🤔 RP Noé Romanens - 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
- Introduction rapide à la PF
- Opérateurs javascript super-coooool 😎
  - opérateur `?:`
  - opérateur `??`
  - opérateur `??=`
  - opérateur de décomposition 'spread' `...`
  - Déstructuration
- Date et Heure
  - Obtenir la date et/ou heure actuelle
- Math
  - `Math.PI` - la constante π
  - `Math.abs()` - la |valeur absolue| d'un nombre
  - `Math.pow()` - élever à une puissance
  - `Math.min()` - plus petite valeur
  - `Math.max()` - plus grande valeur
  - `Math.ceil()` - arrondir à la prochaine valeur entière la plus proche
  - `Math.floor()` - arrondir à la précédente valeur entière la plus proche
  - `Math.round()` - arrondir à la valeur entière la plus proche
  - `Math.trunc()` - supprime la virgule et retourne la partie entière d'un nombre
  - `Math.sqrt()` - la raçine carrée d'un nombre
  - `Math.random()` - générer un nombre aléatoire entre 0.0 (compris) et 1.0 (non compris)
- JSON
  - `JSON.stringify()` - transformer un objet Javascript en JSON
  - `JSON.parse()` - transformer du JSON en objet Javascript
- Chaînes de caractères
  - `split()` - un ciseau qui coupe une chaîne là où un caractère apparaît et produit un tableau
  - `trim()`, `trimStart()` et `trimEnd()` - épuration des espaces en trop dans une chaîne (trimming)
  - `padStart()` et `padEnd()` - aligner le contenu dans une chaîne de caractères
- Console
  - `console.log()` - Afficher un message sur la console
  - `console.info()`, `warn()` et `error()` - Afficher un message sur la console (filtrables)
  - `console.table()` - Afficher tout un tableau ou un objet sur la console
  - `console.time()`, `timeLog()` et `timeEnd()` - Chronométrer une durée d'exécution
- Tableaux
  - `forEach` - parcourir les éléments d'un tableau
  - `entries()` - parcourir les couples index/valeurs d'un tableau
  - `in` - parcourir les clés d'un tableau

    - `of` - parcourir les valeurs d'un tableau
    - `find()` - premier élément qui satisfait une condition
    - `findIndex()` - premier index qui satisfait une condition
    - `indexOf()` et `lastIndexOf()` - premier/dernier élément qui correspond
    - `push()`, `pop()`, `shift()` et `unshift()` - ajouter/supprime au début/fin dans un tableau
    - `slice()` - ne conserver que certaines lignes d'un tableau
    - `splice()` - supprimer/insérer/remplacer des valeurs dans un tableau
    - `concat()` - joindre deux tableaux
    - `join()` - joindre des chaînes de caractères
    - `keys()` et `values()` - les clés/valeurs d'un objet
    - `includes()` - vérifier si une valeur est présente dans un tableau
    - `every()` et `some()` - vérifier si plusieurs valeurs sont toutes/quelques présentes dans un tableau
    - `fill()` - remplir un tableau avec des valeurs
    - `flat()` - aplatir un tableau
    - `sort()` - pour trier un tableau
    - `map()` - tableau avec les résultats d'une fonction
    - `filter()` - tableau avec les éléments passant un test
    - `groupBy()` - regroupe les éléments d'un tableau selon un règle
    - `flatMap()` - chaînage de map() et flat()
    - `reduce()` et `reduceRight()` - réduire un tableau à une seule valeur
    - `reverse()` - inverser l'ordre du tableau
- Techniques
    - ``(backticks) - pour des expressions intelligentes
    - `new Set()` - pour supprimer les doublons
- Design pattern — Builder (PCBuilder)
    - Pourquoi l'utiliser
    - Exemple: PC et PCBuilder
    - Variantes utiles
    - Quand l'utiliser (et quand éviter)
- Fonctions
    - Déclaration de fonction
    - Fonctions fléchées (arrow functions)
    - Fonctions immédiatement invoquées (IIFE)
- Immuabilité
- Fonctions pures
- Composition (currying, closure, pipe, compose)
- Récursion
- Annexes (mémo + liens + export PDF)
- Programmation fonctionnelle — Cas pratiques
    - Données utilisées
    - Boîte à outils réutilisable
    - A01 - La plus petite et la plus grande date
    - A02 - Le chiffre d'affaire quotidien moyen
    - A03 - Liste des clients par id
    - A04 - Liste des clients et leur CA
    - A05 - CA et nbre de ventes par jour de la semaine
    - A06 - CA par type de produit
    - A07 - Types de produits et leurs produits
    - A08 - Nbre de vente des produits par type

    - A09 - Toutes les ventes réalisées
    - A10 - TOP 6 des produits générant le plus de CA
    - A11 - Liste des clients triée nom/prénom
    - A12 - TOP 5 des meilleures dates et leur CA
- Exercice Drill 11 — Notes scolaires
    - Utilitaires (frCompare, normalizeString, round2, parseDate, groupBy)
    - A1 - [Branches], trié
    - A2 - [Prénom NOM], trié
    - B - {Branches : nbre de notes}
    - C - [{Branche, nbre notes}], trié
    - T1 - Apprentis contenant Partie connue
    - T2 - Total évaluations de cette branche
    - D - [{Branche, moyenne}], trié
    - E - [{Branche, nbre, moyenne}], trié
    - F - [{Elève, [branches, [notes]]}], trié
    - G - [{Elève, [branches, nbre, min, max, moyenne, [notes]]}], trié
    - H - [{Elève, [branches, nbre, min, max, moyenne, [{date, note}]]}], trié
- Conclusion

Introduction

Ce module 323 est là pour nous apprendre à programmer de manière fonctionnelle en JavaScript, en privilégiant un style déclaratif, des données immuables et des fonctions pures. On va apprendre à transformer, filtrer et agréger des collections sans boucles impératives, puis à composer ces briques pour résoudre des besoins réels.

- Maîtriser map, filter, reduce, sort, flat/flatMap
- Appliquer l'immuabilité et écrire des fonctions pures testables
- Composer des fonctions (currying, closures, pipe/compose)
- Produire des rapports à partir de jeux de données réalistes (A01–A12, Drill 11, autres exos)
- Écrire un code lisible et réutilisable, orienté résultats

Introduction rapide à la PF

La PF (programmation fonctionnelle) en deux mots:

- **Déclaratif**: on dit «quoi faire», pas «comment faire pas à pas».
- **Données immuables**: on ne modifie pas, on crée de nouvelles valeurs.
- **Fonctions pures**: même entrée → même sortie, **sans effets de bord**.
- **Pas de `for` / `while` classiques**: on préfère `map` / `filter` / `reduce`.
- **Composition**: on enchaîne des petites fonctions simples.

Mini-exemples express:

```
const users = [
  { id: 1, name: 'Noé', age: 19 },
  { id: 2, name: 'Ana', age: 22 },
  { id: 3, name: 'Léo', age: 17 },
];

// map: transformer
const namesUpper = users.map(u => u.name.toUpperCase());
// ['NOÉ', 'ANA', 'LÉO']
```

```
// filter: garder une partie
const adults = users.filter(u => u.age >= 18);
// [{ id:1, ... }, { id:2, ... }]

// reduce: «plier» en une valeur
const totalAge = users.reduce((acc, u) => acc + u.age, 0);
// 58
```

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

Constante représentant π (~3.14159). Utile pour périmètre et aire d'un cercle.

```
const r = 3;
const perimetre = 2 * Math.PI * r;   // 18.849...
const aire      = Math.PI * r ** 2;  // 28.274...
```

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

Retourne la distance à zéro (toujours positive).

```
Math.abs(-3);         // 3
Math.abs(5 - 12);     // 7
[-2, -1, 0, 3].map(Math.abs); // [2,1,0,3]
```

`Math.pow()` - élever à une puissance

Élever un nombre à une puissance. Équivalent moderne: l'opérateur `**`.

```
Math.pow(2, 3); // 8
2 ** 3;         // 8
Math.pow(9, 0.5); // 3   (racine carrée)
```

`Math.min()` - plus petite valeur

Plus petit des arguments. Avec un tableau, utiliser l'opérateur spread.

```
Math.min(3, 7, 1);                 // 1
const arr = [3, 7, 1];
Math.min( ... arr);                 // 1
// Cas vide
Math.min();                        // Infinity
const minSafe = arr.length ? Math.min( ... arr) : undefined;
```

`Math.max()` - plus grande valeur

Plus grand des arguments. Avec un tableau, utiliser l'opérateur spread.

```
Math.max(3, 7, 1);                 // 7
const arr = [3, 7, 1];
Math.max( ... arr);                 // 7
// Cas vide
Math.max();                        // -Infinity
const maxSafe = arr.length ? Math.max( ... arr) : undefined;
```

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

Arrondi vers le haut (vers +∞).

```
Math.ceil(2.01);  // 3
Math.ceil(2.9);   // 3
Math.ceil(-1.2);  // -1
```

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

Arrondi vers le bas (vers −∞).

```
Math.floor(2.9);   // 2
Math.floor(2.01);  // 2
Math.floor(-1.2);  // -2
```

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

Arrondi au plus proche (les .5 vont vers +∞).

```
Math.round(2.49); // 2
Math.round(2.5);  // 3
Math.round(-1.5); // -1
```

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

Supprime la partie décimale (ne fait pas d'arrondi).

```
Math.trunc(2.9);   // 2
Math.trunc(-1.9);  // -1
```

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

Racine carrée d'un nombre (NaN pour un négatif).

```
Math.sqrt(9);   // 3
Math.sqrt(2);   // 1.4142...
Math.sqrt(-1);  // NaN
```

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

Retourne un flottant x tel que 0 ≤ x < 1. Utile pour générer un entier dans un intervalle.

```
const x = Math.random(); // 0 <= x < 1
const int0to9 = Math.floor(Math.random() * 10);         // 0..9
const dice1to6 = Math.floor(Math.random() * 6) + 1;     // 1..6
const randInt = (min, max) => Math.floor(Math.random() * (max - min + 1)) + min;
```

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

Sérialise un objet JS en chaîne JSON. Paramètre 2/3 pour contrôler replacer et indentation.

```
const obj = { id: 1, nom: 'Ana', tags: ['js','fp'] };
const json = JSON.stringify(obj);           // "{\"id\":1,\"nom\":\"Ana\",\"tags\":
[\"js\",\"fp\"]}"
const pretty = JSON.stringify(obj, null, 2); // indenté sur 2 espaces
```

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

Désérialise une chaîne JSON en objet JS. Peut lever une exception si la chaîne est invalide.

```
const txt = '{"id":1,"nom":"Ana"}';
const data = JSON.parse(txt); // { id:1, nom:'Ana' }
// Attention: JSON.parse('{"id":}') lèvera une erreur (try/catch au besoin).
```

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

Découpe une chaîne selon un séparateur (caractère ou regex) et retourne un tableau.

```
'a,b,c'.split(',');                 // ['a','b','c']
'Hello  world'.split(/\s+/);        // ['Hello','world']
'2025-11-10'.split('-').map(Number);// [2025,11,10]
```

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

Supprime les espaces en début/fin (`trim`), uniquement début (`trimStart`) ou fin (`trimEnd`).

```
'  Hello  '.trim();       // 'Hello'
'  Hello  '.trimStart();  // 'Hello  '
'  Hello  '.trimEnd();    // '  Hello'
```

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

Complète une chaîne avec des caractères jusqu'à atteindre une longueur donnée (début/fin).

```
'5'.padStart(2, '0');      // '05'  (ex: numéros, dates)
'JS'.padEnd(5, '.');       // 'JS...'
`${42}`.padStart(4);       // '  42' (espaces par défaut)
```

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)

Variantes de log avec niveaux d'importance (filtrables dans les DevTools).

```
console.info('Info:', { step: 1 });
console.warn('Attention: quota élevé');
console.error('Erreur critique', new Error('Oups'));
```

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

Affiche un tableau d'objets sous forme de table lisible.

```
const users = [{ id:1, nom:'Ana' }, { id:2, nom:'Noé' }];
console.table(users);
```

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

Mesurer une durée entre `time()` et `timeEnd()` (avec logs intermédiaires via `timeLog()`).

```
console.time('task');
// ... code ...
console.timeLog('task', 'milestone atteint');
// ... code ...
console.timeEnd('task'); // 'task: 12.34 ms'
```

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

Itérer pour des effets de bord (log, push, DOM...). Ne retourne rien d'utile (toujours `undefined`).

```
const arr = [1,2,3];
arr.forEach((n, i) => console.log(i, n));
```

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

Retourne un itérateur `[index, valeur]`. Pratique avec `for...of`.

```
const arr = ['a','b','c'];
for (const [i, v] of arr.entries()) {
  console.log(i, v);
}
```

`in` - parcourir les clés d'un tableau

`for...in` parcourt les clés (indices sous forme de chaînes) et propriétés énumérables. Pour les tableaux, préférer `for...of` ou les méthodes `map/filter/...`.

```
const arr = ['a','b'];
for (const i in arr) { console.log(i, arr[i]); } // i: '0','1'
```

`of` - parcourir les valeurs d'un tableau

`for...of` parcourt directement les valeurs d'un itérable (tableau, Set, Map...).

```
for (const v of ['a','b','c']) { console.log(v); }
```

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

Retourne le premier élément qui passe le test, sinon `undefined`.

```
const users = [{id:1},{id:2}];
users.find(u => u.id === 2); // {id:2}
users.find(u => u.id === 3); // undefined
```

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

Retourne l'index du premier match, sinon `-1`.

```
['a','b','c'].findIndex(x => x === 'b'); // 1
['a','b'].findIndex(x => x === 'z');     // -1
```

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

Recherche par égalité stricte (valeurs primitives). `indexOf` depuis le début, `lastIndexOf` depuis la fin.

```
const arr = ['a','b','a'];
arr.indexOf('a');      // 0
arr.lastIndexOf('a');  // 2
arr.indexOf('z');      // -1
```

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

Méthodes mutables: `push` / `pop` fin du tableau, `unshift` / `shift` début du tableau.

```
const a = [1];
a.push(2);     // a = [1,2]
a.pop();       // a = [1]
a.unshift(0);  // a = [0,1]
a.shift();     // a = [1]
```

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

Retourne une copie partielle (non mutante). Fin exclue. Indices négatifs pris depuis la fin.

```
const a = ['a','b','c','d'];
a.slice(1, 3);   // ['b','c']
a.slice(-2);     // ['c','d']
```

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

Modifie le tableau en place et retourne les éléments supprimés.

```
const a = ['a','b','c'];
a.splice(1, 1);            // supprime 'b' → retourne ['b']; a = ['a','c']
a.splice(1, 0, 'X','Y');   // insère → a = ['a','X','Y','c']
```

`concat()` - joindre deux tableaux

Retourne un nouveau tableau avec les éléments de plusieurs listes.

```
[1,2].concat([3,4]);   // [1,2,3,4]
// alternatif moderne
[...['a','b'], ...['c']]; // ['a','b','c']
```

`join()` - joindre des chaînes de caractères

Concatène les éléments d'un tableau en une chaîne (séparateur par défaut: `,`).

```
['a','b','c'].join('-'); // 'a-b-c'
```

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

Lister les clés, valeurs (ou couples) d'un objet via `Object.*`.

```
const user = { id:1, nom:'Ana' };
Object.keys(user);    // ['id','nom']
Object.values(user);  // [1,'Ana']
Object.entries(user); // [['id',1],['nom','Ana']]
```

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

Teste la présence d'une valeur (utilise SameValueZero: gère aussi `NaN`).

```
[1,2,3].includes(2);     // true
[NaN].includes(NaN);     // true
['a','b'].includes('z'); // false
```

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

`every` : tous les éléments passent le test. `some` : au moins un élément passe.

```
const nums = [2,4,6];
nums.every(n => n % 2 === 0); // true
nums.some(n => n > 5);        // true
```

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

Remplit une portion (mutant) avec une valeur. Utile pour initialiser des tableaux.

```
new Array(3).fill(0);        // [0,0,0]
const a = [1,2,3,4]; a.fill(9, 1, 3); // [1,9,9,4]
// Attention: a.fill({}) réplique la même référence d'objet
```

`flat()` - aplatir un tableau

Aplati d'un certain niveau (1 par défaut). `Infinity` pour tout aplatir.

```
[1,[2,3],[4,[5]]].flat();     // [1,2,3,4,[5]]
[1,[2,3],[4,[5]]].flat(2);    // [1,2,3,4,5]
```

`sort()` - pour trier un tableau

Trie sur place. Par défaut: ordre lexicographique. Utiliser une fonction de comparaison pour les nombres/objets.

```
[3, 20, 5].sort();           // [20,3,5] (lexicographique)
[3, 20, 5].sort((a,b) => a-b); // [3,5,20]
[{nom:'Ana'},{nom:'Noé'}].sort((a,b) => a.nom.localeCompare(b.nom));
```

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

En langage humain: transformer chaque élément du tableau en quelque chose d'autre, sans changer la longueur. Idéal pour formater, extraire des champs, calculer des valeurs dérivées. Ne pas muter les éléments originaux.

```
// Transformer
const nums = [1, 2, 3];
const doubled = nums.map(n => n * 2); // [2, 4, 6]

// Extraire un champ
const produits = [{ id: 1, nom: 'Stylo', prix: 2.5 }, { id: 2, nom: 'Cahier', prix: 4.2 }];
const noms = produits.map(p => p.nom); // ['Stylo','Cahier']

// Formater
const labels = produits.map(p => `${p.nom} — CHF ${p.prix.toFixed(2)}`);

// Ajouter un champ dérivé (sans muter)
const avecTVA = produits.map(p => ({ ...p, prixTTC: +(p.prix * 1.077).toFixed(2) }));
```

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

En langage humain: garder uniquement ce qui nous intéresse selon un test (prédicat). La longueur peut diminuer. Parfait pour nettoyer, rechercher, appliquer des règles métier.

```
// Garder les pairs
const nums = [1,2,3,4,5,6];
const pairs = nums.filter(n => n % 2 === 0); // [2,4,6]

// Filtrer des objets
const users = [{nom:'Ana', age:22},{nom:'Noé', age:17},{nom:'Léa', age:19}];
const majeurs = users.filter(u => u.age >= 18); // Ana, Léa

// Nettoyer une liste (retirer valeurs falsy)
const brut = ['A', '', null, 'B', undefined, 'C'];
const propre = brut.filter(Boolean); // ['A','B','C']
```

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

Regroupe par clé dérivée. Variante portable via `reduce` (sans API expérimentale).

```
const groupBy = (arr, keyFn) => arr.reduce((acc, x) => {
  const k = keyFn(x);
  (acc[k] ||= []).push(x);
```

```
  return acc;
}, {});
// ex:
groupBy(['an','as','bo'], s => s[0]); // { a:['an','as'], b:['bo'] }
```

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

Applique un `map` qui retourne des tableaux, puis aplati d'un niveau en une seule passe.

```
['a b','c'].flatMap(s => s.split(' ')); // ['a','b','c']
// équiv. à: arr.map(...).flat()
```

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

En langage humain: tout combiner en un résultat unique (nombre, objet, tableau, Set…) en accumulant pas à pas. On démarre avec une valeur initiale et on «plie» la liste.

```
// Somme
const nums = [2, 5, 7];
const somme = nums.reduce((acc, n) => acc + n, 0); // 14

// Min / Max
const min = nums.reduce((m, n) => n < m ? n : m, +Infinity); // 2
const max = nums.reduce((m, n) => n > m ? n : m, -Infinity); // 7

// Compter les occurrences
const fruits = ['pomme','banane','pomme','kiwi'];
const compte = fruits.reduce((acc, f) => ((acc[f] = (acc[f]||0)+1), acc), {});
// { pomme: 2, banane: 1, kiwi: 1 }

// Regrouper par clé (clé -> tableau)
const animaux = [{nom:'Tigrou', type:'chat'},{nom:'Rex', type:'chien'},{nom:'Maya',
type:'chat'}];
const groupes = animaux.reduce((acc, a) => ((acc[a.type] ??= []).push(a), acc), {});
```

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

Inverse l'ordre des éléments (mutant).

```
const a = [1,2,3];
a.reverse(); // a = [3,2,1]
```

Techniques

``(backticks) - pour des expressions intelligentes

Les «template literals» permettent l'interpolation (`${expr}`), les chaînes multi-lignes et des tags avancés.

```
const nom = 'Ana', total = 12.5;
const msg = `Bonjour ${nom}, votre total est CHF ${total.toFixed(2)}.`;
// Multi-ligne:
const html = `
  <div class="box">
    <p>${msg}</p>
```

```
  </div>
`;
```

`new Set()` - pour supprimer les doublons

Structure d'ensemble: garde des valeurs uniques. Idéal pour dédupliquer.

```
const arr = [1,2,2,3,1];
const uniques = [...new Set(arr)]; // [1,2,3]
// API utile:
const s = new Set(); s.add(1); s.add(1); s.has(1); // true
```

Design pattern — Builder (PCBuilder)

Pourquoi l'utiliser

- Séparer la construction d'un objet complexe de sa représentation finale.
- Offrir une API fluide (chaînage) avec des valeurs par défaut et une validation centralisée.
- Éviter une explosion de constructeurs avec de multiples paramètres optionnels.
- Rendre la création lisible, progressive et sûre.

Cas d'usage typiques:

- Objets avec beaucoup d'options (PC, configuration, requêtes HTTP, exports).
- Champs parfois exclusifs (SSD/HDD), valeurs par défaut (1 écran), conversions (string → number).

Exemple: PC et PCBuilder

```
// Modèle PC (objet final)
class PC {
  constructor(cpu, ramGB, ssdGB, hddGB, gpu, screens, os) {
    this.cpu = cpu || '';
    this.ramGB = Number.isFinite(ramGB) ? ramGB : 0;
    this.ssdGB = Number.isFinite(ssdGB) ? ssdGB : 0;
    this.hddGB = Number.isFinite(hddGB) ? hddGB : 0;
    this.gpu = gpu || '';
    this.screens = Number.isFinite(screens) ? screens : 1;
    this.os = os || '';
  }
  toString() {
    return `PC[cpu=${this.cpu}, ram=${this.ramGB}Go, ssd=${this.ssdGB}Go, hdd=${this.hddGB}Go,
gpu=${this.gpu}, ecrans=${this.screens}, os=${this.os}]`;
  }
}

// Builder fluide
class PCBuilder {
  constructor() {
    this._cpu = '';
    this._ramGB = 0;
    this._ssdGB = 0;
    this._hddGB = 0;
    this._gpu = '';
    this._screens = 1;
    this._os = '';
  }
```

```
  setCPU(cpu) { this._cpu = cpu || ''; return this; }
  setRAM(ramGB) { this._ramGB = Number(ramGB); return this; }
  setSSD(ssdGB) { this._ssdGB = Number(ssdGB); return this; }
  setHDD(hddGB) { this._hddGB = Number(hddGB); return this; }
  setGPU(gpu) { this._gpu = gpu || ''; return this; }
  setScreens(screens) { this._screens = Number(screens); return this; }
  setOS(os) { this._os = os || ''; return this; }
  build() {
    // Point unique pour valider/cohérence (ex.: RAM>0, au moins un stockage, etc.)
    return new PC(this._cpu, this._ramGB, this._ssdGB, this._hddGB, this._gpu, this._screens,
this._os);
  }
}

// Utilisation
const pc = new PCBuilder()
  .setCPU('Intel i7')
  .setGPU('RTX 4070')
  .setRAM(32)
  .setSSD(1000)
  .setOS('Ubuntu 24.04')
  .build();
```

Points clés:

- Chaque setter retourne `this` → chaînage fluide.
- Le builder encapsule conversions/valeurs par défaut → évite la logique éparpillée.
- `build()` est le point unique de validation finale.

Variantes utiles

- Champs obligatoires: exposer un `PCBuilder.withRequired(cpu)` ou un constructeur de builder qui exige certains paramètres.
- Validation forte dans `build()` (ex.: RAM > 0, au moins un disque défini, écrans ≥ 1).
- Immutabilité: retourner un nouveau builder à chaque `setX` (style plus “fonctionnel”), au prix de plus d’allocations.
- Présets: `PCBuilder.gamerPreset()` ou `PCBuilder.officePreset()` qui préremplissent des valeurs.
- Builder “fonctionnel” (with): `withCPU(b)(val)` qui retourne un nouveau builder; pratique pour composer/mapper.

Quand l’utiliser (et quand éviter)

- À utiliser si:
  - Beaucoup d’options/facettes, dont la combinaison varie.
  - Besoin d’une API lisible qui guide l’appelant.
  - Validation/cohérence métier non triviales.
- À éviter si:
  - L’objet a 2–3 champs simples → un simple constructeur/objet littéral suffit.
  - Les paramètres sont tous obligatoires et peu nombreux → un constructeur clair suffit.

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 fléchées (arrow functions)

Pourquoi les utiliser ?

- Courtes et lisibles pour des callbacks (`map`, `filter`, etc.).
- Pas de nouveau **`this`** créé (utile avec des méthodes/timeout).
- Parfaites pour des fonctions simples et anonymes.

Exemples concrets:

```
// 1) De fonction classique → fonction fléchée
function add(a, b) { return a + b; }
const addArrow = (a, b) => { return a + b; };

// 2) Retour implicite (quand c'est une seule expression)
const square = n => n * n;

// 3) Retourner un objet littéral → parenthèses obligatoires
const userPreview = user => ({ id: user.id, name: user.name });

// 4) En callbacks avec map/filter/reduce
const people = [
  { name: 'Mina', score: 10 },
  { name: 'Alex', score: 18 },
  { name: 'Zoé',  score: 15 },
];

const names = people.map(p => p.name);                // ['Mina','Alex','Zoé']
const passed = people.filter(p => p.score >= 15);     // Alex, Zoé
const avg    = people.reduce((a, p, _, arr) => a + p.score / arr.length, 0);
// 14.333...

// 5) this lexical (rapide): pas de nouveau this
const timer = {
  count: 0,
  start() {
    setTimeout(() => {
```

```
      this.count += 1; // ici this === timer
    }, 10);
  }
};
```

À retenir:

- Retour implicite pratique, mais cela ne convient pas en cas de plusieurs instructions.
- Pour retourner un objet, il faut utiliser des parenthèses: `() => ({ ... })`.
- Idéal en petites briques à composer.

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

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

Immuabilité

Idée simple: on ne mute pas une valeur existante; on crée une nouvelle valeur.

Exemples rapides:

```
// Objets
const user = { id: 1, name: 'Ana' };
// ❌ mutation
user.name = 'Anaïs';
// ✅ immuable (nouvel objet)
const user2 = { ...user, name: 'Anaïs' };

// Tableaux
const a = [1, 2];
// ❌ mutation
a.push(3);
// ✅ immuable (nouveau tableau)
const b = [...a, 3];

// Retirer un élément: filter
const without2 = b.filter(n => n !== 2);

// Remplacer un élément: map
const replaced = b.map(n => (n === 3 ? 30 : n));
```

Anti-pattern classique avec `map`:

```
// ❌ Muter les objets internes
const eleves = [
  { nom: 'Mina', notes: [5, 4] },
```

```
  { nom: 'Alex', notes: [3.5, 4.5] },
];
const r1 = eleves.map(e => {
  const avg = e.notes.reduce((a, n, _, arr) => a + n / arr.length, 0);
  e.moyenne = +avg.toFixed(2); // mutation
  return e;
});

// ✅ Retourner de nouveaux objets
const r2 = eleves.map(e => {
  const avg = e.notes.reduce((a, n, _, arr) => a + n / arr.length, 0);
  return { ...e, moyenne: +avg.toFixed(2) };
});
```

Cas un peu plus profond (mise à jour d'un item dans une liste imbriquée):

```
const state = {
  cart: {
    items: [
      { id: 'p1', qty: 1, price: 2.5 },
      { id: 'p2', qty: 2, price: 4.2 },
    ]
  }
};

// ✅ immuable: recréer chaque niveau
const inc = id => ({ cart: { items } }) => ({
  cart: {
    items: items.map(it => it.id === id ? { ...it, qty: it.qty + 1 } : it)
  }
});

const newState = inc('p1')(state);
```

Note: pour des structures très profondes, `structuredClone(obj)` (si dispo) ou des utilitaires dédiés peuvent aider, mais garder le principe: ne pas muter, recréer.

Fonctions pures

Définition courte: même entrée → même sortie, pas d'effet de bord.

Impure → Pure:

```
// ❌ Impure: lit une variable globale et la modifie
let rate = 0.077;
const addVatImpure = price => price * (1 + rate);
rate = 0.08; // change le résultat sans changer l'input

// ✅ Pure: injecter les dépendances en paramètres (currying possible)
const addVat = vat => price => price * (1 + vat);
const addVat077 = addVat(0.077);

// ❌ Impure: side effect (modifie le tableau reçu)
const addItemImpure = (arr, x) => { arr.push(x); return arr; };

// ✅ Pure: retourner un nouveau tableau
const addItem = (arr, x) => [...arr, x];
```

```
// ❌ Impure: dépend de l'heure courante
const greetingImpure = name => `Hello ${name} @ ${new Date().toISOString()}`;

// ✅ Pure: l'horloge est passée en argument (testable)
const greeting = (clock, name) => `Hello ${name} @ ${clock.nowISO()}`;
const fixedClock = { nowISO: () => '2025-11-03T10:00:00.000Z' };
```

Règle pratique: placer les effets (log, réseau, DOM, Date, Math.random) aux bords de l'application. Le cœur (les fonctions de transformation) reste pur, donc facile à tester.

Composition (currying, closure, pipe, compose)

Currying

Transformer une fonction multi-arguments en chaîne de fonctions unaires pour mieux composer.

```
// add(a, b) → curry → add(a)(b)
const add = a => b => a + b;
const inc = add(1);
inc(41); // 42

// Utilité: passer une fonction unaire à map/filter
const plus10 = add(10);
[1,2,3].map(plus10); // [11,12,13]

// Petit utilitaire de curry (démonstration simple)
const curry2 = fn => a => b => fn(a, b);
const multiply = (a, b) => a * b;
const double = curry2(multiply)(2);
double(6); // 12
```

Closure

Une fonction «se souvient» des variables de son contexte de création.

```
// 1) Compteur avec état privé
const makeCounter = () => {
  let count = 0;
  return () => ++count;
};
const c1 = makeCounter();
c1(); // 1
c1(); // 2

// 2) Fabrique de prédicats (réutilisable avec filter)
const minAge = min => user => user.age >= min; // min capturé
const canDrink = minAge(18);
[{age:16},{age:20}].filter(canDrink); // [{age:20}]
```

pipe / compose

Composer des fonctions pures en chaîne lisible.

```
const pipe = (...fns) => x => fns.reduce((v, f) => f(v), x);
const compose = (...fns) => x => fns.reduceRight((v, f) => f(v), x);
```

```
// Fonctions simples
const trim = s => s.trim();
const lower = s => s.toLowerCase();
const slugify = s => s.replace(/\s+/g, '-');

// Pipeline lisible (gauche → droite)
const toSlug = pipe(trim, lower, slugify);
toSlug('  Hello World  '); // 'hello-world'
```

Cas pratique: panier → filtrer en stock → prix TTC → total.

```
const inStock = p => p.inStock;
const map = fn => arr => arr.map(fn);
const filter = fn => arr => arr.filter(fn);
const reduce = (fn, init) => arr => arr.reduce(fn, init);

const addVat = vat => price => +(price * (1 + vat)).toFixed(2);
const priceTtc = vat => p => ({ ...p, price: addVat(vat)(p.price) });
const sum = reduce((a, n) => a + n, 0);

const totalTtc = vat => pipe(
  filter(inStock),
  map(priceTtc(vat)),
  map(p => p.price * p.qty),
  sum,
);

const products = [
  { name: 'Stylo',  price: 2.5, qty: 2, inStock: true },
  { name: 'Cahier', price: 4.2, qty: 1, inStock: false },
  { name: 'Gomme',  price: 1.1, qty: 3, inStock: true },
];

totalTtc(0.077)(products); // ex: 7.82
```

Astuce: `compose` fait la même chose que `pipe` mais de droite → gauche. Il est possible d'utiliser celui dont la lecture semble la plus naturelle.

Récursion

Idée: une fonction s'appelle elle-même jusqu'à une condition d'arrêt.

Exemple simple (compte à rebours):

```
// Itératif
const countdownIter = n => {
  const out = [];
  for (let i = n; i >= 0; i--) out.push(i);
  return out;
};

// Récursif
const countdownRec = n => (n < 0 ? [] : [n, ...countdownRec(n - 1)]);
```

Parcours d'un arbre (somme des tailles):

```
const tree = {
  name: 'root', size: 2,
  children: [
    { name: 'A', size: 3, children: [] },
    { name: 'B', size: 1, children: [ { name: 'B1', size: 4, children: [] } ] },
  ]
};

const sumSizes = node => node.size + node.children.reduce((a, c) => a + sumSizes(c), 0);
sumSizes(tree); // 10
```

Alternative itérative (éviter les très grandes profondeurs → risque de stack overflow en JS):

```
const sumSizesIter = root => {
  let total = 0;
  const stack = [root];
  while (stack.length) {
    const node = stack.pop();
    total += node.size;
    stack.push( ... node.children);
  }
  return total;
};
```

À retenir:

- Il faut toujours définir une condition d'arrêt claire.
- En JS, la TCO (optimisation de récursion terminale) n'est pas fiable → pour de grandes profondeurs, il est préférable d'utiliser une version itérative (pile manuelle).

Annexes (mémo + liens + export PDF)

Mémo express:

- map: `(fn) => Array` → transforme chaque élément, garde la longueur.
- filter: `(pred) => Array` → garde les éléments où `pred(x)` est true.
- reduce: `(reducer, init) => any` → plie en une valeur.
- pure: aucune dépendance cachée, pas d'effet de bord.
- immuable: créer de nouvelles valeurs (`{...obj}` , `[...arr]`).
- pipe/compose: composer des fonctions unaires.

Liens utiles:

- MDN Array.map: developer.mozilla.org → Array/map
- MDN Array.filter: developer.mozilla.org → Array/filter
- MDN Array.reduce: developer.mozilla.org → Array/reduce

Astuces PDF:

- Placer les styles d'impression dans `res/pdf-style.css` et les activer avant l'export.
- Impression: Cmd/Ctrl+P → «Enregistrer au format PDF».
- Conseils:
  - Police lisible (ex: Inter, 12–13px impression), marges régulières.
  - Utiliser des titres courts, du code bien indenté, et éviter les lignes trop longues.

Programmation fonctionnelle — Cas pratiques

Ces exemples montrent comment résoudre des besoins concrets en utilisant `map`, `filter`, `reduce`, `flat`, `flatMap`, `sort` ... en mode déclaratif et réutilisable.

Données utilisées

```
// jsonData.ventes est un tableau d'objets "vente"
// Chaque "vente" a une date (YYYY-MM-DD), un jour, un client et une liste de produits
// (structure représentative)
const jsonData = {
  ventes: [
    {
      date: "2024-09-02",
      jour: "lundi", // parfois calculable via new Date(date).getDay()
      client: { id: "ID...", nom: "Nom", prenom: "Prénom" },
      produits: [
        { id: "IDP...", type: "Boisson chaude", nom: "Café", prix: 2.0 },
        // ...
      ]
    },
    // ...
  ]
};
```

Boîte à outils réutilisable

```
// Capitaliser un prénom: "alEx" -> "Alex"
const formatPrenom = p => p ? p.charAt(0).toUpperCase() + p.slice(1).toLowerCase() : p;

// Somme du prix d'une vente (liste de produits)
const sommePrixVente = (vente) => vente.produits.reduce((s, p) => s + p.prix, 0);

// Top N générique (copie pour ne pas muter)
const topN = (arr, n, keyFn) => [...arr].sort((a, b) => keyFn(b) - keyFn(a)).slice(0, n);
```

A01 - La plus petite et la plus grande date

Objectif: retrouver `dateMin` et `dateMax` en une passe.

```
const rapportA01 = (jsonData) => jsonData.ventes.reduce((acc, v) => {
  const d = v.date;
  return {
    dateMin: acc.dateMin === null || d < acc.dateMin ? d : acc.dateMin,
    dateMax: acc.dateMax === null || d > acc.dateMax ? d : acc.dateMax
  };
}, { dateMin: null, dateMax: null });
```

A02 - Le chiffre d'affaire quotidien moyen

Idée: sommer tous les CA des ventes et diviser par le nombre de dates distinctes.

```
const rapportA02 = (jsonData) => {
  const { totalCA, jours } = jsonData.ventes.reduce((acc, v) => {
    const caVente = v.produits.reduce((s, p) => s + p.prix, 0);
    acc.totalCA += caVente;
```

```
    acc.jours.add(v.date);
    return acc;
  }, { totalCA: 0, jours: new Set() });
  return totalCA / jours.size;
};
```

A03 - Liste des clients par id

Objectif: indexer les clients par `id` (clé → valeur).

```
const rapportA03 = (jsonData) => jsonData.ventes.reduce((acc, v) => {
  const { id, nom, prenom } = v.client;
  if (!acc[id]) acc[id] = { nom, prenom };
  return acc;
}, {});
```

A04 - Liste des clients et leur CA

On agrège le CA par client, puis on retourne les valeurs sous forme de tableau.

```
const rapportA04 = (jsonData) => {
  const map = jsonData.ventes.reduce((acc, v) => {
    const { id, nom, prenom } = v.client;
    const caVente = v.produits.reduce((s, p) => s + p.prix, 0);
    if (!acc[id]) acc[id] = { id, nom, prenom, ca: 0 };
    acc[id].ca += caVente;
    return acc;
  }, {});
  return Object.values(map);
};
```

A05 - CA et nombre de ventes par jour de la semaine

Selon les données, `vente.jour` peut être déjà présent. Sinon, on peut dériver avec `new Date(vente.date).getDay()`.

```
const rapportA05 = (jsonData) => jsonData.ventes.reduce((acc, v) => {
  const jour = v.jour;
  const caVente = v.produits.reduce((s, p) => s + p.prix, 0);
  const courant = acc[jour] || { ca: 0, nbre_ventes: 0 };
  acc[jour] = { ca: courant.ca + caVente, nbre_ventes: courant.nbre_ventes + v.produits.length
};
  return acc;
}, {});
```

A06 - CA par type de produit

On aplatit d'abord toutes les lignes de produits, puis on agrège par `type`.

```
const rapportA06 = (jsonData) => jsonData.ventes
  .flatMap(v => v.produits)
  .reduce((acc, p) => {
    acc[p.type] = (acc[p.type] || 0) + p.prix;
    return acc;
  }, {});
```

A07 - Types de produits et leurs produits (noms uniques)

On utilise des `Set` pour dédupliquer, puis on transforme en tableaux simples.

```
const rapportA07 = (jsonData) => {
  const sets = jsonData.ventes.reduce((acc, v) => {
    return v.produits.reduce((a, p) => {
      (a[p.type] ||= new Set()).add(p.nom);
      return a;
    }, acc);
  }, {});
  return Object.fromEntries(
    Object.entries(sets).map(([type, s]) => [type, Array.from(s)])
  );
};
```

A08 - Nombre de ventes des produits par type

Agrégation imbriquée: `type` → `nom` → compteur.

```
const rapportA08 = (jsonData) => jsonData.ventes.reduce((acc, v) => {
  return v.produits.reduce((a, p) => {
    (a[p.type] ||= {});
    a[p.type][p.nom] = (a[p.type][p.nom] || 0) + 1;
    return a;
  }, acc);
}, {});
```

A09 - Toutes les ventes réalisées (liste à plat)

On projette chaque produit en une ligne “vente” au format souhaité.

```
const rapportA09 = (jsonData) => jsonData.ventes.flatMap(v =>
  v.produits.map(p => ({ id: p.id, type: p.type, nom: p.nom, prix: p.prix }))
);
```

A10 - TOP 6 des produits générant le plus de CA

Agrégation par `id` de produit → tri décroissant → coupe `TOP 6`.

```
const rapportA10 = (jsonData, TOP = 6) => {
  const agregats = jsonData.ventes.reduce((acc, v) => {
    return v.produits.reduce((a, p) => {
      if (!a[p.id]) a[p.id] = { id: p.id, nom: p.nom, ca: 0 };
      a[p.id].ca += p.prix;
      return a;
    }, acc);
  }, {});
  return Object.values(agregats).sort((a, b) => b.ca - a.ca).slice(0, TOP);
};
```

A11 - La liste des clients, triée par nom puis par prénom

On dédoublonne par `id`, on transforme le nom/prénom, puis on trie.

```
const rapportA11 = (jsonData) => {
  const uniques = Object.values(jsonData.ventes.reduce((acc, v) => {
    const { id, nom, prenom } = v.client;
    if (!acc[id]) acc[id] = { nom, prenom };
    return acc;
```

```
  }, {}));
  return uniques
    .map(c => ({ nom: c.nom.toUpperCase(), prenom: formatPrenom(c.prenom) }))
    .sort((a, b) => a.nom.localeCompare(b.nom) || a.prenom.localeCompare(b.prenom));
};
```

A12 - Le TOP 5 des meilleures dates et leur CA

Agrégation par date → conversion en tableau → tri → coupe `TOP 5`.

```
const rapportA12 = (jsonData, TOP = 5) => {
  const caParDate = jsonData.ventes.reduce((acc, v) => {
    const caVente = v.produits.reduce((s, p) => s + p.prix, 0);
    acc[v.date] = (acc[v.date] || 0) + caVente;
    return acc;
  }, {});
  return Object.entries(caParDate)
    .map(([date, ca]) => ({ date, ca }))
    .sort((a, b) => b.ca - a.ca)
    .slice(0, TOP);
};
```

Exercice Drill 11 — Notes scolaires

Ces exemples reprennent un dataset d'évaluations scolaires `jsonData.evaluations` avec des champs typiques: `prenom`, `nom`, `branche`, `date` (dd.mm.yyyy), `note` (nombre).

Utilitaires (frCompare, normalizeString, round2, parseDate, groupBy)

```
const frCompare = (a, b) => a.localeCompare(b, 'fr', { sensitivity: 'base' });

const normalizeString = (s) =>
  (s ?? '')
    .normalize('NFD')
    .replace(/[\u0300-\u036f]/g, '')
    .toLowerCase();

const round2 = (n) => Math.round(n * 100) / 100;

const parseDate = (dstr) => {
  const [d, m, y] = dstr.split('.').map(Number);
  return new Date(y, m - 1, d);
};

const groupBy = (array, keyFn) =>
  array.reduce((acc, item) => {
    const key = keyFn(item);
    (acc[key] ||= []).push(item);
    return acc;
  }, {});
```

A1 - [Branches], trié

Objectif: extraire la liste unique des branches, triées FR.

```
const ex11A1 = (jsonData) =>
  Array.from(new Set(jsonData.evaluations.map(e => e.branche))).sort(frCompare);
```

A2 - [Prénom NOM], trié

Objectif: lister les personnes uniques, triées par prénom puis nom, avec NOM en MAJUSCULES.

```
const ex11A2 = (jsonData) => {
  const map = jsonData.evaluations.reduce((m, e) => {
    const key = `${e.prenom}|${e.nom}`;
    if (!m.has(key)) m.set(key, { prenom: e.prenom, nom: e.nom });
    return m;
  }, new Map());
  return Array.from(map.values())
    .sort((a, b) =>
      a.prenom.localeCompare(b.prenom, 'fr', { sensitivity: 'base' }) ||
      a.nom.localeCompare(b.nom, 'fr', { sensitivity: 'base' })
    )
    .map(p => `${p.prenom} ${p.nom.toUpperCase()}`);
};
```

B - {Branches : nbre de notes}

Objectif: compter les notes par branche et organiser l'objet final dans l'ordre FR des clés.

```
const ex11B = (jsonData) => {
  const counts = jsonData.evaluations.reduce((acc, e) => {
    acc[e.branche] = (acc[e.branche] || 0) + 1;
    return acc;
  }, {});
  return Object.keys(counts).sort(frCompare)
    .reduce((obj, b) => (obj[b] = counts[b], obj), {});
};
```

C - [{Branche, nbre notes}], trié

Objectif: tableau trié des paires { branche, count }.

```
const ex11C = (jsonData) => {
  const counts = jsonData.evaluations.reduce((acc, e) => {
    acc[e.branche] = (acc[e.branche] || 0) + 1;
    return acc;
  }, {});
  return Object.keys(counts).sort(frCompare)
    .map(b => ({ branche: b, count: counts[b] }));
};
```

T1 - Apprentis contenant Partie connue

Objectif: si `saisie` vide → tous; sinon filtrer (insensible accent/casse) les "Prénom NOM" contenant la partie connue.

```
const ex11T1 = (jsonData, saisie = "") => {
  const personnes = Array.from(jsonData.evaluations.reduce((m, e) => {
    const key = `${e.prenom}|${e.nom}`;
    if (!m.has(key)) m.set(key, { prenom: e.prenom, nom: e.nom });
    return m;
  }, new Map()).values());
```

```
  const filtre = (saisie || "").trim();
  const candidats = filtre === ""
    ? personnes
    : personnes.filter(p =>
        normalizeString(`${p.prenom} ${p.nom}`)
          .includes(normalizeString(filtre)));

  return candidats
    .sort((a, b) =>
      a.prenom.localeCompare(b.prenom, 'fr', { sensitivity: 'base' }) ||
      a.nom.localeCompare(b.nom, 'fr', { sensitivity: 'base' })
    )
    .map(p => `${p.prenom} ${p.nom.toUpperCase()}`);
};
```

T2 - Total évaluations de cette branche

Objectif: si vide → total général; sinon, total des évals de la branche (insensible accent/casse).

```
const ex11T2 = (jsonData, nomBranche = "") => {
  const b = (nomBranche || "").trim();
  if (b === "") return jsonData.evaluations.length;
  return jsonData.evaluations
    .filter(e => normalizeString(e.branche) === normalizeString(b)).length;
};
```

D - [{Branche, moyenne}], trié

Objectif: moyenne arrondie à 2 décimales par branche (tri FR).

```
const ex11D = (jsonData) => {
  const groupes = groupBy(jsonData.evaluations, e => e.branche);
  return Object.keys(groupes).sort(frCompare).map(b => {
    const notes = groupes[b].map(x => x.note);
    const moyenne = round2(notes.reduce((s, v) => s + v, 0) / notes.length);
    return { branche: b, moyenne };
  });
};
```

E - [{Branche, nbre, moyenne}], trié

Objectif: même que D, en ajoutant le `count`.

```
const ex11E = (jsonData) => {
  const groupes = groupBy(jsonData.evaluations, e => e.branche);
  return Object.keys(groupes).sort(frCompare).map(b => {
    const notes = groupes[b].map(x => x.note);
    const count = notes.length;
    const moyenne = round2(notes.reduce((s, v) => s + v, 0) / count);
    return { branche: b, count, moyenne };
  });
};
```

F - [{Elève, [branches, [notes]]}], trié

Objectif: par élève → branches triées FR → notes triées chronologiquement (valeurs numériques uniquement).

```
const ex11F = (jsonData) => {
  const eleveMap = jsonData.evaluations.reduce((m, e) => {
    const key = `${e.prenom}|${e.nom}`;
    if (!m.has(key)) m.set(key, { prenom: e.prenom, nom: e.nom, branches: new Map() });
    const ref = m.get(key);
    if (!ref.branches.has(e.branche)) ref.branches.set(e.branche, []);
    ref.branches.get(e.branche).push({ date: e.date, note: e.note });
    return m;
  }, new Map());

  return Array.from(eleveMap.values())
    .sort((a, b) =>
      a.prenom.localeCompare(b.prenom, 'fr', { sensitivity: 'base' }) ||
      a.nom.localeCompare(b.nom, 'fr', { sensitivity: 'base' })
    )
    .map(p => {
      const branches = Array.from(p.branches.keys())
        .sort(frCompare)
        .map(br => {
          const notes = p.branches.get(br)
            .slice()
            .sort((x, y) => parseDate(x.date) - parseDate(y.date))
            .map(x => x.note);
          return { branche: br, notes };
        });
      return { eleve: `${p.prenom} ${p.nom.toUpperCase()}`, branches };
    });
};
```

G - [{Elève, [branches, nbre, min, max, moyenne, [notes]]}], trié

Objectif: comme F, en ajoutant statistiques par branche.

```
const ex11G = (jsonData) => {
  const eleveMap = jsonData.evaluations.reduce((m, e) => {
    const key = `${e.prenom}|${e.nom}`;
    if (!m.has(key)) m.set(key, { prenom: e.prenom, nom: e.nom, branches: new Map() });
    const ref = m.get(key);
    if (!ref.branches.has(e.branche)) ref.branches.set(e.branche, []);
    ref.branches.get(e.branche).push({ date: e.date, note: e.note });
    return m;
  }, new Map());

  return Array.from(eleveMap.values())
    .sort((a, b) =>
      a.prenom.localeCompare(b.prenom, 'fr', { sensitivity: 'base' }) ||
      a.nom.localeCompare(b.nom, 'fr', { sensitivity: 'base' })
    )
    .map(p => {
      const branches = Array.from(p.branches.keys())
        .sort(frCompare)
        .map(br => {
          const sorted = p.branches.get(br).slice()
            .sort((x, y) => parseDate(x.date) - parseDate(y.date));
          const notes = sorted.map(x => x.note);
          const nbre = notes.length;
          const min = Math.min(...notes);
          const max = Math.max(...notes);
          const moyenne = round2(notes.reduce((s, v) => s + v, 0) / nbre);
          return { branche: br, nbre, min, max, moyenne, notes };
        });
```

```
      return { eleve: `${p.prenom} ${p.nom.toUpperCase()}`, branches };
    });
};
```

H - [{Elève, [branches, nbre, min, max, moyenne, [{date, note}]]}], trié

Objectif: comme G, mais conserver les notes au format { date, note }.

```
const ex11H = (jsonData) => {
  const eleveMap = jsonData.evaluations.reduce((m, e) => {
    const key = `${e.prenom}|${e.nom}`;
    if (!m.has(key)) m.set(key, { prenom: e.prenom, nom: e.nom, branches: new Map() });
    const ref = m.get(key);
    if (!ref.branches.has(e.branche)) ref.branches.set(e.branche, []);
    ref.branches.get(e.branche).push({ date: e.date, note: e.note });
    return m;
  }, new Map());

  return Array.from(eleveMap.values())
    .sort((a, b) =>
      a.prenom.localeCompare(b.prenom, 'fr', { sensitivity: 'base' }) ||
      a.nom.localeCompare(b.nom, 'fr', { sensitivity: 'base' })
    )
    .map(p => {
      const branches = Array.from(p.branches.keys())
        .sort(frCompare)
        .map(br => {
          const sorted = p.branches.get(br).slice()
            .sort((x, y) => parseDate(x.date) - parseDate(y.date));
          const notesNums = sorted.map(x => x.note);
          const nbre = notesNums.length;
          const min = Math.min( ... notesNums);
          const max = Math.max( ... notesNums);
          const moyenne = round2(notesNums.reduce((s, v) => s + v, 0) / nbre);
          const notes = sorted.map(x => ({ date: x.date, note: x.note }));
          return { branche: br, min, max, nbre, moyenne, notes };
        });
      return { eleve: `${p.prenom} ${p.nom.toUpperCase()}`, branches };
    });
};
```

Conclusion

J'ai découvert une nouvelle façon de réfléchir au code : penser en mode déclaratif avec des trucs comme map, filter et reduce, garder mes données inchangées, et construire plein de petites fonctions simples qui font chacune un truc clair.
Je dois encore bosser mes réflexes sur la normalisation (accents, majuscules/minuscules), la gestion des dates et surtout le nommage pour que tout soit lisible direct.
Je pense que ce serait utile d'ajouter quelques tests rapides et de petites données d'exemple pour vérifier chaque rapport sans se prendre la tête.
Ce que j'ai moins aimé : certains passages un peu trop verbeux, la marche un peu violente entre les exos faciles et les trucs plus complexes, et la répétition dans les agrégations quand on n'a pas d'utilitaires.

Auteur : Noé Romanens
Version : v1
Date : 11.11.2025