Javascript

La porte d’entrée vers la programmation web

Par **Tiavina Michael Ralainirina**

Version 1.0

Dernière mise à jour le 28/09/2025

Sommaire

A propos de l’auteur

Tiavina Michael Ralainirina est un **développeur logiciel** senior basé à Madagascar. Depuis 2016, il conçoit et développe des sites web, des logiciels ainsi que des applications mobiles. Au cours des huit dernières années, il a collaboré avec plusieurs startups internationales, notamment **FoodChéri**, **Seazon**, **Oscaro Power**, **Yourz** et **xBrain**.

Parallèlement à ses missions professionnelles, il contribue activement au monde du logiciel libre en développant des solutions open source dans le domaine du développement web, comme MUI Tiptap Editor .

En plus de son activité de développeur, André est également écrivain, blogueur et formateur. Il a accompagné et formé des dizaines de stagiaires et de débutants en programmation.

Vous pouvez le suivre sur GitHub ou visiter son site personnel : https://tiavina-michael-ralainirina.onrender.com

Introduction

Dans ce livre, vous apprendrez le langage de programmation JavaScript.

Il est indispensable de maîtriser les bases de l'algorithmique, les fondamentaux de la programmation (variables, fonctions, etc.), ainsi que le HTML et le CSS.

Le code source de ce livre est disponible sur GitHub : https://github.com/tiavina-mika/javascript-for-beginners .

Partie 1 : Introduction à Javascript

Qu'est-ce que Javascript

Javascript est un langage de programmation principalement utilisé pour créer des pages web interactives. Il permet d'ajouter des fonctionnalités dynamiques aux sites web, comme des animations, des formulaires interactifs, et bien plus encore. JavaScript s'exécute côté client, c'est-à-dire dans le navigateur de l'utilisateur, ce qui permet une expérience utilisateur plus fluide et réactive.

En plus de son utilisation dans le développement web, JavaScript est également utilisé côté serveur grâce à des environnements comme Node.js. Cela permet aux développeurs d'utiliser le même langage pour le front-end (interface utilisateur) et le back-end (serveur), facilitant ainsi le développement d'applications complètes.

Comment utiliser JavaScript

Pour exécuter du JavaScript côté client (frontend), on peut l'écrire dans un fichier HTML à l'intérieur d'une balise <script>. Pour l'utiliser côté serveur (backend), il faut installer Node.js et exécuter le fichier avec la commande appropriée.

JavaScript en Frontend (Client Side)

Côté client, JavaScript fonctionne en combinaison avec HTML et CSS.

- HTML donne la structure de la page web,
- CSS lui applique la mise en forme (couleurs, polices, disposition),
- JavaScript la rend interactive en permettant à l'utilisateur d'agir sur les éléments de la page : cliquer sur des boutons, remplir des formulaires, déclencher des animations, etc.

Le code JavaScript côté client est exécuté directement dans le navigateur de l'utilisateur.

On peut l'écrire de 2 façons:

- Directement dans le fichier html

```html
<!DOCTYPE html>
  <html>
  <head>
      <meta charset="UTF-8">
      <title>JS depuis le frontend</title>
  </head>
  <body>
    <h1>Apprendre JavaScript</h1>
    <script>
      console.log("Bonjour depuis le client !");
    </script>
  </body>
</html>
```

Pour tester ce code, enregistrez-le sous le nom *index.html* puis ouvrez-le dans votre navigateur. Ouvrez la console de développement (F12) pour voir le message affiché.

- Dans un fichier .js à part

On peut aussi écrire du JS dans un fichier externe et l'importer dans une page HTML avec la balise *<script>* dans l'attribut **src**, on met le chemin vers le fichier JS (*dossier/nom-fichier.js*)

- *index.html*

```
<!DOCTYPE html>
  <html>
  <head>
      <meta charset="UTF-8">
      <title>JS depuis le frontend</title>
  </head>
  <body>
    <h1>Apprendre JavaScript</h1>
    <script src="index.js"></script>
  </body>
</html>
```

Ici, *index.js* et *index.html* sont dans le même dossier

- *index.js*

```
console.log("Bonjour depuis le navigateur dans un fichier js");
```

JavaScript en Backend (Server Side)

Côté serveur (sur les serveurs web), JavaScript est utilisé pour :

- accéder aux bases de données,
- gérer les fichiers,
- mettre en place des fonctionnalités de sécurité,
- et envoyer des réponses aux navigateurs des utilisateurs.

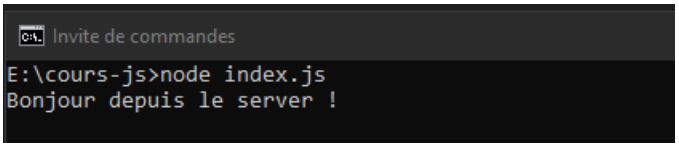
Cela permet de créer des applications web dynamiques et complètes, des deux côtés de la connexion.

Pour exécuter du JavaScript côté serveur, il faut installer Node.js. Node.js permet d'exécuter du code JavaScript en dehors du navigateur, sur le serveur.

Étapes :
1. Installer Node.js depuis le site officiel : https://nodejs.org
2. Créer un fichier nommé *index.js* (dans *E://cours-js/index.js* par exemple) avec le contenu suivant :
3. Ouvrir un terminal et exécuter les commandes :
cd E://cours-js
`node index.js`

```
    console.log("Bonjour depuis le server !");
```

Le message s'affiche dans le terminal.



Conclusion

JavaScript est un langage polyvalent qui fonctionne aussi bien sur le navigateur (client side, frontend) que sur le serveur (server side, backend). Apprendre à l'utiliser dans les deux environnements est essentiel pour développer des applications web complètes.

Partie 2 : Le language Javascript

Les commentaires en javascript

Un commentaire est une note ou une explication ajoutée dans le code source d'un programme informatique. Il est destiné à être lu par les développeurs et n'est pas exécuté par l'ordinateur.

- commentaire sur une ligne

```
// Ceci est un commentaire sur une ligne
console.log("Hello World"); // Ceci est un commentaire à la fin d'une ligne de code
```

- commentaire sur plusieurs lignes

```
/**
    Ceci est un commentaire sur
    plusieurs lignes.
    Il peut s'étendre sur plusieurs lignes.
*/
```

Les logs

Les logs sont des messages affichés dans la **console** du navigateur ou de l'environnement d'exécution (comme **Node.js**) pour aider les développeurs à comprendre ce qui se passe dans leur code.

```
 console.log("Hello World");
```

On utilise les logs pour:

- Afficher des informations dans la console
- Débugger notre code
- Suivre le flux d'exécution
- Afficher des valeurs de variables
- Afficher des messages d'erreur

Les variables

Définition variable

En JavaScript, on peut déclarer des variables de trois façons : **var**, **let**, et **const**.
Ces trois mots-clés permettent de stocker des données, mais ils ne fonctionnent pas exactement de la même manière.

- var – Ancienne méthode

```
 var message = "Bonjour";
console.log(message); // Bonjour
```

- **Portée (scope)** : fonctionnelle, la variable est accessible dans toute la fonction où elle est définie.
- **Problème** : peut causer des bugs car elle peut être redéfinie ou utilisée avant sa déclaration.
- Aujourd'hui, on l'utilise très rarement (plutôt pour du code très ancien).

- let – La plus utilisée

```
let age = 25;
console.log(age); // 25
age = 32;
console.log(age); // 32
```

- **Portée (scope)** : bloc, la variable n'existe que dans le bloc { ... } où elle est définie.
- On peut **réassigner** sa valeur (pratique pour des compteurs, des boucles, etc.).
- C'est la méthode recommandée pour déclarer une variable qui change.

- const – Pour les valeurs qui ne changent pas

```
const pi = 3.14;
console.log(pi); // 3.14
pi = 3.14159; // Erreur : assignment to constant variable
```

- **Portée (scope)** : bloc, comme let.
- On **ne peut pas changer** la valeur après l'avoir définie.
- Idéal pour des constantes ou des objets qui ne doivent pas être réassignés.

⚠ **Attention :**
Si on utilise const avec un objet ou un tableau, on peut toujours modifier **l'intérieur** (les propriétés ou éléments), mais pas réassigner la variable :

```
const fruits = ["Banane", "Orange"];
fruits.push("Pomme"); // autorisé
fruits = ["Fraise"]; // Erreur: Assignment to constant variable.
```

Bonnes pratiques

- Utiliser **const** par défaut
- Passer à **let** seulement si la valeur doit changer (exemple : compteur dans une boucle)
- Éviter **var** : obsolète et peut causer des bugs

Les types de variables en JavaScript

En JavaScript, les variables peuvent contenir plusieurs types de valeurs.
C'est un **langage faiblement typé**, donc une même variable peut changer de type en cours d'exécution.

- Types primitifs

Ce sont les types les plus simples.

Types	Exemples	Description
string	"Bonjour", 'Salut'	Représente du texte. Peut utiliser des guillemets simples, doubles ou backticks.
number	42, -3.14, 0, Infinity	Représente tous les nombres (entiers et décimaux).
bigint	123n, -9007199254740991n	Pour les très grands entiers (au-delà de la limite sûre des nombres).
boolean	true, false	Valeur logique : vrai ou faux.

undefined	undefined	Valeur d'une variable déclarée mais non initialisée.
null	null	Valeur intentionnellement vide ou « absence de valeur ».

```
const nom = "Alice"; // Texte
const age = 25; // Nombre
const estActif = true; // Booléen
const inconnu = null; // Null
let indefini; // Undefined

console.log(typeof nom); // string
console.log(typeof age); // number
console.log(typeof estActif); // boolean
console.log(typeof inconnu); // object (null est un type spécial)
console.log(typeof indefini); // undefined
```

- `typeof` retourne le type primitif ou "`object`" pour les objets et tableaux.
- Les tableaux sont considérés comme des **objets** par `typeof`.

- Types complexes

Ce sont des objets qui peuvent contenir plusieurs valeurs.

Types	Exemple	Description
Object	{ nom: "Alice", age: 30 }	Structure de données qui stocke des paires **clé-valeur**. Très utilisé pour représenter des entités.
Array	[1, 2, 3],["pomme", "banane"]	**Liste ordonnée** d'éléments (qui peuvent être de n'importe quel type). C'est en réalité un type d'objet spécial.
Function	function direBonjour() { return "Salut !"; }	Bloc de code réutilisable. En JavaScript, les fonctions sont des objets de première classe (on peut les stocker dans des variables, les passer en paramètre)

```
// object
const  personne = { nom: "Alice", age: 30 };

// array (tableau)
const personnes = ["Alice", "Bob", "Charlie"];

// function
function saluer() {
    console.log("Bonjour !");
}

// date
const aujourdhui = new Date();

console.log(typeof personne); // object
```

```
console.log(typeof personnes); // object (array est un type spécial d'object)
console.log(typeof saluer); // function
console.log(typeof aujourdhui); // object (date est un type spécial d'object)
```

- Savoir le type des variables avec *typeof*

```
 const nom = "Bob";
console.log(typeof nom); // string

const age = 25;
console.log(typeof age); // number

const estMajeur = true;
console.log(typeof estMajeur); // boolean

const phones = ["iPhone", "Samsung", "Nokia"];
console.log(typeof phones); // object (array is a type of object)

const voiture = { marque: "Toyota", modele: "Corolla" };
console.log(typeof voiture); // object

const variableIndefinie = undefined;
console.log(typeof variableIndefinie); // undefined

const rien = null;
console.log(typeof rien); // object (c'est un bug historique de JS)

function maFonction() { return "Hello"; }
console.log(typeof maFonction); // function
```

- Les opérateurs en JavaScript

Les **opérateurs** permettent de faire des calculs, des comparaisons et de manipuler des valeurs.

- Opérateurs arithmétiques

Utilisés pour faire des calculs :

```
const a = 10;
const b = 5;

console.log(a + b); // 15 (addition)
console.log(a - b); // 5 (soustraction)
console.log(a * b); // 50 (multiplication)
console.log(a / b); // 2 (division)
console.log(a % b); // 0 (reste de la division)
console.log(a ** b); // 100000 (puissance)
```

- Opérateurs d'affectation

Ils servent à **donner une valeur** à une variable, souvent avec un calcul.

```
 let a = 10;
let b = 5;

a += b; // a = a + b
console.log(a); // 15

a -= b; // a = a - b
console.log(a); // 10

a *= b; // a = a * b
console.log(a); // 50

a /= b; // a = a / b
console.log(a); // 10

a %= b; // a = a % b
console.log(a); // 0
```

- Opérateurs de comparaison

Ils renvoient **true** ou **false** selon le résultat.

```
const a = 10;
const b = 5;
const c = '10';

 console.log(a == b);  // false (égalité de valeur)
 console.log(a != b);  // true (différence de valeur)
 console.log(a === c); // false (égalité stricte)
 console.log(a !== c); // true (différence stricte)
 console.log(a > b);   // true (supérieur)
 console.log(a < b);   // false (inférieur)
 console.log(a >= b);  // true (supérieur ou égal)
 console.log(a <= b);  // false (inférieur ou égal)
```

== : compare les valeurs (type ignoré).
=== : compare valeur et type.
!= : différent (valeur).
!== : différent strict (valeur et type).
>, <, >=, <= : comparaison numérique classique.

Bonne pratique : Utiliser === et !== (comparaison stricte) pour éviter les surprises.

1.1.1.1. Opérateurs logiques

Permet d'écrire une condition en une seule ligne.

```
const estMajeur = true;
const aPermis = false;

console.log(estMajeur && aPermis); // false
console.log(estMajeur || aPermis); // true
console.log(!estMajeur); // false
console.log(!aPermis); // true
```

- && : retourne true si les deux conditions sont vraies.
- || : retourne true si au moins une condition est vraie.
- ! : inverse la valeur (true devient false, false devient true).

- Opérateur ternaire

Utilisés pour combiner plusieurs conditions.

```
const person = age >= 18 ? "Majeur" : "Mineur";
console.log(person); // "Majeur"
```

- Opérateurs d'incrémentation et décrémentation

Pratique pour les compteurs :

```
let compteur = 0;

compteur++; // compteur = compteur + 1;
console.log(compteur); // 1

compteur++; // compteur = compteur + 1;
console.log(compteur); // 2

compteur--; // compteur = compteur - 1;
console.log(compteur); // 1

compteur--; // compteur = compteur - 1;
console.log(compteur); // 0
```

++ : ajoute 1 (équivaut à +1)
-- : enlève 1 (équivaut à -1)

Les bases de Javascript

Les conditions (`if`, `else`, `switch`)

Pour exécuter du code en fonction d'une situation.

- . if

Le bloc s'exécute si la condition est vraie.

```
const age = 18;

if (age >= 18) {
    console.log("Majeur"); // Majeur
}
```

- . if … else

Le bloc **else** s'exécute si la condition est **fausse**.

```
 const age = 12;

if (age >= 18) {
    console.log("Majeur");
} else {
    console.log("Mineur"); // Mineur
}
```

- **.** if … else if … else

Permet de tester plusieurs conditions dans l'ordre.

```
const age = 20;

if (age < 18) {
    console.log("Mineur");
} else if (age >= 18 && age < 65) {
    console.log("Adulte"); // Adulte
} else {
    console.log("Senior");
}
```

- . switch

```
 const jour = "mardi";

switch (jour) {
    case "lundi":
        console.log("1er jour de la semaine");
        break;
```

```
    case "mardi":
        console.log("2e jour de la semaine");
        break;
    case "mercredi":
        console.log("3e jour de la semaine");
        break;
    case "jeudi":
        console.log("4e jour de la semaine");
        break;
    case "vendredi":
        console.log("5e jour de la semaine");
        break;
    case "samedi":
        console.log("6e jour de la semaine");
        break;
    case "dimanche":
        console.log("7e jour de la semaine");
        break;
    default:
        console.log("Jour inconnu");
}
```

- `switch` teste la valeur exacte.
- Chaque `case` correspond à une valeur possible.
- `break` empêche l'exécution des cases suivantes.
- `default` s'exécute si aucune case ne correspond.

Les boucles

Pour répéter une action automatiquement.

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

// Nombre: 0
// Nombre: 1
// Nombre: 2
// Nombre: 3
// Nombre: 4
```

- La boucle affiche les valeurs de i de 0 à 4.
- Le commentaire montre toutes les valeurs affichées sur chaque itération.

Les fonctions

Pour éviter de répéter le même code.

```
 function saluer(nom) {
    return "Bonjour " + nom;
}
```

```
console.log(saluer("Alice")); // Bonjour Alice
console.log(saluer("Bob")); // Bonjour Bob
console.log(saluer("Charlie")); // Bonjour Charlie
```

- nom est le paramètre de la fonction.
- return renvoie une valeur qui peut être utilisée ou affichée.
- La fonction peut être appelée plusieurs fois avec différents noms.

Les événements (en frontend)

Pour réagir aux actions de l'utilisateur (clic, saisie, etc.).

```
const button = document.getElementById("myButton");
button.addEventListener("click", function() {
    console.log("Bouton cliqué !");
});
```

- Se déclenche lorsqu'un élément est cliqué.
- Les autres événements sont : `click, mouseover, mouseout, keydown, keyup, input, change, submit`

Les variables avancées

Les chaînes de caractères (String)

- Déclaration d'une string

```
 const nom = "Alice"; // guillemets
const prenom = 'Dupont'; // apostrophes
const message = `Bonjour`; // backticks (guillemets inversées)
```

- Concaténation (assembler du texte)

```
const nom = "Alice";
const prenom = 'Dupont';
const message = `Bonjour, ${nom} ${prenom} !`; // Utilisation de backticks pour l'interpolation
const phrase = "Votre nom est " + nom + " et votre prénom est " + prenom + "."; // Concaténation classique

console.log(message); // Bonjour, Alice Dupont !
console.log(phrase); // Votre nom est Alice et votre prénom est Dupont.
```

Avec les backticks (`), on peut insérer des variables plus simplement :
Plus lisible et plus moderne que la concaténation classique.

- Transformer un String en Array (split)

`split()` permet de **diviser un String** en **Array**, en utilisant un **séparateur**.

```
const phrase = "JavaScript est génial, non ?";
const mots = phrase.split(" "); // Sépare par espace

console.log(mots); // ["JavaScript", "est", "génial,", "non", "?"]

const parties = phrase.split(", "); // Sépare par virgule et espace
console.log(parties); // ["JavaScript est génial", "non ?"]
```

- Propriétés et méthodes utiles sur les strings

Méthode	Description
.length	Retourne le **nombre de caractères** dans la chaîne
.toUpperCase()	Convertit la chaîne en **majuscules**.
.toLowerCase()	Convertit la chaîne en **minuscules**.
.charAt(index)	Retourne le **caractère** à la position donnée.
.indexOf(sousChaine)	Retourne la **position** de la première occurrence de la sous-chaîne, ou **-1** si non trouvée.
.includes(sousChaine)	Retourne **true** si la chaîne contient la sous-chaîne, sinon **false**.

.startsWith(sousChaine)	Retourne **true** si la chaîne commence par la sous-chaîne.
.endsWith(sousChaine)	Retournetruesi la chaîne **se termine par** la sous-chaîne.
.slice(debut, fin)	Extrait **une partie** de la chaîne (la positionfinn'est pas incluse).
.trim()	Supprime les **espaces** au **début** et à la **fin**.
.replace(ancien, nouveau)	Remplace la **première occurrence** de l'**ancien** par **nouveau**.
.replaceAll(ancien, nouveau)	Remplace **toutes les occurrences** (ES2021+).

```
const phrase = " JavaScript est génial";

console.log(phrase.trim()) // "JavaScript est génial"
console.log(phrase.toUpperCase()) // " JAVASCRIPT EST GÉNIAL"
console.log(phrase.toLowerCase()) // " javascript est génial"
console.log(phrase.charAt(1)) // "J"
console.log(phrase.indexOf("génial")) // 16
console.log(phrase.replace("génial", "super")) // " JavaScript est super"
console.log(phrase.length) // 21
console.log(phrase.slice(1, 11)) // "JavaScript"
console.log(phrase.split(" ")) // ['JavaScript', 'est', 'génial']
console.log(phrase.includes("est")) // true
console.log(phrase.startsWith("python")) // false
console.log(phrase.endsWith("génial")) // true
```

Les nombres (Number)

En JavaScript, un **Number** peut être :

- un **entier** (ex. 5),
- un **nombre décimal** (ex. 3.14),
- positif ou négatif.

- Déclaration de nombres

```
 const age = 25; // entier (integer)
const prix = 60.2; // décimal (float)
const temperature = -18; // négatif
const bigNumber = 9007199254741991n; // grand entier (bigint)

console.log(age); // 25
console.log(prix); // 60.2
console.log(temperature); // -18
console.log(bigNumber); // 9007199254741991n
```

- Opérations de base

```
const a = 10;
const b = 5;
const addition = a + b;
const soustraction = a - b;
const multiplication = a * b;
const division = a / b;
const modulo = a % b;

console.log(addition); // 15
console.log(soustraction); // 5
console.log(multiplication); // 50
console.log(division); // 2
console.log(modulo); // 0
```

- Conversion en nombre

Parfois, les valeurs viennent en **texte** et il faut les convertir en nombre

```
 const text = "45";

const nombre1 = Number(text); // conversion explicite
const nombre2 = +text; // conversion explicite

console.log(nombre1); // 45
console.log(nombre2); // 45
```

- Méthodes utiles pour les nombres

```
console.log(Math.round(4.7)); // 5
console.log(Math.floor(4.7)); // 4
console.log(Math.ceil(4.2)); // 5
console.log(Math.max(1, 3, 2)); // 3
console.log(Math.min(1, 3, 2)); // 1
console.log((3.14159).toFixed(2)); // "3.14"
console.log(parseInt("42px")); // 42
console.log(parseFloat("3.14em")); // 3.14
console.log(Math.random()); // Nombre aléatoire entre 0 et 1
console.log(Math.random() * 10); // Nombre aléatoire entre 0 et 10
console.log(Math.PI); // 3.141592653589793
console.log(Math.sqrt(16)); // 4
console.log(Math.pow(2, 3)); // 8 (2^3)
```

Les booléens (*Boolean*)

Un **booléen** est une variable qui ne peut avoir que **deux valeurs** :

- **true** : vrai

- **false** : faux

Ils sont très utiles pour **les conditions** et **les comparaisons**.

- Déclaration de booléens

```
const vrai = true;
const faux = false;

console.log(vrai); // true
console.log(faux); // false
```

- Comparaisons

Les booléens sont souvent le résultat d'une **comparaison**.

```
const a = 10;
const b = 5;

console.log(a > b); // true
console.log(a < b); // false
console.log(a >= b); // true
console.log(a <= b); // false
console.log(a === b); // false

const nom = "Alice";
console.log(nom === "Alice"); // true
console.log(nom !== "Bob"); // true
console.log(nom === "alice"); // false (sensible à la casse)
console.log(nom.includes("A")); // true
console.log(nom.includes("w")); // false
console.log(nom.length > 3); // true
console.log(nom.startsWith("A")); // true
console.log(nom.endsWith("e")); // true

const fruits = ["pomme", "banane", "cerise"];
console.log(fruits.includes("banane")); // true
console.log(fruits.length === 3); // true
```

- Conversion en booléen

On peut convertir une valeur en booléen de deux façons :

- Avec la fonction native Boolean()
- Avec l'opérateur de négation logique ! deux fois (!!)

```
 // Conversion en booléen avec Boolean()
console.log(Boolean(0)); // false
console.log(Boolean(1)); // true
console.log(Boolean("")); // false
console.log(Boolean("Hello")); // true
console.log(Boolean(null)); // false
```

```
console.log(Boolean(undefined)); // false
console.log(Boolean(NaN)); // false

// Conversion en booléen avec !!
console.log(!!0); // false
console.log(!!1); // true
console.log(!!""); // false
console.log(!!"Hello"); // true
console.log(!!null); // false
console.log(!!undefined); // false
console.log(!!NaN); // false
```

- Tester si une variable est boolean

```
const vraiBool = true;
const textBool = "true";
const nombre = 15;

console.log(typeof vraiBool === 'boolean'); // true
console.log(typeof textBool === 'boolean'); // false
console.log(typeof nombre === 'boolean'); // false
```

- Combinaison de Boolean

```
console.log(true && true); // true
console.log(true && false); // false
console.log(false && true); // false
console.log(false && false); // false

console.log(true || true); // treue
console.log(true || false); // true
console.log(false || true); // true
console.log(false || false); // false

console.log(!true); // false
console.log(!false); // true


console.log(5 > 3); // true
console.log(!(5 > 3)); // false
console.log(5 < 3); // false
console.log(5 > 3 && 2 < 4); // true
console.log(5 > 3 || 2 > 4); // true
console.log((5 > 3 && 2 < 4) || (1 === 1)); // true
console.log((5 < 3 && 2 > 4) || (1 !== 1)); // false
```

- Exemples

```
 const estMajeur = age >= 18;
console.log(estMajeur); // true

const asPermis = false;
const peutConduire = estMajeur && asPermis;
console.log(peutConduire); // false

const peutBoire = estMajeur || asPermis;
console.log(peutBoire); // true

const estMineur = !estMajeur;
console.log(estMineur); // false

const estInterdit = !(estMajeur && asPermis);
console.log(estInterdit); // true
```

Les objets (Obejct)

- Déclaration d'un objet

Un **objet** est un type complexe permettant de **regrouper des données** (propriétés) et des **fonctions** (méthodes) sous une seule variable.

```
 const personne = {
    nom: "Alice",
    age: 30,
    majeur: true,
    adresse: {
        rue: "123 Rue Principale",
        codePostal: "75001"
    },
    loisirs: ["lecture", "voyage", "musique"],
    saluer: function() {
        console.log(`Bonjour, je m'appelle ${this.nom} et j'ai ${this.age} ans.`);
    }
}

console.log(personne);
/**
 {
    nom: "Alice",
    prenom: "Dupont",
    age: 30,
    majeur: true,
    adresse: {
        rue: "123 Rue Principale",
        codePostal: "75001"
    },
```

```
    loisirs: ["lecture", "voyage", "musique"],
    saluer: [Function]
}
*/
```

- `nom`, `prenom`, `age`, `majeur` : propriétés
- `saluer()` : méthode (fonction interne à l'objet)

- Accéder aux propriétés

```
console.log(personne.nom); // Alice
console.log(personne["prenom"]); // Dupont
console.log(personne.age); // 30
console.log(personne.majeur); // true
console.log(personne.adresse.rue); // 123 Rue Principale
console.log(personne.loisirs[1]); // voyage
console.log(personne.saluer()); // Bonjour, je m'appelle Alice et j'ai 30 ans.
```

- Modifier une propriété

```
 const pays = {
    nom: "Japon",
    capitale: "Tokyo"
}

pays.nom = "Egypte"; // modifier une propriété

console.log(pays);
// { nom: 'Egypte', capitale: 'Tokyo' }
```

- Ajouter une nouvelle propriété

```
const voiture = {
    marque: "Toyota",
    modele: "Corolla"
}

voiture.annee = 2020; // Ajouter une nouvelle propriété
voiture.couleur = "Rouge"; // Ajouter une nouvelle propriété

console.log(voiture);
// { marque: 'Toyota', modele: 'Corolla', annee: 2020, couleur: 'Rouge' }
```

- Utiliser une méthode de l'objet

```
 const personne = {
```

```
    saluer: function() {
        console.log("Bonjour !");
    }
}

personne.saluer(); // Appel de la méthode saluer de l'objet personne
// "Bonjour !"
```

- Copier un objet

Copier un objet en JavaScript avec la syntaxe de déstructuration `{ ... }`, aussi appelée "spread operator".

- Exemple simple

```
const sport = {
    nom: "Football",
    joueurs: 11,
}

const copieSport = { ...sport };

console.log(sport) // { nom: 'Football', joueurs: 11 }
console.log(copieSport) // { nom: 'Football', joueurs: 11 }
```

Ici, *copieSport* est un nouvel objet, indépendant de *sport*.
Si tu modifies l'un, l'autre ne change pas (pour les propriétés simples).

- Verification d'indépendance

```
copieSport.nom = "Basketball";

console.log(sport) // { nom: 'Football', joueurs: 11 }
console.log(copieSport) // { nom: 'Basketball', joueurs: 11 }
```

Modifier la copie **ne modifie pas** l'original **.**

- ⚠ Attention : c'est une copie superficielle (shallow copy)

Si ton objet contient des objets imbriqués, ils seront référencés, pas copiés.

```
 const sport = {
    nom: "Football",
    joueurs: 11,
    terrain: {
        longueur: 100,
        largeur: 50
    }
}

const copieSport = { ...sport };
```

```
copieSport.terrain.longueur = 120;

console.log(copieSport.terrain.longueur); // 120
console.log(sport.terrain.longueur); // 120 (car copieSport.terrain référence le même objet que sport.terrain)
```

Pour copier en profondeur (deep copy), on peut utiliser :

- *structuredClone(sport)* (solution moderne)
- ou *JSON.parse(JSON.stringify(sport))* (solution classique)

```
const jsonString = JSON.stringify(copieSport);
console.log(jsonString); // '{"nom":"Football","joueurs":11,"terrain":{"longueur":100,"largeur":50}}'
const parsedObject = JSON.parse(jsonString);
console.log(parsedObject); // { nom: 'Football', joueurs: 11, terrain: { longueur: 100, largeur: 50 } }
parsedObject.terrain.longueur = 150;
console.log(parsedObject.terrain.longueur); // 150
console.log(copieSport.terrain.longueur); // 120 (car parsedObject.terrain est un nouvel objet distinct)
```

- Récupérer clés et valeurs d'un objet

```
const ordinateur = {
    marque: "Apple",
    nom: "MacBook Pro",
    annee: 2020,
}
```

- `Object.keys()` : retourne un **Array** contenant toutes les clés.

```
const keys = Object.keys(ordinateur)
console.log(keys);
// ['marque', 'nom', 'annee']
```

- `Object.values()` : retourne un **Array** contenant toutes les valeurs.

```
const values = Object.values(ordinateur);
console.log(values);
// ['Apple', 'MacBook Pro', 2020]
```

- `Object.entries()` : retourne un **Array** de paires [clés, valeur].

```
const entries = Object.entries(ordinateur);
console.log(entries);
// [['marque', 'Apple'], ['nom', 'MacBook Pro'], ['annee', 2020]]
```

Les tableaux (Array)

- Déclaration d'un tableau

```
const fruits = ["Pomme", "Banane", "Cerise"];
console.log(fruits);
// ["Pomme", "Banane", "Cerise"]
```

- Accéder aux éléments

```
console.log(fruits[0]); // Pomme
console.log(fruits[1]); // Banane
console.log(fruits[2]); // Cerise

console.log(fruits[3]); // undefined
```

- Modifier un élément

```
const fruits = ["Pomme", "Banane", "Cerise"];
fruits[1] = "Orange"; // Modifier le 2e élément
console.log(fruits);
// ["Pomme", "Orange", "Cerise"]
```

- Ajouter des éléments

```
 const fruits = ["Pomme", "Banane"];
fruits.push("Orange"); // Ajouter un élément à la fin
console.log(fruits);
// ["Pomme", "Banane", "Orange"]

fruits.unshift("Fraise"); // Ajouter un élément au début
console.log(fruits);
// ["Fraise", "Pomme", "Banane", "Orange"]
```

- Supprimer des éléments

```
const fruits = ["Pomme", "Banane", "Orange"];

fruits.pop(); // Supprime le dernier élément
console.log(fruits);
// ["Pomme", "Banane"]

fruits.shift(); // Supprime le premier élément
console.log(fruits);
```

```
// ["Banane"]
```

- Parcourir un tableau

```
const legumes = ['carotte', 'poireau', 'poivron'];

for (let i = 0; i < legumes.length; i++) {
    console.log(i + ': ' + legumes[i]);
}

// carotte
// poireau
// poivron
```

- Ordonner un tableau (sort())

```
const nombres = [3, 6, 2, 8, 4];

nombres.sort((a, b) => a - b); // Tri croissant
console.log(nombres); // [2, 3, 4, 6, 8]

nombres.sort((a, b) => b - a); // Tri décroissant
console.log(nombres); // [8, 6, 4, 3, 2]
```

- *.sort()* trie les éléments en place (le tableau original est modifié).
- Pour des nombres, il faut passer une fonction de comparaison (a, b) => a - b pour un tri croissant.
- (a, b) => b - a fait un tri décroissant.

Exemple avec un tableau de chaînes de caractères :

```
const legumes = ['tomate', 'salade', 'carotte'];

legumes.sort();
console.log(legumes);
// ['carotte', 'salade', 'tomate']

legumes.reverse(); // Inverse l'ordre des éléments
console.log(legumes);
// ['tomate', 'salade', 'carotte']
```

Pour les chaînes, .sort() trie par ordre alphabétique par défaut.

- Inverser l'ordre des éléments d'un tableau (reverse())

```
const fruits = ["Pomme", "Banane", "Mangue", "Orange"];

fruits.reverse();
console.log(fruits); // ["Orange", "Mangue", "Banane", "Pomme"]
```

- .reverse() modifie directement le tableau original.
- Le premier élément devient le dernier, le deuxième devient l'avant-dernier, etc.
- On peut combiner .sort() et .reverse() pour trier puis inverser l'ordre facilement.

- Copier un tableau

Avec l'opérateur de spread (...)

```
const animaux = ['chien', 'chat', 'hamster', 'perroquet'];

const animauxCopie = [...animaux]; // copie avec spread operator
console.log(animauxCopie);
// ['chien', 'chat', 'hamster', 'perroquet']
```

- Crée un nouveau tableau contenant les mêmes éléments.
- Modifications sur **copieFruits** n'affectent pas fruits.

```
const animaux = ['chien', 'chat', 'hamster'];

const animauxCopie = [...animaux];
animauxCopie.push('poisson');
console.log(animauxCopie); // ['chien', 'chat', 'hamster', 'poisson']
console.log(animaux); // ['chien', 'chat', 'hamster']
```

- Fusionner un tableau

- Avec **concat()** (Ancienne méthode)

```
 const fruits = ['pomme', 'banane', 'orange'];
const legumes = ['oignon', 'carotte', 'concombre'];

const aliments = fruits.concat(legumes);
console.log(aliments);
// ['pomme', 'banane', 'orange', 'oignon', 'carotte', 'concombre']
```

- **.concat()** retourne un nouveau tableau.
- Les tableaux originaux restent inchangés.

- Avec l'opérateur **spread (…)** (Nouvelle méthode)

```
const fruits = ['pomme', 'banane', 'orange'];
const legumes = ['oignon', 'carotte', 'concombre'];
```

```
const aliments = [...fruits, ...legumes]; // fusionner deux tableaux avec spread operator
console.log(aliments);
// ['pomme', 'banane', 'orange', 'oignon', 'carotte', 'concombre']
```

- Très pratique pour fusionner plusieurs tableaux.
- On peut ajouter des éléments au milieu ou au début également :

- Transformer un Array en String (join)

```
const couleurs = ['rouge', 'jaune', 'vert'];

const text1 = couleurs.join(", ");
console.log(text1); // "rouge, jaune, vert"

const text2 = couleurs.join(" - ");
console.log(text2); // "rouge - jaune - vert"

const text3 = couleurs.join("");
console.log(text3); // "rougejaunevert"
```

- Méthodes utiles

- **Length:**

Retourne le nombre d'éléments dans le tableau.

```
const couleurs = ['vert', 'jaune', 'rouge'];
console.log(couleurs.length); // 3
```

- **indexOf()**

Retourne l'**indice** (position) du premier élément trouvé.
Si l'élément n'existe pas, retourne -1.

```
const couleurs = ['vert', 'jaune', 'rouge'];
console.log(couleurs.indexOf('rouge')); // 2
console.log(couleurs.indexOf('bleu')); // -1
```

- *.indexOf("rouge")* : retourne 2 car **"rouge"** est à l'index 2 (les index commencent à 0).
- *indexOf("bleu")* : retourne -1 car **"bleu"** n'existe pas dans le tableau.

- **includes()**

Vérifie si un élément existe dans le tableau. Retourne true ou false.

```
const couleurs = ['vert', 'jaune', 'rouge'];
console.log(couleurs.includes('rouge')); // true
console.log(couleurs.includes('bleu')); // false
```

- *.includes("rouge")* : retourne true car "rouge" est présent dans le tableau.
- *.includes("bleu")* : retourne false car "bleu" n'existe pas dans le tableau.

Différence avec indexOf() :

- `indexOf()` retourne l'index (ou -1 si introuvable).
- `includes()` retourne true ou false directement, ce qui est pratique pour les conditions.

- **push()**

Ajoute un ou plusieurs éléments à la **fin** du tableau.

```
const animaux = ['chien', 'chat'];
animaux.push('hamster'); // ajouter un élément à la fin
console.log(animaux);
// ['chien', 'chat', 'hamster']

console.log(animaux.length); // 3
```

- *.push("Orange")* ajoute "Orange" à la fin du tableau.
- Le tableau est modifié directement (pas de nouveau tableau créé).
- *.length* permet de voir la nouvelle taille du tableau.

- **unshift()**

Ajoute un ou plusieurs éléments **au début** du tableau.

```
const animaux = ['chien', 'chat'];
animaux.unshift('hamster'); // ajoute au début
console.log(animaux);
// ['hamster', 'chien', 'chat']

console.log(animaux.length); // 3
```

- *.unshift("Fraise")* ajoute "Fraise" au début du tableau.
- Le tableau est modifié directement.

- **pop()**

Supprime le **dernier élément** du tableau et le retourne.

```
const animaux = ['lapin', 'ecureuil', 'chat'];
const dernierAnimal = animaux.pop(); // supprime le dernier élément et le retourne
console.log(animaux); // ['lapin', 'ecureuil']
console.log(dernierAnimal); // 'chat'
console.log(animaux.length); // 2
```

- *.pop()* supprime **le dernier élément** du tableau.
- La méthode **retourne** l'élément supprimé.
- Le tableau original est **modifié directement**.

- **shift()**

Supprime le **premier élément** du tableau et le retourne.

```
```

```
const animaux = ['lapin', 'ecureuil', 'chat'];
const premierAnimal = animaux.shift(); // prendre le premier élément et le supprimer
console.log(animaux); // ['ecureuil', 'chat']
console.log(premierAnimal); // 'lapin'
console.log(animaux.length); // 2
```

- *.shift()* supprime le **premier** élément du tableau.
- La méthode **retourne** l'élément supprimé.
- Le tableau original est **modifié directement**.

- **Splice()**

Supprimer, ajouter ou remplacer des éléments dans un tableau.

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

fruits.splice(1, 1);
console.log(fruits); // ['pomme', 'orange']

// supprimer 0 élément à l'index 1 et ajouter 'kiwi' et 'mangue'
fruits.splice(1, 0, 'kiwi', 'mangue');
console.log(fruits); // ['pomme', 'kiwi', 'mangue', 'orange']

// supprimer 1 élément à l'index 2 et ajouter 'fraise'
fruits.splice(2, 1, 'fraise');
console.log(fruits); // ['pomme', 'kiwi', 'fraise', 'orange']
```

- *splice(index, nombreASupprimer, élément1, élément2, ...)*
  - **index** : position de départ
  - **nombreASupprimer** : combien d'éléments supprimer
  - Les éléments suivants sont ajoutés à cette position
- Le tableau original est modifié directement

- **Slice()**

Copier ou extraire une portion d'un tableau

```
const aliments = ['pomme', 'carotte', 'orange'];

const fruits = aliments.slice(0, 2); // copie avec slice
console.log(fruits); // ['pomme', 'carotte']

console.log(aliments); // ['pomme', 'carotte', 'orange']
```

- *slice(debut, fin)* :
  - **debut** : index de départ (inclus)
  - **fin** : index de fin (exclu)
- La méthode ne modifie pas le tableau original.
- Si on met seulement slice(debut), tous les éléments à partir de debut sont copiés

- Exemples

```
const fruits = ['pomme', 'banane', 'orange'];
console.log(fruits.length); // 3
console.log(fruits[0]); // 'pomme'
console.log(fruits[fruits.length - 1]); // 'orange'

fruits[1] = 'poire';
console.log(fruits); // ['pomme', 'poire', 'orange']

fruits.push('raisin');
console.log(fruits); // ['pomme', 'poire', 'orange', 'raisin']

fruits.unshift('fraise');
console.log(fruits); // ['fraise', 'pomme', 'poire', 'orange', 'raisin']

fruits.pop();
console.log(fruits); // ['fraise', 'pomme', 'poire', 'orange']

fruits.shift();
console.log(fruits); // ['pomme', 'poire', 'orange']

const copieFruits = [...fruits];
console.log(copieFruits); // ['pomme', 'poire', 'orange']
fruits[0] = 'avocat';
fruits[1] = 'cerise';
console.log(fruits); // ['avocat', 'cerise', 'orange']

const tousLesFruits = [...fruits, ...copieFruits]
console.log(tousLesFruits); // ['avocat', 'cerise', 'orange', 'pomme', 'poire', 'orange']

const fruitsString = tousLesFruits.join(', ');
console.log(fruitsString); // 'avocat, cerise, orange, pomme, poire, orange'

const aOrange = tousLesFruits.includes("orange");
console.log(aOrange); // true

const indexCerise = tousLesFruits.indexOf("cerise");
console.log(indexCerise); // 1
```

- Exemple complexe

- Grouper une liste par une propriété

```
const aliments = [
    { nom: "banane", type: "fruits", poids: 120 },
    { nom: "carotte", type: "légumes", poids: 80 },
    { nom: "poulet", type: "viande", poids: 200 },
    { nom: "pomme", type: "fruits", poids: 100 },
```

```
]

const alimentsParType = {}

for (const aliment of aliments) {
    const type = aliment.type
     // si le type n'existe pas encore dans l'objet alimentsParType
    if (!alimentsParType[type]) {
        alimentsParType[type] = []
    }

    // on ajoute l'aliment dans le tableau correspondant à son type
    alimentsParType[type].push(aliment)
}

console.log(alimentsParType)
// {
//   fruits: [ { nom: 'banane', type: 'fruits', poids: 120 }, { nom: 'pomme', type: 'fruits', poids: 100 } ],
//   légumes: [ { nom: 'carotte', type: 'légumes', poids: 80 } ],
//   viande: [ { nom: 'poulet', type: 'viande', poids: 200 } ]
// }
```

- Exemple avec les méthodes modern de boucles

```
 const aliments = [
    { nom: "banane", type: "fruits", prix: 5000 },
    { nom: "carotte", type: "légumes", prix: 3000 },
    { nom: "poulet", type: "viande", prix: 15000 },
    { nom: "pomme", type: "fruits", prix: 4000 },
]

const fruits = aliments.filter((aliment) => aliment.type === "fruits")
console.log(fruits) // [{ nom: "banane", type: "fruits", prix: 5000 }, { nom: "pomme", type: "fruits", prix: 4000 }]

const banane = aliments.find((aliment) => aliment.nom === "banane")
console.log(banane) // { nom: "banane", type: "fruits", prix: 5000 }

const nomsAliments = aliments.map((aliment) => aliment.nom)
console.log(nomsAliments) // ["banane", "carotte", "poulet", "pomme"]

const totalPrix = aliments.reduce((total, aliment) => total + aliment.prix, 0)
console.log(totalPrix) // 27000

const nomsAliments2 = []
aliments.forEach((aliment) => nomsAliments2.push(aliment.nom))
console.log(nomsAliments2) // ["banane", "carotte", "poulet", "pomme"]

const fruitsMoinsChers = aliments
    .filter((aliment) => aliment.type === "fruits") // On garde que les fruits
    .sort((a, b) => a.prix - b.prix) // On trie par prix croissant
```

```
    .map((aliment) => aliment.nom) // On garde que le nom
console.log(fruitsMoinsChers) // ["pomme", "banane"]

const fruitsCommenceParP = []
for (const aliment of aliments) {
    if (aliment.type === "fruits" && aliment.nom.startsWith("p")) {
        fruitsCommenceParP.push(aliment.nom)
    }
}
console.log(fruitsCommenceParP) // ["pomme"]
```

Les dates

```
const maintenant = new Date();
console.log(maintenant); // Affiche la date et l'heure actuelles (ex. 2024-01-15T14:30:45.123Z)
console.log(maintenant.getFullYear()); // Affiche l'année actuelle (ex. 2024)
console.log(maintenant.getMonth() + 1); // Affiche le mois actuel (1-12) (ex. 1 pour janvier)
console.log(maintenant.getDate()); // Affiche le jour du mois actuel (1-31) (ex. 15)
console.log(maintenant.getHours()); // Affiche l'heure actuelle (0-23) (ex. 14 pour 14h)
console.log(maintenant.getMinutes()); // Affiche les minutes actuelles (0-59) (ex. 30)
console.log(maintenant.getSeconds()); // Affiche les secondes actuelles (0-59) (ex. 45)
console.log(maintenant.toLocaleDateString()); // Affiche la date au format local (ex. 15/01/2024)

const dateSpecifique = new Date('2023-12-25T10:00:00');
console.log(dateSpecifique); // Affiche la date spécifique (ex. 2023-12-25T10:00:00.000Z)
console.log(dateSpecifique.toLocaleDateString()); // Affiche la date spécifique au format local (ex. 25/12/2023)
```

Les Fonctions avancées

Une fonction, c'est un bloc de code qu'on peut réutiliser

Déclaration de fonction (Function Declaration)

C'est la façon la plus classique de créer une fonction.

```
function direBonjour(nom) {
    return "Bonjour " + nom + " !";
}

// Appel de la fonction
console.log(direBonjour("Alice")); // Bonjour Alice !
console.log(direBonjour("Bob"));   // Bonjour Bob !
console.log(direBonjour()); // Bonjour undefined !
```

- *function nomFonction(paramètres) { ... }* définit une fonction.
- **return** permet de renvoyer une valeur.
- La fonction peut être appelée plusieurs fois avec différents arguments.

Expression de fonction (Function Expression, rarement utilisée)

Ici la fonction est stockée dans une variable.
Elle ne peut pas être appelée avant sa déclaration.

```
const direBonsoir = function(nom) {
    return `Bonsoir, ${nom} !`;
}

console.log(direBonsoir("Alice")); // Bonsoir, Alice !
console.log(direBonsoir("Bob"));   // Bonsoir, Bob !
```

Fonction fléchée (Arrow Function)

Syntaxe plus courte, simple et moderne.

```
const somme = (a, b) => a + b;

console.log(somme(5, 3)); // 8
console.log(somme(10, 15)); // 25
console.log(somme(-2, 7)); // 5
```

Note :

```
const somme = (a, b) => a + b;
console.log(somme(5, 3)); // 8

// Equivalent avec return
const somme = (a, b) => {
    return a + b;
```

```
}
console.log(somme(5, 3)); // 8
```

Fonction anonyme (Anonymous Function)

C’est une fonction sans nom, souvent utilisée comme callback.

```
setTimeout(function() {
    console.log("Ceci s'affiche après 2 secondes");
}, 2000);
// Ceci s'affiche après 2 secondes
```

Utilisation d’une fonction

En JavaScript, une fonction peut retourner une valeur grâce au mot-clé `return`… ou ne rien retourner, auquel cas elle renvoie undefined par défaut

- Fonction qui retourne une valeur

```
const multiplier = (a, b) => a * b;
const resultat = multiplier(5, 3);
console.log(resultat); // 15
```

La fonction envoie un résultat qui peut être stocké ou utilisé dans d'autres calculs.

```
const multiplier = (a, b) => a * b;
const somme = (a, b) => a + b;

const resultat1 = multiplier(2, 3);
console.log(resultat1); // 6

const resultat2 = somme(4, 4);
console.log(resultat2); // 8

const total = resultat1 + resultat2;
console.log(total); // 14
```

- Fonction qui ne retourne rien

```
 const afficherMessage = (message) => {
    console.log("message: " + message);
    // pas de return => return undefined
}

afficherMessage("Bonjour tout le monde !");
// message: Bonjour tout le monde !
```

```
const texte = afficherMessage("Bonjour !");
// message: Bonjour !
console.log(texte);
// undefined
```

Ici, la fonction effectue une action (affichage) mais ne donne aucune valeur exploitable.

- Fonction avec paramètres par défaut

```
function saluer(nom = "Invité") {
    return `Bonjour, ${nom} !`;
}

console.log(saluer()); // Bonjour, Invité !
console.log(saluer("Alice")); // Bonjour, Alice !
console.log(saluer("Bob")); // Bonjour, Bob !
```

Ici la fonction saluer prend un paramètre nom avec une valeur par défaut "Invité".
Si aucun argument n'est passé lors de l'appel de la fonction, elle utilise cette valeur par défaut.

- Fonction avec paramètres obligatoires

```
const somme = (a, b) => a + b;

console.log(somme(3, 4)); // 7
console.log(somme(10)); // NaN
console.log(somme()); // NaN
```

- **a** et **b** n'ont pas de valeur par défaut, ils sont obligatoires
- si on ne passe pas les arguments, ils seront *undefined*
- et *undefined* + *undefined* = *NaN* (Not a Number)

- Fonction avec paramètres obligatoires ET par défaut

```
function surfaceRectangle(longueur, largeur = 1) {
    return longueur * largeur;
}

console.log(surfaceRectangle(5, 3)); // 15
console.log(surfaceRectangle(4)); // 4 = 4 * 1
console.log(surfaceRectangle()); // NaN
```

- **longueur** est obligatoire et **largeur** par défaut
- Pour éviter *NaN*, on peut ajouter une vérification

- Exemples

- Fonction avec conditions

```
```

```
function surfaceRectangle(longueur, largeur = 1) {
    if (longueur <= 0 || largeur <= 0) {
        console.error("Les dimensions doivent être positives.");
        return null;
    }

    if (typeof longueur !== 'number' || typeof largeur !== 'number') {
        console.error("Les dimensions doivent être des nombres.");
        return null;
    }

    return longueur * largeur;
}

console.log(surfaceRectangle(5, 3)); // 15
console.log(surfaceRectangle(4)); // 4
console.log(surfaceRectangle(-2, 3)); // Erreur
console.log(surfaceRectangle(4, 'a')); // Erreur
```

- Fonction avec différents types d'arguments

```
const trouverPersonneParId = ({
    persons = [],
    id
}) => {
    const personne = persons.find(p => p.id === id);
    return personne || null;
};

const modifierPersonne = (personne) => {
    if (!personne) {
        console.error("Personne non trouvée");
        return;
    }

    // Bonnes pratiques: ne pas modifier l'objet original, mais créer une copie
    const copiePersonne = { ...personne };

    copiePersonne.nom = "Nom Modifié";

    return copiePersonne;
}

const personnes = [
    { id: 1, nom: "Alice" },
    { id: 2, nom: "Bob" },
    { id: 3, nom: "Charlie" }
];

const personneTrouvee = trouverPersonneParId({ persons: personnes, id: 2 });
```

```
const personneTrouveeModifiee = modifierPersonne(personneTrouvee);

 console.log(personneTrouvee); // { id: 2, nom: "Bob" }
console.log(personneTrouveeModifiee); // { id: 2, nom: "Nom Modifié" }
console.log(personnes); // [{ id: 1, nom: "Alice" }, { id: 2, nom: "Bob" }, { id: 3, nom: "Charlie" }]

const personneNonTrouvee = trouverPersonneParId({ persons: personnes, id: 5 }); // null
const personneNonTrouveeModifiee = modifierPersonne(personneNonTrouvee); // Affiche une erreur: Personne non trouvée
```

Les boucles avancées

Méthodes classiques

- . for

```
for (initialisation; condition; incrémentation) {
    // Code à exécuter à chaque itération
}
```

- **initialisation** : définie avant de commencer la boucle (ex. compteur).
- **condition** : la boucle continue tant que cette condition est vraie.
- **incrémentation** : modifie le compteur à chaque tour (ex. i++).

```
for (let i = 1; i <= 3; i++) {
    console.log(`Itération numéro ${i}`);
}

// Itération numéro 1
// Itération numéro 2
// Itération numéro 3
```

- **let i = 0** : compteur commence à 0.
- **i < 3** : la boucle s'exécute tant que i est inférieur à 3.
- **i++** : incrémente i de 1 à chaque tour.

- . for … of

Permet de **parcourir les éléments** d'un tableau directement.

```
const phones = ["iPhone", "Samsung"];

for (const phone of phones) {
    console.log(phone);
}

// iPhone
// Samsung
```

- . for … in

Parcourir **les clés (ou index) d'un objet ou d'un tableau**.

.1 Sur un array

Dans un tableau, cela permet de récupérer chaque index, puis d'accéder à l'élément avec tableau[index].

```
const phones = ["iPhone", "Samsung"];

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

```
// 0 : iPhone
// 1 : Samsung
```

- **index** : prend les valeurs "0", "1" (les positions du tableau).
- **phones[index]** : permet d'obtenir la marque correspondante.

.2 Sur un object

```
const pays = {
    nom: "Madagascar",
    capitale: "Antananarivo",
};

for (const key in pays) {
    console.log(key + " : " + pays[key]);
}
// nom : Madagascar
// capitale : Antananarivo
```

- . while

Exécute une boucle tant qu'une condition est vraie. (Rarement utilisé)

```
const fruits = ["pomme", "banane", "cerise"];
let i = 0;
while (i < fruits.length) {
    console.log(`${i} : ${fruits[i]}`);
    i++;
}

// 0 : pomme
// 1 : banane
// 2 : cerise
```

- . do … while

Similaire à `while`, mais s'exécute au moins une fois avant de vérifier la condition. (Rarement utilisé)

```
const fruits = ["pomme", "banane", "cerise"];
let i = 0;
while (i < fruits.length) {
    console.log(`${i} : ${fruits[i]}`);
    i++;
}

// 0 : pomme
// 1 : banane
// 2 : cerise
```

Méthodes plus modernes

- . forEach()

Parcourt chaque élément d'un tableau avec une fonction callback.

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

fruits.forEach((fruit, index) => {
    fruits[index] = fruit.toUpperCase();
});

console.log(fruits); // ["POMME", "BANANE", "CERISE"]
```

- **fruit :** contient l'élément actuel.
- **Index :** contient la position de l'élément dans le tableau.
- La fonction fléchée (fruit, index) => { ... } est exécutée pour chaque élément.
- IMPORTANT
  - La méthode forEach modifie directement le tableau original.
  - Si on veut garder le tableau original intact, on peut utiliser map à la place, ou on crée une copie du tableau avant de le modifier.

- . map()

Retourne un nouveau tableau après transformation des éléments.

```
 const ordinateurs = ["Dell", "Asus", "Acer"];

const ordinateursMajuscules = ordinateurs.map(ordi => ordi.toUpperCase());

console.log(ordinateursMajuscules); // ["DELL", "ASUS", "ACER"]
console.log(ordinateurs); // ["Dell", "Asus", "Acer"]
```

- **ordinateur** : représente chaque élément du tableau.
- La fonction transforme chaque nom en majuscules.
- `map()` retourne un nouveau tableau avec les résultats.

Note :

- L'**index** est optionnel dans map().
- map() est pratique pour transformer un tableau sans le modifier directement.
- Si aucune transformation n'est nécessaire, `map()` n'est pas utile (utiliser `forEach()` ou `for...of`).

- . filter()

Crée un nouveau tableau contenant seulement les éléments qui passent un test.

```
const ordinateurs = ["Dell", "Asus", "Acer"];

const ordinateursAvecA = ordinateurs.filter(ordi => ordi.toLowerCase().includes("a"));
console.log(ordinateursAvecA); // ["Asus", "Acer"]

console.log(ordinateurs); // ["Dell", "Asus", "Acer"]
```

- Seuls les éléments qui ont un nom avec un « a » sont conservés.
- `filter()` retourne un nouveau tableau.

- . find()

Retourne le premier élément qui satisfait une condition.

```
const ordinateurs = ["Dell", "Asus", "Acer"];

const ordinateurAvecA = ordinateurs.find(ordi => ordi.toLowerCase().includes("a"));
console.log(ordinateurAvecA); // "Asus"

console.log(ordinateurs); // ["Dell", "Asus", "Acer"]
```

- La fonction de test *ordi => ordi.includes("a")* renvoie **true** pour le premier élément correspondant.
- `find()` retourne directement cet élément.
- Si aucun élément ne correspondait, le résultat aurait été *undefined*.

- . reduce()

Réduire un tableau à une seule valeur

```
const prix = [10, 20, 30, 40, 50];
const totalPrix = prix.reduce((accumulateur, valeurCourante) => accumulateur + valeurCourante, 0);
console.log(totalPrix); // 150
```

- **.reduce(callback, valeurInitiale)**
  - callback(accumulateur, valeurCourante) : fonction appliquée à chaque élément
  - valeurInitiale : valeur de départ de l'accumulateur
- Dans l'exemple, accumulateur commence à 0 et additionne chaque élément du tableau.

- .every()

Teste si tous les éléments du tableau passent le test implémenté par la fonction fournie. Elle retourne un **booléen** (true ou false).

```
const nombres = [5, 10, 15, 20];

const tousSupA5 = nombres.every(nombre => nombre > 6);
console.log(tousSupA5); // false, car 5 n'est pas > 6

const fruits = ["ananas", "banane", "orange"];
const tousContiennentA = fruits.every(fruit => fruit.includes('a'));
console.log(tousContiennentA); // true, toutes les chaînes contiennent 'a'
```

- .some()

Teste si au moins un élément du tableau passe le test implémenté par la fonction fournie. Renvoie un **booléen**.

```
const nombres = [10, -5, 8, -3, 2];
const contientNegatif = nombres.some(nombre => nombre < 0);
```

```
console.log(contientNegatif); // true (car -5 et -3 sont négatifs)

const fruits = ["pomme", "banane", "orange"];
const contientBanane = fruits.some(fruit => fruit.startsWith('fraise'));
console.log(contientBanane); // false (car "fraise" n'est pas dans le tableau)
```

Tableau d'objet

- Déclaration d'un tableau d'objets

```
const phones = [
    { marque: "Samsung", prix: 899 },
    { marque: "Apple", prix: 999 },
    { marque: "Xiaomi", prix: 499 }
]
```

- Accéder à un élément spécifique

```
const phones = [
    { marque: "Samsung", prix: 899 },
    { marque: "Apple", prix: 999 },
    { marque: "Xiaomi", prix: 499 }
]

console.log(phones[0]); // { marque: "Samsung", prix: 899 }
console.log(phones[1]); // { marque: "Apple", prix: 999 }
console.log(phones[2]); // { marque: "Xiaomi", prix: 499 }
```

- Parcourir le tableau d'objets

```
const phones = [
    { marque: "Samsung", prix: 899 },
    { marque: "Apple", prix: 999 },
    { marque: "Xiaomi", prix: 499 }
]

for (const phone of phones) {
    console.log(`${phone.marque} - ${phone.prix}€`);
}

// Samsung - 899€
// Apple - 999€
// Xiaomi - 499€
```

- Méthode map pour transformer un tableau d'objets

```
const phones = [
    { marque: "Samsung", prix: 899 },
```

```
    { marque: "Apple", prix: 999 },
    { marque: "Xiaomi", prix: 499 }
]

const phonesEnDollars = phones.map(phone => phone.prix * 1.1);
console.log(phonesEnDollars); // [988.9, 1098.9, 548.9]
```

- Méthode filter pour sélectionner certains objets

```
 const phones = [
    { marque: "Samsung", prix: 899 },
    { marque: "Apple", prix: 999 },
    { marque: "Xiaomi", prix: 499 }
]

const phonePlusChers = phones.filter(phone => phone.prix > 500);
console.log(phonePlusChers);
// [{ marque: "Samsung", prix: 899 }, { marque: "Apple", prix: 999 }]
```

Partie 3 : Les fonctionnalités natives de Javascript

JSON (convertir objets en texte et inversement)

Pratique pour stocker ou échanger des données.

```
const personne = {
    nom: "Doe",
    prenom: "John"
}

const text = JSON.stringify(personne);
console.log(text); // '{"nom":"Doe","prenom":"John"}'
console.log(typeof text); // "string"

const obj = JSON.parse(text);
console.log(obj); // { nom: 'Doe', prenom: 'John' }
console.log(typeof obj); // "object"
```

setTimeout et setInterval (temporisation)

Exécuter du code après un certain temps ou de façon répétée.

```
setTimeout(() => {
    console.log("Hello après 1 secondes");
}, 1000);

// Hello après 1 secondes
```

```
let compteur = 0;
const interval = setInterval(() => {
    compteur++;
    console.log(compteur);
}, 1000);

// Stopper après 5 secondes
setTimeout(() => {
    clearInterval(interval);
    console.log("Intervalle arrêté");
}, 3000);

// 0
// 1
// 2
// Intervalle arrêté
```

console

L'objet utilisé pour afficher des messages dans la console du navigateur.

```
console.log("Ceci est un message de log.");
console.warn("Ceci est un avertissement.");
console.error("Ceci est un message d'erreur.");
console.table([{ nom: "Alice", age: 25 }, { nom: "Bob", age: 30 }]); // Affiche un tableau dans la console
```

Les API natives du navigateur

JavaScript ne se limite pas au langage de programmation en lui-même (variables, boucles, fonctions, etc.). Lorsqu'il est exécuté dans un navigateur web, il a accès à un ensemble d'objets globaux et d'API fournies par l'environnement du navigateur. Ces fonctionnalités permettent d'interagir avec la page web, l'utilisateur, le stockage local, et bien plus encore.

window

L'objet global qui représente la fenêtre du navigateur.
Il contient des propriétés et des méthodes pour contrôler la fenêtre du navigateur.

```
console.log(window.innerWidth); // Largeur de la fenêtre
console.log(window.innerHeight); // Hauteur de la fenêtre
window.alert("Bonjour!"); // Affiche une alerte
console.log(window.location.href); // URL actuelle
window.location.href = "https://www.example.com"; // Redirige vers une nouvelle URL
window.open("https://www.example.com", "_blank"); // Ouvre une nouvelle fenêtre ou un nouvel onglet
window.close(); // Ferme la fenêtre actuelle (ne fonctionne que pour les fenêtres ouvertes par script)
```

document

L'objet qui représente le document HTML chargé dans la fenêtre.
Il permet d'accéder et de manipuler le contenu de la page (DOM).

```
console.log(document.title); // Titre du document
console.log(document.URL); // URL du document
console.log(document.body); // Corps du document
const newElement = document.createElement("p");
newElement.textContent = "Ceci est un nouvel élément paragraphe.";
document.body.appendChild(newElement); // Ajoute le nouvel élément au corps du document

document.getElementById("monBoutonId").addEventListener("click", function() {
    alert("Bouton cliqué!");
});
```

LocalStorage

Objets pour stocker des données localement dans le navigateur.
localStorage conserve les données même après la fermeture du navigateur.

- Les propriétés de localStorage

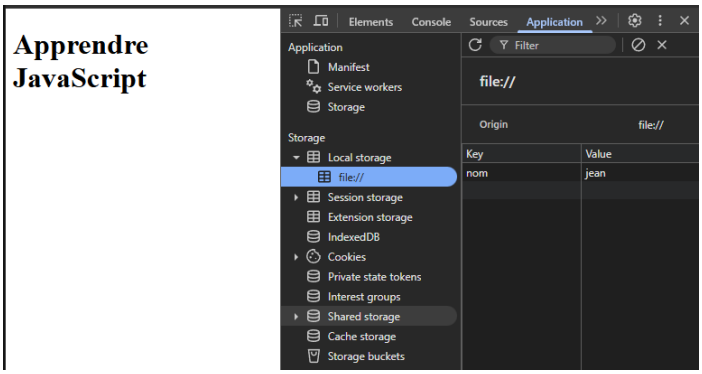
- setItem

Stocke une paire clé-valeur dans le stockage local du navigateur.

```
localStorage.setItem("nom", "Jean");
```

Pour le voir les données stockées dans localStorage :

- Pour le voir les données stockées dans localStorage :
- Ouvrez la console de votre navigateur (F12 ou clic droit -> Inspecter -> Console).
- Cliquez sur l'onglet "**Application**" (ou "Stockage" dans certains navigateurs).
- Recherchez la section "**Local Storage**" pour voir les données stockées.
- Vous verrez key: **nom**, value: **Jean** dans un tableau.
- Pour supprimer la clé "**nom**", vous pouvez utiliser *localStorage.removeItem("nom");* dans la console.
- Ou cliquez droit sur la clé "nom" dans l'interface et sélectionnez "**Supprimer**".



- getItem

Récupère la valeur associée à une clé spécifique dans le stockage local.

```
const nom = localStorage.getItem("nom");
console.log(nom); // Jean
```

- removeItem

Supprime une paire clé-valeur spécifique du stockage local.

```
localStorage.removeItem("nom");
console.log(localStorage.getItem("nom")); // null
```

- Que stocké dans le localStorage

- Le localStorage stocke des données sous forme de paires clé-valeur, où les deux sont des chaînes de caractères (strings).
- Vous pouvez stocker des informations telles que les préférences utilisateur, les paramètres d'application, les données de session, etc.
- Par exemple, on peut stocker le nom d'utilisateur, le thème choisi (clair/sombre), ou même des objets JSON sérialisés en chaînes de caractères.

History API

Permet de manipuler l'historique de navigation.

```
 history.back(); // Va à la page précédente
history.forward(); // Va à la page suivante
history.go(-1); // Va à la page précédente
history.go(1); // Va à la page suivante
```

Partie 4 : Bonnes pratiques

Langues

Toujours utiliser l'anglais dans le code :

- Nom de fichier
- Nom de variables
- Nom de fonctions
- Commentaires

Exemple :

```
// Nom fichier : userProfile.js
// variable
const userName = "Alice";

// fonction
const getUserInfo = () => {
    //
}

// constante
const API_URL = "https://api.example.com/user";

// commentaire
// Get user information from API
```

Nommage des variables

- Nommer avec un ou plusieurs mots en **camelCase**
- Si c'est un booléen, commencer par **is**, **has**, **can**, **should**

```
const name = "John";
const age = 30;
const cars = [
    { brand: "Ford", model: "Mustang", year: 1969 },
    { brand: "Tesla", model: "Model 3", year: 2020 },
    { brand: "Toyota", model: "Corolla", year: 2018 }
];
const isAdmin = true;
```

Nommage des fonctions

- Nommer comme une **action/phrase** en **camelCase**
- Commencer par un verbe

```
 const getUserInfo = (name) => {
    return { name }
}
```

```
const updateUserInfo = (user) => {
    user.lastUpdated = new Date().toLocaleTimeString();
    return user;
}

const displayUserInfo = (user) => {
    console.log(`Nom: ${user.name}`);
    console.log(`Dernière mise à jour: ${user.lastUpdated}`);
}
```

Nommage des constantes

- Toujours utiliser `const`
- En **MAJUSCULES** avec _ comme séparateur

```
 const MAX_USERS = 10;
```

Les differents Naming styles

```
// camel case (variables, fonctions)
const userName = "Alice";
const hasAccess = true;

// snake case (rare en JavaScript, mais parfois utilisé dans les bases de données)
const user_name = "Bob";
const has_access = false;

// kebab case (pas utilisable dans les variables JavaScript, uniquement pour les noms de fichiers ou CSS)
// const user-name = "Charlie"; // Invalid in JavaScript

// Pascal case (utilisé souvent pour les noms de classes)
class UserAccount {
    constructor(name) {
        this.name = name;
    }
}
```

Écrire du code lisible

- Indentation cohérente (2 ou 4 espaces)
- Lignes pas trop longues (max ~80-100 caractères)
- Espaces autour des opérateurs

```
const total = price + taxe; // ❌
const total = price + taxe; // ✅
```

Éviter le code dupliqué

Créer des fonctions réutilisables

```
const calculateTax = (amount, tax) => amount * tax;

calculateTax(1000, 0.2); // 200
calculateTax(500, 0.1); // 50
```

Ajouter des commentaires utiles

- Pas de commentaires évidents
- Expliquer pourquoi, pas ce que fait le code

```
// ✅ Explain why
// Retry API request until success
if (!connected) tryRequest();

// ❌ Bad: Just restate the code
// Increment counter
counter++
```

Partie 5 : Ressources Javascript

Ce livre vous donne les bases solides et les techniques pratiques nécessaires pour bien démarrer votre parcours en JavaScript. Cependant, il ne peut pas couvrir l'ensemble des sujets liés à JavaScript. La programmation JavaScript regorge de notions à explorer, et cette annexe vous indique des pistes pour poursuivre votre apprentissage et approfondir vos connaissances.

Référence

Il arrive qu'on ait besoin d'un dictionnaire pour comprendre un livre. De la même façon, en programmation JavaScript, il est précieux de pouvoir consulter une référence complète couvrant les mots-clés, les concepts, les méthodes et les différentes composantes de la syntaxe du langage. De telles ressources existent à la fois sous forme de livres et sur le web.

Sites web

- MDN Web Docs (https://developer.mozilla.org/fr/docs/Web/JavaScript)
- MDN Web Docs - API (https://developer.mozilla.org/fr/docs/Web/API)
- Tutoriel JavaScript Moderne (https://javascript.info/)

Livres

- Apprenez à programmer avec JavaScript (https://openclassrooms.com/fr/courses/7696886-apprenez-a-programmer-avec-javascript#table-of-content)

HTML et CSS

En abordant ce livre, vous maîtrisez sans doute déjà les bases de HTML et CSS.
JavaScript sait exploiter toute la richesse du CSS, aussi bien pour styliser les éléments que pour les animer dynamiquement sur la page.
Si vous souhaitez réviser ou consolider vos connaissances en HTML et CSS, voici quelques ressources pratiques.

Sites web

- MDN Web Docs - HTML (https://developer.mozilla.org/fr/docs/Web/HTML)
- MDN Web Docs - CSS (https://developer.mozilla.org/fr/docs/Web/CSS)
- W3Schools - HTML (https://www.w3schools.com/html/)
- W3Schools - CSS (https://www.w3schools.com/css/)
- Flex playground (https://flexbox.tech/)

Livres

OpenClassrooms - Créez votre site web avec HTML5 et CSS3 (https://openclassrooms.com/fr/courses/1603881-creez-votre-site-web-avec-html5-et-css3)

Conclusion

Félicitations ! Vous venez de franchir une étape essentielle dans votre parcours de développeur·euse. JavaScript, bien plus qu'un simple langage de script, est aujourd'hui l'un des piliers du développement web — et même au-delà, avec les applications mobiles, les serveurs (Node.js), les interfaces de bureau, et l'intelligence artificielle.

Vous avez maintenant les bases solides : variables, fonctions, objets, gestion des événements, interaction avec le DOM, et même les API du navigateur. Mais JavaScript est un langage vivant, en constante évolution. Ce que vous avez appris ici n'est pas une fin, mais un tremplin.

Continuez à expérimenter.
Cassez du code, réparez-le, poussez vos projets plus loin.
Explorez les frameworks modernes (React, Vue, Svelte…), plongez dans les bonnes pratiques (ESLint, modules, asynchronisme), et n'oubliez jamais de lire la documentation officielle.