



# JAVASCRIPT



UNE INITIATION À JAVASCRIPT



# SOMMAIRE 1/2



Introduction

Bibliothèques et Frameworks

Interface

Mon premier script

Syntaxe de base

Les opérateurs

Les objets

Les tableaux



# SOMMAIRE 2/2

+  
•

Les conditions  
Incrémentation et décrémentation  
Les boucles  
Les fonctions  
Manipulation du DOM  
Mini projet : Shifumi  
Les formulaires  
Les classes  
Les promesses  
Les fonctions asynchrones

○





# INTRODUCTION

Définition, historique, caractéristiques



- Un langage **populaire**
- **Dynamisme** et **interaction** des pages web
- Un langage **côté client**
- **Exécuté** dans le **navigateur du client**







# Historique :

- Créé en **1995** par **Brendan Eich**
- Standardisé en **1997** (ECMAScript)
- Évolution de **langage de script** à **langage de programmation** polyvalent





# ECMAScript :

- **Norme** sur laquelle repose **JavaScript**
- **Les versions ES5 (2009) et ES6 (2015)** ont apportées **des mises à jour majeures** du langage.
- Depuis ES6, **ECMAScript** suit un **cycle de mise à jour annuel** avec de nouvelles fonctionnalités ajoutées chaque année. Les versions plus récentes, telles que ECMAScript 2020 (ES11), 2021 (ES12), et ainsi de suite, apportent **des améliorations continues** au langage pour répondre aux besoins des développeurs et pour **améliorer la productivité**.

# Caractéristiques principales :

J A V A S C R I P T

- Langage **interprété**
- **Typage** faible
- Langage **orienté objet**
- Langage **fonctionnel**
- Manipulation du **DOM**





# Utilisations courantes



- **Interactivité** des pages web : ajouter de formulaires, carrousels, boutons, etc.
- **Validation de formulaires** : validation avant soumission
- **Animations** : animations fluides et des transitions d'éléments
- **Développement de jeux** : jeux simples dans le navigateur.
- **Applications web** : grâce aux **frameworks** tels **React**, **Angular** et **Vue.js**



# BIBLIOTHÈQUES ET FRAMEWORKS

Présentation rapide de quelques bibliothèques et frameworks connus

# Les bibliothèques :

Une **bibliothèque** JavaScript (également appelée "librairie JavaScript") est un **ensemble de fonctions, d'outils et de routines préécrites et préstructurées** qui visent à faciliter le **développement d'applications web**.

*Définition ChatGPT*





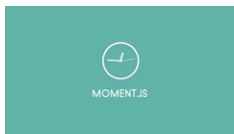
# Les bibliothèques :



- **jQuery** (DOM, animations, requêtes Ajax)



- **Lodash** (collection de fonctions : manipuler les tableaux, objets, etc)



- **Moment.js** (manipulation et formatage des dates et heures)



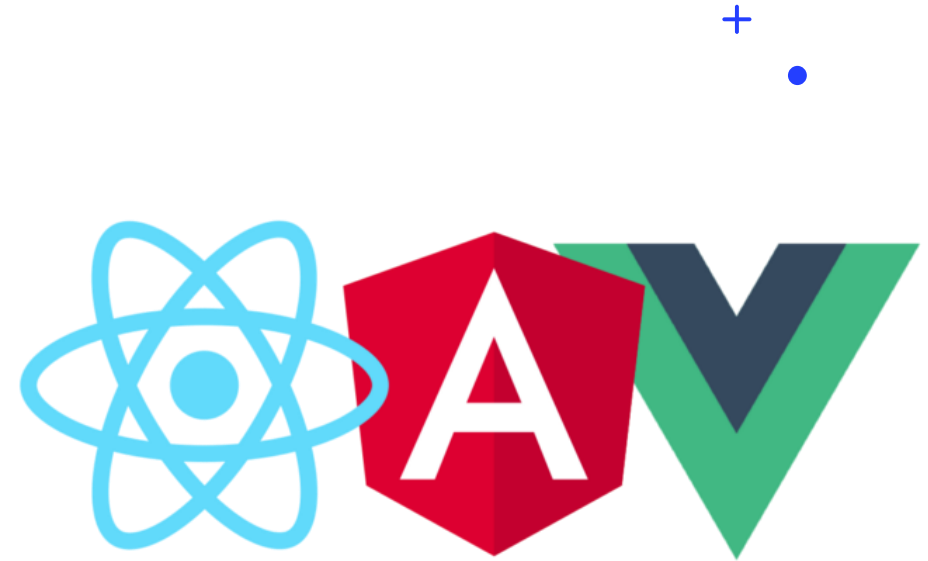
- **Axios** (gestion des requêtes HTTP avec prise en charge des promesses)



# Les frameworks :

Ensemble structuré et préconçu de bibliothèques, d'outils, de conventions et de modèles de conception qui facilitent le **développement d'applications web** complexes et interactives..

*Définition ChatGPT*







# Les frameworks :



- **React** (développé par **Facebook**, approche par composants)
- **Angular** (développé par **Google**, composants et design pattern **MVC**)
- **Vue.js** (partage des similitudes avec les 2 premiers, plus accessible et léger)
- **Ember.js** (modèle de conception **COC**)
- **Backbone.js** (framework minimaliste pour développer des applications côté client)
- **Meteor.js** (framework **fullstack**, gestion des données avec **MongoDB**)



# INTERFACE

Navigateurs, éditeurs de texte, éditeurs de code, IDE





# Quelle interface choisir ?

- **Navigateurs** : Vous pouvez écrire et exécuter des scripts JavaScript directement dans la console de développement de votre navigateur web.
- **Éditeurs de texte** : Vous pouvez utiliser un éditeur de texte simple tel que **Notepad** (sur Windows) ou **TextEdit** (sur macOS) pour écrire du code JavaScript. Enregistrez le fichier avec une extension .js, puis ouvrez-le dans un navigateur pour exécuter le script.
- **Éditeurs de code léger** : **Visual Studio Code**, **Sublime Text** et Atom sont largement utilisés. Ils offrent des fonctionnalités utiles telles que la coloration syntaxique, la complétion automatique et l'intégration de la console de développement.
- **Environnement de développement intégré (IDE)** : Des IDE tels que **WebStorm**, **Visual Studio** (avec des extensions web) et **Eclipse** (avec des plugins web) offrent des fonctionnalités avancées pour le développement JavaScript.



# Google

Effectuez une recherche sur Google ou saisissez une URL

Gmail Images



DevTools is now available in French! [Always match Chrome's language](#) [Switch DevTools to French](#)

Elements Console Sources Network Performance Memory Application

top Filter

```
> let maVariable = "Coucou tout le monde"
< undefined
> console.log(maVariable)
  Coucou tout le monde
< undefined
>
```



# MON PREMIER SCRIPT

Hello world !





Il y a **trois manières courantes** d'insérer des scripts dans une page web :

### ❶ Balise `<script>` dans le `<head>` :



Permet de charger et d'exécuter le script avant que le reste de la page ne soit rendu. Cependant, cela peut potentiellement ralentir le chargement de la page si le script est long à s'exécuter.

```
<!DOCTYPE html>
<html>
<head>
  <title>Ma Page Web</title>
  <script src="chemin/vers/votre/script.js"></script>
</head>
<body>
  <!-- Le reste du contenu de votre page -->
</body>
</html>
```



## ② Balise `<script>` à la fin du `<body>` :

Placement de la balise `<script>` juste avant la fermeture de la balise `</body>`. Cela permet au reste de la page de se charger avant d'exécuter le script, ce qui peut améliorer les performances de chargement.

```
<!DOCTYPE html>
<html>
<head>
  <title>Ma Page Web</title>
</head>
<body>
  <!-- Le reste du contenu de votre page -->
  <script src="chemin/vers/votre/script.js"></script>
</body>
</html>
```



### ③ Événement DOMContentLoaded :

Le script est exécuté une fois que le **DOM** (Document Object Model) de la page est complètement chargé. Cela permet de contrôler précisément le moment où votre script est exécuté, sans retarder le chargement de la page.

```
<!DOCTYPE html>
<html>
<head>
  <title>Ma Page Web</title>
  <script>
    document.addEventListener("DOMContentLoaded", function() {
      // Votre code JavaScript ici
    });
  </script>
</head>
<body>
  <!-- Le reste du contenu de votre page -->
</body>
</html>
```

# À VOUS DE JOUER !



✓ Réalisez un premier script pour afficher « **Hello World** » dans la **console** puis un second script pour l’afficher dans une page web.

**BONNES PRATIQUES :**

`element.textContent` ou `element.innerHTML` ?



# SYNTAXE DE BASE

Let playerLevel = "beginner"





# Les variables :

Une **variable** est un conteneur qui stocke la donnée temporairement au sein de votre code. Cela vous permet d'enregistrer des données.

En *JavaScript*, vous pouvez déclarer des variables en utilisant les mots-clés **var** (dépréciée), **let** ou **const** (pour des constantes).

```
let maVariable = "Valeur de ma variable";  
maVariable = "Nouvelle valeur";
```





# Types de données :



JavaScript est un langage à **typage dynamique**, ce qui signifie que les types de données sont déterminés automatiquement. Voici les types de données de **base** en JavaScript :

- Nombre (**Number**) : entiers et nombres à virgule flottante.
- Chaîne de caractères (**String**) : texte encadré par des guillemets simples ou doubles.
- Booléen (**Boolean**) : vrai (**true**) ou faux (**false**).
- Tableau (**Array**) : une liste ordonnée d'éléments, déclarée entre crochets [].
- Objet (**Object**) : une collection de propriétés (clés et valeurs), déclarée entre accolades {}.



## 📖 Bon à savoir :

- **Null** : Représente l'absence intentionnelle de valeur. Une variable avec la valeur null est **intentionnellement vide**. +
- **Indéfini (Undefined)** : Représente une variable qui a été **déclarée** mais **n'a pas encore été initialisée** avec une **valeur**. ●

Il existe d'autres types de variables plus « complexes » comme les **Objets**, les **Tableaux** et les **Fonctions**.



# Commentaires :



Les **commentaires** permettent d'ajouter des notes dans le code pour expliquer ce qu'il fait.

```
// Déclaration d'une variable
let age = 25;

// Calcul de la somme
let total = 10 + 5;

/*
  Fonction pour calculer le carré d'un nombre
  @param {number} x - Le nombre dont on veut calculer le carré
  @returns {number} Le carré du nombre
*/
function calculerCarre(x) {
  return x * x;
}
```



# Types de données :



JavaScript est un langage à **typage dynamique**, ce qui signifie que les types de données sont déterminés automatiquement. Voici les types de données de **base** en JavaScript :

- Nombre (**Number**) : entiers et nombres à virgule flottante.
- Chaîne de caractères (**String**) : texte encadré par des guillemets simples ou doubles.
- Booléen (**Boolean**) : vrai (**true**) ou faux (**false**).
- Tableau (**Array**) : une liste ordonnée d'éléments, déclarée entre crochets [].
- Objet (**Object**) : une collection de propriétés (clés et valeurs), déclarée entre accolades {}.





# LES OPÉRATEURS

Let playerLevel = "beginner"



# Les opérateurs :

## Opérateurs arithmétiques :

- `+` : Addition
- `-` : Soustraction
- `*` : Multiplication
- `/` : Division
- `%` : Modulo (reste de la division)
- `**` : Exponentiation (ES6)

## Opérateurs d'assignation :

- `=` : Affectation simple
- `+=` : Addition et affectation
- `-=` : Soustraction et affectation
- `*=` : Multiplication et affectation
- `/=` : Division et affectation
- `%=` : Modulo et affectation
- `**=` : Exponentiation et affectation (ES6)

## Opérateurs de comparaison :

- `==` : Égal à (valeur)
- `===` : Égal à (valeur et type)
- `!=` : Différent de (valeur)
- `!==` : Différent de (valeur et type)
- `>` : Supérieur à
- `<` : Inférieur à
- `>=` : Supérieur ou égal à
- `<=` : Inférieur ou égal à

## Opérateurs logiques :

- `&&` : ET logique
- `||` : OU logique
- `!` : NON logique



# Les opérateurs :

+

•

○

## Opérateurs de concaténation de chaînes :

- ``+`` : Concaténation de chaînes

## Opérateurs de ternaire :

- ``condition ? valeurSiVrai : valeurSiFaux`` : Opérateur ternaire, retourne ``valeurSiVrai`` si ``condition`` est vraie, sinon retourne ``valeurSiFaux``.

## Autres opérateurs :

- ``typeof`` : Retourne le type de données d'une valeur
- ``instanceof`` : Vérifie si un objet est une instance d'une classe
- ``in`` : Vérifie si une propriété existe dans un objet
- ``delete`` : Supprime une propriété d'un objet

En savoir plus sur les opérateurs :

[https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Expressions\\_and\\_Operators#op%C3%A9rateurs\\_d'affectation](https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Expressions_and_Operators#op%C3%A9rateurs_d'affectation)

# À VOUS DE JOUER !



## Exercice 1 :

Déclarez une variable pour stocker une température en degrés Celsius.  
Convertissez cette température en degrés Fahrenheit en utilisant la formule :  $F = C * 9/5 + 32$ .

Affichez la température en degrés **Celsius** et **Fahrenheit**.

## Exercice 2 :

Déclarez deux variables pour stocker la longueur et la largeur d'un rectangle.

Calculez l'**aire du rectangle** en multipliant la longueur par la largeur.

Affichez le résultat.

## Exercice 3 :

Déclarez deux variables pour stocker un prénom et un nom.

**Concaténez** les deux variables pour former une chaîne complète.

Affichez la chaîne complète.

## Exercice 4 :

Déclarez une variable pour stocker le montant HT (hors taxe) d'un produit.

Déclarez une constante pour stocker le taux de TVA (par exemple, 20%).

Calculez le montant TTC (toutes taxes comprises) en ajoutant la TVA au montant HT.

Affichez le **montant HT**, le **montant de la TVA** et le **montant TTC**.

# À VOUS DE JOUER !



## Exercice 5 :

```
((4 >= 6) || ("herbe" != "verte")) && !(((12 * 2) == 144) && true)
```

Est-ce **true** ?

## Exercice 6 :

Déclarez deux variables pour stocker le poids en kilogrammes et la taille en mètres d'une personne.

Calculez l'indice de masse corporelle (IMC) en utilisant la formule :

$IMC = \text{poids} / (\text{taille} * \text{taille})$ .

**Affichez l'IMC avec deux décimales.**

*NB : vous pouvez vous aider de la méthode `toFixed()`.*

## Exercice 7 :

Déclarez une constante pour le montant minimum de commande pour la livraison gratuite.

Déclarez une variable pour le montant total de la commande.

Si le montant total est supérieur ou égal au montant minimum, affichez "Livraison gratuite !".

Sinon, affichez "Frais de livraison : X euros".

## Exercice 8 :

Déclarez une variable pour stocker un nombre binaire en String

Convertissez cette variable en décimal en utilisant la fonction `parseInt()`.

Affichez le **nombre binaire** et sa **conversion décimale**.



# LES OBJETS

```
const myObject = new Object
```





# Définition :

+

•

En JavaScript, les objets jouent un rôle fondamental. Ils sont utilisés pour **stocker des données** et **des fonctionnalités** de manière organisée sous la forme de **propriétés** et de **méthodes**.

Les objets en JavaScript peuvent être comparés à des boîtes qui contiennent différentes choses à l'intérieur.

Un **objet** est essentiellement une **collection de clés** (noms de propriétés) et de **valeurs associées à ces clés**. La valeur peut être de n'importe quel type de données, y compris des chaînes de caractères, des nombres, des tableaux, d'autres objets ou même des fonctions.

# Illustration :

- **nom**, **age**, et **profession** sont des propriétés qui contiennent des valeurs simples
- **adresse** est une propriété contenant un autre objet imbriqué à l'intérieur.
- **hobbies** est une propriété qui contient un tableau
- **direBonjour** est une méthode (fonction) qui affiche "Bonjour !" lorsqu'elle est appelée.

```
// Syntaxe littérale d'objet
const personne = {
  nom: "John",
  age: 30,
  profession: "Développeur",
  adresse: {
    ville: "Paris",
    codePostal: "75001"
  },
  hobbies: ["lecture", "voyage", "natation"],
  direBonjour: function() {
    console.log("Bonjour !");
  }
};
```





# Ajouter, modifier et supprimer des propriétés :

## À RETENIR :

- Un objet en JavaScript peut posséder **plusieurs propriétés** qui auront pour chacune d'elles une **valeur**.
- Pour déclarer un objet en JavaScript, vous devez utiliser les **accolades {}**.
- Pour **ajouter** ou **récupérer** une propriété, vous devez utiliser le point.

```
// Ajouter une nouvelle propriété
personne.email = "john@example.com";

// Modifier une propriété existante
personne.age = 31;

// Supprimer une propriété
delete personne.hobbies;
```

```
const personne = {  
  nom: "John",  
  age: 30,  
  direBonjour() {  
    console.log("Bonjour !");  
  }  
};  
  
personne.direBonjour(); // Affiche "Bonjour !"
```

```
const personne = {  
  nom: "John",  
  age: 30,  
};  
  
// Déstructuration d'objets  
const { nom, age } = personne;  
  
console.log(nom); // Affiche "John"  
console.log(age); // Affiche 30
```

## Méthodes raccourcies :

- Vous pouvez également utiliser une **notation raccourcie** pour définir des méthodes dans un objet.
- Plutôt que d'écrire `direBonjour: function() { ... }`, vous pouvez écrire **`direBonjour() { ... }`** :

## Déstructuration d'objets :

- La **déstructuration d'objets** permet d'**extraire les valeurs des propriétés d'un objet** dans des **variables distinctes**, ce qui rend le code plus concis

# Propriétés calculées :

- En **ES6**, vous pouvez utiliser des expressions pour **définir dynamiquement le nom des propriétés d'un objet**.

```
const nomDePropriete = "nom";
const personne = {
  [nomDePropriete]: "John",
  age: 30,
};

console.log(personne.nom); // Affiche "John"
```

```
const personne = { nom: "John" };
const details = { age: 30, profession: "Développeur" };

// Fusionner les propriétés de "details" dans "personne"
const personneAvecDetails = Object.assign(personne, details);

console.log(personneAvecDetails);
```

## Méthode Object.assign() :

- **Object.assign()** permet de copier les propriétés de plusieurs objets sources dans un objet cible.



# LES TABLEAUX

```
const mesFilms = [« Alien", "Jurassic Park", "Le Seigneur des Anneaux"]
```



# Définition :



En JavaScript, les **tableaux** sont des **structures de données** utilisées pour stocker des **collections ordonnées d'éléments**.

Un tableau peut contenir **n'importe quel type de données**, y compris des nombres, des chaînes de caractères, des objets et même d'autres tableaux.

Les tableaux sont largement utilisés pour **organiser et manipuler des données** dans les programmes JavaScript.



# Création de tableaux :



```
// Syntaxe littérale de tableau  
const fruits = ["pomme", "orange", "banane", "kiwi"];
```

Dans l'exemple ci-dessus, nous avons créé un tableau **fruits** contenant quatre éléments.

Les éléments sont séparés par des **virgules** et sont accessibles par leur **index** (position) dans le tableau.

Les **indices** commencent à **zéro**, donc le premier élément a l'index 0, le deuxième a l'index 1, et ainsi de suite.

# Gestion des éléments :

- On **accède** aux éléments du tableau en utilisant leur **index**.
- Il est également possible de **modifier les éléments du tableau** par l'index.
- La propriété **length** d'un tableau vous permet de connaître la **taille** (nombre d'éléments) du tableau.
- Pour ajouter de nouveaux éléments à la fin du tableau, on utilise **la méthode push()**
- Pour supprimer le dernier élément du tableau, on utilise la méthode **pop()**

```
console.log(fruits[0]); // Affiche "pomme"  
console.log(fruits[2]); // Affiche "banane"
```

```
fruits[1] = "citron";  
console.log(fruits); // Affiche ["pomme", "citron", "banane", "kiwi"]
```

```
console.log(fruits.length); // Affiche 4
```

```
fruits.push("mangue");  
console.log(fruits); // Affiche ["pomme", "citron", "banane", "kiwi", "mangue"]
```

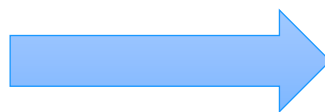
```
fruits.pop();  
console.log(fruits); // Affiche ["pomme", "citron", "banane", "kiwi"]
```



# Quelques méthodes :

## Méthode `.map()`

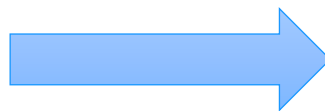
```
const numbers = [4, 9, 16, 25]
let x = numbers.map(Math.sqrt)
console.log(x); // sortie: [2, 3, 4, 5]
```



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

## Méthode `.reduce()`

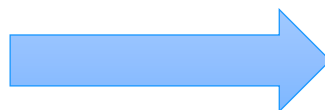
```
const numbers = [20, 5, 5]
function maFct(total, n)
{ return total - n; }
console.log(numbers.reduce(maFct))
//sortie: 10
```



La méthode **reduce()** réduit le tableau à une valeur unique. La méthode `reduce()` exécute une fonction fournie pour chaque valeur du tableau (de gauche à droite). La valeur de retour de la fonction est stockée dans un accumulateur (résultat/total).

## Méthode `.include()`

```
const tab = [1, 2, 3]
tab.includes(3 //sortie: true
tab.includes(0) //sortie: false
```



La méthode **include()** vérifie si le tableau inclut l'élément passé dans la méthode. La méthode renvoie **true** si l'élément est présent dans le tableau, et **false** sinon.

+

•

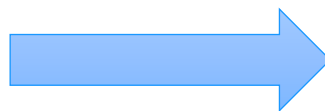




# Quelques méthodes :

## Méthode `.forEach()`

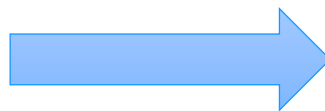
```
const tab = [1, 2, 3, 4, 5]
tab.forEach( i => {
  console.log(i)    //sortie: 1 2 3 4 5
});
```



La méthode **forEach()** permet de parcourir les éléments du tableau.

## Méthode `.filter()`

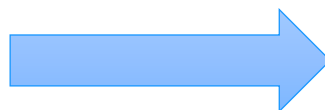
```
const tab = [0, 1, 2, 3, 4, 5]
// élément(s) supérieur(s) à 2
let result = tab.filter(n => n > 2)
console.log(result) //sortie: [3, 4, 5]
```



La méthode **filter()** crée un nouveau tableau avec les éléments qui remplissent une condition.

## Méthode `.sort()`

```
const os = ["Windows", "Linux", "Apple"]
os.sort()
console.log(os)
//sortie: ["Apple", "Linux", "Windows"]
```



La méthode **sort()** trie les éléments d'un tableau dans l'ordre lexicographique (par ordre alphabétique pour les chaînes de caractères) ou numérique (pour les nombres).

Attention toutefois, la méthode effectue un **tri mutatif**, modifiant le tableau d'origine directement.

+

•



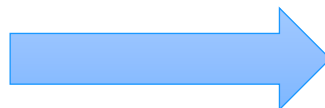
# Quelques méthodes :

## Méthode `.every()`

```
const tab = [1, 2, 3, 4];

// tous les éléments sont < 5
let b = tab.every(n => n < 5);
console.log(b) //sortie: true

// tous les éléments sont > 2
let c = tab.every(n => n > 2);
console.log(c) //sortie: false
```



La méthode **every()** est utilisée pour vérifier si tous les éléments du tableau satisfont une condition donnée en utilisant une fonction de test. +

**array.every**(callback(element, index, array))

La fonction **callback** prend trois arguments :

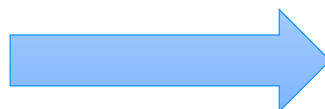
- **element** (la valeur de l'élément en cours de test)
- **index** (l'index de l'élément en cours de test=)
- **array** (le tableau sur lequel `every()` est appelée).

## Méthode `.some()`

```
const arr = [1, 2, 3];

// au moins un élément est > 2
let b = arr.some(n => n > 2)
console.log(b) // output: true

// au moins un élément est <= 0
let c = arr.some(n => n < 0)
console.log(c) // output: false
```



La méthode **some()** renvoie **true** si au moins un des éléments du tableau satisfait la condition spécifiée par la fonction de rappel.

Elle s'arrête dès qu'elle trouve un élément qui satisfait la condition et ne vérifie pas les autres éléments du tableau. Si aucun élément ne passe le test, la méthode `some()` renvoie **false**.



# Copie par valeur et par référence + •

En JavaScript, dans le cas d'une **copie par valeur**, si vous **modifiez la valeur d'une des deux variables**, la valeur de **l'autre ne change pas**.

Dans le cas d'une **copie par référence**, si vous **changez la valeur de la première variable**, la valeur de la seconde est affectée aussi.



# COPIE PAR VALEUR ET PAR RÉFÉRENCE

UNE INITIATION À JAVASCRIPT

```
1 //////////////////////////////////////////////////
2 // Copie par valeur
3 //////////////////////////////////////////////////
4 let variableSimple1 = 25
5 let variableSimple2 = variableSimple1
6
7 variableSimple2 = 30
8
9 // Le console.log va afficher 25, le fait d'avoir changé la valeur de variableSimple2 ne change rien
  pour variableSimple1
10 console.log("variableSimple1", variableSimple1)
11 console.log("variableSimple2", variableSimple2)
12
13 //////////////////////////////////////////////////
14 // Copie par référence
15 //////////////////////////////////////////////////
16 let variableComplexe1 = ['pomme', 'cerise']
17 let variableComplexe2 = variableComplexe1
18 let variableComplexe3 = [...variableComplexe1];
19
20 variableComplexe2.push('poire')
21
22 // Le console.log va afficher pomme cerise ET poire. On a modifié la seconde variable, mais le
  contenu de la première a été changé aussi, parce que c'est le même contenu.
23 console.log('variableComplexe1', variableComplexe1)
24 console.log('variableComplexe2', variableComplexe2)
25 console.log('variableComplexe3', variableComplexe3)
```

Lorsqu'on copie une **variable simple**, JavaScript réalise une copie par valeur (la valeur est dupliquée).

Lorsqu'on copie une **variable complexe** (ici un tableau), JavaScript réalise une copie par référence (les deux variables pointent sur la même valeur).

# À VOUS DE JOUER !



## Exercice 1 :

Créez un objet **personne** avec les propriétés vous concernant suivantes :  
prenom, nom, age, ville

Affichez les propriétés prenom, nom, age et ville de l'objet **personne** dans des console.log() séparés.

## Exercice 2 :

Créez un tableau **personnes** contenant les objets suivants :

- {prenom: "John", nom: "Doe", age: 25, ville: "Rouen"}
- {prenom: "Jane", nom: "Doe", age: 30, ville: "Paris"}
- {prenom: "Jim", nom: "Doe", age: 35, ville: "Caen"}

Affichez le tableau **personnes** dans la console avec **log()** et **table()**.

## Exercice 3 :

Ajoutez un objet à la fin du tableau **personnes** avec la méthode **push()** :

- {prenom: " Marc", nom: "Doe", age: 32, ville: "Marseille"}

Affichez le tableau **personnes** dans la console avec **log()** et **table()**.

## Exercice 4 :

Modifiez le premier objet du tableau **personnes** en utilisant la syntaxe suivante : **personnes[0].prenom = "Jean"**

Supprimez le deuxième objet du tableau **personnes** en utilisant la méthode **splice()**

Affichez le tableau **personnes** dans la console avec **log()** et **table()**.

# À VOUS DE JOUER !



## Exercice 5 :

Créez un tableau contenant les nombres de 1 à 10.

- Affichez le premier élément du tableau.
- Affichez le dernier élément du tableau.
- Affichez la longueur du tableau.

## Exercice 6 :

Créez un tableau contenant les chiffres de 10 à 100 de 10 en 10.

- Affichez le tableau.
- Inversez le tableau avec la méthode **reverse()** puis affichez le tableau.

## Exercice 7 :

Créez un tableau **numbers** = [ 2, 5, 1, 9, 0, 3, 7, 4, 6, 8 ]

- Affichez le tableau.
- Triez le tableau avec la méthode **sort()** puis affichez le tableau.

## Exercice 8 :

En repartant du tableau **numbers** trié :

- Ajoutez le nombre 11 à la fin du tableau avec la méthode **push()**.
- Affichez le tableau.
- Ajoutez le nombre 0 au début du tableau avec la méthode **unshift()**.
- Affichez le tableau.
- Supprimez le dernier élément du tableau avec la méthode **pop()**.
- Affichez le tableau.



# LES CONDITIONS

If, else, else if, switch, ternaire



# Les conditions en JavaScript :

+



En JavaScript, les conditions sont des structures de contrôle utilisées pour effectuer des actions différentes en fonction d'une **condition** ou d'une **expression booléenne** (qui peut être soit **true** soit **false**).

Les conditions permettent au programme de prendre des décisions et de s'exécuter de manière sélective en fonction de la valeur d'une **expression**.





# Les structures de contrôle :

+

•

**if** : La déclaration if **permet d'exécuter un bloc de code** si une condition donnée est évaluée à **true**. Si la condition est fausse (**false**), le bloc de code dans le if **ne sera pas exécuté**.

**else** : L'instruction else est utilisée avec if pour **fournir un bloc de code alternatif à exécuter** si la condition du if est **fausse**.

**else if** : L'instruction else if est utilisée pour **tester plusieurs conditions en série**. Elle est utilisée avec if et else pour fournir une autre condition à vérifier si la condition du if est **fausse**.

**Opérateur ternaire** : L'opérateur ternaire (**condition ? valeurSiVrai : valeurSiFaux**) est une **syntaxe courte** pour exprimer une condition en une seule ligne.

**switch** : La déclaration switch permet **de gérer plusieurs cas différents** en fonction de **la valeur d'une expression**.



```
const age = 18;

if (age >= 18) {
  console.log("Vous êtes majeur.");
}
```

```
const age = 15;

if (age >= 18) {
  console.log("Vous êtes majeur.");
} else {
  console.log("Vous êtes mineur.");
}
```

```
const score = 80;

if (score >= 90) {
  console.log("Très bien !");
} else if (score >= 70) {
  console.log("Bien !");
} else {
  console.log("À améliorer !");
}
```

# IF

# ELSE

# ELSE IF



# TERNNAIRE

# SWITCH

```
const age = 25;  
const statut = age >= 18 ? "majeur" : "mineur";  
console.log(statut); // Affiche "majeur"
```

```
const jour = "lundi";  
  
switch (jour) {  
  case "lundi":  
    console.log("C'est le début de la semaine.");  
    break;  
  case "vendredi":  
    console.log("Bonne fin de semaine !");  
    break;  
  default:  
    console.log("Autre jour de la semaine.");  
}
```



# INCRÉMENTATION DÉCRÉMENTATION

+  
•  
○

+  
•  
○

++ --



# Incrément et décrétement :

En JavaScript, l'**incrément** et le **décrément** sont des opérations permettant d'augmenter ou de diminuer la valeur d'une variable numérique d'une unité.

## Incrémentation :

L'opérateur d'incrément est représenté par **++**. Il est utilisé pour augmenter la valeur d'une variable numérique d'une unité.

## Décrémentation :

L'opérateur de décrémentation est représenté par **--**. Il est utilisé pour diminuer la valeur d'une variable numérique d'une unité.

```
var number = 0;  
number++;  
alert(number); // Affiche : « 1 »  
number--;  
alert(number); // Affiche : « 0 »
```

+

•



# LES BOUCLES

```
for(i=0, i<monTableau.length, i++) { ... }
```





# Les boucles en JavaScript :

+



En JavaScript, les boucles sont **des structures de contrôle** qui permettent **d'exécuter un bloc de code de manière répétée** tant qu'une condition est vraie, ou jusqu'à ce qu'une condition soit satisfaite.

Les boucles sont utilisées pour **automatiser des tâches répétitives** ou pour **parcourir des collections de données** telles que les tableaux.



# Boucle For

La boucle **for** est utilisée pour exécuter un bloc de code un certain nombre de fois.  
Elle se compose de **trois parties** :

**L'initialisation** : Définit une variable et l'initialise généralement à une valeur de départ.

**La condition** : Une expression booléenne qui est évaluée avant chaque itération. Si la condition est vraie, le bloc de code est exécuté. Sinon, la boucle se termine.

**L'itération** : Une instruction qui modifie la variable utilisée dans la condition à chaque itération.

```
for (let i = 0; i < 5; i++) {  
    console.log(i);  
}
```





# Boucle While

La boucle **while** est utilisée pour exécuter un bloc de code **tant qu'une condition est vraie**. La condition est vérifiée avant chaque itération.

```
let i = 0;

while (i < 5) {
  console.log(i);
  i++;
}
```



# Boucle Do While

La boucle **do...while** est similaire à la boucle **while**, mais elle **vérifie la condition après l'exécution du bloc de code**. Cela garantit que le bloc de code **est exécuté au moins une fois**, même si la condition est fausse dès le départ.

```
let i = 0;

do {
  console.log(i);
  i++;
} while (i < 5);
```

Il est important de faire attention aux boucles pour éviter les boucles infinies, où la condition ne devient jamais fausse. Pour éviter cela, on peut utiliser des déclarations **break** pour sortir de la boucle de manière anticipée.



# Autres boucles et directives de contrôle :

En plus des boucles `for`, `while`, `do...while`, JavaScript propose également les boucle **`for...of`** et **`for...in`** ainsi que des **directives de contrôle** de boucle telles que **`break`** et **`continue`**.

```
const personne = {
  nom: "John",
  age: 30,
  profession: "Développeur"
};

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

La boucle **`for...in`** est une boucle spécifique en JavaScript utilisée pour parcourir **les propriétés d'un objet**.

Elle itère sur toutes les propriétés énumérables de l'objet, y compris les propriétés héritées de ses prototypes. Cette boucle est généralement utilisée avec des **objets**, mais elle peut également être utilisée avec des **tableaux**.

```
const fruits = ["pomme", "orange", "banane"];

for (const fruit of fruits) {
  console.log(fruit);
}
```

La boucle **`for...of`** permet de parcourir les éléments d'un **tableau**, d'une **chaîne de caractères**, d'un **objet itérable** ou d'une **collection d'éléments** d'une manière plus simple et lisible que les autres boucles.



# Autres boucles et directives de contrôle :

```
for (let i = 0; i < 5; i++) {  
  console.log(i);  
  if (i === 2) {  
    break; // Sort de la boucle lorsque i atteint 2  
  }  
}
```

La **directive break** est utilisée pour sortir d'une boucle **de manière anticipée** lorsqu'une **condition donnée** est **satisfaite**. Elle est souvent utilisée pour **éviter les boucles infinies** ou pour arrêter la boucle lorsque certaines conditions sont remplies.

```
for (let i = 0; i < 5; i++) {  
  if (i === 2) {  
    continue; // Passe à l'itération suivante lorsque i atteint 2  
  }  
  console.log(i);  
}
```

La **directive continue** est utilisée pour **interrompre l'itération** en cours dans une boucle et **passer à l'itération suivante**. Cela permet de sauter certaines itérations de la boucle en fonction d'une condition donnée.



# LES FONCTIONS



```
function myFonction() {...}
```



# Les fonctions en JavaScript :

+

•

Les **fonctions** permettent de **regrouper un ensemble d'instructions** pour effectuer une tâche spécifique. Cela rend le code **plus organisé, modulaire et réutilisable**.

Une fonction peut :

- contenir des informations, qu'on appelle **paramètres**
- retourner un **résultat**
- effectuer une **action**

## Portée (scope) des variables :

Les variables déclarées à l'intérieur d'une fonction sont généralement locales à cette fonction, ce qui signifie qu'elles ne sont pas accessibles en dehors de celle-ci. Cela contribue à éviter les conflits de noms de variables.



# Déclaration et appel de fonction :

+

•

On **déclare** une fonction en utilisant le mot-clé **function**, suivi du **nom de la fonction** et de ses **paramètres** entre parenthèses.

```
// Déclaration d'une fonction nommée "saluer" avec un paramètre "nom"
function saluer(nom) {
    console.log(`Bonjour, ${nom} !`);
}
```

On **appelle** une fonction en **utilisant son nom** suivi des **parenthèses**. Si la fonction prend des **paramètres**, il faut fournir des **valeurs** pour ces paramètres lors de l'appel.

```
saluer("Alice"); // Cela affichera "Bonjour, Alice !" dans la console
saluer("Bob");   // Cela affichera "Bonjour, Bob !" dans la console
```



# Retours, fonctions anonymes et fléchées

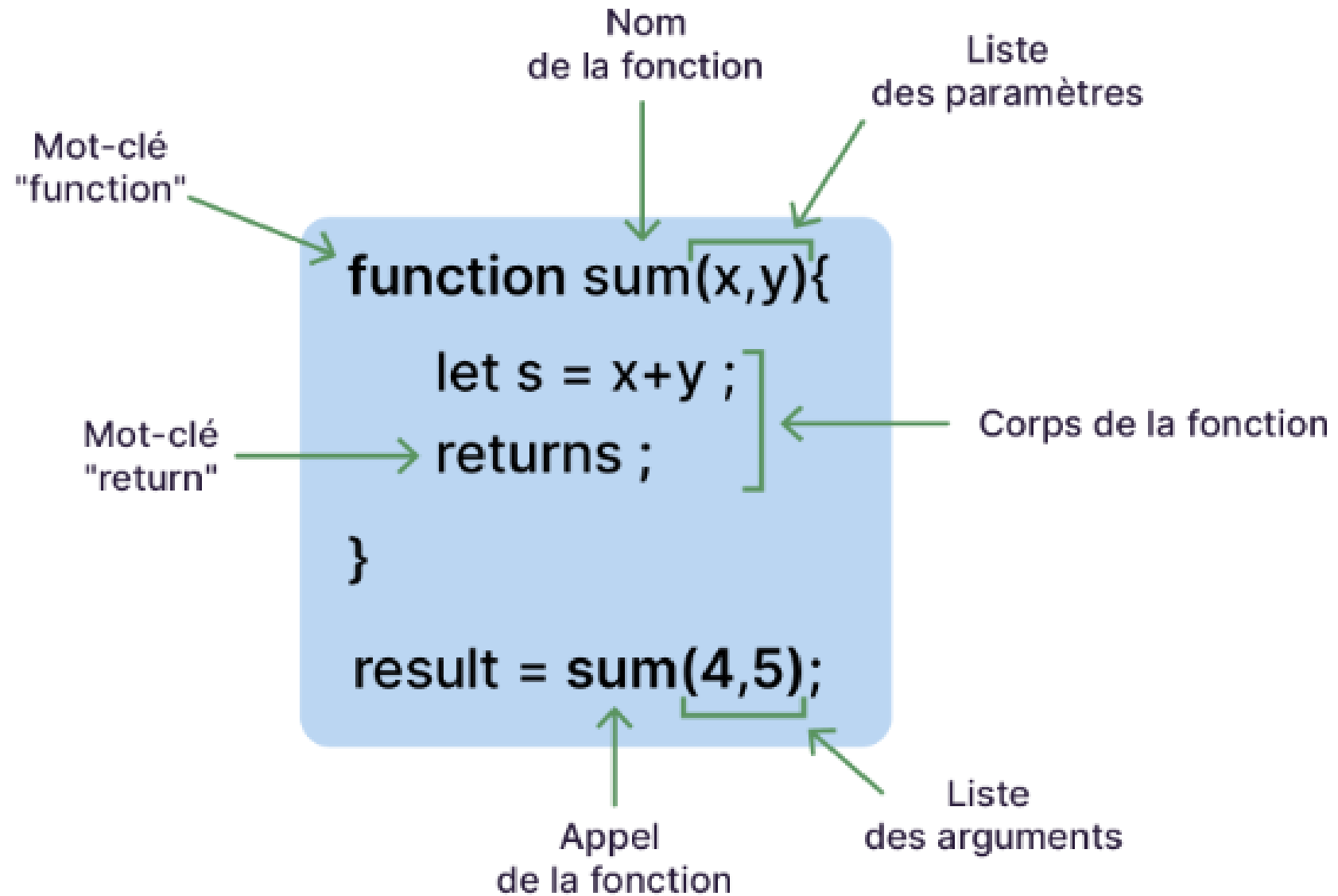
```
function multiplier(a, b) {  
    return a * b;  
}  
  
let resultat = multiplier(5, 3); // Le résultat sera 15  
console.log(resultat); // Affiche 15 dans la console
```

En plus des **fonctions nommées**, vous pouvez également utiliser des **fonctions anonymes** (sans nom) et des **fonctions fléchées** (arrow functions) pour définir des fonctions de **manière plus concise**.

Les fonctions peuvent également **renvoyer des valeurs** en utilisant l'instruction **return**. Cela permet **d'obtenir des résultats** de calculs ou d'opérations **effectués** à l'intérieur de la fonction.

```
// Fonction anonyme  
let carre = function(x) {  
    return x * x;  
};  
  
// Fonction fléchée équivalente  
let carreArrow = (x) => x * x;
```







# MANIPULER UNE PAGE WEB

```
document.getElementById("maDiv")
```

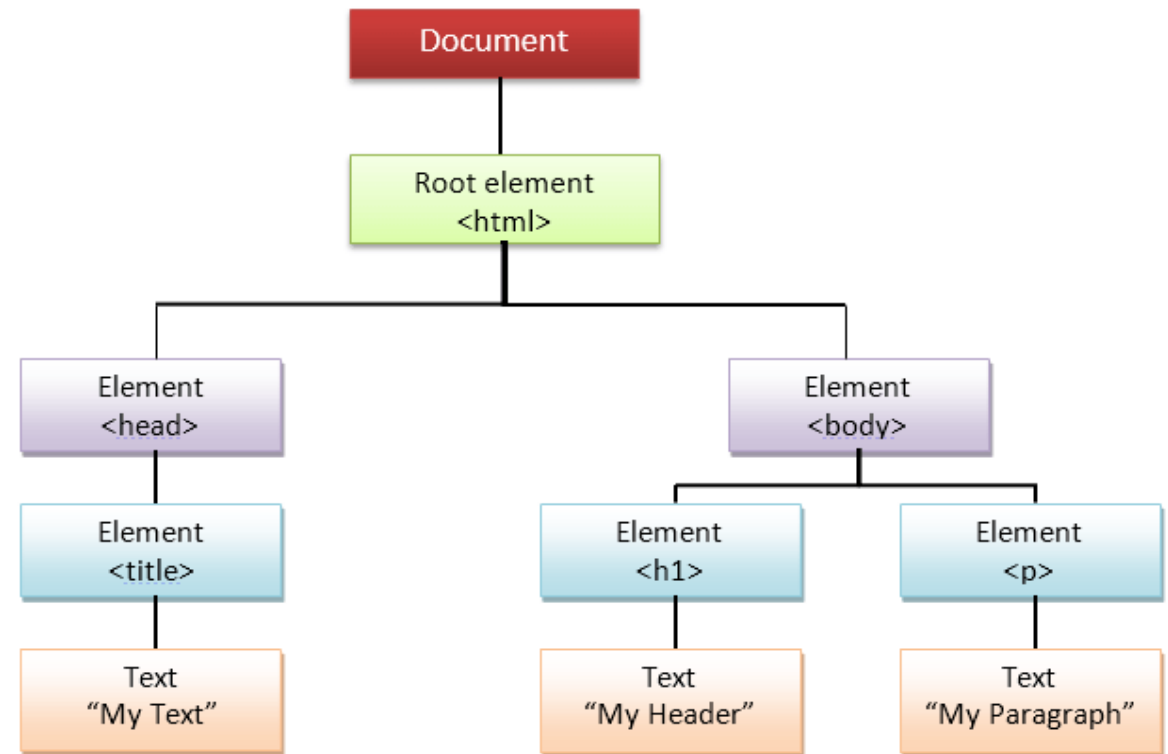


# Manipuler les éléments de la page Web

- Manipuler les éléments d'une page web en utilisant JavaScript va nous permettre de créer des **interactions dynamiques et réactives** sur un site Web.
- Il est en effet possible **d'accéder aux éléments HTML**, de **modifier leurs propriétés**, de **gérer les événements** et bien plus encore.

Le **DOM** est une **représentation** du **document HTML** source.

Il s'agit pour l'essentiel d'une conversion de la structure et du contenu du document HTML en un modèle objet utilisable par divers programmes.





# Le DOM :

+

●

Le **Document Object Model** ou **DOM** (pour modèle objet de document) est une interface de programmation pour les documents HTML, XML et SVG.

Il fournit une **représentation structurée du document** sous forme d'un **arbre**.

Le DOM représente le document comme un **ensemble de nœuds** et **d'objets** possédant des **propriétés** et des **méthodes**.

Les nœuds peuvent également avoir des **gestionnaires d'événements** qui se déclenchent lorsqu'un événement se produit.



# Accéder aux éléments HTML :

+

•

```
// Sélection par ID
let elementById = document.getElementById('monId');

// Sélection par classe
let elementsByClass = document.getElementsByClassName('maClasse');

// Sélection par balise
let elementsByTag = document.getElementsByTagName('div');

// Sélection par sélecteur CSS
let elementBySelector = document.querySelector('.maClasse');
let elementsBySelectorAll = document.querySelectorAll('div');
```

On utilise les méthodes de sélection d'éléments pour accéder aux éléments **HTML** à partir du **DOM**.

On peut utiliser des **sélecteurs CSS** ou des **méthodes spécifiques** pour obtenir une référence à un élément.



# Attribut defer :

En utilisant **defer**, les navigateurs modernes exécutent les scripts différés une fois que le **contenu de la page HTML** a été entièrement **analysé** et que le **DOM est prêt**.

Cela garantit que les scripts seront exécutés **avant** que l'événement **DOMContentLoaded** ne soit déclenché. On peut donc interagir avec le DOM dès que les scripts sont exécutés, ce qui simplifie **la manipulation des éléments** et **l'ajout d'écouteurs d'événements**.

C'est une **meilleure pratique** pour l'inclusion de scripts dans les pages web, car cela permet d'éviter les **problèmes de blocage du rendu** et **d'améliorer les performances globales** d'un site.

```
<script src="mon-script.js" defer></script>
```



# Modifier les propriétés des éléments :

+

•

Il est possible de **modifier** les **propriétés** des **éléments HTML** pour **changer** leur **apparence** ou leur **comportement**.

```
let element = document.getElementById('monElement');

// Modifier le texte à l'intérieur d'un élément
element.textContent = 'Nouveau texte';

// Modifier la valeur d'un attribut
element.setAttribute('src', 'nouveau_image.jpg');

// Modifier les classes d'un élément
element.classList.add('nouvelleClasse');
element.classList.remove('ancienneClasse');
```



# Gérer les événements :

```
let bouton = document.getElementById('monBouton');

// Ajouter un écouteur d'événement pour le clic
bouton.addEventListener('click', () => {
    alert('Bouton cliqué !');
});

// Supprimer un écouteur d'événement
bouton.removeEventListener('click', maFonction);
```

On peut ajouter des **écouteurs d'événements** pour **répondre** aux **actions des utilisateurs**, comme les clics, les survols de souris, etc.

La **programmation événementielle** consiste à réagir à des **événements** et **exécuter du code** au moment où ces événement se produisent.

+

•



| Nom de l'événement      | Description                                       |
|-------------------------|---|
| <code>load</code>       | fin de chargement de la page web                  |
| <code>click</code>      | clic sur un élément                               |
| <code>dblclick</code>   | double clic sur un élément                        |
| <code>keydown</code>    | une touche est appuyée                            |
| <code>keypress</code>   | une touche est maintenue enfoncée                 |
| <code>keyup</code>      | une touche est relâchée                           |
| <code>mouseenter</code> | le curseur entre au dessus d'un élément           |
| <code>mouseleave</code> | le curseur quitte l'élément                       |
| <code>select</code>     | sélection d'une option dans un select             |
| <code>change</code>     | changement de valeur sur un select, un checkbox.. |
| <code>submit</code>     | soumission d'un formulaire                        |
| <code>focus</code>      | l'élément reçoit le focus                         |
| <code>blur</code>       | l'élément perd le focus                           |

[En savoir plus sur les événements](#)



# Modifier le style des éléments :

```
let element = document.getElementById('monElement');

// Modifier le style CSS d'un élément
element.style.backgroundColor = 'blue';
element.style.fontSize = '18px';
```

JavaScript permet de **modifier le style** des éléments pour effectuer des changements visuels.

```
// Ajout d'un écouteur d'événement pour afficher la div
boutonAfficher.addEventListener('click', () => {
    maDiv.style.display = 'block'; // ou 'inline', 'flex', etc.
});

// Ajout d'un écouteur d'événement pour masquer la div
boutonMasquer.addEventListener('click', () => {
    maDiv.style.display = 'none';
});
```



# Créer de nouveaux éléments :

Il est possible de **créer, d'ajouter** ou de **supprimer** des éléments dynamiquement.

```
let nouvelElement = document.createElement('div');
nouvelElement.textContent = 'Nouvel élément créé';
document.body.appendChild(nouvelElement);

let parentElement = document.getElementById('parent');
let enfantElement = document.getElementById('enfant');
parentElement.removeChild(enfantElement);
```

La méthode **insertBefore()** permet d'insérer un nœud avant un nœud existant dans l'arborescence.

La méthode **insertAdjacentHTML()** permet d'insérer du code HTML à un emplacement spécifié par rapport à l'élément cible.

La propriété **innerHTML** peut être utilisée pour ajouter du contenu HTML à un élément existant (attention toutefois aux risques d'attaques XSS).



# Interpolation et innerHTML :

```
// Données
let nom = "John Doe";
let age = 30;
let profession = "Développeur Web";

// Création du contenu HTML en utilisant les template literals
let profilHTML = `
  <div class="profil">
    <h2>${nom}</h2>
    <p>Âge : ${age} ans</p>
    <p>Profession : ${profession}</p>
  </div>
`;

// Ajout du contenu au document
document.body.innerHTML = profilHTML;
```

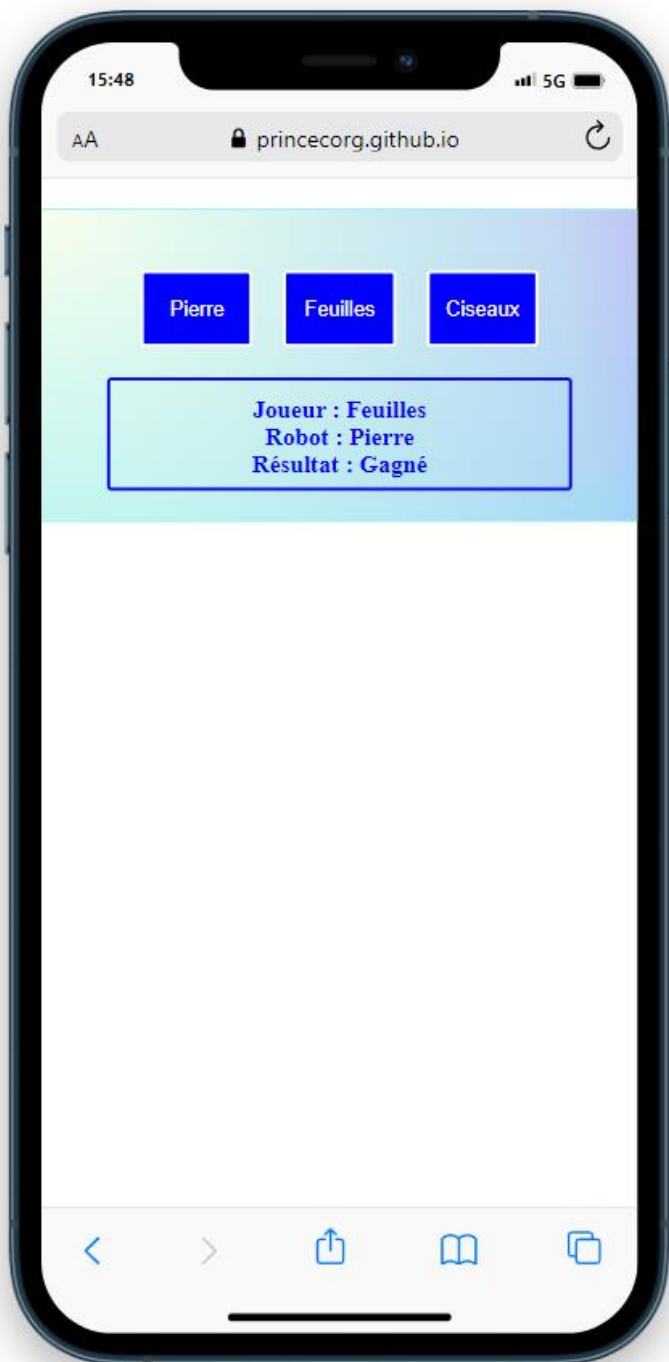
L'interpolation avec **innerHTML** permet d'insérer des valeurs **variables** ou **dynamiques** dans du **contenu HTML**.

L'interpolation est plus **sécurisée** que la **concaténation simple (+)**.



# MINI PROJET : SHIFUMI

« Pierre, feuille, ciseaux »



## SHIFUMI

**Réalisez une page Web** permettant à l'internaute de jouer contre votre programme.

L'utilisateur choisit un des trois boutons et le programme fait de même de manière **aléatoire**.

Comparez les résultats une fois l'événement déclenché puis afficher le résultat dans un conteneur dédié.

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1 />
    <title>Shifumi</title>
    <link rel="stylesheet" href="./css/style.css" />
  </head>
  <body>
    <main>
      <div class="box">
        <button>Pierre</button>
        <button>Feuilles</button>
        <button>Ciseaux</button>
      </div> .box

      <div class="resultat"></div>
    </main>
    <script src="./js/shifumi.js"></script>
  </body>
</html>
```



# LES FORMULAIRES

Une remarque, une question ? Contactez-nous



# Les formulaires en JavaScript :

+



Les formulaires en JavaScript sont utilisés pour **recueillir des données** à partir de l'utilisateur sur une page web. Ce sont des **éléments clés** d'une page web **interactive**.

JavaScript est souvent utilisé pour **améliorer l'interactivité** et la **validation des données** dans les formulaires.

Le formulaire est composé d'une **balise form** qui englobe une série d'autres balises qui composent le formulaire : **labels**, **input**, **textarea** et **select**.





# Les balises de formulaire :

+

•

**Label** : sert à indiquer un **texte**, lié à un **champ de saisie**.

```
<label for="nom">Nom :</label>
<input type="text" id="nom" name="nom">
```

**Input** : balise la plus courante d'un formulaire, elle permet à l'utilisateur de **saisir des données**.

```
<label for="commentaire">Commentaire :</label>
<textarea id="commentaire" name="commentaire" rows="4" cols="50"></textarea>
```

**Textarea** : permet la saisie de plusieurs lignes de texte.

**Select** : crée une **liste déroulante** où l'utilisateur peut sélectionner une **option** à partir d'une liste prédéfinie.

```
<select id="pays" name="pays">
  <option value="france">France</option>
  <option value="espagne" selected>Espagne</option>
  <option value="italie">Italie</option>
  <option value="allemagne">Allemagne</option>
</select>
```



# Les types de la balise Input

+

•

La **valeur de l'attribut type** d'une balise **Input** conditionne grandement son fonctionnement.

En voici quelques exemples.

- **text** : pour saisir un texte ;
- **password** : pour saisir un texte tout en cachant ce qui est saisi ;
- **email** : pour saisir un texte et vérifier que son format correspond bien à celui d'un e-mail ;
- **number** : pour saisir un nombre ;
- **checkbox** : pour afficher une case à cocher ;
- **radio** : pour afficher un **bouton radio**, c'est-à-dire un bouton qui permet à l'utilisateur de sélectionner un seul élément dans une liste ;
- **date** : pour saisir une date à l'aide d'un calendrier qui s'affiche au clic sur le champ.

[OpenClassrooms : apprenez à programmer avec JavaScript](#)



# Récupérer la valeur d'un champ : + •

Pour la plupart des champs il suffit d'utiliser la propriété **value**.

Cela fonctionne notamment pour :

- Les inputs de type texte, numérique, e-mail, mot de passe ;
- La balise **textarea** ;
- La balise **select**.

```
let baliseNom = document.getElementById("nom")
let nom = baliseNom.value
console.log(nom); // affiche ce qui est contenu dans la balise name
```



# Cases à cocher et boutons radio:

**Case à cocher** : la valeur du champ est **true** si la case est cochée ou **false**. On utilise la propriété **checked** pour le vérifier.

**Boutons radio** : ils partagent tous **la même valeur** pour l'attribut **name**. Il faut donc **boucler sur l'ensemble des boutons** pour déterminer qui a la propriété **checked** à **true**.

```
let baliseAccepter = document.getElementById("accepter")
let accepter = baliseAccepter.checked
console.log(accepter); // affiche true ou false
```

```
let baliseCouleur = document.querySelectorAll('input[name="couleur"]')
let couleur = ""
for (let i = 0; i < baliseCouleur.length; i++) {
  if (baliseCouleur[i].checked) {
    couleur = baliseCouleur[i].value
    break
  }
}
```



# event.preventDefault() :

+

•

Si l'on clique sur le **bouton Envoyer**, l'événement **submit** est envoyé, et le navigateur veut **envoyer les données au serveur et réafficher la page**.

Pour **empêcher ce comportement par défaut**, il faut donc :

- écouter l'événement **submit**
- empêcher ce comportement grâce à la méthode **preventDefault**.

```
form.addEventListener("submit", (event) => {  
    // On empêche le comportement par défaut  
    event.preventDefault();  
    console.log("Il n'y a pas eu de rechargement de page");  
});
```



# Règles de validation :

Un bloc **try-catch** dans la validation du formulaire peut être utile pour **gérer les erreurs de manière plus contrôlée** et fournir une **meilleure expérience utilisateur** en affichant des messages d'erreur plus conviviaux.

```
document.getElementById("monFormulaire").addEventListener("submit", function(event) {  
    event.preventDefault(); // Empêche la soumission automatique du formulaire  
  
    try {  
        if (validerFormulaire()) {  
            // Soumettre le formulaire si la validation réussit  
            document.getElementById("monFormulaire").submit();  
        }  
    } catch (erreur) {  
        // Gérer l'erreur et afficher un message à l'utilisateur  
        alert("Une erreur est survenue : " + erreur.message);  
    }  
});
```

+

•



# Règles de validation :

```
function validerFormulaire() {  
    var nom = document.getElementById("nom").value;  
    var email = document.getElementById("email").value;  
  
    if (nom === "") {  
        throw new Error("Le champ Nom ne peut pas être vide");  
    }  
  
    var emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;  
    if (!emailRegex.test(email)) {  
        throw new Error("Veuillez saisir une adresse email valide");  
    }  
  
    // Autres validations pour les autres champs  
  
    // Si tout est valide, le formulaire peut être soumis  
    return true;  
}
```

You, maintenant • Uncommitted changes

+

•

Les **expressions régulières** (Regex) vont permettre de définir un format de contrôle précis et il est possible d'utiliser **throw new Error(message)** pour lancer soi-même ses exceptions.



# LES CLASSES

Class Etudiant extends Personne { ... }





# Les classes en JavaScript :

+



Les **classes** ont été introduites dans **ECMAScript 6** (ES6) pour fournir une **syntaxe plus orientée objet** et une meilleure **encapsulation** par rapport à la manipulation directe des **prototypes**.

```
// Définition d'une classe
class Personne {
  constructor(nom, age) {
    this.nom = nom;
    this.age = age;
  }

  // Méthode de la classe
  sePresenter() {
    console.log(`Bonjour, je m'appelle ${this.nom} et j'ai ${this.age} ans.`)
  }
}
```



# Instanciación d'objets :

+



```
// Instanciación d'objets à partir de la classe  
const personne1 = new Personne("Alice", 30);  
const personne2 = new Personne("Bob", 25);
```

**personne1.sePresenter()** affichera « Bonjour, je m'appelle Alice et j'ai 30 ans. »



# Héritage :

+

•

Les classes permettent également d'utiliser l'héritage pour créer des **sous-classes** avec des **propriétés** et des **méthodes supplémentaires**.

```
class Etudiant extends Personne {  
  constructor(nom, age, niveau) {  
    super(nom, age);  
    this.niveau = niveau;  
  }  
  
  sePresenter() {  
    console.log(  
      `Je suis ${this.nom}, j'ai ${this.age} ans et je suis en ${this.niveau}.`  
    );  
  }  
}
```

Utilisation de **super** pour appeler le constructeur de la classe parente



# LES PROMESSES



Je te promets le ciel au-dessus de ta couche ...



# Les promesses :

+

●

Les **promesses** sont un mécanisme essentiel pour **gérer les opérations asynchrones** de manière **plus propre et plus lisible**.

Elles sont utilisées pour **traiter des tâches qui prennent du temps à s'exécuter**, telles que les **appels réseau**, les **opérations de fichiers**, etc.

Les promesses aident à éviter le "**callback hell**" (l'enchevêtrement de plusieurs niveaux de rappels) en fournissant une **syntaxe plus structurée** pour gérer les retours de ces **opérations asynchrones**.



# Les principaux concepts :

+

•

```
const myPromise = new Promise((resolve, reject) => {  
  // Effectuer une tâche asynchrone  
  if (/* succès */) {  
    resolve("Résultat réussi");  
  } else {  
    reject("Échec de la tâche");  
  }  
});
```

On peut **chaîner des méthodes** à une **promesse** en utilisant les méthodes `.then()` et `.catch()`. La méthode `.then()` est appelée lorsque la promesse est résolue avec succès, tandis que `.catch()` est appelée lorsque la promesse est rejetée..

Une promesse est créée en appelant le **constructeur Promise** et en passant une fonction avec deux arguments (**resolve** et **reject**). Cette fonction représente le **travail asynchrone** à effectuer. **resolve** est appelé lorsque la tâche est réussie, et **reject** est appelé en cas d'échec.

```
myPromise  
  .then(result => {  
    console.log(result); // Affiche "Résultat réussi"  
  })  
  .catch(error => {  
    console.error(error); // Affiche "Échec de la tâche"  
  });
```



# Chaînage - promesses multiples : +

```
myPromise
  .then(result => {
    return anotherAsyncFunction(result); // Retourne une nouvelle promesse
  })
  .then(finalResult => {
    console.log(finalResult);
  })
  .catch(error => {
    console.error(error);
  });
```

Les méthodes `.then()` peuvent renvoyer **une nouvelle promesse**, ce qui permet de **chaîner plusieurs opérations asynchrones** de manière lisible.

```
const value = 42;
const resolvedPromise = Promise.resolve(value);
```

```
const promise1 = someAsyncFunction();
const promise2 = anotherAsyncFunction();

Promise.all([promise1, promise2])
  .then(results => {
    // Traiter les résultats des promesses ici
  })
  .catch(error => {
    console.error(error);
  });
```

On peut utiliser **Promise.all()** pour attendre que plusieurs promesses soient résolues avant de continuer..

Si l'on a besoin de créer une promesse qui est **résolue** dès qu'une certaine condition est remplie, il faut utiliser **Promise.resolve()**.



# FONCTIONS ASYNCHRONES

Async - Await





# Les fonctions asynchrones :

+

•

Les **fonctions asynchrones** avec **async/await** sont devenues une **norme pour gérer les opérations asynchrones** en JavaScript en raison de leur **lisibilité** et de leur **facilité d'utilisation**.

Elles **simplifient** grandement le processus **de gestion de la logique asynchrone**, en **évitant** le besoin de **manipuler manuellement les promesses** et les **rappels**.



# Déclaration – Await – Gestion des erreurs : +

```
async function fetchData() {  
  // Code asynchrone à exécuter  
}
```

Déclaration d'une **fonction asynchrone**

```
async function fetchData() {  
  const result = await someAsyncFunction();  
  console.log(result);  
}
```

On peut utiliser le mot-clé **await** devant une promesse pour **attendre que la promesse se résolve ou soit rejetée**.

```
async function fetchData() {  
  try {  
    const result = await someAsyncFunction();  
    console.log(result);  
  } catch (error) {  
    console.error(error);  
  }  
}
```

Les **erreurs** peuvent être **gérées** à l'aide d'un bloc **try/catch** comme dans le **code synchrone**.



# Fonction native fetch() :

Elle est **souvent utilisée** pour **récupérer des ressources** à partir d'APIs, de **serveurs** ou d'autres sources de données sur le Web. **fetch** renvoie une **promesse** qui se résout avec un **objet Response** représentant la **réponse HTTP** de la requête.

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json(); // Renvoie une promesse résolue avec les données
  })
  .then(data => {
    console.log(data); // Traiter les données récupérées
  })
  .catch(error => {
    console.error('There was a problem with the fetch operation:', error);
  });
```




# RESSOURCES




UNE INITIATION À JAVASCRIPT

- Retrouvez ici une collection (**Wakelet**) de sites et/ou ressources associés au langage Javascript.



Christophe Vallot  
@cvallot

17 éléments • Vues • 1 

```
ws.on("message", m => {  
  let a = m.split(" ")  
  switch(a[0]){  
    case "connect":  
      if(a[1]){  
        if(clients.has(a[1])){  
          ws.send("connected");  
          ws.id = a[1];  
        }else{  
          ws.id = a[1]  
          clients.set(a[1], {client: {position: {x: 0, y: 0, id: 0}}})  
          ws.send("connected")  
        }  
      }  
    }else{  
      let id = Math.random().toString().slice(2, 8)  
      ws.id = id;  
      clients.set(id, {client: {position: {x: 0, y: 0, id: 0}}})  
    }  
  }  
})
```

Javascript

Documentation, tutos, librairies, exercices, etc...



### Définitions

#### Variable

Association d'un nom à une valeur. Elle est déclarée via un mot-clé et peut prendre plusieurs formes : texte, nombre, booléen, etc.

#### Objet

Représentation qui se rattache au monde physique : un livre, une page de livre, une lettre.

#### Opérateurs logiques

Caractères spéciaux qui permettent de lier plusieurs conditions ou de comparer des valeurs :

&& || < > >= <= == === != !

#### Interpolation

Méthode qui consiste à entourer une chaîne de caractère avec des backticks pour générer du HTML.

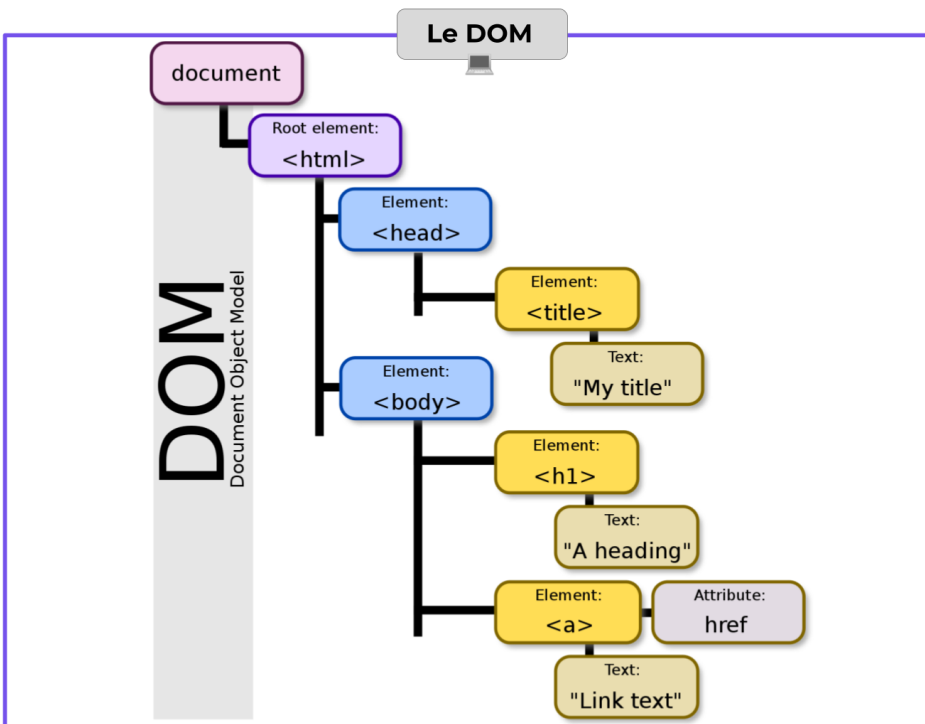
#### Programmation événementielle

Type de programmation où le code est exécuté en réaction à des événements, c'est-à-dire, des actions réalisées par l'utilisateur.

#### Expression régulière

Chaîne de caractère qui décrit, selon une syntaxe précise, un ensemble de chaînes de caractères possibles.

### Le DOM



### Bonnes pratiques

- ✓ Nommer les variables et les fonctions de manière explicite
- ✓ Indenter et commenter son code pour qu'il soit facile à lire
- ✓ Tester son code régulièrement avec console.log
- ✓ Pratiquer avec des projets personnels pour développer son expérience
- ✓ Lire la documentation à disposition

### Erreurs classiques

- ✗ Faire des fonctions trop longues ou qui exécutent trop d'actions
- ✗ Réaliser une opération mathématique sur une chaîne de caractères
- ✗ Ne pas prendre en compte la portée des variables.
- ✗ Ne pas analyser les erreurs dans la console.



+



o



•



# MERCI

Christophe VALLOT