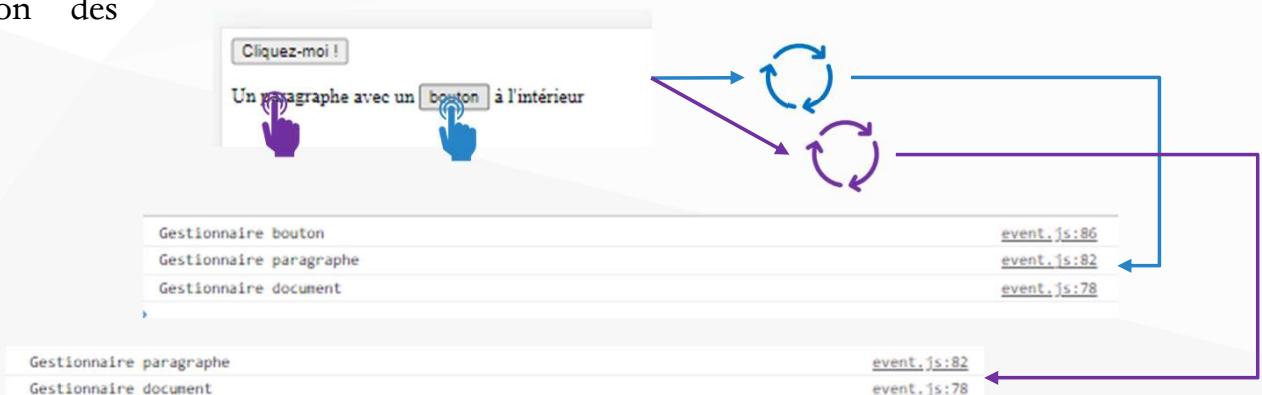
    <p id="para">Un paragraphe avec un
        <button id="propa">bouton</button> à l'intérieur
    </p>

    <script src="ressources/event.js"></script>
</body>
```

```
// Gestion du clic sur le document
document.addEventListener("click", function () {
    console.log("Gestionnaire document");
});

// Gestion du clic sur le paragraphe
document.getElementById("para").addEventListener("click", function () {
    console.log("Gestionnaire paragraphe");
});

// Gestion du clic sur le bouton
document.getElementById("propa").addEventListener("click", function (e) {
    console.log("Gestionnaire bouton");
});
```

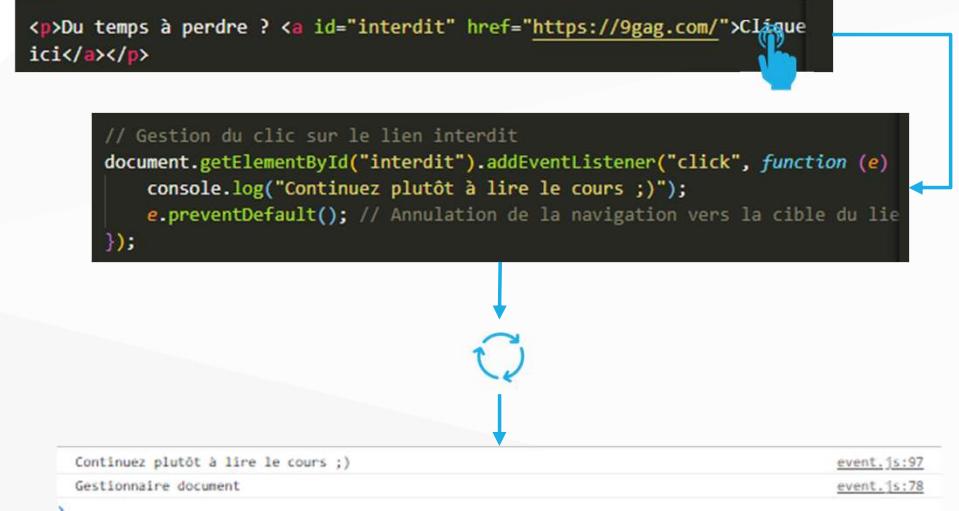


LA PROPAGATION DES ÉVÉNEMENTS

Stopper la propagation

```
// Gestion du clic sur le bouton
document.getElementById("propa").addEventListener("click", function (e) {
    console.log("Gestionnaire bouton");
    e.stopPropagation(); // Arrêt de la propagation de l'événement
});
```

Modifier le comportement par défaut





Les Formulaires

Manipulation des formulaires

AFPA CENTRE DE POMPEY



Les formulaires

Zone de texte - input et textarea

required

oblige la saisie

value

valeur de la zone de texte

gestion du
focus

focus : saisie en cours dans la zone de texte
blur : changement de cible provoque un blur sur l'ancienne zone de texte qui avait le focus

```
// Affichage d'un message contextuel pour la saisie du pseudo
pseudoElt.addEventListener("focus", function () {
    document.getElementById("aidePseudo").textContent = "Entrez votre pseudo";
});
// Suppression du message contextuel pour la saisie du pseudo
pseudoElt.addEventListener("blur", function (e) {
    document.getElementById("aidePseudo").textContent = "";
});
```

Les formulaires

Les cases à cocher et boutons radios

autocomplete="current-password"

indique : « ce champ contient le mot de passe existant d'un utilisateur déjà inscrit ».

Le navigateur peut proposer d'autocompléter si l'utilisateur a enregistré un mot de passe pour ce site.

autocomplete="new-password"

indique : « ce champ sert à créer un nouveau mot de passe ».

Le navigateur ne proposera pas de remplir un ancien mot de passe, et peut proposer d'en générer un nouveau.

change

indique lorsque l'utilisateur modifie son choix.

Case à cocher

dispose d'une propriété `checked`

Radio

Les attributs `name` et `value`

```
// Affichage de la demande de confirmation d'inscription
document.getElementById("confirmation").addEventListener("change", function (e)
{
    console.log("Demande de confirmation : " + e.target.checked);
});
```

Les formulaires

Liste déroulante et Formulaire

select + option

`change` est déclenché sur les modifications apportées à la liste.

Comme pour les boutons radio, la propriété `e.target.value` de l'événement `change` contient la valeur de l'attribut `value` de la balise `option` associé au nouveau choix, et non pas le texte affiché dans la liste déroulante.

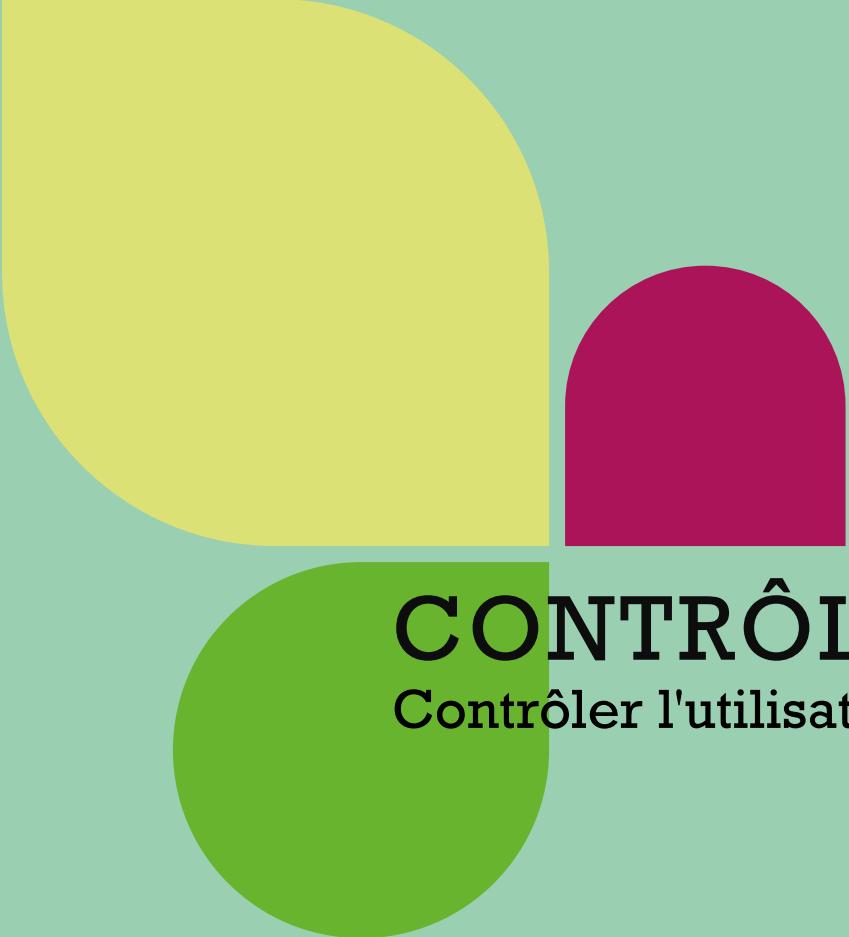
Formulaire

La balise `form` est l'élément à cibler pour accéder au contenu du formulaire par l'attribut `elements` rassemblant les champs de saisie

soumission

•`input` de type `submit` et `input` de type `reset`

C'est à partir de cette soumission que nous allons tester et contrôler la saisie de l'utilisateur et en cas d'erreur utiliser `preventDefault`



Contrôle de la saisie

Contrôler l'utilisateur

Sur les éléments du formulaire

REGEX

Vous avez aussi la possibilité de mettre en place des patterns sur la balise input afin de mettre en place un contrôle sur la saisie à la validation.

L'attribut pattern peut être utilisé pour les champs de type text, tel, email, url, password, search.

Les REGEX vont nous aider à mettre en place des contrôles sur la saisie.

Mise en forme

```
<p>
  <label for="mdp">Mot de passe</label> :
  <input type="password" name="mdp" id="mdp"
  pattern="(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}"
  title="doit contenir au moins 1 chiffre, 1 majuscule, 1 minuscule
  et au moins 8 caractères" required>
  <span id="aideMdp"></span>
</p>
```

Sur les différentes phases du formulaire

Contrôle des champs

Le contrôle de validité peut se faire de plusieurs manières, éventuellement combinables :

- Soit au fur et à mesure de la saisie d'une donnée
 - `input`
- Soit à la fin de la saisie d'une donnée
 - `blur`
- Soit au moment où l'utilisateur soumet le formulaire.
 - `submit`

Exemple

```
// Vérification de la longueur du mot de passe saisi
document.getElementById("mdp").addEventListener("input", function (e) {
    var mdp = e.target.value; // Valeur saisie dans le champ mdp
    var longueurMdp = "faible";
    var couleurMsg = "red"; // Longueur faible => couleur rouge
    if (mdp.length >= 8) {
        longueurMdp = "suffisante";
        couleurMsg = "green"; // Longueur suffisante => couleur verte
    } else if (mdp.length >= 4) {
        longueurMdp = "moyenne";
        couleurMsg = "orange"; // Longueur moyenne => couleur orange
    }
    var aideMdpElt = document.getElementById("aideMdp");
    aideMdpElt.textContent = "Longueur : " + longueurMdp; // Texte de l'aide
    aideMdpElt.style.color = couleurMsg; // Couleur du texte de l'aide
});

// Contrôle du courriel en fin de saisie
document.getElementById("courriel").addEventListener("blur", function (e) {
    var valideCourriel = "";
    if (e.target.value.indexOf("@") === -1) {
        // Le courriel saisi ne contient pas le caractère @
        valideCourriel = "Adresse invalide";
    }
    document.getElementById("aideCourriel").textContent = valideCourriel;
});
```

Sur les différentes phases du formulaire

Suite

Sur la sortie du champs de saisie,
ma fonction contrôle la saisie.

Contrôle du champs Email par exemple

```
// méthode sans REGEX
// Contrôle du courriel en fin de saisie
document.getElementById("courriel").addEventListener("blur", function (e) {
    var valideCourriel = "";
    if (e.target.value.indexOf("@") === -1) {
        // Le courriel saisi ne contient pas le caractère @
        valideCourriel = "Adresse invalide";
    }
    document.getElementById("aideCourriel").textContent = valideCourriel;
});

// méthode avec REGEX
// Contrôle du courriel en fin de saisie
document.getElementById("courriel").addEventListener("blur", function (e) {
    // Correspond à une chaîne de la forme xxx@yyy.zzz
    var regexCourriel = /.+@.+\.+$/;
    var valideCourriel = "";
    if (!regexCourriel.test(e.target.value)) {
        valideCourriel = "Adresse invalide";
    }
    document.getElementById("aideCourriel").textContent = valideCourriel;
});
```



Requêtes HTTP Communiquer avec un serveur

QUELQUES NOTIONS

Quelques notions théoriques qu'il faut connaître avant de se lancer dans la communication avec un serveur en JavaScript.

Service Web : programme s'exécutant sur un serveur accessible depuis internet et fournissant un service.

- Pour ce faire, il met à disposition une API.

API : Application Programming Interface - interface de communication avec les services Web, au travers de requêtes.

Requêtes : données qui respectent le protocole de communication et qui sont envoyées au serveur.

Protocoles : SMTP (envoi de mails), IMAP (réception de mails), HTTP et HTTPS, FTP, WebDAV, etc...

Requêtes HTTP asynchrones : **AJAX** (Asynchronous JavaScript and XML) est la technologie utilisée pour gérer ce type de requêtes.

- Cela permet de ne pas bloquer le navigateur pendant l'attente d'une réponse du serveur.

JSON : le format de données standard actuel pour l'échange des données sur le WEB.

LE PROTOCOLE HTTP(S)

HTTP : HyperText Transfert Protocol

- Communiquer avec un site internet, chargement des pages HTML, des styles CSS, etc...
- Envoie et récupération d'informations avec les formulaires.

- Méthodes :
 - **GET** : permet de récupérer des ressources
 - **POST** : permet de créer ou modifier une ressource
 - **PUT** : permet de modifier une ressource
 - **DELETE** : permet de supprimer une ressource
 - **HEAD** : demande des informations sur la ressource sans obtenir la ressource.
 - **TRACE, OPTION, CONNECT...**
- **URL** : l'adresse du service web à atteindre
- **Données** : les données qu'on envoie mais aussi qu'on reçoit
- **Code HTTP** : code numérique qui indique comment s'est déroulée la requête
 - 200 : tout s'est bien passé
 - 404 : la ressource n'existe pas
 - 500 : une erreur avec le service web
 - ... https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP

GET vs POST

GET

Cette méthode de requête existe depuis le début du Web. Elle est utilisée pour demander une ressource, par exemple un fichier HTML, au serveur Web.

- Paramètres URL

La requête GET peut recevoir des informations supplémentaires que le serveur Web doit traiter.

Ces paramètres d'URL sont simplement ajoutés à l'URL. La syntaxe est très simple :

- La chaîne de requête est introduite par un ? Et chaque paramètre est nommé, et composé donc d'un nom et d'une valeur : Nom=Valeur
- Si plusieurs paramètres doivent être inclus, ils sont reliés par un &
- un encodage des caractères spéciaux est effectué (exemple : @).

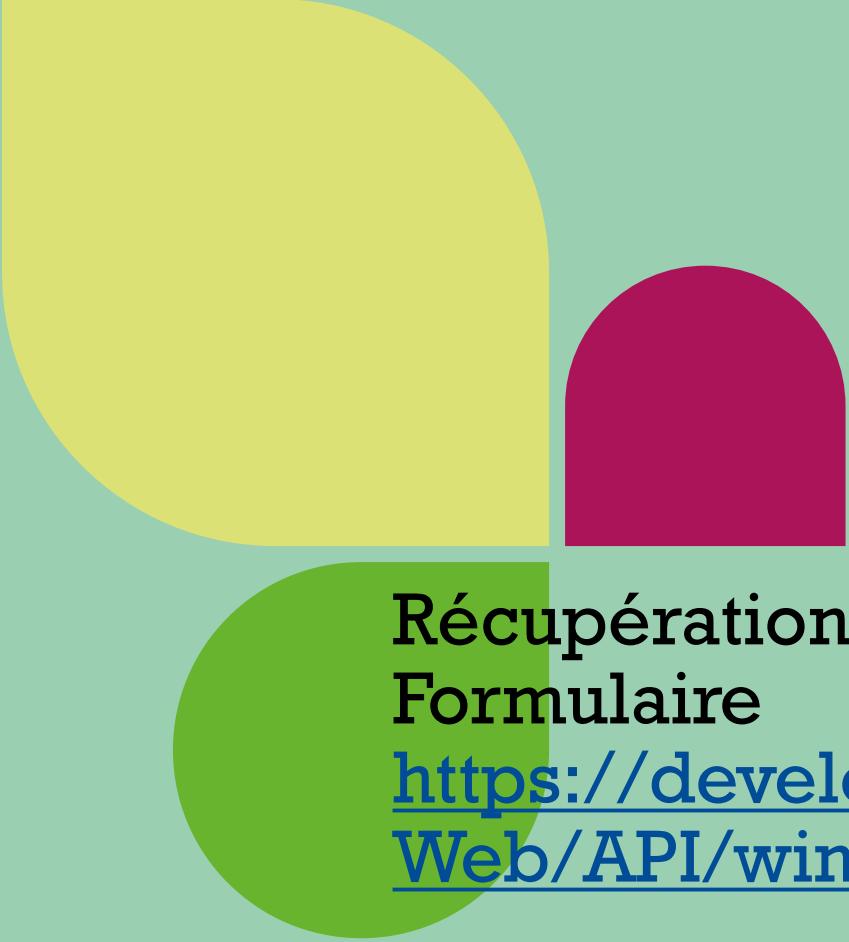
POST

Si vous souhaitez envoyer de grandes quantités de données, par exemple des images, ou des données confidentielles de formulaires au serveur, la méthode GET n'est pas idéale car toutes les données que vous envoyez sont écrites ouvertement dans la barre d'adresse du navigateur.

Dans ces cas, la méthode POST est la plus adaptée.

Cette méthode n'écrit pas les paramètres de l'URL, mais les ajoute à l'en-tête HTTP.

- Les requêtes POST sont principalement utilisées pour les formulaires en ligne.



Récupération des données du Formulaire

[https://developer.mozilla.org/fr/docs/
Web/API/window](https://developer.mozilla.org/fr/docs/Web/API/window)

Exemple

Depuis un formulaire

Soit le formulaire suivant :

Formulaire de Saisie

Nom :

Prénom :

Email :

Date d'anniversaire : jj/mm/aaaa

Adresse :

```
<form id="userForm" action="compte.html" method="get">
  <div class="form-group">
    <label for="nom">Nom :</label>
    <input type="text" id="nom" name="nom" required pattern="[A-Za-zÀ-Öö-ÿ\s]+"/>
    <div class="error" id="nomError"></div>
  </div>
  <div class="form-group">
    <label for="prenom">Prénom :</label>
    <input type="text" id="prenom" name="prenom" required pattern="[A-Za-zÀ-Öö-ÿ\s]+"/>
    <div class="error" id="prenomError"></div>
  </div>
  <div class="form-group">
    <label for="email">Email :</label>
    <input type="email" id="email" name="email" required>
    <div class="error" id="emailError"></div>
  </div>
  <div class="form-group">
    <label for="dateAnniversaire">Date d'anniversaire :</label>
    <input type="date" id="dateAnniversaire" name="dateAnniversaire" required>
    <div class="error" id="dateAnniversaireError"></div>
  </div>
  <div class="form-group">
    <label for="adresse">Adresse :</label>
    <input type="text" id="adresse" name="adresse" required>
    <div class="error" id="adresseError"></div>
  </div>
  <button type="submit">Soumettre</button>
</form>
```

window

Notre fenêtre en cours...

Lors de l'envoi du formulaire par la méthode GET, on envoie au travers de la requête HTTP **de manière visible**, les informations saisies dans notre formulaire.

Avec `window.location`, on obtient l'ensemble des propriétés disponibles.

- `window` représente notre page en cours
- `location` contient toutes les informations de la page courante

```
> window.location
<  Location {ancestorOrigins: DOMStringList,
  href: 'http://127.0.0.1:5500/HTTP_et_
  API/compte.html?nom=...versaire=2025-03-10
  &adresse=centre+afpa+de+Pompey', origin:
  'http://127.0.0.1:5500', protocol: 'htt
  p:', host: '127.0.0.1:5500', ...} i
  ▶ ancestorOrigins: DOMStringList {length:
  ▶ assign: f assign()
  hash: ""
  host: "127.0.0.1:5500"
  hostname: "127.0.0.1"
  href: "http://127.0.0.1:5500/HTTP_et_AF
  origin: "http://127.0.0.1:5500"
  pathname: "/HTTP_et_API/compte.html"
  port: "5500"
  protocol: "http:"
  ▶ reload: f reload()
  ▶ replace: f replace()
  search: "?nom=boebion&prenom=jerome&ema
  ▶ toString: f toString()
  ▶ valueOf: f valueOf()
  Symbol(Symbol.toPrimitive): undefined
  ▶ [[Prototype]]: Location
```

window.location. search

L'attribut search

L'attribut `search` contient l'ensemble des paramètres envoyés au serveur sous la forme

- ?param=valeur¶m=valeur...

Ensuite, on peut aisément récupérer cette donnée dans une variable et en extraire les informations dans un tableau.

```
> let param = window.location.search  
< undefined  
> console.info(param);  
?  
VM469:1  
nom=boebion&prenom=jerome&email=jero%40fr  
ee.fr&dateAnniversaire=2025-03-  
10&adresse=centre+afpa+de+Pompey
```

Récupération des données d'une URL

Côté Javascript

Maintenant que nous connaissons les éléments à cibler pour récupérer les données du formulaire, il nous reste à écrire un script Javascript.

Voici un exemple de récupération des données **de manière basique** au travers d'une URL.

- Les tests dans ce code permettent de remplacer les codes ascii que le formulaire utilise lors de l'envoi.
 - *Dans cet exemple, pour l'email par exemple il remplace : par le code ascii correspondant %40 et les espaces par +*

```
function getParamsURL() {  
  
    // objet contenant les parametres  
    let varParams = {};  
    // recuperation du l'url en retirant le ?  
    let search = window.location.search.substring(1);  
    // controle  
    console.info(search);  
  
    // decompostion des parametre en retirant le &  
    let varSearch = search.split('&');  
    // controle  
    console.info(varSearch);  
  
    // parcours des elements et contruction de mon objet  
    for (let i=0; i < varSearch.length; i++) {  
        // suppression du =  
        let parameter = varSearch[i].split('=');  
        // mise en forme du caractere ASCII @  
        if (parameter[0] === "email") {  
            parameter[1] = parameter[1].replace('%40', '@');  
        }  
        // mise en forme du caractere ASCII +  
        if (parameter[0] === "adresse") {  
            parameter[1] = parameter[1].replaceAll('+', ' ');  
        }  
        // construction de la reponse  
        varParams[parameter[0]] = parameter[1].toUpperCase();  
    }  
    // controle  
    console.info(varParams);  
  
    return varParams;  
}
```

Affichage

Mise en forme

Ensuite dans la page de récupération, là où le formulaire nous redirige, nous n'avons plus qu'à afficher les données saisies.

```
document.addEventListener("DOMContentLoaded", function () {  
    // appel de la fonction  
    let params = getParamsURL();  
  
    console.dir(params);  
  
    // Mise en forme dans la page HTML  
    document.getElementById("nom").textContent = params.nom;  
    document.getElementById("prenom").textContent = params.prenom;  
    document.getElementById("email").textContent = params.email;  
    document.getElementById("dateAnniversaire").textContent = params.dateAnniversaire;  
    document.getElementById("adresse").textContent = params.adresse;  
});
```

FORMDATA

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

FORMDATA

L'objet `FormData` a été standardisé et facilite grandement l'envoi vers un serveur.

- Il peut être utilisé indépendamment d'un formulaire, en lui ajoutant une à une les données à transmettre grâce à sa méthode `append`.
- Cette méthode prend en paramètres le nom et la valeur de la donnée ajoutée (clé/valeur).

Ensuite, on peut retrouver les données contenantes dans notre objet `FormData` avec la méthode `entries()` qui retourne un `itérator` permettant d'accéder à l'ensemble des données.

Cet objet est souvent utilisé avec la méthode POST

```
document.getElementById('monFormulaire').addEventListener('submit', function(event) {  
    event.preventDefault(); // Empêche l'envoi du formulaire  
  
    // Crée un nouvel objet FormData à partir du formulaire  
    const formData = new FormData(event.target);  
  
    // Affiche les données du formulaire dans la console  
    for (let [name, value] of formData.entries()) {  
        console.log(`${name}: ${value}`);  
    }  
});
```

<https://fr.javascript.info/formdata>

MÉTHODES DISPONIBLES

Diverses méthodes sont disponibles avec formdata

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

- `append()` : Ajoute une nouvelle valeur à une clé existante dans un objet FormData, ou ajoute la clé si elle n'existe pas encore.
- `delete()` : Supprime une paire clé/valeur d'un objet FormData.
- `entries()` : Renvoie un itérateur permettant de passer en revue toutes les paires clé/valeur contenues dans cet objet.
- `get()` : Renvoie la première valeur associée à une clé donnée à partir d'un objet FormData.
- `getAll()` : Renvoie un tableau de toutes les valeurs associées à une clé donnée à partir d'un objet FormData.
- `has()` : Renvoie un booléen indiquant si un objet FormData contient une certaine clé.
- `key()` : Renvoie un itérateur permettant de parcourir toutes les clés des paires clé/valeur contenues dans cet objet.
- `set()` : Renvoie un itérateur permettant de parcourir toutes les valeurs contenues dans cet objet.
- `values()` : Renvoie un itérateur permettant de parcourir toutes les valeurs contenues dans cet objet.



localStorage et sessionStorage

Au sein du navigateur

LOCALSTORAGE

- fonctionnalités JavaScript qui permet de stocker des données de manière persistante ou temporaire dans le navigateur web.
- Les données stockées dans `localStorage` restent disponibles même après la fermeture et la réouverture du navigateur.
- Les données stockées dans `sessionStorage` restent disponibles dans la session ou l'onglet du navigateur.
- utilisé pour stocker des données locales telles que des préférences utilisateur, des données de session ou des informations de configuration.
- accessibles uniquement via JavaScript et spécifiques au domaine et au protocole utilisés pour accéder à la page.

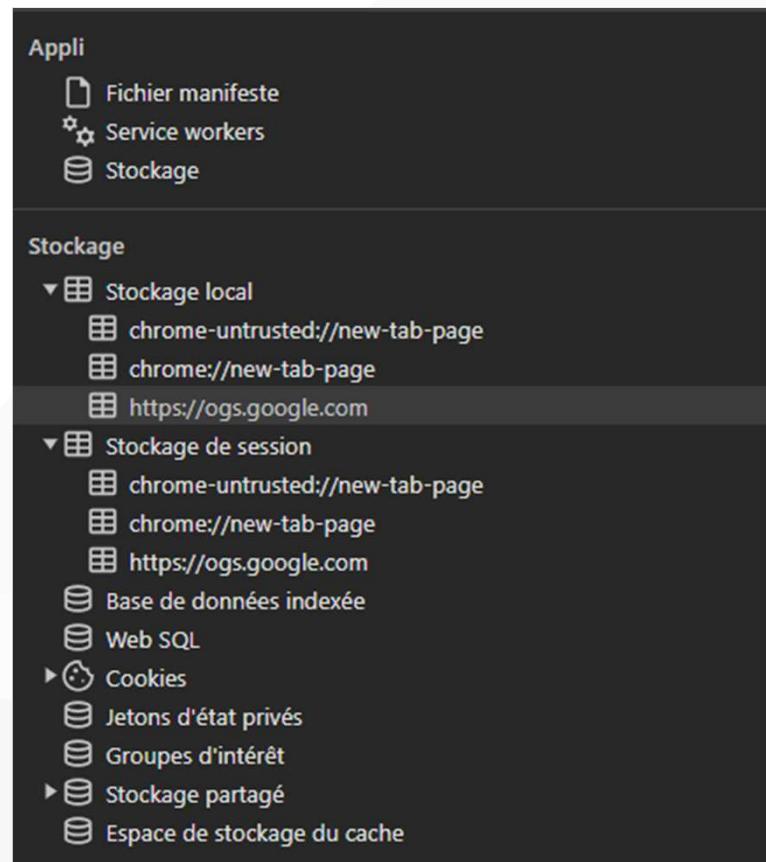
- Attention : En termes de sécurité web, l'utilisation doit être abordée avec prudence, car il peut présenter certains risques si les données sensibles sont mal gérées ou mal protégées.
- Données sensibles** : Ces données pourraient être accessibles à des scripts malveillants ou à d'autres attaques.
- Injection de code** : Ne stockez pas de données directement à partir d'entrées utilisateur sans validation et échappement appropriés. Cela pourrait permettre à des attaquants d'injecter du code malveillant dans les données stockées.
- Taille limitée** : il y a généralement une limite de taille (généralement quelques mégaoctets) par domaine. Stocker de grandes quantités de données peut affecter les performances du site web.
- Utilisation sécurisée du HTTPS** : Assurez-vous que votre site web utilise HTTPS pour chiffrer les données échangées entre le navigateur de l'utilisateur et votre serveur. Cela réduit le risque d'interception des données stockées.
- Gestion des autorisations** : Les navigateurs modernes demandent généralement la permission de l'utilisateur avant de stocker des données. Assurez-vous de respecter les préférences de l'utilisateur et de ne stocker que les données nécessaires.
- Nettoyage des données** : Évitez d'accumuler des données obsolètes ou inutiles. Assurez-vous de nettoyer régulièrement les données qui ne sont plus nécessaires pour réduire les risques en cas de compromission.

DANS L'INSPECTEUR

Depuis l'inspecteur de votre navigateur, vous pouvez avoir accès aux différents espaces de stockage.

Depuis la console, vous pouvez également manipuler le `localStorage` ou la `sessionStorage` au travers des méthodes disponibles :

- `getItem()`
- `removeItem()`
- `setItem()`
- `length()`
- `clear()`



LOCALSTORAGE VS SESSIONSTORAGE

	localStorage	sessionStorage
Type d'objet		Objet JavaScript
Portée	Domaine du site	Session de navigation
Durée de stockage	Persistant (reste jusqu'à ce qu'il soit explicitement supprimé)	Temporaire (disponible uniquement pendant la durée de la session)
Volume de stockage		Plusieurs Mo (selon le navigateur)
Accessibilité		Tous les scripts de la même origine (domaine, protocole et port)
Persistante	Persistant, reste jusqu'à ce qu'il soit explicitement supprimé	Temporaire, disparaît lorsque la session se termine (par exemple, lorsque l'onget ou le navigateur est fermé)
Utilisation typique	Stockage de données utilisateur, telles que des préférences ou des données de configuration	Stockage de données temporaires nécessaires pendant la session de navigation de l'utilisateur, telles que des informations de session utilisateur ou des états temporaires
Sécurité		Doit être utilisé avec prudence pour éviter les fuites de données sensibles
Exemple d'utilisation	Stockage des préférences utilisateur	Stockage des informations de session utilisateur

EXEMPLE

Reprenons notre exemple précédent en remplaçant la méthode basique par l'utilisation du `formData` et de `localStorage`

- Au niveau du formulaire, sur la soumission, je crée un élément `formData` à partir de mon formulaire.
- Ensuite, afin de pouvoir le transmettre, je dois effectuer une conversion en `objet Javascript` et le transformer en `JSON` (`JSON.stringify`).
 - *En effet, le localStorage ne peut contenir des objets complexes tel que formData.*

```
document
.getElementById("userForm")
.addEventListener("submit", function (event) {
  event.preventDefault();

  let isValid = checkForm();

  if (isValid) {
    // Stocker les informations dans le localStorage

    // creation du formData
    let formData = new FormData(event.target);
    // Affiche les données du formulaire dans la console
    for (let [name, value] of formData.entries()) {
      console.log(` ${name}: ${value}`);
    }

    // Convertit les données du formulaire en objet
    // le localStorage ne peut stocker que du texte donc le JSON est parfait
    // Cela implique de transformer l'objet FormData en un objet JavaScript standard,
    // puis de le convertir en chaîne JSON avec JSON.stringify().
    const formDataObject = {};
    formData.forEach((value, key) => {
      formDataObject[key] = value;
    });

    // Sauvegarde les données dans localStorage
    localStorage.setItem("userInfo", JSON.stringify(formDataObject));

    // Rediriger vers la page de compte
    window.location.href = "compte.html";
  }
});
```

EXEMPLE SUITE

Enfin sur la page de compte, je vais pouvoir récupérer les informations depuis le `localStorage` et reparcourir les données.

1. Lecture depuis le `localStorage` en fonction du nom donné et conversion en objet Javascript (`JSON.parse`)
2. Parcours de l'objet Javascript, pour l'afficher dans les bons éléments HTML.

```
// Récupérer les informations de l'utilisateur depuis le localStorage
// Lorsque vous récupérez les données depuis localStorage,
// vous devez les convertir de JSON en objet JavaScript avec JSON.parse().
const userInfoFormData = JSON.parse(localStorage.getItem("userInfo"));

// contrôle
console.info(userInfoFormData);

// si l'objet est bien présent dans le localStorage
// mise à jour des éléments HTML
if (userInfoFormData) {

    // parcours du formData
    for(const pair of Object.entries(userInfoFormData)) {

        // contrôle
        console.table(pair);

        // ciblage de l'élément correspondant
        const tag = document.getElementById(pair[0]);
        // remplissage de la valeur de l'élément
        tag.textContent = pair[1];
    }

} else {
    alert('Aucune donnée de formulaire trouvée.');
}
```

EXEMPLE DE CLASSE UTILITAIRE

Voici un exemple de classe Javascript contenant des méthodes d'accès sur `localStorage` et `sessionStorage`.

Pour pouvoir utiliser ces méthodes, il suffit de les importer :

```
// Importer les fonctions du module de stockage
import {
    saveTolocalStorage,
    saveToSessionStorage,
    getAllLocalStorageItems,
    getAllSessionStorageItems,
    clearLocalStorage,
    clearSessionStorage
} from './storage-utils.js';
```

```
// Fonctions utilitaires pour la gestion du stockage
export function saveTolocalStorage(key, data) {
    localStorage.setItem(key, JSON.stringify({
        ...data,
        date: new Date().toLocaleString()
    }));
}

export function saveToSessionStorage(key, data) {
    sessionStorage.setItem(key, JSON.stringify({
        ...data,
        date: new Date().toLocaleString()
    }));
}

export function getAllLocalStorageItems() {
    return Object.keys(localStorage)
        .map(key => ({
            key,
            value: JSON.parse(localStorage.getItem(key))
        }));
}

export function getAllSessionStorageItems() {
    return Object.keys(sessionStorage)
        .map(key => ({
            key,
            value: JSON.parse(sessionStorage.getItem(key))
        }));
}

export function clearLocalStorage() {
    localStorage.clear();
}

export function clearSessionStorage() {
    sessionStorage.clear();
}
```



A decorative graphic in the upper left corner features three overlapping circles. A large yellow circle is at the top left, a smaller red circle is to its right, and a green circle is at the bottom left, partially overlapping the yellow one.

Gestion des erreurs

TRY..CATCH...FINALLY

Peu importe notre niveau en programmation, nos scripts comportent parfois des erreurs.

- Elles peuvent être dues à nos erreurs, à une entrée utilisateur imprévue, à une réponse erronée du serveur et à mille autres raisons.

Généralement, un script “meurt” (s’arrête immédiatement) en cas d’erreur, en l’affichant dans la console.

La structure try..catch est identique à celle vu avec Java.

```
try {  
    // code...  
} catch (err) {  
    // Gestion des erreurs  
}
```

En cas d’erreur, JavaScript génère un objet contenant les détails à son sujet. L’objet est ensuite passé en argument à catch

Pour toutes les erreurs intégrées, l’objet d’erreur a deux propriétés principales :

- **Name** : Nom de l’erreur. Par exemple, pour une variable non définie, il s’agit de "ReferenceError".
- **Message** : Message textuel sur les détails de l’erreur.

Il existe d’autres propriétés non standard disponibles dans la plupart des environnements. L’un des plus largement utilisés et supportés est :

- **Stack** : Pile d’exécution en cours : chaîne contenant des informations sur la séquence d’appels imbriqués ayant entraîné l’erreur. Utilisé à des fins de débogage.

CEPENDANT, IL FAUT TENIR COMPTE DES COMPORTEMENTS DE NOTRE CODE, DÛ À L'ASYNCHRONE. LE COMPORTEMENT DE TRY CATCH EST SYNCHRONE.

Si une exception se produit dans le code “planifié”, comme dans setTimeout, try...catch ne l’attrapera pas

```
try {
  setTimeout(function() {
    noSuchVariable; // le script mourra ici
  }, 1000);
} catch (err) {
  alert( "won't work" );
}
```

C'est parce que la fonction elle-même est exécutée ultérieurement, lorsque le moteur a déjà quitté la structure try...catch.

Pour capturer une exception dans une fonction planifiée, try...catch doit être à l'intérieur de cette fonction.

```
setTimeout(function() {
  try {
    noSuchVariable; // try...catch gère l'erreur !
  } catch {
    alert( "error is caught here!" );
  }
}, 1000);
```

LEVER NOS PROPRES EXCEPTIONS

L'instruction `throw` génère une erreur.

La syntaxe est la suivante :

- `throw <error object>`

Techniquement, on peut utiliser n'importe quoi comme objet d'erreur.

- Cela peut même être une primitive, comme un nombre ou une chaîne, mais il est préférable d'utiliser des objets, de préférence avec les propriétés `name` et `message` (pour rester quelque peu compatibles avec les erreurs intégrées).

JavaScript comporte de nombreux constructeurs intégrés pour les erreurs standards : `Error`, `SyntaxError`, `ReferenceError`, `TypeError` et autres.

Nous pouvons également les utiliser pour créer des objets d'erreur.

```
let error = new Error(message);
// ou
error = new SyntaxError(message);
error = new ReferenceError(message);

error = new Error("Houston, we have a problem o_O");

alert(error.name); // Error
alert(error.message); // Houston, we have a problem o_O

// JSON.parse génère une SyntaxError
try {
  JSON.parse("{ bad json o_O }");
} catch (err) {
  alert(err.name); // SyntaxError
  alert(err.message); // Unexpected token b in JSON at position 2
}

let json = '{ "age": 30 }'; // données incomplètes

try {

  let user = JSON.parse(json); // <-- pas d'erreurs

  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name"); // (*)
  }

  alert( user.name );

} catch (err) {
  alert( "JSON Error: " + err.message ); // JSON Error: Incomplete data: no name
}
```

LES ERREURS INTÉGRÉES

En JavaScript, il existe plusieurs types d'erreurs intégrées qui peuvent être levées pendant l'exécution d'un programme.

Voici quelques-unes des erreurs les plus courantes :

1. **EvalError** : Indique une erreur concernant la fonction globale eval(). Cette erreur n'est plus levée par le moteur JavaScript moderne, mais elle existe toujours pour des raisons de compatibilité.
2. **RangeError** : Levée lorsqu'une valeur n'appartient pas à l'ensemble ou à la plage de valeurs autorisées. Par exemple, cela peut se produire lorsque vous essayez de créer un tableau avec une taille négative.
3. **ReferenceError** : Levée lorsqu'on fait référence à une variable qui n'existe pas.
4. **SyntaxError** : Levée lorsqu'il y a une erreur dans l'analyse du code JavaScript. Cela se produit souvent à cause de fautes de frappe ou de syntaxe incorrecte.
5. **TypeError** : Levée lorsqu'une valeur n'est pas du type attendu. Par exemple, l'appel d'une méthode sur une valeur null ou undefined.
6. **URIError** : Levée lorsqu'une fonction globale manipulant des URI est utilisée de manière incorrecte. Par exemple, encodeURI ou decodeURI.
7. **AggregateError** : Introduite dans ES2021, elle est utilisée pour regrouper plusieurs erreurs en une seule. Elle est souvent utilisée avec les promesses pour capturer plusieurs rejets.
8. **InternalError** : Levée lorsqu'une erreur interne se produit dans le moteur JavaScript. Ce type d'erreur est rarement vu par les développeurs, car il est généralement lié à des bugs dans le moteur JavaScript lui-même.

PROPAGER UNE EXCEPTION

Dans l'exemple précédent, nous utilisons try...catch pour gérer des données incorrectes.

Mais est-il possible qu'une autre erreur inattendue se produise dans le bloc try {...} ?

- Comme une erreur de programmation (variable is not defined) ou quelque chose d'autre, pas seulement cette “donnée incorrecte”.

Catch ne doit traiter que les erreurs qu'il connaît et “envoyer” toutes les autres.

La technique “rethrowing” peut être expliquée plus en détail comme :

- Catch obtient toutes les erreurs.
- Dans le bloc catch (err) {...} nous analysons l'objet d'erreur err.
- Si nous ne savons pas comment le gérer, nous faisons throw err.

```
let json = '{ "age": 30 }'; // données incomplètes
try {

  let user = JSON.parse(json);

  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name");
  }

  blabla(); // erreur inattendue

  alert( user.name );

} catch (err) {

  if (err instanceof SyntaxError) {
    alert( "JSON Error: " + err.message );
  } else {
    throw err; // propager (*)
  }

}
```

Exercice

créer un formulaire en HTML et un script permettant la récupération des informations qui s'afficheront dans une deuxième page.

[Bonus] : mettre en œuvre un appel à une API (Meteo, Leaflet ou Swapi)

API et Promesses

Faire des requêtes vers des serveurs distants... API, gestion de l'asynchrone.





AJAX

Asynchronous JavaScript and XML

AJAX

AJAX est une pratique de programmation qui consiste à construire des pages web plus complexes et plus dynamiques en utilisant une technologie connue sous le nom de XMLHttpRequest

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

permet de mettre à jour simplement des parties du DOM d'une page web HTML au lieu de devoir recharger la page entière.

permet également de travailler de manière asynchrone, c'est-à-dire que votre code continue à s'exécuter pendant que la partie de votre page web essaie de se recharger.

Avec les sites web interactifs et les standards modernes du web, AJAX est progressivement remplacé par des fonctions dans les cadres JavaScript et l'API standard officielle Fetch API.

FONCTIONNEMENT

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

Il y a 5 étapes pour mettre en place une requête AJAX

1. On crée un nouvel objet XMLHttpRequest.
2. On appelle sa méthode `open()` afin de l'initialiser.
3. On ajoute un gestionnaire d'évènement pour son évènement `load`, qui se déclenchera lorsque la réponse sera reçue sans erreur.
 - Dans ce gestionnaire, on appelle la méthode `initialize()` avec les données.
4. On ajoute un gestionnaire d'évènement pour son évènement `error`, qui se déclenchera s'il y a une erreur avec la requête.
5. On envoie la requête.
6. On enveloppe tout ce code dans un bloc `try...catch`, afin de gérer les éventuelles erreurs déclenchées par `open()` ou `send()`.

```
JS ajax.js > ...
1 const request = new XMLHttpRequest();
2
3 try {
4   request.open("GET", "products.json");
5
6   request.responseType = "json";
7
8   request.addEventListener("load", () => initialize(request.response));
9   request.addEventListener("error", () => console.error("Erreur XHR"));
10
11 request.send();
12 } catch (error) {
13   console.error(`Erreur XHR ${request.status}`);
14 }
15
```

INTERROGER UN SERVEUR

L'objet `XMLHttpRequest` permet de créer des requêtes HTTP en JavaScript

- `open` permet de configurer la requête HTTP
 - Type de requête (GET, POST ou PUT)
 - url cible
 - Booléen indiquant si requête asynchrone ou non
- `send` envoie la requête HTTP
 - `null` en paramètre si GET
 - Les paramètres si POST
- `responseText` est la réponse du serveur sous forme textuel

Requêtes synchrones et requêtes asynchrones

- Pour gérer une requête asynchrone, on va utiliser les événements qui vont notifier de la disponibilité de la réponse.
- Un événement de type `load` indique la fin de traitement de la requête par le serveur

```
const requestURL = 'https://neojero.github.io/JsonApi/motif.json';

// Création de la requête
const request0 = new XMLHttpRequest();

try {
    // le 3ème paramètre indique en synchrone ou asynchrone
    request0.open("GET", requestURL, true);

    // par défaut, la requête répond en texte
    request0.responseType = "json";

    // Ajout d'un écouteur pour l'événement "load"
    request0.addEventListener("load", () => {
        if (request0.status >= 200 && request0.status < 300) {
            console.info(request0.responseText);
        } else {
            console.error(`Erreur HTTP : ${request0.statusText}`);
        }
    });

    // Ajout d'un écouteur pour l'événement "error"
    request0.addEventListener("error", () => {
        console.error(`Erreur réseau lors de la requête`);
    });

    // Envoi de la requête
    request0.send();
} catch (error) {
    console.error(`Erreur inattendue : ${error.message}`);
}
```



XMLHttpRequest

Les propriétés

En créant une connexion, nous avons alors accès à un ensemble de propriétés permettant de pouvoir traiter la réponse

```
> console.warn(new XMLHttpRequest());
⚠ ▶ XMLHttpRequest {onreadystatechange: null, readyState: 0, timeout: 0,
withCredentials: false, upload: XMLHttpRequestUpload, ...} ⓘ
  onabort: null
  onerror: null
  onload: null
  onloadend: null
  onloadstart: null
  onprogress: null
  onreadystatechange: null
  ontimeout: null
  readyState: 0
  response: ""
  responseText: ""
  responseType: ""
  responseURL: ""
  responseXML: null
  status: 0
  statusText: ""
  timeout: 0
  ▶ upload: XMLHttpRequestUpload {onloadstart: null, onprogress: null, o
  withCredentials: false
  ▶ [[Prototype]]: XMLHttpRequest
```

JSON

Le langage JavaScript permet de gérer facilement ce format de données.



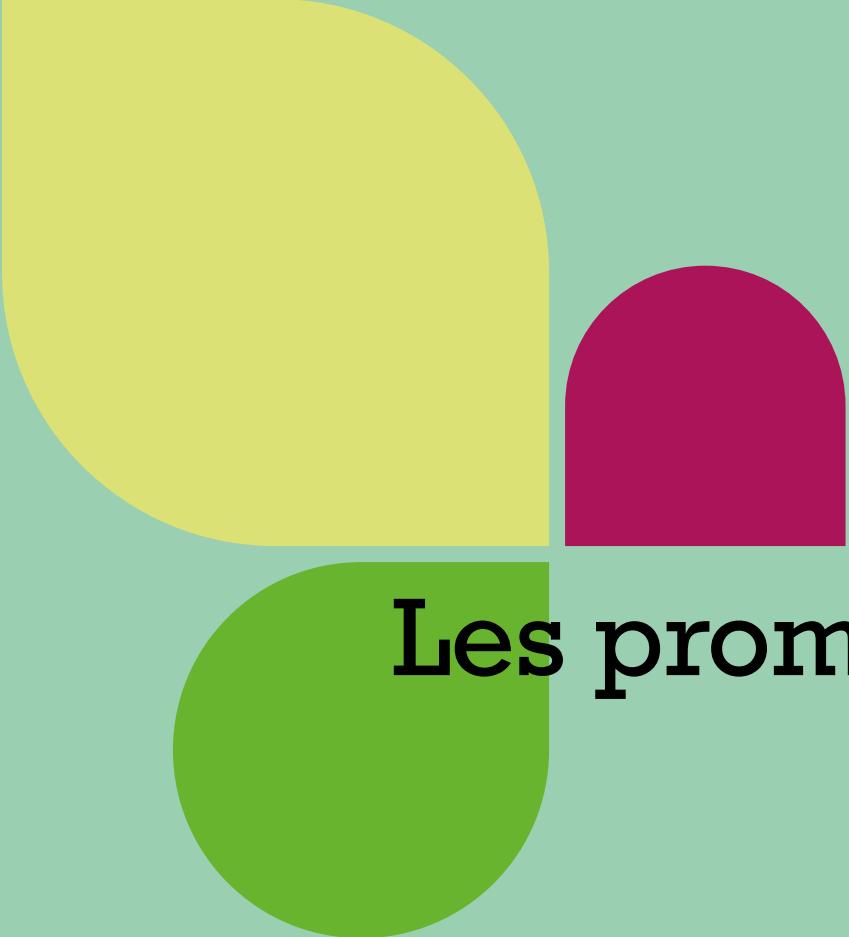
La fonction `JSON.parse` permet de transformer une chaîne de caractères conforme au format JSON en un objet JavaScript.

JSON vers Javascript



La fonction `JSON.stringify` joue le rôle inverse : elle transforme un objet JavaScript en chaîne de caractères conforme au format JSON.

Javascript vers JSON



Les promesses

Asynchrone

Par défaut, Javascript est un langage synchrone.

Dans un contexte Web, cela peut poser problèmes d'où l'utilisation de l'asynchrone afin de permettre de continuer l'exécution du code indépendamment de l'attente du résultat d'une autre méthode.

En JavaScript, nous allons utiliser beaucoup de fonctions de callback qui vont générer de l'attente pendant que le reste du programme va se poursuivre.

Inconvénient : on ne peut pas prédire la fin de l'exécution de la fonction de callback et l'ordre des différentes fonctions surtout si on a plusieurs callbacks.

solution : Les promesses vont permettre d'attendre qu'une opération asynchrone ait réussi et soit terminée avant d'exécuter un autre bout de code

Principe

```
```js
const maPromesse = new Promise((resolve, reject) => {
 const succes = true;
 if (succes) {
 resolve("Opération réussie !");
 } else {
 reject("Échec de l'opération.");
 }
});
```

- Une promesse peut être :
  - soit en cours (promis mais pas encore fait)
  - soit honorée (promis et réalisé)
  - soit rompue (on ne fait pas ce qu'on a promis et on a prévenu).

Beaucoup d'API fonctionnent aujourd'hui sur ce principe et nous aurons que peu souvent à créer nos promesses.

Cet objet à travers son exécuteur reçoit deux arguments à savoir les fonctions **resolve** et **reject**.

Le paramètre "**resolve**" correspond à une fonction qui sera exécuté si nous considérons que l'opération dans la promesse s'est bien déroulée.

Le paramètre "**reject**" est également une fonction mais celle-ci est utilisée pour indiquer à l'utilisateur une erreur.

# Les promesses

Lorsque notre promesse est créée, celle-ci possède deux propriétés internes.

Pour obtenir et exploiter le résultat d'une promesse, on va généralement utiliser la méthode `then()` et `catch()` du constructeur `Promise`.

Propriété state    `pending` (en attente)  
(état)

`fulfilled` (promesse tenue ou résolue)

`rejected` (promesse rompue ou rejetée)

Propriété result    va contenir la valeur de notre choix.

```
maPromesse
 .then(resultat => console.log(resultat))
 .catch(erreur => console.error(erreur));
````
```

Version fonction fléchée

```
promesse().then((result) => {
  // instruction pour le traitement du résultat
}).catch((error) => {
  // instruction pour le traitement d'erreurs
});
```

Utiliser et exploiter

Exemple d'une promesse

Voici un exemple simple d'une promesse permettant d'illustrer la syntaxe

Exemple

```
const direSalutation = function (salutation) {  
  return new Promise( function(resolve, reject) {  
    if(salutation === 'bonjour') {  
      resolve('Vous avez dit ' + salutation);  
    } else {  
      reject('Vous n\'avez pas salué');  
    }  
  })  
}
```

```
direSalutation("hello")  
  .then(function(result) {  
    console.log(result);  
  })  
  .catch(function(error) {  
    console.log(error);  
  });
```

Reprise de l'exemple HTTP

Si nous devions reprendre l'exemple de l'envoi HTTP, nous pourrions alors utiliser la promesse de cette manière.

```
// Utilisation de la fonction fetchData avec une Promise
fetchData(requestURL)
  .then(response => {
    console.info(response);
    let data = response.members;
    console.table(data);
  })
  .catch(error => {
    console.error(error.message);
 });
```

```
const requestURL = 'https://neojero.github.io/JsonApi/motif.json';

// Fonction qui retourne une Promise pour la requête
function fetchData(url) {
  return new Promise((resolve, reject) => {
    const request0 = new XMLHttpRequest();

    try {
      request0.open("GET", url, true);
      request0.responseType = "json";

      // Traitement de resolve
      request0.addEventListener("load", () => {
        if (request0.status >= 200 && request0.status < 300) {
          resolve(request0.response);
        } else {
          reject(new Error(`Erreur HTTP : ${request0.statusText}`));
        }
      });

      // traitement de reject
      request0.addEventListener("error", () => {
        reject(new Error(`Erreur réseau lors de la requête`));
      });

      request0.send(null);

    } catch (error) {
      reject(new Error(`Erreur inattendue : ${error.message}`));
    }
  });
}
```

Chainage et parallèle

Traitement de plusieurs promesses

Le chainage des promesses en Javascript permet d'attendre le résultat d'une promesse pour ensuite l'utiliser dans une autre promesse et ainsi de suite.

- Utilisation de `.then()` pour enchaîner des opérations asynchrones.
- Gestion des erreurs avec `.catch()` et `.finally()`.

La mise en parallèle permet de lancer des promesses en même temps :

- `Promise.all()` : Attendre que plusieurs promesses se résolvent.
- `Promise.race()` : Récupérer la première promesse résolue/rejetée.

Chainage :

```
```js
fetch('https://api.exemple.com/data')
 .then(reponse => reponse.json())
 .then(donnees => console.log(donnees))
 .catch(erreur => console.error("Erreur :", erreur));
````
```

Parallèle :

```
```js
Promise.all([promesse1, promesse2])
 .then(resultats => console.log(resultats))
 .catch(erreur => console.error(erreur));
````
```

```
async function afficherPersonnages() {
  try {
    const [luke, leia] = await Promise.all([
      fetch('https://swapi.dev/api/people/1/').then(r => r.json()),
      fetch('https://swapi.dev/api/people/5/').then(r => r.json())
    ]);
    console.log("Luke :", luke.name);
    console.log("Leia :", leia.name);
  } catch (erreur) {
    console.error("Erreur :", erreur);
  }
}

afficherPersonnages();
```

Méthodes disponibles

De l'objet Promise

Il existe d'autres méthodes disponibles avec l'objet Promise.

JavaScript Promise Methods

There are various methods available to the Promise object.

| Method | Description |
|-----------------------------------|---|
| <code>all(iterable)</code> | Waits for all promises to be resolved or any one to be rejected |
| <code>allSettled(iterable)</code> | Waits until all promises are either resolved or rejected |
| <code>any(iterable)</code> | Returns the promise value as soon as any one of the promises is fulfilled |
| <code>race(iterable)</code> | Wait until any of the promises is resolved or rejected |
| <code>reject(reason)</code> | Returns a new Promise object that is rejected for the given reason |
| <code>resolve(value)</code> | Returns a new Promise object that is resolved with the given value |
| <code>catch()</code> | Appends the rejection handler callback |
| <code>then()</code> | Appends the resolved handler callback |
| <code>finally()</code> | Appends a handler to the promise |



API Fetch

Le remplacement de XMLHttpRequest

API FETCH

Elle permet d'utiliser JavaScript depuis une page pour construire et envoyer une requête HTTP à un serveur afin de récupérer des données.

Lorsque le serveur répond en fournissant les données, le code JavaScript peut les utiliser afin de mettre à jour la page, généralement en utilisant les API de manipulation du DOM.

Les données sont généralement demandées au format JSON, mais il peut tout aussi bien s'agir de HTML ou de texte.

Cette méthode est employée largement par les sites utilisant de nombreuses données tels que Amazon, YouTube, eBay, etc.

Avec ce modèle :

- Les mises à jour des pages sont plus rapides et il n'est plus nécessaire d'attendre un rechargement de la page : le site apparaît alors comme plus rapide et réactif.
- Il y a moins de données téléchargées pour chaque mise à jour, ce qui signifie une consommation moindre de la bande passante.
- Si cela n'était pas vraiment un problème sur un ordinateur de bureau avec une connexion à très haut débit, cela pouvait vite freiner la navigation sur les appareils mobiles et/ou aux endroits où l'accès à Internet est moins rapide.

FETCH : ENSEMBLE D'OBJET ET DE FONCTIONS AFIN D'EXÉCUTER DES REQUÊTES HTTP DE MANIÈRE ASYNCHRONE. IL PERMET D'EXÉCUTER DES REQUÊTES HTTP SANS RECHARGEMENT DU NAVIGATEUR.

Envoi d'une requête de type GET

```
fetch("url")
  .then(function(res) {
    if (res.ok) {
      return res.json();
    }
  })
  .then(function(value) {
    console.log(value);
  })
  .catch(function(err) {
    // Une erreur est survenue
 });
```

Envoi d'une requête de type POST

Content-Type et Accept : indique au service Web, le type de données qu'on envoie
body : les données à envoyer.

```
fetch("url", {
  method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(jsonBody)
});
```

EXEMPLE

Récapitulons ce que fait ce fragment de script.

on utilise la fonction globale `fetch()` qui est le point d'entrée de l'API Fetch. Cette fonction prend l'URL comme paramètre

Ensuite, `fetch()` est une API asynchrone qui renvoie une promesse.

Comme `fetch()` renvoie une promesse, nous passons une fonction à la méthode `then()`. Cette méthode sera appelée lorsque le navigateur aura reçu une réponse du serveur pour la requête HTTP.

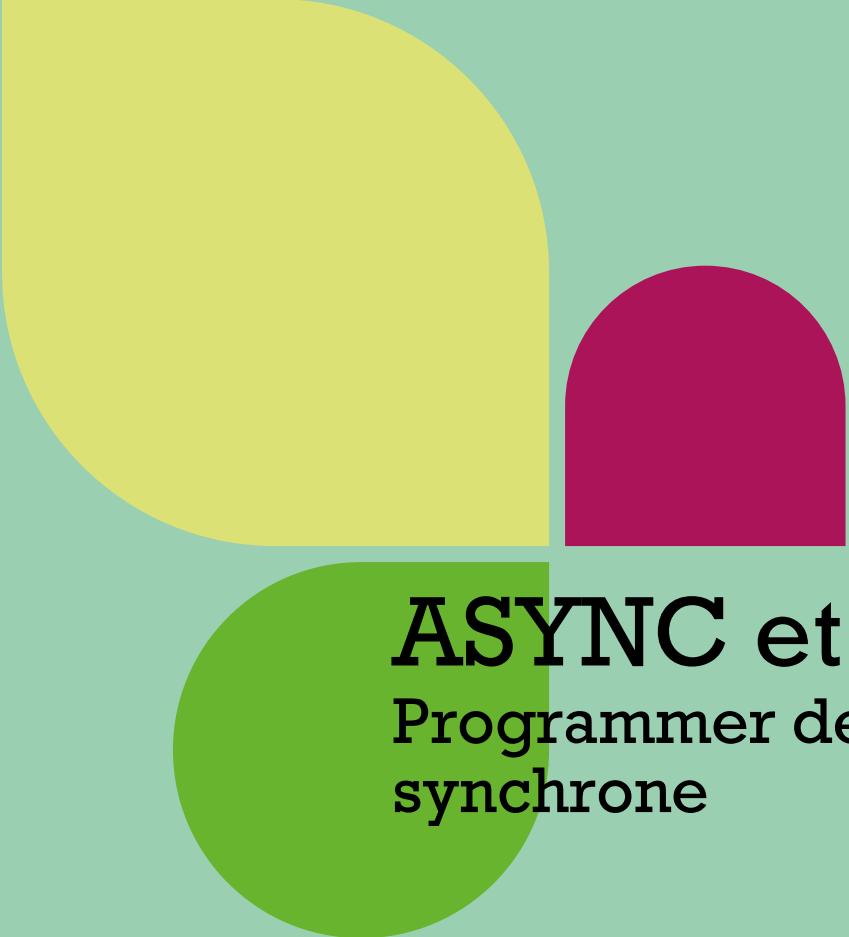
`response.json()` est également asynchrone

On passe une fonction à la méthode `then()` de cette nouvelle promesse. Cette fonction sera appelée lorsque le json sera prêt.

Enfin, on chaîne un gestionnaire `catch()` pour intercepter toute erreur qui serait déclenchée dans l'une des fonctions asynchrones ou des gestionnaires associés.

```
const requestURL = 'https://neojero.github.io/JsonApi/motif.json';

// Utilisation de l'API fetch pour récupérer les données
fetch(requestURL)
  // FETCH renvoie une promesse.
  .then(response => {
    // le gestionnaire leve une erreur si la requête a échoué
    if (!response.ok) {
      throw new Error(`Erreur HTTP : ${response.statusText}`);
    }
    // Sinon, en cas de réussite, la réponse est traité au format json
    // et retourne la réponse
    return response.json(); // Analyse la réponse en JSON
  })
  // traitement de la réponse
  .then(data => {
    console.info(data);
    let members = data.members;
    console.table(members);
  })
  .catch(error => {
    console.error(error.message);
  });
}
```



ASYNC et AWAIT

Programmer de l'asynchrone avec le synchrone

CONCEPT

`async` et `await` sont des fonctionnalités introduites dans ES2017 pour simplifier le travail avec les Promesses et permettre d'écrire du code asynchrone de manière plus lisible et synchrone.

- Une fonction `async` retourne toujours une Promesse
- `await` ne peut être utilisé qu'à l'intérieur d'une fonction `async`
- `await` met en pause l'exécution de la fonction jusqu'à ce que la Promesse soit résolue
- La gestion des erreurs se fait préférentiellement avec `try/catch`

- fonction `async` de base

```
async function fetchUserData() {  
    // Une fonction async peut utiliser await  
    // Elle retourne automatiquement une Promesse  
    return "Données utilisateur";  
}  
  
// Utilisation  
fetchUserData().then(data => {  
    console.log(data); // "Données utilisateur"  
});
```

```
async function recupererDonnees() {  
    try {  
        // Simulation d'un appel API  
        const reponse = await fetch('https://api.example.com/donnees');  
        const donnees = await reponse.json();  
        return donnees;  
    } catch (erreur) {  
        console.error('Erreur lors de la récupération:', erreur);  
    }  
}
```

LA PRINCIPALE DIFFÉRENCE AVEC LES PROMESSES CLASSIQUES

Sans `async/await`

```
function ancien() {
    return fetch('url')
        .then(reponse => reponse.json())
        .then(donnees => console.log(donnees))
        .catch(erreur => console.error(erreur));
}
```

Avec `async/await`

```
async function nouveau() {
    try {
        const reponse = await fetch('url');
        const donnees = await reponse.json();
        console.log(donnees);
    } catch (erreur) {
        console.error(erreur);
    }
}
```

EXEMPLE

Transformation de l'exemple avec l'utilisation de `async` et `await`

```
const requestURL = 'https://neojero.github.io/JsonApi/motif.json';

// Fonction asynchrone pour récupérer les données
async function fetchData() {
    try {
        const response = await fetch(requestURL);

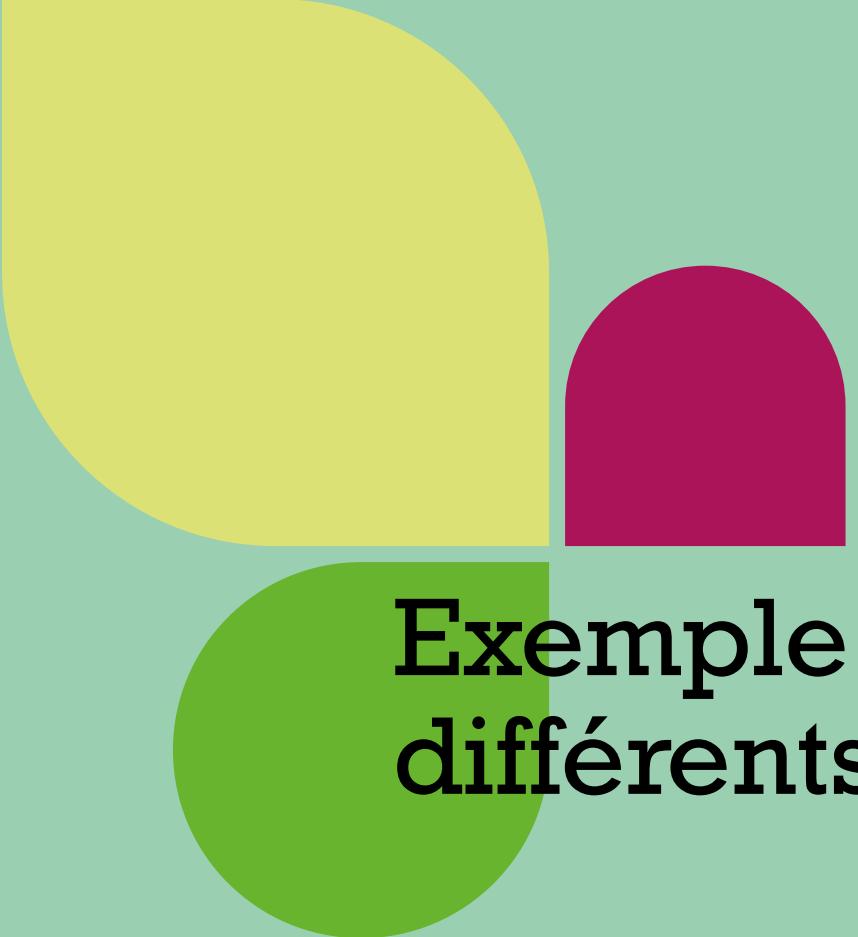
        if (!response.ok) {
            throw new Error(`Erreur HTTP : ${response.statusText}`);
        }

        const data = await response.json(); // Analyse la réponse en JSON
        console.info(data);

        let members = data.members;
        console.table(members);

    } catch (error) {
        console.error(error.message);
    }
}

// Appel de la fonction asynchrone
fetchData();
```



Exemple selon les
différents appels

Appel d'une API synchrone

Exemple 1

Appel basique

```
// 1. Avec XMLHttpRequest
function getWithXHR() {
    toggleSpinner(true);
    resultDiv.innerHTML = "Chargement avec XMLHttpRequest...";
    const xhr = new XMLHttpRequest();
    xhr.open('GET', 'https://swapi.dev/api/people/1/');
    xhr.onload = function() {
        toggleSpinner(false);
        if (xhr.status >= 200 && xhr.status < 300) {
            const data = JSON.parse(xhr.responseText);
            displayResult(data, "XMLHttpRequest");
        } else {
            resultDiv.innerHTML = `<p style="color: red;">Erreur : ${xhr.statusText}</p>`;
        }
    };
    xhr.onerror = function() {
        toggleSpinner(false);
        resultDiv.innerHTML = `<p style="color: red;">Erreur réseau.</p>`;
    };
    xhr.send();
}
```

Appel d'une API avec promesses

Exemple 2

Appel par fetch

```
// 2. Avec Fetch (version classique)
function getWithFetch() {
  toggleSpinner(true);
  resultDiv.innerHTML = "Chargement avec Fetch...";
  fetch('https://swapi.dev/api/people/1/')
    .then(response => {
      if (!response.ok) {
        throw new Error('Erreur réseau');
      }
      return response.json();
    })
    .then(data => {
      toggleSpinner(false);
      displayResult(data, "Fetch");
    })
    .catch(error => {
      toggleSpinner(false);
      resultDiv.innerHTML = `<p style="color: red;">Erreur : ${error.message}</p>`;
    });
}
```

Appel d'une API avec `async` et `Await`

Exemple 3

Await et Async

```
// 4. Avec async/await (Fetch et Axios)
async function getFilmWithAsyncAwait() {
    toggleSpinner(true);
    resultDiv.innerHTML = "Chargement du film avec async/await...";
    try {
        // Version Fetch
        const response = await fetch('https://swapi.dev/api/films/1/');
        if (!response.ok) throw new Error('Erreur réseau');
        const filmData = await response.json();

        // Version Axios (décommenter pour tester)
        // const { data: filmData } = await axios.get('https://swapi.dev/api/films/1/');

        toggleSpinner(false);
        displayFilmResult(filmData, "async/await");
    } catch (error) {
        toggleSpinner(false);
        resultDiv.innerHTML = `<p style="color: red;">Erreur : ${error.message}</p>`;
    }
}
```

Bonus : utilisation de Axios

Exemple 4

Appel par Axios

```
// 3. Avec Axios (version classique)
function getWithAxios() {
    toggleSpinner(true);
    resultDiv.innerHTML = "Chargement avec Axios...";
    axios.get('https://swapi.dev/api/people/1/')
        .then(response => {
            toggleSpinner(false);
            displayResult(response.data, "Axios");
        })
        .catch(error => {
            toggleSpinner(false);
            resultDiv.innerHTML = `<p style="color: red;">Erreur : ${error.message}</p>`;
        });
}
```

Bilan

Bilan des 3 méthodes

Problèmes et avantages

Appel Synchrone (Bloquant)

Problèmes :

- Bloque complètement l'exécution du script
- Interface utilisateur devient non réactive
- Mauvaise expérience utilisateur
- Non recommandé pour les applications modernes

Appel avec Promesses

Avantages :

- Non bloquant
- Gestion des succès et erreurs avec .then() et .catch()
- Chaînage possible des opérations
- Code plus lisible que les callbacks traditionnels

Appel avec Async/Await

Avantages :

- Syntaxe proche du code synchrone
- Plus lisible et plus proche du style de programmation séquentiel
- Gestion des erreurs avec try/catch standard
- Facilite la lecture et l'écriture de code asynchrone complexe



ALLER PLUS LOIN

Librairie et Framework

Librairie - Framework

De nombreuses possibilités.

De nombreux outils ont été créés pour faciliter la vie du développeur JavaScript et éviter de réinventer la roue à chaque nouveau projet :

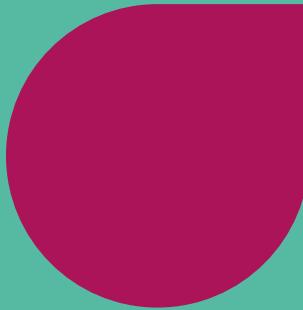
- Les librairies (ou bibliothèques), qui complètent le langage standard pour faire gagner du temps ou améliorer la qualité du code.
- Les Framework, qui fournissent un ensemble de services de base formant le squelette de l'application.

Ce qui est possible...

1. API FETCH : L'un des usages de JQuery est de faire des requêtes HTTP. L'API fetch() fait maintenant complètement parti des navigateurs, elle permet de faire des requêtes HTTP et vu que c'est intégré au moteur des navigateurs pas besoin d'importer d'autres librairies, c'est top !
2. Le JavaScript natif pour changer le contenu de la page. Désormais disponible directement depuis JavaScript (innerHTML, Événements etc..)
3. **STIMULUS** offre des possibilités. Il permet notamment via le HTML de suivre beaucoup d'événements sans se prendre la tête et sans écrire des millions de lignes de code (inutile). Pour aller plus loin, jetez un œil à [Stimulus Use](#), avec plein d'événements, faciles à importer, sans prise de tête.
<https://stimulus.hotwired.dev/>
4. **AXIOS** est un client HTTP, basée sur les promesses, compatible avec node.js et les navigateurs.
<https://axios-http.com/fr/docs/intro>
5. **Frameworks** : Angular, React, Vue.js, Next.js...
6. **API** : de nombreuses API existent et permettent d'ajouter des éléments directement dans notre HTML : [FontAwesome](#), [OpenStreetMap](#), [Nominatim](#), [OpenWeather](#), [Leafletjs](#) etc...



Version 5 - révision 2025



afpa.fr



Jérôme BOEBION