

# JAVASCRIPT

Créé par [Alexandre Rivet](#)

# INTRODUCTION

# DÉFINITION

- Créé en 1995
- Un langage de programmation les plus populaires et fait partie des *standards* avec le HTML et CSS
- Langage interprété par les principaux navigateurs web: Google Chrome, Safari, Firefox, Edge, etc.
- Son évolution est gérée par le group ECMA international

# FONDAMENTAUX

- JavaScript est un langage dynamique
- JavaScript est un langage (principalement) côté client
- JavaScript est un langage interprété
- JavaScript est un langage orienté objet

# LANGAGE DYNAMIQUE

- Mettre à jour le contenu des pages web sans devoir changer le code manuellement
- En opposition aux langages *statiques* tels que HTML et CSS. Pour rappel, le HTML est un langage de balisage utilisé pour structurer le contenu d'une page web et le CSS lui sera là pour mettre en forme et donc styliser ce contenu
- Intervient donc pour manipuler le contenu HTML ou les styles CSS

# LANGAGE (PRINCIPALEMENT) CÔTÉ CLIENT (1/2)

- Un site web est un ensemble de ressources et de fichiers liés entre eux
- Pour accéder à un site Web de n'importe où, il faut l'héberger sur un serveur
- Un serveur est un ordinateur connecté à d'autres serveurs qui va héberger les fichiers et les envoyer à un client sur sa demande

# LANGAGE (PRINCIPALEMENT) CÔTÉ CLIENT (2/2)

- Lorsqu'on accède à un site web via un navigateur, le serveur envoie donc tous les fichiers puis le navigateur se charge de les **exécuter côté client**
- En opposition, un langage côté serveur est exécuté côté serveur et les navigateurs ne sont pas capables de les comprendre

*Note: Le JavaScript est principalement côté client mais on pourrait aussi s'en servir côté serveur à condition de l'utiliser dans un environnement adéquat (avec Node.js notamment)*

# LANGAGE INTERPRÉTÉ

- Langage interprété: Capable d'être exécuté directement sous condition d'avoir le bon **interpréteur** (ce que possède chaque navigateur web)
- En opposition au langage **compilé** tel que le langage C qui transforme d'abord en une autre forme pour pouvoir exécuter le code



# LANGAGE ORIENTÉ OBJET

Détaillé ici

# ENVIRONNEMENT DE TRAVAIL

# EDITEURS

- Editeurs de textes simples: [Sublime Text](#), [Visual Studio Code](#) (recommandé)
- Des environnements de développement intégrés (IDE): [WebStorm](#), [IntelliJ IDEA](#)
- Editeurs en ligne: [CodePen](#), [JSFiddle](#)

# NAVIGATEURS

- Google Chrome (recommandé)
- Mozilla Firefox
- Safari
- Microsoft Edge

# DEBOGAGE

# PREMIER PROJET

# LA PAGE HTML

```
<!doctype html>  
<html>  
  <head>  
    <title>Hello World Page</title>  
  </head>  
  <body>  
    Hello World!  
  </body>  
</html>
```

# ECRIRE DU JAVASCRIPT - DANS UNE BALISE HTML

Résultat	index.html
<pre>&lt;!doctype html&gt; &lt;html&gt;   &lt;head&gt;     &lt;title&gt;Hello World Page&lt;/title&gt;   &lt;/head&gt;   &lt;body&gt;     &lt;button onclick="alert('Hello World!')"&gt;Hello&lt;/button&gt;   &lt;/body&gt; &lt;/html&gt;</pre>	

*De nouvelles techniques permettent d'éviter ce genre de syntaxe qui est même déconseillé et considéré comme une mauvaise pratique.*

*Aujourd'hui le web et donc les composants sont de plus en plus complexes et donc la séparation des morceaux de code est la norme aujourd'hui.*



# ECRIRE DU JAVASCRIPT - DANS LA PAGE HTML

Résultat	index.html
<pre>&lt;!doctype html&gt; &lt;html&gt;   &lt;head&gt;     &lt;title&gt;Hello World Page&lt;/title&gt;   &lt;/head&gt;   &lt;body&gt;     &lt;button&gt;Hello&lt;/button&gt;     &lt;script type="text/javascript"&gt;       document.querySelector('button').onclick = function() {         alert('Hello World!');       }     &lt;/script&gt;   &lt;/body&gt; &lt;/html&gt;</pre>	

*Cette solution est meilleure que la précédente mais encore une fois pas celle à retenir pour les mêmes raisons que la précédente. Cela est surtout vrai si on écrit du code que l'on souhaite réutiliser entre plusieurs pages*

# ECRIRE DU JAVASCRIPT - DANS UN FICHIER SÉPARÉ

Résultat	index.html	main.js
	<pre>&lt;!doctype html&gt; &lt;html&gt;   &lt;head&gt;     &lt;title&gt;Hello World Page&lt;/title&gt;   &lt;/head&gt;   &lt;body&gt;     &lt;button&gt;Hello&lt;/button&gt;     &lt;script src="./main.js" type="text/javascript"&gt;&lt;/script&gt;   &lt;/body&gt; &lt;/html&gt;</pre>	

*Cette solution sera donc la meilleure des 3 vues qui permettra une meilleure réutilisation et une meilleure scalabilité du code. Si on change le code à un seul endroit, la modification se répercutera partout.*

# ORDRE D'EXÉCUTION DU CODE PAR LA NAVIGATEUR (1/3)

Par défaut un navigateur va lire et exécuter le code dans l'ordre de son écriture. Il va donc analyser chaque ligne et en fonction faire des choses additionnels. Par exemple, quand le navigateur va croise une balise script, il va s'arrêter dessus (donc bloquer le reste du traitement HTML), pour charger le code JavaScript et exécuté.

## ORDRE D'EXÉCUTION DU CODE PAR LA NAVIGATEUR (2/3)

Admettons que le codé soit exécuter avant que le reste de la page et que le code fasse référence à des éléments du DOM, cela posera un problème car les éléments ne seront pas présent dans la page et donc le code plantera.

# ORDRE D'EXÉCUTION DU CODE PAR LA NAVIGATEUR (3/3)

Il existe 3 manières de contourner ça:

- Mettre tout le JavaScript que l'on souhaite exécuter dans

```
document.addEventListener('DOMContentLoaded',  
function() {})
```

- Mettre ses imports de JavaScript en fin de `body` pour qu'il soit chargé et exécuté en dernier
- Ajouter un attribut `async` ou `defer` sur la balise `script` qui permettra de mieux ordonner le chargement ou même de charger en asynchrone

# SYNTAXE & VARIABLES

# SYNTAXE

# INDENTATION

L'indentation est un concept fondamental qui consiste à la mise en forme et la structure du code. Elle consiste à ajouter des espaces ou tabulations en début de ligne pour délimiter visuellement des blocs de code en fonction des instructions.

En JavaScript, l'indentation n'a pas d'impact direct sur l'exécution du programme mais elle facilite grandement la lecture et la compréhension du code.



## LES COMMENTAIRES (1/3)

Les commentaires sont des parties de code qui sont ignorées lors de l'exécution du programme. Ils servent à fournir des explications, des notes, des remarques ou à désactiver temporairement certaines parties du code sans les supprimer réellement.

Les commentaires sont essentiels pour améliorer la lisibilité du code et pour faciliter la collaboration entre les développeurs.

## LES COMMENTAIRES (2/3)

### COMMENTAIRES SUR UNE LIGNE

Pour écrire un commentaire sur une seule ligne, vous pouvez utiliser deux slash '//'. Tout ce qui suit les deux slash sur la même ligne sera considéré comme un commentaire.

```
// Ceci est un commentaire sur une seule ligne  
var foo = 'bar'; // Vous pouvez également placer un commentaire à côté du code
```

## LES COMMENTAIRES (3/3)

### COMMENTAIRES SUR PLUSIEURS LIGNES

Pour écrire des commentaires sur plusieurs lignes, vous pouvez utiliser un slash suivi d'une étoile '/' pour commencer le commentaire, et une étoile suivie d'un slash '\*' pour le terminer.

```
/* Ceci est un commentaire  
   sur plusieurs lignes en JavaScript  
   utile pour expliquer de grandes parties de code */  
var foo = 'bar'; /*  
   Vous pouvez également utiliser les commentaires sur plusieurs lignes  
   pour désactiver temporairement une partie du code :  
   var z = 20;  
*/
```

## LE POINT-VIRGULE (1/3)

En JavaScript, le point-virgule ';' est utilisé pour terminer une instruction. Cependant, il n'est pas toujours obligatoire de l'utiliser à la fin de chaque instruction, car JavaScript a une fonctionnalité appelée **Automatic Semicolor Insertion (ASI)** qui insère automatiquement des point-virgules dans certains cas.

## LE POINT-VIRGULE (2/3)

Le point-virgule est principalement utilisé pour séparer les instructions. Si vous avez plusieurs instructions sur une même ligne, vous devez utiliser un point-virgule pour les séparer.

```
// Déclaration de variable suivie d'un point-virgule
// pour terminer l'instruction
var foo = 'bar';
// Utilisation d'une fonction suivie d'un point-virgule
// pour terminer l'instruction
console.log(foo);
```

## LE POINT-VIRGULE (3/3)

Code avec point-virgule facultatif:

```
var x = 5 // Point-virgule facultatif  
var y = 10 // Point-virgule facultatif  
console.log(x + y) // Point-virgule facultatif
```

L'ASI convertit ce code en:

```
var x = 5; // Point-virgule inséré automatiquement  
var y = 10; // Point-virgule inséré automatiquement  
console.log(x + y); // Point-virgule inséré automatiquement
```

Il est important de noter qu'il est recommandé de toujours inclure les point-virgules de manière explicite pour rendre le code plus robuste, compréhensible et éviter tous problèmes de syntaxe même par l'ASI.

# VARIABLES - PRÉSENTATION

## DÉFINITION

Une variable est un conteneur qui peut contenir une valeur, et ce conteneur peut changer de contenu au cours de l'exécution du programme.

Les variables permettent de stocker des données, telles que des nombres, des chaînes de caractères, des objets, des tableaux et d'autres types de données afin de les utiliser et les manipuler dans le code.



## DÉCLARATION (1/3)

Pour déclarer une variable, vous pouvez utiliser le mot-clé **var**, **let** ou **const**, selon le type de variable que vous souhaitez créer:

- **var** (déprécié): C'était la façon standard de déclarer des variables avec ECMAScript 6
- **let**: Utilisé pour déclarer des variables dont la valeur peut être modifiée au cours du programme
- **const**: Utilisé pour déclarer des valeurs constantes, à l'inverse de **let**, dont la valeur ne peut pas être modifiée au cours du programme

## DÉCLARATION (2/3)

Concernant le nom d'une variable, vous avez une grande liberté de nommage tout en respectant certaines règles:

- Un nom de variable doit commencer par une lettre (a-z, A-Z) ou `_`: `age`
- Le reste du nom de variable peut contenir des lettres, des chiffres (0-9) ou `_`: `name1`
- Un nom de variable ne doit pas contenir d'espace
- Un nom de variable est sensible à la casse, ce qui signifie que `age`, `Age` ou `AGE` sont 3 noms variables différents

## DÉCLARATION (3/3)

- Il y a des mots-clés qui sont réservés au langage. Vous ne pouvez pas les utiliser comme noms de variables
- Il est généralement recommandé d'utiliser la convention lower camel case pour définir les noms de variables. Cette convention dit que lorsqu'un nom de variable est composé de plusieurs mots, on colle les mots en utilisant une majuscule sauf pour le premier: `myAge`
- Il est recommandé de choisir des noms de variables significatifs et descriptifs pour améliorer la lisibilité du code: `tmp` => `myAge`

## INITIALISATION (1/2)

Initialiser une variable signifie lui attribuer une valeur initiale lors de sa déclaration. L'initialisation est la première étape pour créer une variable et lui donner une valeur de départ. On va pouvoir initialiser une variable, soit directement au moment de sa déclaration ou après l'avoir déclarée.

## INITIALISATION (2/2)

Pour initialiser une variable, on utilise l'opérateur = qui correspond à l'opérateur d'assignation et non à l'opérateur d'égalité (au sens mathématique).

```
// Déclaration et initialisation en même temps  
let age = 30;  
// Déclaration d'abord puis initialisation  
let name;  
name = 'John';
```

On utilise le mot clé (var, let ou const) que pour déclarer la variable. Lorsqu'on la manipule, on a plus besoin de le spécifier, on peut se contenter d'utiliser son nom.

*Note: Si vous déclarez une variable sans l'initialiser, elle sera automatiquement initialisée avec la valeur **undefined***

## MODIFICATION

Une variable est vouée à changer au cours de l'exécution du programme. Pour affecter de nouveau la même variable déjà initialisée, on va se contenter de réutiliser le même opérateur =.

```
// Déclaration et initialisation en même temps
let age = 30;
// Déclaration d'abord puis initialisation
let name;
name = 'John';

/* On modifie la valeur existante (30)
 * Après cette ligne, age vaudra la nouvelle valeur (40) */
age = 40;
```

## LET & VAR (1/8)

Un langage est soumis à modification tout au long de sa vie et n'est jamais figé pour toujours. En informatique, tout va très vite et ces évolutions fait que, parfois, certains langages changent de philosophie de design. Et comme tout bouge très vite, les langages ne peuvent pas décider du jour au lendemain d'abandonner certains composants pour les nouveaux, car cela serait dramatique pour le web en général pour le cas du JavaScript.

## LET & VAR (2/8)

Il y a une sorte de rétro compatibilité à assurer sinon la plupart des sites seraient bogués.

Lorsque le JavaScript a été créé et encore il y a quelques années, le seul mot clé permettant de déclarer les variables étaient var. Or après plusieurs années d'utilisation, les créateurs de JavaScript ont fini par penser que ce mot clé pouvait porter à confusion et ont ainsi décidé d'en créer un nouveau le let. Ils en ont profité pour corriger quelques problèmes des déclarations de variable utilisant var.



## LET & VAR (3/8)

### HOISTING (1/2)

```
// ✓  
age = 30;  
var age;
```

Lorsqu'on utilisait la syntaxe avec var, on n'était pas obligé de déclarer la variable avant de la manipuler dans le code. Cela est dû au fait que JavaScript va traiter les déclarations var avant le reste du code JavaScript. On appelle ça la remontée ou le hoisting.

## LET & VAR (4/8)

### HOISTING (2/2)

```
// ✗ (renvoie une erreur)  
age = 30;  
let age;
```

Le comportement de hoisting a été jugé inadapté et a donc été corrigé dans les déclarations avec let. Le but est toujours d'améliorer le code et d'en créer plus compréhensible et plus clair avec la bonne structure.

## LET & VAR (5/8)

### REDÉCLARATION (1/2)

```
// ✓  
var age = 30;  
var age = 20;
```

Lorsqu'on utilisait la syntaxe avec var, on avait le droit de déclarer plusieurs fois la même variable (ce qui avait pour conséquence de modifier la valeur stockée)

## LET & VAR (6/8)

### REDÉCLARATION (2/2)

```
// ✗ (renvoie une erreur)  
let age = 30;  
let age = 20;
```

La nouvelle syntaxe en `let` n'autorise plus cela. Si on souhaite modifier la valeur stockée, il suffit juste de la réaffecter via le nom.

Une fois plus, cette décision a été prise pour des raisons de clarté et de pertinence dans le code.

# LET & VAR (7/8)

## PORTÉE DES VARIABLES (1/2)

```
function myFunc() {  
  if (true) {  
    var x = 10;  
  }  
  // ✓ La variable x est accessible en dehors du bloc if  
  console.log(x);  
}  
myFunc(); // Affiche 10
```

Les variables déclarées en var sont fonctionnelles (portée dans une fonction), ce qui signifie qu'elles sont liées à la fonction dans laquelle elles sont déclarées (ou globales si déclarées en dehors de tout fonction)

# LET & VAR (8/8)

## PORTÉE DES VARIABLES (2/2)

```
function myFunc() {  
  if (true) {  
    let y = 10;  
  }  
  // ✗ La variable x est inaccessible en dehors du bloc if  
  console.log(y);  
}  
myFunc(); // Erreur: la variable y n'est pas définie
```

Les variables déclarées en var sont liées au bloc (ensemble de code entre {}). Elles gardent donc leur déclaration et leur affectation dans l'ordre où elles apparaissent dans le code.

# VARIABLES - TYPES

## TYPES (1/3)

Type	Exemples
Number	<pre>let a = 1; let b = -3.14;</pre>
String	<pre>let a = "Une chaîne de caractères"; let b = 'Une seconde avec "guillemets"'; let c = 'Une troisième avec \'apostrophes\'';</pre>

Petite remarque, on peut utiliser ' ou " pour délimiter une chaîne de caractères. Mais si la chaîne contient le même caractère que le délimiteur, il faudra l'échapper avec \.



## TYPES (2/3)

Type	Examples
Boolean	<pre>let trueBool = true; let falseBool = false;</pre>
null	<pre>let emptyVariable = null;</pre>
undefined	<pre>let undefinedVariable;</pre>

## TYPES (3/3)

Type	Exemples
Object	<pre>let obj1 = {}; let obj2 = new Object();</pre>
Symbol	<pre>let symbol1 = Symbol(); let symbol2 = Symbol('My Symbole');</pre>

Pour vérifier le type d'une variable, vous pouvez utiliser l'opérateur `typeof`:

```
console.log(typeof variableName);
```

# VARIABLES - OPÉRATEURS

## OPÉRATEURS ARITHMÉTIQUES (1/3)

```
let x = 10;  
let y = 20;
```

Nom	Opérateur	Exemples
Addition	+	<pre>let result = x + y;</pre>
Soustraction	-	<pre>let result = x - y;</pre>
Multiplication	*	<pre>let result = x * y;</pre>

## OPÉRATEURS ARITHMÉTIQUES (2/3)

```
let x = 10;  
let y = 20;
```

Nom	Opérateur	Exemples
Division	/	<pre>let result = x / y;</pre>
Modulo	%	<pre>let result = x % y;</pre>
Exponentielle	**	<pre>let result = x ** y;</pre>

## OPÉRATEURS ARITHMÉTIQUES (3/3)

Même règles qu'en mathématiques pour la priorité des calculs.

```
let x = 1 - 2 - 3; // -4  
let y = 1 - (3 - 2 * 4); // 6  
let z = 2 ** 3 ** 2; // 512
```

## OPÉRATEURS D'AFFECTATION (1/2)

```
let x = 10;  
let y = 20;
```

Nom	Opérateur	Exemples
Assignment	=	<pre>let z = 30;</pre>
Addition affectation	+=	<pre>x += 10;</pre>
Soustraction affectation	-=	<pre>y -= 10;</pre>

## OPÉRATEURS D'AFFECTATION (2/2)

```
let x = 10;  
let y = 20;
```

Nom	Opérateur	Exemples
Multiplication affectation	<code>*=</code>	<code>x *= 10;</code>
Division affectation	<code>/=</code>	<code>y /= 10;</code>
Modulo affectation	<code>%=</code>	<code>x %= 10;</code>



## OPÉRATEUR DE CONCATÉNATION (1/2)

La concaténation est le processus de combiner des chaînes de caractères pour former une nouvelle chaîne.

Elle s'effectue avec l'opérateur +.

```
let a = 10;
let b = 20;
let c = '30';
let d = 'quarante';
let result1 = a + b; // 30
let result2 = 'j\'ai ' + result1 + 'ans'; // j'ai 30ans
let result3 = c + d; // 30quarante
let result4 = b + c; // 2030
let result5 = a + b + d; // 30quarante
let result6 = d + a + b; //quarante1020
```

Il y a toujours la notion de priorité. Mais dès lors qu'il y a une chaîne de caractère, alors tout ce qui suit sera considéré comme chaînes de caractères.

## OPÉRATEUR DE CONCATÉNATION (2/2) - LITTÉRAUX DE GABARITS

Il existe une troisième méthode pour entourer les chaînes de caractères: ```. La grande différence avec les deux autres méthodes c'est que JavaScript va interpréter les expressions à l'intérieur et les remplacer.

```
let a = 10;
let b = 20;
console.log(`a (${a}) + b (${b}) = ${a + b}`); // "a (10) + b (20) = 30"
// vs qui produit la même chose mais en moins élégant
console.log('a (' + a + ') + b (' + b + ') = ' + (a + b));
```

# STRUCTURES DE CONTRÔLE

# PRÉSENTATION (1/2)

Les structures de contrôle en JavaScript permettent de contrôler l'exécution du code en fonction de certaines conditions. Elles vous permettent de prendre des décisions et d'exécuter des blocs de code spécifiques en fonction des valeurs des variables ou d'autres conditions.

## PRÉSENTATION (2/2)

Il existe deux grands types de structures de contrôle:

- Les conditions qui permettent d'exécuter un certain nombre d'instructions si et seulement si une condition donnée est vérifiée.
- Les boucles qui permettent d'exécuter un bloc de code autant de fois que nécessaire tant qu'une condition donnée est vérifiée.

# OPÉRATEURS DE COMPARAISON (1/4)

Les opérateurs de comparaison en JavaScript permettent de comparer des valeurs et de déterminer si une condition est vraie ou fausse. Ils sont couramment utilisés dans les structures de contrôle.

# OPÉRATEURS DE COMPARAISON (2/4)

Nom	Opérateur	Exemples
Egalité	==	<pre>/* true (valeur égale même si type différent) */ console.log(5 == '5');</pre>
Non-égalité	!=	<pre>// true (valeurs différentes) console.log(10 != '5');</pre>
Egalité stricte	===	<pre>// false (types différents) console.log(5 === '5');</pre>
Non-égalité stricte	!==	<pre>// true (types différents) console.log(5 !== '5');</pre>

# OPÉRATEURS DE COMPARAISON (3/4)

Nom	Opérateur	Exemples
Infériorité	<	<pre>console.log(10 &lt; 15); // true console.log(10 &lt; 10); // false</pre>
Infériorité ou égalité	<=	<pre>console.log(10 &lt;= 15); // true console.log(10 &lt;= 10); // true</pre>
Supériorité	>	<pre>console.log(15 &gt; 10); // true console.log(15 &gt; 15); // false</pre>
Supériorité ou égalité	>=	<pre>console.log(15 &gt;= 10); // true console.log(15 &gt;= 15); // true</pre>



## OPÉRATEURS DE COMPARAISON (4/4)

Lorsqu'on utilise un opérateur de comparaison, le JavaScript va automatiquement comparer la valeur à gauche de l'opérateur à celle de droite. Il renverra **true** si la comparaison est vérifiée et **false** si elle ne l'est pas.

## CONDITIONS (1/5)

Les conditions en JavaScript permettent de prendre des décisions dans votre code en fonction de certaines conditions ou expressions booléennes. Ces décisions permettent d'exécuter différentes parties de code en fonction des valeurs des variables ou des résultats des expressions.

## CONDITIONS (2/5) - INSTRUCTION IF

L'instruction if permet d'exécuter un bloc de code si une condition est vraie. Si la condition est fausse, le bloc de code est donc ignoré.

```
const age = 25;  
if (age >= 18) {  
    console.log("Vous êtes majeur !");  
}
```

## CONDITIONS (3/5) - INSTRUCTION ELSE

L'instruction else est utilisée avec if pour exécuter un autre bloc de code si la condition if est fausse.

```
const hour = 14;  
if (hour < 12) {  
  console.log("Bonne matinée !");  
} else {  
  console.log("Bonne après-midi !");  
}
```

## CONDITIONS (4/5) - INSTRUCTION ELSE IF

L'instruction `else if` est utilisée pour tester plusieurs conditions à la suite. Si aucune des conditions précédentes n'est vraie, cette instruction permet de tester une autre condition.

```
const mark = 75;
if (mark >= 90) {
  console.log("Excellent !");
} else if (mark >= 50) {
  console.log("Bon travail !");
} else {
  console.log("Vous pouvez faire mieux !");
}
```

# CONDITIONS (5/5) - INSTRUCTION SWITCH

L'instruction switch est utilisée pour sélectionner un bloc de code à exécuter parmi plusieurs options en fonction de la valeur d'une expression.

```
const day = "lundi";
switch (day) {
  case "lundi":
    console.log("C'est le début de la semaine.");
    break;
  case "mardi":
    console.log("C'est bientôt le week-end !");
    break;
  default:
    console.log("C'est un autre jour de la semaine.");
}
```

## OPÉRATEURS LOGIQUES (1/5)

Les opérateurs logiques en JavaScript sont utilisés pour combiner des expressions booléennes et effectuer des opérations logiques sur leurs valeurs. Ils sont couramment utilisés dans les conditions et les boucles pour effectuer des tests plus complexes.

## OPÉRATEURS LOGIQUES (2/5) - ET

L'opérateur logique && (ET) renvoie true si toutes les expressions booléennes qu'il combine sont vraies.

Sinon il renverra false. Il effectue une évaluation paresseuse, ce qui signifie que si la première expression est fausse, les expressions suivantes ne sont pas évaluées car le résultat global sera de tout façon false.

```
const a = true;  
const b = false;  
const c = true;  
console.log(a && b); // false  
console.log(a && c); // true
```



# OPÉRATEURS LOGIQUES (3/5) - OU

L'opérateur logique `||` (OU) renvoie `true` si au moins l'une des expressions booléennes qu'il combine est vraie. S'il n'y a aucune expression vraie, il renverra `false`. Comme le ET, il effectue une évaluation paresseuse, ce qui signifie que si la première expression est `true`, les expressions suivantes ne sont pas évaluées car le résultat global sera de tout façon `true`.

```
const a = true;
const b = false;
const c = false;
console.log(a || b); // true
console.log(b || c); // false
```

## OPÉRATEURS LOGIQUES (4/5) - NON

L'opérateur logique ! (NON) renverse la valeur d'une expression booléenne. S'il est appliqué à `true`, il renverra `false` et s'il est appliqué à `false`, il renverra `true`.

```
const a = true;  
const b = false;  
console.log(!a); // false  
console.log(!b); // true
```

# OPÉRATEURS LOGIQUES (5/5)

Lorsqu'on combine plusieurs opérateurs logiques, il est important de comprendre leur précedence. L'opérateur **!** (**NON**) a la plus haute précedence, suivi de **&&** (**ET**), et enfin l'opérateur **||** (**OU**). Il est souvent recommandé d'utiliser des parenthèses pour clarifier les expressions logiques complexes.

```
const a = true;
const b = false;
const c = true;
console.log((a || b) && c); // true
console.log(a || (b && c)); // true
```

# ASSOCIATIVITÉ DES OPÉRATEURS

Les règles d'associativité des opérateurs déterminent l'ordre dans lequel les opérateurs sont effectués lorsqu'il y a plusieurs opérateurs du même type dans une expression. En JavaScript, les opérateurs ont des règles d'associativité spécifiques, qui dictent si les opérations doivent être évaluées de gauche à droite ou de droite à gauche.

# ASSOCIATIVITÉ ET PRÉCÉDENCE (1/8)

Voici un tableau qui classe les opérateurs de la plus haute précedence (0) à la plus basse (10). Les opérateurs ayant le même chiffre de précedence vont être traités selon la même priorité et il faudra regarder leur associativité.

## ASSOCIATIVITÉ ET PRÉCÉDENCE (2/8)

Précédence	Nom	Symbole	Associativité
0	Groupement	( ... )	N/A
1	Post-incrémentation	... ++	N/A
1	Post-décrémentation	... --	N/A

## ASSOCIATIVITÉ ET PRÉCÉDENCE (3/8)

Précédence	Nom	Symbole	Associativité
2	NON	! ...	Droite
2	Pré- incrémentation	++ ...	Droite
2	Pré- décrémentation	-- ...	Droite

## ASSOCIATIVITÉ ET PRÉCÉDENCE (4/8)

Précédence	Nom	Symbole	Associativité
3	Exponentiel	$\dots ** \dots$	Droite
3	Multiplication	$\dots * \dots$	Gauche
3	Division	$\dots / \dots$	Gauche
3	Modulo	$\dots \% \dots$	Gauche



## ASSOCIATIVITÉ ET PRÉCÉDENCE (5/8)

Précédence	Nom	Symbole	Associativité
4	Addition	$\dots + \dots$	Gauche
4	Soustraction	$\dots - \dots$	Gauche

# ASSOCIATIVITÉ ET PRÉCÉDENCE (6/8)

Précédence	Nom	Symbole	Associativité
5	Inférieur strict	$\dots < \dots$	Gauche
5	Inférieur ou égal	$\dots \leq \dots$	Gauche
5	Supérieur strict	$\dots > \dots$	Gauche
5	Supérieur ou égal	$\dots \geq \dots$	Gauche

# ASSOCIATIVITÉ ET PRÉCÉDENCE (7/8)

Précédence	Nom	Symbole	Associativité
6	Egalité (valeur)	$\dots == \dots$	Gauche
6	Inégalité	$\dots != \dots$	Gauche
6	Egalité (valeur et type)	$\dots === \dots$	Gauche
6	Inégalité (valeur et type)	$\dots !== \dots$	Gauche

# ASSOCIATIVITÉ ET PRÉCÉDENCE (8/8)

Précédence	Nom	Symbole	Associativité
7	ET	... && ...	Gauche
8	OU	...    ...	Gauche
9	Ternaire	... ? ... : ...	Droite
10	Affectation	... = ... ... += ... etc.	Droite

# OPÉRATEUR TERNAIRE (1/2)

L'opérateur ternaire est un opérateur qui permet d'écrire des conditions de manière concise en JavaScript. Il est souvent utilisé pour prendre une décision simple et retourner une valeur en fonction d'une condition.

```
// Syntaxe
const result = condition ? valueIfTrue : valueIfFalse
// Exemple
const age = 25;
const message = age >= 18 ? 'Majeur' : 'Mineur';
console.log(message); // 'Majeur'
```

# OPÉRATEUR TERNAIRE (2/2)

L'opérateur ternaire est très utile lorsque vous avez une condition simple et que vous voulez attribuer une valeur en fonction de cette condition sans avoir à utiliser une instruction if ... else.

```
// Ternaire
let message = age >= 18 ? 'Majeur' : 'Mineur';
// if else plus classique
message = 'Mineur';
if (age >= 18) {
  message = 'Majeur';
}
```

A ne pas utiliser à outrance, car cela peut complexifier la lecture du code.

# OPÉRATEURS D'INCRÉMENTATION ET DE DÉCRÉMENTATION

## (1/3)

Les opérateurs d'incrémentation et de décrémentation en JavaScript sont utilisés pour augmenter ou diminuer la valeur d'une variable d'une unité. Ces opérateurs sont couramment utilisés dans les boucles et les calculs arithmétiques.

# OPÉRATEURS D'INCRÉMENTATION ET DE DÉCRÉMENTATION

## (2/3)

L'opérateur d'incrémentation `++` augmente la valeur d'une variable numérique d'une unité. Lorsqu'il est placé avant la variable (pré-incrémentation), la variable est d'abord augmentée, puis la nouvelle valeur est renvoyée. Lorsqu'il est placé après (post-incrémentation), la valeur actuelle est renvoyée puis la variable est augmentée.

```
let x = 5;
let y = ++x;
console.log(x, y); // 6, 6
let z = y++;
console.log(y, z); // 7, 6
```



# OPÉRATEURS D'INCRÉMENTATION ET DE DÉCRÉMENTATION

## (3/3)

L'opérateur de décrémentation -- diminue la valeur d'une variable numérique d'une unité. Lorsqu'il est placé avant la variable (pré-décrémentation), la variable est d'abord diminuée, puis la nouvelle valeur est renvoyée. Lorsqu'il est placé après (post-décrémentation), la valeur actuelle est renvoyée puis la variable est diminuée.

```
let x = 5;  
let y = --x;  
console.log(x, y); // 4, 4  
let z = y--;  
console.log(y, z); // 3, 4
```

# LES BOUCLES

Les boucles en JavaScript sont des structures de contrôle qui permettent d'exécuter des blocs de code plusieurs fois, en fonction des conditions ou d'un nombre prédéfini d'itérations.

Les boucles sont essentielles pour automatiser des tâches répétitives et pour traiter des ensembles de données.

# LA BOUCLE FOR

La boucle `for` est utilisée lorsque vous connaissez à l'avance le nombre d'itérations que vous souhaitez effectuer.

Elle se compose en trois parties: l'initialisation, la condition et l'expression d'incrément ou de décrémentation.

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

# LA BOUCLE WHILE

La boucle `while` est utilisée lorsque vous souhaitez répéter un bloc de code tant qu'une condition est vraie.

La condition est vérifiée avant chaque itération.

```
let count = 0;
while (count < 5) {
  console.log(count);
  count++;
}
```

# LA BOUCLE DO ... WHILE

La boucle do ... while est similaire à la boucle while, mais elle garantit que le bloc de code est exécuté au moins une fois, même si la condition est fausse dès le départ.

```
let x = 1;  
do {  
  console.log(x);  
  x++;  
} while (x <= 5);
```

# LA BOUCLE FOR ... OF

La boucle for ... of permet de parcourir les éléments d'une collection, comme un tableau, sans se soucier des indices.

```
const nombres = [1, 2, 3, 4, 5];  
for (const nombre of nombres) {  
  console.log(nombre);  
}
```

# LA BOUCLE FOR ... IN

La boucle `for ... in` permet de parcourir les propriétés d'un objet. Cependant, il est recommandé de l'utiliser principalement pour les objets, et non les tableaux.

```
const person = {  
  nom: "Alice",  
  age: 30,  
  profession: "Développeuse"  
};  
  
for (const prop in person) {  
  console.log(`${prop}: ${person[prop]}`);  
}
```

# INSTRUCTIONS BREAK ET CONTINUE

L'instruction `break` est utilisée pour sortir prématurément d'une boucle, tandis que l'instruction `continue` est utilisée pour passer à l'itération suivante sans exécuter le reste du code dans le bloc de boucle pour cette itération.

```
for (let i = 1; i <= 10; i++) {  
  if (i === 5) {  
    break; // Sort de la boucle lorsque i atteint 5  
  }  
  if (i === 3) {  
    continue; // Passe à l'itération suivante lorsque i est égal à 3  
  }  
  console.log(i);  
}
```



# LES BOUCLES

Il est très important de faire attention à ne pas créer de boucles infinies (qui ne se terminent jamais) et de s'assurer que les conditions de sortie sont correctement définies pour éviter tous problèmes.

# FONCTIONS

# PRÉSENTATION

Les fonctions en JavaScript sont des blocs de code nommés et réutilisables qui effectuent une tâche spécifique ou calcule une valeur et peuvent être appelées à partir d'autres parties de code. Elles jouent un rôle central dans la structuration et la modularité du code.

# DÉFINITION

Une fonction est définie en utilisant le mot-clé `function`, suivi du nom de la fonction (identificateur), d'une liste d'arguments entre parenthèses, et du bloc de code à exécuter entre accolades. Les arguments sont les valeurs que vous passez à la fonction pour qu'elle les utilise dans son traitement.

```
function functionName(argument1, argument2) {  
    // Bloc de code à exécuter  
}
```

# APPEL D'UNE FONCTION

Une fois définie, une fonction peut être appelée en utilisant son nom, suivi des parenthèses. Les valeurs spécifiées dans l'appel sont transmises aux paramètres correspondants de la fonction.

```
functionName(valeurArgument1, valeurArgument2);
```

# VALEUR DE RETOUR

Une fonction peut retourner une valeur à l'endroit où elle est appelée à l'aide du mot-clé `return`. Si aucune valeur n'est renvoyée, la fonction renverra automatiquement `undefined`.

```
function sum(a, b) {  
  return a + b;  
}  
  
const result = sum(5, 3); // Appel de la fonction et récupération du résultat  
console.log(result); // Résultat : 8
```

# PORTÉE DES VARIABLES

Les variables déclarées à l'intérieur d'une fonction ont une portée locale, ce qui signifie qu'elles ne sont accessibles que dans le contexte de la fonction. Cela permet d'éviter les conflits de noms et d'isoler les variables du reste du programme.

# FONCTIONS ANONYMES

Une fonction anonyme en JavaScript est une fonction qui n'a pas de nom associée. Elle est définie sans spécifier d'identificateur après le mot-clé **function**. Les fonctions anonymes sont souvent utilisées dans des contextes où une fonction est nécessaire pour une tâche spécifique, mais où il n'est pas nécessaire de lui donner un nom distinct.

```
// Définition d'une fonction anonyme stockée dans la variable square
const square = function(x) {
  return x ** 2;
};
// Utilisation de la fonction anonyme
const result = carre(4);
console.log(result); // Résultat : 16
```



# FONCTIONS FLÉCHÉES

Les fonctions fléchées sont une syntaxe plus concise pour définir des fonctions en JavaScript. Elles sont particulièrement utiles pour les fonctions courtes et n'ont pas leur propre contexte (**this**)

```
const square = x => x ** 2;  
console.log(square(4)); // Résultat : 16
```

# PROGRAMMATION ORIENTÉE OBJET

# PARADIGMES DE PROGRAMMATION - PROCÉDURALE (1/3)

La programmation procédurale est un paradigme de programmation qui se concentre sur l'écriture de code en organisant les instructions en procédures ou fonctions. Chaque procédure représente une série d'instructions qui effectuent une tâche spécifique. Ce paradigme vise à diviser un programme en blocs de code réutilisables, ce qui facilite la maintenance, la lisibilité et la réutilisation du code.

# PARADIGMES DE PROGRAMMATION - PROCÉDURALE (2/3)

## CARACTÉRISTIQUES

- Organisé en fonctions
- Divisé en sous-problèmes
- Variables locales à la fonction
- Programmation séquentielle

# PARADIGMES DE PROGRAMMATION - PROCÉDURALE (3/3)

## EXEMPLE

```
function add(a, b) {  
  return a + b;  
}  
  
const x = 5;  
const y = 3;  
const result = add(x, y);  
console.log(result); // Cela va afficher 8
```

# PARADIGMES DE PROGRAMMATION - FONCTIONNELLE

## (1/3)

La programmation fonctionnelle est un paradigme de programmation qui se concentre sur l'utilisation des fonctions de manière plus fondamentale et prééminente que dans d'autres paradigmes. La programmation fonctionnelle favorise la simplicité, la prévisibilité et la facilité de test.

# PARADIGMES DE PROGRAMMATION - FONCTIONNELLE

## (2/3)

### CARACTÉRISTIQUES

- Fonctions pures
- Immutabilité des données
- Fonction d'ordre supérieure
- Composition de fonctions

# PARADIGMES DE PROGRAMMATION - FONCTIONNELLE

(3/3)

## EXEMPLE

```
const numbers = [1, 2, 3, 4, 5];  
  
// Crée un tableau avec les carrés des numbers  
const squared = numbers.map(x => x ** 2);  
  
// Calcule la somme des nombres  
const sum = numbers.reduce((acc, val) => acc + val, 0);  
  
console.log(squared); // [1, 4, 9, 16, 25]  
console.log(sum); // 15
```



# PARADIGMES DE PROGRAMMATION - ORIENTÉE OBJET

## (1/3)

La programmation orientée objet (POO) est un paradigme de programmation qui se base sur la notion d'objets, qui sont des instances de classes. Le code est orienté autour d'entités autonomes appelés objets, et ces objets interagissent entre eux en utilisant des méthodes (fonctions) et des propriétés (variables)

# PARADIGMES DE PROGRAMMATION - ORIENTÉE OBJET

## (2/3)

### CARACTÉRISTIQUES

- Classes
- Objets
- Propriétés
- Méthodes
- Héritage

# PARADIGMES DE PROGRAMMATION - ORIENTÉE OBJET

## (3/3)

### EXEMPLE

Résultat	index.html	pets.js	main.js
----------	------------	---------	---------

```
function speakToDom(o){document.body.innerHTML+=o+"<br>"}const pet=new Pet("pet"),dog=new Dog("dog"),cat=new Cat("cat");[pet,dog,cat].
```

# VALEURS

# MANIPULATION DU DOM

# GESTION DES ERREURS

# ASYNCHRONICITÉ ET REQUÊTES AJAX

# MODULES & OUTIL DE DÉVELOPPEMENT