



# Immutabilité

## Concepts clés de la Programmation Fonctionnelle

Auteur / Enseignant :

Alexandre Leroux ([alex@shrp.dev](mailto:alex@shrp.dev)) - 2024

Toute reproduction, représentation, modification, publication, adaptation de tout ou partie des éléments de ce support de formation, quel que soit le moyen ou le procédé utilisé, est interdite, sauf autorisation écrite préalable de l'auteur.

Icônes et illustrations libres de droit : <https://www.flaticon.com>


Document à usage personnel. Ne pas diffuser.

---

## Valeur vs Référence

Selon les spécificités du langage de programmation employé, **il existe 2 façons de passer des arguments à une fonction** (au choix ou pas) :

1. **Par valeur** : l'argument passé en paramètre d'une fonction correspond à une copie de la valeur de la variable employée.
2. **Par référence** : l'argument passé en paramètre d'une fonction correspond à une référence vers la variable employée, cette dernière contenant la valeur.



Une variable de type **référence** contient une référence vers ses données logées dans la mémoire.

Une variable de type **valeur** contient ses données directement.

---

## Incidence d'un argument passé par référence à une fonction

Le passage d'argument à une fonction par référence représente un **risque d'effet de bord**.

En effet, avec ce mode de communication, la fonction est directement en capacité de modifier la valeur de la variable passée en argument, hors de son "scope" d'origine.

## Immutabilité

Pour éviter toute mauvaise surprise causée par un effet de bord, la Programmation Fonctionnelle préconise **l'emploi de données immuables**.

**Cela signifie que toutes les données manipulées sont constantes.**

Lorsqu'une valeur doit évoluer, on crée une nouvelle variable (en clonant l'originale) et on lui affecte la nouvelle valeur.

La valeur d'origine doit rester intacte.

```
const checkPassword = (str) =>  
  str==="abracadabra";
```

```
let openTheDoor = false;
```

```
const pwd = "abracadabra";
```

```
openTheDoor = checkPassword(pwd);
```

```
console.log(openTheDoor);  
//true
```

Mutabilité



```
const checkPassword = (str) =>  
  str==="abracadabra";
```

```
const openTheDoor = false;
```

```
const pwd = "abracadabra";
```

```
const openTheDoorWithPwd =  
  checkPassword(pwd);
```

```
console.log(openTheDoor); //false
```

```
console.log(  
  openTheDoorWithPwd); //true
```

Immutabilité



---

## Gestion des données "complexes"

Les données structurées telles que les **objets, array, set, map...** nécessitent une manipulation plus élaborée pour assurer leur immutabilité.

En effet, l'emploi d'une constante ne garantit pas l'immutabilité des données.

En effet, il est toujours possible de faire évoluer la valeur des attributs ou des données contenues dans un **Array** ou un **Object** déclaré sous forme de constante.

```
const johnDoe = {  
  name: "John Doe",  
  grade: 1  
};
```

```
johnDoe.name = "Foo Bar";  
johnDoe.grade = 3;
```

```
console.log(johnDoe);  
// [object Object]  
{  
  name: "Foo Bar",  
  grade: 3  
}
```

Le mot clé  
const  
n'empêche  
pas la  
mutation sur  
un Objet



## Mutation de valeurs sur une constante de type Object



```
const fruits = ["Lemon","Apple"];
```

```
fruits.push("Orange");
```

```
console.log(fruits);  
//  
// [object Array] (3)  
["Lemon","Apple","Orange"]
```

Le mot clé  
const  
n'empêche  
pas la  
mutation sur  
un Objet



## Mutation de valeurs sur une constante de type Array

```
const johnDoe = {  
  name: "John Doe",  
  grade: 1  
};  
  
const clone = {...johnDoe};//clone d'objet  
  
clone.grade = 2;//mutation de valeur sur le clone  
  
console.log(clone);  
// [object Object]  
{  
  name: "John Doe",  
  "grade": 2  
}
```



## Clone et manipulation d'objet

```
const fruits = ["Pomme","Poire","Ananas"];

const clone = [...fruits]; // clone d'array

clone.push("Pastèque");
//la valeur de clone a changé
["Pomme","Poire","Ananas","Pastèque"]

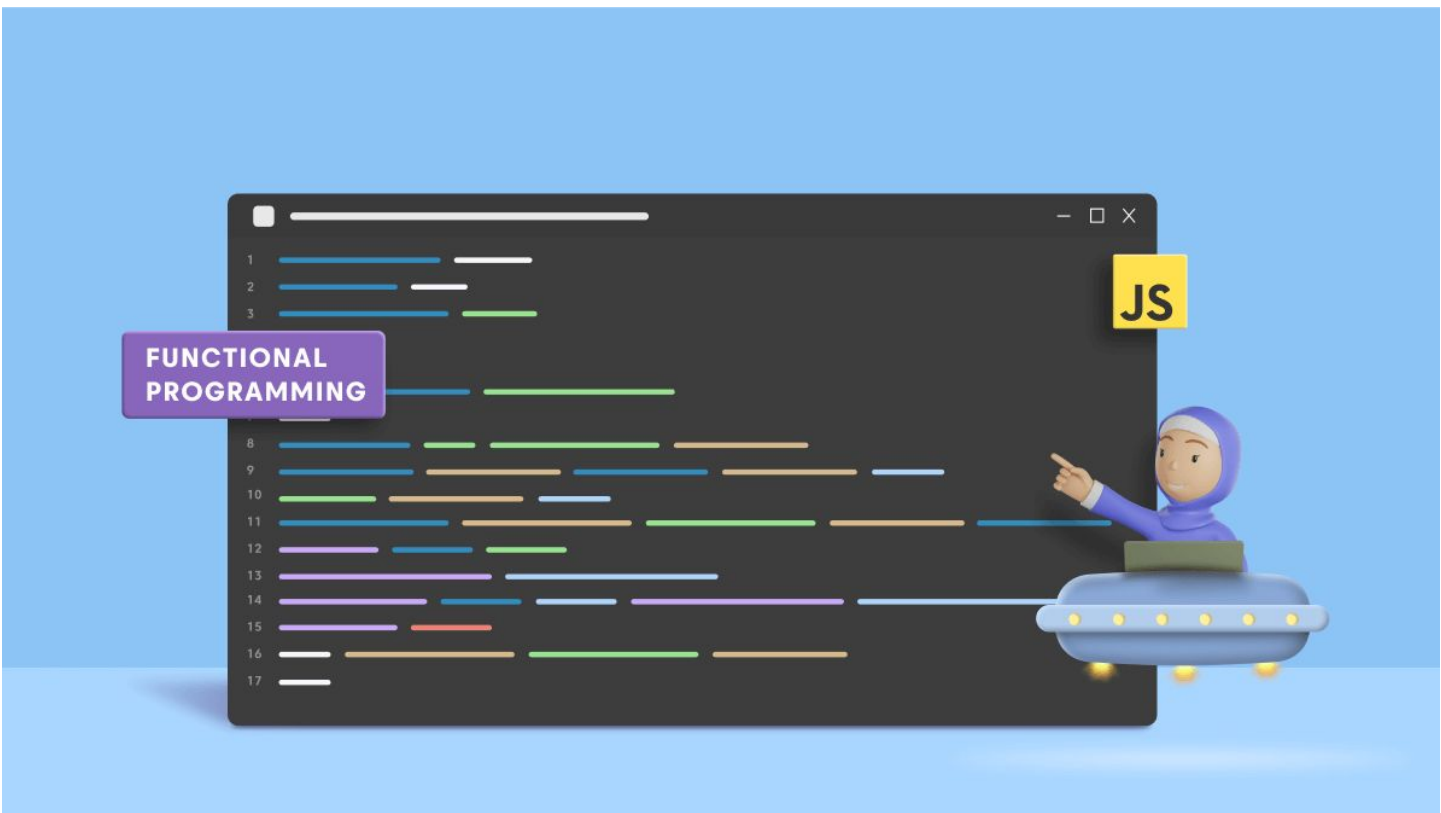
const clone2 = [...clone, "Cerise"];

console.log(clone2);
//["Pomme","Poire","Ananas","Pastèque","Cerise"]

//la valeur de fruits ne change pas
//["Pomme","Poire","Ananas"]
```



## Clone et manipulation de liste



## 7 functional programming techniques for JavaScript developers

<https://www.syncfusion.com/blogs/post/7-functional-programming-techniques-for-javascript-developers.aspx>

Auteur : Alexandre Leroux (alex@shrp.dev) - <https://shrp.dev>

---

## Concept d'immuabilité en JS / TS

Le concept d'immuabilité est central en programmation fonctionnelle.

L'immuabilité signifie que les valeurs affectées à une variable ne peuvent pas être modifiées (...la "variable" est donc une constante).

L'objectif est d'éviter les erreurs liées aux *"effets de bord"*.



**var**  
**let**

**mutabilité**



**const**

**immutabilité**  
**(partielle)**

let et var vs const

---

Pour éviter les effets de bord, en JS, l'emploi des mots clés *var* et *let* est **proscrit**.

Les mots clés *let* et *var* autorisent la ré-affectation de valeur à une variable.

Le mot clé **const** est donc **préféré**.

Cependant, le mot clé **const** ne garantit pas l'immuabilité d'un objet (Object) ni d'un tableau (Array).

```
const planets = [];  
  
planets.push('Mars');  
  
//la valeur de planets a évolué
```

**Mutabilité d'une liste déclarée avec  
le mot clé const**

```
const pluton =  
{name:"Pluton",type:"Planet"};  
  
pluton.type = "Dwarf Planet";  
  
//la valeur de pluton a évolué
```

**Mutabilité d'un objet déclaré avec  
le mot clé const**



---

## Object.freeze({})

En JS, pour figer la valeur d'une constante de type Object, il est possible d'utiliser sa méthode **freeze** lors de l'initialisation.

```
const earth = Object.freeze({ fr_FR: "Terre", en_EN: "Earth" });

earth.it_IT = "Terra";

//ne produit aucun effet car l'objet earth a été figé

console.log(earth);

//{ fr_FR: 'Terre', en_EN: 'Earth' }
```

## Object.freeze({})

---

## Readonly, la solution apportée par TypeScript

En TypeScript, il est possible d'employer le mot clé ***readonly*** pour déclarer des attributs de classe immuables.

Il est également possible d'employer le type générique ***Readonly<T>*** pour appliquer l'immuabilité sur les structures de données telles que `Array`, `Object`... (normalement mutables).

```
interface Todo {  
  title: string;  
}  
  
const todo: Readonly<Todo> = {  
  title: "Cette valeur est immuable",  
};  
  
todo.title = "Hello"; //provoque une erreur
```

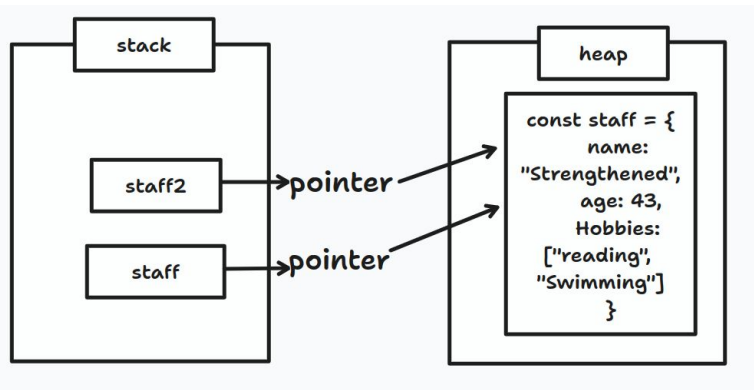
## Readonly Type en TypeScript

<https://www.typescriptlang.org/docs/handbook/utility-types.html#readonlytype>

```
const staff2 = staff;

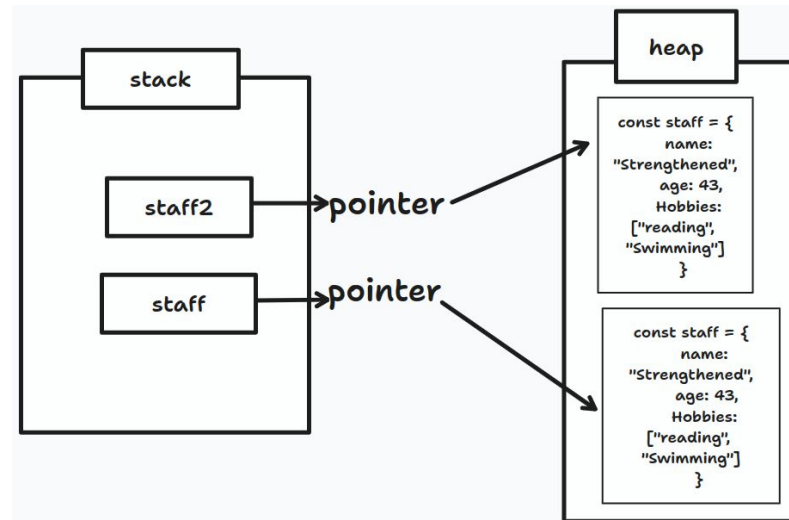
console.log(staff);

console.log(staff2);
```



```
const staff = {
  name: "Strengthened",
  age: 43,
  Hobbies: ["reading", "Swimming"]
}

const staff2 = Object.assign({}, staff);
```



## Mutability vs Immutability in JavaScript

<https://www.freecodecamp.org/news/mutability-vs-immutability-in-javascript/>

---

# Immutabilité des listes

---

## Emploi de l'opérateur Spread pour l'immutabilité

Pour respecter le concept d'immutabilité, il est nécessaire d'employer l'opérateur **spread** afin de créer une copie à partir de la liste d'origine : **copie identique** ou **copie modifiée** (ajout, suppression, modification, tri...).

Les méthodes **push**, **pop**, **unshift**... étant proscrites dans une approche fonctionnelle.

```
const planets = [  
  "Terre", "Vénus", "Mars", "Saturne",  
  "Uranus", "Jupiter", "Mercure", "Neptune",  
];
```

```
const planetsClone = [...planets];
```

```
//création d'une copie de la liste planets
```

## Copie de liste à l'aide de l'opérateur spread



```
const planets = ['Terre'];
```

```
//création d'une nouvelle liste à partir de la liste planets et  
insertion en dernière position
```

```
const clonedPlanetsWithMarsAtEnd = [...planets, "Mars"];  
//['Terre', 'Mars']
```

```
//création d'une nouvelle liste à partir de la liste planets et  
insertion en première position
```

```
const clonedPlanetsWithMarsAtBeginning = ["Pluton", ...planets];  
//['Pluton', 'Terre']
```

**Copie de liste et ajout en dernière ou première position  
(alternative à push ou unshift)**

```
const planets = [  
  "Terre", "Vénus", "Mars", "Saturne",  
  "Uranus", "Jupiter", "Mercure", "Neptune",  
];  
  
const clonedAndSortedPlanets = [...planets].sort();  
  
/*  
[  
  'Jupiter', 'Mars',  
  'Mercure', 'Neptune',  
  'Saturne', 'Terre',  
  'Uranus',   'Vénus'  
]  
*/
```

copie + tri

```
const planets = [
  "Terre", "Vénus", "Mars", "Saturne", "Uranus", "Jupiter", "Mercure", "Neptune",
];
```

```
const clonedAndReversedPlanets = [...planets].revert();
```

```
/*
[
  'Neptune', 'Mercure',
  'Jupiter', 'Uranus',
  'Saturne', 'Mars',
  'Vénus',    'Terre'
]
*/
```

copie + tri inversé

```
const planets = [
  "Terre", "Vénus", "Mars", "Saturne", "Uranus", "Jupiter", "Mercure", "Neptune",
];

const planetsWithoutVenus = [
  ...planets.slice(0, 1),
  ...planets.slice(2, planets.length),
];

/*
[
  'Terre',    'Mars', 'Saturne', 'Uranus', 'Jupiter',
  'Mercure', 'Neptune'
]
*/
```

copie + suppression  
(alternative à pop)

---

# Immutabilité des objets

```
const earth = { fr: "Terre", it: "Terra" };
```

```
const earthClone = { ...earth };
```

## Copie d'objet

```
const terre = { fr: "Terre", it: "Terra" };  
const earth = { ...terre, en: "Earth" };  
  
console.log(earth);  
  
//{fr: "Terre", it: "Terra", en: "Earth"}
```

## Copie d'objet et création d'attribut en dernière position

```
const terre = { fr: "Terre", it: "Terra" };  
  
const earth = { en: "Earth", ...terre };  
  
console.log(earth);  
  
//{en: "Earth", fr: "Terre", it: "Terra"}
```

## Copie d'objet et création d'attribut en première position



```
const terre = { fr: "Terre", en: "Earth" };  
  
const { fr, ...earth } = terre;  
  
console.log(earth);  
  
//{en: "Earth"}
```

## Copie d'objet et suppression d'attribut

---

# Shallow clone vs Deep Clone

---

## Shallow Copy vs Deep Copy

Il existe 2 façons de cloner un élément en JS (*shallow* et *deep*).

- ***Shallow Copy*** : copie "*superficielle*",
- ***Deep Copy*** : copie "*profonde*".

```
const food = { beef: '🥩', bacon: '🥓' }

// "Spread"
{ ...food }

// "Object.assign"
Object.assign({}, food)

// "JSON"
JSON.parse(JSON.stringify(food))

// RESULT:
// { beef: '🥩', bacon: '🥓' }
```

## 3 Ways to Clone Objects in JavaScript

<https://www.samanthaming.com/tidbits/70-3-ways-to-clone-objects/>

```
const finals = [  
  {  
    year: 2018,  
    location: {  
      city: "Moscow",  
      country: "Russia",  
    },  
    score: "4-2"  
  },  
  {  
    year: 2022,  
    location: {  
      city: "Lusail",  
      country: "Qatar",  
    },  
    score: "2-2"  
  }  
];
```

## Tableau JS contenant des objets complexes / profonds

```
const shallowClone = [...finals]; // clone avec la syntaxe spread

assert(arrayClone.at(-1)?.location.country === "Qatar"); // true

shallowClone.at(-1).location.country = "France";
// modification d'une valeur au sein du clone

assert(shallowClone.at(-1)?.location.country === "France");
// France

assert(finals.at(-1)?.location.country === "Qatar"); // France
// l'original est impacté par la modification du clone
// La valeur obtenue est "France" mais devrait être "Qatar"
// en modifiant une valeur sur le clone,
// on modifie la valeur de l'original
```

## Shallow Clone avec la syntaxe Spread

```
const shallowClone = Array.from(finals);

assert(shallowClone.at(-1)?.location.country === "Qatar"); //true

shallowClone.at(-1)!.location.country = "France";
//modification d'une valeur au sein du clone

assert(shallowClone.at(-1)?.location.country === "France");
//true
//le clone est bien modifié

assert(finals.at(-1)?.location.country==="Qatar");//false
//l'original est impacté par la modification du clone
//La valeur obtenue est "France" mais devrait être "Qatar"
//en modifiant une valeur sur le clone,
//on modifie la valeur de l'original
```

## Shallow Clone avec Array.from

---

## Défauts du clonage superficiel (Shallow Clone) avec Spread, Object.assign ou Array.from

- Si l'objet à copier contient des objets complexes / profonds, une modification sur le clone a un impact l'original,
- Si l'objet à copier contient des fonctions, elles ne seront pas copiées.



```
const deepClone = JSON.parse(JSON.stringify(finals));  
  
assert(finals.at(-1)?.location.country === "Qatar"); //true  
  
deepClone.at(-1).location.country = "France";  
//modification d'une valeur au sein du clone
```

```
assert(deepClone.at(-1)?.location.country === "France"); //true  
//le clone est bien modifié
```

```
assert(finals.at(-1)?.location.country === "Qatar"); //true  
//la modification sur le clone  
//ne provoque pas de modification sur l'original
```

## Deep Clone avec JSON

---

## Défauts du clonage profond (Deep Clone) avec JSON

- Ne fonctionne qu'avec les objets de type Number, String et Objets sans fonction ni symbol.

```
import _ from "lodash";

const lodashDeepClone = _.cloneDeep(finals);

lodashDeepClone.at(-1)!.location.country = "Germany";
//modification d'une valeur au sein du clone

assert(finals.at(-1)?.location.country === "Qatar");
//true
//l'original n'est pas impacté par la modification du clone

assert(lodashDeepClone.at(-1)?.location.country === "Germany");
//true
//le clone est bien modifié
```

## Clone avec Lodash


---

## Risques de pertes de performance

**Cloner des éléments riches en données** (volumineux et ou complexes), peut avoir un effet dangereux sur la **mémoire allouée au programme** : **risques de saturation et/ou de ralentissements d'exécution**.

Dans un langage "impur", il est préférable d'effectuer des **clonages stratégiques** pour éviter de dupliquer les données et de les conserver en mémoire.

D'autres **techniques d'optimisation** telles que la **Memoization** peuvent permettre d'éviter de recalculer systématiquement certaines données.



Un langage fonctionnel pur comme Haskell gère nativement l'allocation de la mémoire ce qui optimise ses performances.

Le langage Haskell s'adapte spécifiquement au Paradigme Fonctionnel autant dans la syntaxe que dans le cadre de son exécution.



**VS**



## A Functional Love Affair: Why I Chose Haskell Over JavaScript

<https://medium.com/@kerronp/a-functional-love-affair-why-i-chose-haskell-over-javascript-fe9b2c619c42>

```
const a = Array.from(  
  {length: 1000},  
  () => Math.floor(Math.random() * 1000)  
);  
  
console.log(a);  
/*  
// [object Array] (1000)  
[38,10,26,10,36,5,3...]  
*/
```

Génération d'une liste contenant 1000 valeurs numériques aléatoires

```
const a = Array.from({length: 1000}, () =>
Math.floor(Math.random() * 1000));

const clone = [...a]; // clone de la liste a

console.log(a.length===clone.length); // true

clone.unshift(0); // ajout de la valeur 0 en 1ère position


console.log(clone.length); // 1001

const clone2 = [1,...a];
// clone et ajout de la valeur 1 en première position

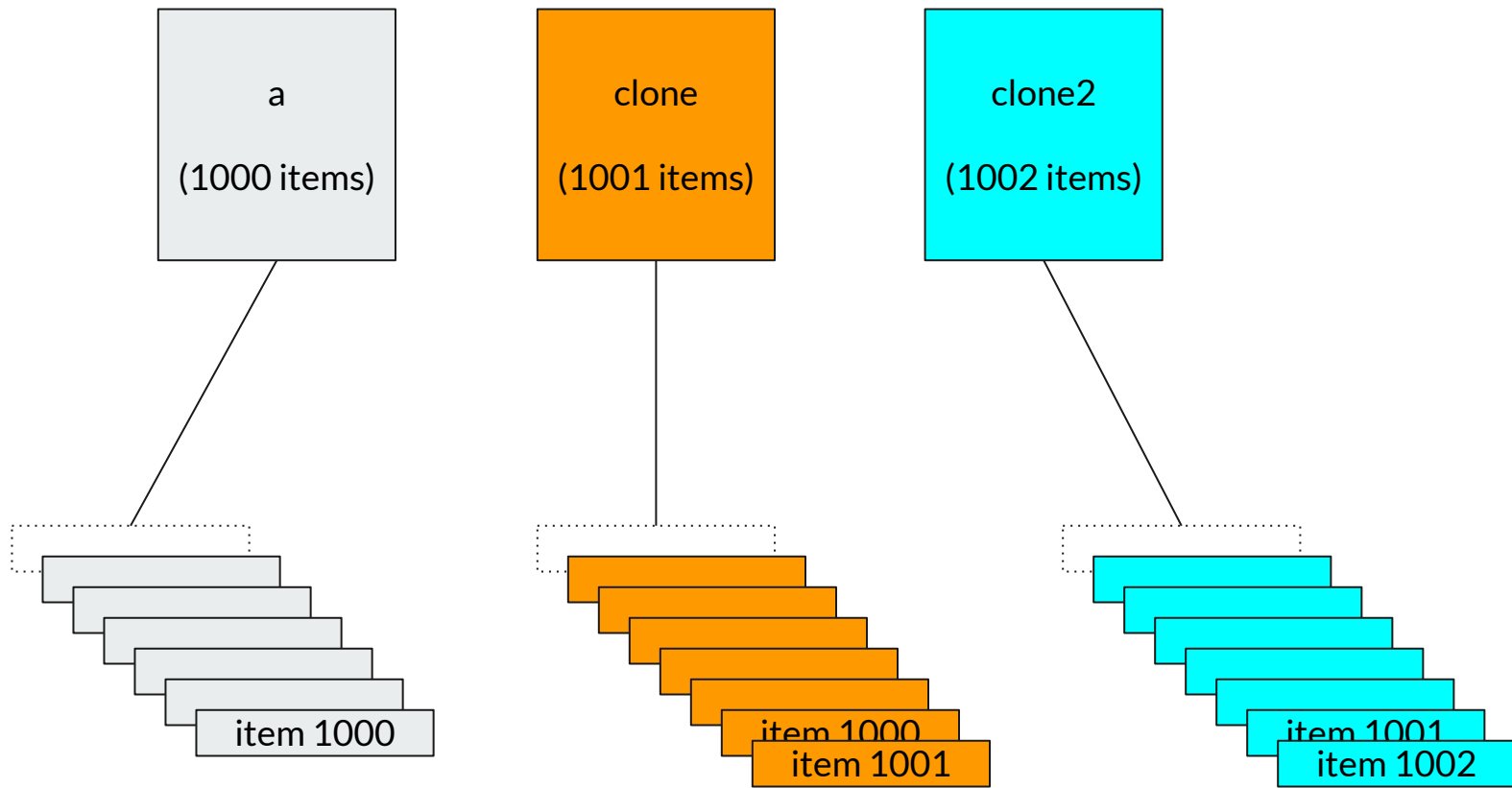
console.log(clone2);
```

## Clonage de la liste créée précédemment





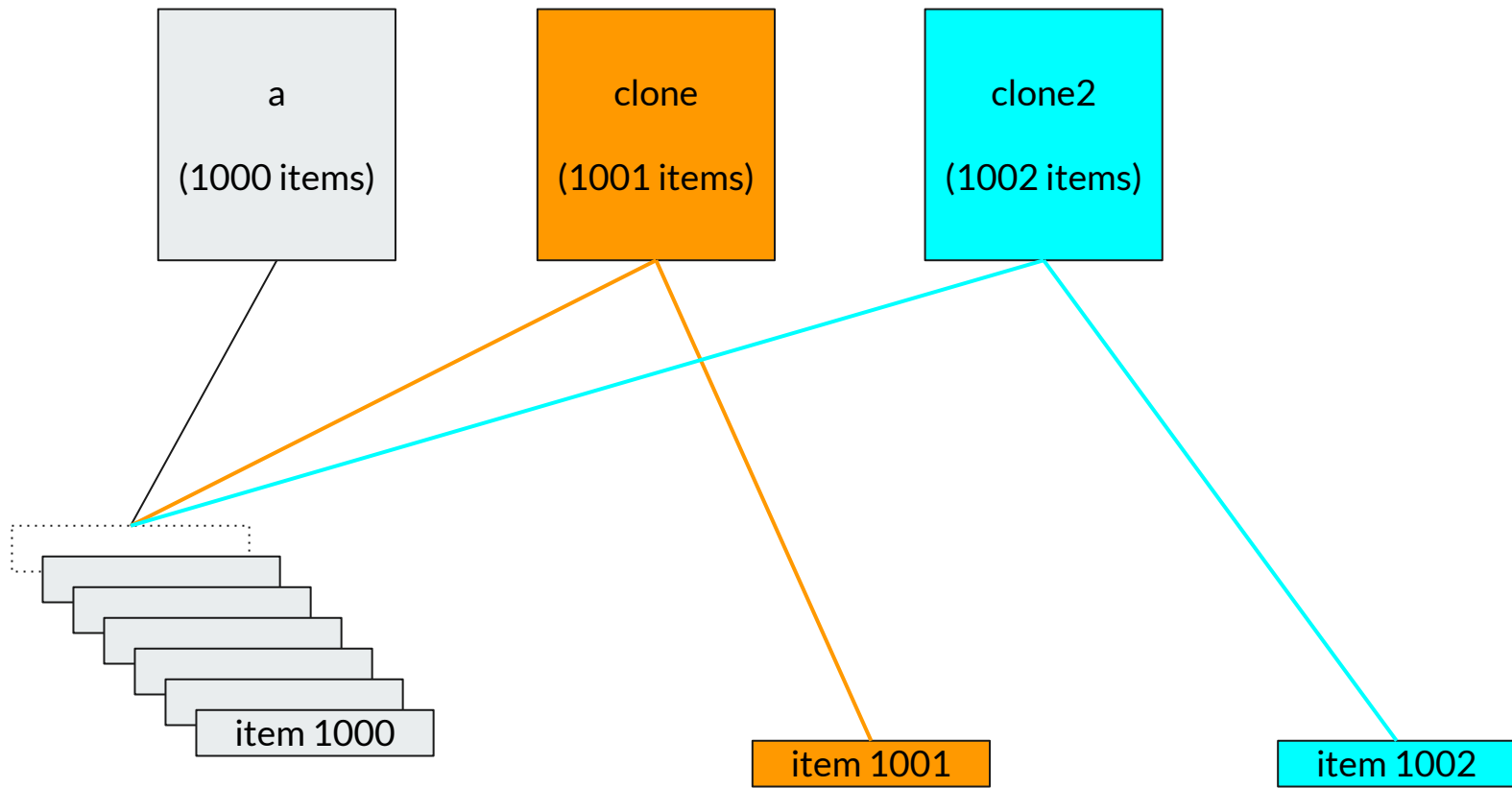
Dans l'exemple précédent, 3 listes de  $\pm 1000$  éléments sont conservés en mémoire, ce qui peut conduire à des problématiques de performance...



---

Pour éviter cette problématique, une solution peut consister à :

- **conserver une référence vers la liste d'origine** pour éviter de la dupliquer en mémoire.
- **conserver une référence vers les nouvelles valeurs,**
- "calculer" le contenu de la liste à chaque usage.





## La programmation fonctionnelle

<https://grafikart.fr/tutoriels/programmation-fonctionnelle-878>