🤔 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

Ce module m'a permis de découvrir la programmation fonctionnelle en JavaScript à travers des cas concrets, notamment l'analyse d'un gros jeu de données de notes scolaires (`jsonData`).
Les objectifs principaux pour moi :

- Comprendre la différence entre approche impérative et fonctionnelle.
- Manipuler les tableaux avec `map`, `filter`, `reduce`, etc.
- Limiter les effets de bord en privilégiant des fonctions pures.
- Travailler avec des données réelles (notes, élèves, branches) et en extraire des infos utiles.
- Gagner en lisibilité, maintenabilité et réutilisabilité dans mon code.

L'idée générale : prendre des données comme celles de `jsonData.evaluations` et, grâce aux outils Javascript modernes, construire rapidement des statistiques, tris, regroupements et filtrages sans tout recoder à la main avec des boucles partout.

Opérateurs javascript super-cooool 😎

opérateur `?:`

Cet opérateur ternaire évalue une condition et renvoie l'une des deux expressions selon que cette condition est vraie ou fausse. Il se compose d'une partie test avant le `?`, d'une valeur retournée si la condition est vérifiée et d'une autre si elle ne l'est pas, séparées par `:`. En remplaçant un bloc `if…else`, il permet de produire une valeur directement dans une expression ou une affectation sans alourdir le code. Comme tout opérateur, le résultat peut être utilisé dans une nouvelle expression ou passé en paramètre à une fonction.

```
const note = 5.2;
const resultat = note >= 4 ? 'réussi' : 'échec'; // 'réussi'
```

opérateur `??`

L'opérateur de coalescence nullish renvoie l'opérande de gauche si celle-ci n'est ni `null` ni `undefined`, sinon il renvoie l'opérande de droite. Cette construction sert à définir une valeur par défaut uniquement lorsque une variable n'est pas initialisée, sans remplacer les valeurs considérées falsy comme `0`, une chaîne vide ou `false`. Elle est particulièrement utile pour distinguer les cas où un paramètre est absent de ceux où il contient une valeur valide mais fausse selon l'évaluation booléenne.

```
const branche = null ?? 'Inconnue'; // 'Inconnue'
const note = 0 ?? 4;                // 0 (0 est accepté)
```

⚠️ Contrairement à `||`, les valeurs `0`, `''` ou `false` ne sont pas remplacées.

```
const a = 0 || 4;  // 4  (dangereux si 0 est valide)
const b = 0 ?? 4;  // 0  (correct ici)
```

opérateur `??=`

Cet opérateur d'affectation conditionnelle n'écrase pas les valeurs existantes : il vérifie si la propriété ou la variable ciblée est `null` ou `undefined` et, dans ce cas seulement, lui assigne l'expression de droite. Son comportement équivaut à écrire `x ?? (x = y)` tout en évaluant l'identifiant une seule fois, ce qui évite les effets de bord lorsqu'une évaluation coûte cher. On l'utilise pour établir des valeurs par défaut ou initialiser des options sans risquer de supprimer des données déjà présentes.

```
const config = { seuilReussite: null };

config.seuilReussite ??= 4; // devient 4
config.mode ??= 'standard'; // devient 'standard'
```

opérateur de décomposition 'spread' `...`

La syntaxe de décomposition `...` étale le contenu d'un tableau, d'un objet ou d'un itérable à l'endroit où plusieurs éléments ou propriétés sont attendus. On s'en sert pour fusionner plusieurs tableaux en un seul, créer des copies superficielles d'objets ou ajouter des propriétés à un objet en combinant des sources. Contrairement à l'opérateur de reste utilisé dans les paramètres de fonction, le spread développe les valeurs plutôt que de les collecter et permet d'écrire des manipulations de collections de manière concise.

```
const base = ['Français', 'Maths'];
const options = ['Physique', 'Chimie'];

const toutesBranches = [...base, ...options];

const eleve = { nom: "TERNET", prenom: "Alain" };
const details = { classe: "EMF", annee: "2024-2025" };

const eleveComplet = { ...eleve, ...details };
```

Déstructuration

La déstructuration est une syntaxe qui permet d'extraire des valeurs d'un tableau ou des propriétés d'un objet directement dans des variables distinctes. Elle reprend la forme littérale de la structure d'origine et peut inclure un opérateur de reste (`...`) pour collecter les éléments non extraits. En évitant d'accéder à chaque propriété individuellement, cette technique simplifie l'écriture du code lorsque l'on manipule des structures de données complexes.

```
const [premiereEval, ...autres] = jsonData.evaluations;
const { etablissement, annee_scolaire } = jsonData;
```

Date et Heure

Lien : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Date

Obtenir la date et/ou heure actuelle

L'objet `Date` représente un moment précis mesuré en millisecondes écoulées depuis le 1er janvier 1970 à minuit UTC. Créer une instance avec `new Date()` permet d'obtenir la date et l'heure actuelles selon le fuseau horaire local. Les méthodes d'instance telles que `getDate()`, `getMonth()` (où janvier vaut 0) et `getFullYear()` retournent respectivement le jour, le mois et l'année. D'autres méthodes comme `toLocaleDateString()` et `toLocaleTimeString()` formatent la date ou l'heure en fonction de la locale, tandis que `toISOString()` fournit une représentation ISO 8601.

```
const maintenant = new Date();

console.log(maintenant.toLocaleDateString());
console.log(maintenant.toLocaleTimeString());

const jour = maintenant.getDate();
const mois = maintenant.getMonth() + 1; // janvier = 0
const annee = maintenant.getFullYear();

console.log(`${jour}.${mois}.${annee}`);
console.log(maintenant.toISOString());
```

Math

Lien : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Math

`Math.PI` - la constante π

La propriété statique `Math.PI` fournit une approximation de la constante π, largement utilisée en trigonométrie et en géométrie. Elle permet d'utiliser cette valeur sans devoir la définir soi-même et fait partie des nombreuses constantes disponibles sur l'objet `Math`.

```
console.log(Math.PI); // 3.141592653589793
```

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

La méthode statique `Math.abs()` renvoie la valeur absolue d'un nombre, c'est-à-dire sa distance à zéro sans tenir compte du signe. Elle est couramment utilisée pour calculer des écarts ou comparer des grandeurs indépendamment de leur orientation.

```
const ecart = Math.abs(4 - 5.7); // 1.7
```

`Math.pow()` - élever à une puissance

La méthode `Math.pow(base, exposant)` retourne la valeur de `base` élevée à la puissance `exposant`. Cette opération est équivalente à l'opérateur d'exponentiation (`**`) mais fonctionne exclusivement avec des valeurs numériques et permet d'écrire des calculs de puissances de manière explicite.

```
console.log(Math.pow(2, 3)); // 8
console.log(2 ** 3);         // 8
```

`Math.min()` - plus petite valeur

La méthode `Math.min()` renvoie le plus petit nombre parmi un ensemble de valeurs passées en arguments. Si aucun argument n'est fourni, elle renvoie `Infinity`. Cette fonction est pratique pour déterminer rapidement une valeur minimale sans devoir trier les données.

```
console.log(Math.min(3, 7, -2, 0)); // -2
```

`Math.max()` - plus grande valeur

La méthode `Math.max()` renvoie le plus grand nombre parmi les valeurs passées en arguments. En l'absence d'arguments, elle renvoie `-Infinity`. Utilisée avec la syntaxe spread, elle permet de déterminer aisément la valeur maximale d'un tableau.

```
console.log(Math.max(3, 7, -2, 0)); // 7
```

`Math.ceil()` - arrondir à l'entier supérieur

La fonction `Math.ceil()` renvoie le plus petit entier supérieur ou égal à un nombre donné. Elle arrondit toujours vers le haut, quel que soit le signe de l'argument, et peut servir à calculer le nombre de lots nécessaires pour contenir des éléments.

```
console.log(Math.ceil(4.2)); // 5
```

`Math.floor()` - arrondir à l'entier inférieur

La méthode `Math.floor()` renvoie le plus grand entier inférieur ou égal à un nombre. Contrairement à `ceil()`, elle arrondit toujours vers le bas et est utile pour tronquer des indices ou positionner des éléments sur une grille.

```
console.log(Math.floor(4.9)); // 4
```

`Math.round()` - arrondir à l'entier le plus proche

La méthode `Math.round()` arrondit une valeur au nombre entier le plus proche. Lorsque la partie fractionnaire vaut 0,5 ou plus, l'entier supérieur est renvoyé. Elle sert à produire des résultats entiers à partir de calculs décimaux.

```
console.log(Math.round(4.4)); // 4
console.log(Math.round(4.6)); // 5
```

`Math.trunc()` - partie entière

La méthode `Math.trunc()` retourne la partie entière d'un nombre en supprimant simplement les décimales. Elle se distingue de `Math.floor()` et `Math.ceil()` car elle se contente de retirer la fraction sans arrondir selon le signe et peut s'appliquer à des nombres positifs ou négatifs.

```
console.log(Math.trunc(4.9));  // 4
console.log(Math.trunc(-4.9)); // -4
```

`Math.sqrt()` - racine carrée

La méthode `Math.sqrt()` calcule la racine carrée d'un nombre non négatif et renvoie `NaN` pour un argument négatif. Ce calcul est indispensable en géométrie pour déterminer des distances ou des diagonales et sert en statistique pour le calcul de l'écart type.

```
console.log(Math.sqrt(9)); // 3
```

`Math.random()` - nombre aléatoire `[0, 1)`

La méthode `Math.random()` renvoie un nombre pseudo-aléatoire compris entre 0 (inclus) et 1 (exclus). Ces valeurs ne sont pas cryptographiquement sécurisées mais conviennent pour des simulations courantes ou des tirages aléatoires. Pour obtenir un entier dans une plage spécifique, on multiplie le résultat par l'amplitude souhaitée puis on l'arrondit.

```
const tirage = Math.random(); // ex: 0.37
const noteRandom = Math.round(1 + Math.random() * 5); // 1 à 6
```

JSON

Lien : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/JSON

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

La méthode statique `JSON.stringify()` convertit une valeur JavaScript (objet, tableau, chaîne, nombre) en chaîne de caractères au format JSON. Elle est utilisée pour sérialiser des données avant de les stocker ou de les envoyer sur un réseau et accepte un second paramètre optionnel permettant de filtrer ou de transformer les valeurs pendant la conversion.

```
const extrait = {
  etablissement: jsonData.etablissement,
  nbEvaluations: jsonData.evaluations.length
};

console.log(JSON.stringify(extrait));
```

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

La méthode `JSON.parse()` analyse une chaîne JSON valide et reconstruit l'objet ou la valeur JavaScript qu'elle décrit. On peut fournir un second argument, appelé *reviver*, qui est une fonction utilisée pour transformer chaque paire clé/valeur lors de la création du nouvel objet. Une erreur est levée si la chaîne n'est pas au format JSON.

```
const texteJson = '{"nom":"TARISTE","note":5.9}';
const evalObj = JSON.parse(texteJson);

console.log(evalObj.nom);
console.log(evalObj.note);
```

Chaînes de caractères

Lien : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/String

`split()` - découper en tableau

La méthode `split()` divise une chaîne en plusieurs segments selon un séparateur fourni et renvoie un tableau des sous-chaînes. Le séparateur peut être une chaîne ou une expression régulière, et un second argument optionnel limite le nombre de segments retournés. La chaîne d'origine n'est pas modifiée, ce qui permet de conserver sa valeur initiale.

```
const date = "19.08.2024";
const [jour, mois, annee] = date.split('.');
```

`trim()`, `trimStart()` et `trimEnd()`

Les méthodes `trim()`, `trimStart()` et `trimEnd()` retournent une nouvelle chaîne sans les espaces ou caractères blancs placés au début et/ou à la fin de la chaîne de départ. Elles sont pratiques pour nettoyer des entrées utilisateur ou des textes importés, et la valeur d'origine reste inchangée.

```
const nomSale = "  VOYANTE  ";
nomSale.trim();      // "VOYANTE"
nomSale.trimStart(); // "VOYANTE  "
nomSale.trimEnd();   // "  VOYANTE"
```

`padStart()` et `padEnd()`

Les méthodes `padStart()` et `padEnd()` complètent une chaîne en y ajoutant des caractères au début ou à la fin jusqu'à atteindre une longueur cible. On peut définir la chaîne de remplissage ; si celle-ci est trop longue, elle est tronquée pour respecter la longueur finale. Ces fonctions facilitent l'alignement de textes ou la création d'identifiants au format fixe.

```
const num = '5';
num.padStart(3, '0'); // '005'
num.padEnd(4, '-');   // '5---'
```

Console

Lien : https://developer.mozilla.org/fr/docs/Web/API/console

`console.log()`

La méthode `console.log()` écrit une représentation textuelle de ses arguments dans la console du navigateur ou de l'environnement d'exécution. Elle sert principalement au débogage et accepte plusieurs valeurs qui seront affichées séparément ou concaténées selon l'implémentation.

```
console.log('Début de l’analyse des évaluations');
```

`console.info()`, `warn()`, `error()`

Les méthodes `console.info()`, `console.warn()` et `console.error()` génèrent respectivement des messages d'information, d'avertissement et d'erreur. Elles permettent de distinguer différents niveaux de gravité et sont souvent formatées différemment dans les outils de développement, avec des icônes ou des couleurs distinctes. En filtrant les types de messages, on peut se concentrer sur les avertissements ou les erreurs lors du débogage.

```
console.info('Chargement terminé.');
console.warn('Aucune note pour cette branche.');
console.error('Erreur critique.');
```

`console.table()`

La méthode `console.table()` affiche des tableaux ou des objets sous forme tabulaire dans la console. Les entrées sont présentées en lignes et colonnes, ce qui facilite la lecture et la comparaison des valeurs sans avoir à écrire de code pour le formatage.

```
console.table(jsonData.evaluations.slice(0, 5));
```

`console.time()`, `timeLog()`, `timeEnd()`

Les méthodes `console.time()`, `console.timeLog()` et `console.timeEnd()` permettent de mesurer la durée d'exécution d'un morceau de code. On fournit un identifiant commun pour démarrer le chronomètre, enregistrer des temps intermédiaires et arrêter la mesure. Ces fonctions sont utiles pour évaluer les performances sans modifier la logique principale du programme.

```
console.time('calcul');

const moyenne = jsonData.evaluations
  .filter(e => e.branche === 'Maths')
  .reduce((sum, e) => sum + e.note, 0) / jsonData.evaluations.filter(e => e.branche ===
'Maths').length;

console.timeLog('calcul');
console.timeEnd('calcul');
```

Tableaux

En JavaScript, un tableau (`Array`) est une structure permettant de stocker une liste ordonnée de valeurs, qu'il s'agisse de nombres, de chaînes ou d'objets. L'API des tableaux fournit de nombreuses méthodes pour itérer, rechercher, transformer et manipuler ces collections sans écrire de boucles complexes. Dans les sections qui suivent, on applique ces méthodes sur le tableau `evaluations` pour en extraire des informations pertinentes.

Lien : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Array

```
const evaluations = jsonData.evaluations;
```

forEach

La méthode `forEach()` exécute une fonction fournie pour chaque élément du tableau. Elle est conçue pour parcourir la collection et appliquer des effets de bord, mais elle ne renvoie pas de nouveau tableau et ne permet pas d'interrompre l'itération.

```
evaluations.slice(0, 3).forEach(e => {
  console.log(`${e.nom} ${e.prenom} : ${e.note}`);
});
```

entries()

La méthode `entries()` retourne un itérateur fournissant des paires `[index, valeur]` pour chaque élément du tableau. Utilisée avec `for...of`, elle permet de parcourir les indices et les éléments simultanément.

```
for (const [index, evalObj] of evaluations.slice(0, 3).entries()) {
  console.log(index, evalObj.nom, evalObj.note);
}
```

in

La boucle `for...in` parcourt les clés énumérables d'un objet ; appliquée à un tableau, elle renvoie des chaînes représentant les indices. Elle est moins utilisée que `for...of` lorsqu'on s'intéresse aux valeurs, mais reste pratique pour inspecter les indices.

```
for (const i in evaluations.slice(0, 3)) {
  console.log(i); // 0, 1, 2
}
```

of

La syntaxe `for...of` permet d'itérer directement sur les valeurs d'un tableau ou de tout autre objet itérable. Elle simplifie la lecture du code en donnant accès à chaque élément sans passer par ses indices.

```
for (const evalObj of evaluations.slice(0, 3)) {
  console.log(evalObj.nom, evalObj.note);
}
```

find()

La méthode `find()` recherche le premier élément d'un tableau qui satisfait la fonction de test fournie. Elle renvoie cet élément ou `undefined` si aucun ne correspond, ce qui permet de réaliser des recherches ciblées.

```
const premiereSous4 = evaluations.find(e => e.note < 4);
```

findIndex()

La méthode `findIndex()` renvoie l'indice du premier élément du tableau qui satisfait la fonction de test donnée. Si aucun élément ne passe le test, elle retourne -1. Cette valeur d'indice peut ensuite être utilisée pour accéder ou modifier l'élément.

```
const indexVoyante = evaluations.findIndex(e => e.nom === 'VOYANTE');
```

indexOf() / lastIndexOf()

Les méthodes `indexOf()` et `lastIndexOf()` recherchent une valeur exacte dans un tableau de primitives et renvoient respectivement la première ou la dernière position trouvée. Elles utilisent l'égalité stricte (`===`) et retournent -1 si la valeur est absente, ce qui permet de savoir si un élément est présent et à quelle position.

```
const branches = ['Maths', 'Physique', 'Maths', 'Français'];
branches.indexOf('Maths');     // 0
branches.lastIndexOf('Maths'); // 2
```

push(), pop(), shift(), unshift()

Ces méthodes modifient un tableau en ajoutant ou en retirant des éléments à ses extrémités. `push()` ajoute un élément à la fin et retourne la nouvelle longueur ; `pop()` supprime et renvoie le dernier élément. À l'inverse, `unshift()` ajoute un élément au début et renvoie la nouvelle longueur, tandis que `shift()` retire et renvoie le premier élément. Comme elles modifient directement le tableau, il faut être vigilant si d'autres variables y font référence.

```
const notes = [4, 5];
notes.push(5.5);
notes.pop();
notes.unshift(3);
notes.shift();
```

slice()

La méthode `slice()` crée une copie superficielle d'une portion de tableau entre des indices donnés (début inclus, fin exclue). Elle renvoie un nouveau tableau sans modifier l'original, ce qui la rend idéale pour extraire des sous-ensembles ou préparer des données sans effet de bord.

```
const top5 = evaluations.slice(0, 5);
```

splice()

La méthode `splice()` modifie un tableau en place en supprimant, remplaçant ou insérant des éléments à une position donnée. Elle prend un indice de départ, le nombre d'éléments à retirer et éventuellement les éléments à insérer, et renvoie le tableau des éléments supprimés.

```
const copie = [...evaluations];
copie.splice(0, 1); // supprime le premier
```

concat()

La méthode `concat()` renvoie un nouveau tableau résultant de la fusion de plusieurs tableaux ou valeurs. Les tableaux originaux ne sont pas modifiés et les éléments sont copiés dans la nouvelle instance.

```
const a = evaluations.slice(0, 2);
const b = evaluations.slice(2, 4);
const fusion = a.concat(b);
```

join()

La méthode `join()` crée une chaîne en concaténant tous les éléments d'un tableau, séparés par une chaîne de délimitation optionnelle. Elle est souvent utilisée pour présenter des listes sous forme textuelle, comme la création d'une liste d'éléments séparés par des virgules.

```
const noms = ['VOYANTE', 'TERNET', 'TARISTE'];
noms.join(', '); // "VOYANTE, TERNET, TARISTE"
```

keys() et values() (objets)

Les méthodes statiques `Object.keys()` et `Object.values()` renvoient respectivement un tableau des noms de propriétés et un tableau des valeurs d'un objet. Elles ne retournent que les propriétés propres et énumérables de l'objet, ce qui est utile pour parcourir ou manipuler les contenus d'un enregistrement.

```
const exemple = evaluations[0];
Object.keys(exemple);   // ['date','nom','prenom','branche','note']
Object.values(exemple); // [...]
```

includes()

La méthode `includes()` vérifie si une valeur existe dans un tableau et renvoie un booléen. Elle utilise l'égalité stricte (`===`) et constitue une manière simple et lisible de tester la présence d'un élément.

```
const toutesBranches = evaluations.map(e => e.branche);
toutesBranches.includes('Maths');
```

every() et some()

La méthode `every()` teste si tous les éléments d'un tableau satisfont une condition et renvoie `true` seulement si c'est le cas. À l'inverse, `some()` renvoie `true` dès qu'au moins un élément satisfait la condition. Ces deux méthodes permettent d'évaluer des ensembles sans écrire de boucles explicites.

```
evaluations.every(e => e.note >= 4); // tous réussis ?
evaluations.some(e => e.note < 3);   // au moins un gros échec ?
```

fill()

La méthode `fill()` remplace une plage d'éléments dans un tableau par une valeur statique et renvoie le tableau modifié. On peut spécifier des indices de début et de fin pour contrôler la portion à remplir ; en l'absence de limites, tout le tableau est écrasé.

```
new Array(3).fill('en attente');
```

flat()

La méthode `flat()` renvoie un nouveau tableau en concaténant les éléments d'un tableau imbriqué selon une profondeur donnée. Par défaut, elle réduit d'un niveau et n'altère pas le tableau d'origine, ce qui est utile pour simplifier des structures hiérarchiques.

```
[ [1, 2], [3, 4] ].flat(); // [1, 2, 3, 4]
```

sort()

La méthode `sort()` trie les éléments d'un tableau en place et renvoie le tableau modifié. Sans fonction de comparaison, les éléments sont convertis en chaînes et triés par ordre lexicographique. En fournissant une fonction de comparaison, on peut effectuer un tri numérique ou personnalisé.

```
const notesTriees = [...evaluations]
  .map(e => e.note)
  .sort((a, b) => a - b);
```

map()

La méthode `map()` crée un nouveau tableau contenant les résultats de l'appel d'une fonction pour chaque élément du tableau d'origine. Elle est utilisée pour transformer des données sans modifier le tableau initial, et la fonction reçoit la valeur, l'indice et le tableau complet en paramètres.

```
const nomsComplets = evaluations.map(e => `${e.nom} ${e.prenom}`);
```

filter()

La méthode `filter()` retourne un nouveau tableau composé des éléments qui satisfont la fonction de test fournie. Les éléments rejetés sont ignorés et le tableau d'origine reste inchangé, ce qui en fait un outil couramment utilisé pour extraire des sous-ensembles.

```
const maths = evaluations.filter(e => e.branche === 'Maths');
const echec = evaluations.filter(e => e.note < 4);
```

groupBy()

La méthode statique `Object.groupBy()` regroupe les éléments d'un tableau selon la valeur renvoyée par une fonction de classement. Elle retourne un objet dont chaque propriété correspond à un groupe et contient un tableau des éléments associés. Cette fonctionnalité n'est disponible que dans les environnements JavaScript récents.

```
const parBranche = Object.groupBy(evaluations, e => e.branche);
// parBranche['Maths'] → toutes les évaluations de Maths
```

flatMap()

La méthode `flatMap()` applique une fonction à chaque élément d'un tableau puis aplati d'un niveau le résultat, retournant un nouveau tableau. Elle combine l'effet de `map()` suivi d'un `flat(1)` et s'avère plus efficace que l'enchaînement séparé de ces méthodes.

```
const elevesUniques = [...new Set(
  evaluations.flatMap(e => `${e.nom} ${e.prenom}`)
)];
```

reduce() / reduceRight()

Les méthodes `reduce()` et `reduceRight()` appliquent une fonction de réduction à un tableau pour le condenser en une seule valeur. `reduce()` parcourt les éléments de gauche à droite tandis que `reduceRight()` itère en sens inverse. On fournit un accumulateur initial et une fonction qui reçoit l'accumulateur et la valeur courante à chaque itération.

```
const sommeNotes = evaluations.reduce((sum, e) => sum + e.note, 0);
const moyenneGlobale = sommeNotes / evaluations.length;
```

reverse()

La méthode `reverse()` inverse l'ordre des éléments d'un tableau en place et renvoie le tableau modifié. Cette modification affecte toutes les références au tableau, il faut donc s'assurer que l'ordre inversé est souhaité avant d'utiliser cette méthode.

```
const ordreInverse = [...evaluations].reverse();
```

Techniques

`` (backticks) - templates

Les littéraux de gabarits, encadrés par des accents graves (backticks), permettent de créer des chaînes contenant des expressions JavaScript évaluées via la syntaxe `${expression}`. Ils facilitent la construction de chaînes multi-lignes et l'interpolation de variables, et peuvent être transformés par des fonctions spéciales appelées « tagged templates ».

```
const e = evaluations[0];
`Le ${e.date}, ${e.prenom} ${e.nom} a obtenu ${e.note} en ${e.branche}.`;
```

`new Set()` - supprimer les doublons

L'objet `Set` représente une collection de valeurs uniques et conserve l'ordre d'insertion. Ajouter une valeur déjà présente n'a aucun effet, ce qui en fait un outil efficace pour éliminer les doublons d'un tableau ou tester l'unicité d'une entrée. Les ensembles peuvent être parcourus avec `for...of` ou convertis en tableaux via l'opérateur de décomposition.

```
const eleves = [...new Set(evaluations.map(e => `${e.nom} ${e.prenom}`))].sort();
```

Fonctions

Déclaration de fonction

En JavaScript, la déclaration d'une fonction avec le mot-clé `function` crée une nouvelle valeur de type fonction associée à un identifiant. Les fonctions sont des objets de première classe : elles peuvent être stockées dans des variables, passées en argument et retournées par d'autres fonctions. Elles renvoient la valeur indiquée par la dernière instruction `return` ; en l'absence de `return`, la fonction renvoie `undefined`.

```
function moyenneBranche(branche) {
  const notes = evaluations.filter(e => e.branche === branche);
  return notes.reduce((sum, e) => sum + e.note, 0) / notes.length;
}

const moyenneEleve = function (nom, prenom) {
  const notes = evaluations.filter(e => e.nom === nom && e.prenom === prenom);
  return notes.reduce((sum, e) => sum + e.note, 0) / notes.length;
};

const estReussi = (note) => note >= 4;

const labelNote = (note) => note >= 4 ? 'OK' : 'NOK';
```

Fonctions immédiatement invoquées (IIFE)

Une fonction immédiatement invoquée (IIFE) est définie et exécutée dans la même expression grâce à des parenthèses entourant la définition et l'appel. Ce patron crée un contexte local isolé et permet d'initialiser des variables privées ou d'exécuter du code d'initialisation sans polluer l'espace global. Les IIFE sont souvent utilisées pour encapsuler du code et éviter les collisions de noms.

```
(() => {
  const nbEvaluations = evaluations.length;
  console.log(`Nombre total d'évaluations : ${nbEvaluations}`);
})();
```

Conclusion

Ce module m'a obligé à structurer ma manière de coder.
Travailler sur un gros jeu de données comme `jsonData` m'a montré l'intérêt concret de la programmation fonctionnelle :

- parcourir, filtrer et transformer des tableaux sans multiplier les boucles imbriquées ;
- écrire des fonctions courtes, réutilisables et prévisibles ;
- utiliser des méthodes comme `map`, `filter`, `reduce`, `groupBy`, combinées aux opérateurs modernes (`...`, `??`, backticks, `new Set`) pour obtenir rapidement les infos utiles.

Ces outils rendent le code plus clair, plus compact et plus simple à maintenir, surtout lorsqu'il s'agit de manipuler des données réelles comme des résultats scolaires.