RP - 323 - Programmation fonctionnelle

> [!TIP] **Référence Javascript:** https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference
> **Tester du code JS** : https://runjs.app/play
> **Convertir en PDF** : https://marketplace.visualstudio.com/items?itemName=manuth.markdown-converter

Table des matières

Introduction

Dans ce module, je vais apprendre la programmation fonctionnelle en JavaScript et comprendre ses différences avec la programmation impérative.

Pendant ce module, je vais :

- Découvrir les paradigmes de programmation et l'intérêt de la programmation fonctionnelle
- Utiliser les fonctions de base comme map, filter et reduce
- Appliquer des concepts comme fonctions pures, immuabilité, closures, currying et récursion
- Mettre en pratique les bonnes pratiques et patterns, et apprendre à refactoriser du code

Ce rapport permettra de présenter les notions vues durant le module, en expliquant les méthodes et en donnant des exemples concrets de leur utilisation.

Opérateurs javascript super-cooool 😎

opérateur `?:`

> L'expression `question?valeur1:valeur2` retournera `valeur1` si `question` vaut `true` sinon elle retournera `valeur2`.

```
const age = 15;
const resultat = age >= 18 ? 'majeur' : 'mineur'; // 'mineur'
```

opérateur `??`

Cet opérateur logique se nomme l'opérateur de "coalescence des nuls".

> Renvoie son opérande de droite lorsque son opérande de gauche vaut `null` ou `undefined` et qui renvoie son opérande de gauche sinon.

```
const foo1 = null ?? 'default'; // "default"
const foo2 = 0 ?? 42; // 0
```

> [!CAUTION] Contrairement à l'opérateur logique OU (`||`), l'opérande de gauche sera également renvoyé s'il s'agit d'une valeur équivalente à `false` et pas seulement `null` et `undefined`.
>
> ⚠ En d'autres termes **ATTENTION** ‼ lors de l'utilisation de `||` pour fournir une valeur par défaut à une variable, car on peut rencontrer des comportements inattendus lorsqu'on considère certaines valeurs comme correctes et utilisables (par exemple une chaine vide `''` ou `0`) ‼

```

```

```
const foo3 = 0 || 42; // 42 => ATTENTION !
const foo4 = 1 || 42; // 1
const foo5 = null || 'salut !'; // 'salut !'
const foo6 = '' || 'salut !'; // 'salut !' => ATTENTION !
```

opérateur ??=

Cet opérateur logique se nomme l'opérateur d'affectation de "coalescence des nuls", également connu sous le nom d'opérateur affectation logique nulle.

> Évalue l'opérande de droite et l'attribue à gauche **UNIQUEMENT si l'opérande de gauche est nulle** (`null` ou `undefined`).

```
const a = { duration: 50 };
a.duration ??= 10; // pas fait
a.speed ??= 25; // fait => { duration: 50, speed: 25 }
```

opérateur de décomposition 'spread' ...

L'opérateur de décomposition spread `...` permet de décomposer un itérable (comme un tableau) en en ses éléments distincts. Cela permet de rapidement copier tout ou une partie d'un tableau existant dans un autre tableau ou d'en extraire facilement des parties.

```
// Combiner des valeurs existantes dans un nouveau tableau
const numbersOne = [1, 2, 3];
const numbersTwo = [4, 5, 6];
const numbersCombined = [...numbersOne, ...numbersTwo];

// Extraire uniquement ce qui est utile d'un tableau
const numbers = [1, 2, 3, 4, 5, 6];
const [one, two, ...rest] = numbers;

// Mariage d'objets avec mise à jour :-)
const myVehicle = {
    brand: 'Ford',
    model: 'Mustang',
    color: 'red',
};
const updateMyVehicle = {
    type: 'car',
    year: 2021,
    color: 'yellow',
```

```
};
const myUpdatedVehicle = { ...myVehicle, ...updateMyVehicle };
```

Déstructuration

L'opérateur de décomposition spread `...` sert aussi à isoler certains éléments afin de les utiliser ensuite, et de **mettre le reste** d'un coup ailleurs.

```
const valeurs = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const [a, b, ...c] = valeurs;
console.log(a); // 1
console.log(b); // 2
console.log(c); // [3, 4, 5, 6, 7, 8, 9, 10]
```

Date et Heure

Lien vers la documentation officielle :
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Date

Obtenir la date et/ou heure actuelle

```
const maintenant = new Date(); // Obtenir l'un comme l'autre

console.log(maintenant.toLocaleDateString()); // ex: "06.06.2025"
console.log(maintenant.toLocaleTimeString()); // ex: "15:23:42"

const jour = maintenant.getDate();
const mois = maintenant.getMonth() + 1; // Attention : janvier = 0
const annee = maintenant.getFullYear();
const heure = maintenant.getHours();
const minute = maintenant.getMinutes();
const seconde = maintenant.getSeconds();
console.log(`${jour}/${mois}/${annee} - ${heure}h${minute}`);

// Au format ISO (standard international)
console.log(maintenant.toISOString()); // ex: "2025-06-06T13:23:42.123Z"
```

Math

Lien vers la documentation officielle :
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Math

`Math.PI` - la constante π

Représente la valeur de π (pi), soit environ 3.14159. Sert pour tous les calculs impliquant des cercles ou des rotations.

```
// Calculer la circonférence d'un cercle
let r = 5;
let c = 2 * Math.PI * r; // 31.4159

// Calculer l'aire d'un cercle
let a = Math.PI * r ** 2; // 78.5398

// Convertir des degrés en radians
let deg = 180;
let rad = deg * (Math.PI / 180); // π radians
```

`Math.abs()` - la |valeur absolue| d'un nombre

Retourne la valeur positive d'un nombre, peu importe son signe.

```
Math.abs(-7);   // 7
Math.abs(7);    // 7
Math.abs(-3.5); // 3.5
```

`Math.pow()` - élever à une puissance

Permet d'élever un nombre à la puissance d'un autre.

```
Math.pow(2, 3);  // 8  → 2³
Math.pow(5, 2);  // 25 → 5²
Math.pow(9, 0.5); // 3 → racine carrée
```

```
// Même chose avec l'opérateur moderne :
2 ** 3; // 8
```

`Math.min()` - plus petite valeur

Math.min() retourne la plus petite valeur parmi les arguments passés. Si aucun argument n'est fourni → retourne Infinity.

```
// Plus petit nombre parmi des valeurs
console.log(Math.min(3, 7, 2, 9)); // 2

// Avec des nombres négatifs
console.log(Math.min(-5, -2, -10)); // -10

// Trouver le min dans un tableau avec spread
const notes = [12, 15, 18, 10];
console.log(Math.min(...notes)); // 10

// Comparer des résultats calculés
const a = 5, b = 8, c = 3;
console.log(Math.min(a + b, c * 4)); // 12

// Aucun argument → Infinity
console.log(Math.min()); // Infinity
```

`Math.max()` - plus grande valeur

Math.max() retourne la plus grande valeur parmi les arguments passés. Si aucun argument n'est fourni → retourne -Infinity.

```
// Plus grand nombre parmi des valeurs
console.log(Math.max(3, 7, 2, 9)); // 9

// Avec des nombres négatifs
console.log(Math.max(-5, -2, -10)); // -2

// Trouver le max dans un tableau avec spread
const notes = [12, 15, 18, 10];
console.log(Math.max(...notes)); // 18

// Comparer des résultats calculés
```

```
const a = 5, b = 8, c = 3;
console.log(Math.max(a + b, c * 4)); // 13

// Aucun argument → -Infinity
console.log(Math.max()); // -Infinity
```

`Math.ceil()` - arrondir à la prochaine valeur entière la plus proche

Arrondit un nombre vers le haut (à l'entier supérieur).

```
Math.ceil(4.2);   // 5
Math.ceil(7.001); // 8
Math.ceil(-3.4);  // -3

// Utile pour calculer le nombre de pages nécessaires
let total = 23;
let parPage = 5;
let pages = Math.ceil(total / parPage); // 5 pages
```

`Math.floor()` - arrondir à la précédente valeur entière la plus proche

Arrondit un nombre vers le bas (à l'entier inférieur).

```
Math.floor(4.9);   // 4
Math.floor(7.99);  // 7
Math.floor(-3.2);  // -4

// Utile pour obtenir un index de tableau entier
let index = Math.floor(Math.random() * 10); // entre 0 et 9
```

`Math.round()` - arrondir à la valeur entière la plus proche

Math.round rend l'entier le plus proche de la valeur donné

```
Math.round(4.3); // → 4
Math.round(4.5); // → 5
Math.round(4.7); // → 5
Math.round(-2.5); // → -2
```

```
Math.round(18.756 * 10) / 10; // → 18.8
Math.round(3.1415 * 100) / 100; // → 3.14
Math.round(3.34 * 2) / 2 // → 3.5
```

`Math.trunc()` - supprime la virgule et retourne la partie entière d'un nombre

Enlève tout ce qui est après la virgule sans arrondir. Garde seulement la partie entière.

```
Math.trunc(4.9);   // 4
Math.trunc(-3.7);  // -3
Math.trunc(0.99);  // 0

// Utile pour ignorer les décimales sans modifier la valeur
let prix = 12.75;
let euros = Math.trunc(prix); // 12
```

`Math.sqrt()` - la raçine carrée d'un nombre

Math.sqrt(x) retourne la racine carrée d'un nombre x. Si x est négatif, le résultat sera NaN.

```
//Racine carrée simple
console.log(Math.sqrt(9)); // 3
console.log(Math.sqrt(16)); // 4

//Racine carrée d'un nombre décimal
console.log(Math.sqrt(2)); // 1.4142135623730951

//Vérifier si un nombre est un carré parfait
const n = 25;
console.log(Math.sqrt(n) % 1 === 0); // true -> carré parfait

//Racine carrée d'un nombre négatif
console.log(Math.sqrt(-4)); // NaN
```

`Math.random()` - générer un nombre aléatoire entre 0.0 (compris) et 1.0 (non compris)

Math.random() retourne un nombre décimal aléatoire ≥ 0 et < 1.

```
//Nombre aléatoire entre 0 et 1
console.log(Math.random()); // ex: 0.482345

//Nombre aléatoire entre 0 et 10
const max = 10;
console.log(Math.random() * max); // ex: 7.8234

//Nombre entier aléatoire entre 0 et 9
console.log(Math.floor(Math.random() * 10)); // ex: 4

//Nombre entier aléatoire entre min et max inclus
function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
}
console.log(getRandomInt(5, 15)); // ex: 12

//Choisir un élément aléatoire dans un tableau
const fruits = ["pomme", "banane", "cerise"];
const randomFruit = fruits[Math.floor(Math.random() * fruits.length)];
console.log(randomFruit); // ex: "banane"
```

JSON

Lien vers la documentation officielle :
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/JSON

`JSON.stringify()` - transformer un objet Javascript en JSON

Convertit un objet ou un tableau JavaScript en chaîne JSON (texte). Utile pour sauvegarder ou envoyer des données.

```
let obj = { nom: "Eri", age: 18 };
let jsonObj = JSON.stringify(obj);
// '{"nom":"Eri","age":18}'

// Avec un tableau
let fruits = ["pomme", "banane", "cerise"];
let jsonArray = JSON.stringify(fruits);
// '["pomme","banane","cerise"]'
```

```
// Exemple pratique : sauvegarder des données
localStorage.setItem("fruits", JSON.stringify(fruits));
```

`JSON.parse()` - transformer du JSON en objet Javascript

Convertit une chaîne JSON (texte) en objet ou tableau JavaScript utilisable. Inverse de JSON.stringify()

```
let jsonObj = '{"nom":"Eri","age":18}';
let obj = JSON.parse(jsonObj);
// { nom: "Eri", age: 18 }

let jsonArray = '["pomme","banane","cerise"]';
let fruits = JSON.parse(jsonArray);
// ["pomme", "banane", "cerise"]

// Exemple pratique : lecture depuis localStorage
let saved = localStorage.getItem("fruits");
let data = JSON.parse(saved);
```

Chaînes de caractères

Lien vers la documentation officielle :
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/String

`split()` - un ciseau qui coupe une chaîne là où un caractère apparaît et produit un tableau

Sépare une chaîne de caractères là où un séparateur apparaît et retourne un tableau des morceaux.

```
let phrase = "je,suis,un,chat";
let mots = phrase.split(",");
// ["je", "suis", "un", "chat"]

let texte = "bonjour le monde";
let lettres = texte.split("");
```

```
// ["b","o","n","j","o","u","r"," ","l","e"," ","m","o","n","d","e"]

// Utile pour traiter des données CSV
let csv = "nom;age;ville";
let parts = csv.split(";");
// ["nom", "age", "ville"]
```

`trim()`, `trimStart()` et `trimEnd()` - épuration des espaces en trop dans une chaîne (trimming)

- trim() → enlève les espaces au début et à la fin d'une chaîne.
- trimStart() → enlève les espaces au début seulement.
- trimEnd() → enlève les espaces à la fin seulement.

```
let texte = "   hello world   ";

texte.trim();      // "hello world"
texte.trimStart(); // "hello world   "
texte.trimEnd();   // "   hello world"

// Utile pour nettoyer des saisies utilisateur
let input = "   Diogo  ";
let clean = input.trim(); // "Diogo"
```

`padStart()` et `padEnd()` - aligner le contenu dans une chaîne de caractères

- padStart(n, char) → ajoute des caractères au début pour atteindre une longueur n.
- padEnd(n, char) → ajoute des caractères à la fin pour atteindre une longueur n. Utile pour formater des textes ou des nombres.

```
let num = "7";
num.padStart(3, "0"); // "007"
num.padEnd(3, ".");   // "7.."

let mot = "chat";
mot.padStart(6, " "); // "  chat" → alignement à droite
mot.padEnd(6, " ");   // "chat  " → alignement à gauche
```

Console

Lien vers la documentation officielle : https://developer.mozilla.org/fr/docs/Web/API/console

`console.log()` - Afficher un message sur la console

```
console.log('Coucou !'); // Coucou !
```

`console.info()`, `warn()` et `error()` - Afficher un message sur la console (filtrables)

- console.info() → message d'information standard.
- console.warn() → avertissement, souvent avec un triangle jaune.
- console.error() → message d'erreur, souvent en rouge. Pratique pour déboguer et distinguer les types de messages.

```
console.info("Le script a démarré.");
console.warn("La variable x n'est pas initialisée.");
console.error("Impossible de charger le fichier.");
```

`console.table()` - Afficher tout un tableau ou un objet sur la console

Affiche les données sous forme de tableau lisible avec colonnes et lignes, pratique pour visualiser rapidement des objets ou tableaux complexes.

```
let fruits = ["pomme", "banane", "cerise"];
console.table(fruits);
/*
┌─────────┬──────────┐
│ (index) │ Values   │
├─────────┼──────────┤
│    0    │ 'pomme'  │
│    1    │ 'banane' │
│    2    │ 'cerise' │
└─────────┴──────────┘
```

```
*/

let users = [
  { nom: "Job", age: 20 },
  { nom: "Diogo", age: 17 }
];
console.table(users);
/*
┌─────────┬─────────┬─────┐
│ (index) │ nom     │ age │
├─────────┼─────────┼─────┤
│    0    │ 'Job'   │ 20  │
│    1    │ 'Diogo' │ 17  │
└─────────┴─────────┴─────┘
*/
```

`console.time()`, `timeLog()` et `timeEnd()` - Chronométrer une durée d'exécution

Permet de chronométrer combien de temps s'exécute un bloc de code.

- console.time("label") → démarre le chronomètre.
- console.timeLog("label") → affiche le temps écoulé sans arrêter.
- console.timeEnd("label") → affiche le temps écoulé et arrête le chronomètre.

```
console.time("boucle");
for(let i = 0; i < 1000000; i++) { /* traitement */ }
console.timeLog("boucle"); // temps écoulé intermédiaire
console.timeEnd("boucle"); // temps total écoulé
```

Tableaux

Lien vers la documentation officielle :
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Array

`forEach` - parcourir les éléments d'un tableau

Exécute une fonction pour chaque élément d'un tableau.

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

fruits.forEach((fruit, index) => {
  console.log(index, fruit);
});
// 0 "pomme"
// 1 "banane"
// 2 "cerise"

// Exemple pratique : ajouter un "!" à chaque élément
fruits.forEach((fruit, i, arr) => arr[i] = fruit + "!");
console.log(fruits); // ["pomme!", "banane!", "cerise!"]
```

`entries()` - parcourir les couples index/valeurs d'un tableau

Retourne un itérateur qui fournit [index, valeur] pour chaque élément du tableau.

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

for (const [index, fruit] of fruits.entries()) {
  console.log(index, fruit);
}
// 0 "pomme"
// 1 "banane"
// 2 "cerise"
```

`in` - parcourir les clés d'un tableau

Permet de parcourir les index (ou clés) d'un tableau ou les propriétés d'un objet.

```
et fruits = ["pomme", "banane", "cerise"];

for (let i in fruits) {
  console.log(i, fruits[i]);
}
// 0 "pomme"
// 1 "banane"
// 2 "cerise"

// Avec un objet
let user = { nom: "Diogo", age: 17 };
```

```
for (let key in user) {
  console.log(key, user[key]);
}
// "nom" "Diogo"
// "age" 17
```

`of` - parcourir les valeurs d'un tableau

Permet de parcourir directement les valeurs d'un tableau (ou d'un objet itérable) sans se soucier des indices.

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

for (let fruit of fruits) {
  console.log(fruit);
}
// "pomme"
// "banane"
// "cerise"

// Avec une chaîne de caractères
for (let lettre of "chat") {
  console.log(lettre);
}
// "c" "h" "a" "t"
```

`find()` - premier élément qui satisfait une condition

Parcourt un tableau et retourne le premier élément pour lequel la fonction renvoie true. Renvoie undefined si aucun élément ne correspond.

```
let nombres = [5, 12, 8, 130, 44];

// Trouver le premier nombre > 10
let premierGrand = nombres.find(n => n > 10);
console.log(premierGrand); // 12

// Avec des objets
let users = [
  { nom: "Job", age: 20 },
  { nom: "Diogo", age: 17 }
];
let majeur = users.find(u => u.age >= 20);
```

```
console.log(majeur); // { nom: "Job", age: 20 }
```

`findIndex()` - premier index qui satisfait une condition

Parcourt un tableau et retourne l'index du premier élément pour lequel la fonction renvoie true. Renvoie -1 si aucun élément ne correspond.

```
let nombres = [5, 12, 8, 130, 44];

// Trouver l'index du premier nombre > 10
let indexGrand = nombres.findIndex(n => n > 10);
console.log(indexGrand); // 1 (12)

// Avec des objets
let users = [
  { nom: "Diogo", age: 17 },
  { nom: "Job", age: 20 }
];
let indexMajeur = users.findIndex(u => u.age >= 20);
console.log(indexMajeur); // 1
```

`indexOf()` et `lastIndexOf()` - premier/dernier élément qui correspond

- indexOf(value) → retourne l'index du premier élément égal à value.
- lastIndexOf(value) → retourne l'index du dernier élément égal à value. Renvoie -1 si l'élément n'est pas trouvé.

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

fruits.indexOf("banane");     // 1
fruits.lastIndexOf("banane"); // 3

// Utilité : vérifier si un élément existe
if (fruits.indexOf("pomme") !== -1) {
  console.log("Pomme trouvée !");
}
```

`push()`, `pop()`, `shift()` et `unshift()` - ajouter/supprime au début/fin dans un tableau

- push(value) → ajoute un élément à la fin du tableau.
- pop() → supprime et retourne le dernier élément.
- shift() → supprime et retourne le premier élément.
- unshift(value) → ajoute un élément au début du tableau.

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

// Ajouter
fruits.push("cerise");   // ["pomme", "banane", "cerise"]
fruits.unshift("kiwi");  // ["kiwi", "pomme", "banane", "cerise"]

// Supprimer
fruits.pop();   // "cerise", fruits = ["kiwi", "pomme", "banane"]
fruits.shift(); // "kiwi", fruits = ["pomme", "banane"]
```

`slice()` - ne conserver que certaines lignes d'un tableau

slice permet de extraire une partie du tableau ou d'une chaine de charactère sans le modifié, on choisi un d'où il commence et/ou il fini

```
const fruits = ["pomme","banane","fraise","melon","orange"];

const example1 = fruits.slice(2); // ["fraise","melon","orange"]
const example2 = fruits.slice(0, 3); // ["pomme","banane","fraise"]
const example3 = fruits.slice(1, -1) // ["banane","fraise","melon"]
```

`splice()` - supprimer/insérer/remplacer des valeurs dans un tableau

Modifie le tableau sur place :

- Supprimer : splice(index, nombre)
- Ajouter : splice(index, 0, valeurs...)
- Remplacer : splice(index, nombre, valeurs...)

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

// Supprimer 1 élément à l'index 1
fruits.splice(1, 1); // ["pomme", "cerise"]
```

```
// Ajouter "kiwi" à l'index 1
fruits.splice(1, 0, "kiwi"); // ["pomme", "kiwi", "cerise"]

// Remplacer "cerise" par "mangue"
fruits.splice(2, 1, "mangue"); // ["pomme", "kiwi", "mangue"]
```

`concat()` - joindre deux tableaux

Crée un nouveau tableau en combinant deux (ou plusieurs) tableaux. Ne modifie pas les tableaux originaux.

```
let fruits = ["pomme", "banane"];
let legumes = ["carotte", "brocoli"];

let nourriture = fruits.concat(legumes);
// ["pomme", "banane", "carotte", "brocoli"]

// On peut enchaîner plusieurs tableaux
let boissons = ["eau", "jus"];
let tout = fruits.concat(legumes, boissons);
// ["pomme","banane","carotte","brocoli","eau","jus"]
```

`join()` - joindre des chaînes de caractères

Assemble les éléments d'un tableau en une chaîne, en les séparant par un séparateur choisi (par défaut ,).

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

fruits.join();       // "pomme,banane,cerise"
fruits.join(" - ");  // "pomme - banane - cerise"
fruits.join("");     // "pommebananecerise"

// Utile pour créer du texte ou du CSV
let csv = fruits.join(";"); // "pomme;banane;cerise"
```

`keys()` et `values()` - les clés/valeurs d'un objet

- Object.keys(obj) → retourne un tableau contenant toutes les clés de l'objet.
- Object.values(obj) → retourne un tableau contenant toutes les valeurs de l'objet.

```
let user = { nom: "Diogo", age: 17, ville: "Bulle" };

Object.keys(user);   // ["nom", "age", "ville"]
Object.values(user); // ["Diogo", 17, "Bulle"]

// Parcourir facilement un objet
for (let key of Object.keys(user)) {
  console.log(key, user[key]);
}
// nom Diogo
// age 17
// ville Bulle
```

`includes()` - vérifier si une valeur est présente dans un tableau

Renvoie true si le tableau contient la valeur donnée, false sinon.

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

fruits.includes("banane"); // true
fruits.includes("kiwi");   // false

// Utile pour vérifier rapidement avant d'ajouter ou traiter un élément
if (!fruits.includes("kiwi")) {
  fruits.push("kiwi");
}
```

`every()` et `some()` - vérifier si plusieurs valeurs sont toutes/quelques présentes dans un tableau

- every(callback) → retourne true si tous les éléments satisfont la condition.
- some(callback) → retourne true si au moins un élément satisfait la condition.

```
let nombres = [2, 4, 6, 8];

// Tous les nombres sont pairs ?
nombres.every(n => n % 2 === 0); // true

// Au moins un nombre est > 5 ?
nombres.some(n => n > 5); // true
```

```
// Utilité : validation rapide d'un tableau
let ages = [12, 18, 20];
let tousMajeurs = ages.every(a => a >= 18); // false
let auMoinsUnMajeur = ages.some(a => a >= 18); // true
```

`fill()` - remplir un tableau avec des valeurs

Remplit un tableau existant avec une même valeur sur tout ou partie du tableau. Modifie le tableau en place.

```
let tab = [0, 0, 0, 0];

// Remplir tout le tableau avec 5
tab.fill(5); // [5, 5, 5, 5]

// Remplir uniquement de l'index 1 à 3
tab.fill(1, 1, 3); // [5, 1, 1, 5]

// Utile pour initialiser ou réinitialiser un tableau
let lettres = Array(4).fill("a"); // ["a", "a", "a", "a"]
```

`flat()` - aplatir un tableau

flat() permet de transformer un tableau qui contient des sous-tableau en un seul tableau, il peut aplatir plusieur de niveaux tableaux

```
const nombres = [1,2, [3,4,[5,6]]];

const example1 = nombres.flat(); // [1, 2, 3, 4, [5, 6]]
const exaple2 = nombres.flat(2); // [1, 2, 3, 4, 5, 6]
```

`sort()` - pour trier un tableau

Trie les éléments d'un tableau en place. Par défaut, trie en ordre alphabétique. Pour trier des nombres, il faut fournir une fonction de comparaison.

```
let fruits = ["banane", "pomme", "cerise"];
fruits.sort();
// ["banane", "cerise", "pomme"]

let nombres = [10, 2, 30, 4];
```

```
// Tri par défaut (ordre alphabétique)
nombres.sort(); // [10, 2, 30, 4] → incorrect pour les nombres

// Tri numérique croissant
nombres.sort((a, b) => a - b); // [2, 4, 10, 30]

// Tri numérique décroissant
nombres.sort((a, b) => b - a); // [30, 10, 4, 2]

nombres.sort((a, b) => a.localeCompare(b)); // Tri alphabétique correct
```

`map()` - tableau avec les résultats d'une fonction

map() permet de transformer chaque élément d'un tableau avec une fonction et de renvoyer un nouveau tableau contenant les résultats. Le tableau original ne change pas

```
const nombres = [1, 2, 3, 4];
const doubles = nombres.map(x => x * 2);

console.log(doubles); // [2, 4, 6, 8]

const villes = [
  { nom: "Lausanne", canton: "VD" },
  { nom: "Genève", canton: "GE" }
];
const cantons = villes.map(v => v.canton);
console.log(cantons); // ["VD", "GE"]

const dataMotos = [
  {
    marque: "Yamaha",
    modeles: [
      { nom: "R1", prix: 20000 },
      { nom: "MT-07", prix: 8000 }
    ]
  },
  {
    marque: "Honda",
    modeles: [
      { nom: "CBR600RR", prix: 12000 },
      { nom: "CB500F", prix: 6500 }
    ]
  }
];

const resultat = dataMotos.map(moto => ({
```

```
  marque: moto.marque,
  modele: moto.modeles.map(modele => modele.nom)
}));
console.log(resultat);
/*
[
  { marque: "Yamaha", modele: ["R1", "MT-07"] },
  { marque: "Honda",  modele: ["CBR600RR", "CB500F"] }
]
*/
```

`filter()` - tableau avec les éléments passant un test

Retourne un nouveau tableau contenant uniquement les éléments pour lesquels la fonction renvoie true. Ne modifie pas le tableau original.

```
let nombres = [1, 2, 3, 4, 5, 6];

// Garder les nombres pairs
let pairs = nombres.filter(n => n % 2 === 0);
// [2, 4, 6]

// Garder les utilisateurs majeurs
let users = [
  { nom: "Eri", age: 18 },
  { nom: "Job", age: 20 },
  { nom: "Diogo", age: 17 }
];
let majeurs = users.filter(u => u.age >= 18);
// [{ nom: "Eri", age: 18 }, { nom: "Job", age: 20 }]
```

`groupBy()` - regroupe les éléments d'un tableau selon un règle

Crée un objet où les clés correspondent à une catégorie définie par une fonction, et les valeurs sont des tableaux contenant les éléments correspondant à chaque catégorie.

```
let nombres = [6, 7, 8, 9, 10];

// Regrouper selon pair ou impair
let groupes = nombres.groupBy(n => (n % 2 === 0 ? "pair" : "impair"));
/*
{
```

```
  pair: [6, 8, 10],
  impair: [7, 9]
}
*/

// Avec des objets
let users = [
  { nom: "Eri", age: 18 },
  { nom: "Job", age: 20 },
  { nom: "Diogo", age: 17 }
];
let parAge = users.groupBy(u => u.age >= 18 ? "majeur" : "mineur");
/*
{
  majeur: [{ nom: "Eri", age: 18 }, { nom: "Job", age: 20 }],
  mineur: [{ nom: "Diogo", age: 17 }]
}
*/
```

`flatMap()` - chaînage de map() et flat()

flatMap() combine map + flat(1) , il applique une fonction à chaque élément d'un tableau et aplatit le résultat d'un niveau.

```
const nombres = [1, 2, 3];
const result = nombres.flatMap(x => [x, x * 2]);

console.log(result); // [1, 2, 2, 4, 3, 6]

const produits = [
  { nom: "Pomme", categories: ["fruit", "bio"] },
  { nom: "Carotte", categories: ["légume", "bio"] }
];

const categories = produits.flatMap(p => p.categories);
console.log(categories); // ["fruit", "bio", "légume", "bio"]
```

`reduce()` et `reduceRight()` - réduire un tableau à une seule valeur

permet de transformer un tableau en une seule valeur (somme, moyenne, texte, objet, etc.) en appliquant une fonction sur chaque élément. (reduce de gauche a droite/ reduceRight de droite à gauche)

```
// reduce(callback, valeurInitiale)
const total = [1, 2, 3, 4].reduce((acc, val) => acc + val, 0);
// acc = accumulateur, val = valeur courante
console.log(total); // 10

// reduceRight() fait la même chose mais de droite à gauche
const mot = ["a", "b", "c"].reduceRight((acc, val) => acc + val, "");
console.log(mot); // "cba"

const nombres = [10, 25, 3, 47, 19];
const max = nombres.reduce((acc, val) => (val > acc ? val : acc));
console.log(max); // 47

const fruits = ["🍏", "🍌", "🍏", "🍇", "🍌", "🍌"];
const compteur = fruits.reduce((acc, fruit) => {
  acc[fruit] = (acc[fruit] || 0) + 1;
  return acc;
}, {});
console.log(compteur);
// { '🍏': 2, '🍌': 3, '🍇': 1 }


const utilisateurs = [
  { id: 1, nom: "Alice" },
  { id: 2, nom: "Bob" },
  { id: 3, nom: "Charlie" }
];

const dictionnaire = utilisateurs.reduce((acc, user) => {
  acc[user.id] = user.nom;
  return acc;
}, {});
console.log(dictionnaire);
// { 1: 'Alice', 2: 'Bob', 3: 'Charlie' }

// Trouver la date min et max d'un tableau d'objets (actionA1)
const datesMinMax = {
    dateMin: jsonData.ventes.reduce((min, vente) => min.date > vente.date ? vente
: min, jsonData.ventes[0]).date,
    dateMax: jsonData.ventes.reduce((max, vente) => max.date < vente.date ? vente
: max, jsonData.ventes[0]).date
};
console.log(datesMinMax);
// { dateMin: '2025-01-01', dateMax: '2025-11-01' } (exemple)

// Calculer le chiffre d'affaires moyen par jour (actionA2)
const totalVentes = jsonData.ventes.reduce(
    (total, vente) => total + vente.produits.reduce((s, p) => s + p.prix, 0), 0
```

```
);
const nbJours = jsonData.ventes.reduce(
    (jours, vente) => jours.includes(vente.date) ? jours : [...jours, vente.date],
[]
).length;
const caMoyen = totalVentes / nbJours;
console.log(caMoyen);

// Compter les ventes par type de produit (actionA6)
const CAParType = jsonData.ventes
    .flatMap(v => v.produits)
    .reduce((acc, produit) => {
        acc[produit.type] = (acc[produit.type] || 0) + produit.prix;
        return acc;
    }, {});
console.log(CAParType);
// { "boisson": 150, "snack": 80, ... }

// Lister les produits uniques par type (actionA7)
const produitsParType = jsonData.ventes
    .flatMap(v => v.produits)
    .reduce((acc, p) => {
        if (!acc[p.type]) acc[p.type] = [];
        if (!acc[p.type].includes(p.nom)) acc[p.type].push(p.nom);
        return acc;
    }, {});
console.log(produitsParType);
// { "boisson": ["Coca", "Pepsi"], "snack": ["Chips", "Cookies"] }

// Top 5 des ventes par chiffre d'affaires (actionA10)
const TOP_VALUE = 5;
const topVentes = jsonData.ventes
    .flatMap(v => v.produits)
    .reduce((acc, produit) => {
        const exist = acc.find(p => p.id === produit.id);
        if (exist) exist.ca += produit.prix;
        else acc.push({ id: produit.id, nom: produit.nom, ca: produit.prix });
        return acc;
    }, [])
    .sort((a, b) => b.ca - a.ca)
    .slice(0, TOP_VALUE);
console.log(topVentes);
```

`reverse()` - inverser l'ordre du tableau

Modifie le tableau en place en inversant l'ordre de ses éléments.

```
let fruits = ["pomme", "banane", "cerise"];
fruits.reverse();
// ["cerise", "banane", "pomme"]

// Utile pour afficher du plus récent au plus ancien
let messages = ["msg1", "msg2", "msg3"];
messages.reverse(); // ["msg3", "msg2", "msg1"]
```

Techniques

``(backticks) - pour des expressions intelligentes

Les backticks (`) permettent de créer des chaînes de caractères faciles à lire, où l'on peut :

- Mettre directement des variables ou des calculs dedans
- Écrire du texte sur plusieurs lignes

```
const nom = "Diogo";
const age = 17;

// Insérer des variables directement dans le texte
const texte = `Bonjour, je m'appelle ${nom} et j'ai ${age} ans.`;
console.log(texte);
// Affiche : Bonjour, je m'appelle Diogo et j'ai 17 ans.

// Écrire sur plusieurs lignes facilement
const multiLignes = `Ligne 1
Ligne 2
Ligne 3`;
console.log(multiLignes);
// Affiche :
// Ligne 1
// Ligne 2
// Ligne 3
```

`new Set()` - pour supprimer les doublons

Crée une collection unique d'éléments. Les doublons sont automatiquement éliminés.

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

// Créer un Set pour supprimer les doublons
let unique = new Set(nombres);
console.log(unique); // Set(5) {1, 2, 3, 4, 5}

// Revenir à un tableau
let tableauUnique = [...unique];
console.log(tableauUnique); // [1, 2, 3, 4, 5]

// Utile pour filtrer rapidement des valeurs répétées
let mots = ["chat", "chien", "chat", "oiseau", "chien"];
let motsUniques = [...new Set(mots)];
console.log(motsUniques); // ["chat", "chien", "oiseau"]
```

Fonctions

Déclaration de fonction

Standard

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

Sous forme d'expression de fonction

```
const doStuff = function (a, b, c) {
    return a + b + c;
};
```

Sous forme d'expression de fonction anonyme

```
const doStuff = (a, b, c) => {
    return a + b + c;
```

```
};
```

Sous forme raccourcie

S'il n'y a qu'un seul argument et que son corps n'a qu'une seule expression, on peut omettre le return et le corps de la fonction :

```
const doStuff = (a) => `Salut ${a} !`;
```

Fonctions immédiatement invoquées (IIFE)

IIFE = Immediately Invoked Function Expressions.

Ces fonctions sont définies et **exécutées immédiatement**. Elles sont souvent utilisées pour créer un **contexte isolé** ou encapsuler du code sans polluer l'espace global.

```
(function(){ ... })()
```

ou

```
(() => { ... })()
```

Conclusion

Durant ce module, j'ai apprécié découvrir la programmation fonctionnelle en JavaScript. Même si certaines fonctions comme reduce m'ont demandé un peu de réflexion au début, j'ai appris beaucoup de méthodes pratiques pour manipuler et transformer des données efficacement. Ces nouvelles compétences me seront surement très utiles pour mes futurs projets en développement.