Vieillissement des métaux

soit la BD suivante

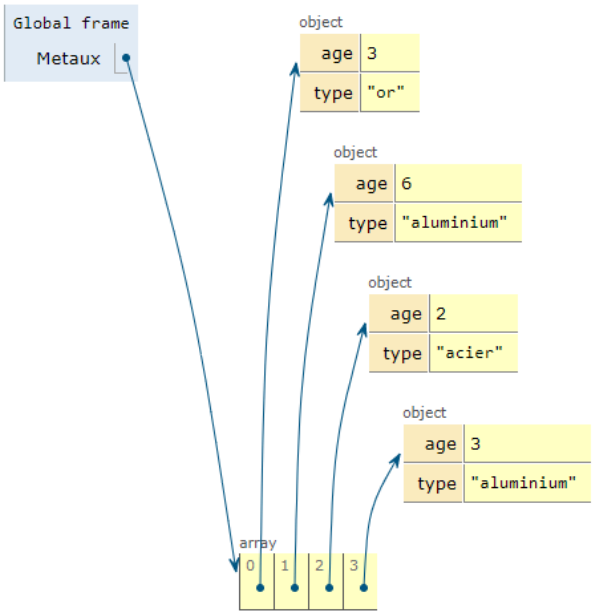
```
Metaux = [{ age: 3, type: 'or' },
{ age: 6, type: 'aluminium' },
{ age: 2, type: 'acier' },
{ age: 3, type: 'aluminium' },];
```

Metaux est un tableau d'objets.

`Metaux[0] == { age: 3, type: 'or' }` est un objet.

Les propriétés décrivent

1. le type de métal utilisé `Métaux[0].type`[1] `== 'or'`
2. l'âge `Métaux[0]['age'] == 3`.



[1] ou `Métaux[0]['type']`

Learn to Chain Map, Filter, and Reduce[2]

Notre objectif est clair : utilisons les méthodes sur Array en lieu et place de fonction que nous aurions pu développer.

Prenons un exemple avec la fonction getAges et tentons de la transformer.

La fonction getAges calcule pour tous les objets de type aluminium l'âge.

Pour prendre en compte une usure, l'âge est multiplié par un coefficient de régulation (de corrosion) qui est pour l'aluminium de 7 (valeur donnée au hasard).

```
1. function getAges(data) {
2.   let sum = 0;
3.   for (let i = 0; i < data.length; i++) {
4.     if (data[i].type === 'aluminium') {
5.       let tempAge = data[i].age;
6.       sum += (tempAge * 7);
7.     }
8.   }
9.   return sum;
10.}
11.console.log(getAges(Métaux));
```

code

Amélioration :

Je ne reviens pas sur la difficulté de changer le type de métal ('aluminium') ou le coefficient de corrosion (7) dans le code précédent.

Voici les solutions :

code

code

code

[2] Apprenez à chaîner les transformations, les filtres et réductions.

Chaînage

Prenons l'exemple de la méthode filtre, et relisons les informations sur la valeur de retour. (MDN)

Valeur de retour

Un nouveau tableau contenant les éléments qui respectent la condition du filtre. Si aucun élément ne respecte la condition, c'est un tableau vide qui est renvoyé.

Un filtre[3] rend un nouveau tableau, on peut appliquer de nouveau un filtre sur ce tableau.

Il est donc possible d'écrire :

```
1. data.filter().filter().filter();
```

Filter, map, reduce en action

Voici la transformation de la fonction getAges en utilisant les fonctions de base. Notez l'utilisation du chaînage.

```
1. let aluminiums = Metaux
2.    .filter(function(metal) {
3.      return metal.type === 'aluminium'})
4.    .map(function(metal){
5.      return metal.age * 7})
6.    .reduce(function(sum, age) {
7.      return sum + age});
8.
9.    console.log(`ages = ${aluminiums}`)
```

code

---

[3] la méthode map renvoie elle aussi un tableau, on pourra écrire :

```
data.map().filter()
```

Utilisation de fonction fléchée

```
1. let aluminiums = Metaux
2.     .filter((metal)   => metal.type === 'aluminium')
3.     .map(   (metal)   => metal.age * 7)
4.     .reduce((sum,age) => sum + age);
```

code

Nous pouvons pour plus de clarté[4] déclarer des fonctions

```
1. let sum   = (sum,age) => sum + age;
2. isAluminium  = (metal) => metal.type === 'aluminium',
3. aluminYears  = (metal) =>  metal.age * 7,
4.
5. let aluminiums = Metaux
6.    .filter(isAluminium)
7.    .map(aluminYears)
8.    .reduce(sum);
```

code

Appliquons la destruction[5]

```
1. let isaluminium = ( { type }) => type === 'aluminium';
2. let aluminiumYears = ( { age } ) =>  age * 7;
3. let sum = (sum, age) => sum + age;
4.
5. let aluminiums = Metaux
6.    .filter(isaluminium)
7.    .map(aluminiumYears)
8.    .reduce(sum);
```

code

[4] ou pour la réutilisation de fonctions
[5] sans grande amélioration de visibilité

Améliorations

Reprenons le cas de la fonction de filtre sur l'aluminium

```
1. let isAluminium = (metal) => metal.type === 'aluminium';
```

Il est facile de filtrer l'ensemble des différents types de métaux en écrivant

```
1. isAluminium = (metal) => metal.type === 'aluminium';
2. isOr = (metal) => metal.type === 'or';
3. isPlomb = (metal) => metal.type === 'Plomb';
4. isAcier = (metal) => metal.type === 'Acier';
5. isPlatine = (metal) => metal.type === 'Platine';
```

Pour éviter la réécriture de code pour chaque type de métal, l'utilisation de la closure est très efficace.

⚠ Closure[6]

L'utilisation d'une closure sur metal améliore grandement la souplesse de vos méthodes de filtre. On évite d'écrire pour chaque type de métal une fonction de filtre spécifique.

```
1. let metalType =  function (metal) {
2.              return function  ({ type }) {
3.                   return  type === metal;
4.              }
5. }
```

Que l'on peut ecrire également :

```
1. let metalType = (metal) => ({ type }) => type === metal;
```

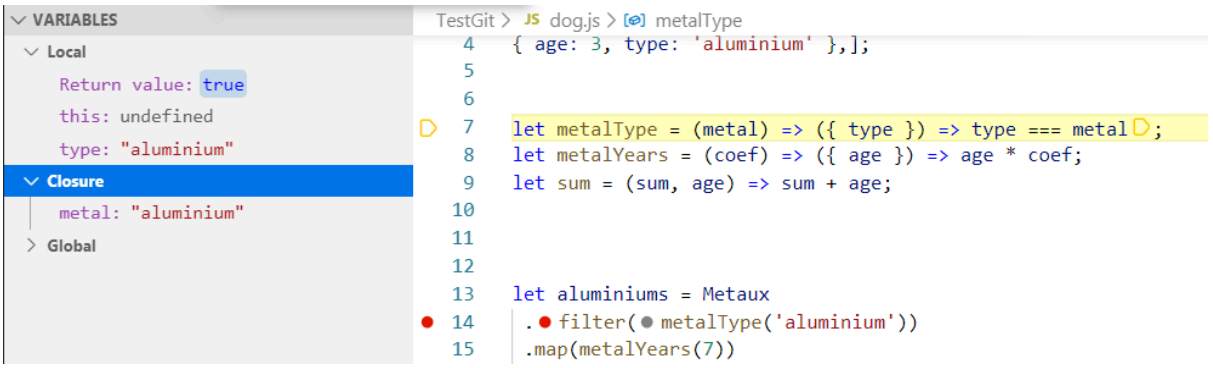
Je vous laisse comparer la réduction de code sur le tableau suivant

---

[6] articles

Déclaration	Avec losure
isAluminium = (metal) => metal.type === 'aluminium'; isOr = (metal) => metal.type === 'or'; isPlomb = (metal) => metal.type === 'plomb'; isAcier = (metal) => metal.type === 'acier'; isPlatine = (metal) => metal.type === 'platine';	metalType = (metal) => ({ type }) => type === metal;
Appel	**Appel**
Metaux.filter(isAluminium) Metaux.filter(isOr)	Metaux.filter(metalType('aluminium')) Metaux.filter(metalType('or'))

Voici une image de débogage montrant la closure sur le type de métal.



Appliquons également cette technique pour définir le coefficient de corrosion !

```
let metalYears = (coef) => ({ age }) => age * coef;
```

Le code se réécrit ainsi :

```
let metalType = (metal) => ({ type }) => type === metal;
let metalYears = (coef) => ({ age }) => age * coef;
let sum = (sum, age) => sum + age;

let isaluminium = metalType('aluminium');
let aluminiumYears = metalYears(7);

let aluminiums = Metaux
  .filter(isaluminium)
.map(aluminiumYears)
.reduce(sum);
```

code

Code final

Je vous laisse regarder le code permettant de passer au méthode, le type de métal et le coefficient de vieillissement.

```
let metalType = (metal) => ({ type }) => type === metal;
let metalYears = (coef) => ({ age }) => age * coef;
let sum = (sum, age) => sum + age;

let aluminiums = Métaux
  .filter(metalType('aluminium'))
  .map(metalYears(7))
  .reduce(sum);

let aciers = Métaux
  .filter(metalType('acier"))
  .map(metalYears(5))
  .reduce(sum);

console.log(`aluminiums = ${aluminiums}`)
console.log(`aciers = ${aciers}`);
```

code

Bonus :

Analyser ce code