

**JS**

# Welcome to **JavaScript**



JS

## I. Introduction

- A. Un langage dynamique
- B. Où écrire le JavaScript
- C. La console
- D. Les bonnes pratiques

## II. Les variables

- A. C'est quoi une variable
- B. Nommer une variable
- C. Déclarer une variable
- D. Les types de valeur

## III. Les opérateurs

- A. Les opérateurs arithmétiques
- B. Les opérateurs d'affectation
- C. Les opérateurs de chaînes
- D. Les opérateurs de comparaison
- E. Les opérateurs logiques

## IV. Boucles et conditions

- A. Les conditions if ... else
- B. Les conditions avec switch
- C. La condition ternaire
- D. Les boucles

## V. Les tableaux

- A. Créer un tableau
- B. Rentrer dans un tableau
- C. Les multidimensionnels
- D. Parcourir un tableau

## VI. Les fonctions

- A. Créer une fonction
- B. Renvoyer une valeur
- C. Les fonctions fléchées
- D. Les fonctions callback

## VII. Les objets

- A. C'est quoi un objet
- B. Constructeur d'objets
- C. Classes et prototypes
- D. Les classes en JavaScript

## VIII. Les objets natifs

- A. L'objet String
- B. L'objet Number
- C. L'objet Math
- D. L'objet Date
- E. L'objet Array

## **IX. Le DOM**

- A. Document Object Model
- B. Accéder au DOM
- C. Modifier le DOM
- D. Les événements



1

# Introduction

Comment fonctionne JavaScript ?



# JavaScript en quelques mots

## FRONT-END

JavaScript est un langage de programmation **côté client** qui permet de créer et d'afficher le contenu d'une page web de façon **dynamique**.

Cela peut être fait en fonction de données saisies par l'utilisateur par exemple. Ainsi, contrairement à une **page web statique**, le contenu affiché pourra donc être différent selon chaque utilisateur.

## BACK-END

JavaScript peut aujourd'hui également être exécuté **côté serveur** avec Node.js pour y créer toutes sortes d'applications ou outils.

La conception d'**applications mobiles** est aussi possible avec le langage React Native également écrit en JavaScript.

Créé en 1995 par Bredan Eich, **JavaScript** a longtemps servi à créer de simples animations dans des sites web, mais il est aujourd'hui devenu l'un des langages de développement web les plus populaires.

Pour info, **JavaScript** n'a aucun rapport avec le langage **Java** : « Java is to JavaScript as ham is to hamster !! »



## Un langage dynamique

JavaScript (ou ECMAScript) est un langage qui va nous permettre de gérer dynamiquement le contenu d'une page web, c'est à dire en fonction d'éléments pouvant varier d'un moment à un autre ou d'un utilisateur à un autre.

Par opposition, d'autres langages comme HTML ou CSS sont généralement appelés statiques puisque leur contenu ne pourra pas changer selon des éléments extérieurs.

Même s'il existe aujourd'hui de plus en plus d'éléments de langages HTML ou CSS permettant à notre code de varier dynamiquement selon certains paramètres.



## Un langage dynamique

JavaScript permet de modifier des éléments HTML ou CSS en fonction de certains évènements ou données.

Par exemple, lorsqu'un utilisateur clique sur un élément d'une page web, cet évènement peut provoquer un changement du visuel de cette page web ou bien même changer son contenu.

Une donnée peut être fournie par l'environnement (la date ou même l'heure) ou bien par l'utilisateur lui-même (son âge ou son nom). Ces données peuvent elles aussi occasionner un changement du contenu ou du style de notre page web.



## Interprété & orienté objet

Vous entendrez aussi souvent dire que JavaScript est un langage interprété (par opposition aux langages compilés comme Java ou C++). Cela veut simplement dire qu'un logiciel s'est chargé de l'interpréter afin de pouvoir l'exécuter : c'est généralement le navigateur web qui fait office d'interprète.

Quant au concept de langage orienté objet, c'est une notion qui peut vite devenir complexe et nous aurons le temps d'aborder ce thème un peu plus tard. Retenez juste ici que la programmation objet est une façon de concevoir le code à travers la manipulation et la création de différents objets.



## Où écrire le code JavaScript ?

- ◉ Directement dans la **balise** ouvrante d'un **élément HTML**
- ◉ Dans une balise **<script>**, au sein d'une page HTML
- ◉ Dans un **fichier** séparé portant l'extension **.js**.



## Dans un élément HTML

Utilisée dans les débuts de JavaScript, cette pratique est aujourd'hui fortement déconseillée, au même titre que l'utilisation de l'attribut **style** dans un élément HTML.

Cette pratique était surtout utilisée pour réagir à un évènement se produisant sur un élément HTML. Il suffisait pour cela d'ajouter dans la balise ouvrante de l'élément un attribut faisant référence à cet évènement, comme **onclick** pour réagir au clic d'un utilisateur par exemple.

La valeur donnée à cet attribut était alors tout simplement l'instruction JavaScript que l'on souhaitait déclencher lorsque l'évènement se produisait.

```
<button onclick="alert('Bienvenue !')> Cliquez ici</button>
```



## Dans une balise <script>

Tout comme on peut utiliser une balise `<style>` pour écrire du code CSS à l'intérieur d'un fichier HTML, on peut utiliser une balise `<script>` pour y écrire du code JavaScript.

Cette balise peut être placée dans l'élément `<head>` tout comme dans l'élément `<body>`. Ces différents emplacements permettront de choisir le moment d'exécution de notre code JavaScript.

Cette méthode permet de commencer à mieux séparer les différents langages utilisés, même s'ils restent tous au sein d'un même fichier HTML.

```
<script>
    document.querySelector('button')
        .addEventListener('click', () =>
            alert('Bienvenue!'))
</script>
```



## Dans un fichier JS séparé

Pour garder un code le plus propre et le plus aéré possible, il est vraiment recommandé de séparer au mieux les différentes couches de code utilisées pour la création d'une page web.

De cette façon, notre code sera beaucoup plus lisible et sa maintenance sera forcément facilitée, surtout si le même code JavaScript est utilisé dans différentes pages HTML.

La meilleure façon de faire est donc de créer un fichier séparé pour y écrire notre code JavaScript et de l'enregistrer avec l'extension `.js`. Ce fichier sera ensuite appelé depuis notre fichier HTML.

On utilise pour cela la même balise `<script>` dans laquelle on ajoute l'attribut `src` pour renseigner l'URL de notre fichier JavaScript.



## Dans un fichier JS séparé

Dans le cas où l'on souhaite appliquer plusieurs scripts depuis le même fichier HTML, il faut alors les écrire séparément dans plusieurs fichiers JavaScript, puis ensuite les appeler un par un depuis des balises `<script>` individuelles.

Tout comme lorsqu'on écrit le code JavaScript directement dans une balise `<script>`, ces balises peuvent être placées dans l'élément `<head>` ou dans l'élément `<body>`.

```
<head>
    <script src="premier_script.js" ></script>
</head>
<body>
    <script src="deuxième_script.js" ></script>
</body>
```



## Quand appeler le script JS ?

JavaScript agissant sur des éléments HTML, il faut donc que ceux-ci soient déjà chargés avant d'être évoqués dans notre script. Le navigateur lisant notre code de bas en haut, il apparaît donc logique de placer notre balise `<script>` à la fin de l'élément `<body>`.

La question des performances liées aux temps de chargement des différents fichiers est aussi à prendre en compte puisque le traitement du code HTML/CSS est bloqué par le navigateur le temps qu'il finisse de charger le fichier JavaScript.

Mais il est encore trop tôt pour se pencher plus profondément sur ces notions et vous aurez tout le temps d'y revenir plus tard. Je vous laisse un article sur le sujet si vous êtes intéressés :



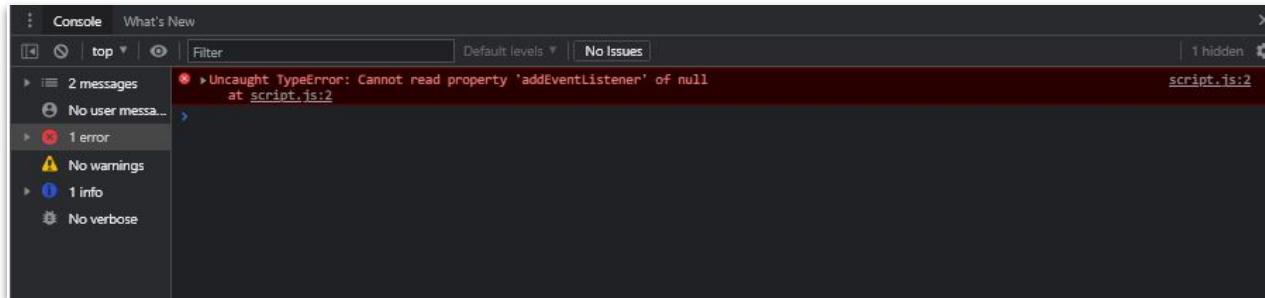
<https://www.alsacreations.com/astuce/lire/1562-script-attribut-async-defer.html>

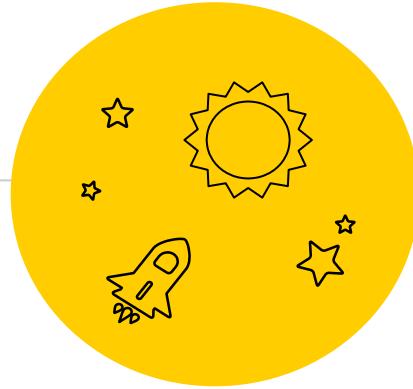


## Utiliser la console

Un des principaux outils disponibles pour travailler avec JavaScript est la console accessible depuis tous les navigateurs web (touche f12)

Cet outil permet d'y écrire du code JavaScript, d'y afficher le résultat d'un programme, tout comme d'y repérer les erreurs éventuelles qui peuvent s'y produire.





## Les bonnes pratiques



## Les commentaires en JS

De la même manière qu'il est recommandé de commenter son code HTML ou CSS, il est très important de s'habituer dès le début à renseigner le mieux possible son code JavaScript avec des commentaires clairs et pertinents.

Cela permet d'être facilement compris par les personnes amenées à travailler sur notre code, mais également de se comprendre soi-même lorsqu'on revient sur un code que nous avons écrit il y a quelques mois ou même quelques semaines...

Ces commentaires ne seront pas lus par le navigateur, on peut donc aussi "neutraliser" un bout de code JavaScript en utilisant la syntaxe des commentaires. Les commentaires resteront par contre toujours lisibles dans le code source de la page web.



## Les commentaires en JS

Pour écrire un commentaire d'une ou plusieurs lignes, il suffit de l'entourer des signes /\* et \*/.

Pour un commentaire d'une seule ligne, on le fait précéder du signe //.

```
script.js  X
script.js > ...
1
2  document.querySelector('button').addEventListener('click', () => alert('Bienvenue!'))
3
4  // Un commentaire sur une seule ligne
5
6  /*
7  Un commentaire
8  sur plusieurs lignes
9  */
10
```



## L'indentation

L'indentation en JavaScript se fait de la même manière qu'en HTML et CSS, cela consiste à décaler certaines lignes de code par rapport à d'autres.

Cela permet de garder un code beaucoup plus lisible et aéré, et donc plus agréable à lire et plus simple à comprendre aussi.

Nous aurons l'occasion de mettre cela en pratique assez rapidement, mais comme pour les autres langages, l'indentation se fait tout simplement dès qu'une partie de code est comprise dans une autre.



## Les points-virgules

Le code JavaScript est principalement composé d'instructions et on informe qu'une instruction est terminée en y ajoutant un point-virgule à la fin.

C'est une pratique qu'il faut vraiment se forcer à tout le temps respecter, et cela même si vous verrez beaucoup de code JavaScript dans lequel les point-virgules n'ont pas été mis.

Le langage JavaScript peut en effet comprendre de lui-même qu'une instruction est terminée et l'interpréter correctement mais il peut également ajouter des points-virgules en pensant qu'une instruction est terminée alors qu'elle ne l'est pas.

Pour éviter des bugs inattendus dans notre code, il est donc préférable de bien penser à terminer chaque instruction par un point-virgule.

2

# Les variables

Classer les données



## C'est quoi une variable ?

Dans un langage de programmation, une variable est ce qui nous permet de stocker une information de façon à pouvoir l'utiliser ultérieurement.

Une variable fait en fait référence à une case mémoire de l'ordinateur dans laquelle est stockée l'information de notre choix. Il suffit ensuite de lui choisir un nom pour retrouver plus facilement cette case au milieu de toutes les autres.

Vous pouvez vous représenter une variable comme une boîte avec une étiquette indiquant ce que cette boîte contient comme information.



# C'est quoi une variable ?





## Nommer une variable

Pour déclarer puis "initialiser" une variable, il suffit d'abord de la nommer, pour pouvoir plus tard lui assigner la valeur que l'on souhaite stocker à l'intérieur.

Certaines conventions doivent être respectées au moment de choisir le nom d'une variable :

- Il ne doit pas commencer par un chiffre
- Il ne peut pas contenir d'espace ni de caractères spéciaux (excepté \$ et l'underscore \_)
- Il ne peut contenir de mots "réservés", c'est à dire des mots-clés du langage JavaScript
- Il doit être le plus significatif possible



[https://developer.mozilla.org/#reserved\\_keywords\\_as\\_of\\_ecmascript\\_2015](https://developer.mozilla.org/#reserved_keywords_as_of_ecmascript_2015)



## Nommer une variable

Si le nom de la variable contient plusieurs mots, il est courant d'utiliser l'écriture **camelCase** ou bien l'underscore **\_** pour séparer les mots :

*nombreVisiteurs* ou *nombre\_visiteurs*

Le langage JavaScript est également sensible à la casse, faites donc bien attention en écrivant le nom d'une variable :

*nom\_Variable* ≠ *nom\_variable*



## Déclarer une variable

Le symbole utilisé pour stocker une valeur dans une variable est le signe égal `=`. En JavaScript, ce signe est un symbole d'*affectation* ou *assignation* :

```
nomUtilisateur = "Philippe";  
ageUtilisateur = 45;
```

Ici, la variable nommée `nomUtilisateur` contient la valeur *Philippe*, et la variable `ageUtilisateur` contient la valeur *45*.

Comme son nom l'indique, le contenu d'une variable peut-être amené à changer, il suffit pour cela de lui assigner une nouvelle valeur, et celle-ci viendra remplacer, ou "écraser" l'ancienne valeur :

```
nomUtilisateur = "Soufiane";
```



## Déclarer une variable

Pour créer une variable, un mot-clé doit être utilisé. Il en existe plusieurs dont nous verrons les spécificités. Depuis la dernière version de JavaScript, appelée **ES6**, il est recommandé d'utiliser le mot-clé **let**.

```
let nomUtilisateur = "Philippe";
```

Maintenant, la variable **nomUtilisateur** existe et la valeur qu'elle contient ("**Philippe**") peut donc être utilisée dans notre script.

Deux autres mots-clés existent pour déclarer une variable : le mot-clé **var** était précédemment utilisé à la place du mot-clé **let**. Il peut encore être utilisé aujourd'hui dans certains cas.

Le mot-clé **const** permet, lui, de créer une variable appelée constante dont la valeur ne pourra pas être changée

```
const pi = 3.14116;
```



## Mémento

- On choisit le nom d'une variable selon le type d'information que l'on voudrait y stocker :

*nomUtilisateur*

- On crée ensuite cette variable en la déclarant à l'aide d'un mot-clé :

*let nomUtilisateur;*

- On initialise cette variable en lui assignant la valeur qu'elle va contenir :

*nomUtilisateur = "Philippe";*

- On peut très bien créer et initialiser une variable en une seule et même étape :

*let nomUtilisateur = "Philippe";*



## Différences entre "let" et "var"



<https://blog.nicolas.brondin-bernard.com/differences-entre-var-let-et-const-en-javascript/>



<https://www.youtube.com/watch?v=4z1-0AqjQOQ&t=346s>



[https://www.youtube.com/watch?v=jYR\\_KOLGTgU](https://www.youtube.com/watch?v=jYR_KOLGTgU)



## Afficher dans la console

Comme précisé précédemment, un des outils les plus utilisés pour faire des vérifications en JavaScript est la console présente dans tous les navigateurs.

On peut tout simplement écrire directement dedans, ou bien demander à y afficher quelque chose depuis notre script en utilisant la méthode `console.log()` :

```
let nomUtilisateur = "Philippe";
console.log(nomUtilisateur);
```

On demande alors au navigateur d'afficher dans la console le résultat de ce qui se trouve entre les parenthèses.



## Afficher dans la console

```
File Edit Selection View Go Run Terminal Help  
mon-script.js X  
scripts > mon-script.js > ...  
1  
2 let nomUtilisateur = "Philippe";  
3 console.log( nomUtilisateur );  
4  
5
```



Elements Performance Console Sources Network  
top ▾ Filter  
Philippe



## Les types de valeurs en JS

Il existe 7 types de valeurs en JavaScript. Toute valeur stockée dans une variable appartiendra donc forcément à l'un de ces types.

Le type de valeur **string**, appelé aussi **chaîne de caractères**, permet de stocker des valeurs alphanumériques. Ce type de valeur doit être écrit entre guillemets, simples ou doubles.

```
let mon_nom = "Philippe";
let user_name = 'Tom74';
```



## Les types de valeurs en JS

Si des guillemets doubles sont utilisés à l'intérieur d'une chaîne de caractères, des guillemets simples doivent être utilisés pour entourer celle-ci, et inversement :

```
let une_phrase = 'mon nom est "Philippe".';
let une_autre_phrase = "j'aime le chocolat";
```

Une autre solution consiste à échapper avec un antislash \ certains caractères utilisés dans une chaîne de caractères :

```
let une_phrase = "mon nom est \"Philippe\". ";
let une_autre_phrase = 'j\'aime le chocolat ';
```



## Les types de valeurs en JS

Le type de valeur **number** permet de stocker des valeurs numériques : des nombres entiers ou des nombres avec des virgules, des valeurs positives comme des valeurs négatives. En JavaScript, un nombre à virgule est écrit avec un point.

```
let mon_age = 45;  
let prix_dessert_du_jour = 9.95;
```

Ce type de valeur permet également de stocker le résultat d'une opération mathématique :

```
let prix_dessert_en_promo = 9.95 - 2;
```



## Les types de valeurs en JS

Le type de valeur **boolean** permet de stocker une variable dont la valeur ne pourra être que *true* ou *false*. Cela permet en général de vérifier si une information est vraie ou fausse, et de donner des instructions différentes en fonction de cela :

```
let is_user_logged = true;
```

Nous apprendrons à manipuler ce type de valeur lorsque nous aborderons les conditions en JavaScript.

**NB:** Un booléen s'écrit sans guillemets, sinon cela ne serait plus un booléen mais une simple chaîne de caractères.



## Les types de valeurs en JS

Le type de valeur ***undefined*** est le type donné par JavaScript à une variable qui a été créée mais à laquelle aucune valeur n'a été assignée.

Le type de valeur ***null*** est légèrement différent : cela correspond à une valeur assignée à une variable mais qui ne représente rien en soi.

Les types de valeurs ***object*** et ***symbol*** sont, elles, plus spécifiques à ce qu'on appelle des objets en JavaScript. Nous nous pencherons sur ces notions ultérieurement.

Ne vous inquiétez pas si ces notions vous semblent abstraites, voire incompréhensibles pour le moment, c'est parfaitement normal. Vous aurez tout le temps de les découvrir et de les comprendre au moment voulu.



## Les types de valeurs en JS

L'opérateur `typeof` en JavaScript permet de vérifier le type de valeur stockée dans une variable :

The screenshot shows a browser's developer tools console tab labeled "Console". The console interface includes icons for selection, copy, Elements, Performance, Console (which is selected), Sources, and Network. Below the tabs are buttons for refresh, stop, top, and filter, along with a "Filter" input field. The main area displays the following JavaScript code and its output:

```
> let user_logged = true;
    typeof user_logged;
< 'boolean'
>
```

Attention cependant à l'utilisation de cet opérateur qui peut dans certains cas donner des résultats qui peuvent surprendre...



## undefined / is not defined

Deux termes se ressemblent et peuvent sembler vouloir dire la même chose en JavaScript : **undefined** et **is not defined**.

Comme on vient de le voir, **undefined** est le type donné par JavaScript à une variable qui a été créée mais à laquelle aucune valeur n'a été assignée.

**is not defined** est quant à lui une erreur indiquée lorsqu'on essaie d'utiliser une variable qui n'existe pas.

```
> let variable_connue;
  • undefined
> variable_inconnue;
✖ ▶ Uncaught ReferenceError: variable_inconnue is not defined
      at <anonymous>:1:1
> |
```



## Not A Number ...

Un terme est utilisé par JavaScript lorsque des opérations mathématiques sont effectuées à l'aide de valeurs n'étant pas des nombres : **NaN**, une abréviation pour **Not A Number**.

```
> 4 * "string";
< NaN
>
```

Cette valeur renvoyée par JavaScript vous permettra d'identifier plus facilement l'erreur se trouvant dans votre script.



## True or false

Lorsqu'une condition doit être vérifiée pour pouvoir exécuter un script en particulier, on demande en fait à JavaScript d'évaluer si ce qu'on lui soumet est vrai ou faux. Il le fera à l'aide des booléens `true` et `false`.

Autrement dit, toute valeur ou expression en JavaScript peut être évaluée `true` ou `false`. Pour être plus précis, toute valeur vaut initialement `true` excepté les valeurs suivantes :

*Undefined*

*Nan* (Not A Number)

*False*

*Null*

`""` (chaîne de caractère vide)

`0` (le chiffre zéro)

Gardez cela en simple mémo pour le moment pour y revenir au moment où vous en aurez besoin.



3

# Les opérateurs

Ou comment manipuler les valeurs



## Les différents opérateurs

Les opérateurs en JavaScript sont des symboles utilisés pour manipuler de différentes manières les variables et leurs valeurs. Les principaux opérateurs disponibles sont les suivants :

- Les opérateurs arithmétiques
- Les opérateurs d'affectation
- Les opérateurs de chaînes
- Les opérateurs de comparaison
- Les opérateurs logiques
- L'opérateur ternaire conditionnel



# Les opérateurs arithmétiques

Les opérateurs **arithmétiques** sont des symboles qui vont nous permettre d'effectuer différentes opérations mathématiques sur des valeurs de type **number**.

Opérateur	Nom de l'opération associée
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo (reste d'une division euclidienne)
**	Exponentielle (élévation à la puissance d'un nombre par un autre)



## Les opérateurs arithmétiques

L'opérateur **modulo** permet généralement de vérifier si la valeur d'une variable est paire ou impaire :

```
> let passenger_seat = 24;
< undefined
> passenger_seat % 2;
< 0
> |
```

Le résultat de l'opération *passenger\_seat* étant **0**, on en déduit alors que la valeur de cette variable est impaire.



## Les opérateurs arithmétiques

Lorsque plusieurs opérations sont effectuées à la suite, *l'ordre de priorité* des opérations en mathématiques sera appliqué.

Pour forcer cet ordre de priorité, il faudra alors entourer de deux *parenthèses* l'opération que l'on souhaite rendre prioritaire.

```
> 2 * 6 + 4;  
↳ 16  
> 2 * (6 + 4);  
↳ 20  
> |
```



## Les opérateurs d'affectation

Lors de la création d'une variable nous avons vu que le symbole égal = était utilisé pour lui affecter une valeur.

Je peux donc maintenant assigner à une variable le résultat d'une opération mathématique effectuée sur la valeur d'une autre variable :

```
> let dessert = 9.90;
← undefined
> let dessert_en_promo = dessert * 0.2;
← undefined
> |
```



## Les opérateurs d'affectation

Il est également possible de combiner un opérateur *arithmétique* avec l'opérateur d'*affectation* pour effectuer une opération sur une variable, puis assigner à cette variable le résultat de l'opération :

```
> let users_number = 100;
< undefined
> users_number += 50;
< 150
> users_number;
< 150
```

La variable *users\_number* comprend maintenant la valeur **150** qui est venue écraser l'ancienne valeur.



## Les opérateurs d'affectation

Une méthode appelée **incrémentation** ou **décrémentation** est une écriture raccourcie permettant d'ajouter ou bien d'ôter **1** à la valeur d'une variable :

```
nombreVisiteurs = nombreVisiteurs + 1;
```

```
nombreVisiteurs += 1;
```

```
nombreVisiteurs ++;
```

Cette notation est fréquemment utilisée lorsqu'on souhaite incrémenter ou décrémenter un compteur.

**NB:** Nous verrons également plus tard qu'il est possible de distinguer une post-incrémantion et une pré-incrémantion en décidant de placer le symbole **++** avant ou après la variable.



## Les opérateurs de chaînes

Lorsqu'on manipule des chaînes de caractères en JavaScript, le symbole `+` nous permet de fusionner deux chaînes de caractères entre elles. Cette opération est appelée une **concaténation**.

```
> let prenom = "Filip";
← undefined
> let nom = "Lopez";
← undefined
> let phrase = "Bonjour, mon nom est " + prenom + ' ' + nom;
← undefined
> |
```



## Les opérateurs de chaînes

Depuis la dernière mise à jour du langage JavaScript appelée **ES6** ou **ECMAScript 2015**, la manipulation des chaînes de caractères a été simplifiée, avec notamment l'utilisation des *backquotes*.

L'exemple précédent peut maintenant s'écrire de la façon suivante :

```
> let phrase2 = `Bonjour, mon nom est ${prenom} ${nom}`;
```

```
< undefined
```

```
> |
```

Tout ce qui sera écrit entre les symboles `${...}` sera automatiquement remplacé par la valeur à laquelle cela renvoie.



## Les opérateurs de chaînes

**NB:** Il est important de préciser le fonctionnement de JavaScript lors de l'utilisation de l'opérateur `+` entre des chaînes de caractères et des nombres.

Après avoir rencontré une chaîne de caractère suivie de l'opérateur `+`, JavaScript va considérer les nombres suivants comme des chaînes de caractères également. Plus précisément, il va *convertir* ces nombres en chaînes de caractères.

```
> "une chaîne de caractères suivie de " + 1 + 2 + 3;  
< 'une chaîne de caractères suivie de 123'  
> |
```





## Les opérateurs de comparaison

Il est très courant dans un langage de programmation de choisir d'exécuter un programme qu'à certaines conditions. C'est là que les opérateurs de **comparaison** interviennent. Ils permettent d'effectuer des comparaisons entre plusieurs valeurs ou expressions en renvoyant un **booléen** comme résultat de cette comparaison.

```
> let age_visiteur = 16;
< undefined
> age_visiteur >= 18;
< false
> |
```

Ici, on pourrait alors décider de refuser l'accès à certaines fonctionnalités d'un site grâce au résultat de cette comparaison.



# Les opérateurs de comparaison

Opérateur	Définition
<code>==</code>	Permet de tester l'égalité sur les valeurs
<code>===</code>	Permet de tester l'égalité en termes de valeurs et de types
<code>!=</code>	Permet de tester la différence en valeurs
<code>&lt;&gt;</code>	Permet également de tester la différence en valeurs
<code>!==</code>	Permet de tester la différence en valeurs ou en types
<code>&lt;</code>	Permet de tester si une valeur est strictement inférieure à une autre
<code>&gt;</code>	Permet de tester si une valeur est strictement supérieure à une autre
<code>&lt;=</code>	Permet de tester si une valeur est inférieure ou égale à une autre
<code>&gt;=</code>	Permet de tester si une valeur est supérieure ou égale à une autre



## Les opérateurs de comparaison

L'opérateur de comparaison `==` permet de vérifier l'égalité *stricte* entre deux expressions, c'est-à-dire au niveau de leur *valeur* mais également au niveau de leur *type*.

```
> "4" == 4;
< true
> "4" === 4;
< false
> |
```

Dans le premier cas, JavaScript ayant deux types différents à comparer, il va convertir la *chaîne de caractère* en un *nombre* pour pouvoir ensuite simplement comparer leur valeur.



## Les opérateurs de comparaison

En JavaScript, le symbole `!` signifie la *négation* : il va en fait renvoyer *l'inverse* de l'expression qui va suivre.  
L'opérateur `!=` va donc permettre de vérifier qu'une expression est bien *differente* d'une autre.

```
> let mot_mystere = "ballon";
< undefined
> let reponse1 = "soleil";
< undefined
> reponse1 != mot_mystere;
< true
> |
```

On pourrait donc ici envoyer un message d'erreur informant que le mot mystère n'a pas été trouvé.



## Les opérateurs logiques

Trois opérateurs logiques sont disponibles en JavaScript : `&&` (AND), `||` (OR) et `!` (NOT).

L'opérateur `&&` va nous permettre de nous assurer que *deux* conditions soient vérifiées : il renverra alors *true*. Il suffit qu'une des deux conditions ne soit pas vérifiée pour qu'il renvoie *false*.

```
> let jour = "lundi";
< undefined
> let heure = 10;
< undefined
> jour != "dimanche" && heure >= 9;
< true
```

Si l'une des deux expressions était fausse, alors l'accès à la promo de la semaine serait par exemple impossible.



## Les opérateurs logiques

L'opérateur `||` va nous permettre de nous assurer qu'*au moins une* condition sur deux est vérifiée : il renverra alors `true`. Il renvoie également `true` si les deux conditions sont vérifiées.

```
> let avis_utilisateur = "pas terrible";
< undefined
> avis_utilisateur == "mauvais" || avis_utilisateur == "pas terrible";
< true
```

Un message d'excuse voire un geste commercial pourrait alors être envisagé si l'une des deux expressions est vérifiée ...



## Les opérateurs logiques

Comme vu précédemment, l'opérateur `!` nous permet de retourner *l'inverse* de l'expression placée juste après ce symbole.

```
> let connected_user = true;
```

```
< undefined
```

```
> !connected_user;
```

```
< false
```

```
> |
```

Cet opérateur nous donne la possibilité de rapidement vérifier la véracité ou non d'une information : ici, si un utilisateur est connecté ou non.



## Les opérateurs logiques

**NB:** Pensez à bien utiliser le symbole `==` lorsque vous souhaitez comparer deux expressions. Attention à ne pas utiliser le symbole `=`. Ce dernier n'est pas un opérateur de *comparaison* mais un opérateur d'*affectation*.

```
> let avis_utilisateur = "pas terrible";
< undefined
> avis_utilisateur = "mauvais" || avis_utilisateur = "pas terrible";
✖ Uncaught SyntaxError: Invalid left-hand side in assignment
```

Vous éviterez ainsi beaucoup de temps perdu à ne pas comprendre pourquoi un script que vous pensez correct ne fonctionne pas...



## Précédence et associativité

Lorsque plusieurs opérateurs JavaScript sont utilisés dans la même expression, des règles de priorité appelées règles de **précédence** et d'**associativité** sont appliquées pour décider de l'ordre dans lequel ces différents opérateurs seront traités.

Les opérateurs possédant le plus haut degré de **précédence (0)** seront traités en priorité. Si des opérateurs possèdent le même degré, on appliquera alors les règles d'**associativité**. Celles-ci déterminent si une expression est lue **de gauche à droite**, ou inversement **de droite à gauche**.

Ces règles et leur application peuvent sembler très obscures à ce moment de votre apprentissage du JavaScript mais rassurez-vous : vous n'avez absolument pas besoin de maîtriser ces concepts pour le moment... 😊



# Précédence et associativité

Précérence	Opérateur (nom)	Opérateur (symbole)	Associativité
0	Groupement	( ... )	Non applicable
1	Post-incrémentation	... ++	Non applicable
1	Post-décrémentation	... —	Non applicable
2	NON (logique)	! ...	Droite
2	Pré-incrémentation	++ ...	Droite
2	Pré-décrémentation	— ...	Droite
3	Exponentiel	... ** ...	Droite
3	Multiplication	... * ...	Gauche
3	Division	... / ...	Gauche
3	Modulo	... % ...	Gauche
4	Addition	... + ...	Gauche
4	Soustraction	... - ...	Gauche

Précérence	Opérateur (nom)	Opérateur (symbole)	Associativité
5	Inférieur strict	... < ...	Gauche
5	Inférieur ou égal	... <= ...	Gauche
5	Supérieur strict	... > ...	Gauche
5	Supérieur ou égal	... >= ...	Gauche
6	Égalité (en valeur)	... == ...	Gauche
6	Inégalité (en valeur)	... != ...	Gauche
6	Egalité (valeur et type)	... === ...	Gauche
6	Inégalité (valeur ou type)	... !== ...	Gauche
7	ET (logique)	&&	gauche
8	OU (logique)		gauche
9	Ternaire	... ? ... : ...	Droite
10	Affectation (simple ou combiné)	... = ... , ... += ... , ... -= ... , etc.	Droite



## Raccourci avec l'opérateur ||

L'opérateur logique `||` peut nous permettre de prévoir une valeur par défaut, dans le cas où la valeur ne serait pas fournie par un utilisateur par exemple.

Comme cet opérateur nécessite qu'une seule des expressions soit évaluée *true*, si la première expression vaut *false*, JavaScript va continuer à lire la suite. Mais si cette première expression vaut *true* alors JavaScript ne lira pas ce qui suit.

```
> "" || "user";
< 'user'
```

Dans le cas présent, JavaScript nous retourne la première expression évaluée à *true*, soit le deuxième choix proposé. En effet, rappelez-vous qu'une *chaîne de caractères vide* est évaluée *false* par JavaScript.



## L'opérateur ternaire

L'opérateur **ternaire** est une spécificité apportée par **ES6**. Il permet d'écrire des conditions de manière très raccourcie et simplifiée. Sa syntaxe est la suivante :

*condition à vérifier ? code à exécuter si true : code à exécuter si false ;*

```
> let user_age = 16;
← undefined
> user_age >= 18 ? "open" : "closed";
← 'closed'
```

Nous étudierons cet opérateur plus en détail au moment d'aborder les conditions en JavaScript.



## Les structures de contrôle

Les **structures de contrôle** sont tout simplement les instructions que nous allons utiliser pour **contrôler** l'exécution d'un bloc de code JavaScript.

Elles sont principalement gérées à l'aide de **conditions** et/ou de **boucles** : c'est à dire que les deux peuvent très bien être combinées ou non.

Les **conditions** nous permettent de choisir des paramètres à respecter obligatoirement pour que notre code puisse être exécuté.

Les **boucles** nous permettent d'exécuter notre code **en boucle**, autant de fois que nous l'aurons décidé, selon des paramètres que nous aurons préalablement établis.



4

# Boucles & conditions

Le début de la programmation...



## Les conditions if... else if... else

Les **conditions** sont un moyen de contrôler de manière extrêmement précise la façon dont nous voulons que notre code JavaScript soit exécuté.

Nous pouvons ainsi décider qu'un script en particulier soit exécuté si une ou des conditions sont remplies, et qu'un script différent soit exécuté dans le cas contraire. La syntaxe principale est la suivante :

```
if(condition à remplir) {  
    code à exécuter;  
}  
else {  
    code à exécuter;  
}
```



```
> let user_age = 16, acces = "";  
< undefined  
> if (user_age >= 18) {  
    acces = "open";  
} else {  
    acces = "closed";  
}  
< 'closed'
```



## Les conditions if... else if... else

Nous pouvons écrire autant de conditions qu'on veut en utilisant à chaque fois le mot-clé `else if`. Il faudra cependant bien penser à terminer avec un `else` qui sera là pour gérer tous les cas non envisagés par les conditions précédentes.

Javascript va lire notre code de haut en bas, ligne après ligne. Dès qu'il arrivera à une condition pouvant être satisfaite, il exécutera le bloc de code correspondant et ignorera toutes les conditions suivantes. Si aucune condition ne peut être remplie, alors JavaScript exécutera le dernier bloc de code correspondant au `else final`.

Il est donc primordial d'être très attentif à l'ordre dans lequel nous allons écrire nos différentes conditions pour être certain de ne pas avoir de résultat non attendu dans l'exécution de notre code.



## Les conditions if... else if... else

```
> let message_accueil, heure;
< undefined
> if (heure < 8) {
    message_accueil = "il n'est pas encore 8h";
}
else if (heure < 12) {
    message_accueil = "il est entre 8h et 12h";
}
else if (heure < 18) {
    message_accueil = "il est entre 12h et 18h";
}
else if (heure < 24){
    message_accueil = "il est entre 18h et minuit";
}
else {
    message_accueil = "je ne sais pas l'heure qu'il est";
}
< "je ne sais pas l'heure qu'il est"
```



## Les conditions if... else if... else

Dans l'exemple précédent, comme nous n'avons pas assignée de valeur à notre variable **heure**, elle ne correspond donc à aucune des conditions listées et l'instruction exécutée sera alors celle prévue "*par défaut*" dans le **else** final.

Si par inattention, nous nous étions trompés dans l'ordre d'écriture des conditions en proposant par exemple en deuxième choix la condition **if (heure < 24)**, nous aurions alors provoqué l'erreur suivante :

Si la variable **heure** avait pour valeur **13**, la condition **if (heure < 24)** serait alors satisfaite avant même que soit lue la condition **if (heure < 18)**, et le message d'accueil affiché ne serait pas celui escompté... 😞



## Les conditions if... else if... else

```
> let message_accueil, heure=13;  
< undefined  
> if (heure < 8) {  
    message_accueil = "il n'est pas encore 8h";  
}  
else if (heure < 24){  
    message_accueil = "il est entre 18h et minuit";  
}  
else if (heure < 12) {  
    message_accueil = "il est entre 8h et 12h";  
}  
else if (heure < 18) {  
    message_accueil = "il est entre 12h et 18h";  
}  
else {  
    message_accueil = "je ne sais pas l'heure qu'il est";  
}  
< 'il est entre 18h et minuit'<
```



## Les conditions avec switch

L'instruction **switch** représente une alternative à l'utilisation de **if ... else**. Le fonctionnement et la syntaxe sont un peu différents.

Avec **switch**, nous allons nous focaliser sur une **variable** précise et énumérer les différentes **valeurs** que cette variable pourrait prendre avec, pour chaque cas, un code à exécuter.

```
switch (variable) {  
    case valeur 1 : code à exécuter ;  
    break ;  
    case valeur 2 : code à exécuter ;  
    break ;  
    default : code à exécuter ;  
}
```



## Les conditions avec switch

En face de chaque cas de **valeur** possible est indiqué le **code** correspondant à exécuter, puis on termine avec le mot-clé **break** qui nous permet de sortir de l'instruction **switch** : sinon JavaScript continuerait à tester inutilement les autres cas énumérés.

Tout à la fin, le mot-clé **default** nous permet de prévoir un code à exécuter pour le cas où la variable ne correspondrait à aucun des cas prévus. Cette instruction a la même fonction que le **else** final dans la syntaxe des **if ... else**.

Comme avec les **if ... else**, Javascript va lire notre code de haut en bas, ligne après ligne. Dès qu'il arrivera à une condition pouvant être satisfaite, il exécutera le bloc de code correspondant et ignorera toutes les conditions suivantes.



## Les conditions avec switch

```
> let meteo = "pluie", tenue = "";
< undefined
> switch(meteo) {
    case "soleil": tenue = "t-shirt";
    break;
    case "nuageux": tenue = "pull";
    break;
    case "pluie": tenue = "k-way";
    break;
    default: tenue = "comme tu veux";
}
< 'k-way'
```



## Les conditions avec switch

Il est possible d'utiliser des *opérateurs de comparaisons* à l'intérieur de l'instruction **switch** pour proposer de manière plus précise différentes conditions à tester. Lors de ce type d'utilisation de l'instruction **switch**, le booléen **true** sera utilisé à la place de la variable pour pouvoir le comparer à chaque condition énumérée.

Lorsqu'une condition est satisfaite, elle est donc évaluée à **true** et le bloc de code correspondant est alors exécuté.

```
switch (true) {  
    case condition 1 : code à exécuter ;  
    break ;  
    case condition 2 : code à exécuter ;  
    break ;  
    default : code à exécuter ;  
}
```



## Les conditions avec switch

```
> let message_accueil, heure = 13;
← undefined
> switch (true) {
    case heure < 8 : message_accueil = "il n'est pas encore 8h";
    break;
    case heure < 12 : message_accueil = "il est entre 8h et 12h";
    break;
    case heure < 18 : message_accueil = "il est entre 12h et 18h";
    break;
    case heure < 24 : message_accueil = "il est entre 18h et minuit";
    break;
    default : message_accueil = "je ne sais pas l'heure qu'il est";
}
← 'il est entre 12h et 18h'
```



## La condition ternaire

Comme vu précédemment, la syntaxe **ES6** nous offre une syntaxe raccourcie pour écrire une structure conditionnelle : l'**opérateur ternaire**. Pour rappel, la syntaxe est la suivante :

*condition à vérifier ? code à exécuter si true : code à exécuter si false ;*

C'est une syntaxe beaucoup plus légère et lisible si elle est correctement utilisée. Il est également possible de multiplier les conditions proposées de la manière suivante :

*condition 1 ? code à exécuter si true :*

*condition 2 si false ? code à exécuter si true :*

*condition 3 si false ? code à exécuter si true :*

*code à exécuter si false ;*



## La condition ternaire

```
> let meteo = "pluie";
← undefined
> let tenue =
    meteo == "soleil" ? "t-shirt" :
    meteo == "nuages" ? "pull" :
    meteo == "pluie" ? "k-way" :
    "comme tu veux";
← undefined
> tenue;
← 'k-way'
```



## Bonnes pratiques

Une bonne pratique à adopter en programmation consiste à *alléger son code* pour le rendre le plus *visible* et *efficace* possible. Il est indispensable pour cela d'éviter les répétitions et tout ce qui peut rendre notre code redondant ou inutilement verbeux.

Un exemple dans l'utilisation des *booléens* illustre très bien cela. Lors de l'écriture d'une *condition*, on a souvent le réflexe de vouloir vérifier qu'une expression est vraie en la comparant à *true*. Or la fonction première de la condition *if* est justement de confirmer si cette expression est *true* ou bien *false*.

Il est donc très simple d'alléger la syntaxe de nos comparaisons de la manière suivante :

*if(user\_logged == true)*  
*if(user\_logged)*





## Les boucles

Les **boucles** nous permettent d'exécuter une instruction **en boucle**, tant qu'une condition ou des conditions que nous aurons préalablement définies seront vérifiables.

Pour fonctionner correctement, une **boucle** a besoin de trois paramètres principaux :

- Une **valeur référence** à pouvoir tester tout au long de la boucle
- Une **condition** à vérifier pour sortir ou non de la boucle
- Un **itérateur** pour pouvoir changer la valeur référence à chaque tour de boucle

**NB:** Quelque soit le type de boucle utilisée, il est impératif de prévoir que la condition que nous avons définie puisse être fausse à un moment ou un autre pour éviter de tomber dans une boucle infinie et de faire crasher notre programme... 🚧



## La boucle while

La boucle **while (tant que)** est celle qui possède la syntaxe la plus simple : une **condition** et une **instruction** à exécuter tant que cette condition sera valide :

```
while (condition) {  
    code à exécuter ;  
}
```

Il est donc nécessaire de veiller à ce que les trois paramètres indispensables au bon fonctionnement d'une boucle soient bien tous présents, notamment l'itérateur pour pouvoir sortir de la boucle lorsque la condition ne sera plus satisfaite.

Dans la boucle **while**, la variable utilisée pour vérifier la validité de notre condition sera initialisée **en amont** puis sera incrémentée **à l'intérieur** de notre boucle, à chaque tour qu'elle effectuera.



## La boucle while

Nous voulons par exemple incrémenter un compteur un certain nombre de fois. Nous créons donc une boucle avec un *test* à passer à l'entrée de celle-ci suivi de deux *instructions* à exécuter si le test est réussi : *afficher* la valeur du compteur dans la console, puis *incrémenter* le compteur.

A chaque tour de boucle la même opération se répète : test, affichage puis incrémantation, jusqu'à ce que le compteur ne puisse plus passer le test d'entrée dans la boucle qui s'arrête donc.

```
> let compteur = 0;  
← undefined  
> while(compteur < 5){  
    console.log(compteur);  
    compteur++;  
}
```



0
1
2
3
4



## La boucle do while

La boucle **do while** (*fais tant que*) fonctionne de la même manière que la boucle **while**. La différence principale se trouve dans la chronologie des différentes étapes.

Dans cette boucle, *l'instruction* à exécuter précède la *condition* à vérifier. De ce fait, même si la condition n'est jamais satisfaite, le code sera toujours exécuté au moins une fois.

```
do {  
    code à exécuter ;  
}  
while (condition)
```

Le choix d'utiliser l'un ou l'autre de ces deux boucles se fera selon vos besoins, mais surtout selon la lisibilité que vous souhaitez apporter à votre code.



## La boucle do while

```
let secret_number = 15;
let user_number = 0;

do {
    user_number = +prompt('Please choose a number between 1 and 20');
    if (user_number == secret_number) {
        alert('Well done !');
    }
}
while (user_number != secret_number)
```

L'instruction ***prompt*** permet d'afficher une pop-up pour demander à l'utilisateur de saisir une donnée.

L'instruction ***alert*** permet d'afficher une simple pop-up contenant un message.



## La boucle for

On retrouve dans la boucle **for** les trois paramètres indispensables au bon fonctionnement d'une boucle. Ils sont ici placés entre parenthèses, et séparés par des points-virgules.

```
for (initialisation; condition; itération) {  
    instruction à exécuter ;  
}
```

*L'initialisation* de la variable référence se fait ici à l'intérieur de la boucle. Elle est habituellement nommée **i (itérator)** mais vous pouvez parfaitement lui donner un autre nom. Elle est généralement initialisée à **zéro**.

Le deuxième paramètre, soit la **condition**, permet de rentrer dans la boucle. *L'itération* se fait à la sortie de la boucle, c'est à dire après l'exécution de *l'instruction* à l'intérieur de la boucle.



## La boucle for

```
> for (let i=0; i<5; i++) {  
  console.log(i);  
}  
  
0  
1  
2  
3  
4
```

*let i=0;* → *initialisation*

*i < 5;* → *condition*

*i++;* → *itération*

Des déclinaisons spécifiques de la boucle **for** existent nous permettant de parcourir des **objets** : ce sont les boucles **for ... in**, **for ... of** et **for await ... of**. Nous les verrons au moment d'aborder les objets.



## continue & break

Deux instructions sont disponibles à l'intérieur des fonctions, et donc à l'intérieur des boucles, pour optimiser leur performance dans certaines situations.

L'instruction **continue** permet de directement passer à l'*itération* suivante sans même exécuter l'*instruction* prévue. Cela nous permet de n'appliquer des instructions que dans certaines conditions.

L'instruction **break** permet d'*arrêter* l'exécution d'une boucle dès qu'une condition particulière est satisfaite. En effet, une fois la valeur recherchée trouvée, il n'est pas utile de continuer l'exécution de la boucle.



## L'instruction continue

On pourrait par exemple utiliser l'instruction **continue** pour afficher uniquement les chiffres *pairs* entre **0** et **20** :

```
for (let i=0; i<10; i++) {  
    if (i%2 != 0) {  
        continue;  
    }  
    console.log("chiffre pair : " + i);  
}
```



```
chiffre pair : 0  
chiffre pair : 2  
chiffre pair : 4  
chiffre pair : 6  
chiffre pair : 8
```

Quand un chiffre est *impair*, l'instruction **console..log** est ignorée et la boucle passe à l'itération suivante.



## L'instruction break

L'instruction **break** est très utile lorsqu'on recherche une valeur précise dans un *tableau* ou un *objet*.

Nous utiliserons ici cette instruction pour afficher le chiffre **5** dès que celui-ci sera trouvé lors du décompte.

```
for (let i=0; i<20; i++) {  
    if (i == 5) {  
        console.log("chiffre 5 trouvé !");  
        break;  
    }  
    console.log(i);  
}
```



0
1
2
3
4
chiffre 5 trouvé !

Dès que le compteur arrive au chiffre **5**, l'instruction contenue dans le *if* est exécutée et la boucle est arrêtée.



5

# Les tableaux

Stocker toujours plus...



## Créer un tableau

Un **tableau** est un type de variable permettant de contenir **plusieurs valeurs** quelque soit leur type : des chaînes de caractères, des nombres, ou même d'autres tableaux. Voir même d'autres tableaux contenant d'autres tableaux...

Pour être plus précis, un tableau est en fait un **objet** mais nous aurons le temps d'étudier cela plus en profondeur au moment d'aborder les objets.

On peut créer un tableau comme on crée une **variable**, en plaçant simplement entre crochets les différentes valeurs qu'il contient :

```
let mon_tableau = [ valeur1, valeur2, valeur3 ];
```



## Rentrer dans un tableau

Chaque *valeur* d'un tableau possède un *indice numérique* pour accéder directement à cette valeur. Les indices commencent à partir de **0**.

On accède à l'une des valeurs d'un tableau en indiquant le nom du tableau suivi de *crochets* contenant l'indice correspondant à cette valeur :

*prenoms[0];*

```
> let fruits = ["bananes", "mangues", "kiwis"];
< undefined
> fruits[0];
< 'bananes'
```



## Rentrer dans un tableau

On peut également modifier la valeur d'un tableau, ou même ajouter une valeur à un tableau avec la même syntaxe.

```
          0      1      2
> let fruits = ["bananes", "mangues", "kiwis"];
< undefined
> fruits[2] = "pommes";
< 'pommes'
> fruits;
          0      1      2
< ▶ (3) [ 'bananes', 'mangues', 'pommes' ]
```



## Rentrer dans un tableau

```
> let fruits = ["bananes", "mangues", "kiwis"];
← undefined
> fruits[3] = "bananes";
← 'bananes'
> fruits;
          0           1           2           3
← ▶ (4) ['bananes', 'mangues', 'kiwis', 'bananes']
```



## Les multidimensionnels

On appelle *tableau multidimensionnel* un tableau contenant plusieurs niveaux d'écriture, c'est à dire qu'il contient un ou plusieurs tableaux pouvant eux aussi contenir un ou plusieurs tableaux...

On accède aux différentes valeurs de ce type de tableau en suivant la même logique que pour un simple tableau :

```
> let un_peu_tout = [20, ["John", "Doe"], 104, "code", ["JavaScript", 2015, "es6"]];
< undefined
> un_peu_tout[4];
< ▶ (3) ['JavaScript', 2015, 'es6']
> un_peu_tout[4][2];
< 'es6'
```



## Parcourir un tableau

Lorsqu'on cherche une valeur particulière au sein d'un tableau, on a la possibilité de parcourir les valeurs de ce tableau une à une en utilisant la boucle **for**.

```
let langages = ['HTML', 'CSS', 'JavaScript', 'PHP'];

for (let i=0; i<langages.length; i++) {
    console.log(langages[i]);
};
```



HTML
CSS
JavaScript
PHP

On précisera alors dans la **condition** de la boucle la **longueur du tableau** avec la syntaxe **.length**. Puis dans l'**instruction**, on accèdera aux différentes valeurs du tableau en donnant comme **indice** la variable incrémentée à chaque tour de boucle.



## Parcourir un tableau

La variable *i* prenant une valeur différente à chaque tour de la boucle, *langages[i]* correspondra donc à un élément différent du tableau à chaque fois :

```
Tour 1 : langages[i] --> langages[0] --> 'HTML'  
Tour 2 : langages[i] --> langages[1] --> 'CSS'  
Tour 3 : langages[i] --> langages[2] --> 'JavaScript'  
Tour 4 : langages[i] --> langages[3] --> 'PHP'
```



## Parcourir un tableau

Il existe cependant une boucle avec une syntaxe beaucoup plus légère pour parcourir les tableaux : la boucle **for ... of**.

Tout va reposer ici sur la façon de **nommer nos variables**. Habituellement, le nom donné au tableau est un dénominatif **pluriel**, et le nom de la variable référence dans la boucle est son **singulier**. Cette variable référence représentera successivement chaque élément du tableau.

```
let langages = ['HTML', 'CSS', 'JavaScript', 'PHP'];
for (let langage of langages) {
    console.log(langage);
}
```



<i>Tour 1</i> : <i>langage</i> --> 'HTML'
<i>Tour 2</i> : <i>langage</i> --> 'CSS'
<i>Tour 3</i> : <i>langage</i> --> 'JavaScript'
<i>Tour 4</i> : <i>langage</i> --> 'PHP'



6

# Les fonctions

Don't Repeat Yourself...



## C'est quoi une fonction ?

Le but d'une **fonction** est de pouvoir effectuer plusieurs fois la même tâche sans avoir à réécrire à chaque fois le code correspondant à celle-ci.

C'est donc un **ensemble d'instructions** déjà écrites et qu'on peut réutiliser à volonté. Lorsque nous aurons besoin de cette fonction, nous aurons seulement à l'appeler par son nom suivi de parenthèses.

*nom\_fonction()*

Nous pouvons créer les fonctions que nous voulons selon nos besoins mais il existe également des fonctions déjà toutes prêtées, appelées **fonctions natives**. Vous en avez déjà utilisé quelques unes sans le savoir : `console.log()` ou même les différentes boucles comme `for()` ou `switch()`...



## Créer une fonction

Pour créer une fonction, nous utilisons le mot-clé ***function*** suivi du nom que nous avons choisi et d'une paire de ***parenthèses***. Puis nous ajoutons entre ***accolades*** les différentes ***instructions*** qui constituent le cœur de notre fonction.

```
function ma_fonction () {  
    instructions;  
}
```

**NB:** Comme pour toutes les autres variables, il est très important de bien choisir le ***nom*** que nous allons donner à notre fonction. Il faut que celui-ci soit assez précis et évocateur pour tout de suite comprendre à quoi sert cette fonction. On utilise généralement un ***verbe*** suivi d'un nom :

```
check_password()
```



# Créer une fonction

```
1 > function say_hello() {  
    console.log("Hello World!");  
}  
< undefined  
2 > say_hello();  
3 Hello World!
```

- 1 - *Création de la fonction*
- 2 - *Appel de la fonction*
- 3 - *Exécution de la fonction*



## Créer une fonction

Il est possible de stocker notre fonction dans une *variable*. La différence entre les deux syntaxes va porter sur l'accessibilité des fonctions : quand peut-on les appeler ?

```
let ma_fonction = function () {  
    instructions;  
}
```

Dans le premier cas, la fonction sera accessible partout, même avant qu'elle soit déclarée. Dans le second, on ne pourra l'appeler qu'après l'avoir déclarée... Cela peut sembler difficile à comprendre pour le moment. Je vous laisse donc un lien pour y revenir plus tard :



<https://javascript.info/function-expressions>



## Créer une fonction

Nous pouvons indiquer à l'intérieur des parenthèses des *paramètres* à utiliser à l'intérieur de notre fonction. Au moment d'appeler la fonction, il faudra alors fournir les données correspondantes à ces paramètres.

Créons par exemple une fonction permettant de multiplier deux nombres entre eux : nous aurons besoin de fournir à la fonction deux nombres pour qu'elle puisse fonctionner. On appellera cela lui *passer des arguments*.

```
> function multiply(number1, number2) {  
    console.log(number1 * number2);  
}  
< undefined
```



```
> multiply(4,5);  
20  
< undefined
```



## Renvoyer une valeur

Une fonction peut *retourner* une valeur qu'on pourra stocker dans une variable ou bien utiliser directement. On utilise pour cela le mot-clé **return** suivi de la valeur à retourner.

Attention, cette *instruction* met automatiquement fin à l'exécution de notre fonction. Si d'autres instructions sont présentes après, elles seront tout simplement ignorées. Il faut donc être vigilant sur l'endroit où l'écrire.

**NB:** Cette instruction est complètement différente de celle d'un **console.log()** dont le rôle est simplement d'afficher quelque chose dans la console.

```
> function multiply(numb1, numb2) {  
    return numb1 * numb2;  
}  
let mult_result = multiply(4,5);  
  
multiply(mult_result, 2);  
< 40
```



## Renvoyer une valeur

**NB:** Par défaut, une fonction doit toujours *retourner* quelque chose, c'est pour cette raison que la console nous retourne ***undefined*** si cette instruction n'est présente dans une fonction.

```
> function multiply(number1, number2) {  
    console.log(number1 * number2);  
}  
< undefined
```



```
> multiply(4,5);  
20  
< undefined
```



Le premier ***undefined*** est juste un comportement de la **console** qui attend de chaque déclaration écrite qu'elle produise un résultat et qui, quand ce n'est pas le cas, renvoie ***undefined***.



## Fonction anonyme

Il est possible de créer une fonction *anonyme*, c'est à dire *sans lui donner de nom*. On n'utilisera ce type de fonction qu'à un seul endroit de notre script. C'est pour cette raison qu'on ne lui donne pas de nom : parce que nous n'aurons pas besoin de la rappeler à un autre endroit.

On utilise le plus souvent ce type de fonction lorsqu'on veut la lier à un *événement* en particulier : lors du click sur un bouton par exemple. Elle sera alors exécutée à chaque fois que cet événement se produira.

```
document.querySelector('button').onclick = function() { alert('fonction exécutée !') }
```



## Fonction récursive

En programmation, le concept de récursivité permet de produire une boucle à l'aide d'une fonction. Une **fonction récursive** est en fait une fonction qui va *s'appeler elle-même* dans ses instructions.

Tout comme dans une boucle, il faudra inclure une **condition** à vérifier avant d'appeler la fonction pour ne pas que celle-ci tourne indéfiniment.

```
function compteur(num) {  
    if (num > 0) {  
        console.log(num);  
        return compteur(num-1);  
    }  
};  
  
compteur(5);
```



5
4
3
2
1



## Les fonctions fléchées

L'une des nouveautés amenées par **ES6** est une syntaxe raccourcie pour les fonctions : les **fonctions fléchées** :

```
let myFonction = function (params) {  
    return a value ;  
}
```



```
let myFonction = (params) => a value ;
```

Si notre fonction possède un **return** comme seule **instruction**, alors les **accolades** et le mot-clé **return** ne sont plus nécessaires. Mais si notre fonction contient **plusieurs instructions**, il faudra continuer à les placer entre des **accolades** et préciser aussi le **return**.



## Les fonctions fléchées

Si une fonction n'a besoin que d'un seul *paramètre*, alors les *parenthèses* ne seront plus nécessaires :

```
> let say_hello = name => 'Hello ' + name;
< undefined
> say_hello('Sam');
< 'Hello Sam'
```

La syntaxe de *fonction fléchée* est très utilisée pour alléger le code lors de l'écriture d'une fonction *anonyme* :

```
document.querySelector('button').onclick = () => alert('fonction exécutée !')
```



## Les fonctions fléchées

Les *fonctions fléchées* peuvent être pratiques à utiliser dans certains cas mais dans certains contextes, cela peut s'avérer déconseillé. Ces différences d'utilisation vont souvent concernez des notions pas encore, ou peu abordées.

Il serait cependant intéressant de revenir sur le sujet lorsque la question se posera. Dans cette optique, je vous laisse ici un article traitant de ce sujet :



<https://dmitripavlutin.com/differences-between-arrow-and-regular-functions/>



## Les fonctions callback

Un concept important en JavaScript est celui de **callback** : c'est quand une **fonction** est passée en **argument** à l'intérieur d'une autre fonction et ne sera exécutée que sous condition. Ce concept est couramment utilisé lors de la gestion d'actions **asynchrones**, c'est à dire quand une action ne doit être accomplie qu'après la fin d'une autre action.

Au moment d'étudier les objets, nous verrons des fonctions natives comme **map** ou **filter** qui utilisent le **résultat** d'une fonction passée en argument pour produire un résultat en particulier. Ces fonctions sont très utiles pour **boucler un tableau** et pouvoir agir sur chacune de ses valeurs.



## Les fonctions callback

```
function mySandwich(param1, param2, callback) {
    alert(`Started making my sandwich with ${param1} and ${param2}`);
    callback();
}

mySandwich('meat', 'cheese', function() {
    alert('Now eating my sandwich');
});
```

Le dernier *paramètre* de la fonction *mySandwich* est une fonction *callback* qui ne sera exécutée qu'après exécution de la première instruction. Au moment d'appeler la fonction *mySandwich*, une fonction *anonyme* est donnée en dernier *paramètre* : elle sera la fonction *callback*.



## Le Rest parameter ...

Le **Rest parameter** est une des nouvelles fonctionnalités d'**ES6** qui nous permet de pouvoir écrire une **fonction** sans avoir à préciser le nombre de **paramètres** qui seront utilisés par cette fonction.

```
let myFonction = function (...params) { ... }
```

Dans cette syntaxe, **params** représente tous les **arguments** qui seront passés à cette fonction. Ils seront alors stockés dans un tableau portant le même nom : **params**. Le **Rest parameter** peut très bien être utilisé en plus d'autres arguments dont le nombre sera précisé.

```
let myFonction = function (param1, param2, ...params) { ... }
```

Dans cette syntaxe, **params** représentera alors tous les **arguments** passés en plus des deux premiers. Ils seront également stockés dans un tableau portant du même nom : **params**.



## Le Rest parameter ...

```
> let welcome_user = (...langages) => {
    return langages;
};

welcome_user("CSS", "JavaScript", "PHP", "React");
< ▶ (4) ['CSS', 'JavaScript', 'PHP', 'React']
```



```
> let welcome_user = (name, ...langages) => {
    return `Hello ${name}! So, you want to learn about ${langages}`;
};

welcome_user("Sami", "CSS", "JavaScript", "PHP", "React");
< 'Hello Sami! So, you want to learn about CSS,JavaScript,PHP,React'
```



7

# Les objets

La Programmation Orientée Objet



## C'est quoi un objet ?

La *Programmation Orientée Objet* est simplement une façon de concevoir l'écriture de son code. En JavaScript, tout est vu comme un **objet** : c'est à dire que tous les éléments composant un programme sont des **objets** que nous faisons interagir les uns avec les autres.

Pour rappel, un **objet** est l'une des valeurs pouvant être stockée dans une variable en JavaScript. Il consiste en un ensemble de différentes données : des **propriétés** et des **méthodes**. Les propriétés sont de simples données et les méthodes des fonctionnalités.

Mettons par exemple que notre objet soit une voiture. Il pourrait avoir comme propriétés le nom de la marque, le nom du modèle, le nombre de places, et comme méthodes le fait de pouvoir rouler, de freiner, de tourner...



## Déclarer un objet littéral

Il existe de multiples façons de déclarer un objet en JavaScript, et nous n'allons pas toutes les citer ici. Nous allons nous concentrer sur les plus simples :

La **déclaration littérale** consiste à déclarer un **objet** comme on déclare une simple **variable**. La syntaxe est donc la suivante :

```
let voiture = {  
    marque : "Volkswagen",  
    modèle : "Touran",  
    nombre_places : 7,  
    rouler : function() { ... }  
}
```



## Parcourir un objet

On accède au différentes *propriétés* et *méthodes* d'un objet avec un *point* ou des *crochets*. On peut ainsi lire, les modifier ou même en ajouter.

*nom\_objet.propriété* / *nom\_objet["propriété"]*  
*nom\_objet.méthode* / *nom\_objet["méthode"]*

```
> let voiture = {  
    marque: "Volkswagen",  
    modèle: "Touran",  
    nombre_places: 7,  
    rouler: () => alert('Je roule')  
};  
< undefined
```



```
> voiture.modele;  
< 'Touran'  
> voiture.diesel = true;  
< true  
> voiture["rouler"] = () => alert('je roule très bien');  
< () => alert('je roule très bien')
```



## Parcourir un objet

```
> voiture;
<   ▼ {marque: 'Volkswagen', modele: 'Touran', nombre_places: 7, diesel: true, rouler: f}
      i
      diesel: true
      marque: "Volkswagen"
      modele: "Touran"
      nombre_places: 7
    ▶ rouler: () => alert('je roule très bien')
```

En regardant notre *objet* complet, on voit que la propriété *diesel* a bien été ajoutée et que la méthode *rouler* a bien été modifiée.



## Parcourir un objet

Comme il existe la boucle spécifique `for ... of` pour parcourir les tableaux, il existe la boucle `for ... in` pour parcourir les objets. La variable `property` prendra tour à tour le nom de chaque `propriété` de l'objet.

```
for (const property in object) {  
    instruction;  
}
```

```
for (const property in voiture) {  
    console.log(property + " : " + voiture[property]);  
}
```



```
marque : Volkswagen  
modele : Touran  
nombre_places : 7  
rouler : () => alert('Je roule')
```



## Constructeur d'objets

Il est courant d'avoir à créer plusieurs *objets* qui auraient le même type de *propriétés* et de *méthodes*, mais avec des valeurs différentes, comme par exemple les différents produits d'un site de vente en ligne.

Une **fonction constructeur** existe en Javascript, nous permettant de créer ce type de *schéma* et donc de pouvoir fabriquer plus rapidement des objets partageant les mêmes critères.

Ainsi les objets fabriqués à partir de ce schéma hériteront des propriétés et des méthodes définies lors de la création de notre constructeur d'objets.

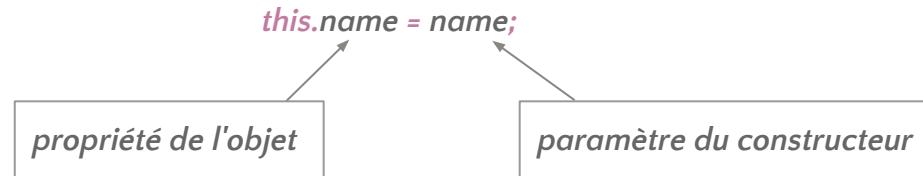
Comme pour les autres fonctions, il nous faudra la *nommer*. Par convention, on lui ajoutera une *majuscule* pour mieux l'identifier comme étant une **fonction constructeur**.



## Constructeur d'objets

```
function Article (name, price, sizes) {  
    this.name = name;  
    this.price = price;  
    this.sizes = sizes;  
    this.popup = function () {  
        alert(` check our brand new article: ${this.name}!!`)  
    }  
}
```

Le mot-clé **this** fait référence à l'**objet** dont nous sommes en train de créer le schéma. Au moment de fabriquer les différentes **copies** de cet objet, **this** fera référence à l'objet qui sera en train d'être créé.





## Constructeur d'objets

Pour créer un nouvel *objet* à partir de ce *constructeur*, on utilise le mot-clé **new** et on passe en *arguments* de notre fonction les valeurs qui seront propres à notre nouvel objet :

```
let vintage_shirt = new Article ("Vintage Shirt", 39.95, ['M', 'L']);
```



<i>this.name</i>	-->	<i>vintage-shirt.name</i>	-->	"Vintage Shirt"
<i>this.price</i>	-->	<i>vintage-shirt.price</i>	-->	39.95
<i>this.sizes</i>	-->	<i>vintage-shirt.sizes</i>	-->	['M', 'L']



## Constructeur d'objets

Chaque *nouvel objet* créé avec le constructeur **Article** possèdera donc les propriétés *name*, *price*, *sizes* ainsi que la méthode *popup()*:

```
> vintage_shirt;
< ▶Article {name: 'Vintage Shirt', price: 39.95, sizes: Array(2), popup: f}
  name: "Vintage Shirt"
  ▶popup: f ()
  price: 39.95
  ▶sizes: (2) ['M', 'L']
```

```
> vintage_shirt.popup();
```





## Constructeur d'objets

Il est également possible pour un *objet* de posséder ses propres *propriétés* ou *méthodes* qui ne seront bien sûr pas disponibles pour les autres objets provenant pourtant du même constructeur.

```
> vintage_shirt.colors = ["white", "pink", "black"];
< ▷(3) ['white', 'pink', 'black']
> vintage_shirt;
< ▶ Article {name: 'Vintage Shirt', price: 39.95, sizes: Array(2), colors: Array(3), popup: f}
  ▶ colors: (3) ['white', 'pink', 'black']
    name: "Vintage Shirt"
  ▶ popup: f()
    price: 39.95
  ▶ sizes: (2) ['M', 'L']
```



## Constructeur d'objets

Si nous vérifions notre **constructeur**, nous verrons bien qu'il ne possède pas cette nouvelle *propriété*, puisqu'elle n'était pas présente lors de sa création. Et si nous créons un nouvel objet à partir de ce même constructeur, cette nouvelle propriété n'existera donc pas non plus pour lui.

```
> let yellow_shirt = new Article ("Yellow Shirt", 19.95, ['S', 'M', 'L', 'XL']);
< undefined
> yellow_shirt;
< ▼Article {name: 'Yellow Shirt', price: 19.95, sizes: Array(4), popup: f} ⓘ
  name: "Yellow Shirt"
  ▶popup: f ()
  price: 19.95
  ▶sizes: (4) ['S', 'M', 'L', 'XL']
```



## Classes et prototypes

La plupart des langages de *Programmation Orientée Objet* fonctionnent selon un système de **classes**.

Une **classe** est un schéma servant à construire plusieurs objets similaires. Elle possèdent des propriétés et des méthodes dont **hériteront** les objets construits, ainsi qu'un **constructeur**. Un peu comme la fonction **Constructeur** en JavaScript finalement.

On peut ensuite créer des **instances de classes** dans lesquelles l'ensemble des attributs et des méthodes seront copiés.

Cependant on ne pourra plus supprimer de méthodes ou propriétés dans ce **constructeur**, ni en ajouter de nouvelles. Il faudra alors créer des **classes enfants** auxquelles on pourra ajouter de nouvelles méthodes ou propriétés et dans lesquelles on pourra également choisir desquelles hériter.



## Classes et prototypes

JavaScript fonctionne, lui, selon un système de **prototypes**. La différence est difficile à cerner quand on débute mais vous aurez tout le temps pour vous familiariser avec ces concepts.

En JavaScript, chaque **objet** possède une propriété appelée **prototype** qui va réunir différentes propriétés et méthodes et qui fait office de fonction constructrice de l'objet.

Quand on crée un nouvel objet en utilisant un **constructeur**, les attributs et méthodes ne seront pas copiés dans le nouvel objet lui-même mais seront définies dans cette propriété **prototype**, ce qui établira un lien avec son **constructeur**.

On peut alors ajouter une méthode au **prototype** d'un objet et la rendre ainsi disponible pour tous les objets issus du même **constructeur**.



## Classes et prototypes

```
> function Article (name, price, sizes) {
    this.name = name;
    this.price = price;
    this.sizes = sizes;
}
< undefined
> let chemise = new Article("Vintage Shirt", 39.95, ['M', 'L']);
< undefined
> Article.prototype.soldout = () => "50% on this product !!";
< () => "50% on this product !!""
> chemise.soldout();
< '50% on this product !!'
```



# Classes et prototypes

Catégorie	Basé sur les classes (Java)	Basé sur des prototypes (JavaScript)
<b>Classe et instance</b>	La classe et l'instance sont des entités distinctes.	Tous les objets peuvent hériter d'un autre objet.
<b>Définition</b>	Définir une classe avec une définition de classe ; instancier une classe avec des méthodes de construction.	Définir et créer un ensemble d'objets avec des fonctions de construction.
<b>Création d'un nouvel objet</b>	Créer un objet unique avec l'opérateur <code>new</code> .	Pareil.
<b>Construction de la hiérarchie des objets</b>	Construire une hiérarchie d'objets en utilisant des définitions de classes pour définir des classes enfants à partir de classes existantes.	Construire une hiérarchie d'objets en assignant un objet comme prototype associé à une fonction de construction.
<b>Modèle d'héritage</b>	Hériter des propriétés en suivant la chaîne de classes.	Hériter des propriétés en suivant la chaîne des prototypes.
<b>Extension des propriétés</b>	La définition de la classe spécifie <i>toutes</i> les propriétés de toutes les instances d'une classe. Impossible d'ajouter des propriétés dynamiquement au moment de l'exécution.	La fonction ou le prototype du constructeur spécifie un ensemble <i>initial</i> de propriétés. On peut ajouter ou supprimer dynamiquement des propriétés à des objets individuels ou à l'ensemble des objets.

Credit : MDN



## Les classes en JavaScript

Une des nouveautés amenées par l'**ES6** est la possibilité de créer également des **classes** en JavaScript.

On va donc créer une classe en utilisant le mot-clé **class**, puis y définir un **constructeur** qui servira à initialiser les **propriétés** communes aux futurs nouveaux objets, et également définir les **méthodes** dont ils hériteront.

Pour créer un nouvel objet à partir de ce modèle, c'est à dire créer une instance de cette classe, on va utiliser le mot-clé **new** comme avec notre fonction **Constructeur**.



## Les classes en JavaScript

```
> class Car {
    constructor(marque, modele) {
        this.marque = marque;
        this.modele = modele;
    }
    annonce() {
        return `je suis une ${this.marque} ${this.modele}`;
    }
}

let p406 = new Car('Peugeot', '406');
← undefined
> p406.annonce();
← 'je suis une Peugeot 406'
```



## Les classes en JavaScript

Pour créer une ***classe enfant*** qui héritera des propriétés et méthodes de sa ***classe mère***, il suffit d'utiliser le mot-clé ***extends*** en plus du mot-clé ***class***.

Nous devrons ensuite utiliser le mot-clé ***super()*** à l'intérieur du ***constructeur*** pour que l'héritage du ***constructeur parent*** se fasse correctement. Il reste à ajouter les nouvelles propriétés et méthodes spécifiques à cette nouvelle ***classe***.

Il est donc maintenant possible de créer à partir de cette classe de ***nouvelles instances*** qui hériteront de ses propres caractéristiques, mais également de celles sa classe mère.



## Les classes en JavaScript

```
> class Car {
    constructor(marque, modele) {
        this.marque = marque;
        this.modele = modele;
    }
}

class Suv extends Car {
    constructor(marque, modele, crossover) {
        super(marque, modele);
        this.crossover = crossover;
    }
}

let bz4x = new Suv('Toyota', 'bz4X', true);
← undefined
> bz4x
← ► Suv {marque: 'Toyota', modele: 'bz4X', crossover: true}
```



---

8

# Les objets natifs

Propriétés et méthodes



## Les objets natifs

Nous venons de voir comment créer nous mêmes des *objets* mais il existe également en JavaScript des *objets natifs* ou prédéfinis qui sont donc déjà créés et qui possèdent donc déjà un certain nombre de propriétés et de méthodes.

Ces objets sont à notre disposition et nous pouvons utiliser leurs propriétés et méthodes selon nos besoins. Nous allons en étudier quelques uns comme *String* ou *Array*...

Vous verrez que certains possèdent le même nom que les *types de variables* existant en JavaScript. Ces types représentent en fait des *valeurs primitives* en JavaScript et sont distincts des *objets natifs*.

Au moment d'utiliser une méthode d'un *objet natif* comme *String* sur une variable de *type string*, JavaScript va tout simplement *transformer* cette variable en un objet String le temps de l'utilisation de cette méthode. C'est ce qui peut créer cette confusion entre les deux.



## L'objet String

L'objet **String** possède la propriété **length** qui renvoie la longueur d'une chaîne de caractères. Les espaces contenus dans une chaîne de caractères sont considérés comme des caractères et seront comptabilisés également.

```
> let myString = "test";
< undefined
> myString.length;
< 4
> let otherString = "un test";
< undefined
> otherString.length;
< 7
```



# L'objet String

L'objet **String** possède également beaucoup de méthodes dont nous allons étudier les principales.

La méthode **includes()** permet de vérifier si une chaîne de caractères est présente dans une autre. Elle renverra la réponse au test sous forme de **booléen**. Attention, cette méthode est sensible à la **casse**. Il est également possible de préciser à partir de quel **index** commencer la recherche.

```
> let my_string = "JavaScript";
< undefined
> my_string.includes('script');
< false
> my_string.includes('Script');
< true
```



## L'objet String

Différentes méthodes permettent de *découper* une chaîne de caractères de différentes manières. Ces méthodes se distinguent au niveau des *paramètres* à fournir ainsi que des *résultats* retournés.

Quelles que soient les valeurs retournées avec toutes ces méthodes, la chaîne de caractères d'origine restera toujours inchangée.

Deux méthodes `slice()` et `substring()` permettent d'*extraire* une partie de la chaîne d'origine, entre deux indices donnés (début inclus et fin exclus). Si le deuxième indice n'est pas donné, le découpage s'arrêtera à la fin de la chaîne.



<https://medium.com/difference-between-slice-substring-and-substr-in-javascript-d410a4e0da70>



# L'objet String

```
> let myString = "JavaScript";
< undefined
> myString.substring(0,4);
< 'Java'
> myString.substring(4);
< 'Script'
> myString.substring(-6);    not valid...
< 'JavaScript'
```

```
> myString.slice(0, 4);
< 'Java'
> myString.slice(4);
< 'Script'
> myString.slice(-6);      from end
< 'Script'
```



## L'objet String

La méthode `replace()` renvoie une chaîne de caractères formée à partir de celle d'origine dont certains caractères ont été *remplacés*. Cette méthode est très souvent combinée avec l'utilisation d'*expressions régulières*.

```
> let myString = "We love CSS";
← undefined
> myString.replace("CSS", "JavaScript");
← 'We love JavaScript'
> let string_with_errors = "bla error bla bla bla error bla error error bla bla";
← undefined
> string_with_errors.replace(/ error/g, '');
← 'bla bla bla bla bla bla'
```



## L'objet String

La méthode `split()` renvoie un *tableau* contenant la chaîne de caractères *sectionnée* selon un caractère de séparation donné en indice.

```
> let myString = "We love JavaScript";
< undefined
> myString.split(' ');
< ▶ (3) ['we', 'love', 'JavaScript']
> myString.split('.');
< ▶ (18) ['w', 'e', '.', 'l', 'o', 'v', 'e', '.', 'J', 'a', 'v', 'a', 's', 'c', 'r', 'i', 'p', 't']
```



## L'objet String

La méthode `indexOf()` recherche le ou les caractères donnés en indice et renvoie la *première occurrence* trouvée. Si aucune occurrence n'est trouvée, la méthode renvoie `-1`.

```
> let myString = "We love JavaScript";
← undefined
> myString.indexOf('love');
← 3
> myString.indexOf('css');
← -1
```



# L'objet String

De nombreuses autres méthodes utilisables sur les chaînes caractères existent : pour mettre les caractères en *minuscules* ou en *majuscules* avec `toUpperCase()` et `toLowerCase()`, ou pour supprimer les *espaces* en début et fin de chaîne avec `trim()`.

Il n'est pas nécessaire de citer toutes ces méthodes ici. C'est tout simplement en fonction des besoins que vous aurez que vous prendrez le temps de découvrir chacune d'entre elles.

En prime, un article récapitulant les différents moyens de combiner des chaînes de caractères entre elles :



<https://www.samanthaming.com/tidbits/15-4-ways-to-combine-strings/>



## L'objet Number

La plupart des propriétés et des méthodes de l'objet **Number** sont appelées **statiques** et doivent donc être utilisées depuis l'objet **Number** lui-même.

Les **propriétés** de cet objet sont très spécifiques. Elles servent par exemple à afficher la plus petite entité numérique possible. Nous allons donc plutôt nous concentrer sur certaines de ses **méthodes** qui peuvent s'avérer très utiles.

La méthode **`isNaN()`** permet de vérifier qu'une valeur donnée est un **nombre valide** ou pas. Rappelez vous que la désignation ***Not a Number*** ne sert pas à vérifier si une valeur est bien un nombre mais plutôt à vérifier si elle est un nombre valide, ou encore le résultat d'une opération valide.



<https://www.samanthaming.com/tidbits/61-better-nan-check-with-es6-number-isnan/>



## L'objet Number

```
> let my_value = "string";
< undefined
> Number.isNaN(my_value);
< false
> my_value *= 10;
< NaN
> Number.isNaN(my_value);
< true
```



## L'objet Number

Les méthodes `parseFloat()` et `parseInt()` permettent de respectivement *transformer* une chaîne de caractères en un *nombre* enlevant les chiffres après la *décimale* ou bien en les laissant.

Il faut impérativement que le premier caractère de cette chaîne soit un *chiffre*, autrement le résultat renvoyé sera `NaN`.

L'utilisation de ces deux méthodes peut être bien sûr beaucoup plus complexe que cela, si besoin, en précisant par exemple une *base* à la conversion pour la méthode `parseInt()`.

Voici un article récapitulant les différents moyens de transformer une chaîne de caractères en un nombre :



<https://betterprogramming/-best-way-to-convert-a-string-to-a-number-in-javascript>



## L'objet Number

```
> Number.parseInt("10.5px");
← 10
> Number.parseFloat("10.5px");
← 10.5
> Number.parseFloat("m10");
← NaN
> Number.parseInt("0xF", 16);
← 15
```

```
> // Convert a string into number :
+ "123";
← 123
> "123" * 1;
← 123
> Number("123");
← 123
```



# L'objet Number

La méthode `toFixed()` permet de décider du *nombre* de chiffres après la *décimale*. Si besoin, le dernier chiffre gardé après la décimale sera *arrondi* au chiffre supérieur. Des *zéros* peuvent aussi être ajoutés.

La méthode `toString()` permet de transformer un *nombre* en une *chaîne de caractères*. Une base peut aussi lui être précisée pour réaliser cette conversion.

```
> 10.876.toFixed(1);
< '10.9'
> 10.832.toFixed(2);
< '10.83'
> 10.25.toFixed(4);
< '10.2500'
```

```
> let my_number = 123;
< undefined
> my_number.toString();
< '123'
> my_number.toString(2);
< '1111011'
```



## L'objet Math

Toutes les propriétés et des méthodes de l'objet **Math** doivent être utilisées depuis l'objet **Math** lui-même.

Les propriétés qu'il possède mettent tout simplement à notre disposition des  *constantes mathématiques* comme la *valeur PI* ou d'autres moins connues.

Les méthodes de cet objet permettent d'utiliser de nombreuses *méthodes de calcul* comme le *sinus* ou le *cosinus* d'un angle par exemple.

```
> Math.PI  
< 3.141592653589793  
  
> Math.sin(Math.PI / 2)  
< 1  
  
> Math.cos(Math.PI);  
< -1
```



## L'objet Math

D'autres méthodes bien moins spécifiques existent et nous permettent par exemple d'*arrondir* un nombre décimal comme `Math.round()` qui va arrondir le nombre passé en argument au nombre entier le plus *proche*.

Trois autres méthodes nous permettent d'*arrondir* un nombre au nombre entier *supérieur*, ou *inférieur*...

```
> Math.round(10.54);
< 11
> Math.round(10.48);
< 10
> Math.ceil(10.28);
< 11
> Math.floor(10.28);
< 10
> Math.trunc(10.72);
< 10
```

round : au plus proche

ceil : au-dessus

floor : au-dessous

trunc : sans décimale



## L'objet Math

`Math.random()` va générer un **nombre aléatoire** entre **0** (inclus) et **1** (exlus). Il suffira juste d'adapter l'utilisation de cette méthode pour choisir précisément l'intervalle dans lequel générer ce nombre.

```
> // Agrandir l'intervalle depuis zéro  
    // Math.floor(Math.random() * max)  
  
    Math.floor(Math.random() * 10);  
< 4
```

```
> // Choisir l'intervalle exact [entre 10 et 30]  
    // Math.floor(Math.random() * (max - min)) + min  
  
    Math.floor(Math.random() * (30 - 10)) + 10;  
< 15
```



[https://www.w3schools.com/js/js\\_random.asp](https://www.w3schools.com/js/js_random.asp)



## L'objet Math

Les méthodes `Math.min()` et `Math.max()` vont respectivement permettre d'indiquer le nombre *le plus petit* et le nombre *le plus grand* dans une série de nombres donnée en arguments.

```
> Math.min(10, 15, 50);
< 10
> Math.max(10, 15, 50);
< 50
```



## L'objet Date

Pour manipuler des dates en JavaScript nous devons créer un objet avec le constructeur **Date()**. Nous allons ainsi avoir accès à la **date locale actuelle** de notre système sous forme d'objet.

Sans le mot-clé **new**, la date sera renvoyée sous la forme d'une chaîne de caractères.

De nombreuses méthodes comme **getHour()**, **getDay()** ou **getMonth()** vont ensuite nous permettre de récupérer une partie de la date comme l'heure, le jour, le mois... Mais cela se fera sous un format **numérique** débutant à **zéro**.

**NB:** Techniquement, une date en JavaScript correspond au nombre de millisecondes écoulées depuis le 01/01/1970 (origine temporelle utilisée par de nombreux systèmes informatiques)



## L'objet Date

```
> Date();
< Tue Nov 02 2021 07:58:42 GMT+0100 (heure normale d'Europe centrale)

> new Date();
< Tue Nov 02 2021 07:58:58 GMT+0100 (heure normale d'Europe centrale)

> typeof new Date();
< 'object'

> new Date().getFullYear();
< 2021

> new Date().getMonth();
< 10 [ 10 : novembre ]

> new Date().getDay();
< 2 [ 2 : mardi ]

> new Date('1995-12-17');
< Sun Dec 17 1995 01:00:00 GMT+0100 (heure normale d'Europe centrale)
```



## L'objet Date

Des méthodes comme `toLocaleDateString()` permettent également de convertir ces données sous un format *alphabétique*.

Un premier argument permet de préciser la *langue* dans laquelle convertir ces données. Un deuxième argument donné sous la forme d'un *objet* apportera des précisions sur le type de données à convertir ainsi que sur la manière de les convertir.



[https://developer.mozilla.org/JavaScript/Global\\_Objects/Date/toLocaleDateString](https://developer.mozilla.org/JavaScript/Global_Objects/Date/toLocaleDateString)



## L'objet Date

```
> let options = {
    weekday: 'long',
    year: 'numeric',
    month: 'long',
    day: 'numeric'
};
< undefined
> new Date().toLocaleDateString('fr', options);
< 'mardi 2 novembre 2021'
```



## L'objet Array

Nous avons vu précédemment que les **tableaux** en JavaScript sont avant tout des **objets**. Ils dépendent donc de l'objet global **Array**.

Comme l'objet **String**, il possède la propriété **length** qui va renvoyer la **longueur** d'un tableau, soit le nombre d'éléments qu'il contient

Plusieurs méthodes permettent d'**ajouter** ou **supprimer** des éléments dans un tableau. Avec les méthodes **push()** et **pop()**, les éléments seront ajoutés ou supprimés à **la fin** du tableau.

Avec les méthodes **unshift()** et **shift()**, les éléments seront ajoutés au supprimés **au début** du tableau.



## L'objet Array

```
> let fruits = ["mangue", "kaki", "ananas"];
< undefined
> fruits.push("banane");
< 4    [ nouvelle taille du tableau ]
> fruits;
< ▶ (4) [ 'mangue', 'kaki', 'ananas', 'banane' ]
> fruits.pop();
< 'banane'  [ élément enlevé ]
> fruits;
< ▶ (3) [ 'mangue', 'kaki', 'ananas' ]
```



## L'objet Array

```
> let fruits = ["mangue", "kaki", "ananas"];
← undefined
> fruits.unshift("kiwi");
← 4 [ nouvelle taille du tableau ]
> fruits;
← ▶(4) ['kiwi', 'mangue', 'kaki', 'ananas']
> fruits.shift();
← 'kiwi' [ élément enlevé ]
> fruits;
← ▶(3) ['mangue', 'kaki', 'ananas']
```



## L'objet Array

La méthode ***splice()*** permet également d'***ajouter*** ou de ***supprimer*** des éléments dans un tableau mais à l'***endroit*** qu'on aura choisi.

Cette méthode prend ***trois arguments*** : le premier donne l'***index*** où se placer dans le tableau, le deuxième donne le ***nombre*** d'éléments à ***supprimer***, et le troisième donne le ou les éléments à ***ajouter***.

La méthode retourne un ***tableau*** contenant les éléments qui ont été supprimés du tableau d'origine.



# L'objet Array

```
> let fruits = ["mangue", "kaki", "ananas"];
← undefined
> fruits.splice(1, 0, "banane");
← ▶ []
> fruits;
← ▶ (4) ['mangue', 'banane', 'kaki', 'ananas']
> fruits.splice(1, 2);
← ▶ (2) ['banane', 'kaki']
> fruits;
← ▶ (2) ['mangue', 'ananas']
> fruits.splice(0, 0, "litchi", "datte");
← ▶ []
> fruits;
← ▶ (4) ['litchi', 'datte', 'mangue', 'ananas']
```



## L'objet Array

La méthode `join()` retourne une **chaîne de caractères** composée des différents éléments du tableau, séparés soit par des virgules soit par un séparateur fourni en argument

Certaines méthodes déjà vues avec l'objet `String` sont également disponibles avec l'objet `Array` comme `includes()`, `slice()` ou encore `concat()`.

Concernant la **fusion** de plusieurs tableaux en un seul, une nouvelle fonctionnalité de l'**ES6**, appelée le **spread operator**, permet de faire cela encore plus facilement.



<https://mindsers.blog/fr/post/rest-parameter-et-spread-operator-en-javascript/>



## L'objet Array

```
> let fruits = ["mangue", "kaki", "ananas", 10];
← undefined
> fruits.join();
← 'mangue,kaki,ananas,10'
> fruits.join(' ');
← 'mangue kaki ananas 10'
> fruits.join('-');
← 'mangue-kaki-ananas-10'
```



## L'objet Array

```
> let fruits = ["mangue", "kaki", "ananas", "banane"];
< undefined
> fruits.includes("mangue", 1);
< false
> fruits.slice(1, 3);
< ▶ (2) ['kaki', 'ananas']
> fruits.slice(1);
< ▶ (3) ['kaki', 'ananas', 'banane']
> fruits;
< ▶ (4) ['mangue', 'kaki', 'ananas', 'banane']
> let new_fruits = fruits.slice(1, 3);
< undefined
> new_fruits;
< ▶ (2) ['kaki', 'ananas']
```



# L'objet Array

```
> let fruits = ["mangue", "kaki", "ananas", "banane"];
← undefined
> let legumes = ["aubergine", "courgette", "tomate"];
← undefined
> let epices = ["cannelle", "gingembre", "paprika"];
← undefined
> let aliments = fruits.concat(legumes, epices);
← undefined
> aliments;
← ▶ (10) ['mangue', 'kaki', 'ananas', 'banane', 'aubergine', 'courgette', 'tomate', 'cannelle',
           'gingembre', 'paprika']
> let food = [...fruits, ...legumes, ...epices];
← undefined
> food;
← ▶ (10) ['mangue', 'kaki', 'ananas', 'banane', 'aubergine', 'courgette', 'tomate', 'cannelle',
           'gingembre', 'paprika']
```



## Les fonctions map( ) and co...

La fonction `map()` va nous permettre d'exécuter la *même action* sur *chaque élément* d'un tableau, puis va ensuite retourner un tableau contenant le résultat final. Pour pouvoir réaliser cela, nous allons passer une fonction *callback* en argument de `map()`.

Le but de ce type de fonction étant de *retourner un tableau*, si ce dernier n'a pas un usage particulier de prévu, il sera alors plus logique d'utiliser une simple fonction comme `forEach()`.

```
> let numbers = [1, 2, 3, 4];
< undefined
> numbers.map(index => index * 2);
< ▶ (4) [2, 4, 6, 8]
```



## Les fonctions map( ) and co...

Il existe plusieurs fonctions dérivées de `map()` qui nous permettent de manipuler les données d'un tableau selon certaines conditions pour ensuite en récupérer le résultat.

La fonction `some()` va vérifier qu'*au moins une* des données du tableau remplisse une **condition** donnée et retournera donc un **booléen**.

La fonction `every()` va, quant à elle, s'assurer que *la totalité* des données du tableau remplissent une **condition** donnée et retournera également un **booléen**.

La fonction `sort()` va *trier* les données du tableau par ordre **alphabétique** ou bien selon une fonction **comparative** donnée en argument.

La fonction `filter()` renverra *uniquement* les données du tableau qui auront rempli la **condition** donnée.



## map( )

```
> let fruits = ["mangue", "kaki", "ananas", "banane"];  
      //----- map() -----//  
      // turn every word capitalized  
let big_fruits = fruits.map(index => index.toUpperCase());  
  
big_fruits;  
< ▶ (4) ['MANGUE', 'KAKI', 'ANANAS', 'BANANE']
```



## some( )

```
> let fruits = ["mangue", "kaki", "ananas", "banane"];  
//----- some() -----//  
  
// is some word having letter "k" in it ?  
let k_fruits = fruits.some(index => index.includes('k'));  
  
k_fruits;  
< true  
  
> // is some word having letter "x" in it ?  
let x_fruits = fruits.some(index => index.includes('x'));  
  
x_fruits;  
< false
```



## every( )

```
> let fruits = ["mangue", "kaki", "ananas", "banane"];  
      //----- every() -----//  
  
    // is every word having letter "z" in it ?  
    let z_fruits = fruits.every(index => index.includes('z'));  
  
    z_fruits;  
↳ false  
  
> // is every word having letter "a" in it ?  
    let a_fruits = fruits.every(index => index.includes('a'));  
  
    a_fruits;  
↳ true
```



## filter( )

```
> let fruits = ["mangue", "kaki", "ananas", "banane"];  
    //---- filter() ----//  
    // return only words having letter "s" in it  
let s_fruits = fruits.filter(index => index.includes('s'));  
  
s_fruits;  
↳ ▶ ['ananas']
```



## sort( )

```
> let fruits = ["mangue", "kaki", "ananas", "banane"];
   //---- sort() -----//

   // class in alphabetic orders (need to be lowercase)
let sorted_fruits = fruits.sort();

sorted_fruits;
< ▶ (4) ['ananas', 'banane', 'kaki', 'mangue']

> // class in ascending orders
let numbers = [40, 1, 45, 89, 8, 12, 3, 50];

let ordered_numbers = numbers.sort((a, b) => a - b);

ordered_numbers;
< ▶ (8) [1, 3, 8, 12, 40, 45, 50, 89]
```



<https://betterprogramming.pub/understanding-the-sort-method-of-arrays-a9f2d5f83230>



## reduce( )

La méthode `reduce()` va exécuter une *même action* sur chaque élément itéré d'un tableau. Puis le *résultat* sera stocké dans un *accumulateur* dont la valeur sera actualisée à chaque itération.

```
my_array.reduce(  
    reducer(accumulator, current_value) => {  
        return action result;  
    },  
    starting_accumulator  
)
```

La valeur initiale de l'accumulateur est *facultative*. Si elle n'est pas donnée, elle sera par défaut la *première valeur* du tableau auquel on applique `reduce()` et on commencera donc à itérer à partir de la deuxième.

`reducer()` est une fonction qu'on va écrire et puis passer en *callback* de la fonction principale `reduce()`.



## reduce()

```
[1, 2, 3, 4].reduce( (acc, curr) => acc + curr )
```



1 - Une valeur initiale est assignée à l'accumulateur.

→ *acc = 1 (première valeur du tableau)*

2 - Le tableau est itéré pour que l'instruction du callback soit appliquée sur chacun de ses éléments.

→ *additionner l'accumulateur et la valeur itérée du tableau*

3 - Le résultat de l'instruction est retourné et devient la nouvelle valeur de l'accumulateur pour la prochaine itération.

→ *1ère itération : acc = 1 (valeur de acc) + 2 (deuxième valeur du tableau)*

→ *2ème itération : acc = 3 (nouvelle valeur de acc) + 3 (troisième valeur du tableau) ...*

4 - Lorsque l'itération du tableau est terminée, l'accumulateur est retourné

→ *acc = 10 à la fin (1 + 2 + 3 + 4)*



## reduce()

```
1 var arr = ['a', 'b', 'c', 'd', 'e'];
2
3 function add(x, y) {
4     return x + y;
5 }
6
7 arr.reduce(add)
```

callback function

x : accumulateur  
y : valeur itérée du tableau

reduce-add.js hosted with ❤ by GitHub

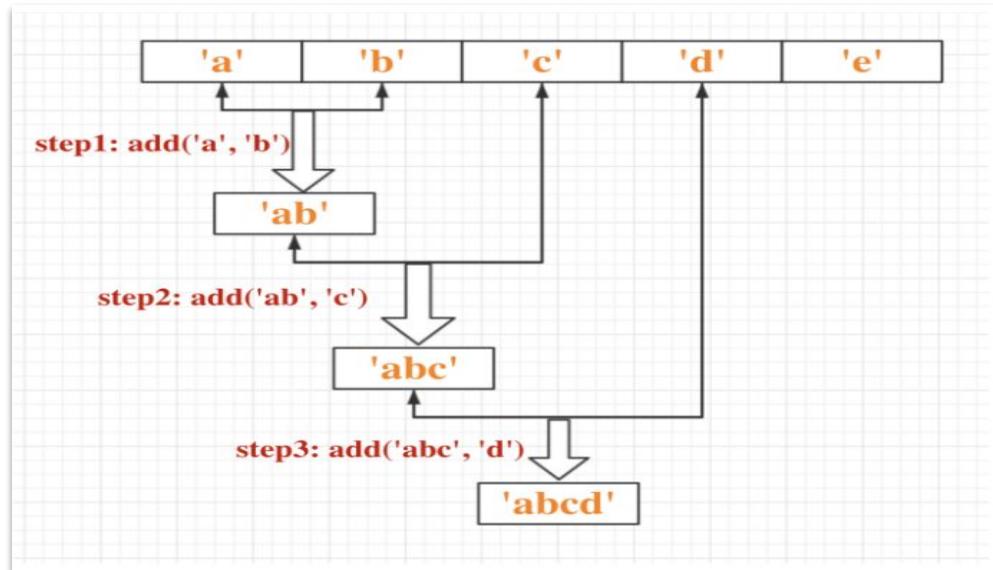
[view raw](#)

Dans cet exemple, `add()` est la fonction qui sert de `reducer()` et qui est ensuite passée en `callback` à la fonction principale `reduce()`.

La valeur initiale de l'`accumulateur x` est par défaut la `première valeur` du tableau. Puis elle deviendra à chaque tour le `résultat` de son addition avec la valeur suivante du tableau, etc...



## reduce( )



Crédit : bytefish - Medium



## reduce( )

Voici quelques articles pour vous aider à mieux cerner la façon dont `reduce()` fonctionne :



<https://medium.com/@rossmawd/the-reduce-js-method-11ec77829e71>



<https://javascript.plainenglish.io/4-practices-to-help-you-understand-array-reduce-f3138cfef095>



<https://andepaulj.medium.com/javascript-reduce-79aab078da23>



<https://enlear.academy/replace-filter-map-sort-whatever-with-array-reduce-2bc3342f474d>

9

# Le DOM

Document Object Model



## C'est quoi le DOM ?

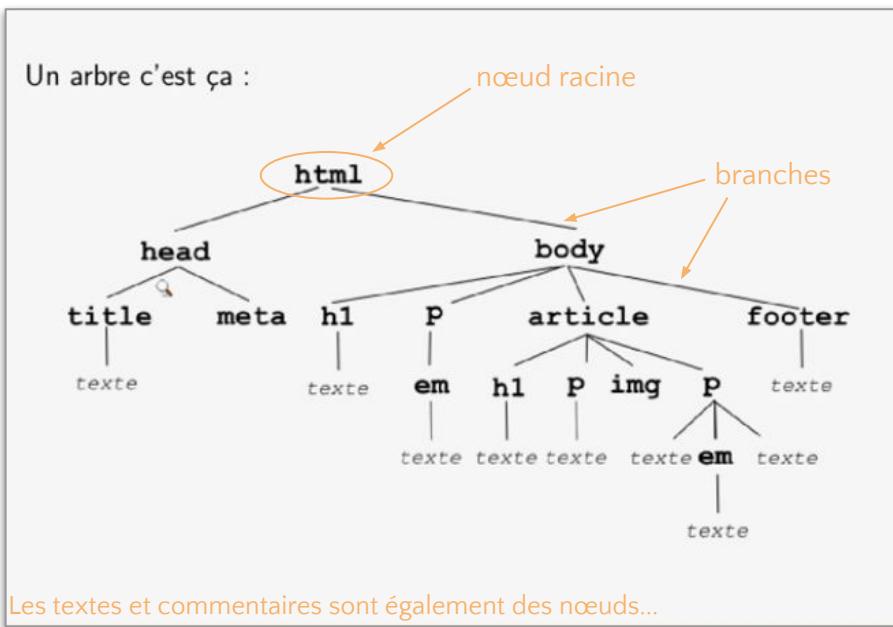
Le **Document Object Model** est une interface de programmation, appelée aussi **API**, qui est fournie par le navigateur et qui représente les documents HTML.

Le **DOM** est donc une représentation de la structure HTML d'une page web et c'est à travers cette représentation que JavaScript va pouvoir interagir avec nos éléments HTML.

Un document HTML est représenté comme un **arbre**, une arborescence, composé de **noeuds** (objets) représentant les différents éléments HTML de la page.



# C'est quoi le DOM ?





## Les interfaces du DOM

Le **DOM** possède lui-même de multiples interfaces qui vont nous permettre de manipuler tous les éléments qui composent une page web.

Parmi ces interfaces : l'objet **Window** qui représente la fenêtre d'un navigateur et qui est à la base de nombreux autres objets très utiles comme **Navigator**, **GeoLocation**, **Screen** ou encore **Document** que nous allons étudier de plus près.

L'objet **Window** possède de nombreuses propriétés et méthodes. Les plus connues sont les suivantes :

*`prompt()` `alert()` `confirm()`*



## Les interfaces du DOM

Les interfaces ***Event*** et ***EventTarget*** vont nous permettre de gérer tout ce qui concerne les évènements ayant lieu dans le ***DOM*** comme les clics d'une souris, les touches appuyées d'un clavier...

L'interface ***Document*** se charge de représenter le document HTML et va servir de point d'entrée dans l'arborescence du ***DOM*** et nous permet donc d'accéder au contenu de la page web.

Les interfaces ***Node*** et ***Element*** vont représenter chaque nœud ou objet qui composent un document HTML et mettre à notre disposition de nombreuses méthodes pour interagir avec eux.



## Accéder au DOM

Avec l'interface **DOM**, nous allons pouvoir manipuler les différents éléments HTML d'une page en utilisant les propriétés et méthodes de l'objet **Document**. Parmi les propriétés disponibles :

Les propriétés **body** et **head** vont respectivement retourner les nœuds représentant les éléments **<body>** et **<head>** de notre page HTML.

```
> document.body;
<‐ ▼<body>
    <h1 class="home">Titre principal</h1>
    <p id="para1">Un premier paragraphe</p>
    ▶<div>...</div>
    <p>Un dernier paragraphe</p>
</body>
```

```
> document.head;
<‐ ▼<head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
    </head>
```



## Accéder au DOM

Les propriétés *title* et *url* vont respectivement renvoyer le contenu de l'élément `<title>` de la page web et l'*URL* de la page. La propriété *links* va retourner la liste de tous les éléments `<a>` et `<area>` présents dans le document. Il existe encore beaucoup d'autres propriétés que nous n'allons pas évoquer ici.

```
> document.title;
< 'Cours JavaScript'
> document.URL;
< 'http://127.0.0.1:5500/dom.html'
```



## Accéder au DOM

Parmi les nombreuses méthodes de l'objet ***Document*** pour accéder aux éléments :

**`querySelector()`** et **`querySelectorAll()`** dans lesquelles il suffit d'utiliser les **sélecteurs CSS** pour cibler les éléments recherchés.

**`querySelector()`** va renvoyer le premier élément de la page correspondant à la requête, ou bien va renvoyer **`null`** si aucun élément correspondant à la requête n'a été trouvé.

**`querySelectorAll()`** va renvoyer une liste de tous les ***nœuds*** correspondant à la requête. Elle renverra une liste vide si aucun élément correspondant à la requête n'est présent dans la page.



## Accéder au DOM

```
<body>

    <h1 class='home'>Titre principal</h1>
    <p id='para1'>Un premier paragraphe</p>
    <div>
        <p>Un paragraphe dans une div</p>
        <p class='home'>Un autre paragraphe dans une div</p>
    </div>
    <p>Un dernier paragraphe</p>

<script src="dom.js"></script>
</body>
```



```
> document.querySelector('p');
<-- <p id="para1">Un premier paragraphe</p>
> document.querySelector('a');
<-- null
> document.querySelectorAll('.home');
<-- ► NodeList(2) [h1.home, p.home]
```



## Accéder au DOM

Les deux méthodes `querySelector()` et `querySelectorAll()` peuvent être utilisées de deux manières différentes et pourront alors donner des résultats différents.

Utilisées depuis l'objet ***Document***, les recherches se feront dans toute la page. Mais utilisées depuis l'objet ***Element*** (qui représente un élément choisi dans la page), les recherches ne se feront alors qu'à l'intérieur de cet élément.

```
> document.querySelectorAll('.home');
<▶ NodeList(2) [h1.home, p.home]
> let div = document.querySelector('div');
<undefined
> div.querySelectorAll('.home');
<▶ NodeList [p.home]
```



## Accéder au DOM

La méthode `getElementById()` renverra l'unique élément correspondant à l'*id* donné, à l'intérieur du document.

La méthode `getElementsByClassName()` renverra une liste de tous les éléments HTML qui possèdent cette classe.

La méthode `getElementsByTagName()` renverra une liste de tous les éléments HTML correspondant au nom de l'élément indiqué.

Toutes ces méthodes peuvent aussi être utilisées depuis l'objet *Document*, ou bien depuis l'objet *Element* qui aura été choisi.



## Accéder au DOM

```
> document.getElementById('para1');
<  <p id="para1">Un premier paragraphe</p>
> document.getElementsByClassName('home');
<  ▷ HTMLCollection(2) [h1.home, p.home]
> document.getElementsByTagName('p');
<  ▷ HTMLCollection(5) [p#para1, p, p.home, p, p, para1: p#para1]
```



## NodeList / HTML Collection

Bien que ciblant les mêmes éléments `<p>` de notre document, les méthodes `getElementsbyTagName('p')` et `querySelectorAll('p')` nous donnent des résultats sensiblement différents.

La première méthode nous renvoie une **HTML Collection** qualifiée de "*live*", c'est-à-dire qu'elle est mise à jour dynamiquement, en fonction des changements qui sont effectués dans le document.

Par contre, la deuxième méthode nous renvoie une **NodeList** qualifiée de "*statique*" qui, elle, ne tiendra pas compte de ces changements.

```
> document.getElementsByTagName('p');
<- ▶ HTMLCollection(5) [p#para1, p, p.home, p, p, para1: p#para1]
> document.querySelectorAll('p');
<- ▶ NodeList(5) [p#para1, p, p.home, p, p]
```



## NodeList / HTML Collection

Ces deux listes ressemblent à des *tableaux* mais n'en sont pas vraiment. C'est au moment de vouloir utiliser certaines *méthodes* spécifiques aux tableaux que la console vous le rappelle gentiment :

✖ ➔ **Uncaught TypeError: p\_nodeList.map is not a function  
at nodlist.html:31**

On peut utiliser sans problème des boucles comme *for* ou *for...of*, exception faite pour la boucle *forEach* qui est spécifique aux tableaux (mais elle fonctionnera quand même avec une *nodeList* 😊 ...)

Il existe cependant une solution très simple pour contourner ce problème : transformer nos *nodeList* et *HTMLCollection* en de vrais tableaux :

*Array.from(p\_nodeList)* ou *[...p\_nodeList]*



## Parcourir le DOM

Plusieurs propriétés de Document permettent de cibler des *nœuds* de la page HTML par rapport à un nœud donné. Les résultats prendront en compte tous les nœuds, y compris les noeuds *text* et les noeuds *commentaires*. Il existe donc pour chacune de ces méthodes une méthode correspondante qui ne prend en compte que les *éléments*.

`childNode` renvoie la liste des *nœuds* enfants et `children` renvoie uniquement la liste des *éléments* enfants.

`parentNode` renvoie le *nœud* parent et `parentElement` renvoie l'*élément* parent.

`firstChild` et `lastChild` renvoient respectivement le premier *noeud* enfant et le dernier *noeud* enfant.

`firstElementChild` et `lastElementChild` renvoient le premier *élément* enfant et le dernier *élément* enfant.

`previousSibling` et `nextSibling` renvoient le *nœud* précédent et le *nœud* suivant (situés au même niveau).

`previousElementSibling` et `nextElementSibling` renverront l'*élément* précédent et l'*élément* suivant.



## Modifier le DOM : le texte

Plusieurs propriétés nous permettent de récupérer et même de modifier le contenu d'un élément.

Avec `innerHTML`, c'est tout le **contenu HTML** de l'élément qui est récupéré (y compris les balises éventuelles qui s'y trouveront), tandis qu'avec `outerHTML`, c'est la **syntaxe HTML** complète de l'élément qui est accessible.

La propriété `innerText` nous propose le "*contenu textuel « visuellement rendu » de l'élément*" (dixit MDN). Alors que la propriété `textContent` va nous donner "*le contenu textuel d'un nœud et de ses descendants*" (toujours dixit MDN)



<https://betterprogramming.pub/whats-best-innertext-vs-innerhtml-vs-textcontent-903ebc43a3fc>



## Modifier le DOM : le texte

```
<h2>Les subtilités cachées du <a href="#">DOM</a></h2>
```



```
> document.querySelector('h2').innerHTML;
< 'Les subtilités cachées du <a href="#">DOM</a>'

> document.querySelector('h2').outerHTML;
< '<h2>Les subtilités cachées du <a href="#">DOM</a></h2>'

> document.querySelector('h2').innerText;
< 'Les subtilités cachées du DOM'

> document.querySelector('h2').textContent;
< 'Les subtilités cachées du DOM'
```



## Modifier le DOM : ajouter

Pour ajouter un nouvel élément au *DOM*, il faut commencer par créer l'élément souhaité en utilisant la méthode `document.createElement()` puis ajouter du texte, si besoin, avec la propriété `textContent`.

Plusieurs méthodes sont disponibles pour ajouter l'élément au *DOM* : les méthodes `prepend()` et `append()` permettent de le positionner par rapport à un *nœud* choisi : soit *avant* son premier enfant, soit *après* son dernier enfant. Avec ces deux méthodes, il est possible d'ajouter plusieurs éléments à la fois.

La méthode `appendChild()` va positionner l'élément en tant que *dernier enfant* d'un élément choisi. Cette méthode ne permet d'ajouter qu'un seul nœud à la fois.



[https://dev.to/ibn\\_abubakre/append-vs-appendchild-a4m](https://dev.to/ibn_abubakre/append-vs-appendchild-a4m)



## Modifier le DOM : ajouter

```
// ajout d'une div
let div = document.createElement('div');
document.querySelector('body').append(div);

// création d'un paragraphe
let para1 = document.createElement('p');
para1.textContent = 'texte créé avec la propriété "textContent"';

// création d'un 2eme paragraphe
let para2 = document.createElement('p');
para2.append('texte créé avec la méthode "append()"');

// ajout des deux paragraphes dans la div
document.querySelector('div').append(para1, para2);

// création et ajout d'une 2eme div
document.querySelector('body').appendChild(document.createElement('div'));
```



## Modifier le DOM : ajouter

La méthode `insertBefore()` va ajouter un élément enfant *juste avant* un autre élément enfant :

```
noeud_parent.insertBefore( new_node, before_who )
```

La méthode `insertAdjacentElement()` va positionner un élément *par rapport* à un autre selon des *mots-clés* (quatre possibilités) précisés en argument :

```
noeud_cible.insertAdjacentElement( mot-clé, new_node )
```

Les méthodes `insertAdjacentText()` et `insertAdjacentHTML()` fonctionnent comme la méthode précédente mais pour insérer respectivement une *chaîne de caractères* ou bien une *syntaxe HTML*.

Ajoutées plus récemment, les méthodes `before()` et `after()` permettent d'ajouter un élément *avant* ou *après* un autre élément.



# Modifier le DOM : ajouter

```
// insertBefore() : place le paragraphe avant le <a>, dans le <main>
document.querySelector('main').insertBefore(para1, document.querySelector('a'));

// --- insertAdjacentElement() --- //

// place le paragraphe juste avant le <a>
document.querySelector('a').insertAdjacentElement("beforebegin", para2);

// place le paragraphe juste après le <a>
document.querySelector('a').insertAdjacentElement("afterend", para3);

// place le paragraphe en 1er enfant du <main> [après qu'il s'ouvre]
document.querySelector('main').insertAdjacentElement("afterbegin", paraA);

// place le paragraphe en dernier enfant du <main> [avant qu'il se ferme]
document.querySelector('main').insertAdjacentElement("beforeend", paraB);

// before() et after() : ajoute l'élément avant ou après le <a>
document.querySelector('a').before(h2);
document.querySelector('a').after(h3);
```



## Modifier le DOM : supprimer

La méthode `replaceChild()` permet de remplacer dans le DOM le nœud enfant d'un parent par un autre nœud enfant.

```
parentNode.replaceChild(new_child, old_child);
```

La méthode `removeChild()` permet de simplement supprimer le noeud enfant d'un parent dans le DOM.

```
parentNode.removeChild(child_to_remove);
```

La méthode `remove()` permet de supprimer un élément de l'arborescence du **DOM**, sans préciser de parent.

```
node_to_remove.remove();
```



## Modifier le DOM : supprimer

```
// --- replaceChild() --- //

// find <h2>
let old_title = document.querySelector('h2');

// create <h1>
let new_title = document.createElement('h1');
new_title.textContent = "Nouveau titre h1";

// replace <h2> by <h1>
document.querySelector('main').replaceChild(new_title, old_title);

// --- remove an element --- //

// removeChild()
let para1 = document.querySelectorAll('p')[0];
document.querySelector('main').removeChild(para1);

// remove()
let para2 = document.querySelectorAll('p')[1];
para2.remove();
```



## Modifier le DOM : les attributs

La méthode `hasAttribute()` permet de s'assurer de la présence d'un *attribut* dans un élément. La valeur renournée sera un *booléen*.

La propriété `attributes` permet d'accéder à la liste des *attributs* avec leur *valeur*, pour un élément donné. Tandis que la propriété `getAttributeNames()` va renvoyer une simple liste des *noms* de ses attributs.

La méthode `getAttribute()` va renvoyer la *valeur* de l'attribut donné et la méthode `setAttribute()` permettra de *changer* la valeur d'un attribut, ou de la créer.

La méthode `removeAttribute()` permet de *supprimer* un attribut. La méthode `toggleAttribute()` permet d'*inverser* la situation d'un attribut dont la valeur est un *booléen*.

Les propriétés `className` et `id` vont respectivement renvoyer la liste des *classes* possédées par un élément et le nom de son *id*.



## Modifier le DOM : les attributs

```
> let img_att = document.querySelector('img').attributes;
< undefined
> for (let att of img_att) { console.log(att) };
    src="https://picsum.photos/id/102/500/300"
    alt="some raspberries"
    title="some raspberries"
    class="img-accueil pink-border"
    id="img-master"
< undefined
> document.querySelector('img').className;
< 'img-accueil pink-border'
> document.querySelector('img').id;
< 'img-master'
```



## Modifier le DOM : le style

La propriété **style** de l'objet **Element** permet d'assigner, *inline*, des déclarations de style à un élément du **DOM**. Cette déclaration ayant le degré maximum de **spécificité**, elle écrasera toutes les autres déclarations visant les mêmes **propriétés CSS** de l'élément.

Les déclarations faites via la propriété **style** d'un élément utilisent l'ensemble des propriétés CSS, écrites sous la forme **camelCase**.

```
document.querySelector('h1').style.backgroundColor = "firebrick"
```

Pour pouvoir accéder à l'ensemble des propriétés CSS d'un élément, il sera préférable d'utiliser la méthode **getComputedStyle()** de l'objet **Window** qui va renvoyer un objet **CSSStyleDeclaration** regroupant toutes les propriétés CSS d'un élément.

Tandis que la propriété **style** de l'objet **Element** renverra le même objet mais il ne permettra d'accéder qu'aux déclarations CSS qui auront été effectuées *inline*.



## Modifier le DOM : le style

```
> let para = document.querySelector('p');
← undefined
> // un background a été déclaré dans fichier CSS
  para.style.backgroundColor;
← '' 😳👀
> getComputedStyle(para).backgroundColor;
← 'rgb(0, 255, 255)' 😊
> // déclaration inline via la propriété "style" :
  para.style.backgroundColor = "rgb(0, 40, 40)";
← 'rgb(0, 40, 40)'
> para.style.backgroundColor; ↗
← 'rgb(0, 40, 40)' ↘
> getComputedStyle(para).backgroundColor;
← 'rgb(0, 40, 40)'
```



## Gérer les événements

En programmation, il est possible de *surveiller* les **événements** se produisant dans l'environnement d'une *page web* comme l'utilisation de la *souris* ou le redimensionnement d'une *fenêtre*, afin de pouvoir y réagir.

Il existe trois façons de gérer les **événements** en JavaScript :

- Directement à l'*intérieur* d'un élément HTML, via un *attribut* spécifique à l'événement choisi (cette pratique est aujourd'hui *déconseillée*)
- Depuis un élément, à l'aide de *propriétés* JavaScript spécifiques à l'objet **Event**
- Depuis un élément, en utilisant la méthode *addEventListener()* (la manière la plus *performante*)



## Gérer les événements

```
<body>

    <!-- Avec l'attribut HTML onclick : ❌ -->
    <button onclick="alert('j'ai été cliqué')>Je réagis au click</button>

    <script>

        // Avec la propriété JS onclick ✓
        document.querySelector('button').onclick = () => alert('j'ai été cliqué');

        // Avec la méthode addEventListener() ✓ 🚀
        document.querySelector('button').addEventListener('click', () => alert('j'ai été cliqué'));

    </script>
</body>
```



## Gérer les événements

La méthode `addEventListener()` est utilisée depuis un élément. Elle fonctionne avec deux *arguments* principaux : le *type d'événement* à écouter, puis une fonction *callback* à exécuter une fois que l'événement se sera produit

Il est possible de *désactiver* cette méthode, en utilisant cette fois-ci la méthode `removeEventListener()`. Ainsi, la fonction callback ne sera plus exécutée si l'événement écouté venait à se reproduire.

Le *callback* à désactiver devant être précisé en *argument* de la fonction `removeEventListener()`, il est alors déconseillé de passer une fonction *anonyme* en tant que *callback*.



## Gérer les événements

```
// Désactiver l'écoute avec removeEventListener()

// création du callback à passer en argument de l'eventListener
function click_alert() {
    alert('j\'ai été cliqué');
    // removeEventListener() avec les mêmes arguments que addEventListener()
    document.querySelector('button').removeEventListener('click', click_alert);
};

// addEventListener() qui contient donc aussi le removeEventListener() 😊
document.querySelector('button').addEventListener('click', click_alert);
```



## Gérer les événements

Lorsque la méthode `addEventListener()` est activée, elle donne accès à l'objet ***event*** qui, avec sa propriété ***target***, renvoie à l'élément ***cible*** de l'eventListener.

On pourra alors facilement agir sur cet élément en réponse à l'événement produit. Il faut pour cela que cet objet ***event*** soit placé en ***argument*** de la fonction ***callback*** :

```
element-cible.addEventListener( "click", callback(event) );
```

Ainsi, à l'intérieur du ***callback***, on pourra directement accéder aux propriétés de ***style*** de l'élément ***cible***, à son ***contenu*** ou encore à son ***environnement*** HTML :

```
event.target.style || event.target.textContent || event.target.parentNode ...
```



## Gérer les événements

```
// function to change button background depending on current background
const toggle_color = (event) => {
    if (event.target.style.backgroundColor === 'firebrick') {
        event.target.style.backgroundColor = "white"
    }
    else {
        event.target.style.backgroundColor = "firebrick"
    }
}

// put eventlistener on each button to toggle background on clicked one
let buttons = document.querySelectorAll("button");
for (let button of buttons){
    button.addEventListener('click', toggle_color)
}
```



## Capturing & Bubbling

Lorsque des **événements** sont déclenchés dans le **DOM**, ils suivent deux phases de **propagation** : une phase-aller et une phase-retour.

La **première** phase est appelée **capturing** : l'événement se propage depuis l'élément `<html>` et parcourt sa descendance jusqu'à l'élément le plus **éloigné** de lui contenant un **eventlistener**.

La **deuxième** phase est appelée **bubbling** : suivant le chemin inverse, l'événement se propage depuis l'élément contenant un **eventlistener** jusqu'à son ancêtre le plus lointain : l'élément `<html>`.

Les **réponses** aux événements se produisent par défaut lors de la phase de **bubbling**. Il est cependant possible d'**inverser** ce comportement en passant un 3ème **argument** à la méthode `addEventListener()`. Cet argument est un **booléen** dont la valeur par défaut est **false**.



# Capturing & Bubbling

```
<div>
  <button>CLICK ME</button>
</div>

<script>
  document.querySelector('div').addEventListener('click', () => console.log("<div> was clicked !"), true);
  document.querySelector('button').addEventListener('click', () => console.log("<button> was clicked !"));
</script>
```



```
> // comportement par défaut :
← undefined
<button> was clicked !
<div> was clicked !
```

```
> // en ajoutant true à addEventListener() :
← undefined
<div> was clicked !
<button> was clicked !
```



## Capturing & Bubbling

Il est également possible d'*annuler* tout simplement la *propagation* de l'événement avec la méthode `stopPropagation()`.

Cette méthode appartient à l'objet `Event` auquel on a accès lorsqu'un `eventListener` est activé. Selon la *phase* de déclenchement choisie, la *propagation* s'arrêtera juste après le déclenchement du premier événement capté lors de cette phase.

La méthode `preventDefault()` permet d'annuler le *comportement* par défaut associé à un *événement*, comme l'envoi d'un formulaire lors du click sur le bouton `submit`. Mais cette méthode n'annulera pas la réponse à l'événement telle qu'elle a été programmée.

En effet, l'action en *réponse* à un événement (gérée par un `eventlistener`) et l'action *par défaut* qui découle de cet événement sont deux actions *indépendantes* l'une de l'autre.



<https://www.sitepoint.com/event-bubbling-javascript/>



# Capturing & Bubbling

```
<section>
  <div>
    <button>CLICK ME</button>
  </div>
</section>

<script>
  document.querySelector('section').addEventListener
  ('click', () => console.log("<section> was clicked !"));

  document.querySelector('div').addEventListener('click',
  () => console.log("<div> was clicked !"));

  document.querySelector('button').addEventListener
  ('click', (event) => {
    console.log("<button> was clicked !");
    event.stopPropagation();
  });
</script>
```

```
> // phase bubbling choisie :
← undefined
<button> was clicked !
```

```
> // phase capturing
← undefined
<section> was clicked !
<div> was clicked !
<button> was clicked !
```