# Fondamentaux JavaScript

## Introduction

### Qu'est-ce que JavaScript?

JavaScript est un **langage de programmation** qui permet d'ajouter de l'**interactivité** et de la **dynamique** aux sites Web.

Le JavaScript est utilisé principalement pour enrichir l'expérience utilisateur et se charge côté client (navigateur).

### Histoire

JavaScript a été créé en **1995** par **Brendan Eich** pour le navigateur **Netscape**.

### Relation avec HTML et CSS

- **HTML** : structure de la page web
- **CSS** : style et mise en forme
- **JavaScript** : comportement et interactivité

### Plateformes compatibles

JavaScript est compatible avec un grand nombre de **plateformes** et **environnements**.

- **Navigateurs Web** : Chrome, Firefox, Safari, etc
- **Serveurs** : grâce à Node.js
- **Applications mobiles** : en utilisant React Native, Ionic, etc

### Navigateurs

JavaScript est pris en charge par tous les **navigateurs modernes**.

| Navigateurs | Support de JavaScript |
| --- | --- |
| Google Chrome | Oui |
| Firefox | Oui |
| Safari | Oui |
| Microsoft Edge | Oui |

Pour les anciennes versions de navigateurs, certaines fonctionnalités de JavaScript pourraient ne pas être supportées.

### Serveurs (Node.js)

**Node.js** est un environnement d'exécution de **JavaScript** côté serveur.

Node.js est construit sur le moteur JavaScript V8 de Chrome et permet de créer des applications performantes et évolutives.

## Variables et Types de données

### Déclaration de variables

En JavaScript, on peut déclarer des variables à l'aide de trois mots-clés: `var`, `let` et `const`.

| Mot-clé | Description |
| --- | --- |
| **var** | Une variable déclarée avec `var` a une portée de fonction |
| **let** | Une variable déclarée avec `let` a une portée de bloc |

| | |
|---|---|
| **const** | Une variable déclarée avec `const` a une portée de bloc et ne peut pas être modifiée après sa déclaration |

Exemples de déclarations :

```
var maVariable = 10;
let maSecondeVariable = 20;
const maConstante = 30;
```

Privilégiez l'utilisation de `let` et `const` dans les projets modernes, car ils offrent de meilleures garanties en termes de portée et de sécurité de données.

---

### `var`

`var` est l'ancienne manière de déclarer des variables. Il est remplacé progressivement par `let` et `const`.

Exemple:

```
var name = "John Doe";
```

Utilisez `let` et `const` à la place de `var` pour éviter des problèmes liés à la portée des variables.

---

### `let`

`let` est similaire à `var`, mais avec une **portée de bloc**.

Exemple:

```
let age = 25;
```

Portée de bloc signifie que la variable est limitée à l'intérieur du bloc de code `{}` où elle a été déclarée.

---

### `const`

`const` est utilisé pour déclarer des **constantes**, c'est-à-dire des variables dont la valeur ne changera pas.

Exemple:

```
const pi = 3.14;
```

Les noms de constantes sont souvent écrits en majuscules (par exemple, `const PI = 3.14;`) pour les différencier des autres variables.

---

## Types de données

En JavaScript, il y a **5 types** de données principaux:

- **String**
- **Number**
- **Boolean**
- **null**
- **undefined**

Par exemple :

- String : 'Bonjour'
- Number : 42
- Boolean : true ou false
- null : représente une absence intentionnelle d'objet ou de valeur
- undefined : une variable non initialisée

---

## String

Les **chaînes de caractères** sont utilisées pour représenter du **texte**.

Exemple:

```
let name = "John Doe";
```

Les chaînes de caractères peuvent être définies avec des guillemets simples ou doubles, tant que le début et la fin correspondent.

---

## Number

Les **nombres** en JavaScript peuvent être **entiers** ou **décimaux**.

Exemple:

```
let age = 25;
let pi = 3.14;
```

Les décimaux peuvent aussi être écrits avec la notation scientifique. Par exemple, 3e8 représente la vitesse de la lumière (300,000,000).

---

## Boolean

Les booléens représentent **vrai** (`true`) ou **faux** (`false`).

Exemple:

```
let isActive = true;
```

Les booléens sont un type de données couramment utilisé pour les conditions et les opérations logiques.

---

## Null en JavaScript

`null` est une valeur spéciale qui représente l'absence de valeur.

Exemple:

```
let emptyVariable = null;
```

`null` est différent de `undefined`. `null` est utilisé lorsque vous voulez spécifier qu'une variable n'a aucune valeur, tandis qu'une variable non déclarée a une valeur `undefined`.

---

## Valeur undefined

`undefined` est la valeur d'une variable qui n'a pas encore été **définie**.

Exemple:

```
let notDefined;
console.log(notDefined); // undefined
```

"undefined" signifie qu'une variable n'a pas été définie, tandis que "null" indique une valeur intentionnellement inexistante.

---

## Conversion de types

**JavaScript** est un langage **faiblement typé**, ce qui signifie qu'il peut effectuer des conversions de types implicites.

- Exemple de conversion automatique :
  1. Conversion de chaîne de caractères en nombre :

     ```
     const result = "42" - 2;
     console.log(result);  // 40
     ```

2. Conversion de nombre en chaîne de caractères :

```
const result2 = 42 + "2";
console.log(result2);  // "422"
```

Les opérateurs + et - sont impliqués dans la conversion de types. Soyez attentifs aux conversions inattendues lors de l'utilisation de ces opérateurs sur des valeurs de types différents.

---

## Méthodes de conversion

- `Number(value)` : Convertit une valeur en **nombre**.
- `String(value)` : Convertit une valeur en **chaîne de caractères**.
- `Boolean(value)` : Convertit une valeur en **booléen**.

Ces méthodes sont utiles pour convertir une valeur d'un type de données à un autre en fonction des besoins du programme.

---

## Opérations implicites

```
let a = "5";
let b = 2;
console.log(a + b); // "52"
```

Les opérations avec des types différents peuvent entraîner une conversion de type automatique, aussi appelée "coercition". Dans cet exemple, le nombre 2 est converti en chaîne de caractères et concaténé avec la chaîne "5".

---

# Fondamentaux JavaScript

## Introduction aux opérateurs

Les **opérateurs** permettent de réaliser des **opérations** sur des variables et des valeurs.

- Exemples d'opérateurs courants :
  - Arithmétiques : `+`, `/`, `%`, `-` , `*`
  - Comparaison : `==`, `===`, `!=`, `!==`, `<`, `>`, `<=`, `>=`
  - Logiques : `&&`, `||`, `!`

`===` compare aussi le type.

---

## Opérateurs arithmétiques

Les **opérateurs arithmétiques** sont utilisés pour réaliser des calculs sur des nombres.

| Opérateur | Description | Exemple |
|---|---|---|
| `+` | Addition | 5 + 3 |
| `-` | Soustraction | 5 - 3 |
| `*` | Multiplication | 5 * 3 |
| `/` | Division | 6 / 3 |
| `%` | Modulo | 7 % 3 |
| `++` | Incrémentation | x++ |
| `--` | Décrémentation | x-- |

---

## Addition

```
let a = 5;
let b = 2;
let somme = a + b; // somme = 7
```

L’opérateur '+' est utilisé pour additionner deux variables en JavaScript.

### Soustraction

```
let a = 5;
let b = 2;
let difference = a - b; // difference = 3
```

### Multiplication

```
let a = 5;
let b = 2;
let produit = a * b; // produit = 10
```

### Division

```
let a = 5;
let b = 2;
let quotient = a / b; // quotient = 2.5
```

La division peut donner un résultat décimal.

### Modulo

```
let a = 5;
let b = 2;
let reste = a % b; // reste = 1
```

Le modulo permet de calculer le reste de la division de deux nombres. Il est utile pour vérifier la parité, tester la divisibilité, etc.

## Opérateurs de comparaison

Les **opérateurs de comparaison** sont utilisés pour comparer des valeurs.

| Opérateur | Description | Exemple |
| --- | --- | --- |
| `==` | Égal à | 5 == 8; //false |
| `!=` | Non égal à | 5 != 8; //true |
| `===` | Égal en valeur et en type | 5 === "5" //false |
| `!==` | Différent en valeur ou en type | 5 !== "5" //true |
| `>` | Supérieur à | 5 > 8; //false |
| `<` | Inférieur à | 5 < 8; //true |
| `>=` | Supérieur ou égal à | 5 >= 8; //false |
| `<=` | Inférieur ou égal à | 5 <= 8; //true |

### Égalité

```
let a = 5;
let b = "5";
```

```
let isEqual = a == b; // isEqual = true
```

Il est recommandé d'utiliser l'égalité stricte (`===`) qui vérifie aussi le type des variables pour éviter les erreurs potentielles.

---

### Inégalité

```
let a = 5;
let b = "5";
let isNotEqual = a != b; // isNotEqual = false
```

Le contraire d'une inégalité est une égalité. Les deux opérateurs d'égalité sont **==** et **===**. La différence réside dans le fait que **===** compare également le type des deux variables, tandis que **==** ne compare que leur valeur.

---

### Inférieur (strict/égale)

```
let a = 5;
let b = 2;
let isInferieur = a < b; // isInferieur = false
let isInferieurEgal = a <= b; // isInferieurEgal = false
```

Les opérateurs d'infériorité permettent de comparer deux valeurs entre elles pour déterminer si la première est inférieure strictement (ou égale) à la seconde.

---

### Supérieur (strict/égale)

```
let a = 5;
let b = 2;
let isSuperieur = a > b; // isSuperieur = true
let isSuperieurEgal = a >= b; // isSuperieurEgal = true
```

Les opérateurs de comparaison permettent de vérifier si un élément est supérieur ou égal à un autre.

---

## Opérateurs logiques

Les **opérateurs logiques** sont utilisés pour **combiner** des conditions.

- `&&` : ET logique (AND)
- `||` : OU logique (OR)
- `!` : NON logique (NOT)

| Opération | Expression | Résultat |
|---|---|---|
| AND | A && B | Vrai si A et B sont vrais, sinon Faux |
| OR | A \|\| B | Vrai si A ou B sont vrais, sinon Faux |
| NOT | !A | Vrai si A est Faux, sinon Faux |

Ces opérateurs permettent de créer des conditions complexes en combinant plusieurs expressions booléennes.

---

### ET (&&)

```
let a = true;
let b = false;
let result = a && b; // result = false
```

L'opérateur logique ET (&&) retourne true si les deux opérandes sont true, sinon il retourne false.

---

### OU (||)

```
let a = true;
let b = false;
let result = a || b; // result = true
```

L'opérateur OU logique (||) retourne **true** si au moins l'une des deux opérandes est **true**, sinon il retourne **false**.

---

### NON (!)

```
let a = true;
let result = !a; // result = false
```

L'opérateur "!" permet de faire une **négation** logique.

---

# Structures de contrôle

## Instructions conditionnelles

Les instructions conditionnelles permettent d'exécuter du code en fonction d'une **condition** donnée.

```
if (condition) {
  // Code à exécuter si la condition est vraie
} else {
  // Code à exécuter si la condition est fausse
}
```

Exemple :

```
let age = 17;

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

Les instructions conditionnelles sont un élément essentiel de la programmation et permettent de créer des logiques complexes en prenant des décisions basées sur des conditions.

---

### `if`

La déclaration `if` exécute du code si une condition est **vraie (true)**.

```
if (condition) {
  // Bloc de code exécuté si la condition est vraie
}
```

- La **condition** est une expression qui est évaluée à true ou false.
- Le **bloc de code** entre les accolades est exécuté si la condition est vraie (true).

---

### Syntaxe

```
if (condition) {
  // code à exécuter si la condition est vraie
}
```

La **condition** doit être une expression qui renvoie un **booléen** (vrai ou faux), et le code à l'intérieur des accolades est exécuté uniquement si la condition est vraie.

---

## Exemple d'utilisation

```
let age = 18;

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

Cette slide montre un exemple de base d'utilisation d'un **if statement** avec la condition d'être majeur.

---

## `else`

La déclaration `else` exécute du code si la condition dans le `if` est **fausse** (false).

```
if (condition) {
  // Code à exécuter si la condition est vraie (true)
} else {
  // Code à exécuter si la condition est fausse (false)
}
```

Utilisez `else` pour couvrir tous les cas qui ne sont pas traités par les conditions précédentes.

---

## Syntaxe

```
if (condition) {
  // code à exécuter si la condition est vraie
} else {
  // code à exécuter si la condition est fausse
}
```

La condition doit être une expression qui peut être évaluée comme vrai ou faux (boolean).

---

## Exemple d'utilisation

```
let age = 16;

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

Cet exemple montre l'utilisation d'une structure **if-else** pour déterminer si une personne est majeure ou mineure en fonction de son âge.

---

## `else if`

La déclaration `else if` permet de tester **plusieurs conditions**.

```
let age = 25;

if (age < 18) {
  console.log("Mineur");
} else if (age < 25) {
  console.log("Jeune adulte");
```

```
} else {
  console.log("Adulte");
}
```

| Condition | Résultat |
| --- | --- |
| age < 18 | Mineur |
| age < 25 | Jeune adulte |
| Sinon | Adulte |

### Syntaxe

```
if (condition1) {
  // code à exécuter si la condition1 est vraie
} else if (condition2) {
  // code à exécuter si la condition1 est fausse et la condition2 est vraie
} else {
  // code à exécuter si les deux conditions sont fausses
}
```

Cette structure est appelée "branchement conditionnel" et permet d'exécuter différentes sections de code en fonction des conditions données.

### Exemple d'utilisation

```
let age = 16;

if (age >= 18) {
  console.log("Vous êtes majeur.");
} else if (age < 18 && age >= 13) {
  console.log("Vous êtes adolescent.");
} else {
  console.log("Vous êtes enfant.");
}
```

Cet exemple montre comment utiliser une structure conditionnelle avec `if`, `else if` et `else` pour déterminer si une personne est majeure, adolescente ou enfant en fonction de son âge.

## Boucles

Les boucles permettent de **répéter** du code un certain nombre de fois ou jusqu'à ce qu'une **condition** soit satisfaite.

- **For loop** :

```
for (let i = 0; i < 10; i++) {
  console.log("Itération", i);
}
```

- **While loop** :

```
let i = 0;
while (i < 10) {
  console.log("Itération", i);
  i++;
}
```

- **For...of loop** (pour parcourir des éléments d'un tableau ou d'une chaîne) :

```
const array = [1, 2, 3];
for (const element of array) {
  console.log("Élément", element);
}
```

---

## `while`

La boucle `while` exécute un bloc de code **tant qu'une condition donnée est vraie**.

```
let compteur = 0;

while (compteur < 5) {
  console.log("Compteur: " + compteur);
  compteur++;
}
```

| Compteur | Valeur à la fin de l'itération |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 3 |
| 3 | 4 |
| 4 | 5 |

Si la condition n'est jamais fausse, la boucle tournera indéfiniment. Vérifiez toujours qu'une telle situation ne peut pas se produire pour éviter les problèmes.

---

### Syntaxe

```
while (condition) {
  // code à répéter tant que la condition est vraie
}
```

Le `while` est une boucle qui vérifie la condition avant chaque itération. Le code à l'intérieur sera exécuté tant que la condition est vraie.

---

### Exemple d'utilisation

```
let i = 0;

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

Cette boucle while sert à afficher les valeurs de i de 0 à 4. Il est important de mettre à jour la variable de contrôle (i) pour éviter une boucle infinie.

---

## `do/while`

La boucle `do/while` exécute un **bloc de code** au moins une fois, puis répète tant que la **condition donnée** est vraie.

```
let i = 0;

do {
  // Code à exécuter
```

```
  i++;
} while (i < 5);
```

La boucle `do/while` est différente de la boucle `while` car elle exécute le bloc de code au moins une fois, même si la condition n'est pas vérifiée au départ.

---

## Syntaxe

```
do {
  // code à répéter
} while (condition);
```

La boucle "do-while" exécute le bloc de code au moins une fois, puis vérifie la condition pour déterminer si elle doit continuer à s'exécuter.

---

## Exemple d'utilisation

```
let i = 0;

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

Cet exemple montre l'utilisation de la boucle **do-while** qui est similaire à la boucle **while** mais exécute le code au moins une fois, même si la condition n'est pas remplie.

---

# `for`

La boucle `for` exécute un bloc de code un certain nombre de fois, selon un **compteur**.

```
for (let i = 0; i < 5; i++) {
  console.log("Le compteur est à " + i);
}
```

| Composant | Description |
| --- | --- |
| `let i=0` | Initialise le **compteur** i à 0. |
| `i<5` | Condition d'exécution : i doit être inférieur à 5. |
| `i++` | Incrémente le **compteur** i de 1 à chaque itération. |

---

## Syntaxe

```
for (initialisation; condition; incrémentation) {
  // code à répéter
}
```

Le contenu de la boucle for sera executé tant que la condition est vraie. Ne pas oublier d'incrémenter la variable pour éviter une boucle infinie.

---

## Exemple d'utilisation

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

Cet exemple montre une boucle for qui permet d'itérer sur les nombres de 0 à 4 et d'afficher leur valeur.

---

# Fonctions

## Déclaration de fonctions

Les **fonctions** en JavaScript permettent d'**encapsuler** du code pour le **réutiliser** à plusieurs endroits et **améliorer la modularité**.

```
function maFonction(param1, param2) {
  // code à exécuter
  return resultat;
}
```

Les fonctions peuvent être déclarées avec ou sans paramètres et peuvent également retourner ou non une valeur (à l'aide de l'instruction 'return').

---

### Syntaxe

```
function nomDeLaFonction(param1, param2) {
  // Instructions
  return resultat;
}
```

Les **paramètres** passés à la fonction sont utilisés dans les instructions et la fonction retourne un **résultat**.

---

### Exemples d'utilisation

```
function addition(a, b) {
  return a + b;
}

let somme = addition(5, 3);
```

Cette fonction `addition()` prend deux paramètres (a et b) et retourne leur somme. On stocke ensuite le résultat dans la variable `somme`.

---

## Paramètres et arguments

---

### Passage par valeur

Les types de données **simples** (string, boolean, number) sont **passés par valeur**.

```
function modifierValeur(a) {
  a = 42;
}

let b = 10;
modifierValeur(b);
console.log(b); // b reste égal à 10
```

Le passage par valeur signifie que seules les valeurs sont transmises à la fonction, sans affecter la variable d'origine.

---

### Passage par référence

Les **objets** et **tableaux** sont passés par référence.

```
function ajouterElement(tableau) {
  tableau.push(42);
}
```

```
let monTableau = [1, 2, 3];
ajouterElement(monTableau);
console.log(monTableau); // monTableau est maintenant [1, 2, 3, 42]
```

Cela signifie que lorsque vous modifiez un objet ou un tableau à l'intérieur d'une fonction, cela affecte également l'objet ou le tableau en dehors de la fonction.

## Portée des variables

### Globale

Les variables déclarées en dehors des fonctions sont accessibles **partout** dans le code (attention aux **conflits**).

```
let variableGlobale = "Je suis globale!";

function afficherGlobale() {
  console.log(variableGlobale);
}

afficherGlobale(); // Affiche "Je suis globale!"
```

Les variables globales peuvent être pratiques, mais elles peuvent aussi causer des problèmes si elles sont trop nombreuses ou mal gérées. Il est souvent préférable d'utiliser des variables locales et de les passer en paramètres aux fonctions.

### Locale

Les variables déclarées à l'intérieur d'une fonction ont une **portée locale** et ne sont accessibles que dans cette fonction.

```
function afficherLocale() {
  let variableLocale = "Je suis locale!";
  console.log(variableLocale);
}

afficherLocale(); // Affiche "Je suis locale!"
console.log(variableLocale); // Erreur: variableLocale n'est pas définie
```

Il est important de bien distinguer les variables locales des variables globales pour éviter les conflits et les problèmes de référencement.

### Bloc

`let` et `const` ont une portée de **bloc**, limitée aux instructions entre `{}`.

```
if (true) {
  let variableDeBloc = "Je suis dans un bloc!";
}

console.log(variableDeBloc); // Erreur: variableDeBloc n'est pas définie
```

La portée de bloc avec `let` et `const` permet d'éviter les problèmes liés aux variables globales et facilite la compréhension du code.

# Bonnes pratiques

## Style de code

### Indentation

Utilisez des **indentations** pour améliorer la **lisibilité** de votre code.

Exemple :

```
if (a > b) {
    console.log("a est supérieur à b");
} else {
    console.log("b est supérieur ou égal à a");
}
```

L'indentation est généralement de 2 ou 4 espaces, selon les préférences du développeur ou les conventions de l'équipe.

---

### Nommage des variables

Utilisez des noms de variables **clairs** et **descriptifs** pour faciliter la compréhension du code.

Exemple :

```
// Mauvais
let x = 42;

// Bon
let age = 42;
```

---

### Commentaires

Écrivez des **commentaires** pour expliquer le code qui pourrait être difficile à comprendre pour les autres.

Exemple :

```
// Calcul de la somme des deux nombres
let somme = a + b;
```

---

## Debugging

---

### Console.log()

Utilisez `console.log()` pour afficher des informations dans la **console** et faciliter le **débogage**.

Exemple :

```
console.log("a=", a, "b=", b, "somme=", somme);
```

---

### Debugger du navigateur

Utilisez les **outils de débogage** fournis par les navigateurs pour inspecter et corriger les erreurs dans votre code.

Exemple :

- Utilisez les onglets "**Console**" et "**Sources**" dans **Chrome DevTools** ou **Firefox Developer Tools** pour analyser et déboguer le code JavaScript.

---

### Exemple de HTML/CSS/JavaScript

Dans un fichier `mapage.html` :

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Ma page Web simple</title>
    <style>
        /* Styles pour le contenu principal */
        main {
            padding: 20px;
            text-align: center;
        }
    </style>
</head>
<body>
    <main>
        <p>Bienvenue sur ma page Web !</p>
        <button id="myButton">Cliquez ici</button>
    </main>

    <script>
        // Ajout d'un comportement au bouton
        document.getElementById('myButton').addEventListener('click', function() {
            alert('Vous avez cliqué sur le bouton !');
        });
    </script>
</body>
</html>
```