🤔 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 racine carrée d'un nombre
  - `Math.random()` - générer un nombre aléatoire entre 0.0 (compris) et 1.0 (non compris)
- JSON

    - `JSON.stringify()` - transformer un objet Javascript en JSON
    - `JSON.parse()` - transformer du JSON en objet Javascript
- Chaînes de caractères
    - `split()` - un ciseau qui coupe une chaîne là où un caractère apparaît et produit un tableau
    - `trim()`, `trimStart()` et `trimEnd()` - épuration des espaces en trop dans une chaîne (trimming)
    - `padStart()` et `padEnd()` - aligner le contenu dans une chaîne de caractères
- Console
    - `console.log()` - Afficher un message sur la console
    - `console.info()`, `warn()` et `error()` - Afficher un message sur la console (filtrables)
    - `console.table()` - Afficher tout un tableau ou un objet sur la console
    - `console.time()`, `timeLog()` et `timeEnd()` - Chronométrer une durée d'exécution
- Tableaux
    - `forEach` - parcourir les éléments d'un tableau
    - `entries()` - parcourir les couples index/valeurs d'un tableau
    - `in` - parcourir les clés d'un tableau
    - `of` - parcourir les valeurs d'un tableau
    - `find()` - premier élément qui satisfait une condition
    - `findIndex()` - premier index qui satisfait une condition
    - `indexOf()` et `lastIndexOf()` - premier/dernier élément qui correspond
    - `push()`, `pop()`, `shift()` et `unshift()` - ajouter/supprimer au début/fin dans un tableau
    - `slice()` - ne conserver que certaines lignes d'un tableau
    - `splice()` - supprimer/insérer/remplacer des valeurs dans un tableau

    - `concat()` - joindre deux tableaux
    - `join()` - joindre des chaînes de caractères
    - `keys()` et `values()` - les clés/valeurs d'un objet
    - `includes()` - vérifier si une valeur est présente dans un tableau
    - `every()` et `some()` - vérifier si plusieurs valeurs sont toutes/quelques présentes dans un tableau
    - `fill()` - remplir un tableau avec des valeurs
    - `flat()` - aplatir un tableau
    - `sort()` - pour trier un tableau
    - `map()` - tableau avec les résultats d'une fonction
    - `filter()` - tableau avec les éléments passant un test
    - `groupBy()` - regroupe les éléments d'un tableau selon une règle
    - `flatMap()` - chaînage de map() et flat()
    - `reduce()` et `reduceRight()` - réduire un tableau à une seule valeur
    - `reverse()` - inverser l'ordre du tableau
- Recherche dans les structures de données
    - Trouver dans un tableau simple
    - Trouver dans un objet
    - Trouver dans un tableau d'objets
    - Trouver dans un objet contenant des tableaux
- Transformations de structures de données
    - Arrondir des nombres
    - Transformer un tableau en objet
    - Transformer un objet en tableau
    - Aplatir des structures imbriquées avec `flat()`
    - Transformer et aplatir avec `flatMap()`
    - Transformations complexes de structures imbriquées
    - Conversions aller-retour : Tableau ↔ Objet
- Techniques
    - ``(backticks) - pour des expressions intelligentes

    - `new Set()` - pour supprimer les doublons
- Fonctions
    - Déclaration de fonction
    - Fonctions immédiatement invoquées (IIFE)
- Conclusion

La programmation fonctionnelle est une façon de coder qui utilise des fonctions pour transformer les données plutôt que de modifier les données directement. Ce module (323) apprend à utiliser les méthodes `map()`, `filter()` et `reduce()` en JavaScript.

Les objectifs de ce module sont :

- Comprendre la programmation fonctionnelle
- Bien utiliser `map()`, `filter()` et `reduce()`
- Chaîner ces méthodes pour résoudre des problèmes
- Écrire du code plus clair et facile à lire
- Comparer le code procédural avec le code fonctionnel

Opérateurs javascript super-cooool 😎

> 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'
```

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 !
```

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 };
```

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]
```

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: "06.06.202
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: "2025-06-06T13:23
```

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

Donne la valeur de π (Pi), environ 3.14159. Utilisé pour les cercles et formes rondes.

```
const rayon = 5;
const perimetre = 2 * Math.PI * rayon; // 31.41592653589793
const aire = Math.PI * rayon * rayon; // 78.53981633974483
```

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

Donne la valeur positive d'un nombre. Ignore le signe (- ou +).

```
console.log(Math.abs(-5)); // 5
console.log(Math.abs(5)); // 5
console.log(Math.abs(-3.14)); // 3.14
```

`Math.pow()` - élever à une puissance

Calcule un nombre à la puissance d'un autre : base^exposant.

```
console.log(Math.pow(2, 3)); // 8 (2 puissance 3)
console.log(Math.pow(5, 2)); // 25 (5 au carré)
console.log(Math.pow(10, -2)); // 0.01
```

`Math.min()` - plus petite valeur

Trouve le plus petit nombre dans une liste.

```
console.log(Math.min(1, 5, 3, 9, 2)); // 1
console.log(Math.min(-10, -5, 0)); // -10
console.log(Math.min()); // Infinity
```

`Math.max()` - plus grande valeur

Trouve le plus grand nombre dans une liste.

```
console.log(Math.max(1, 5, 3, 9, 2)); // 9
console.log(Math.max(-10, -5, 0)); // 0
console.log(Math.max()); // -Infinity
```

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

Arrondit vers le haut. Donne toujours un entier égal ou plus grand.

```
console.log(Math.ceil(4.2)); // 5
console.log(Math.ceil(4.9)); // 5
console.log(Math.ceil(-4.2)); // -4
console.log(Math.ceil(5)); // 5
```

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

Arrondit vers le bas. Donne toujours un entier égal ou plus petit.

```
console.log(Math.floor(4.2)); // 4
console.log(Math.floor(4.9)); // 4
console.log(Math.floor(-4.2)); // -5
console.log(Math.floor(5)); // 5
```

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

Arrondit à l'entier le plus proche. Si >= 0.5, monte. Sinon, descend.

```
console.log(Math.round(4.4)); // 4
console.log(Math.round(4.5)); // 5
console.log(Math.round(4.6)); // 5
console.log(Math.round(-4.5)); // -4
```

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

Enlève la partie après la virgule. Garde juste le nombre entier.

```
console.log(Math.trunc(4.9)); // 4
console.log(Math.trunc(-4.9)); // -4
console.log(Math.trunc(4.1)); // 4
console.log(Math.trunc(-4.1)); // -4
```

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

Calcule la racine carrée d'un nombre.

```
console.log(Math.sqrt(9)); // 3
console.log(Math.sqrt(16)); // 4
console.log(Math.sqrt(2)); // 1.4142135623730951
console.log(Math.sqrt(-1)); // NaN (pas possible pour les nombr
```

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

Donne un nombre aléatoire entre 0 et 1. Utile pour créer du hasard.

```
console.log(Math.random()); // ex: 0.123456789
```

```
// Un nombre aléatoire entre 0 et 9
const randomInt = Math.floor(Math.random() * 10);

// Un nombre aléatoire entre 1 et 6 (comme un dé)
const dice = Math.floor(Math.random() * 6) + 1;

// Un nombre aléatoire entre 10 et 20
const randomBetween = Math.random() * 10 + 10;
```

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

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

Convertit un objet JavaScript en texte JSON. Utile pour envoyer des données ou les sauvegarder.

```
const personne = { nom: 'Alice', age: 25 };
const json = JSON.stringify(personne);
console.log(json); // {"nom":"Alice","age":25}

// Avec indentation pour lisibilité
const jsonFormaté = JSON.stringify(personne, null, 2);
```

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

Convertit du texte JSON en objet JavaScript. Inverse de stringify().

```
const json = '{"nom":"Alice","age":25}';
const personne = JSON.parse(json);
```

```
console.log(personne.nom); // Alice
console.log(personne.age); // 25
```

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

Coupe une chaîne en morceau selon un séparateur. Retourne un tableau.

```
const texte = 'apple,banana,orange';
const fruits = texte.split(',');
console.log(fruits); // ['apple', 'banana', 'orange']

const phrase = 'Bonjour tout le monde';
const mots = phrase.split(' ');
console.log(mots); // ['Bonjour', 'tout', 'le', 'monde']
```

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

Enlève les espaces au début et/ou à la fin d'une chaîne.

```
const texte = '  Bonjour  ';
console.log(texte.trim()); // 'Bonjour'
console.log(texte.trimStart()); // 'Bonjour  '
console.log(texte.trimEnd()); // '  Bonjour'
```

`padStart()` et `padEnd()` - aligner le contenu dans

une chaîne de caractères

Ajoute des caractères au début ou à la fin pour atteindre une longueur désirée.

```
const nombre = '5';
console.log(nombre.padStart(3, '0')); // '005'
console.log(nombre.padEnd(3, '.')); // '5..'

const nom = 'Alice';
console.log(nom.padStart(10, '-')); // '-----Alice'
```

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

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

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

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

Affichent des messages avec différents niveaux. Utile pour filtrer les messages importants.

```
console.info('Information'); // Message blanc
console.warn('Attention'); // Message jaune
console.error('Erreur'); // Message rouge
```

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

Affiche un tableau ou un objet sous forme de table. Très lisible.

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

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

Mesure le temps d'exécution d'un code.

```
console.time('monCalcul');
// Faire du travail...
for (let i = 0; i < 1000000; i++) {}
console.timeEnd('monCalcul'); // monCalcul: 2.5ms
```

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

`forEach` - parcourir les éléments d'un tableau

Boucle sur chaque élément. Ne retourne rien.

```
const fruits = ['apple', 'banana', 'orange'];
fruits.forEach((fruit, index) => {
  console.log(index, fruit);
```

```
});
```

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

Retourne un iterateur avec l'index et la valeur.

```
const fruits = ['apple', 'banana'];
for (const [index, value] of fruits.entries()) {
  console.log(index, value);
}
```

`in` - parcourir les clés d'un tableau

Boucle sur les index (clés) du tableau.

```
const fruits = ['apple', 'banana', 'orange'];
for (const index in fruits) {
  console.log(index);
}
```

`of` - parcourir les valeurs d'un tableau

Boucle sur les valeurs du tableau.

```
const fruits = ['apple', 'banana', 'orange'];
for (const fruit of fruits) {
  console.log(fruit);
}
```

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

Retourne le premier élément qui passe le test.

```
const nombres = [10, 20, 30, 40];
const resultat = nombres.find(x => x > 25);
console.log(resultat); // 30
```

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

Retourne l'index du premier élément qui passe le test.

```
const nombres = [10, 20, 30, 40];
const index = nombres.findIndex(x => x > 25);
console.log(index); // 2
```

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

Trouve la position d'une valeur dans un tableau.

```
const fruits = ['apple', 'banana', 'orange', 'banana'];
console.log(fruits.indexOf('banana')); // 1
console.log(fruits.lastIndexOf('banana')); // 3
```

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

Ajoute ou enlève des éléments.

```
const arr = [1, 2, 3];
arr.push(4); // Ajoute à la fin => [1, 2, 3, 4]
arr.pop(); // Enlève à la fin => [1, 2, 3]
arr.unshift(0); // Ajoute au début => [0, 1, 2, 3]
arr.shift(); // Enlève au début => [1, 2, 3]
```

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

Crée une copie d'une partie du tableau.

```
const arr = [1, 2, 3, 4, 5];
const part = arr.slice(1, 4); // [2, 3, 4]
console.log(arr); // [1, 2, 3, 4, 5] (original inchangé)
```

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

Modifie le tableau en supprimant ou ajoutant des éléments.

```
const arr = [1, 2, 3, 4, 5];
arr.splice(2, 1); // Enlève 1 élément à l'index 2
console.log(arr); // [1, 2, 4, 5]
```

`concat()` - joindre deux tableaux

Crée un nouveau tableau en combinant plusieurs tableaux.

```
const arr1 = [1, 2];
const arr2 = [3, 4];
const result = arr1.concat(arr2);
console.log(result); // [1, 2, 3, 4]
```

`join()` - joindre des chaînes de caractères

Combine tous les éléments en une seule chaîne avec un séparateur.

```
const fruits = ['apple', 'banana', 'orange'];
console.log(fruits.join(', ')); // "apple, banana, orange"
console.log(fruits.join('-')); // "apple-banana-orange"
```

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

Retourne les clés ou les valeurs d'un objet.

```
const obj = { nom: 'Alice', age: 25 };
console.log(Object.keys(obj)); // ['nom', 'age']
console.log(Object.values(obj)); // ['Alice', 25]
```

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

Retourne true ou false selon si la valeur existe.

```
const fruits = ['apple', 'banana', 'orange'];
console.log(fruits.includes('banana')); // true
console.log(fruits.includes('grape')); // false
```

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

every() = tous passent le test. some() = au moins un passe.

```
const nombres = [2, 4, 6];
console.log(nombres.every(x => x % 2 === 0)); // true
console.log(nombres.some(x => x > 5)); // true
```

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

Remplit un tableau avec une valeur.

```
const arr = [1, 2, 3, 4];
arr.fill(0);
console.log(arr); // [0, 0, 0, 0]

const arr2 = new Array(3).fill('x');
console.log(arr2); // ['x', 'x', 'x']
```

`flat()` - aplatir un tableau

Aplatit un tableau imbriqué.

```
const arr = [1, [2, 3], [4, [5, 6]]];
console.log(arr.flat()); // [1, 2, 3, 4, [5, 6]]
console.log(arr.flat(2)); // [1, 2, 3, 4, 5, 6]
```

`sort()` - pour trier un tableau

Trie les éléments du tableau.

```
const nombres = [30, 10, 20];
nombres.sort((a, b) => a - b);
console.log(nombres); // [10, 20, 30]

const fruits = ['banana', 'apple', 'orange'];
fruits.sort();
console.log(fruits); // ['apple', 'banana', 'orange']
```

Transforme chaque élément avec une fonction.

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

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

Garde seulement les éléments qui passent le test.

```
const nombres = [1, 2, 3, 4, 5];
const pairs = nombres.filter(x => x % 2 === 0);
console.log(pairs); // [2, 4]
```

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

Groupe les éléments par catégorie.

```
const personnes = [
  { nom: 'Alice', age: 25 },
  { nom: 'Bob', age: 25 },
```

```
  { nom: 'Charlie', age: 30 }
];
const groupes = Object.groupBy(personnes, p => p.age);
// { 25: [{nom: 'Alice', age: 25}, {nom: 'Bob', age: 25}], 30:
```

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

Map() puis flat() en une seule étape.

```
const nombres = [1, 2, 3];
const result = nombres.flatMap(x => [x, x * 2]);
console.log(result); // [1, 2, 2, 4, 3, 6]
```

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

Combine tous les éléments en une seule valeur.

```
const nombres = [1, 2, 3, 4];
const somme = nombres.reduce((total, x) => total + x, 0);
console.log(somme); // 10
```

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

Inverse l'ordre des éléments.

```
const arr = [1, 2, 3, 4];
arr.reverse();
console.log(arr); // [4, 3, 2, 1]
```

Recherche dans les structures de données

Trouver dans un tableau simple

Vérifier la présence d'une valeur

```
const nombres = [10, 20, 30, 40, 50];

// Avec includes() – retourne true/false
console.log(nombres.includes(30)); // true
console.log(nombres.includes(99)); // false

// Avec indexOf() – retourne l'index ou -1
console.log(nombres.indexOf(30)); // 2
console.log(nombres.indexOf(99)); // -1

// Avec find() – retourne l'élément ou undefined
const resultat = nombres.find(n => n > 25);
console.log(resultat); // 30

// Avec some() – vérifie si au moins un élément correspond
console.log(nombres.some(n => n > 40)); // true
console.log(nombres.some(n => n > 100)); // false
```

Trouver tous les éléments correspondants

```
const nombres = [10, 20, 30, 40, 50];

// Avec filter() – retourne un nouveau tableau avec tous les él
const grands = nombres.filter(n => n > 25);
console.log(grands); // [30, 40, 50]

const pairs = nombres.filter(n => n % 2 === 0);
console.log(pairs); // [10, 20, 30, 40, 50]
```

Trouver l'index d'un élément

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

// Premier index
console.log(fruits.indexOf('banane')); // 1

// Dernier index
console.log(fruits.lastIndexOf('banane')); // 3

// Index avec condition personnalisée
const index = fruits.findIndex(f => f.startsWith('o'));
console.log(index); // 2 (orange)
```

```
const personne = {
    nom: 'Alice',
    age: 25,
    ville: 'Genève',
    emploi: 'développeuse'
};

// Accès direct
console.log(personne.nom); // 'Alice'
console.log(personne['age']); // 25

// Vérifier si une propriété existe
console.log('nom' in personne); // true
console.log('pays' in personne); // false
console.log(personne.hasOwnProperty('ville')); // true

// Avec valeur par défaut si la propriété n'existe pas
const pays = personne.pays ?? 'Suisse';
console.log(pays); // 'Suisse'
```

Chercher dans les clés ou valeurs

```
const personne = {
    nom: 'Alice',
    age: 25,
    ville: 'Genève'
};

// Obtenir toutes les clés
const cles = Object.keys(personne);
console.log(cles); // ['nom', 'age', 'ville']

// Vérifier si une clé existe
console.log(cles.includes('age')); // true

// Obtenir toutes les valeurs
const valeurs = Object.values(personne);
console.log(valeurs); // ['Alice', 25, 'Genève']

// Vérifier si une valeur existe
console.log(valeurs.includes('Genève')); // true

// Obtenir paires clé-valeur
const entrees = Object.entries(personne);
console.log(entrees);
// [['nom', 'Alice'], ['age', 25], ['ville', 'Genève']]

// Chercher une clé qui a une certaine valeur
const cleAge = Object.entries(personne)
    .find(([cle, valeur]) => valeur === 25);
console.log(cleAge); // ['age', 25]
```

Trouver dans un tableau d'objets

Trouver un objet spécifique

```
const etudiants = [
    { id: 1, nom: 'Alice', age: 20, note: 15 },
    { id: 2, nom: 'Bob', age: 22, note: 12 },
    { id: 3, nom: 'Charlie', age: 21, note: 18 },
    { id: 4, nom: 'Diana', age: 20, note: 16 }
];

// Trouver le premier objet qui correspond
const bob = etudiants.find(e => e.nom === 'Bob');
console.log(bob); // { id: 2, nom: 'Bob', age: 22, note: 12 }

// Trouver par ID
const etudiant3 = etudiants.find(e => e.id === 3);
console.log(etudiant3); // { id: 3, nom: 'Charlie', age: 21, no

// Trouver l'index
const indexBob = etudiants.findIndex(e => e.nom === 'Bob');
console.log(indexBob); // 1
```

Trouver tous les objets correspondants

```
const etudiants = [
    { id: 1, nom: 'Alice', age: 20, note: 15 },
    { id: 2, nom: 'Bob', age: 22, note: 12 },
    { id: 3, nom: 'Charlie', age: 21, note: 18 },
    { id: 4, nom: 'Diana', age: 20, note: 16 }
];

// Tous les étudiants de 20 ans
const vingtAns = etudiants.filter(e => e.age === 20);
console.log(vingtAns);
// [{ id: 1, nom: 'Alice', age: 20, note: 15 },
//  { id: 4, nom: 'Diana', age: 20, note: 16 }]
```

```
// Tous ceux qui ont une note supérieure à 14
const bonnesNotes = etudiants.filter(e => e.note > 14);
console.log(bonnesNotes);
// [{ id: 1, nom: 'Alice', age: 20, note: 15 },
//  { id: 3, nom: 'Charlie', age: 21, note: 18 },
//  { id: 4, nom: 'Diana', age: 20, note: 16 }]

// Vérifier si au moins un étudiant a 18/20
const auMoinsUnExcellent = etudiants.some(e => e.note >= 18);
console.log(auMoinsUnExcellent); // true

// Vérifier si tous les étudiants ont plus de 10
const tousReussis = etudiants.every(e => e.note > 10);
console.log(tousReussis); // true
```

Extraire des valeurs spécifiques

```
const etudiants = [
    { id: 1, nom: 'Alice', age: 20, note: 15 },
    { id: 2, nom: 'Bob', age: 22, note: 12 },
    { id: 3, nom: 'Charlie', age: 21, note: 18 }
];

// Obtenir seulement les noms
const noms = etudiants.map(e => e.nom);
console.log(noms); // ['Alice', 'Bob', 'Charlie']

// Obtenir seulement les notes
const notes = etudiants.map(e => e.note);
console.log(notes); // [15, 12, 18]

// Vérifier si un nom existe dans le tableau
const aliceExiste = etudiants.some(e => e.nom === 'Alice');
console.log(aliceExiste); // true
```

```
// Obtenir la note maximale
const noteMax = Math.max(...etudiants.map(e => e.note));
console.log(noteMax); // 18
```

Recherche complexe avec conditions multiples

```
const etudiants = [
    { id: 1, nom: 'Alice', age: 20, note: 15, ville: 'Genève' }
    { id: 2, nom: 'Bob', age: 22, note: 12, ville: 'Lausanne' }
    { id: 3, nom: 'Charlie', age: 21, note: 18, ville: 'Genève'
    { id: 4, nom: 'Diana', age: 20, note: 16, ville: 'Berne' }
];

// Étudiants de Genève avec note > 14
const genevoisBonnesNotes = etudiants.filter(
    e => e.ville === 'Genève' && e.note > 14
);
console.log(genevoisBonnesNotes);
// [{ id: 1, nom: 'Alice', age: 20, note: 15, ville: 'Genève' }
//  { id: 3, nom: 'Charlie', age: 21, note: 18, ville: 'Genève'

// Étudiants de 20 ans OU avec note >= 18
const critereSpecial = etudiants.filter(
    e => e.age === 20 || e.note >= 18
);
console.log(critereSpecial);
```

Trouver dans un objet contenant des tableaux

Accéder aux tableaux dans l'objet

```
const classe = {
    nom: 'Informatique 3A',
```

```
    professeur: 'M. Dupont',
    etudiants: [
        { nom: 'Alice', note: 15 },
        { nom: 'Bob', note: 12 },
        { nom: 'Charlie', note: 18 }
    ],
    matieres: ['JavaScript', 'Python', 'SQL']
};

// Accéder à un tableau
console.log(classe.etudiants); // Tout le tableau d'étudiants
console.log(classe.matieres); // ['JavaScript', 'Python', 'SQL'

// Trouver dans un tableau de l'objet
const alice = classe.etudiants.find(e => e.nom === 'Alice');
console.log(alice); // { nom: 'Alice', note: 15 }

// Vérifier si une matière existe
console.log(classe.matieres.includes('Python')); // true

// Filtrer les étudiants
const bonnesNotes = classe.etudiants.filter(e => e.note > 14);
console.log(bonnesNotes);
// [{ nom: 'Alice', note: 15 }, { nom: 'Charlie', note: 18 }]
```

Objets avec tableaux imbriqués

```
const ecole = {
    nom: 'EMF',
    classes: [
        {
            niveau: '3A',
            etudiants: [
                { nom: 'Alice', note: 15 },
                { nom: 'Bob', note: 12 }
            ]
```

```
        },
        {
            niveau: '3B',
            etudiants: [
                { nom: 'Charlie', note: 18 },
                { nom: 'Diana', note: 16 }
            ]
        }
    ]
};

// Trouver une classe spécifique
const classe3A = ecole.classes.find(c => c.niveau === '3A');
console.log(classe3A);

// Trouver un étudiant dans une classe spécifique
const bobEn3A = ecole.classes
    .find(c => c.niveau === '3A')
    .etudiants.find(e => e.nom === 'Bob');
console.log(bobEn3A); // { nom: 'Bob', note: 12 }

// Obtenir tous les étudiants de toutes les classes
const tousEtudiants = ecole.classes
    .flatMap(c => c.etudiants);
console.log(tousEtudiants);
// [{ nom: 'Alice', note: 15 }, { nom: 'Bob', note: 12 },
//  { nom: 'Charlie', note: 18 }, { nom: 'Diana', note: 16 }]

// Trouver tous les étudiants avec note > 14
const excellents = ecole.classes
    .flatMap(c => c.etudiants)
    .filter(e => e.note > 14);
console.log(excellents);
// [{ nom: 'Alice', note: 15 }, { nom: 'Charlie', note: 18 },
//  { nom: 'Diana', note: 16 }]

// Vérifier si au moins une classe a un étudiant nommé Charlie
const charlieExiste = ecole.classes
```

```
    .some(c => c.etudiants.some(e => e.nom === 'Charlie'));
console.log(charlieExiste); // true
```

Cas pratique : données de motos (exercices du module 323)

```
const motos = [
    {
        marque: 'Yamaha',
        modeles: [
            { nom: 'MT-07', puissance: 75, prix: 8000 },
            { nom: 'R1', puissance: 200, prix: 18000 }
        ]
    },
    {
        marque: 'Honda',
        modeles: [
            { nom: 'CB650R', puissance: 95, prix: 9500 },
            { nom: 'CBR1000RR', puissance: 189, prix: 17000 }
        ]
    }
];

// Trouver la marque Yamaha
const yamaha = motos.find(m => m.marque === 'Yamaha');
console.log(yamaha);

// Trouver la moto MT-07
const mt07 = motos
    .find(m => m.marque === 'Yamaha')
    .modeles.find(mod => mod.nom === 'MT-07');
console.log(mt07); // { nom: 'MT-07', puissance: 75, prix: 8000

// Obtenir tous les modèles (tous les objets moto)
const tousModeles = motos.flatMap(m => m.modeles);
```

```
console.log(tousModeles);

// Trouver toutes les motos avec plus de 100 chevaux
const puissantes = motos
    .flatMap(m => m.modeles)
    .filter(mod => mod.puissance > 100);
console.log(puissantes);

// Trouver toutes les motos de moins de 10000 euros
const abordables = motos
    .flatMap(m => m.modeles)
    .filter(mod => mod.prix < 10000);
console.log(abordables);

// Vérifier si au moins une marque a un modèle R1
const r1Existe = motos
    .some(m => m.modeles.some(mod => mod.nom === 'R1'));
console.log(r1Existe); // true

// Obtenir la moto la plus chère
const plusChere = motos
    .flatMap(m => m.modeles)
    .reduce((max, mod) => mod.prix > max.prix ? mod : max);
console.log(plusChere); // { nom: 'R1', puissance: 200, prix: 1
```

Résumé des méthodes de recherche

Structure	Méthode	Usage
Tableau simple	`includes()`	Vérifier présence (true/false)
	`indexOf()`	Trouver l'index d'une valeur
	`find()`	Trouver le premier élément
	`filter()`	Trouver tous les éléments

	`some()`	Au moins un correspond
	`every()`	Tous correspondent
Objet	`.propriete` ou `['cle']`	Accès direct
	`in` ou `hasOwnProperty()`	Vérifier existence clé
	`Object.keys()`	Obtenir les clés
	`Object.values()`	Obtenir les valeurs
	`Object.entries()`	Obtenir paires clé-valeur
Tableau d'objets	`find()`	Trouver un objet
	`filter()`	Trouver plusieurs objets
	`map()`	Extraire une propriété
	`some()` / `every()`	Tests conditionnels
Objet avec tableaux	`.propriete` puis méthodes tableau	Accès puis recherche
	`flatMap()`	Aplatir puis transformer
	Chaînage `.find().find()`	Recherche imbriquée

Transformations de structures de données

JavaScript offre plusieurs méthodes pour arrondir selon vos besoins :

```
const nombre = 4.567;

// Arrondir à l'entier le plus proche
console.log(Math.round(nombre)); // 5
console.log(Math.round(4.4)); // 4

// Toujours arrondir vers le haut
console.log(Math.ceil(nombre)); // 5
console.log(Math.ceil(4.1)); // 5

// Toujours arrondir vers le bas
console.log(Math.floor(nombre)); // 4
```

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

// Supprimer la partie décimale (tronquer)
console.log(Math.trunc(nombre)); // 4
console.log(Math.trunc(-4.9)); // -4

// Arrondir à N décimales
const prix = 19.99567;
console.log(prix.toFixed(2)); // "19.99" (string!)
console.log(Number(prix.toFixed(2))); // 19.99 (number)
console.log(Math.round(prix * 100) / 100); // 19.99

// Arrondir à N décimales (autre méthode)
const arrondir = (nombre, decimales) => {
    const facteur = Math.pow(10, decimales);
    return Math.round(nombre * facteur) / facteur;
};
console.log(arrondir(4.567, 2)); // 4.57
console.log(arrondir(4.567, 1)); // 4.6
```

Transformer un tableau en objet

Avec `Object.fromEntries()`

```
// À partir d'un tableau de paires [clé, valeur]
const paires = [
    ['nom', 'Alice'],
    ['age', 25],
    ['ville', 'Genève']
];
const objet = Object.fromEntries(paires);
console.log(objet);
// { nom: 'Alice', age: 25, ville: 'Genève' }

// À partir d'un tableau avec map()
```

```
const fruits = ['pomme', 'banane', 'orange'];
const objFruits = Object.fromEntries(
    fruits.map((fruit, index) => [index, fruit])
);
console.log(objFruits);
// { 0: 'pomme', 1: 'banane', 2: 'orange' }

// Créer un objet avec des clés personnalisées
const etudiants = [
    { id: 1, nom: 'Alice' },
    { id: 2, nom: 'Bob' },
    { id: 3, nom: 'Charlie' }
];
const objEtudiants = Object.fromEntries(
    etudiants.map(e => [e.id, e])
);
console.log(objEtudiants);
// {
//   1: { id: 1, nom: 'Alice' },
//   2: { id: 2, nom: 'Bob' },
//   3: { id: 3, nom: 'Charlie' }
// }
```

```
// Transformer un tableau en objet avec reduce()
const fruits = ['pomme', 'banane', 'orange'];
const objFruits = fruits.reduce((obj, fruit, index) => {
    obj[fruit] = index;
    return obj;
}, {});
console.log(objFruits);
// { pomme: 0, banane: 1, orange: 2 }

// Grouper par propriété
const etudiants = [
    { nom: 'Alice', ville: 'Genève' },
    { nom: 'Bob', ville: 'Lausanne' },
```

```
    { nom: 'Charlie', ville: 'Genève' }
];
const parVille = etudiants.reduce((obj, etudiant) => {
    const ville = etudiant.ville;
    if (!obj[ville]) {
        obj[ville] = [];
    }
    obj[ville].push(etudiant);
    return obj;
}, {});
console.log(parVille);
// {
//   Genève: [{ nom: 'Alice', ville: 'Genève' }, { nom: 'Charli
//   Lausanne: [{ nom: 'Bob', ville: 'Lausanne' }]
// }
```

Avec `Object.groupBy()` (moderne)

```
const etudiants = [
    { nom: 'Alice', age: 20, ville: 'Genève' },
    { nom: 'Bob', age: 22, ville: 'Lausanne' },
    { nom: 'Charlie', age: 20, ville: 'Genève' }
];

// Grouper par ville
const parVille = Object.groupBy(etudiants, e => e.ville);
console.log(parVille);
// {
//   Genève: [{ nom: 'Alice', ... }, { nom: 'Charlie', ... }],
//   Lausanne: [{ nom: 'Bob', ... }]
// }

// Grouper par âge
const parAge = Object.groupBy(etudiants, e => e.age);
console.log(parAge);
```

```
// {
//   20: [{ nom: 'Alice', ... }, { nom: 'Charlie', ... }],
//   22: [{ nom: 'Bob', ... }]
// }
```

Transformer un objet en tableau

```
const personne = {
    nom: 'Alice',
    age: 25,
    ville: 'Genève'
};

// Obtenir un tableau de paires [clé, valeur]
const entrees = Object.entries(personne);
console.log(entrees);
// [['nom', 'Alice'], ['age', 25], ['ville', 'Genève']]

// Transformer en tableau d'objets
const tableau = Object.entries(personne).map(([cle, valeur]) => 
    propriete: cle,
    valeur: valeur
}));
console.log(tableau);
// [
//   { propriete: 'nom', valeur: 'Alice' },
//   { propriete: 'age', valeur: 'Alice' },
//   { propriete: 'ville', valeur: 'Genève' }
// ]
```

Avec `Object.keys()` et `Object.values()`

```
const notes = {
```

```
    Math: 15,
    Français: 12,
    Anglais: 16
};

// Obtenir seulement les clés
const matieres = Object.keys(notes);
console.log(matieres); // ['Math', 'Français', 'Anglais']

// Obtenir seulement les valeurs
const scores = Object.values(notes);
console.log(scores); // [15, 12, 16]

// Calculer la moyenne
const moyenne = scores.reduce((sum, note) => sum + note, 0) / s
console.log(moyenne); // 14.33...

// Créer un tableau d'objets
const tableauNotes = Object.entries(notes).map(([matiere, note]
    matiere,
    note
}));
console.log(tableauNotes);
// [
//   { matiere: 'Math', note: 15 },
//   { matiere: 'Français', note: 12 },
//   { matiere: 'Anglais', note: 16 }
// ]
```

Aplatir des structures imbriquées avec `flat()`

Tableaux imbriqués simples

```
// Aplatir un niveau
const tab1 = [1, 2, [3, 4]];
```

```
console.log(tab1.flat()); // [1, 2, 3, 4]

// Aplatir plusieurs niveaux
const tab2 = [1, [2, [3, [4, [5]]]]];
console.log(tab2.flat(1)); // [1, 2, [3, [4, [5]]]]
console.log(tab2.flat(2)); // [1, 2, 3, [4, [5]]]
console.log(tab2.flat(3)); // [1, 2, 3, 4, [5]]
console.log(tab2.flat(Infinity)); // [1, 2, 3, 4, 5]

// Supprimer les trous vides
const tab3 = [1, 2, , 4, 5];
console.log(tab3.flat()); // [1, 2, 4, 5]
```

Aplatir des tableaux de tableaux

```
const classes = [
    ['Alice', 'Bob'],
    ['Charlie', 'Diana'],
    ['Eve', 'Frank']
];

const tousEtudiants = classes.flat();
console.log(tousEtudiants);
// ['Alice', 'Bob', 'Charlie', 'Diana', 'Eve', 'Frank']

// Avec des objets imbriqués
const ecoles = [
    [
        { nom: 'Alice', note: 15 },
        { nom: 'Bob', note: 12 }
    ],
    [
        { nom: 'Charlie', note: 18 }
    ]
];
```

```
const tousEleves = ecoles.flat();
console.log(tousEleves);
// [
//   { nom: 'Alice', note: 15 },
//   { nom: 'Bob', note: 12 },
//   { nom: 'Charlie', note: 18 }
// ]
```

Transformer et aplatir avec `flatMap()`

`flatMap()` combine `map()` + `flat()` en une seule opération (plus performant).

```
// Doubler et aplatir
const nombres = [1, 2, 3];
const resultat = nombres.flatMap(x => [x, x * 2]);
console.log(resultat); // [1, 2, 2, 4, 3, 6]

// Découper des phrases en mots
const phrases = ['Hello world', 'Bonjour tout le monde'];
const mots = phrases.flatMap(phrase => phrase.split(' '));
console.log(mots);
// ['Hello', 'world', 'Bonjour', 'tout', 'le', 'monde']

// Filtrer et transformer
const nombres2 = [1, 2, 3, 4, 5];
const resultat2 = nombres2.flatMap(x =>
    x % 2 === 0 ? [x * 2] : []
);
console.log(resultat2); // [4, 8] (seulement les pairs doublés)
```

Cas pratique : données imbriquées

```
const marques = [
    {
        nom: 'Yamaha',
        modeles: [
            { nom: 'MT-07', prix: 8000 },
            { nom: 'R1', prix: 18000 }
        ]
    },
    {
        nom: 'Honda',
        modeles: [
            { nom: 'CB650R', prix: 9500 },
            { nom: 'CBR1000RR', prix: 17000 }
        ]
    }
];

// Obtenir tous les modèles avec leur marque
const tousModeles = marques.flatMap(marque =>
    marque.modeles.map(modele => ({
        marque: marque.nom,
        modele: modele.nom,
        prix: modele.prix
    }))
);
console.log(tousModeles);
// [
//   { marque: 'Yamaha', modele: 'MT-07', prix: 8000 },
//   { marque: 'Yamaha', modele: 'R1', prix: 18000 },
//   { marque: 'Honda', modele: 'CB650R', prix: 9500 },
//   { marque: 'Honda', modele: 'CBR1000RR', prix: 17000 }
// ]

// Filtrer les modèles de moins de 10000 euros
const abordables = marques.flatMap(marque =>
    marque.modeles
        .filter(modele => modele.prix < 10000)
```

```
        .map(modele => ({
            marque: marque.nom,
            modele: modele.nom,
            prix: modele.prix
        }))
);
console.log(abordables);
// [
//   { marque: 'Yamaha', modele: 'MT-07', prix: 8000 },
//   { marque: 'Honda', modele: 'CB650R', prix: 9500 }
// ]
```

Transformations complexes de structures imbriquées

Restructurer complètement les données

```
const donneesBrutes = [
    {
        ville: 'Genève',
        cantons: 'GE',
        etudiants: [
            { nom: 'Alice', notes: [15, 16, 14] },
            { nom: 'Bob', notes: [12, 13, 11] }
        ]
    },
    {
        ville: 'Lausanne',
        cantons: 'VD',
        etudiants: [
            { nom: 'Charlie', notes: [18, 17, 19] }
        ]
    }
];
```

```
// Aplatir complètement avec toutes les informations
const donneesAplaties = donneesBrutes.flatMap(localisation =>
    localisation.etudiants.map(etudiant => ({
        ville: localisation.ville,
        canton: localisation.cantons,
        nom: etudiant.nom,
        noteMin: Math.min(...etudiant.notes),
        noteMax: Math.max(...etudiant.notes),
        moyenne: etudiant.notes.reduce((a, b) => a + b, 0) / et
    }))
);
console.log(donneesAplaties);
// [
//   { ville: 'Genève', canton: 'GE', nom: 'Alice', noteMin: 14
//   { ville: 'Genève', canton: 'GE', nom: 'Bob', noteMin: 11,
//   { ville: 'Lausanne', canton: 'VD', nom: 'Charlie', noteMin
// ]
```

Inverser la structure (pivot)

```
const donnees = [
    { produit: 'A', janvier: 100, fevrier: 120 },
    { produit: 'B', janvier: 80, fevrier: 90 }
];

// Transformer en format long
const formatLong = donnees.flatMap(ligne =>
    Object.entries(ligne)
        .filter(([cle]) => cle !== 'produit')
        .map(([mois, ventes]) => ({
            produit: ligne.produit,
            mois: mois,
            ventes: ventes
        }))
);
console.log(formatLong);
```

```
// [
//   { produit: 'A', mois: 'janvier', ventes: 100 },
//   { produit: 'A', mois: 'fevrier', ventes: 120 },
//   { produit: 'B', mois: 'janvier', ventes: 80 },
//   { produit: 'B', mois: 'fevrier', ventes: 90 }
// ]
```

Dénormaliser des données relationnelles

```
const utilisateurs = [
    { id: 1, nom: 'Alice' },
    { id: 2, nom: 'Bob' }
];

const commandes = [
    { id: 101, userId: 1, produit: 'Laptop' },
    { id: 102, userId: 1, produit: 'Souris' },
    { id: 103, userId: 2, produit: 'Clavier' }
];

// Joindre les données
const commandesCompletes = commandes.map(commande => ({
    ...commande,
    utilisateur: utilisateurs.find(u => u.id === commande.userI
}));
console.log(commandesCompletes);
// [
//   { id: 101, userId: 1, produit: 'Laptop', utilisateur: 'Ali
//   { id: 102, userId: 1, produit: 'Souris', utilisateur: 'Ali
//   { id: 103, userId: 2, produit: 'Clavier', utilisateur: 'Bo
// ]

// Grouper les commandes par utilisateur
const commandesParUtilisateur = utilisateurs.map(utilisateur =>
    ...utilisateur,
    commandes: commandes
```

```
        .filter(c => c.userId === utilisateur.id)
        .map(c => c.produit)
}));
console.log(commandesParUtilisateur);
// [
//   { id: 1, nom: 'Alice', commandes: ['Laptop', 'Souris'] },
//   { id: 2, nom: 'Bob', commandes: ['Clavier'] }
// ]
```

Nettoyer et normaliser des données

```
const donnesSales = [
    { nom: '  alice  ', age: '25', ville: 'genève' },
    { nom: 'BOB', age: '30', ville: 'LAUSANNE' },
    { nom: 'Charlie', age: 22, ville: 'Berne' }
];

const donneesNettoyees = donnesSales.map(personne => ({
    nom: personne.nom.trim().toLowerCase()
        .replace(/^./, c => c.toUpperCase()), // Première lettr
    age: Number(personne.age), // Convertir en nombre
    ville: personne.ville.toLowerCase()
        .replace(/^./, c => c.toUpperCase()) // Première lettre
}));
console.log(donneesNettoyees);
// [
//   { nom: 'Alice', age: 25, ville: 'Genève' },
//   { nom: 'Bob', age: 30, ville: 'Lausanne' },
//   { nom: 'Charlie', age: 22, ville: 'Berne' }
// ]
```

Conversions aller-retour : Tableau ↔ Objet

Cycle complet de transformation

```
// Point de départ : objet
const notes = {
    Math: 15,
    Français: 12,
    Anglais: 16,
    Histoire: 14
};

// 1. Objet → Tableau
const tableauNotes = Object.entries(notes);
console.log(tableauNotes);
// [['Math', 15], ['Français', 12], ['Anglais', 16], ['Histoire

// 2. Transformer (filtrer notes > 13)
const bonnesNotes = tableauNotes
    .filter(([matiere, note]) => note > 13)
    .map(([matiere, note]) => [matiere, note + 1]); // Bonus de

// 3. Tableau → Objet
const nouvelObjet = Object.fromEntries(bonnesNotes);
console.log(nouvelObjet);
// { Math: 16, Anglais: 17, Histoire: 15 }
```

Cas pratique : traitement de données CSV

```
// Données CSV simulées
const lignesCSV = [
    'nom,age,ville',
    'Alice,25,Genève',
    'Bob,30,Lausanne',
    'Charlie,22,Berne'
];

// 1. Parser le CSV en tableau d'objets
const [entetes, ...lignes] = lignesCSV.map(ligne => ligne.split
```

```
const donnees = lignes.map(valeurs =>
    Object.fromEntries(
        entetes.map((entete, i) => [entete, valeurs[i]])
    )
);
console.log(donnees);
// [
//   { nom: 'Alice', age: '25', ville: 'Genève' },
//   { nom: 'Bob', age: '30', ville: 'Lausanne' },
//   { nom: 'Charlie', age: '22', ville: 'Berne' }
// ]

// 2. Transformer (convertir âge en nombre)
const donneesTransformees = donnees.map(personne => ({
    ...personne,
    age: Number(personne.age)
}));

// 3. Reconvertir en CSV
const nouvellesLignes = [
    entetes.join(','),
    ...donneesTransformees.map(obj =>
        entetes.map(entete => obj[entete]).join(',')
    )
];
console.log(nouvellesLignes);
// [
//   'nom,age,ville',
//   'Alice,25,Genève',
//   'Bob,30,Lausanne',
//   'Charlie,22,Berne'
// ]
```

Résumé des méthodes de transformation

Transformation	Méthode principale	Exemple

Tableau → Objet	`Object.fromEntries()`	`Object.fromEntries([['a', 1]])`
	`reduce()`	`arr.reduce((o, v) => ({...o, [v]: 1}), {})`
Objet → Tableau	`Object.entries()`	`Object.entries({a: 1})` → `[['a', 1]]`
	`Object.keys()`	`Object.keys({a: 1})` → `['a']`
	`Object.values()`	`Object.values({a: 1})` → `[1]`
Aplatir	`flat()`	`[[1, 2], [3]].flat()` → `[1, 2, 3]`
	`flatMap()`	`[1, 2].flatMap(x => [x, x*2])`
Grouper	`Object.groupBy()`	`Object.groupBy(arr, item => item.type)`
	`reduce()`	Construire manuellement les groupes
Arrondir	`Math.round()`	Arrondir à l'entier le plus proche
	`toFixed(n)`	Arrondir à n décimales (retourne string)
	`Math.round(x*10**n)/10**n`	Arrondir à n décimales (number)

``(backticks) - pour des expressions intelligentes

Permet d'insérer des variables directement dans une chaîne avec `${}`.

```
const nom = 'Alice';
const age = 25;
console.log(`Bonjour ${nom}, tu as ${age} ans`);
// Bonjour Alice, tu as 25 ans

const prix = 19.99;
console.log(`Prix: ${prix.toFixed(2)} euros`);
// Prix: 19.99 euros
```

`new Set()` - pour supprimer les doublons

Crée une collection qui n'accepte pas les doublons.

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

const fruitsUnique = [...new Set(['apple', 'banana', 'apple'])]
console.log(fruitsUnique); // ['apple', 'banana']
```

Standard

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

Sous forme d'expression de fonction

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

Sous forme d'expression de fonction anonyme

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

Sous forme raccourcie

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

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

Fonctions immédiatement invoquées (IIFE)

IIFE = Immediately Invoked Function Expressions.

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

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

ou

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

Ce document résume les concepts et les fonctions clés de JavaScript pour la programmation fonctionnelle. Les trois méthodes principales (`map()`, `filter()` et `reduce()`) sont essentielles pour transformer et analyser les données de manière efficace et lisible.

Les points clés à retenir :

- Utilisez `map()` pour transformer chaque élément d'un tableau
- Utilisez `filter()` pour garder seulement les éléments qui correspondent

- Utilisez `reduce()` pour combiner tous les éléments en une seule valeur
- Chaîner ces méthodes crée du code plus clair et puissant
- Les backticks rendent les chaînes plus lisibles
- Les Set sont utiles pour enlever les doublons

Avec la pratique, la programmation fonctionnelle devient naturelle et permet d'écrire du meilleur code.