RP - 323 - Programmation fonctionnelle

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

Table des matières

Introduction

Ce module explore les concepts fondamentaux de la programmation fonctionnelle en JavaScript. Il couvre les opérateurs, les méthodes, et les techniques essentielles pour écrire du code moderne, lisible et performant.

Objectif opérationnels :

- Connaître la différence entre programmation impérative et programmation déclarative/fonctionnelle (paradigme de programmation déclarative).
- Connaître des méthodes de description déclarative de problèmes et d'états finaux.
- Connaître les avantages et les inconvénients de la programmation fonctionnelle.

- Connaître les termes de la programmation fonctionnelle (p. ex. fonction pure, donnée immuable, expression lambda, fonction, fermeture [closure], rappel [callback], foncteur), les comprendre et pouvoir les utiliser correctement.
- Connaître des possibilités pour élaborer un concept de réalisation adapté à la programmation fonctionnelle.
- Pouvoir lire, comprendre, maintenir et étendre un code fonctionnel.
- Connaître des concepts de programmation fonctionnelle (p. ex. filter, map, reduce) et des patrons de conception adaptés (p. ex. builder pattern).
- Connaître des méthodes pour exécuter un code de manière distribuée, parallèle ou concurrente.
- Connaître des moyens pour déterminer si des parties de programmes peuvent, après examen, être optimisées ou améliorées grâce à l'implémentation fonctionnelle.
- Connaître les éléments fonctionnels d'un langage de programmation et pouvoir ainsi développer des applications avec un paradigme de programmation impérative et déclarative.
- Connaître des moyens de tester un code fonctionnel implémenté (p. ex. tests unitaires).
- Connaître des directives de code appropriées et les appliquer de manière systématique.
- Connaître les meilleures pratiques de la programmation fonctionnelle et pouvoir les appliquer.

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

Description : constante numérique π (environ 3.14159). Utile pour calculs géométriques.

```
const pi = Math.PI;
console.log(pi); // 3.141592653589793
```

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

Retourne la valeur positive d'un nombre.

```
const abs5 = Math.abs(-5);
console.log(abs5); // 5
```

`Math.pow()` - élever à une puissance

Élévation en puissance (a^b). Utiliser aussi l'opérateur ** en ES2016+.

```
const pow1 = Math.pow(2, 3); // 8
const pow2 = 2 ** 3; // 8
console.log(pow1);
console.log(pow2);
```

`Math.min()` - plus petite valeur

Retourne la plus petite valeur parmi les arguments.

```
const minVal = Math.min(4, 1, 7); // 1
console.log(minVal);
```

`Math.max()` - plus grande valeur

Retourne la plus grande valeur parmi les arguments.

```
const maxVal = Math.max(4, 1, 7); // 7
console.log(maxVal);
```

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

Arrondit vers le haut.

```

```

```
const ceilVal = Math.ceil(3.1); // 4
console.log(ceilVal);
```

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

Arrondit vers le bas.

```
const floorVal = Math.floor(3.9); // 3
console.log(floorVal);
```

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

Arrondit à l'entier le plus proche.

```
const roundA = Math.round(3.4); // 3
const roundB = Math.round(3.6); // 4
console.log(roundA);
console.log(roundB);
```

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

Supprime la partie fractionnaire sans arrondi.

```
const truncA = Math.trunc(3.9); // 3
const truncB = Math.trunc(-3.9); // -3
console.log(truncA);
console.log(truncB);
```

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

```
const sqrt9 = Math.sqrt(9); // 3
console.log(sqrt9);
```

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

Utilisé pour générer valeurs aléatoires. Pour un entier dans [min,max] : Math.floor(Math.random() * (max - min + 1)) + min

```
```

```
const rnd = Math.random();
// exemple: entier entre 1 et 6
const dice = Math.floor(Math.random() * 6) + 1;
console.log(rnd);
console.log(dice);
```

JSON

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

Convertit un objet en chaîne JSON. Utile pour stocker ou envoyer des données.

```
const obj = { nom: 'Alice', age: 30 };
const json = JSON.stringify(obj);
console.log(json); // '{"nom":"Alice","age":30}'
```

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

Inverse de stringify.

```
const jsonStr = '{"nom":"Alice","age":30}';
const parsed = JSON.parse(jsonStr);
const objName = parsed.nom;
console.log(objName); // 'Alice'
```

Chaînes de caractères

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

````
Description : `split(separator, limit?)` divise une chaîne en sous-chaînes selon
le séparateur et renvoie un tableau. Le second argument optionnel `limit`
restreint le nombre d'éléments retournés.

```javascript
const s = 'a,b,c';
const parts = s.split(',');
console.log(parts); // ['a','b','c']
````
````

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

````
Description : `trim()` supprime les espaces en début et fin d'une chaîne.
`trimStart()` et `trimEnd()` ne suppriment que le début ou la fin respectivement.
Utile pour nettoyer les entrées utilisateur.

```javascript
const t = '  hello  ';
const tTrim = t.trim(); // 'hello'
const tStart = t.trimStart(); // 'hello  '
const tEnd = t.trimEnd(); // '  hello'
console.log(tTrim);
console.log(tStart);
console.log(tEnd);
````

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

````
Description : `padStart(targetLength, padString)` et `padEnd(targetLength,
padString)` complètent une chaîne par la gauche ou la droite pour atteindre la
longueur cible. Utile pour aligner ou formater des identifiants.

```javascript
const padA = '5'.padStart(3, '0'); // '005'
const padB = '5'.padEnd(3, '-'); // '5--'
console.log(padA);
console.log(padB);
````

Console

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

````
Description : `console.log()` affiche des informations sur la console du
navigateur ou du terminal. Utile pour le débogage et l'affichage d'états
intermédiaires. Ne renvoie pas de valeur significative.

```javascript
const greeting = 'Coucou !';
console.log(greeting);
````

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

````
Description : `console.info()`, `console.warn()` et `console.error()` affichent
des messages avec des niveaux distincts (info, avertissement, erreur). Les
environnements peuvent filtrer ces niveaux.

```javascript
const info = 'Info';
const warnMsg = 'Attention';
const errMsg = 'Erreur';
console.info(info);
console.warn(warnMsg);
console.error(errMsg);
````

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

````
Description : `console.table()` affiche un tableau ou un objet sous forme
tabulaire, pratique pour visualiser des données structurées (colonnes/ligne).

```javascript
const tableData = [
  { nom: 'Alice', age: 30 },
  { nom: 'Bob', age: 25 },
];
console.table(tableData);
````

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

````
Description : `console.time(label)` démarre un chronomètre nommé,
`console.timeLog(label)` affiche un point intermédiaire, et
`console.timeEnd(label)` arrête et affiche la durée écoulée. Utile pour mesurer la
performance.

```javascript
console.time('op');
// ... faire quelque chose
console.timeLog('op', 'milestone');
console.timeEnd('op');
````

Tableaux

`forEach` - parcourir les éléments d'un tableau
````

Description : `forEach(fn)` exécute la fonction fournie pour chaque élément du tableau. C'est une méthode de parcours destinée aux effets de bord (affichage, mutation). Elle ne renvoie pas de nouveau tableau et ne peut pas être interrompue avec un `return` pour sortir.

```javascript
['croissant', 'gateau', 'pain'].forEach((item) => {
  console.log(item); // Affiche chaque élément : 'croissant', 'gateau', 'pain'
});
```

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

Description : `entries()` retourne un itérateur produisant des paires `[index, valeur]`. Utile quand on veut à la fois l'indice et la valeur. Pour récupérer toutes les paires sous forme de tableau, utilisez le spread ou `Array.from()`.

```
for (const [index, item] of ['croissant', 'gateau', 'pain'].entries()) {
  console.log(`Index ${index}: ${item}`); // Affiche : "Index 0: croissant",
"Index 1: gateau", "Index 2: pain"
}
console.log([...['croissant','gateau','pain'].entries()]); // [[0,'croissant'],
[1,'gateau'],[2,'pain']]
```

`in` - parcourir les clés d'un tableau

Description : `for...in` itère sur les clés énumérables d'un objet (pour un tableau : les indices). Il est généralement déconseillé pour les tableaux quand l'ordre ou les propriétés héritées peuvent poser problème ; préférez `for...of` ou les méthodes de tableau.

```javascript
const desserts = ['croissant', 'gateau'];
for (const index in desserts) {
  console.log(index); // Affiche les indices : '0', '1'
}
```

`of` - parcourir les valeurs d'un tableau

Description : `for...of` parcourt les valeurs d'un itérable (tableau, Map, Set, etc.). C'est la boucle recommandée pour itérer sur les éléments eux-mêmes.

```javascript
for (const item of ['croissant', 'gateau']) {
  console.log(item); // Affiche : 'croissant', 'gateau'
}
```

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

Description : `find(predicate)` renvoie le premier élément satisfaisant la fonction test, ou `undefined` si aucun élément ne correspond. Ne renvoie pas l'indice.

```
const desserts = ['croissant', 'gateau', 'pain'];
const found = desserts.find((item) => item.startsWith('g')); // Trouve 'gateau'
console.log(found);
```

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

Description : `findIndex(predicate)` renvoie l'index du premier élément qui satisfait le test, ou `-1` si aucun élément ne correspond.

```
const desserts = ['croissant', 'gateau', 'pain'];
const index = desserts.findIndex((item) => item === 'gateau'); // Trouve l'index 1
console.log(index);
```

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

Description : `indexOf(value)` et `lastIndexOf(value)` retournent respectivement la première et la dernière position d'une valeur en utilisant la comparaison stricte (`===`). Elles renvoient `-1` si la valeur n'est pas trouvée.

```
const desserts = ['croissant', 'gateau', 'croissant'];
const firstIndex = desserts.indexOf('croissant'); // 0
const lastIndex = desserts.lastIndexOf('croissant'); // 2
console.log(firstIndex, lastIndex);
```

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

Description : ces méthodes modifient (mutent) le tableau en place : `push`/`unshift` ajoutent des éléments (fin/début), `pop`/`shift` retirent et retournent un élément (fin/début).

```
const desserts = ['croissant', 'gateau'];
desserts.push('pain'); // Ajoute à la fin : ['croissant', 'gateau', 'pain']
desserts.pop(); // Supprime le dernier : ['croissant', 'gateau']
desserts.unshift('tarte'); // Ajoute au début : ['tarte', 'croissant', 'gateau']
desserts.shift(); // Supprime le premier : ['croissant', 'gateau']
console.log(desserts);
```

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

Description : `slice(start, end)` crée un NOUVEAU tableau contenant une portion (start inclusive, end exclusive). Ne modifie pas le tableau source.

```
const desserts = ['croissant', 'gateau', 'pain', 'tarte'];
const sliced = desserts.slice(1, 3); // ['gateau', 'pain']
console.log(sliced);
```

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

Description : `splice(start, deleteCount, ...items)` modifie le tableau en place : supprime `deleteCount` éléments à partir de `start`, et insère `items` à cet emplacement. Renvoie les éléments supprimés.

```
const desserts = ['croissant', 'gateau', 'pain'];
desserts.splice(1, 1, 'tarte'); // Remplace 'gateau' par 'tarte' : ['croissant',
'tarte', 'pain']
console.log(desserts);
```

`concat()` - joindre deux tableaux

Description : `concat()` crée et renvoie un NOUVEAU tableau contenant les éléments du tableau appelant suivis des éléments fournis en arguments. Elle ne modifie pas les tableaux sources (opération non mutative). Les arguments peuvent être des valeurs simples ou des tableaux — si un argument est un tableau, ses éléments sont copiés (copie superficielle).

Complexité : proportionnelle à la taille des tableaux concaténés (O(n)).

Exemples :

```
const desserts1 = ['croissant', 'gateau'];
const desserts2 = ['pain', 'tarte'];
const allDesserts = desserts1.concat(desserts2); // ['croissant', 'gateau',
'pain', 'tarte']
console.log(allDesserts);

// concat accepte aussi plusieurs arguments
const merged = desserts1.concat(['glace'], desserts2); //
['croissant','gateau','glace','pain','tarte']
console.log(merged);
```

`join()` - joindre des chaînes de caractères

Description : `join()` transforme les éléments d'un tableau en une seule chaîne de caractères, séparés par le séparateur fourni (par défaut `,`). Les éléments `null` ou `undefined` sont convertis en chaîne vide. `join()` ne modifie pas le tableau d'origine.

Remarques : si le tableau est vide, `join()` renvoie une chaîne vide. Les éléments sont convertis en chaînes via `toString()` avant la concatenation.

Exemples :

```
const desserts = ['croissant', 'gateau', 'pain'];
const joined = desserts.join(', '); // 'croissant, gateau, pain'
console.log(joined);

const withEmpty = ['a', null, undefined, 'b'];
console.log(withEmpty.join('-')); // 'a--b' => null/undefined -> '' donc 'a--b'
console.log([].join('-')); // '' (chaine vide)
```

`keys()` et `values()` - itérateurs de clés / valeurs (Array / Map)

Description : `keys()` et `values()` retournent des objets itérables (Iterator) — pas des tableaux — qui parcourent respectivement les indices/cles et les valeurs de la collection. Pour obtenir un tableau à partir de l'itérateur, on peut utiliser l'opérateur spread (`[...]`) ou `Array.from()`.

Différences importantes :

- Pour les `Array`, `keys()` itère les indices (0, 1, 2...), `values()` itère les éléments.
- Pour une `Map`, `keys()` renvoie les clés telles qu'elles ont été insérées, `values()` les valeurs associées.
- Pour les objets littéraux (plain objects), utiliser `Object.keys()` / `Object.values()` qui renvoient des tableaux.

Exemples :

```
// Array
const desserts = ['croissant', 'gateau'];
const keys = [...desserts.keys()]; // [0, 1]
const values = [...desserts.values()]; // ['croissant', 'gateau']
console.log(keys, values);

// Map
const m = new Map([['a', 1], ['b', 2]]);
console.log([...m.keys()]); // ['a', 'b']
console.log([...m.values()]); // [1, 2]

// Objet littéral -> utiliser Object.keys / Object.values
const obj = { x: 10, y: 20 };
console.log(Object.keys(obj)); // ['x','y']
console.log(Object.values(obj)); // [10,20]
```

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

Description : `includes(value)` retourne `true` si le tableau contient la valeur (utilise l'égalité SameValueZero — gère `NaN`).

```
const desserts = ['croissant', 'gateau', 'pain'];
const hasGateau = desserts.includes('gateau'); // true
console.log(hasGateau);
```

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

Description : `every(predicate)` vérifie que tous les éléments satisfont la condition (renvoie true ou false). `some(predicate)` vérifie qu'au moins un élément satisfait la condition. Ces méthodes s'arrêtent dès que le résultat est déterminé.

```
const desserts = ['croissant', 'gateau', 'pain'];
const allStrings = desserts.every((item) => typeof item === 'string'); // true
const hasPain = desserts.some((item) => item === 'pain'); // true
console.log(allStrings, hasPain);
```

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

Description : `fill(value, start=0, end=array.length)` remplit (mutatif) les éléments du tableau avec `value` entre `start` (inclus) et `end` (exclus).

```
const desserts = new Array(3).fill('croissant'); // ['croissant', 'croissant',
'croissant']
console.log(desserts);
```

`flat()` - aplatir un tableau

Description : `flat(depth=1)` renvoie un nouveau tableau aplatissant les sous-tableaux jusqu'à la profondeur `depth` (par défaut 1). Ne modifie pas le tableau d'origine.

```
const nestedDesserts = ['croissant', ['gateau', 'pain']];
const flatDesserts = nestedDesserts.flat(); // ['croissant', 'gateau', 'pain']
console.log(flatDesserts);
```

`sort()` - pour trier un tableau

Description : `sort()` trie le tableau EN PLACE (mutatif). Par défaut il trie selon l'ordre lexicographique des chaînes ; pour trier des nombres, fournissez une fonction de comparaison `(a,b) => a-b`.

```
const desserts = ['pain', 'croissant', 'gateau'];
const sortedDesserts = desserts.sort(); // ['croissant', 'gateau', 'pain']
console.log(sortedDesserts);

const nums = [3, 1, 10, 2];
nums.sort(); // incorrect pour nombres : ['1','10','2','3']
nums.sort((a, b) => a - b); // correcte pour nombres : [1,2,3,10]
console.log(nums);
```

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

Description : `map(fn)` crée et renvoie un NOUVEAU tableau résultant de l'application de `fn` à chaque élément. N'est pas mutatif.

```
const desserts = ['croissant', 'gateau'];
const upperDesserts = desserts.map((item) => item.toUpperCase()); // ['CROISSANT',
'GATEAU']
console.log(upperDesserts);
```

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

Description : `filter(fn)` crée un NOUVEAU tableau contenant les éléments pour lesquels `fn` retourne `true`. Ne modifie pas l'original.

```
const desserts = ['croissant', 'gateau', 'pain'];
const filteredDesserts = desserts.filter((item) => item.startsWith('g')); //
['gateau']
console.log(filteredDesserts);
```

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

Note: `groupBy` est récent dans certaines versions JS. Voici un exemple manuel :

```
const items = ['apple', 'banana', 'avocado'];
const grouped = items.reduce((acc, item) => {
    const key = item[0];
    (acc[key] ||= []).push(item);
    return acc;
}, {});
console.log(grouped); // { a: ['apple','avocado'], b: ['banana'] }
```

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

Description : `flatMap(fn)` applique `fn` à chaque élément puis aplatit d'un niveau le résultat. Équivalent à `arr.map(fn).flat(1)`, mais plus performant.

```
const flatMapped = [1, 2].flatMap((x) => [x, x * 2]); // [1,2,2,4]
console.log(flatMapped);
```

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

Description : `reduce((acc, val) => ..., initial)` accumule une valeur en parcourant le tableau. Si `initial` est omis, le premier élément du tableau est utilisé comme valeur initiale et la réduction commence au deuxième élément — attention aux tableaux vides (erreur si aucun `initial`).

```
const reduced = [1, 2, 3].reduce((sum, x) => sum + x, 0); // 6
console.log(reduced);
```

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

Description : `reverse()` inverse l'ordre des éléments du tableau EN PLACE (mutation). Renvoie le tableau modifié.

```
const r = [1, 2, 3];
r.reverse(); // [3,2,1]
console.log(r);
```

Techniques

``(backticks) - pour des expressions intelligentes

`` (backticks) servent à créer des template strings et interpoler des variables facilement.

```
const name = 'Alice';
const greetTpl = `Salut ${name} !`;
console.log(greetTpl); // Salut Alice !
```

`new Set()` - pour supprimer les doublons

```
const arrUniq = [1, 2, 2, 3];
const unique = [...new Set(arrUniq)]; // [1,2,3]
```

```
console.log(unique);
```

Fonctions

Déclaration de fonction

Standard :

Une fonction classique est définie avec le mot-clé `function`. Elle peut être appelée avant ou après sa déclaration grâce au **hoisting**.

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

console.log(addition(1, 2, 3)); // 6
```

Expression :

Une expression de fonction est assignée à une variable. Contrairement à une déclaration standard, elle **n'est pas "hoistée"** et doit être définie avant d'être utilisée.

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

console.log(addition(1, 2, 3)); // 6
```

Flèche :

Les fonctions fléchées (arrow functions) sont une syntaxe plus concise introduite avec ES6. Elles sont particulièrement utiles pour les fonctions courtes et les callbacks.

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

console.log(addition(1, 2, 3)); // 6
```

Sous forme raccourcie :

Si une fonction fléchée n'a qu'un seul argument et une seule expression dans son corps, on peut omettre les parenthèses autour de l'argument et le mot-clé `return`.

```
const saluer = (nom) => `Salut ${nom} !`;

console.log(saluer('Alice')); // "Salut Alice !"
```

Qu'est-ce qu'une IIFE ?

Une **IIFE** (Immediately Invoked Function Expression) est une fonction qui est définie et exécutée immédiatement. Elle est souvent utilisée pour encapsuler du code et éviter de polluer l'espace global.

Syntaxe classique

```
(function () {
    const secret = 42;
    console.log(`Le secret est ${secret}`);
})(); // Appelle immédiatement la fonction
```

Syntaxe avec une fonction fléchée

```
(() => {
    const secret = 42;
    console.log(`Le secret est ${secret}`);
})(); // Appelle immédiatement la fonction
```

Fonctions avec des valeurs par défaut

Les paramètres de fonction peuvent avoir des valeurs par défaut. Si aucun argument n'est fourni, la valeur par défaut est utilisée.

```
const saluer = (nom = 'inconnu') => `Bonjour, ${nom} !`;

console.log(saluer()); // "Bonjour, inconnu !"
console.log(saluer('Alice')); // "Bonjour, Alice !"
```

Fonctions retournant des fonctions

Une fonction peut retourner une autre fonction. Cela est utile pour créer des **factories** ou des fonctions personnalisées.

```
const createMultiplier = (facteur) => {
    return (nombre) => nombre * facteur;
};

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

console.log(doubler(5)); // 10
console.log(tripler(5)); // 15
```

Fonctions récursives

Une fonction récursive est une fonction qui s'appelle elle-même. Cela est utile pour résoudre des problèmes comme le calcul de factoriels ou le parcours d'arbres.

```
const factoriel = (n) => (n === 0 ? 1 : n * factoriel(n - 1));

console.log(factoriel(5)); // 120
```

Exemple combiné : `map`, `filter`, et `reduce`

Voici un exemple combinant les trois méthodes pour calculer la somme des carrés des nombres pairs dans un tableau.

```
const nombres = [1, 2, 3, 4, 5];
const sommeDesCarresPairs = nombres
    .filter((n) => n % 2 === 0) // Garde les nombres pairs
    .map((n) => n ** 2) // Calcule le carré de chaque nombre
    .reduce((acc, n) => acc + n, 0); // Fait la somme des carrés

console.log(sommeDesCarresPairs); // 20 (2² + 4²)
```

Conclusion

Ce module m'a permis de découvrir et d'explorer les principaux opérateurs, méthodes et techniques de la programattion fonctionelle en JavaScript. J'ai trouvé le contenu du module intéressant et enrichissant, même si certains concepts étaient un peu plus complexes.