

INTRODUCTION AU JAVASCRIPT

HISTORIQUE

ORIGINE ET CRÉATEURS

JavaScript est un langage de programmation créé en **1995** par **Brendan Eich** chez **Netscape Communications**.

VERSIONS MAJEURES

Année Version

1997	ECMAScript 1 (ES1)
2009	ECMAScript 5 (ES5)
2015	ECMAScript 6 (ES6) / ECMAScript 2015

UTILISATIONS COURANTES

PROGRAMMATION WEB

JavaScript est le langage de programmation standard pour le web, permettant d'ajouter des **fonctionnalités interactives** aux sites web.

NODE.JS

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

APPLICATIONS MOBILES

Les applications mobiles peuvent utiliser **JavaScript** avec des **frameworks** tels que **React Native** et **PhoneGap**.

INTRODUCTION AU JAVASCRIPT

VARIABLES ET TYPES DE DONNÉES

DÉCLARATION DE VARIABLES

var

Utilisé pour déclarer une **variable**. Permet de **redéclarer** la même variable plusieurs fois.

```
var a = 5;  
var a = 7;
```

let

Déclaration de variable. Ne permet pas de **redéclarer** la même variable.

```
let a = 5;  
let a = 7; // Erreur
```

const

Déclaration de variable **constante**. Ne permet pas de redéclarer ni de modifier la valeur.

```
const a = 5;  
a = 7; // Erreur
```

TYPES DE DONNÉES

Les principales catégories de données en JavaScript sont les suivantes :

- **Number**
- **String**
- **Boolean**
- **Object**
- **Array**
- **Null et Undefined**

Number

Les **nombres** en JavaScript incluent les **entiers** et les **flottants**.

```
let entier = 42;  
let flottant = 3.14;
```

String

Les **chaînes de caractères** représentent du texte. Elles peuvent être écrites avec des guillemets simples, doubles ou même avec des **backticks**.

```
let chaine1 = "Hello";
let chaine2 = "World";
let chaine3 = `Hello World`;
```

Boolean

Les **booléens** représentent des valeurs de true ou false.

```
let vrai = true;  
let faux = false;
```

Object

Un **objet** est une collection de paires clé-valeur.

```
let obj = {  
    clé1: "valeur1",  
    clé2: "valeur2",  
};
```

Array

Les **tableaux** permettent de stocker une liste d'**éléments ordonnés**.

```
let arr = [1, 2, 3, 4];
```

Null ET Undefined

null représente une **absence** de valeur et undefined signifie qu'une variable n'a pas été **initialisée**.

```
let vide = null;  
let nonInitialise;  
console.log(nonInitialise); // undefined
```

OPÉRATIONS DE BASE

OPÉRATIONS MATHÉMATIQUES

ADDITION

```
let somme = a + b;
```

SOUSTRACTION

```
let difference = a - b;
```

MULTIPLICATION

```
let produit = a * b;
```

DIVISION

```
let quotient = a / b;
```

MODULO

```
let reste = a % b;
```

OPÉRATIONS SUR LES CHAÎNES DE CARACTÈRES

CONCATÉNATION

Pour **concaténer** deux chaînes de caractères, on utilise l'opérateur `+`.

```
let message = "Hello, " + "World!";
```

INTERPOLATION DE CHAÎNES

L'interpolation de chaînes se fait à l'aide des **backticks** (`) et des variables sont intégrées à l'aide de \${variable}.

```
let nom = "John";
let message = `Hello, ${nom}!`;
```

STRUCTURES CONDITIONNELLES

if

La structure **if** permet d'exécuter du code si une **condition** est vraie.

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

SYNTAXE

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

EXEMPLES D'UTILISATION

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

else

La structure `else` permet d'exécuter du code si la condition du **if** est fausse.

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

SYNTAXE

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

EXEMPLES D'UTILISATION

```
let age = 16;

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

else if

La structure `else if` permet de vérifier une **autre condition** si la condition du `if` est fausse.

```
if (condition1) {  
    // Code à exécuter si condition1 est vraie  
} else if (condition2) {  
    // Code à exécuter si condition2 est vraie mais condition1 est fausse  
} else {  
    // Code à exécuter si aucune des conditions précédentes n'est vraie  
}
```

SYNTAXE

```
if (condition1) {  
    // code à exécuter si la **condition1** est vraie  
} else if (condition2) {  
    // code à exécuter si la **condition2** est vraie  
} else {  
    // code à exécuter si aucune condition n'est vraie  
}
```

EXEMPLES D'UTILISATION

```
let age = 18;

if (age < 18) {
    console.log("Vous êtes mineur.");
} else if (age === 18) {
    console.log("Vous avez 18 ans.");
} else {
    console.log("Vous êtes majeur et avez plus que 18 ans.");
}
```

BOUCLES

BOUCLES while

Les boucles `while` permettent d'exécuter un **bloc de code** tant qu'une **condition** est vraie.

```
var i = 0;

while (i < 5) {
    console.log("Le compteur est à " + i);
    i++;
}
```

SYNTAXE

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

EXEMPLES D'UTILISATION

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

COMPARAISON AVEC D'AUTRES BOUCLES

La boucle `while` est utile lorsque vous **ne savez pas combien de fois** le code doit être exécuté.

Type de boucle	Cas d'utilisation
For	Nombre d'itérations connu à l'avance
While	Nombre d'itérations inconnu
Do-While	Exécution d'au moins une itération, puis contrôle de la condition

BOUCLES do/while

Les boucles do/while sont similaires aux boucles while avec la différence que le **bloc de code** est **exécuté au moins une fois**.

```
let i = 0;

do {
    console.log("Le compteur est à :", i);
    i++;
} while (i < 5);
```

SYNTAXE

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

EXEMPLES D'UTILISATION

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

COMPARAISON AVEC D'AUTRES BOUCLES

Utilisez la boucle `do/while` lorsque vous devez exécuter le bloc de code au moins une fois, même si la condition est initialement fausse.

BOUCLES `for`

Les boucles `for` sont utiles lorsque vous savez **combien de fois** le code doit être exécuté. Elles contiennent une **initialisation**, une **condition** et une **étape**.

```
for (initialisation; condition; étape) {  
    // Code à exécuter  
}
```

Exemple :

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

SYNTAXE

```
for (initialisation; condition; étape) {  
    // code à exécuter  
}
```

EXEMPLES D'UTILISATION

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



authorised training
provider



DIGITAL TALENT

COMPARAISON AVEC D'AUTRES BOUCLES

Utilisez la boucle `for` lorsque vous connaissez le nombre d'itérations à l'avance.

Boucle	Utilisation
For	Nombre d'itérations connu à l'avance
While	Condition à vérifier avant chaque itération
Do-While	Condition à vérifier après chaque itération, au moins une itération

OBJETS ET TABLEAUX

OBJETS

Les **objets** en JavaScript sont des collections de paires **clé-valeur**.

Clé	Valeur
clé1	valeur1
clé2	valeur2

SYNTAXE

Pour déclarer un **objet**, utilisez la syntaxe suivante :

```
let personne = {  
    prenom: "John",  
    nom: "Doe",  
    age: 30,  
};
```

PROPRIÉTÉS

Pour accéder aux **propriétés** d'un objet, utilisez la **notation point** ou la **notation crochets** :

```
console.log(personne.prenom); // "John"  
console.log(personne["nom"]); // "Doe"
```

MÉTHODES

Les **objets** peuvent avoir des **méthodes** (fonctions liées à un objet) :

```
let personne = {
    prenom: "John",
    nom: "Doe",
    direBonjour: function () {
        console.log("Bonjour, je m'appelle " + this.prenom + " " + this.nom);
    },
}
personne.direBonjour();
```

TABLEAUX

Les tableaux sont des objets utilisés pour stocker des **collections ordonnées** de valeurs.

Exemple de création d'un tableau:

```
// Création d'un tableau vide
let tableauVide = [];

// Création d'un tableau avec des éléments
let tableau = [1, 2, 3, 4, 5];
```

SYNTAXE

Pour déclarer un **tableau**, utilisez la syntaxe suivante :

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

ACCÈS AUX ÉLÉMENTS

Pour accéder aux éléments d'un tableau, utilisez la notation **crochets** avec l'indice de l'élément :

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

MANIPULATION DE TABLEAUX

- Accéder à un élément via son index : tableau [1]
- Modifier un élément : tableau [2] = 6
- Ajouter un élément à la fin : tableau.push (7)
- Supprimer le dernier élément : tableau.pop ()
- La longueur du tableau : tableau.length

MÉTHODES UTILES

Voici quelques méthodes utiles pour travailler avec les **tableaux** :

Méthode	Description
push	Ajoute un élément à la fin du tableau
pop	Supprime le dernier élément du tableau
shift	Supprime le premier élément du tableau
unshift	Ajoute un élément au début du tableau
indexOf	Renvoie l'indice d'un élément dans le tableau
forEach	Exécute une fonction pour chaque élément

EXEMPLES D'UTILISATION

```
fruits.push("raisin");
console.log(fruits);

fruits.pop();
console.log(fruits);

fruits.shift();
console.log(fruits);

fruits.unshift("fraise");
console.log(fruits);

console.log(fruits.indexOf("orange"));

fruits.forEach(function (fruit) {
```

LES FONCTIONS

DÉFINITION DE FONCTIONS

Les fonctions permettent de **grouper des instructions** et d'utiliser des **paramètres** pour manipuler des données.

```
function nomDeLaFonction(parametre1, parametre2) {  
    // Les instructions de la fonction  
}  
  
nomDeLaFonction(argument1, argument2);
```

SYNTAXE

```
function nomFonction(param1, param2) {  
    // Instructions  
}
```

PARAMÈTRES ET ARGUMENTS

Les **paramètres** sont les noms définis dans la fonction tandis que les **arguments** correspondent aux valeurs passées lors de l'appel de la fonction.

Paramètres

Nom donné à une variable lors de la création d'une fonction

Arguments

Valeurs passées lors de l'appel de la fonction

```
function maFonction(parametre1, parametre2) {  
    // Code de la fonction  
}  
  
maFonction(argument1, argument2);
```

RETOUR DE VALEURS

Une fonction peut renvoyer une valeur grâce à l'instruction **return**.

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

FONCTIONS ANONYMES

Les **fonctions anonymes** n'ont pas de nom et sont généralement assignées à des **variables**.

```
let maFonctionAnonyme = function () {  
    // Corps de la fonction anonyme  
};
```

SYNTAXE

```
const maFonction = function (param1, param2) {  
    // Instructions  
};
```

UTILISATIONS COURANTES

Les fonctions **anonymes** sont particulièrement utiles pour les **fonctions de rappel (callbacks)**.

Exemples :

```
setTimeout(function () {
    console.log("Appelé après 1 seconde");
}, 1000);

array.forEach(function (element, index) {
    console.log(element, index);
});
```

FONCTIONS FLÉCHÉES

Les **fonctions fléchées** permettent d'écrire des fonctions **plus courtes** et avec un comportement différent du mot-clé `this`.

```
// Fonction classique
function sommeClassique(a, b) {
    return a + b;
}

// Fonction fléchée
const sommeFlechee = (a, b) => a + b;

console.log(sommeClassique(3, 4)); // Résultat: 7
console.log(sommeFlechee(3, 4)); // Résultat: 7
```

SYNTAXE

```
const maFonction = (param1, param2) => {  
    // Instructions  
};
```

AVANTAGES ET DIFFÉRENCES AVEC LES FONCTIONS TRADITIONNELLES

- Syntaxe plus **concise**
- Comportement du mot-clé `this` lié au **contexte englobant**
- Pas d'arguments objet
- Pas de constructeur (ne peut pas être utilisé avec `new`)

GESTION DES ERREURS

GESTION DES ERREURS

Les erreurs sont des **événements inattendus** qui se produisent lors de l'exécution d'un programme. JavaScript fournit des **mécanismes de gestion des erreurs** pour vous aider à attraper et gérer ces erreurs.

try / catch

La structure try / catch vous permet d'exécuter du code susceptible de provoquer une **erreur** et de **gérer** cette erreur.

```
try {  
    // Code susceptible de provoquer une erreur  
} catch (error) {  
    // Gestion de l'erreur  
}
```

SYNTAXE

```
try {
    // Code à exécuter
} catch (erreur) {
    // Code à exécuter en cas d'erreur
}
```

EXEMPLES D'UTILISATION

```
try {
  const resultat = maFonction();
  console.log(resultat);
} catch (e) {
  console.log("Erreur: ", e.message);
}
```

throw

Vous pouvez déclencher une **erreur** en utilisant le mot-clé `throw`. Il vous permet de créer vos propres erreurs ou de déclencher des erreurs existantes.

```
function checkAge(age) {
  if (isNaN(age)) {
    throw "Not a number";
  }
  if (age < 18) {
    throw "Under 18";
  }
}

try {
  checkAge("abc");
} catch (error) {
  console.error("Caught error:", error);
}

.
```

SYNTAXE

```
throw expression;
```

EXEMPLES D'UTILISATION

```
function customError(message) {
    this.message = message;
    this.name = "customError";
}

function maFonction(param) {
    if (!param) {
        throw new customError("Argument manquant");
    }
}

try {
    maFonction();
} catch (e) {
    console.log(e.name + ": " + e.message);
}
```

