Web Avancée: JavaScript Moderne

Par: Pr. Hamza Khalloufi

Plan

1. Introduction
2. Fondamentaux de JavaScript
3. Les fonctions
4. DOM et événements
5. Structure de données, opérateurs moderne Tableaux
6. Cycle de vie de mémoire
7. Mot clé this
8. Comment Javascript fonctionne ?
9. Regard approfondi sur les fonctions
10. JavaScript asynchrone: Promises, Asynch/Await et AJAX

Introduction

- JavaScript, c’est quoi ?

Nous n'avons pas à nous préoccuper de choses complexes comme la gestion de la mémoire.

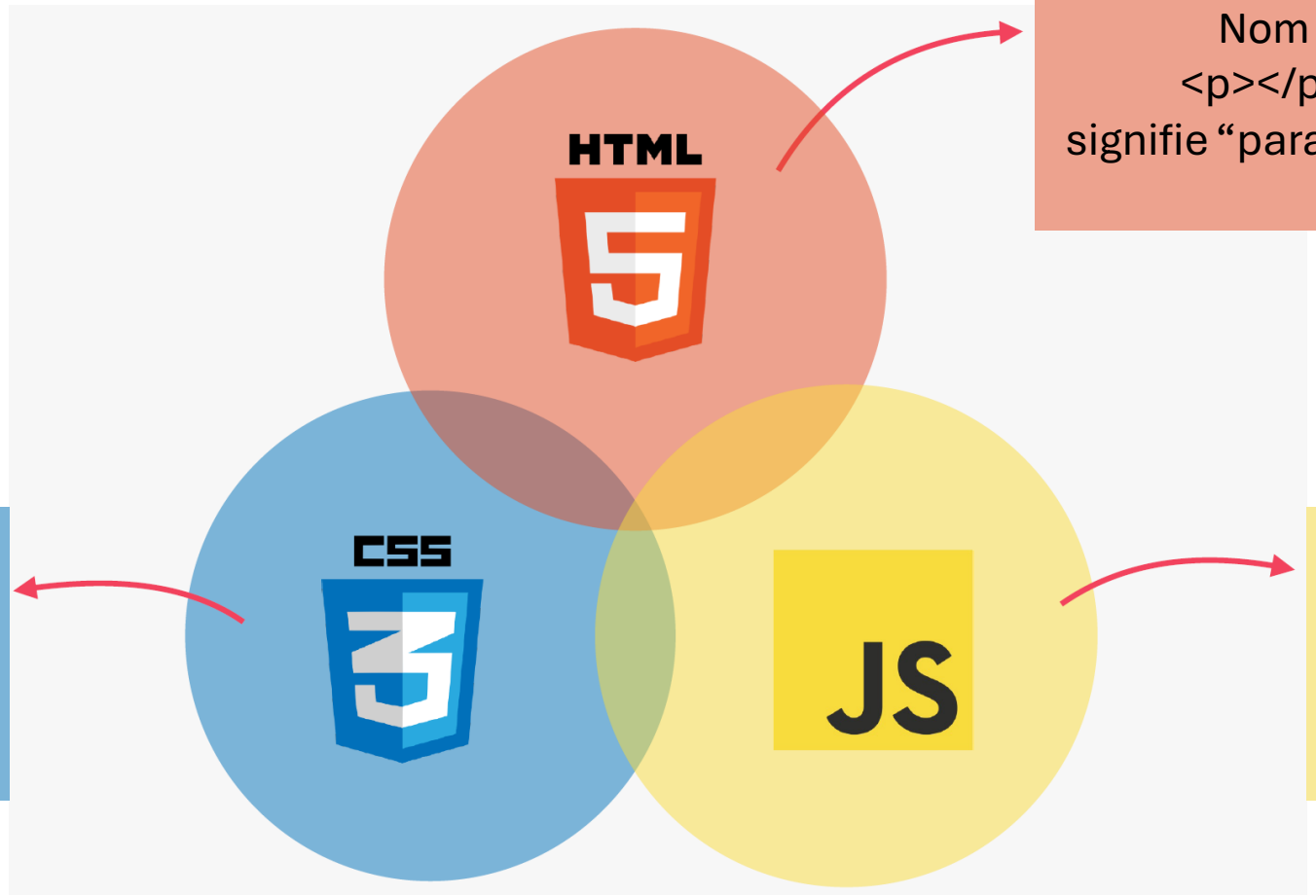
JavaScript est un langage de programmation de haut niveau, orienté objet et multi-paradigme.

Donner des instructions à l'ordinateur pour effectuer des tâches.

Nous pouvons utiliser différents styles de programmation.

Basé sur les objets, pour stocker la plupart des types de données.

Contenu



Nom

signifie “paragraph”

Adjectifs
p {color: red}
signifie “le texte du paragraphe est rouge”

Verbes
p.hide();
signifie “cache le paragraphe”

Il n’y a rien que vous ne puissiez faire avec JavaScript (ou presque…)

Ce que JavaScript permet de faire :

- ✅ Créer des sites web interactifs
- ✅ Développer des applications mobiles (React Native, Ionic)
- ✅ Créer des applications desktop (Electron)
- ✅ Développer des jeux (Phaser, Three.js)
- ✅ Piloter des objets connectés (IoT)
- ✅ Travailler côté serveur (Node.js)
- ✅ Manipuler de la 3D, de la réalité virtuelle, de l’IA
- ✅ Automatiser des tâches, des tests…

JS n’est pas le plus adapté pour :

- ❌ les applications très lourdes côté calcul (ex. calcul scientifique intensif)
- ❌ les programmes bas niveau (pilotes, OS…)

Versions ECMAScript

Version	Année	Nouveautés principales
ES1	1997	Première norme
ES3	1999	try/catch, regexps
ES5	2009	strict mode, JSON, Array.forEach
ES6 (ES2015)	**2015**	**Tournant majeur.**

- ES6 introduit des **fonctionnalités modernes** et rend JavaScript plus puissant et plus lisible :
  - ✅ Déclarations let et const
  - ✅ Fonctions fléchées ()=>
  - ✅ Classes
  - ✅ Modules (import/export)
  - ✅ Template literals `Hello ${name}`
  - ✅ Destructuring
  - ✅ Promesses (Promises)
  - ✅ Paramètres par défaut
  - ✅ Boucle for...of
  - ✅ Opérateur spread/rest

Les types primitifs en JavaScript

- Définition :
  - Les types primitifs sont les types de données les plus simples en JavaScript. Ils ne sont pas des objets et n'ont pas de méthodes (même si JavaScript permet temporairement d’en utiliser grâce à l’encapsulation automatique).
- Liste des types primitifs :

Type	Exemple	Description
String	"Bonjour"	Chaîne de caractères
Number	42, 3.14	Nombre entier ou à virgule flottante
Boolean	true, false	Valeur logique
Undefined	let x;	Variable déclarée mais non initialisée
Null	let x = null;	Absence volontaire de valeur
BigInt	1234567890123456789012345678901234567890n	Très grands nombres entiers
Symbol	Symbol("id")	Valeur unique et immuable, souvent utilisée comme identifiant

Typage dynamique

- JavaScript est un langage à typage dynamique :il n’est pas nécessaire de définir manuellement le type de données d’une variable. Le type est déterminé automatiquement par le moteur JavaScript en fonction de la valeur assignée.

```
let nom = "Hamza";        // string
let age = 30;             // number
let majeur = true;        // boolean
let ville;                // undefined
let rien = null;          // null
let grandNombre = 9007199254740991n; // bigint
let id = Symbol("id");    // symbol
```

La logique booléenne

Qu’est-ce qu’un booléen ?

- En JavaScript, un booléen est un type primitif qui ne peut prendre que deux valeurs :
  - true (vrai)
  - false (faux)
- Utilisé dans les conditions, boucles, et expressions logiques

Opérateurs de comparaison

- 📝 Utilise toujours === au lieu de == pour éviter les conversions automatiques de type.

Opérateur	Signification	Exemple (x = 5, y = "5")
==	Égal à (valeur)	x == y → true
===	Égal strict (valeur + type)	x === y → false
!=	Différent	x != y → false
!==	Différent strict	x !== y → true
> < >= <=	Comparaisons numériques	x > 3 → true

Exemple

```
let x = 5;
let y = "5";
let z = 10;

// == (égalité de valeur)
console.log(x == y); // true  ➜ car 5 == "5" (conversion automatique)

// === (égalité stricte)
console.log(x === y); // false  ➜ types différents (number ≠ string)

// != (différent de valeur)
console.log(x != y); // false  ➜ car 5 == "5"

// !== (différent strict)
console.log(x !== y); // true  ➜ types différents

// >, <, >=, <=
console.log(z > x);  // true
console.log(x >= 5); // true
console.log(z <= 10); // true
```

Opérateurs logiques

- Ces opérateurs permettent de combiner plusieurs conditions dans un if, while, etc.

Opérateur	Nom	Exemple	Résultat si x = true, y = false
&&	ET (AND)	x && y	false
||	OU (OR)	x || y	OU (OR)
!	NON (NOT)	!x	false

Exemple

```
let age = 25;
let estEtudiant = true;
let aPermis = false;

// AND (&&)
if (age >= 18 && estEtudiant) {
  console.log("Étudiant majeur"); // s'affiche
}

// OR (||)
if (aPermis || age >= 18) {
  console.log("Autorisé à conduire théoriquement"); // s'affiche car age >= 18
}

// NOT (!)
let isLoggedIn = false;
if (!isLoggedIn) {
  console.log("Veuillez vous connecter"); // s'affiche
}
```

Valeurs dites “falsy”

- Certaines valeurs sont automatiquement interprétées comme false dans un contexte booléen :
  - False
  - 0
  - "" (chaîne vide)
  - Null
  - Undefined
  - NaN
- 🧠 Toutes les autres valeurs sont dites truthy (interprétées comme true).

Exemple

```
let age = 20;
if (age >= 18) {
  console.log("Majeur"); // s'affiche
} else {
  console.log("Mineur");
}

let nom = "";
if (!nom) {
  console.log("Nom manquant"); // s'affiche car "" est falsy
}
```

Fonctions

Définition :

- Une fonction est un bloc de code qui exécute une tâche spécifique.
- Elle permet de réutiliser du code et de le rendre plus lisible et organisé.

Créer une fonction (déclaration classique)

- Syntaxe:

```
function nomDeLaFonction() {
  // instructions
}
```

- Exemple:

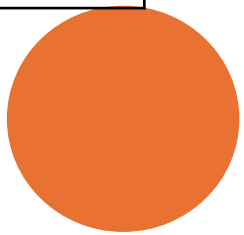
```
function saluer() {
  console.log("Bonjour !");
}
saluer(); // Affiche : Bonjour !
```

Fonction avec paramètres et retour

Rappels :

- Les paramètres sont des variables locales
- Le mot-clé return permet de renvoyer un résultat

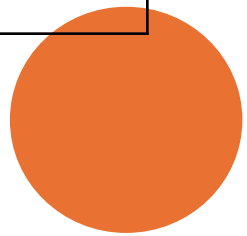
```
function addition(a, b) {
  return a + b;
}
console.log(addition(3, 4)); // 7
```



Fonctions expressions (anonymes)

- Fonction stockée dans une variable
- Utile pour les callbacks

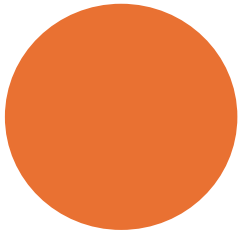
```
const saluer = function() {
  console.log("Hello !");
};
saluer(); // Hello !
```



Fonctions fléchées (ES6)

- Avantages :Syntaxe courte
- Pas de *this* propre➜ pratique dans les callbacks

```
const carre = (x) => x * x;
console.log(carre(5)); // 25
```



Portée (scope) des variables

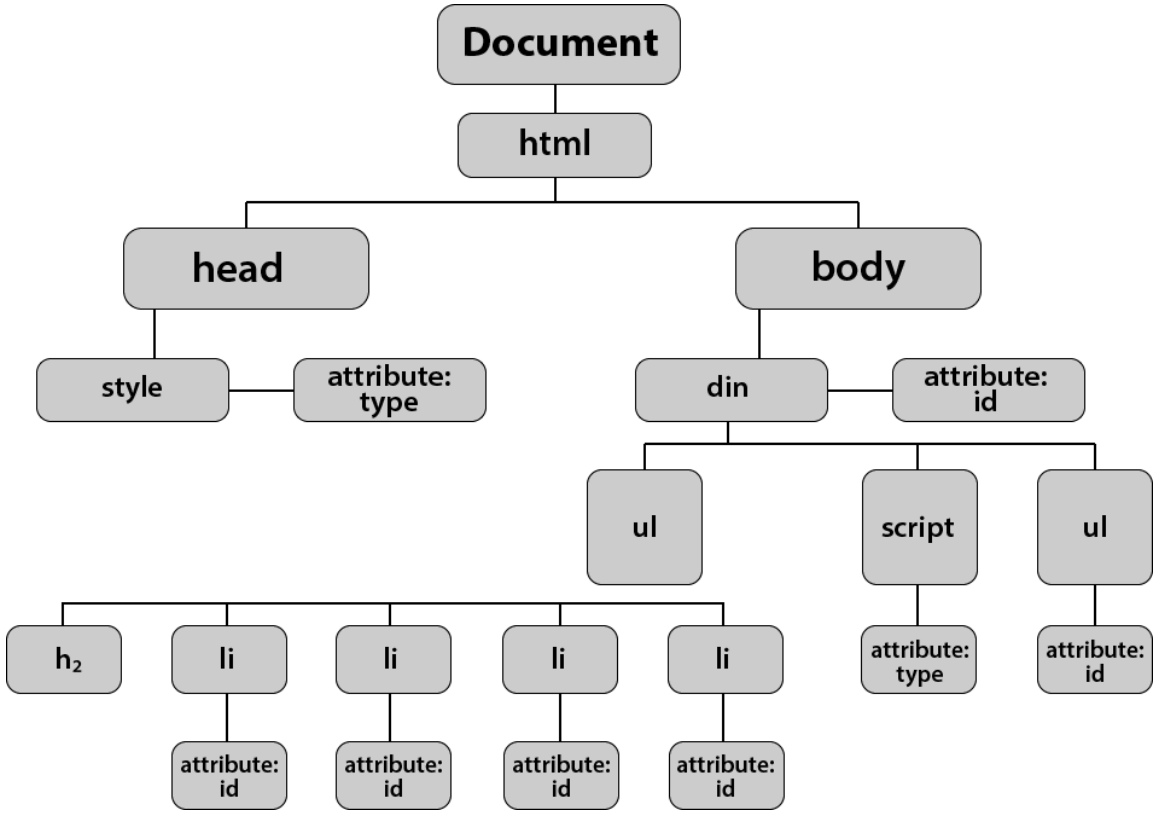
Les variables déclarées dans une fonction sont **locales à cette fonction**.

```
function test() {
  let local = "visible ici";
  console.log(local);
}
console.log(local); // ❌ Erreur : local is not defined
```

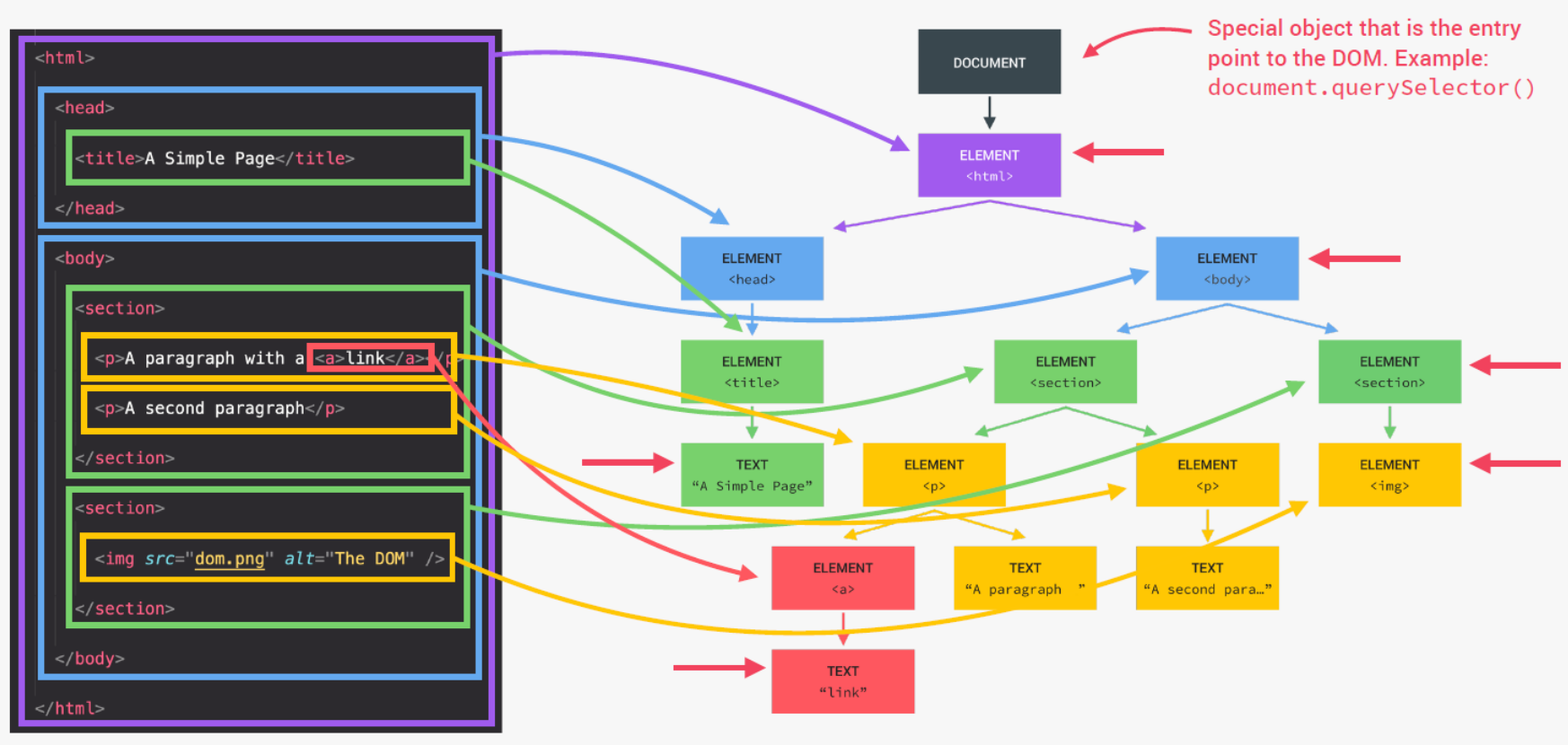
DOM

Définition :

- Le DOM (Document Object Model) est une représentation en mémoire de la structure HTML d'une page.
- Grâce au DOM, JavaScript peut accéder, modifier ou supprimer des éléments HTML dynamiquement.



Document Object Model



Accéder aux éléments du DOM

- Méthodes les plus utilisées :

Méthode	Description
getElementById()	Sélection par id
getElementsByClassName()	Tous les éléments avec une classe
getElementsByTagName()	Tous les éléments d'un certain type
querySelector()	Premier élément correspondant au sélecteur
querySelectorAll()	Tous les éléments correspondant au sélecteur

- Exemple:

```
let titre = document.getElementById("titre");
```

Modifier le contenu HTML ou texte

Tu peux modifier :

- Le texte (innerText)
- Le HTML interne (innerHTML)
- Les styles CSS (element.style)

Exemple:

```
<h1 id="titre">Bonjour</h1>
```

```
let titre = document.querySelector("#titre");
titre.innerText = "Salut les étudiants !";
titre.style.color = "blue";
```

Ajouter ou supprimer des éléments

```
let nouveau = document.createElement("p");
nouveau.innerText = "Paragraphe ajouté";
document.body.appendChild(nouveau);
```

```
let titre = document.querySelector("#titre");
titre.remove(); // Supprime le titre
```

Gestion des événements

- Exemples d’événements : click, mouseover, keydown, etc.
- Permet de réagir aux actions de l'utilisateur.

```
let bouton = document.querySelector("#btn");
bouton.addEventListener("click", () => {
  alert("Tu as cliqué !");
});
```

Mini Projet DOM

```
<button id="plus">+</button>
<span id="valeur">0</span>
<button id="moins">-</button>
```

```
let val = 0;
document.querySelector("#plus").onclick = () => {
  val++;
  document.querySelector("#valeur").innerText = val;
};
document.querySelector("#moins").onclick = () => {
  val--;
  document.querySelector("#valeur").innerText = val;
};
```

Comment fonctionne JavaScript ?

- JavaScript est souvent décrit comme un langage interprété, mais...
  - En réalité, il utilise une compilation Just-In-Time (JIT) dans les navigateurs modernes.

Compilation vs. Interprétation

Aspect	Interprétation	Compilation classique
Code exécuté	Ligne par ligne à la volée	Entièrement compilé avant l'exécution
Vitesse	Plus lent	Plus rapide
Exemple	Python	C, Java (compilé en bytecode)

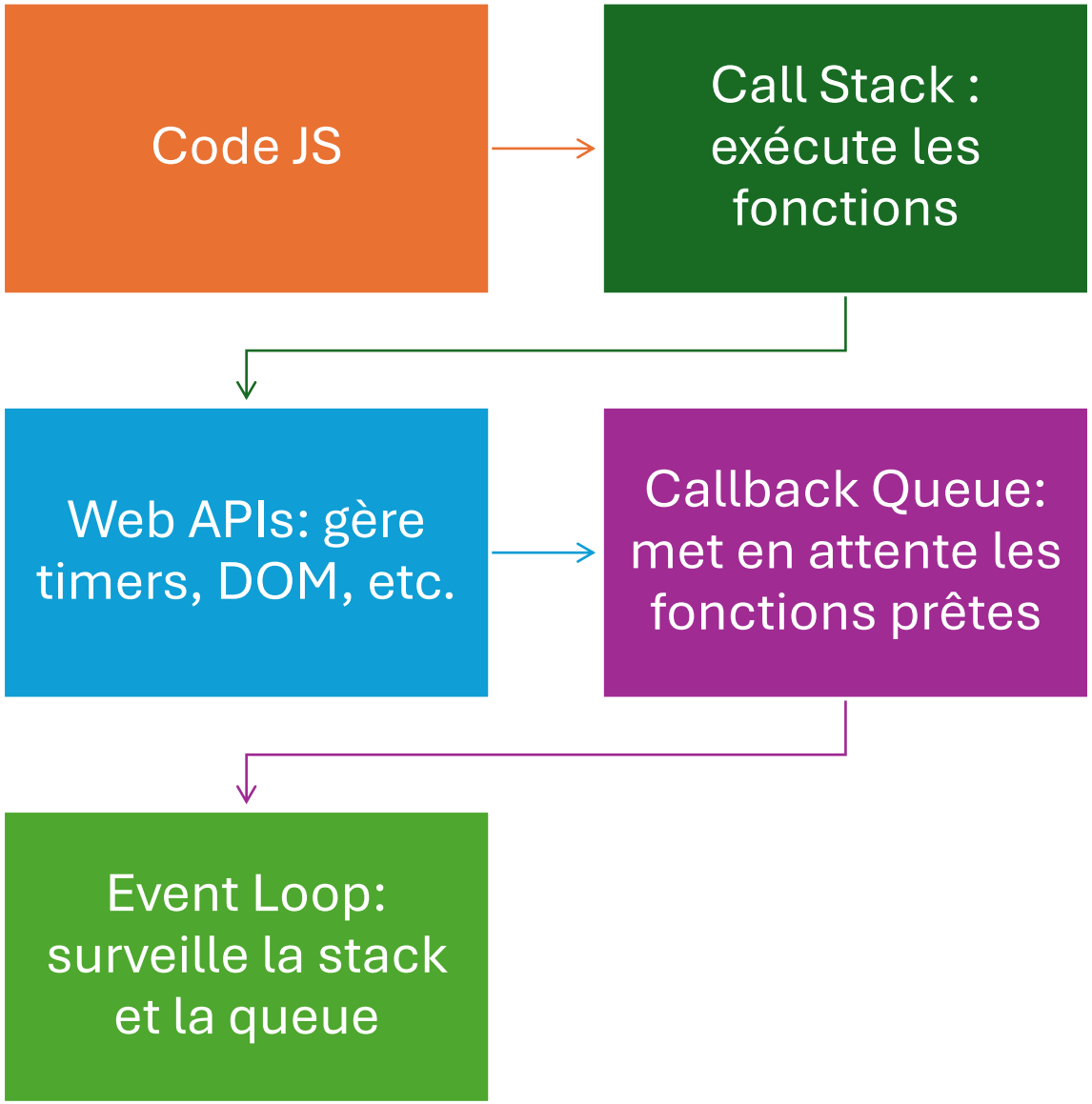
📌 JavaScript utilise une forme **hybride** ➤ compilation à la volée

Compilation Just-In-Time (JIT)

Le runtime JavaScript correspond à l'environnement d'exécution du code :

- Call Stack : pile d'appels des fonctions
- Heap : mémoire dynamique
- Web APIs (dans le navigateur) : *setTimeout*, *DOM*, *fetch*, etc.
- Event Loop : gère les tâches asynchrones
- Callback queue : file d'attente des fonctions à exécuter

Illustration du Runtime



Résumé

- JavaScript est interprété ET compilé JIT
- Il est monothreadé mais gère l'asynchrone via l'event loop
- Le moteur JS moderne (ex. V8) optimise le code en temps réel

Qu’est-ce qu’un Execution Context ?

Définition :

- Un Execution Context est l’environnement dans lequel du code JavaScript est exécuté.
- Chaque fois qu’une fonction est appelée, un nouveau contexte est créé.

Types de Contexte d’Exécution

📌 Le premier contexte créé est toujours le **Global Execution Context**.

Type de contexte	Rôle
Global Execution Context	Contexte principal (créé automatiquement)
Function Execution Context	Créé à chaque appel de fonction
Eval Execution Context	Utilisé si on appelle eval()

Contenu d’un Execution Context

Un contexte d’exécution contient :

1. Variable Environment – où les variables et fonctions sont stockées
2. Scope Chain – chaîne de portée pour l’accès aux variables
3. *this* Binding – référence à l’objet courant

La Call Stack (pile d’appel)

Définition :

- La Call Stack est une structure de données LIFO (Last In, First Out) qui stocke les contextes d'exécution actifs.
- À chaque appel de fonction, un nouveau contexte est empilé, puis dépilé à la fin de son exécution.

Exemple visuel de Call Stack

Étapes :

1. Global context est créé
2. a() est appelé → empilé
3. b() est appelé → empilé
4. console.log() → empilé puis dépilé
5. b() termine → dépilé
6. a() termine → dépilé

```
function a() {
  b();
}

function b() {
  console.log("Hello");
}

a();
```

Erreur typique : Stack Overflow

- 💥 Trop d'appels récursifs → Call Stack Overflow

```
function recurse() {
  recurse();
}
recurse();
```

En résumé

- Chaque appel de fonction crée un contexte d'exécution
- Ces contextes sont empilés dans la Call Stack
- Comprendre cela permet de :
  - Lire les erreurs de stack
  - Mieux utiliser les fonctions récursives
  - Savoir quand et pourquoi le code s’arrête

Qu’est-ce que le Scope ?

Définition :

- Le scope (ou portée) détermine où une variable est accessible dans ton code.
- En JavaScript, il existe différents niveaux de scope selon l’endroit où une variable est déclarée.

Types de Scope en JavaScript

Type de scope	Description
Global Scope	Accessible **partout** dans le fichier
Function Scope	Accessible **seulement dans une fonction**
Block Scope	Accessible **dans un bloc {}** (ES6+)

📌 Les mots-clés *let* et *const* respectent le block scope, contrairement à var.

Exemple de Scope

```
let global = "visible partout";

function test() {
  let local = "visible ici seulement";
  if (true) {
    let bloc = "visible juste dans le bloc";
    console.log(bloc); // OK
  }
  console.log(local);  // OK
  // console.log(bloc); // ❌ Erreur
}

test();
console.log(global);   // OK
// console.log(local); // ❌ Erreur
```

Qu’est-ce que la Scope Chain ?

Définition:

- La scope chain (chaîne de portées) est le chemin que JavaScript suit pour chercher une variable.
- Si une variable n’est pas trouvée dans le scope courant, JS remonte dans la chaîne jusqu’au global scope.

Exemple de Scope Chain

🔍 JavaScript cherche :

- prefixe → dans la fonction ✅
- nom → pas trouvé localement ❌ ➤ va le chercher dans le scope parent ✅

```
let nom = "Hamza";

function afficherNom() {
  let prefixe = "Bonjour";
  console.log(prefixe + " " + nom); // Cherche 'nom' en remontant
}

afficherNom(); // Bonjour Hamza
```

Portée imbriquée

- La fonction interieur() a accès à x, même si elle ne le déclare pas.

```
function exterieur() {
  let x = 10;
  function interieur() {
    let y = 5;
    console.log(x + y); // x est trouvé via la scope
chain
  }
  interieur();
}
```

Erreurs fréquentes liées au scope

- Utiliser une variable en dehors de sa portée
- Confondre let et var
- Réécrire une variable globale sans le vouloir

✅ **Bonnes pratiques:**

- Toujours déclarer tes variables (let, const, jamais sans mot-clé !)
- Utiliser const par défaut
- Limiter la portée pour éviter les effets de bord

Qu’est-ce qu’un environnement de variable

Chaque Execution Context (global ou fonction) crée un environnement de variables, où sont stockés :

- Les variables
- Les fonctions
- La valeur de this

🔍 Ce mécanisme gère la portée et l’accessibilité des variables.

Hoisting (Élévation)

Définition :

- Le hoisting signifie que les déclarations de variables et fonctions sont déplacées en haut de leur scope avant l'exécution du code.

Mais ATTENTION ❗ :

- Seules les déclarations sont « hoistées », pas les initialisations.

Exemple de hoisting

```
console.log(x); // undefined
var x = 5;
```

➞ Interprété par JavaScript comme :

```
var x;
console.log(x); // undefined
x = 5;
```

Hoisting avec let / const

```
console.log(a); // ❌ ReferenceError
let a = 3;
```

Pourquoi une erreur ?

→ C’est là qu’intervient la TDZ.

TDZ (Temporal Dead Zone)

Définition :

- La TDZ est la zone entre le début du scope et la déclaration de la variable (let / const), où la variable existe mais n’est pas accessible.
-  Accéder à la variable dans cette zone → ReferenceError

Illustration TDZ

➡ msg est dans la TDZ jusqu'à la ligne let msg = "Bonjour"

```
function test() {
  console.log(msg);
// ❌ Erreur : TDZ
  let msg =
"Bonjour";
}
```

Résumé : *var* vs *let* vs *const*

Caractéristique	var	let / const
Hoisting	✅ oui	✅ oui (mais non accessible avant déclaration)
Accessible avant déclaration	✅ undefined	❌ ReferenceError (TDZ)
Portée	fonction	bloc {}
Redéclaration	✅ autorisé	❌ interdit

- Bonnes pratiques:
  - ❌ Éviter var sauf cas très précis
  - ✅ Utiliser let pour les variables modifiables
  - ✅ Utiliser const pour tout le reste
  - ✅ Toujours déclarer les variables au début de leur scope

this en JavaScript

Définition :

- Le mot-clé this fait référence à l'objet sur lequel la fonction est appelée.
- ⚠️ Sa valeur dépend du contexte d'exécution, pas de l’endroit où elle est déclarée.

Cas d’usage

1. Contexte objet

➞ this fait référence à l’objet personne

```
const personne = {
  nom: "Hamza",
  saluer: function () {
    console.log("Bonjour " + this.nom);
  },
};

personne.saluer(); // Bonjour Hamza
```

Fonction fléchée (=>)

➞ Les fonctions fléchées ne lient pas leur propre *this*, elles héritent de leur scope parent.

```
const objet = {
  nom: "Hamza",
  saluer: () => {
    console.log(this.nom);
  },
};
objet.saluer(); // ❌
undefined
```

En résumé

Situation	Valeur de this
Dans un objet	L’objet qui appelle la méthode
Fonction simple (non strict)	window (navigateur)
Fonction simple (strict mode)	undefined
Fonction fléchée	Hérite du this du scope parent
Méthode dans une classe	L’instance de la classe
Dans un gestionnaire d'événement DOM	L’élément qui déclenche l’événement

Gestion de mémoire dans Javascript

Définition :

- La gestion de mémoire en JavaScript est automatique : le moteur JS alloue et libère la mémoire selon les besoins du programme.

✅ Pas besoin de libérer manuellement la mémoire (comme en C/C++)

Cycle de vie de la mémoire

Allocation : création d'une variable/objet

Utilisation : manipulation de la donnée

Libération : quand la donnée n’est plus utilisée, elle est libérée par le Garbage Collector

Qu’est-ce que le Garbage Collector ?

Définition:

- Le Garbage Collector (GC) est un mécanisme intégré au moteur JavaScript qui libère automatiquement la mémoire non utilisée.

Il parcourt la mémoire pour détecter les objets devenus inaccessibles et les supprime.

Le concept de "Reachability"

🚨 Seuls les objets accessibles (reachables) sont conservés

Un objet est "reachable" s’il est :

- Une variable globale
- Une variable locale d’une fonction active
- Référencé par un objet accessible

➞ Si plus rien ne pointe vers un objet, il devient inaccessible ➜ collecté

Exemple de GC

```
let utilisateur = {
  nom: "Hamza"
};

utilisateur = null; // L'objet original n’est plus référencé
// ➜ Le GC va le supprimer automatiquement
```

Mécanismes internes

JS utilise principalement un algorithme de "Mark-and-Sweep" :

- Marquer tous les objets accessibles
- Balayer les autres et les supprimer

🛠️ Implémentation propre à chaque moteur (V8, SpiderMonkey...)

Bonnes pratiques pour aider le GC

- ✅ Mettre les références à null quand on n’en a plus besoin
- ✅ Fermer les listeners / timers inutiles
- ✅ Éviter les références circulaires
- ❌ Ne pas stocker trop de données globales ou dans le DOM

Les structures de données principales

Structure	Type	Caractéristiques
Array	Ordonnée	Indexée, itérable, rapide à parcourir
Object	Non ordonnée	Clés/valeurs, lookup rapide par clé
Map	Ordonnée	Clés/valeurs, accepte **tout type de clé**
Set	Ordonnée	Valeurs uniques, pas de doublons

Quand utiliser un Array ?

✅ À utiliser pour :

- Liste ordonnée d’éléments
- Parcours ou itération (for, map, filter, reduce)
- Accès par index

```
let fruits = ["pomme", "banane", "orange"];
fruits.push("kiwi");
```

Quand utiliser un Object ?

✅ À utiliser pour :

- Associer des paires clé/valeur
- Stocker des propriétés nommées
- Accès rapide à une donnée via sa clé (string)

```
let user = {
  nom: "Hamza",
  age: 30
};
console.log(user.nom);
```

Quand utiliser une Map ?

✅ À utiliser pour :

- Paires clé/valeur avec n’importe quel type de clé
- Préserver l’ordre d’insertion
- Itérations fréquentes ou volumineuses

➞ Plus efficace qu’un objet pour les grandes quantités de données dynamiques

```
let map = new Map();
map.set("nom",
"Hamza");
map.set(42,
"Réponse");
```

Quand utiliser un Set ?

✅ À utiliser pour :

- Stocker des valeurs uniques
- Supprimer facilement les doublons
- Faire des tests d'appartenance

```
let unique = new Set([1, 2, 2, 3]);
console.log(unique); // Set(3) {1, 2, 3}
```

Résumé

Besoin	Structure conseillée
Liste ordonnée d’éléments	Array
Stocker des paires clé/valeur	Object, Map
Clés de tout type (objets, fonctions…)	Map
Éviter les doublons	Set
Données en lecture seule	const + structure

Passage par valeur vs par référence

Concept :

- En JavaScript, les types primitifs sont transmis par valeur tandis que les objets et tableaux sont transmis par référence

Cela influence si la fonction peut modifier l'original ou non.

```
let a = 5;
function modifier(x) {
  x = 10;
}
modifier(a); // a reste 5
```

```
let obj = { nom: "Hamza" };
function changer(o) {
  o.nom = "Ali";
}
changer(obj); // obj.nom devient "Ali"
```

Paramètres par défaut

- Depuis ES6, on peut donner une valeur par défaut à un paramètre, ce qui évite des erreurs si on oublie un argument.

```
function saluer(nom = "invité") {
  console.log("Bonjour " + nom);
}
saluer();       // Bonjour invité
saluer("Hamza"); // Bonjour Hamza
```

Fonctions Callback

Définition:

- Un callback est une fonction passée en argument à une autre,qui sera appelée plus tard.

C’est essentiel pour :

- la modularité du code
- la programmation asynchrone (ex: setTimeout, fetch)

Exemple:

```
function executer(callback) {
  console.log("Avant");
  callback();
  console.log("Après");
}
executer(() => console.log("Callback exécuté"));
```

Fonctions qui retournent une fonction

Définition :

- Une fonction peut retourner une autre fonction.
- Cela permet de créer dynamiquement des fonctions personnalisées.

Utile pour :

- les closures
- la programmation fonctionnelle

```
function multiplier(facteur) {
  return function(x) {
    return x * facteur;
  };
}

const doubler = multiplier(2);
console.log(doubler(5)); // 10
```

Introduction aux Tableaux

Définition

- Un tableau (Array) est une structure de données utilisée pour stocker plusieurs valeurs dans une seule variable.

📌 Les éléments sont ordonnés et accessibles via un index (position), à partir de 0.

Déclaration de tableaux

- Exemple de déclaration d’un tableau:

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

- ✅ Un tableau peut contenir n’importe quel type de donnée :

```
let mixte = [42, "bonjour", true, null];
```

Accès aux éléments

- On accède à un élément avec sa position (index) entre crochets [].

```
let nombres = [10, 20, 30];
console.log(nombres[0]); // 10
nombres[1] = 25;       // modifie le 2e élément
```

Méthodes principales

Méthode	Description	Exemple
push()	Ajouter à la fin	arr.push(4)
pop()	Supprimer le dernier	arr.pop()
shift()	Supprimer le premier	arr.shift()
unshift()	Ajouter au début	arr.unshift(1)
length	Taille du tableau	arr.length

Parcourir un tableau

Définition :

- Pour exécuter une action sur chaque élément d'un tableau, on utilise une boucle.

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

for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}
```

- Plus moderne: *for … of*

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

Méthodes avancées utiles

Méthode	Fonction
forEach()	Exécute une fonction pour chaque élément
map()	Crée un nouveau tableau transformé
filter()	Filtre les éléments selon une condition
reduce()	Réduit le tableau à une seule valeur

```
let notes = [10, 12, 15];
let doubles = notes.map(n => n * 2); // [20, 24, 30]
```

Quelques fonctions utiles

```
let arr = [1, 2, 3, 4, 5];


arr.includes(3);  // true
arr.indexOf(4);   // 3
arr.reverse();    // [5, 4, 3, 2, 1]
arr.join(" - ");  // "5 - 4 - 3 - 2 - 1"
```

Introduction à l'asynchronisation

Définition :

- JavaScript est monothreadé : il exécute une tâche à la fois.
- Pour ne pas bloquer le navigateur, on utilise des opérations asynchrones.

📌 Exemples : chargement de données, requêtes réseau, timers, accès fichiers, etc.

- Asynchrone vs Synchrone:

Synchrone	Asynchrone
Les instructions s'exécutent en ordre	Certaines tâches prennent du temps et sont **décalées**
⚠️ Bloque le fil d'exécution	✅ Libère le fil principal

```
//Exemple
console.log("1");
setTimeout(() => console.log("2"), 1000);
console.log("3");
// Résultat : 1, 3, 2
```

setTimeout / setInterval

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

setInterval(() => {
  console.log("Toutes les 2s");
}, 2000);
```

Callback Hell : Le cauchemar du code imbriqué

Qu’est-ce que le Callback Hell ?

- Le Callback Hell désigne une situation où les fonctions imbriquées avec des callbacks deviennent difficiles à lire, maintenir ou déboguer.

C’est très courant avec du code asynchrone enchaîné.

- ❌ Lecture verticale
- ❌ Imbrication en escalier
- ❌ Code fragile et difficile à maintenir

```
setTimeout(() => {
  console.log("1");
  setTimeout(() => {
    console.log("2");
    setTimeout(() => {
      console.log("3");
    }, 1000);
  }, 1000);
}, 1000);
```

Introduction aux Promesses

Définition :

- Une promesse est un objet JavaScript représentant une valeur qui sera disponible plus tard.

📌 Elle est utilisée pour gérer les opérations asynchrones, comme les requêtes réseau.

Pourquoi utiliser les promesses ?

🔧 Avant (avec callbacks) :

- Code difficile à lire
- Problème de callback hell (imbriqué)

✅ Avec les Promesses :Meilleure lisibilité

- Chaînage facile avec .then()
- Gestion claire des erreurs avec .catch()

États d’une Promesse

- Une promesse ne peut changer d’état qu’une seule fois

État	Description
pending	En attente
fulfilled	Réussie avec une valeur (resolve)
rejected	Échouée avec une erreur (reject)

Créer une promesse

```
let maPromesse = new Promise((resolve, reject) =>
{
  let toutVaBien = true;
  if (toutVaBien) {
    resolve("Succès !");
  } else {
    reject("Erreur !");
  }
});
```

Consommer une promesse

- 📌 .then() est appelé si la promesse est réussie
- 📌 .catch() est appelé si la promesse est rejetée

```
maPromesse
  .then(resultat => {
    console.log(resultat); // Succès !
  })
  .catch(erreur => {
    console.log(erreur); // Erreur !
  });
```

Chaînage de promesses

- Chaque .then() reçoit le résultat du précédent

```
fetch("https://api.exemple.com/data")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(err => console.error("Erreur :",
err));
```

async / await : L’alternative moderne aux .then()

Pourquoi async / await ?

- Les promesses .then() fonctionnent, mais peuvent vite devenir difficiles à lire si on enchaîne plusieurs opérations.
- ✅ async / await rend le code asynchrone lisible comme du synchrone

Règle de base

- async permet de déclarer une fonction asynchrone
- await pause l'exécution jusqu’à ce que la promesse soit résolue
- Syntaxe:

```
async function maFonction() {
  let resultat = await unePromesse();
  console.log(resultat);
}
```

Exemple avec await

Résultat clair : Début → Terminé ! → Fin

```
function attendre() {
  return new Promise(resolve => {
    setTimeout(() => resolve("Terminé !"), 2000);
  });
}

async function test() {
  console.log("Début");
  const res = await attendre();
  console.log(res); // "Terminé !"
  console.log("Fin");
}

test();
```

Utiliser await sans async ?

- *await* ne peut être utilisé qu'à l'intérieur d'une fonction *async*

```
const data = await fetch(url);
// ❌ Erreur
```

Gestion des erreurs avec *try...catch*

✅ Plus clair qu’un .catch() chaîné sur plusieurs .then()

```
async function charger() {
  try {
    const reponse = await fetch("https://api.exemple.com/data");
    const data = await reponse.json();
    console.log(data);
  } catch (err) {
    console.error("Erreur :", err);
  }
}
```

Comparaison visuelle : then() vs async/await

- Avec .then() :

```
fetch(url)
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

- Avec async/await :

```
const res = await fetch(url);
const data = await res.json();
console.log(data);
```

- ✅ ✅ ✅ Lisibilité + débogage + contrôle fluide

Résumé

Élément	Utilisation
async	Rend une fonction automatiquement asynchrone
await	Pause jusqu’à résolution de la promesse
try…catch	Gère les erreurs de manière propre
Avantages	Code lisible, linéaire, facile à maintenir

AJAX (XMLHttpRequest) : pourquoi ce n'est plus indispensable ?

Problème avec AJAX	Pourquoi on ne l’enseigne plus activement
Syntaxe longue, verbeuse	❌ Beaucoup de if, readyState, etc.
Gestion asynchrone complexe	❌ Pas de promesse native (sauf avec wrappers)
Moins lisible, moins intuitif	❌ Moins bon pour l’apprentissage
Lourd à maintenir	❌ Moins flexible que fetch()

Aujourd'hui : fetch() + Promise + async/await

Outil	Avantage pédagogique
fetch()	API native, claire, moderne
Promise	Bonne entrée vers la programmation asynchrone
async/await	Syntaxe lisible comme du synchrone