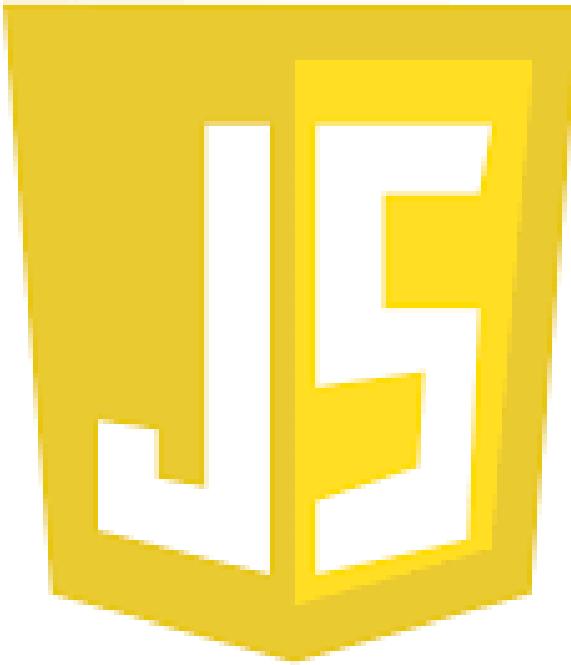


RENFORCEMENT TECHNIQUE – JAVASCRIPT & TYPESCRIPT

JavaScript



LES FONDAMENTAUX ES6+

JavaScript Moderne et Fondamentaux Avancés

- Consolider les bases modernes de JS (ES6+)
- Comprendre les fonctions avancées, les objets, le scope
- Appliquer via un mini-projet JS pur

Partie I – ES6+ : Concepts Modernes

I. **let / const** (Déclaration moderne de variables)

- **let** : variable réassignable, à portée de bloc
- **const** : constante, non réassignable (mais les objets/arrays peuvent être modifiés)

```
const utilisateur = { nom: "Ali" };
utilisateur.nom = "Jean"; // ✅ autorisé
utilisateur = {}; // ❌ interdit
```

Partie I – ES6+ : Concepts Modernes

2. Déstructuration (Extraction simplifiée)

Permet d'extraire des propriétés ou éléments directement dans des variables.

```
const user = { nom: "Leila", age: 28, pays: "France" };
const { nom, pays } = user;
console.log(nom, pays); // Leila France
```

```
const coords = [10, 20];
const [x, y] = coords;
```

Partie I – ES6+ : Concepts Modernes

2. Déstructuration (Extraction simplifiée)

Permet d'extraire des propriétés ou éléments directement dans des variables.

Cas avancé : déstructuration imbriquée

```
const post = {  
    titre: "JS Moderne",  
    auteur: { nom: "Nicolas", niveau: "expert" },  
};
```

```
const {  
    auteur: { nom },  
} = post;  
console.log(nom); // Nicolas
```

Partie I – ES6+ : Concepts Modernes

3. Rest & Spread ...

...rest : capture les éléments restants

...spread : étale ou copie les éléments

```
const [a, ...reste] = [1, 2, 3, 4];
console.log(reste); // [2, 3, 4]
```

```
const obj1 = { nom: "Awa", age: 25 };
const obj2 = { ...obj1, ville: "Dakar" };
console.log(obj2); // { nom: 'Awa', age: 25, ville: 'Dakar' }
```

Partie I – ES6+ : Concepts Modernes

4. Template Literals (Backticks ``)

Permet d'insérer des variables ou expressions dans des chaînes de caractères.

```
const nom = "Harouna";
console.log(`Bonjour ${nom}, bienvenue dans le cours !`);
```

Partie I – ES6+ : Concepts Modernes

5. Modules (import/export)

Permet d'organiser le code en plusieurs fichiers réutilisables.

```
// fichier utils.js
export function doubler(x) {
    return x * 2;
}
```

```
// fichier main.js
import { doubler } from "./utils.js";
console.log(doubler(3)); // 6
```

Partie I – ES6+ : Concepts Modernes

6. Map et Set

1 Set

Un **Set** est une **collection d'éléments uniques** (pas de doublons).

- Il conserve l'ordre d'insertion.
- Utile pour éliminer les doublons dans un tableau.

```
const visiteurs = new Set(["Ali", "Awa", "Ali"]);
console.log(visiteurs.size); // 2 (Ali n'est compté qu'une fois)
console.log(visiteurs.has("Awa")); // true
```

Exemple avancé – Suppression des doublons dans un tableau :

```
const nombres = [1, 2, 3, 2, 1, 4];
const uniques = [...new Set(nombres)];
console.log(uniques); // [1, 2, 3, 4]
```

Partie I – ES6+ : Concepts Modernes

6. Map et Set

2 Map

Un **Map** est une **collection de paires clé → valeur**.

- Contrairement aux objets classiques, les clés peuvent être **de tout type** (objets, fonctions, etc.).
- Maintient aussi l'ordre d'insertion.

Exemple simple :

```
const notes = new Map();
notes.set("maths", 18);
notes.set("anglais", 15);

console.log(notes.get("maths")); // 18
console.log(notes.size); // 2
```

Exemple avancé – Objet comme clé :

```
const user1 = { nom: "Ali" };
const user2 = { nom: "Awa" };

const scores = new Map();
scores.set(user1, 20);
scores.set(user2, 15);

console.log(scores.get(user1)); // 20
```

Partie 2 – Fonctions Avancées & *this*

I. Arrow Functions

Fonctions anonymes courtes qui ne créent pas leur propre this.

```
const noms = ["Ali", "Awa"];
const resultats = noms.map(n => n.toUpperCase());
```

Partie 2 – Fonctions Avancées & *this*

2. Closures

Une fonction interne peut se souvenir des variables de la fonction externe, même après son exécution.

```
function compteur() {
  let count = 0;
  return function () {
    return ++count;
  };
}

const inc = compteur();
console.log(inc()); // 1
console.log(inc()); // 2
```

Partie 2 – Fonctions Avancées & *this*

3. Règles de base pour *this*

Ordre de priorité (du plus fort au plus faible) :

1. **new binding** : dans un constructeur appelé avec new, this = **nouvelle instance**.
2. **Binding explicite** : fn.call(obj), fn.apply(obj), fn.bind(obj) → this = **obj**.
3. **Binding implicite** : obj.meth() → this = **obj** (l'objet avant le point).
4. **Binding par défaut** : hors strict mode → window (navigateur) ; en strict → undefined.

⚠ **Fonctions fléchées** n'ont pas leur propre this : elles capturent le this **lexical** environnant.
call/apply/bind **ne changent pas** le this d'une fléchée.

Partie 2 – Fonctions Avancées & *this*

4. **call et apply : appel immédiat avec *this***

Même effet, syntaxe différente pour les arguments.

```
function hello(greet, punct) {  
    // 'this' n'est pas défini par la fonction elle-même :  
    // il dépend de *comment* on l'appelle (call/apply/bind, new, etc.)  
    return `${greet}, je suis ${this.name}${punct}`;  
}
```

```
const bob = { name: "Bob" };  
  
// Appel immédiat en fixant explicitement 'this' à 'bob'.  
// Les arguments sont passés *séparément*.  
hello.call(bob, "Salut", "!"); // → "Salut, je suis Bob!"
```

```
// Même chose mais avec 'apply' : on passe les arguments dans un *tableau*.  
hello.apply(bob, ["Salut", ""]); // → "Salut, je suis Bob!"
```

Partie 2 – Fonctions Avancées & *this*

5. bind : créer une *nouvelle fonction avec this (et args) figés*

```
function salut(greeting) {  
    // "this" dépendra de la façon dont on appelle la fonction  
    console.log(greeting + " " + this.nom);  
}  
  
// On crée un objet "perso" avec une propriété "nom"  
const perso = { nom: "Alice" };  
  
// On utilise "bind" pour créer une nouvelle fonction "salutAlice"  
// - "this" est fixé à l'objet "perso"  
// - le premier argument "greeting" est fixé à "Bonjour"  
const salutAlice = salut.bind(perso, "Bonjour");  
  
// Comme "this" pointe sur { nom: "Alice" } et "greeting" vaut "Bonjour",  
// ça affiche : "Bonjour Alice"  
salutAlice(); // Bonjour Alice
```

exo

Partie 3 – Objets & Classes

I. Objet littéral & méthodes

Un **objet littéral** est créé directement avec des accolades : il regroupe des paires clé : valeur.

```
const user = {  
    name: "Alice",  
    age: 28  
};
```

- **Clés** : chaînes (implicites) ou noms valides.
- **Valeurs** : n'importe quel type (nb, chaîne, bool, objet, fonction...).

Partie 3 – Objets & Classes

I. Objet littéral & méthodes

Raccourcis modernes

- **Shorthand propriété** (quand la variable a le même nom que la clé) :

```
const name = "Alice";
const user = { name }; // équivaut à { name: name }
```

Clé calculée :

```
const field = "age";
const user = { [field]: 28 }; // { age: 28 }
```

Partie 3 – Objets & Classes

I. Objet littéral & méthodes

Méthodes

- Une **méthode** est une fonction stockée comme valeur d'une propriété.

```
const user = {
  name: "Alice",
  // syntaxe classique
  greet: function () {
    return `Salut, je suis ${this.name}`;
  },
  // syntaxe abrégée (ES6)
  info() {
    return `${this.name} a un super profil.`;
  },
};
user.greet(); // "Salut, je suis Alice"
```

Partie 3 – Objets & Classes

I. Objet littéral & méthodes

this dans les méthodes

- Dans une méthode définie avec function (ou la syntaxe abrégée), this pointe vers **l'objet avant le point** à l'appel.
- **Évite les fonctions fléchées** comme méthodes quand tu as besoin de this : elles *capturent* le this environnant et ne rebindent pas sur l'objet.

```
const bad = {  
    name: "Bob",  
    say: () => `Je suis ${this.name}` // this ≠ bad → undefined en général  
};
```

Partie 3 – Objets & Classes

I. Objet littéral & méthodes

Getters / Setters

Permettent de lire/écrire comme des propriétés tout en exécutant du code.

```
const person = {
    first: "Ada",
    last: "Lovelace",
    get full() { return `${this.first} ${this.last}`; },
    set full(v) {
        const [f, l] = v.split(" ");
        this.first = f; this.last = l ?? "";
    }
};
person.full;           // "Ada Lovelace"
person.full = "Grace Hopper";
```

Partie 3 – Objets & Classes

2. Prototype & héritage

Prototype : la base

En JS, chaque objet possède un lien caché vers un autre objet qu'on appelle **prototype**. C'est comme un “plan” ou un “modèle” que l'objet peut utiliser si une propriété n'est pas trouvée directement.

```
const person = {  
    greet() {  
        return "Salut !";  
    }  
};  
  
const user = Object.create(person); // user hérite de person  
user.name = "Alice";  
  
console.log(user.name); // "Alice" (trouvé sur user)  
console.log(user.greet()); // "Salut !" (pas trouvé sur user → cherché dans prototype:  
person)
```

Ici :

- `user` n'a pas la méthode `greet`.
- JS remonte dans son **prototype (person)** et la trouve.

Partie 3 – Objets & Classes

2. Prototype & héritage

Prototype des fonctions constructrices

Avant class, on utilisait des **fonctions constructrices** avec prototype.

```
function Person(name) {
  this.name = name;
}

// On ajoute une méthode au prototype
Person.prototype.sayHello = function() {
  return `Bonjour, je suis ${this.name}`;
};

const p1 = new Person("Bob");
console.log(p1.sayHello()); // "Bonjour, je suis Bob"
```

Avantage : toutes les instances partagent la même méthode via le prototype (économie mémoire).

Partie 3 – Objets & Classes

2. Prototype & héritage

Héritage de prototype (chaîne)

Le système d'héritage en JS est basé sur une **chaîne de prototypes** :

```
const grandParent = { role: "grand-parent" };
const parent = Object.create(grandParent);
parent.role = "parent";

const child = Object.create(parent);
child.role = "child";

console.log(child.role); // "child" (sur child)
console.log(child.__proto__.role); // "parent"
console.log(child.__proto__.__proto__.role); // "grand-parent"
```

JS cherche toujours une propriété :

1. sur l'objet lui-même,
2. sinon dans son prototype,
3. puis dans le prototype du prototype... jusqu'à Object.prototype.

Partie 3 – Objets & Classes

3. Classes ES6

- Les **classes JS** (ES2015) sont du **sucré syntaxique** sur l'**héritage par prototype** : sous le capot, ce sont des **fonctions spéciales**.
- Elles se définissent **par déclaration** (class A {}) ou **par expression** (const A = class {}).
- Elles n'ajoutent pas un nouveau modèle d'objet, seulement une **syntaxe plus lisible**

Définition & constructeur

```
class Person {  
    constructor(name) { // appelé par 'new'  
        this.name = name; // propriétés d'instance  
    }  
    sayHello() { // méthode du prototype (partagée)  
        return `Bonjour, je suis ${this.name}`;  
    }  
}  
  
const p = new Person("Ada");  
p.sayHello(); // "Bonjour, je suis Ada"
```

Partie 3 – Objets & Classes

3. Classes ES6

Héritage (extends, super)

```
class Student extends Person {
  constructor(name, level) {
    super(name);           // appelle Person.constructor
    this.level = level;
  }
  sayHello() {            // override
    return `${super.sayHello()} (niveau ${this.level})`;
  }
}

new Student("Alice", "M2").sayHello();
// "Bonjour, je suis Alice (niveau M2)"
```

Partie 3 – Objets & Classes

3. Classes ES6

Champs de classe (publics/privés) & méthodes privées

```
class Counter {  
    // champs d'instance  
    value = 0;                      // public  
    #step = 1;                       // privé (ES2022+)  
  
    // méthode publique  
    inc() { this.#bump(); return this; }  
  
    // méthode privée  
    #bump() { this.value += this.#step; }  
}
```

- **# = privé** (inaccessible hors de la classe).
- Les champs sont initialisés **avant** l'exécution du *constructor*.

Partie 3 – Objets & Classes

3. Classes ES6

Méthodes statiques & champs statiques

```
class MathTools {  
    static PI = 3.14159; // champ statique  
    static clamp(v, min, max) { // méthode statique  
        return Math.min(max, Math.max(min, v));  
    }  
}
```

```
MathTools.clamp(10, 0, 5); // 5  
MathTools.PI; // 3.14159
```

Statique = attaché à la **classe**, pas aux instances.

Partie 3 – Objets & Classes

3. Classes ES6

Getters / Setters (sur le prototype)

```
class Person {  
    constructor(first, last) { this.first = first; this.last = last; }  
    get full() { return `${this.first} ${this.last}`; }  
    set full(v) { [this.first, this.last] = v.split(" "); }  
}  
  
const p = new Person("Ada", "Lovelace");  
p.full; // "Ada Lovelace"  
p.full = "Grace Hopper";
```

Partie 3 – Objets & Classes

3. Classes ES6

`instanceof`

Teste si un objet hérite du **prototype** d'un constructeur.

```
class Person {}  
const p = new Person();  
p instanceof Person;          // true  
p instanceof Object;         // true (via la chaîne de prototypes)
```

Partie 3 – Objets & Classes

3. Classes ES6

new.target

Donne le **constructeur réellement invoqué avec new** (utile pour classes abstraites ou pour savoir si une fonction est appelée avec new).

```
class Base {  
    constructor() {  
        if (new.target === Base) throw new Error("Base est abstraite");  
    }  
}
```

```
class Child extends Base {}  
new Child(); // OK  
// new Base(); // Error
```

Partie 3 – Objets & Classes

3. Classes ES6

Dans les fonctions constructrices

```
function Foo() {  
  if (!new.target) return new Foo(); // permet d'appeler sans 'new'  
  this.x = 1;  
}
```

```
const a = Foo();    // 'new' implicite  
const b = new Foo();
```

- *new.target* est *undefined* si la fonction est appelée **sans** new.
- Les **fonctions fléchées** n'ont pas de *new.target* propre (elles capturent celui de l'environnement).

Partie 3 – Objets & Classes

3. Classes ES6

Redéfinir `toString`

```
class Person {  
    constructor(first, last) {  
        this.first = first;  
        this.last = last;  
    }  
    toString() {  
        return `${this.first} ${this.last}`;  
    }  
}
```



```
String(new Person("Ada", "Lovelace")); // "Ada Lovelace"
```

Asynchronisme & Appels API

Comprendre les Promesses et `async/await`

Savoir interagir avec une API (`fetch`, `axios`)

Gérer les erreurs et les appels multiples

Event loop, microtask vs macrotask

L'event loop exécute la **pile** de call stack, puis vide **micro-tasks** (Promesses, queueMicrotask) **avant** les **macro-tasks** (setTimeout, DOM events).

```
console.log("A");                                // 1) synchro → s'affiche tout de suite
setTimeout(() => console.log("B (macro)'), 0); // 2) planifie une **macrotâche** (timers)
Promise.resolve().then(() => console.log("C (micro)")); // 3) planifie une **microtâche**
console.log("D");                                // 4) synchro → s'affiche tout de suite
// Ordre: A, D, C, B
```

Empiler beaucoup de microtâches peut “affamer” les macrotâches (elles attendent que la micro-queue soit vidée

Promesses

Une Promesse représente une valeur **future** : états pending → fulfilled | rejected.

Chaining & propagation d'erreurs

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(r => r.ok ? r.json() : Promise.reject(new Error(r.status)))
  .then(users => {
    console.log(users); // tableau de 10 users
  })
  .catch(err => console.error("KO", err))
  .finally(() => console.log("done"));
```

Composition

- `Promise.all([p1, p2])` → **toutes** doivent réussir (rejette au 1er échec)
- `Promise.allSettled([...])` → résume succès/échecs **sans rejeter**
- `Promise.race([...])` → 1er résultat (succès/échec)
- `Promise.any([...])` → 1er **succès** (rejette si **tous** échouent)

Async/await

Sucré syntaxique sur Promesses ; await **bloque** la fonction `async`, pas le thread.

Variante `async/await`

```
async function loadUsers() {  
    try {  
        const res = await fetch("https://jsonplaceholder.typicode.com/users");  
        if (!res.ok) throw new Error(res.status);  
        const users = await res.json();  
        console.log(users);  
    } catch (err) {  
        console.error("KO", err);  
    } finally {  
        console.log("done");  
    }  
}
```

Points clés

- `fetch` n'échoue (`reject`) que pour une **erreur réseau**. Pour 404/500, il faut tester `res.ok`.
- `await` peut être utilisé plusieurs fois de suite (réponse → JSON).

Séquentiel vs parallèle

latence de A **puis** celle de B.

```
// Séquentiel (plus lent)
const a = await fetchA();
const b = await fetchB();
```

`fetchA()` et `fetchB()` partent en même temps.

```
// Parallèle (plus rapide)
const [a2, b2] = await Promise.all([fetchA(), fetchB()]);
```

HTTP côté client

HTTP côté client : fondamentaux qui comptent

- **Méthodes** : GET (idempotent), POST, PUT/PATCH, DELETE
- **Statuts** : 2xx succès, 4xx erreur client, 5xx serveur
- **En-têtes** : Content-Type: application/json, Authorization: Bearer <token>
- **CORS** : pré-requête (OPTIONS) si non simple request
- **Query/pagination** : ?page=2&limit=20 (ou cursor=...)

HTTP côté client

Axios

```
import axios from "axios";

async function loadUsers() {
  try {
    const { data: users } = await axios.get(
      "https://jsonplaceholder.typicode.com/users",
      {
        timeout: 8000,
        headers: { Accept: "application/json" }
      }
    );
    console.log(users);
  } catch (err) {
    if (axios.isAxiosError(err)) {
      console.error("KO", err.response?.status, err.message);
    } else {
      console.error("KO", err);
    }
  } finally {
    console.log("done");
  }
}
```

HTTP côté client

Atelier