D'autres méthodes de programmation fonctionnelle

La méthode `reduce()`

La méthode `reduce()` est utilisée le plus souvent pour réduire une liste à une seule valeur.

En fait, elle va parcourir chacun des éléments d'un tableau avec une fonction de rappel, et effectuer un calcul avec la valeur de chaque élément et un total intermédiaire, jusqu'à aboutir à un résultat final.

Commençons par un exemple simple :

```
const tableau = [1, 2, 3, 4];
const total = tableau.reduce((acc, curr) => acc + curr);
console.log(total); // 10
```

La méthode `reduce()` prend deux arguments : **une fonction de rappel** et **une valeur initiale**.

La fonction de rappel peut utiliser au minimum deux arguments et au maximum quatre.

Le premier argument passé à la fonction de rappel est l'accumulateur, c'est-à-dire le résultat intermédiaire. Par convention on utilise souvent `acc` pour accumulateur.

Le second argument passé à la fonction de rappel est l'élément en cours d'itération. Par convention on utilise souvent `curr` pour `current value` ou valeur courante.

Le troisième argument passé à la fonction de rappel est l' `index` de l'itération en cours.

Le quatrième argument passé à la fonction de rappel est le tableau sur lequel est utilisé `reduce()`.

Si aucune valeur initiale n'est passée en deuxième argument de `reduce()`, le premier élément du tableau est utilisé comme valeur initiale.

Prenons un second exemple, celui du panier :

```
const tableau = [{prix: 20, quantite: 2}, {prix: 42, quantite: 1}, {prix: 15, quantite: 3}];
const total = tableau.reduce((acc, curr) => acc += curr.quantite * curr.prix, 0);
console.log(total); // 127
```

Nous passons en deuxième argument de `reduce()` la valeur `0` comme valeur initiale.

Donc lors de la première exécution, nous avons :

```
acc = 0 + 2 * 20;
```

Pour la seconde itération nous avons :

```
acc = 40 + 1 * 42;
```

Pour la troisième itération nous avons :

```
acc = 82 + 3 * 15;
```

Comme il s'agit de la dernière itération, cette valeur est retournée. Nous obtenons le total de `127`.

Un exemple avancé de `reduce()` peut être utilisé par exemple pour grouper des objets selon la valeur d'une propriété, c'est ce qu'on appelle une fonction `groupby` :

```
function groupBy(tableau, propriete){
  return tableau.reduce( (acc, curr) => {
    const cle = curr[propriete];
    if(!acc[cle]){
      acc[cle] = [];
    }
    acc[cle].push(curr);
    return acc;
  }, {});
}
```

Cette fonction a deux paramètres : le premier est un tableau d'objets, et le deuxième la propriété que nous souhaitons utiliser pour regrouper les objets.

Nous utilisons la méthode `reduce()` auquel nous passons en premier argument la fonction de rappel, et en deuxième argument un objet littéral vide sur lequel nous allons effectuer le regroupement.

A chaque itération, nous récupérons la valeur de l'objet pour la propriété avec `curr[propriete]` qui va devenir une clé sur l'objet `acc`, c'est-à-dire l'accumulateur.

Si l'accumulateur n'a pas de valeur pour cette clé, cela signifie que la propriété n'existe pas et nous la créons en lui donnant comme valeur initiale un tableau vide.

Ensuite nous ajoutons l'objet courant dans le tableau.

Prenons un exemple :

```
function groupBy(tableau, propriete){
  return tableau.reduce( (acc, curr) => {
    const cle = curr[propriete];
    if(!acc[cle]){
```

```
      acc[cle] = [];
    }
    acc[cle].push(curr);
    return acc;
  }, {});
}

const tableau = [{prix: 25, name: "chaussons"}, {prix: 42, name: "pantalon"}, {prix: 25, name: "polo"}];
const objetsParPrix = groupBy(tableau, "prix");
// {
//   25: [{prix: 25, name: "chaussons"}, {prix: 25, name: "polo"}],
//   42: [{prix: 42, name: "pantalon"}]
// }
```

Vous pouvez ainsi réaliser des traitements avancés facilement !

La méthode `reduceRight()`

La méthode `reduceRight()` fonctionne exactement pareil que la méthode `reduce()` sauf que les itérations se font de la fin vers le début du tableau.

La méthode `flat()`

La méthode `flat()` permet de créer et de retourner un nouveau tableau contenant tous les éléments des tableaux imbriqués.

Comme son nom l'indique, cette méthode permet d'aplatir un tableau.

En fait, elle effectue une récursion sur les tableaux imbriqués et les concatène jusqu'au niveau passé en argument.

Prenons par exemple un tableau avec des éléments imbriqués sur plusieurs niveaux dans des tableaux :

```
const tableau = [1, 2, [3, 4], [[5], [6,7]]];
const tableau2 = tableau.flat();
const tableau3 = tableau.flat(2);

console.log(tableau2); // [1, 2, 3, 4, [5], [6, 7]]
console.log(tableau3); // [1, 2, 3, 4, 5, 6, 7]
```

La méthode `flatMap()`

La méthode `flatMap()` permet de combiner la méthode `map()` et la méthode `flat(1)` de manière optimisée pour la performance.

Par exemple :

```
const test = [1, 3, 5].flatMap(el => [el, el + 1]);
console.log(test); // [1, 2, 3, 4, 5, 6]
```

La méthode `every()`

La méthode `every()` permet d'effectuer un test pour l'ensemble des éléments d'un tableau.

Elle prend en argument une fonction de rappel qui va effectuer le test pour chaque élément. La fonction de rappel reçoit en argument l'élément itéré, son `index` et le tableau sur lequel est utilisé la méthode.

La méthode renverra `true` si tous les éléments passent le test et `false` sinon.

```
const tableau = [2, 5, 8, 4, 12];
const resultat = tableau.every(el => el < 10);
console.log(resultat); // false
```

La méthode `some()`

La méthode `some()` permet de vérifier qu'au moins un élément d'un tableau passe un test.

Elle prend en argument une fonction de rappel qui va effectuer le test pour chaque élément. La fonction de rappel reçoit en argument l'élément itéré, son `index` et le tableau sur lequel est utilisé la méthode.

La méthode renverra `true` si au moins un élément passe le test et `false` sinon.

```
const tableau = [2, 5, 8, 4, 12];
const resultat = tableau.some(el => el < 10);
console.log(resultat); // true
```