🤔 RP - 323 - Programmation fonctionnelle

- Introduction
- Opérateurs javascript super-cooool 😎
  - 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
    - `for...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/supprimer au début/fin d'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
        - Approfondissement de `map()`
    - `filter()` - tableau avec les éléments passant un test
        - Approfondissement de `filter()`
    - `groupBy()` - regroupe les éléments d'un tableau selon une règle
    - `flatMap()` - chaînage de map() et flat()
    - `reduce()` et `reduceRight()` - réduire un tableau à une seule valeur
        - Approfondissement de `reduce()`
        - Combiner `map()`, `filter()` et `reduce()`
        - Composition de fonctions et pipe
        - Récursion
    - `reverse()` - inverser l'ordre du tableau
- Techniques
    - ``(backticks) - pour des expressions intelligentes
    - `new Set()` - pour supprimer les doublons
- Fonctions
    - Déclaration de fonction
    - Fonctions immédiatement invoquées (IIFE)
    - Currying et Closures
- Patterns avancés et cas d'usage réels
    - Parser et transformer du CSV

    - Regroupement par clé
    - Calcul de statistiques
    - Chaînages complexes
- Conclusion

Le but du module 323 est d'apprendre la **Programmation fonctionnelle (PF)**, mais aussi devenir capable de **penser déclaratif** et de construire des solutions **claires, testables et réutilisables**.

À la fin du module, nous saurons :

- utiliser efficacement `map()`, `filter()`, `reduce()`, `flatMap()`, `sort()` pour transformer des tableaux **sans effets de bord** ;
- écrire des **fonctions pures**, travailler avec des **données immuables** et raisonner avec des **fonctions de première classe** ;
- **composer** des petites fonctions (patterns *pipe/compose*), pratiquer le **currying** et exploiter les **closures** ;
- appliquer la **récursion** lorsque la structure du problème s'y prête (arbres, dossiers, parsing, etc.).

Objectifs opérationnels concrets :

- Transformer des données CSV en objets JavaScript structurés
- Filtrer, trier et agréger des données de manière déclarative
- Construire des pipelines de transformation réutilisables
- Manipuler des structures de données imbriquées (arbres, objets complexes)
- Calculer des statistiques (moyennes, min/max, regroupements)
- Optimiser le code pour la lisibilité et la maintenabilité

Opérateurs javascript super-cooool 😎

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

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

// Cas d'usage réel : affichage conditionnel
const clientsAvecStatut = clients.map(client => ({
  ...client,
  statut: client.ca > 10000 ? 'Premium' : 'Standard'
}));
```

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 pseudo = null ?? 'Invité'; // "Invité"
const compteur = 0 ?? 42; // 0 (car 0 n'est pas null/undefined)

// Cas d'usage réel : valeurs par défaut sûres
const nombreHabitants = ville.habitants ?? 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 prix = 0 || 99; // 99 => ATTENTION ! (0 est falsy)
const prix = 0 ?? 99; // 0 (correct avec ??)
```

```
const nom = '' || 'Anonyme'; // 'Anonyme' => ATTENTION !
const nom = '' ?? 'Anonyme'; // '' (correct si on veut garder l
```

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 config = { timeout: 50 };
config.timeout ??= 10; // pas fait (déjà défini)
config.retry ??= 3;    // fait => { timeout: 50, retry: 3 }

// Cas d'usage réel : initialisation d'objets
const stats = {};
stats.total ??= 0;
stats.count ??= 0;
```

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 tableaux
const nombresUn = [1, 2, 3];
const nombresDeux = [4, 5, 6];
const nombresCombines = [...nombresUn, ...nombresDeux]; // [1,2

// Extraire uniquement ce qui est utile
const nombres = [1, 2, 3, 4, 5, 6];
```

```
const [premier, deuxieme, ...reste] = nombres;
// premier: 1, deuxieme: 2, reste: [3,4,5,6]

// Fusion d'objets avec mise à jour (immuabilité!)
const vehicule = {
    marque: 'Ford',
    modele: 'Mustang',
    couleur: 'rouge',
};
const miseAJour = {
    type: 'voiture',
    annee: 2021,
    couleur: 'jaune',
};
const vehiculeMisAJour = { ...vehicule, ...miseAJour };
// { marque: 'Ford', modele: 'Mustang', couleur: 'jaune', type:

// Cas d'usage réel : copier sans muter
const produitsOriginaux = [{ nom: 'A', prix: 10 }];
const produitsAvecTVA = produitsOriginaux.map(p => ({
  ...p,
  prixTTC: p.prix * 1.077
}));
// L'original n'est pas modifié!
```

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 [premier, second, ...autres] = valeurs;
console.log(premier); // 1
console.log(second);  // 2
console.log(autres);  // [3, 4, 5, 6, 7, 8, 9, 10]

// Cas d'usage réel : extraire des propriétés d'objets
```

```
const eleve = { nom: 'Dupont', prenom: 'Marie', age: 20, classe
const { nom, prenom, ...infos } = eleve;
// nom: 'Dupont', prenom: 'Marie', infos: { age: 20, classe: 'I

// Dans map() pour ne garder que certains champs
const villes = [
  { ville: 'Fribourg', canton: 'FR', habitants: 38000, code: 21
  { ville: 'Bulle', canton: 'FR', habitants: 23000, code: 2022
];
const villesSimplifiees = villes.map(({ ville, canton }) => ({
// [{ ville: 'Fribourg', canton: 'FR' }, { ville: 'Bulle', cant
```

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 date et heure actuell

console.log(maintenant.toLocaleDateString()); // ex: "10.11.202
console.log(maintenant.toLocaleTimeString()); // ex: "15:23:42"

const jour = maintenant.getDate();
const mois = maintenant.getMonth() + 1; // Attention : janvier
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: "2025-11-10T13:23

// Cas d'usage réel : parser une date au format "dd.mm.yyyy"
const parseDateDDMMYYYY = (dateString) => {
```

```
  const [jour, mois, annee] = dateString.split('.').map(Number)
  return new Date(annee, mois - 1, jour);
};
const dateEval = parseDateDDMMYYYY("15.03.2025");

// Trouver la plus petite date dans un tableau
const evaluations = [
  { date: "15.03.2025", note: 5 },
  { date: "10.01.2025", note: 4 }
];
const plusPetiteDate = evaluations.reduce((min, e) => {
  const dateActuelle = parseDateDDMMYYYY(e.date);
  const dateMin = parseDateDDMMYYYY(min.date);
  return dateActuelle < dateMin ? e : min;
}, evaluations[0]);
```

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

Constante qui représente π (≈ **3.141592653589793**). Utile pour **cercles**, **angles** et **trigonométrie**.

```
// Périmètre d'un cercle (rayon r)
const rayon = 5;
const perimetre = 2 * Math.PI * rayon; // ≈ 31.4159

// Aire d'un cercle
const aire = Math.PI * rayon * rayon; // ≈ 78.5398
```

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

Renvoie la **valeur absolue** d'un nombre (toujours ≥ 0). Utile pour **distances**, **écarts**, **valeurs sans signe**. Renvoie `NaN` si l'argument n'est pas convertible en nombre.

```
Math.abs(-7);            // 7
Math.abs(0);             // 0
Math.abs(-3.14);         // 3.14

// Cas d'usage : calculer l'écart absolu
const cible = 10, actuel = 4;
const ecart = Math.abs(cible - actuel); // 6

// Appliquer sur un tableau
const temperatures = [-3, 0, 2, -5];
const absolues = temperatures.map(Math.abs); // [3, 0, 2, 5]
```

`Math.pow()` - élever à une puissance

Élève une **base** à une **puissance** : `Math.pow(base, exposant)` Équivalent moderne : `base ** exposant`.

```
Math.pow(2, 3);   // 8
2 ** 3;           // 8 (équivalent, syntaxe moderne)

Math.pow(5, 2);   // 25       // carré
Math.pow(9, 0.5); // 3        // racine carrée
Math.pow(27, 1/3);// 3        // racine cubique

// Cas d'usage : calcul de puissance
const prix = 1000;
const prixApresAugmentation = prix * Math.pow(1.03, 5); // +3%
```

`Math.min()` - plus petite valeur

Retourne la **plus petite** valeur parmi les arguments. Pour un **tableau**, utilise l'**opérateur spread**.

```
Math.min(3, -1, 7);       // -1

const nombres = [4, 1, 9, -2];
Math.min(...nombres);     // -2

// Cas d'usage réel : trouver le prix le plus bas
const motos = [
  { nom: 'Ninja', prix: 15000 },
  { nom: 'CBR', prix: 12000 }
];
const prixMin = Math.min(...motos.map(m => m.prix)); // 12000
```

`Math.max()` - plus grande valeur

Retourne la **plus grande** valeur parmi les arguments. Pour un **tableau**, utilise l'**opérateur spread**.

```
Math.max(3, -1, 7);      // 7

const nombres = [4, 1, 9, -2];
Math.max(...nombres);    // 9

// Cas d'usage réel : trouver le score maximum
const evaluations = [
  { eleve: 'Marie', note: 5.5 },
  { eleve: 'Jean', note: 6.0 }
];
const noteMax = Math.max(...evaluations.map(e => e.note)); // 6
```

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

Renvoie le **plus petit entier ≥ x** (arrondi vers le haut, vers **+∞**).

```
Math.ceil(2.01);   // 3
Math.ceil(3);      // 3
Math.ceil(-1.1);   // -1

// Cas d'usage courant : calcul de nombre de pages
const totalElements = 53, elementsParPage = 10;
const nombrePages = Math.ceil(totalElements / elementsParPage);

// Pagination
const obtenirNombrePages = (total, parPage) => Math.ceil(total
```

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

Renvoie le **plus grand entier ≤ x** (arrondi vers le bas, vers **-∞**).

```
Math.floor(2.99);  // 2
Math.floor(3);     // 3
Math.floor(-1.1);  // -2

// Cas d'usage : discrétiser un nombre en index
const valeur = 4.7;
const index = Math.floor(valeur); // 4

// Générer un nombre aléatoire entier
const nombreAleatoire = (min, max) =>
  Math.floor(Math.random() * (max - min + 1)) + min;
```

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

Arrondit au **plus proche entier** ; les valeurs en **.5** vont vers **+∞**.

```
Math.round(2.49); // 2
```

```
Math.round(2.5);  // 3
Math.round(-1.1); // -1
Math.round(-1.5); // -1
Math.round(-1.6); // -2

// Cas d'usage réel : arrondir au demi-point (notes suisses)
const arrondirAuDemi = (note) => Math.round(note * 2) / 2;
arrondirAuDemi(4.7);  // 4.5
arrondirAuDemi(4.8);  // 5.0
```

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

Retire la partie **décimale** (tronque), **vers 0**.

```
Math.trunc(3.9);    // 3
Math.trunc(-3.9);   // -3
Math.trunc(3);      // 3
Math.trunc('42.7'); // 42  (conversion implicite)

// Différence avec floor pour les négatifs
Math.trunc(-1.9);   // -1
Math.floor(-1.9);   // -2
```

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

Retourne la **racine carrée** de `x` (résultat ≥ 0). Si `x < 0` → `NaN`. Équivalent possible : `x ** 0.5`.

```
Math.sqrt(9);     // 3
Math.sqrt(2);     // 1.4142135623...
Math.sqrt(0);     // 0
Math.sqrt(-1);    // NaN
```

```
// Cas d'usage : distance euclidienne 2D (théorème de Pythagore
const deltaX = 6, deltaY = 8;
const distance = Math.sqrt(deltaX*deltaX + deltaY*deltaY); // 1
```

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

Renvoie un **flottant** `x` tel que `0 ≤ x < 1`. (Pas sécurisé pour la crypto — pour ça, utiliser `crypto.getRandomValues`.)

```
Math.random(); // ex: 0.7321...

// Cas d'usage : entier aléatoire dans [min, max] (inclus)
const entierAleatoire = (min, max) =>
  Math.floor(Math.random() * (max - min + 1)) + min;

entierAleatoire(1, 6); // simule un dé à 6 faces

// Tirer un élément au hasard d'un tableau
const prenomAleatoire = (prenoms) =>
  prenoms[Math.floor(Math.random() * prenoms.length)];

// Pile ou face (booléen)
const pileOuFace = () => Math.random() < 0.5;
```

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

> 🔗 **Interconnexions**

> - Avant `stringify`, prépare les données avec `map()`/`filter()` pour **nettoyer** ou **sélectionner**.
> - Avec des **dates** → pense au *replacer* pour masquer/simplifier.
> - Couplé à des **template literals** (backticks) pour générer des sorties lisibles.

Convertit une **valeur JS** en **chaîne JSON**. Syntaxe :
`JSON.stringify(value, replacer?, space?)`

```
// Basique
const utilisateur = { nom: 'Dupont', age: 28, ville: 'Fribourg'
const json = JSON.stringify(utilisateur);
// '{"nom":"Dupont","age":28,"ville":"Fribourg"}'

// Avec indentation (pretty-print) pour l'affichage
JSON.stringify(utilisateur, null, 2);
/*
{
  "nom": "Dupont",
  "age": 28,
  "ville": "Fribourg"
}
*/

// Filtrer des clés (replacer = tableau)
JSON.stringify(utilisateur, ['nom', 'ville']);
// '{"nom":"Dupont","ville":"Fribourg"}'

// Transformer des valeurs (replacer = fonction)
JSON.stringify({ motDePasse: 'secret123', prix: 3.14159 }, (cle
  cle === 'motDePasse' ? '***' : valeur
);
// '{"motDePasse":"***","prix":3.14159}'

// Cas d'usage réel : afficher des données dans un conteneur HT
const conteneur = document.getElementById("output");
```

```
conteneur.innerHTML = JSON.stringify(resultat, null, 3);
```

> ⚠️ **Points d'attention :** `undefined`, `function` et `Symbol` : **ignorés** dans les objets, deviennent `null` dans les tableaux. Les `Date` sont sérialisées au format **ISO** (ex. `"2025-11-10T13:23:42.123Z"`).

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

> 🔗 **Interconnexions**
>
> - Après `parse`, enchaîne `map()`/`filter()` pour transformer le tableau issu du JSON.
> - Avec un *reviver*, convertis directement les **dates ISO** en `Date`.
> - Valide la structure avec `every()`/`some()` ou `Array.isArray`.

Convertit une **chaîne JSON** en **valeur JS**. Syntaxe : `JSON.parse(text, reviver?)`

```
// Basique
const chaine = '{"nom":"Dupont","age":28,"tags":["js","fp"]}';
const donnees = JSON.parse(chaine);
donnees.nom; // "Dupont"
donnees.tags; // ["js","fp"]
```

```
// Avec "reviver" (ex : convertir les dates ISO en objets Date)
const chaineAvecDate = '{"cree":"2025-11-10T13:23:42.123Z"}';
const objet = JSON.parse(chaineAvecDate, (cle, valeur) =>
  (typeof valeur === 'string' && /^\d{4}-\d{2}-\d{2}T/.test(val
    ? new Date(valeur)
    : valeur
);
objet.cree instanceof Date; // true
```

```
// Gérer les erreurs de parsing (JSON invalide)
try {
  JSON.parse('pas du json valide');
} catch (erreur) {
  console.error('JSON invalide:', erreur.message);
}
```

> ⚠️ **Format JSON vs JS :** JSON ≠ JS : guillemets **doubles** obligatoires, **pas** de commentaires, ni `undefined/NaN/Infinity`.

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

> 🔗 **Interconnexions**
>
> - `trim()` + `split()` + `filter(Boolean)` pour tokeniser proprement.
> - Ensuite `map(Number)` pour convertir en nombres.
> - Utilise `join()` pour reconstruire une chaîne après transformation.

Sépare une **chaîne** en **tableau** selon un **séparateur** (texte ou **RegExp**). Argument optionnel `limit` = nombre max d'éléments.

```
"1,2,3".split(",");            // ["1","2","3"]
"a-b-c-d".split("-", 2);       // ["a","b"]  (limit à 2)
"1,2,3".split(",")[1];         // "2"  (2e élément)

// Cas d'usage : extraire l'année d'une date "dd.mm.yyyy"
"14.03.2025".split(".")[2];    // "2025"

// Lignes multiples
```

```
const texte = "ligne1\nligne2\nligne3";
texte.split("\n");            // ["ligne1","ligne2","ligne3"]

// Espaces multiples avec regex
"  chat   chien   loup  ".trim().split(/\s+/); // ["chat","chie

// Supprimer les éléments vides
"a,,b,".split(",").filter(Boolean);  // ["a","b"]

// Convertir une chaîne en tableau de caractères
"hello".split("");            // ["h","e","l","l","o"]

// Cas réel : parser CSV
const ligneCsv = "Index;Name;Cause;Dynasty";
const colonnes = ligneCsv.split(";").map(c => c.trim());
// ["Index", "Name", "Cause", "Dynasty"]
```

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

Suppriment les **espaces blancs** (espaces, tabulations, retours ligne) en **bordure** de chaîne.

- `trim()` : début **et** fin
- `trimStart()` : début uniquement *(alias historique : `trimLeft()`)*
- `trimEnd()` : fin uniquement *(alias historique : `trimRight()`)*

```
"  bonjour  ".trim();       // "bonjour"
"\n\tok \t".trimStart();    // "ok \t"
" terminé\n".trimEnd();     // " terminé"

// Cas d'usage : nettoyer un tableau de chaînes
const mots = ["  chat ", " chien", "loup  "];
const nettoyes = mots.map(s => s.trim()); // ["chat","chien","l
```

```
// Attention : n'enlève rien AU MILIEU
" a  b ".trim();            // "a  b"

// Cas réel : nettoyer les données CSV
const nom = (cellule || "").trim();
```

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

Complètent une **chaîne** jusqu'à une **longueur cible** en ajoutant des caractères au **début** (`padStart`) ou à la **fin** (`padEnd`). Syntaxe : `str.padStart(longueur, remplissage?)` / `str.padEnd(longueur, remplissage?)`

```
// Zéro-padding (numéro, date, etc.)
String(7).padStart(3, '0');      // "007"
String(3).padStart(2, '0');      // "03"

// Alignement à droite / gauche
"42".padStart(5);                // "   42"
"42".padEnd(5);                  // "42   "

// Masquer un identifiant (ex: numéro de carte)
"1234".padStart(16, '•');        // "••••••••••••1234"

// Colonnes de largeur fixe
const items = ["a", "bb", "ccc"];
const colonnes = items.map(s => s.padEnd(5, ' '));
// ["a    ","bb   ","ccc  "]

// Cas d'usage réel : formater des nombres avec zéros
const jour = 5;
const mois = 3;
const dateFormatee = `${String(jour).padStart(2, '0')}.${String
// "05.03.2025"
```

> Si la chaîne est déjà **≥ longueur cible**, elle est renvoyée **inchangée**.
> `remplissage` par défaut = espace.

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 !

// Afficher plusieurs valeurs
console.log('Utilisateur:', { nom: 'Dupont', age: 28 });

// Avec des variables
const total = 100;
console.log(`Le total est ${total} CHF`);
```

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

Affichent des messages avec un **niveau** (info, avertissement, erreur), **filtrables** dans les DevTools. `warn` met en évidence en **jaune**, `error` en **rouge** (avec stack).

```
console.info('Démarré en %d ms', 123);

const quota = 92;
console.warn('Quota élevé : %s%%', quota);

const statusHttp = 500, url = '/api/data';
console.error('Échec requête', { statusHttp, url });

// Bon à savoir : on peut logger des objets directement
```

```
console.info({ utilisateur: { id: 1, nom: 'Dupont' } });
```

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

Affiche des **données tabulaires** de façon lisible (tableaux, tableaux d'objets, objets).

```
// Tableau d'objets
const utilisateurs = [
  { id: 1, nom: 'Dupont', score: 42 },
  { id: 2, nom: 'Martin', score: 37 },
];
console.table(utilisateurs);

// Sélection de colonnes
console.table(utilisateurs, ['id', 'score']);

// Objet d'objets
const mesures = {
  a: { x: 1, y: 2 },
  b: { x: 3, y: 4 },
};
console.table(mesures);

// Tableau simple
console.table([10, 20, 30]); // colonnes: (index), value
```

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

> 🔗 **Interconnexions**
>
> - Mesure un pipeline `filter`→`map`→`reduce` pour comparer

> implémentations.
> - Sur de gros tableaux, compare un `sort` + `reduce` vs un `reduce` seul.

Crée un **chronomètre** nommé :

- `console.time(label)` démarre
- `console.timeLog(label, ...data?)` affiche le temps écoulé (sans arrêter)
- `console.timeEnd(label)` arrête et affiche le **total**

```
console.time('construction');
// ... code à mesurer ...
console.timeLog('construction', 'après étape 1');
// ... autre travail ...
console.timeEnd('construction'); // "construction: 123.45 ms"



// Label facultatif : "default"
console.time();
faireTravail();
console.timeEnd(); // "default: 4.12 ms"

// Cas d'usage réel : comparer deux approches
console.time('approche1');
const resultat1 = donnees.filter(x => x > 10).map(x => x * 2);
console.timeEnd('approche1');

console.time('approche2');
const resultat2 = donnees.reduce((acc, x) => {
  if (x > 10) acc.push(x * 2);
  return acc;
}, []);
console.timeEnd('approche2');
```

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** du tableau (dans l'ordre). Ne retourne rien (**`undefined`**), sert aux **effets de bord**.

```
const prenoms = ['Alice', 'Bob', 'Claire'];

// (valeur, index, tableau)
prenoms.forEach((prenom, index) => {
  console.log(index, prenom);
});
// 0 'Alice'
// 1 'Bob'
// 2 'Claire'

// Exemple : somme (mais préférer reduce pour ça)
let somme = 0;
[1, 2, 3].forEach(n => { somme += n; }); // 6

// ⚠️ forEach ne retourne rien, ne pas l'utiliser pour transfor
// Préférer map() si on veut un nouveau tableau
```

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

Renvoie une **suite de paires** `[index, valeur]` pour chaque élément du tableau. Utile pour **boucler** en ayant la **position** (0, 1, 2, ...) **et** la **valeur**.

```
const fruits = ['pomme', 'banane', 'orange'];

// Parcours avec position + valeur
```

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

// Sans "déstructuration"
for (const paire of fruits.entries()) {
  const index = paire[0];
  const fruit = paire[1];
  console.log(index, fruit);
}

// Obtenir un "vrai" tableau de paires
[...fruits.entries()]; // [[0,'pomme'], [1,'banane'], [2,'orang
```

`in` - parcourir les clés d'un tableau

Parcourt les **clés** d'un tableau, en tant que **chaînes**.

```
const couleurs = ['rouge', 'vert', 'bleu'];
for (const index in couleurs) {
  console.log(index, couleurs[index]);
  // "0" 'rouge' / "1" 'vert' / "2" 'bleu'
}

// ⚠️ Attention : parcourt aussi les propriétés ajoutées
couleurs.extra = 'jaune';
for (const cle in couleurs) console.log(cle);
// "0","1","2","extra"

// Trous ignorés
const sparse = [];
sparse[2] = 'x';
for (const i in sparse) console.log(i); // "2"
```

`for...of` - parcourir les valeurs d'un tableau

Boucle directement sur les **valeurs** du tableau (dans l'ordre).

```
const animaux = ['chat', 'chien', 'lapin'];
for (const animal of animaux) {
  console.log(animal); // 'chat', 'chien', 'lapin'
}

// Besoin aussi de l'index ? Utilise entries()
for (const [index, animal] of animaux.entries()) {
  console.log(index, animal);
  // 0 'chat' / 1 'chien' / 2 'lapin'
}

// Fonctionne aussi avec les chaînes, Set, Map (leurs valeurs)
for (const caractere of "hey") console.log(caractere); // 'h','
```

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

Retourne la **première valeur** du tableau pour laquelle le **test** renvoie `true`, sinon `undefined`. Signature : `arr.find((valeur, index, tableau) => test, thisArg?)`

```
[5, 12, 8, 130, 44].find(n => n > 10); // 12

[1, 2, 3].find(n => n > 5);            // undefined

// Cas d'usage réel : trouver un utilisateur par nom
const utilisateurs = [
  { id: 1, nom: 'Dupont' },
  { id: 2, nom: 'Martin' },
];
utilisateurs.find(u => u.nom === 'Dupont'); // { id: 1, nom: 'D
```

```
// Cas pratique : trouver une marque
const motos = [
  { marque: 'Honda', modeles: [...] },
  { marque: 'Kawasaki', modeles: [...] }
];
const moto = motos.find(m => m.marque === 'Honda');
```

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

Retourne l'**index** du **premier** élément pour lequel le test renvoie `true`, sinon **–1**. Signature : `arr.findIndex((valeur, index, tableau) => test, thisArg?)`

```
[5, 12, 8, 130, 44].findIndex(n => n > 10); // 1 (car 12)

// Sur objets
const utilisateurs = [{id:1},{id:3},{id:5}];
utilisateurs.findIndex(u => u.id === 3); // 1
utilisateurs.findIndex(u => u.id === 2); // -1

// Cas d'usage : supprimer le premier élément correspondant
const index = utilisateurs.findIndex(u => u.id === 3);
if (index !== -1) utilisateurs.splice(index, 1);
```

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

Renvoient l'**index** de la **première** (`indexOf`) ou **dernière** (`lastIndexOf`) occurrence d'une valeur, sinon **–1**. Signatures : `arr.indexOf(valeur, fromIndex = 0)` `arr.lastIndexOf(valeur, fromIndex = arr.length - 1)`

```
[1, 2, 3, 2].indexOf(2);          // 1
[1, 2, 3, 2].indexOf(2, 2);       // 3 (à partir de l'index 2)

[1, 2, 3, 2].lastIndexOf(2);      // 3
['a','b','a'].lastIndexOf('a', 1);// 0 (en partant de 1 vers la

['x','y'].indexOf('z');           // -1

// Comparaison stricte (===) et références d'objets
const obj = { x: 1 };
[obj, { x: 1 }].indexOf(obj);         // 0
[obj, { x: 1 }].indexOf({ x: 1 });    // -1 (objets différents)

// Cas NaN (attention!)
[NaN].indexOf(NaN);                   // -1 (indexOf ne trouve
[NaN].includes(NaN);                  // true (includes fonctic
```

`push()`, `pop()`, `shift()` et `unshift()` - ajouter/supprimer au début/fin d'un tableau

⚠️ **Ces méthodes mutent** le tableau :

- `push(x, ...)` : **ajoute à la fin**, retourne la **nouvelle longueur**
- `pop()` : **retire la fin**, retourne l'**élément retiré**
- `unshift(x, ...)` : **ajoute au début**, retourne la **nouvelle longueur**
- `shift()` : **retire le début**, retourne l'**élément retiré**

```
const nombres = [1, 2];

// Fin du tableau
nombres.push(3);   // 3   (nombres → [1,2,3])
nombres.pop();     // 3   (nombres → [1,2])

// Début du tableau
nombres.unshift(0); // 3  (nombres → [0,1,2])
```

```
nombres.shift();    // 0  (nombres → [1,2])

// ⚠️ En programmation fonctionnelle, préférer des copies :
const original = [1, 2, 3];
const avecNouvel = [...original, 4];           // [1,2,3,4] (or
const sansDernier = original.slice(0, -1);     // [1,2] (origir
```

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

Retourne une **copie** d'une portion du tableau, **sans le modifier**. `arr.slice(début, fin?)` — `début` inclus, `fin` **exclu**. Indices négatifs = à partir de la fin.

```
const nombres = [0, 1, 2, 3, 4];

nombres.slice(1, 4);   // [1, 2, 3]
nombres.slice(2);      // [2, 3, 4]
nombres.slice(-2);     // [3, 4]

// Cloner un tableau
const copie = nombres.slice(); // [0,1,2,3,4]

// Cas d'usage : pagination simple
const paginer = (elements, page, parPage) =>
  elements.slice((page-1)*parPage, page*parPage);

const produits = [1,2,3,4,5,6,7,8,9,10];
paginer(produits, 2, 3); // [4,5,6] (page 2, 3 éléments par pag
```

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

⚠️ **Modifie** le tableau en place. Signature : `arr.splice(début,`

`nbSuppression, ...élémentsÀInsérer)` Retourne un **tableau des éléments supprimés**.

```
// Supprimer
const a = [0,1,2,3,4];
const supprimes = a.splice(2, 1);   // supprime 1 élément à l'i
// a → [0,1,3,4], supprimes → [2]

// Insérer (sans supprimer)
const b = [0,1,2,3];
b.splice(2, 0, 'x', 'y');       // insère à l'index 2
// b → [0,1,'x','y',2,3]

// Remplacer
const c = [0,1,2,3];
c.splice(1, 2, 'A');            // remplace 1 et 2 par 'A'
// c → [0,'A',3]

// Indices négatifs = depuis la fin
const d = [10,20,30,40];
d.splice(-2, 1);                // supprime 30
// d → [10,20,40]

// Vider un tableau
const e = [1,2,3];
e.splice(0);                    // e → []
```

`concat()` - joindre deux tableaux

Retourne un **nouveau tableau** en **chaînant** des tableaux et/ou valeurs. Ne **modifie pas** les originaux (équiv. courant : l'opérateur **spread**).

```
const a = [1, 2];
const b = [3, 4];
```

```
a.concat(b);            // [1, 2, 3, 4]
a.concat(0, b, 5);      // [1, 2, 0, 3, 4, 5]
[].concat(a, b);        // [1, 2, 3, 4] (clonage simple)

const imbrique = [1, [2, 3]];
imbrique.concat([4, [5]]); // [1, [2,3], 4, [5]]  (pas d'aplati

// Alternative spread (moderne et plus lisible)
[...a, ...b];           // [1, 2, 3, 4]

// Cas d'usage : combiner des résultats
const motosHonda = [{ marque: 'Honda', nom: 'CBR' }];
const motosKawasaki = [{ marque: 'Kawasaki', nom: 'Ninja' }];
const toutesLesMotos = motosHonda.concat(motosKawasaki);
```

`join()` - joindre des chaînes de caractères

Rassemble les **éléments d'un tableau** en **une seule chaîne**. Syntaxe : `arr.join(séparateur?)` — séparateur par défaut **","**.

```
['a','b','c'].join();         // "a,b,c"
['a','b','c'].join(' - ');    // "a - b - c"
[1,2,3].join('');             // "123" (sans séparateur)
['usr','local','bin'].join('/'); // "usr/local/bin"

// Valeurs spéciales et "trous"
[null, undefined, ''].join('|'); // "||"
[1, , 3].join('-');              // "1--3"

// Cas d'usage réel : formater des noms
const prenoms = ['Marie', 'Jean', 'Pierre'];
const phrase = prenoms.join(', '); // "Marie, Jean, Pierre"

// Créer un chemin
const chemin = ['dossier', 'sous-dossier', 'fichier.txt'].join(
// "dossier/sous-dossier/fichier.txt"
```

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

`Object.keys(obj)` renvoie un **tableau des clés** propres et énumérables.
`Object.values(obj)` renvoie le **tableau des valeurs** correspondantes.
(Pas les propriétés du **prototype**.)

```
const utilisateur = { id: 1, nom: 'Dupont', actif: true };

Object.keys(utilisateur);   // ["id", "nom", "actif"]
Object.values(utilisateur); // [1, "Dupont", true]

// Parcourir clés + valeurs
Object.keys(utilisateur).forEach(cle => {
  console.log(cle, utilisateur[cle]);
});

// Avec un tableau (les indices deviennent des chaînes)
Object.keys(['a','b']);   // ["0","1"]
Object.values(['a','b']); // ["a","b"]

// Cas d'usage réel : transformer un objet de comptage en table
const comptageParBranche = { Maths: 5, Français: 3, Anglais: 4
const listeParBranche = Object.entries(comptageParBranche)
  .map(([branche, nombre]) => ({ branche, nombre }));
// [{ branche: 'Maths', nombre: 5 }, ...]
```

> Astuce : pour des **couples [clé, valeur]**, utilise aussi `Object.entries(obj)`.

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

Renvoie **true/false** si le tableau **contient** la valeur. Syntaxe : `arr.includes(valeur, fromIndex = 0)` (indices négatifs acceptés).

```
[1, 2, 3].includes(2);        // true
[1, 2, 3].includes(4);        // false
['a','b','c'].includes('a',1);// false (cherche dès l'index 1)
['a','b','c'].includes('c',-1);// true (depuis la fin)

// Cas spéciaux
[NaN].includes(NaN);          // true (contrairement à indexOf)
const obj = {x:1};
[obj].includes(obj);          // true
[{x:1}].includes({x:1});      // false (références différentes)

// Cas d'usage réel : vérifier des droits
const droitsUtilisateur = ['READ', 'WRITE'];
const peutEcrire = droitsUtilisateur.includes('WRITE'); // true

// Filtrer selon une liste autorisée
const extensionsAutorisees = ['jpg', 'png', 'gif'];
const fichiersValides = fichiers.filter(f =>
  extensionsAutorisees.includes(f.extension)
);
```

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

- `every(test)` → **true si tous** les éléments passent le test.
- `some(test)` → **true si au moins un** élément passe le test. S'arrêtent **dès que le résultat est connu** (optimisation).

```
const nombres = [2, 4, 6];

nombres.every(n => n % 2 === 0);  // true  (tous pairs)
nombres.some(n => n < 0);         // false (aucun négatif)

// Cas d'usage : valider un formulaire (tous non vides?)
const champRequis = ['nom','email','motDePasse'];
```

```
const formulaireValide = champRequis.every(champ =>
  formulaire[champ]?.trim()
);

// Vérifier si une extension est autorisée
const extensionsAutorisees = ['jpg','png','gif'];
const estAutorisee = extensionsAutorisees.some(ext =>
  nomFichier.endsWith('.' + ext)
);

// Sous-ensemble : vérifier que toutes les valeurs requises sor
const permissionsRequises = ['READ','WRITE'];
const aLesPermissions = permissionsRequises.every(p =>
  droitsUtilisateur.includes(p)
);

// Cas pratique : vérifier qu'une marque a AU MOINS un modèle d
const marquesAvec2021 = motos.filter(marque =>
  marque.modeles.some(modele => modele.annee === 2021)
);
```

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

Remplit **en place** un tableau avec une valeur. `arr.fill(valeur, début = 0, fin = arr.length)` — `début` inclus, `fin` **exclu** (indices négatifs ok).

```
// Basique : créer un tableau rempli
Array(5).fill(0);                 // [0,0,0,0,0]
Array(3).fill('?');               // ['?','?','?']

// Plage ciblée
const a = [1,2,3,4];
a.fill(9, 1, 3);                  // [1,9,9,4]

// Depuis la fin
[1,2,3,4].fill(7, -2);            // [1,2,7,7]
```

```
// Retourne le même tableau (mutation)
const b = [1,2,3];
const resultat = b.fill(8);       // b === resultat → true

// Cas d'usage : initialiser un tableau de notes à 0
const notes = Array(10).fill(0);
```

`flat()` - aplatir un tableau

Crée un **nouveau tableau** en aplatissant les **sous-tableaux** jusqu'à une **profondeur** donnée. Syntaxe : `arr.flat(profondeur = 1)`

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

// Les "trous" sont supprimés
[1, , [2, , 3]].flat();            // [1, 2, 3]

// Cas d'usage : aplatir des résultats imbriqués
const marquesAvecModeles = [
  { marque: 'Honda', modeles: ['CBR', 'CRF'] },
  { marque: 'Kawasaki', modeles: ['Ninja', 'Z'] }
];
const tousLesModeles = marquesAvecModeles
  .map(m => m.modeles)
  .flat();
// ['CBR', 'CRF', 'Ninja', 'Z']
```

`sort()` - pour trier un tableau

> 🔗 **Interconnexions**
>
> - Trie des résultats **après** `filter()`/`map()` pour éviter du travail

> inutile.
> - Pour les **accents**/locales : `localeCompare('fr')`.
> - Évite de muter : `arr.slice().sort(...)` si tu veux préserver l'original.

⚠️ **Trie en place** (modifie l'original). Par défaut : **ordre lexicographique** (comme des chaînes). Pour un tri **numérique** ou **personnalisé**, fournis une fonction de comparaison.

```
// Par défaut (lexicographique) - ATTENTION !
[3, 20, 100, 4].sort();                  // [100, 20, 3, 4] ⚠️

// Tri numérique (croissant / décroissant)
[3, 20, 100, 4].sort((a, b) => a - b); // [3, 4, 20, 100] ✓
[3, 20, 100, 4].sort((a, b) => b - a); // [100, 20, 4, 3] ✓

// Objets : trier par propriété
const utilisateurs = [{age: 30},{age: 20},{age: 25}];
utilisateurs.sort((a, b) => a.age - b.age);
// [{20},{25},{30}]

// Chaînes avec accents (locale française)
['é', 'e', 'è', 'a'].sort((a,b) => a.localeCompare(b, 'fr'));
// ['a', 'e', 'é', 'è']

// Cas pratique : trier sans muter l'original
const notesOriginales = [5, 2, 8, 1];
const notesTriees = [...notesOriginales].sort((a, b) => a - b);
// notesOriginales reste [5, 2, 8, 1]

// Tri complexe : par canton puis par population
const villes = [
  { nom: 'Fribourg', canton: 'FR', pop: 38000 },
  { nom: 'Lausanne', canton: 'VD', pop: 140000 },
  { nom: 'Bulle', canton: 'FR', pop: 23000 }
];
```

```
villes.sort((a, b) =>
  a.canton.localeCompare(b.canton) || b.pop - a.pop
);
```

> 🔗 **Interconnexions**
>
> - Chaîne typique : `arr.filter(...).map(...).reduce(...)`.
> - Préfère des **fonctions pures** qui **ne mutent pas** les objets → voir *Immuabilité* et *Fonctions pures*.
> - Avec `flatMap()` si chaque élément peut produire 0..n résultats.

Applique une **fonction** à chaque élément et **retourne un nouveau tableau**. Ne **modifie pas** l'original. Longueur **inchangée**. Signature : `arr.map((valeur, index, tableau) => nouvelleValeur, thisArg?)`

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

// Extraire une propriété
const utilisateurs = [{nom:'Dupont'}, {nom:'Martin'}];
utilisateurs.map(u => u.nom);        // ["Dupont", "Martin"]

// Utiliser l'index
['a','b','c'].map((valeur, index) => `${index}:${valeur}`);
// ["0:a","1:b","2:c"]

// Conversion pratique
['1','2','3'].map(Number);           // [1, 2, 3]

// Cas pratique : formater "Prénom NOM"
const eleves = [
  { prenom: 'Marie', nom: 'Dupont' },
  { prenom: 'Jean', nom: 'Martin' }
];
const nomComplets = eleves.map(e => `${e.prenom} ${e.nom.toUppe
// ["Marie DUPONT", "Jean MARTIN"]
```

```
// Ne garder que certains champs (déstructuration)
const villes = [
  { ville: 'Fribourg', canton: 'FR', habitants: 38000 },
  { ville: 'Bulle', canton: 'FR', habitants: 23000 }
];
const simplifies = villes.map(({ ville, canton }) => ({ ville,
// [{ ville: 'Fribourg', canton: 'FR' }, ...]
```

Approfondissement de `map()`

La fonction `map()` applique une fonction à chaque élément d'un tableau et renvoie un **nouveau tableau**. Elle ne modifie pas le tableau d'origine et conserve la même longueur. Pour chaque élément, `map()` appelle la fonction de rappel avec la valeur, son index et le tableau; la valeur retournée est insérée dans le nouveau tableau. Cette opération est pure et prévisible : mêmes entrées ⇒ mêmes sorties.

Par exemple, transformer un tableau de nombres en doublant chaque élément :

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

⚠️ **Attention à l'immuabilité !** Veillez à ne pas utiliser `map()` pour **modifier** directement les éléments originaux. Les diapositives soulignent qu'une fonction passée à `map()` doit retourner un nouvel objet plutôt que muter l'existant. Si vous devez enrichir des objets, créez-en une copie :

```
// ❌ MAUVAISE pratique : la fonction modifie l'objet
eleves.map(eleve => {
  eleve.moyenne = calculerMoyenne(eleve);
  return eleve;
```

```
});

// ✅ BONNE pratique : retourner un nouvel objet sans muter l'o
eleves.map(eleve => ({
  ...eleve,
  moyenne: calculerMoyenne(eleve)
}));
```

Cas d'usage réels tirés des exercices :

```
// Extraire les marques de motos
const marques = motos.map(m => m.marque);

// Pour chaque marque, extraire la liste des noms de modèles
const marquesAvecNoms = motos.map(({ marque, modeles }) => ({
  marque,
  modeles: modeles.map(({ nom }) => nom)
}));

// Calculer le prix moyen par marque
const prixMoyensParMarque = motos.map(({ marque, modeles }) =>
  const total = modeles.reduce((somme, m) => somme + m.prix, 0)
  return {
    marque,
    prixMoyen: (total / modeles.length).toFixed(2)
  };
});
```

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

> 🔗 **Interconnexions**
>
> - Combine avec `map()` puis `reduce()` pour passer du **bruit** → **signal** → **agrégat**.

> - Pour enlever les doublons avant filtrage : `new Set()` puis `Array.from(...)`.
> - Tester des propriétés dérivées ? D'abord `map()` pour calculer, puis `filter()` sur le résultat.

Garde **uniquement** les éléments pour lesquels le **test** retourne `true`. Retourne un **nouveau tableau**, ne **modifie pas** l'original. Signature : `arr.filter((valeur, index, tableau) => condition, thisArg?)`

```
// Garder les nombres > 10
[5, 12, 8, 130, 44].filter(n => n > 10); // [12, 130, 44]

// Filtrer des objets (ex : actifs seulement)
const utilisateurs = [
  {actif:true, nom:'Dupont'},
  {actif:false, nom:'Martin'},
  {actif:true, nom:'Bernard'}
];
utilisateurs.filter(u => u.actif);
// [{actif:true, nom:'Dupont'},{actif:true, nom:'Bernard'}]

// Retirer les valeurs "falsy" (0, '', null, undefined, NaN, fa
['a', '', null, 'b', 0].filter(Boolean); // ["a", "b"]

// Cas pratique : villes du canton FR avec habitants entre 10k
const villesFR = villes.filter(v =>
  v.canton === 'FR' && v.habitants >= 10000 && v.habitants < 25
);

// Marques ayant AU MOINS un modèle sorti en 2021
const marquesAvec2021 = motos.filter(marque =>
  marque.modeles.some(modele => modele.annee === 2021)
);

// Évaluations de maths avec note >= 5.0
const bonnesMaths = evaluations.filter(e =>
```

```
  e.branche === 'Maths' && e.note >= 5.0
);
```

Approfondissement de `filter()`

La fonction `filter()` sélectionne les éléments d'un tableau qui satisfont un test et renvoie un **nouveau tableau** contenant uniquement ces éléments. Elle exécute la fonction de test pour chaque élément et n'ajoute celui-ci que si le test renvoie `true`. La fonction `filter()` retourne une **copie superficielle** du tableau original, ce qui évite de le modifier.

Exemple pour filtrer des salaires supérieurs à 75'000 CHF :

```
const salaires = [65000, 80000, 55000, 90000];
const salairesEleves = salaires.filter(s => s > 75000);
// [80000, 90000]
```

`filter()` est particulièrement utile lorsqu'elle est combinée avec `map()` et `reduce()` pour former des pipelines de transformation. Veillez à garder la fonction de test pure et à éviter d'altérer les éléments dans le callback.

Cas d'usage réels tirés des exercices :

```
// Filtrer les marques dont le nom commence par 'H'
const marquesH = motos.filter(({ marque }) => marque.startsWith

// Évaluations d'avril avec note < 3.0
const evalAvril = evaluations.filter(({ date, note }) => {
  const mois = parseInt(date.split(".")[1], 10);
  return mois === 4 && note < 3.0;
});

// Villes entre 500 et 1000 habitants (bornes incluses)
const petitesVilles = villes.filter(v =>
```

```
  v.habitants >= 500 && v.habitants <= 1000
);
```

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

Groupe les éléments **par clé** calculée à partir d'une fonction. (En environnements modernes : `array.group(fn)` / `array.groupToMap(fn)`. Sinon, utilise `reduce`.)

```
// Exemple : grouper par ville
const personnes = [
  { nom: 'Ana',  ville: 'Fribourg'  },
  { nom: 'Ben',  ville: 'Fribourg'  },
  { nom: 'Eli',  ville: 'Lausanne' },
];

// Si disponible (nouvelle API)
personnes.group?.(p => p.ville);
// { Fribourg: [ {Ana}, {Ben} ], Lausanne: [ {Eli} ] }

personnes.groupToMap?.(p => p.ville);
// Map(2) { 'Fribourg' => [ {Ana}, {Ben} ], 'Lausanne' => [ {El

// Variante compatible partout (reduce) – LA PLUS UTILISÉE
const parVille = personnes.reduce((acc, p) => {
  (acc[p.ville] ??= []).push(p);
  return acc;
}, {});
// { Fribourg: [ ... ], Lausanne: [ ... ] }

// Autre exemple : pair / impair
[1,2,3,4,5].reduce((acc, n) => {
  const cle = n % 2 ? 'impair' : 'pair';
  (acc[cle] ??= []).push(n);
  return acc;
```

```
}, {});
// { impair: [1,3,5], pair: [2,4] }

// Cas pratique : regrouper empereurs par dynastie
const parDynastie = empereurs.reduce((acc, empereur) => {
  (acc[empereur.dynasty] ??= []).push(empereur);
  return acc;
}, {});
```

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

Fait un **map** puis aplati d'**un niveau** en **une seule passe**. Signature :
`arr.flatMap((valeur, index, tableau) => valeurOuTableau)`

```
// 1) Décomposer puis aplatir
['a b', 'c d'].flatMap(s => s.split(' '));
// ["a","b","c","d"]

// 2) Transformer en multipliant les éléments
[1, 2, 3].flatMap(n => [n, n * 2]);
// [1, 2, 2, 4, 3, 6]

// 3) Filtrer en retournant [] pour exclure
[1, 2, 3, 4].flatMap(n => n % 2 ? [n] : []);
// [1, 3]

// Cas pratique : extraire tous les produits de toutes les vent
const ventes = [
  { client: 'A', produits: [{nom:'P1'}, {nom:'P2'}] },
  { client: 'B', produits: [{nom:'P3'}] }
];
const tousProduits = ventes.flatMap(v => v.produits);
// [{nom:'P1'}, {nom:'P2'}, {nom:'P3'}]

// Créer "MARQUE / Modèle" pour toutes les motos
const liste = motos.flatMap(({ marque, modeles }) =>
```

```
  modeles.map(({ nom }) => `${marque.toUpperCase()} / ${nom}`)
);
// ['HONDA / CBR', 'HONDA / CRF', 'KAWASAKI / Ninja', ...]
```

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

> 🔗 **Interconnexions**
>
> - Utilise `reduce()` pour implémenter `pipe()`/`compose()` (composition de fonctions).
> - `reduceRight()` pour traiter de droite → gauche (ex. *compose*).
> - Si `reduce()` devient trop dense, préfère des étapes `map()`/`filter()` **lisibles**.

Appliquent une **fonction d'accumulation** pour produire **une valeur unique**.

- `reduce` : de **gauche → droite**
- `reduceRight` : de **droite → gauche**

Signature : `arr.reduce((acc, valeur, index, tableau) => nouvelAcc, init?)`

```
// Somme
[1, 2, 3, 4].reduce((acc, n) => acc + n, 0); // 10

// Produit
[1, 2, 3, 4].reduce((acc, n) => acc * n, 1); // 24

// Maximum
[5, 1, 9, 3].reduce((max, n) => n > max ? n : max, -Infinity);

// Comptage par valeur (fréquences)
['a','b','a','c','b','a'].reduce((map, cle) => {
```

```
  map[cle] = (map[cle] || 0) + 1;
  return map;
}, {});
// { a:3, b:2, c:1 }

// Moyenne (calcul somme + nombre)
const stats = evaluations.reduce(
  (acc, { note }) => {
    acc.somme += note;
    acc.nombre += 1;
    return acc;
  },
  { somme: 0, nombre: 0 }
);
const moyenne = stats.nombre ? stats.somme / stats.nombre : 0;

// Total des habitants par canton
const totauxParCanton = villes.reduce((acc, { canton, habitants
  acc[canton] = (acc[canton] || 0) + habitants;
  return acc;
}, {});

// reduceRight : sens inverse (droite → gauche)
['a','b','c'].reduceRight((acc, v) => acc + v, ''); // "cba"
```

Approfondissement de `reduce()`

`reduce()` applique une fonction d'accumulation sur les éléments d'un tableau pour produire une **valeur unique**. La fonction d'accumulation reçoit l'accumulateur et la valeur courante et doit retourner un nouvel accumulateur. `reduce()` peut parcourir le tableau de gauche à droite (`reduce()`) ou de droite à gauche (`reduceRight()`). Ce mécanisme est utile pour calculer des sommes, des maximums, des statistiques ou regrouper des valeurs.

Par exemple, pour sommer les notes d'une classe :

```
const notes = [5, 6, 4, 5];
const total = notes.reduce((acc, note) => acc + note, 0); // 20
```

`reduce()` est puissant mais peut devenir illisible si l'accumulateur réalise des opérations complexes ou modifie l'état. Les diapositives conseillent de privilégier des fonctions dédiées (`groupBy()`, `filter()`, `map()`) lorsque c'est plus lisible.

Cas d'usage réels tirés des exercices :

```
// Pour chaque marque, trouver le modèle le plus puissant
const plusPuissantsParMarque = motos.reduce((acc, { marque, mod
  const lePlusPuissant = modeles.reduce(
    (max, m) => (m.puissance > max.puissance ? m : max),
    modeles[0]
  );
  acc[marque] = lePlusPuissant;
  return acc;
}, {});

// Total des habitants de toute la Suisse
const totalHabitants = villes.reduce(
  (somme, { habitants }) => somme + habitants,
  0
);

// Total par canton + total suisse (avec clé 'CH')
const totaux = villes.reduce((acc, { canton, habitants }) => {
  acc.CH = (acc.CH || 0) + habitants;
  acc[canton] = (acc[canton] || 0) + habitants;
  return acc;
}, {});
```

```
// Liste des cantons sans doublons (ordre d'apparition)
const cantonsUniques = villes.reduce((acc, { canton }) => {
  if (!acc.includes(canton)) acc.push(canton);
  return acc;
}, []);

// Trouver la plus petite date
const plusPetiteDate = evaluations.reduce((min, e) => {
  return parseDate(e.date) < parseDate(min.date) ? e : min;
}, evaluations[0]);

// Compter le nombre d'évaluations par branche
const comptageParBranche = evaluations.reduce((acc, e) => {
  acc[e.branche] = (acc[e.branche] || 0) + 1;
  return acc;
}, {});

// Statistiques complètes par branche (min, max, total, moyenne
const statParBranche = evaluations.reduce((acc, e) => {
  const branche = e.branche, note = e.note;
  if (!acc[branche]) {
    acc[branche] = {
      nombre: 0,
      total: 0,
      min: Infinity,
      max: -Infinity
    };
  }
  acc[branche].nombre += 1;
  acc[branche].total += note;
  if (note < acc[branche].min) acc[branche].min = note;
  if (note > acc[branche].max) acc[branche].max = note;
  return acc;
}, {});
// Puis calculer les moyennes
Object.values(statParBranche).forEach(s => {
  s.moyenne = (s.total / s.nombre).toFixed(2);
});
```

Combiner `map()`, `filter()` et `reduce()`

Ces trois fonctions se combinent pour transformer, filtrer et agréger des données en un pipeline clair. Par exemple, pour filtrer des produits en stock, appliquer une TVA et calculer le total :

```
const produits = [
  { nom: 'A', prix: 20, enStock: true },
  { nom: 'B', prix: 15, enStock: false },
  { nom: 'C', prix: 30, enStock: true }
];
const tauxTVA = 0.077;

const totalTTC = produits
  .filter(p => p.enStock)
  .map(p => p.prix * (1 + tauxTVA))
  .reduce((acc, prix) => acc + prix, 0);
```

Ce chaînage est plus lisible que l'utilisation d'une boucle impérative et correspond à l'esprit déclaratif de la programmation fonctionnelle.

Exemple complexe tiré des exercices : prix moyen des motos de chaque marque

```
const prixMoyensParMarque = motos.map(({ marque, modeles }) =>
  const total = modeles.reduce((somme, m) => somme + m.prix, 0)
  return {
    marque,
    prixMoyen: (total / modeles.length).toFixed(2)
  };
});
```

Exemple : moyennes d'un élève par branche

```
const moyennesClaire = evaluations
  .filter(e =>
    e.nom.toUpperCase() === "VOYANTE" &&
    e.prenom.toUpperCase() === "CLAIRE"
  )
  .reduce((acc, e) => {
    if (!acc[e.branche]) {
      acc[e.branche] = { total: 0, nombre: 0 };
    }
    acc[e.branche].total += e.note;
    acc[e.branche].nombre += 1;
    return acc;
  }, {});

const moyennes = Object.fromEntries(
  Object.entries(moyennesClaire).map(([branche, { total, nombre
    [branche, parseFloat((total / nombre).toFixed(2))]
  )
);
```

Composition de fonctions et pipe

La **composition de fonctions** consiste à combiner plusieurs fonctions unaires pour obtenir un traitement plus complexe. Les **fonctions unaires** n'ont qu'un seul argument, ce qui simplifie la composition et l'utilisation avec `map()`, `filter()` ou `reduce()`.

Le **currying** transforme une fonction prenant plusieurs arguments en une chaîne de fonctions prenant chacune un argument. Cela permet de fixer certains paramètres et de réutiliser la nouvelle fonction :

```
// Fonction currifiée simple
const ajouter = a => b => a + b;
const ajouter10 = ajouter(10);
const resultats = [1, 2, 3].map(ajouter10); // [11, 12, 13]
```

```
// Cas pratique : calculer une note fédérale (currification)
const calculerNote = pointsMax => points => {
  const ratioSur5 = (points / pointsMax) * 5;
  const noteAvecBase = ratioSur5 + 1;
  const noteArrondie = Math.round(noteAvecBase * 2) / 2;
  return Math.max(1, Math.min(noteArrondie, 6)); // borner entr
};

const versNote = calculerNote(60); // fixe pointsMax à 60
const examensAvecNotes = examens.map(ex => ({
  ...ex,
  note: versNote(ex.points)
}));
```

Une **closure** est une fonction qui retient une référence à son environnement lexical, permettant de créer des fonctions avec mémoire.

```
// Générateur de filtres par longueur minimale (closure)
const creerFiltreParLongueur = longueurMin => liste =>
  liste.filter(mot => mot.length > longueurMin);

const filtreCourts = creerFiltreParLongueur(3);
const filtreLongs = creerFiltreParLongueur(6);

const mots = ['chat', 'chien', 'loup', 'hippopotame', 'rat'];
filtreCourts(mots);  // ['chat', 'chien', 'loup', 'hippopotame'
filtreLongs(mots);   // ['hippopotame']
```

La **fonction pipe** exécute des fonctions en chaîne de gauche à droite. Chaque fonction reçoit en entrée le résultat de la précédente. On l'implémente généralement avec `reduce()`:

```
// Implémentation de pipe
const pipe = (...fonctions) => valeurInitiale =>
```

```
  fonctions.reduce((acc, fn) => fn(acc), valeurInitiale);

// Exemple : calculer une note fédérale avec pipe
const ratioSurTotal = pts => pts / 60;
const fois5 = x => x * 5;
const plus1 = x => x + 1;
const demiPoint = x => Math.round(x * 2) / 2;

const calculerNoteFederale = pipe(ratioSurTotal, fois5, plus1,

const notes = [45, 30, 55].map(calculerNoteFederale);
// [4.5, 3.5, 5.5]

// Exemple : traducteur loufoque
const enMajuscules = s => s.toUpperCase();
const voyellesEnI = s => s.replace(/[AEIOUYÀÂÄÉÈÊËÎÏÔÖÛÜŸ]/g, '
const doubleConsonnes = s => s.replace(/([BCDFGHJKLMNPQRSTVWXZ]
const ajouteEmoji = s => s + ' 😎';

const traduire = pipe(enMajuscules, voyellesEnI, doubleConsonne
traduire("Bonjour");  // "BBINNJJIR 😎"
```

La **fonction compose** fonctionne comme pipe mais exécute les fonctions de droite à gauche. Ces deux patterns améliorent la lisibilité et la testabilité du code.

```
const compose = (...fonctions) => valeurInitiale =>
  fonctions.reduceRight((acc, fn) => fn(acc), valeurInitiale);
```

Cas d'usage réel : moteur de recherche d'élèves

```
const normaliser = s =>
  s.normalize('NFD')
   .replace(/[\u0300-\u036f]/g, '')
   .replace(/\s+/g, ' ')
```

```
    .trim()
    .toLowerCase();

const parNomOuPrenom = recherche => liste =>
  recherche === '' ? liste :
  liste.filter(e =>
    normaliser(`${e.nom} ${e.prenom}`).includes(normaliser(rech
  );

const parAnnee = annee => liste =>
  Number.isNaN(annee) ? liste :
  liste.filter(e =>
    parseInt(String(e.date_naissance).split('.')[2], 10) === ar
  );

const parSexe = sexe => liste =>
  !['M', 'F', 'N'].includes(sexe) ? liste :
  liste.filter(e => String(e.sexe).toUpperCase() === sexe);

const versionCondensee = condenser => liste =>
  condenser ?
  liste.map(({ nom, prenom, date_naissance }) =>
    ({ nom, prenom, date_naissance })
  ) : liste;

// Pipeline complet
const construireResultat = pipe(
  parNomOuPrenom('Marie'),
  parAnnee(2005),
  parSexe('F'),
  versionCondensee(true)
);

const resultat = construireResultat(eleves);
```

La **récursion** désigne une fonction qui s'appelle elle-même jusqu'à atteindre un cas de base. Elle remplace souvent les boucles dans des

langages fonctionnels et est utilisée pour parcourir des structures arborescentes ou implémenter des algorithmes classiques (factorielles, suites de Fibonacci, recherche dans un arbre).

```
// Compte à rebours récursif
function compteARebours(n) {
  if (n < 0) return;           // cas de base
  console.log(n);
  compteARebours(n - 1);       // appel récursif
}
compteARebours(5); // 5 4 3 2 1 0

// Somme d'un tableau (récursif)
function somme(tableau, index = 0) {
  if (index >= tableau.length) return 0;  // cas de base
  return tableau[index] + somme(tableau, index + 1);
}
somme([1, 2, 3, 4]); // 10

// Fibonacci
const fib = n => (n <= 1 ? n : fib(n - 1) + fib(n - 2));
fib(9); // 34
```

Cas d'usage réel : parcourir un système de fichiers

```
// Calculer la taille totale d'un volume (récursif)
const tailletotale = noeud => {
  const tailleFichiers = (noeud.fichiers || [])
    .reduce((s, f) => s + f.taille, 0);

  const tailleDossiers = (noeud.dossiers || [])
    .reduce((s, d) => s + tailleTotale(d), 0);

  return tailleFichiers + tailleDossiers;
};
```

```
const total = tailleTotale(filesystem); // 20660497
```

Vérifier la présence d'une extension dans l'arborescence

```
const contientExtension = (noeud, ext) => {
  // Regarde les fichiers du dossier courant
  if ((noeud.fichiers || []).some(f => f.extension === ext)) {
    return true;
  }
  // Sinon, regarde récursivement ses sous-dossiers
  return (noeud.dossiers || []).some(d =>
    contientExtension(d, ext)
  );
};

contientExtension(filesystem, '.mp3'); // true
contientExtension(filesystem, '.mp4'); // false
```

Reconstituer un arbre généalogique

```
// Grouper d'abord par parent
const enfantsParParent = animaux.reduce((acc, { nom, nomParent
  const cle = nomParent === null ? '__ROOT__' : nomParent;
  (acc[cle] ||= []).push(nom);
  return acc;
}, {});

// Construire récursivement le sous-arbre
const construireArbre = nom => {
  const enfants = enfantsParParent[nom] || [];
  const noeud = {};
  for (const enfant of enfants) {
    noeud[enfant] = construireArbre(enfant);
  }
  return noeud;
```

```
};

const arbreGenealogique = { animal: construireArbre('animal') }
```

⚠️ **Attention aux limites !** Les langages généralistes comme JavaScript ont des limites de taille de pile. Des appels récursifs trop profonds peuvent provoquer un **stack overflow**. Si le moteur de votre langage ne supporte pas l'optimisation de **récursion terminale**, privilégiez des approches itératives pour de grandes profondeurs.

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

⚠️ Inverse **en place** l'ordre des éléments et retourne **le même tableau** (mutation).

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

// Sans muter l'original
const b = [1, 2, 3];
const inverse = [...b].reverse(); // [3,2,1]  (b intact)

// Alternative moderne (si disponible) : ne mute pas
[1, 2, 3].toReversed?.(); // [3,2,1]

// Inverser une chaîne
"salut".split('').reverse().join(''); // "tulas"
```

``(backticks) - pour des expressions intelligentes

> 🔗 **Interconnexions**

> - Combine avec `JSON.stringify(obj, null, 2)` pour de beaux logs formatés.
> - Interpole directement des résultats de `map()`/`reduce()` dans une chaîne multi-ligne.

Les backticks (template literals) permettent de créer des chaînes de caractères **multi-lignes** et d'y insérer des expressions JavaScript via la syntaxe `${expression}`. Ils sont particulièrement utiles pour formater du texte sans concaténation et pour interpoler des variables.

```
const nom = 'Dupont';
const age = 28;
const message = `Bonjour, ${nom}! Vous avez ${age} ans.`;
console.log(message); // Bonjour, Dupont! Vous avez 28 ans.

// Chaîne multi-ligne
const texte = `Ligne 1
Ligne 2
Ligne 3`;

// Expressions complexes
const prix = 100;
const tva = 0.077;
const total = `Total TTC: ${(prix * (1 + tva)).toFixed(2)} CHF`

// Cas d'usage : formater des résultats
const eleve = { prenom: 'Marie', nom: 'Dupont' };
const presentation = `${eleve.prenom} ${eleve.nom.toUpperCase()
// "Marie DUPONT"

// Avec map() pour formater une liste
const notes = [5, 4.5, 6];
notes.map((n, i) => `Évaluation ${i+1}: ${n}`);
// ["Évaluation 1: 5", "Évaluation 2: 4.5", "Évaluation 3: 6"]
```

`new Set()` - pour supprimer les doublons

> 🔗 **Interconnexions**
>
> - `const uniques = [...new Set(arr)]` puis `filter()` pour ne garder que ce qui t'intéresse.
> - `Set` + `map()` : créer un index rapide (`Set` des IDs) et filtrer un autre tableau en O(1) par lookup.

Le constructeur `Set` crée un ensemble de valeurs **uniques**. Il est utile pour éliminer les doublons d'un tableau : convertir le tableau en `Set` supprime automatiquement les valeurs répétées, puis on peut reconvertir l'ensemble en tableau.

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

// Cas d'usage : liste de "NOM Prénom" sans doublons
const eleves = [
  { nom: 'Dupont', prenom: 'Marie' },
  { nom: 'Martin', prenom: 'Jean' },
  { nom: 'Dupont', prenom: 'Marie' } // doublon
];
const liste = eleves.map(e => `${e.nom.toUpperCase()} ${e.prenc
const sansDuplique = Array.from(new Set(liste)).sort();
// ["DUPONT Marie", "MARTIN Jean"]

// Vérifier rapidement la présence (O(1))
const extensionsAutorisees = new Set(['jpg', 'png', 'gif']);
const estAutorise = extensionsAutorisees.has('png'); // true

// Compter le nombre de jours uniques
const nombreJoursUniques = [...new Set(ventes.map(v => v.date))

// Extraire les causes de décès uniques
```

```
const causes = new Set();
Object.values(dynasties).forEach(liste => {
  liste.forEach(e => {
    causes.add(e.deathCause || "Unknown");
  });
});
const causesTriees = Array.from(causes).sort();
```

Standard (function statement)

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

Sous forme d'expression de fonction

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

Sous forme d'expression de fonction fléchée (arrow function)

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

Sous forme raccourcie (arrow function avec retour implicite)

S'il n'y a qu'un seul argument et que son corps n'a qu'une seule expression, on peut omettre le `return`, les accolades et même les parenthèses autour du paramètre :

```
const direBonjour = nom => `Salut ${nom} !`;
```

```
direBonjour('Marie'); // "Salut Marie !"

// Avec plusieurs arguments, les parenthèses sont obligatoires
const additionner = (a, b) => a + b;

// Pour retourner un objet, il faut l'entourer de parenthèses
const creerPersonne = (nom, age) => ({ nom, age });
```

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(){
  console.log("Exécuté immédiatement !");
})();
```

ou avec une fonction fléchée :

```
(() => {
  console.log("Exécuté immédiatement !");
})();
```

Cas d'usage :

```
// Isoler des variables
const resultat = (() => {
  const valeurTemporaire = 42 * 2;
  const autreCalcul = valeurTemporaire + 10;
  return autreCalcul;
})();
```

```
// resultat = 94, mais valeurTemporaire n'est pas accessible ic
```

Le **currying** transforme une fonction à plusieurs paramètres en une séquence de fonctions à un seul paramètre. Cela permet de :

- Créer des fonctions spécialisées
- Réutiliser facilement du code
- Composer des fonctions

```
// Fonction currifiée
const multiplier = a => b => a * b;

const doubler = multiplier(2);
const tripler = multiplier(3);

doubler(5);  // 10
tripler(5);  // 15

[1, 2, 3].map(doubler);  // [2, 4, 6]

// Cas pratique : créer un filtre réutilisable
const creerFiltreSeuil = seuil => valeur => valeur > seuil;

const plusGrandQue10 = creerFiltreSeuil(10);
const plusGrandQue100 = creerFiltreSeuil(100);

[5, 15, 120].filter(plusGrandQue10);   // [15, 120]
[5, 15, 120].filter(plusGrandQue100);  // [120]
```

Les **closures** permettent à une fonction de "se souvenir" de son environnement lexical :

```
// Closure : compteur privé
const creerCompteur = () => {
  let compte = 0;  // variable privée
```

```
  return {
    incrementer: () => ++compte,
    obtenir: () => compte,
    reinitialiser: () => { compte = 0; }
  };
};

const compteur = creerCompteur();
compteur.incrementer(); // 1
compteur.incrementer(); // 2
compteur.obtenir();     // 2
compteur.reinitialiser();
compteur.obtenir();     // 0
```

Patterns avancés et cas d'usage réels

Cette section présente des patterns pratiques tirés directement des exercices du module.

Parser et transformer du CSV

Le parsing de CSV est un cas d'usage fréquent qui combine plusieurs techniques fonctionnelles.

```
function convertirCSVEnObjet(contenuCSV) {
  if (typeof contenuCSV !== "string") return {};

  // 1) Découper en lignes (gérer \n et \r\n)
  const lignes = contenuCSV
    .split(/\r?\n/)
    .map(l => l.trim())
    .filter(l => l.length > 0);

  if (lignes.length === 0) return {};

  // 2) Lire l'en-tête, retirer le BOM éventuel
```

```
let entete = lignes[0];
if (entete.charCodeAt(0) === 0xFEFF) {
  entete = entete.slice(1);
}

const nomsColonnes = entete.split(";").map(c => c.trim());

// Trouver les indices des colonnes
const indices = {
  index: nomsColonnes.indexOf("Index"),
  name: nomsColonnes.indexOf("Name"),
  cause: nomsColonnes.indexOf("Cause"),
  dynasty: nomsColonnes.indexOf("Dynasty"),
};

// Vérifier que toutes les colonnes existent
if (Object.values(indices).some(i => i < 0)) return {};

// 3) Transformer les lignes en objets
const empereurs = lignes
  .slice(1) // ignorer l'en-tête
  .map(ligne => ligne.split(";"))
  .map(colonnes => ({
    number: parseInt(colonnes[indices.index], 10),
    name: (colonnes[indices.name] || "").trim(),
    deathCause: (colonnes[indices.cause] || "").trim(),
    dynasty: (colonnes[indices.dynasty] || "").trim(),
  }))
  .filter(e => Number.isFinite(e.number) && e.dynasty !== "")

// 4) Trier chronologiquement
empereurs.sort((a, b) => a.number - b.number);

// 5) Regrouper par dynastie
const parDynastie = empereurs.reduce((acc, e) => {
  if (!acc[e.dynasty]) acc[e.dynasty] = [];
  acc[e.dynasty].push({
    number: e.number,
```

```
      name: e.name,
      deathCause: e.deathCause
    });
    return acc;
  }, {});

  return parDynastie;
}
```

Pattern très fréquent pour organiser des données.

```
// Regrouper par clé simple
const parCanton = villes.reduce((acc, ville) => {
  (acc[ville.canton] ??= []).push(ville);
  return acc;
}, {});

// Regrouper et agréger (somme des populations)
const populationParCanton = villes.reduce((acc, { canton, habit
  acc[canton] = (acc[canton] || 0) + habitants;
  return acc;
}, {});

// Puis trier et transformer en tableau
const liste = Object.entries(populationParCanton)
  .sort(([a], [b]) => a.localeCompare(b))
  .map(([canton, population]) => ({ canton, population }));

// Ou simplement trier un objet par ses clés
const triePar Clé = Object.fromEntries(
  Object.entries(populationParCanton)
    .sort(([a], [b]) => a.localeCompare(b))
);



// Min et max d'un tableau de dates
```

```
const dates = ventes.map(v => v.date);
const dateMin = dates.reduce((min, d) => d < min ? d : min, dat
const dateMax = dates.reduce((max, d) => d > max ? d : max, dat

// CA moyen par jour
const caTotal = ventes.reduce((somme, v) =>
  somme + v.produits.reduce((s, p) => s + p.prix, 0),
  0
);
const nombreJours = new Set(ventes.map(v => v.date)).size;
const caMoyenParJour = caTotal / nombreJours;

// Statistiques par jour de semaine
const joursSemaine = [
  "dimanche","lundi","mardi","mercredi",
  "jeudi","vendredi","samedi"
];

const statsParJour = ventes.reduce((acc, v) => {
  const jour = joursSemaine[new Date(v.date).getDay()];
  const caTicket = v.produits.reduce((s, p) => s + p.prix, 0);

  (acc[jour] ??= { ca: 0, nombreVentes: 0 });
  acc[jour].ca += caTicket;
  acc[jour].nombreVentes += 1;

  return acc;
}, {});

// Top N produits par CA
const TOP = 6;
const caParProduit = ventes
  .flatMap(v => v.produits)
  .reduce((acc, { id, nom, prix }) => {
    (acc[id] ??= { id, nom, ca: 0 }).ca += prix;
    return acc;
  }, {});
```

```
const topProduits = Object.values(caParProduit)
  .sort((a, b) => b.ca - a.ca)
  .slice(0, TOP);



// Clients uniques triés (NOM en MAJ, Prénom capitalisé)
const clientsTries = ventes
  .reduce((acc, v) => {
    const { id, nom, prenom } = v.client;
    if (!acc[id]) acc[id] = { nom, prenom };
    return acc;
  }, {})
  |> Object.values
  |> (clients => clients.map(({ nom, prenom }) => ({
      nom: nom.toUpperCase(),
      prenom: prenom.charAt(0).toUpperCase() + prenom.slice(1).
    })))
  |> (clients => clients.sort((a, b) =>
      a.nom.localeCompare(b.nom, 'fr') ||
      a.prenom.localeCompare(b.prenom, 'fr')
    ));

// Ou sans pipeline operator :
const clientsTries = Object.values(
  ventes.reduce((acc, v) => {
    const { id, nom, prenom } = v.client;
    if (!acc[id]) acc[id] = { nom, prenom };
    return acc;
  }, {})
)
.map(({ nom, prenom }) => ({
  nom: nom.toUpperCase(),
  prenom: prenom.charAt(0).toUpperCase() + prenom.slice(1).toLc
}))
.sort((a, b) =>
  a.nom.localeCompare(b.nom, 'fr') ||
  a.prenom.localeCompare(b.prenom, 'fr')
);
```

```
// Nombre de produits vendus par type et nom
const venduParTypeEtProduit = ventes
  .flatMap(v => v.produits)
  .reduce((acc, p) => {
    (acc[p.type] ??= {});
    acc[p.type][p.nom] = (acc[p.type][p.nom] ?? 0) + 1;
    return acc;
  }, {});

// Pour chaque marque : modèle le moins cher et le plus cher
const extremesParMarque = motos.map(({ marque, modeles }) => {
  const moinsCher = modeles.reduce((min, m) =>
    m.prix < min.prix ? m : min,
    modeles[0]
  );
  const plusCher = modeles.reduce((max, m) =>
    m.prix > max.prix ? m : max,
    modeles[0]
  );
  return {
    marque,
    moinsCher: { nom: moinsCher.nom, prix: moinsCher.prix },
    plusCher: { nom: plusCher.nom, prix: plusCher.prix }
  };
});
```

Au terme de ce module **323 - Programmation fonctionnelle**, nous avons acquis une **maîtrise des concepts et techniques essentiels** de la programmation fonctionnelle en JavaScript.

Ce que nous avons appris :

1. **Pensée déclarative** : Nous savons maintenant exprimer *ce que* nous voulons obtenir plutôt que *comment* le faire, rendant le code plus lisible et maintenable.

2. **Méthodes de tableaux** : Maîtrise complète de `map()`, `filter()`, `reduce()`, `flatMap()`, `sort()`, `every()`, `some()`, et bien d'autres pour transformer et manipuler des données sans effets de bord.

3. **Fonctions pures et immuabilité** : Comprendre l'importance de ne pas muter les données et d'écrire des fonctions sans effets de bord pour un code prévisible et testable.

4. **Composition et currying** : Savoir construire des fonctions complexes à partir de petites fonctions simples, utiliser le currying pour créer des fonctions réutilisables et composer des pipelines de traitement avec `pipe()` et `compose()`.

5. **Récursion** : Appliquer la récursion pour parcourir des structures arborescentes (systèmes de fichiers, arbres généalogiques) et résoudre des problèmes de manière élégante.

Compétences pratiques acquises :

- Parser et transformer des fichiers CSV en structures de données JavaScript
- Filtrer, trier et regrouper des données complexes
- Calculer des statistiques (moyennes, min/max, totaux, fréquences)
- Construire des chaînages de fonctions pour des transformations de données élégantes
- Manipuler des objets et tableaux imbriqués sans mutation
- Créer des fonctions génériques et réutilisables avec currying et closures

Impact sur notre pratique de développement :

La programmation fonctionnelle nous a permis de :

- Écrire du code plus **concis** et **expressif**
- Réduire les bugs liés aux effets de bord et à la mutation d'état

- Créer des fonctions **testables** unitairement
- Améliorer la **lisibilité** et la **maintenabilité** du code
- Raisonner plus facilement sur le comportement du programme

Ce module nous a fourni une **boîte à outils puissante** pour résoudre des problèmes réels de transformation et d'analyse de données, compétences essentielles pour tout développeur moderne.

Points clés à retenir :

> **"Les données sont immuables, les transformations sont pures, et le code raconte une histoire."**

- Toujours privilégier `map()`, `filter()`, `reduce()` aux boucles `for`
- Ne jamais muter les données d'entrée
- Composer des fonctions simples pour construire des comportements complexes
- Utiliser la récursion quand la structure du problème est récursive
- Penser en termes de transformation de données plutôt que d'instructions

Pour aller plus loin :

- Approfondir les concepts de **functors**, **monads** et **category theory**
- Explorer des bibliothèques fonctionnelles comme **Ramda**, **Lodash/FP**
- Étudier des langages purement fonctionnels (**Haskell**, **Elm**, **Elixir**)
- Appliquer ces principes dans d'autres contextes (React, Redux, observables RxJS)

La programmation fonctionnelle n'est pas seulement une technique, c'est une **manière de penser** qui rend le code plus robuste, plus testable et plus maintenable. Les concepts appris ici sont transférables à de nombreux langages et frameworks modernes.

Auteur: Maxime Bardy
Module: 323 - Programmation fonctionnelle
École des Métiers de Fribourg
Date: Novembre 2025