TypeScript : tableaux

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

1. Array : construction
   - `keys`
   - `from`
   - `of`
   - `fill`

Plan

TypeScript

Remarques

- En **JavaScript**, il n'y a pas de méthode `range` pour générer un tableau contenant un intervalle de valeurs entières consécutives.
- Avec **EcmaScript 6**, on peut utiliser des méthodes comme `from` et `keys` pour générer un intervalle de valeurs entières consécutives.

TypeScript

Remarques

- En **JavaScript**, il n'y a pas de méthode `range` pour générer un tableau contenant un intervalle de valeurs entières consécutives.
- Avec **EcmaScript 6**, on peut utiliser des méthodes comme `from` et `keys` pour générer un intervalle de valeurs entières consécutives.

Exemple

```
var list: number[] = Array.from(Array(3).keys())
console.log(list);
// affiche [ 0, 1, 2 ]
```

TypeScript

On peut utiliser la méthode `from` pour créer un tableau à partir d'un itérable

```
console.log(Array.from('wick'));
// affiche [ 'w', 'i', 'c', 'k' ]
```

TypeScript

On peut utiliser la méthode `from` pour créer un tableau à partir d'un itérable

```
console.log(Array.from('wick'));
// affiche [ 'w', 'i', 'c', 'k' ]
```

On peut aussi utiliser la méthode `from` pour créer un tableau à partir d'un autre tableau

```
console.log(Array.from([1, 2, 3], elt => elt * 2));
// affiche [ 2, 4, 6 ]
```

TypeScript

On peut utiliser une fonction fléchée pour modifier les valeurs générées

```
var list: number[] = Array.from({ length: 3 }, (v, k
   ) => k + 1)
console.log(list);
// affiche [ 1, 2, 3 ]
```

TypeScript

On peut utiliser la méthode `of` pour créer un tableau à partir d'une liste de paramètres

```
console.log(Array.of(1, 2, 3));
// affiche [1, 2, 3]
```

TypeScript

On peut utiliser la méthode `of` pour créer un tableau à partir d'une liste de paramètres

```
console.log(Array.of(1, 2, 3));
// affiche [1, 2, 3]
```

L'écriture précédente peut être simplifiée

```
console.log(Array(1, 2, 3));
// affiche [1, 2, 3]
```

TypeScript

Les deux écritures suivantes sont différentes

```
console.log(Array.of(7));
// affiche [7]

console.log(Array(7));
// affiche [ <7 empty items> ]
```

TypeScript

La méthode `fill` permet de remplir les éléments d'un tableau entre deux index avec une valeur statique

```
let tab = Array(7);
tab.fill(2);
console.log(tab);
// affiche [ 2, 2, 2, 2, 2, 2, 2 ]

tab.fill(0, 3);
console.log(tab);
// affiche [ 2, 2, 2, 0, 0, 0, 0 ]

tab.fill(5, 2, 4);
console.log(tab);
// affiche [ 2, 2, 5, 5, 0, 0, 0 ]
```

TypeScript

Les fonctions fléchées sont utilisées pour réaliser les opérations suivantes sur les tableaux

- `forEach()` : pour parcourir un tableau.
- `map()` : pour appliquer une fonction sur les éléments d'un tableau.
- `filter()` : pour filtrer les éléments d'un tableau selon un critère défini sous forme d'une fonction anonyme ou fléchée.
- `reduce()` : pour réduire tous les éléments d'un tableau en un seul selon une règle définie dans une fonction anonyme ou fléchée.
- `some()` : pour vérifier s'il existe au moins un élément qui respect une condition.
- `every()` : pour vérifier si tous les éléments respectent une condition.
- `...`

TypeScript

Quelques fonctions de recherche acceptant les callbacks et les fonctions fléchées

- `find()` : retourne le premier élément qui respecte le prédicat, `undefined` sinon.
- `findIndex()` : retourne l'indic due premier élément qui respecte le prédicat, `-1` sinon.
- `findLast()` : retourne le premier élément, en commençant par la fin, qui respecte le prédicat, `undefined` sinon.
- `findLastIndex()` : retourne l'indice du premier élément, en commençant par la fin, qui respecte le prédicat, `undefined` sinon.
- `at()` : pour accéder aux éléments avec un support pour les index négatifs.
- ...

Utiliser `forEach` pour afficher le contenu d'un tableau

```
var tab = [2, 3, 5];
tab.forEach(elt => console.log(elt));
// affiche 2 3 5
```

Utiliser `forEach` pour afficher le contenu d'un tableau

```
var tab = [2, 3, 5];
tab.forEach(elt => console.log(elt));
// affiche 2 3 5
```

Dans `forEach`, on peut aussi appeler une fonction `afficher`

```
tab.forEach(elt => afficher(elt));

function afficher(value: number) {
    console.log(value);
}
// affiche 2 3 5
```

Utiliser `forEach` pour afficher le contenu d'un tableau

```
var tab = [2, 3, 5];
tab.forEach(elt => console.log(elt));
// affiche 2 3 5
```

Dans `forEach`, on peut aussi appeler une fonction `afficher`

```
tab.forEach(elt => afficher(elt));

function afficher(value: number) {
    console.log(value);
}
// affiche 2 3 5
```


On peut simplifier l'écriture précédente en utilisant les callback

```
tab.forEach(afficher);

function afficher(value: number) {
    console.log(value);
}
// affiche 2 3 5
```

TypeScript

La fonction `afficher` peut accepter deux paramètres : le premier est la valeur de l'itération courante et le deuxième est son indice dans le tableau

```
tab.forEach(afficher);

function afficher(value: number, key: number) {
    console.log(key, value);
}

/* affiche
0 2
1 3
2 5
*/
```

TypeScript

La fonction `afficher` peut accepter un troisième paramètre qui correspond au tableau

```
tab.forEach(afficher);

function afficher(value: number, key: number, t:
   Array<number>) {
     console.log(key, value, t);
}

/* affiche
0 2 [ 2, 3, 5 ]
1 3 [ 2, 3, 5 ]
2 5 [ 2, 3, 5 ]
*/
```

TypeScript

On peut utiliser `map` pour effecteur un traitement sur chaque élément du tableau puis `forEach` pour afficher le nouveau tableau

```
tab.map(elt => elt + 3)
    .forEach(elt => console.log(elt));
// affiche 5 6 8
```

TypeScript

On peut aussi utiliser `filter` pour filtrer quelques éléments

```
tab.map(elt => elt + 3)
    .filter(elt => elt > 5)
    .forEach(elt => console.log(elt));
// affiche 6 8
```

TypeScript

On peut aussi utiliser `find` pour récupérer le premier élément qui satisfait la condition, `undefined` sinon

```
var found = tab
    .map(elt => elt + 3)
    .find(elt => elt > 5)
console.log(found);
// affiche 6

var notFound = tab
    .map(elt => elt + 3)
    .find(elt => elt > 10)
console.log(notFound);
// affiche undefined
```

TypeScript

`findIndex` **retourne la position de l'élément recherché,** `-1` **sinon**

```
var found = tab
    .map(elt => elt + 3)
    .findIndex(elt => elt > 5)
console.log(found);
// affiche 1

var notFound = tab
    .map(elt => elt + 3)
    .findIndex(elt => elt > 10)
console.log(notFound);
// affiche -1
```

TypeScript

Remarque

Attention, selon l'ordre d'appel de ces méthodes, le résultat peut changer.

TypeScript

Remarque

Attention, selon l'ordre d'appel de ces méthodes, le résultat peut changer.

Exemple avec `reduce` : permet de réduire les éléments d'un tableau en une seule valeur

```
var tab = [2, 3, 5];
var somme = tab.map(elt => elt + 3)
    .filter(elt => elt > 5)
    .reduce((sum, elt) => sum + elt);

console.log(somme);
// affiche 14
```

TypeScript

Si on a plusieurs instructions, on doit ajouter les accolades

```
var tab = [2, 3, 5];
var somme = tab.map(elt => elt + 3)
     .filter(elt => elt > 5)
     .reduce((sum, elt) => {
         return sum + elt;
     }
);

console.log(somme);
// affiche 14
```

TypeScript

Remarques

- Le premier paramètre de `reduce` correspond au résultat de l'itération précédente
- Le deuxième correspond à l'élément du tableau de l'itération courante
- Le premier paramètre est initialisé par la valeur du premier élément du tableau
- On peut changer la valeur initiale du premier paramètre en l'ajoutant à la fin de la méthode

TypeScript

Dans cet exemple, on initialise le premier paramètre de `reduce` par la valeur 0

```
var somme = tab.map(elt => elt + 3)
    .filter(elt => elt > 5)
    .reduce((sum, elt) =>  sum + elt, 0);

console.log(somme);
// affiche 14
```

TypeScript

Dans cet exemple, on vérifie s'il existe un élément pair dans le tableau, après modification

```
var tab = [2, 3, 5];

let result = tab.map(elt => elt + 3)
    .some(elt => elt % 2 == 0);

console.log(result);
// affiche true
```

TypeScript

Dans cet exemple, on vérifie si tous les éléments du tableau sont pairs, après modification

```
var tab = [2, 3, 5];

let result = tab.map(elt => elt + 3)
    .every(elt => elt % 2 == 0);

console.log(result);
// affiche false
```

TypeScript

Fonctions fléchées : pourquoi ?

- Simplicité d'écriture du code ⇒ meilleure lisibilité
- Pas de binding avec les objets prédéfinis : `arguments`, `this`...
- ...

Considérons la fonction `somme` suivante

```
function somme(a?: number, b?: number, c?: number): number {
  return a + b + c;
}
```

Considérons la fonction `somme` suivante

```
function somme(a?: number, b?: number, c?: number): number {
  return a + b + c;
}
```

Pour appeler la fonction `somme`, il faut lui passer trois paramètres `number`

```
console.log(somme (1, 3, 5));
// affiche 9
```

Considérons la fonction `somme` suivante

```
function somme(a?: number, b?: number, c?: number): number {
  return a + b + c;
}
```

Pour appeler la fonction `somme`, il faut lui passer trois paramètres `number`

```
console.log(somme (1, 3, 5));
// affiche 9
```

Et si les valeurs se trouvent dans un tableau, on peut utiliser la décomposition

```
let t: Array<number> = [1, 3, 5];
console.log(somme(...t));
```

Considérons la fonction `somme` suivante

```
function somme(a?: number, b?: number, c?: number): number {
  return a + b + c;
}
```

Pour appeler la fonction `somme`, il faut lui passer trois paramètres `number`

```
console.log(somme (1, 3, 5));
// affiche 9
```

Et si les valeurs se trouvent dans un tableau, on peut utiliser la décomposition

```
let t: Array<number> = [1, 3, 5];
console.log(somme(...t));
```

On peut utiliser partiellement la décomposition

```
let t: Array<number> = [1, 3];
console.log(somme(...t, 5));
```

TypeScript

Considérons les deux objets suivants

```
let obj = { nom: 'wick', prenom: 'john'};
let obj2 = obj;
```

TypeScript

Considérons les deux objets suivants

```
let obj = { nom: 'wick', prenom: 'john'};
let obj2 = obj;
```

Modifier l'un $\Rightarrow$ modifier l'autre

```
obj2.nom = 'abruzzi';
console.log(obj);
// affiche { nom: 'abruzzi', prenom: 'john' }

console.log(obj2);
// affiche { nom: 'abruzzi', prenom: 'john' }
```

TypeScript

Pour que les deux objets soient indépendants, on peut utiliser la décomposition pour faire le clonage

```
let obj = { nom: 'wick', prenom: 'john'};
let obj2 = { ...obj };
obj2.nom = 'abruzzi';

console.log(obj);
// affiche { nom: 'wick', prenom: 'john' }

console.log(obj2);
// affiche { nom: 'abruzzi', prenom: 'john' }
```

TypeScript

`Map` (dictionnaire)

- Type fonctionnant avec un couple (clé,valeur)
- La clé doit être unique
- Chaque élément est appelé entrée (`entry`)
- Les éléments sont stockés et récupérés dans l'ordre d'insertion
- Disponible depuis ES5 puis modifié dans ES6

TypeScript

Pour créer un `Map`

```
let map: Map<string, number> = new Map([
    ['php', 17],
    ['java', 10],
    ['c', 12]
]);
console.log(map);
// affiche  Map { 'php' => 17, 'java' => 10, 'c' => 12 }
```

TypeScript

Pour créer un `Map`

```
let map: Map<string, number> = new Map([
    ['php', 17],
    ['java', 10],
    ['c', 12]
]);
console.log(map);
// affiche  Map { 'php' => 17, 'java' => 10, 'c' => 12 }
```

Ajouter un élément à un `Map`

```
map.set('html', 18);
```

TypeScript

Pour créer un `Map`

```
let map: Map<string, number> = new Map([
    ['php', 17],
    ['java', 10],
    ['c', 12]
]);
console.log(map);
// affiche  Map { 'php' => 17, 'java' => 10, 'c' => 12 }
```

Ajouter un élément à un `Map`

```
map.set('html', 18);
```

Pour ajouter plusieurs éléments à la fois

```
map.set('html', 18)
   .set('css', 12);
```

TypeScript

Si la clé existe déjà, la valeur sera remplacée

```
console.log(map.set('html', 20));
// affiche Map { 'php' => 17, 'java' => 10, 'c' => 12, 'html' => 20 }
```

TypeScript

Si la clé existe déjà, la valeur sera remplacée

```
console.log(map.set('html', 20));
// affiche Map { 'php' => 17, 'java' => 10, 'c' => 12, 'html' => 20 }
```

Pour récupérer la valeur associée à une clé

```
console.log(map.get('php'));
// affiche 17
```

TypeScript

Si la clé existe déjà, la valeur sera remplacée

```
console.log(map.set('html', 20));
// affiche Map { 'php' => 17, 'java' => 10, 'c' => 12, 'html' => 20 }
```

Pour récupérer la valeur associée à une clé

```
console.log(map.get('php'));
// affiche 17
```

Pour vérifier l'existence d'une clé

```
console.log(map.has('php'));
// affiche true
```

TypeScript

Si la clé existe déjà, la valeur sera remplacée

```
console.log(map.set('html', 20));
// affiche Map { 'php' => 17, 'java' => 10, 'c' => 12, 'html' => 20 }
```

Pour récupérer la valeur associée à une clé

```
console.log(map.get('php'));
// affiche 17
```


Pour vérifier l'existence d'une clé

```
console.log(map.has('php'));
// affiche true
```

Pour supprimer un élément selon la clé

```
map.delete('php');
```

TypeScript

Pour récupérer la liste des clés d'un `Map`

```
console.log(map.keys());
// affiche [Map Iterator] { 'java', 'c', 'html', 'css' }
```

TypeScript

Pour récupérer la liste des clés d'un `Map`

```
console.log(map.keys());
// affiche [Map Iterator] { 'java', 'c', 'html', 'css' }
```

Pour récupérer la liste des valeurs d'un `Map`

```
console.log(map.values());
// affiche [Map Iterator] { 10, 12, 20, 12 }
```

TypeScript

Pour récupérer la liste des clés d'un `Map`

```
console.log(map.keys());
// affiche [Map Iterator] { 'java', 'c', 'html', 'css' }
```

Pour récupérer la liste des valeurs d'un `Map`

```
console.log(map.values());
// affiche [Map Iterator] { 10, 12, 20, 12 }
```

Pour récupérer la liste des entrées d'un `Map`

```
console.log(map.entries());
/* affiche
[Map Entries] {
  [ 'java', 10 ],
  [ 'c', 12 ],
  [ 'html', 20 ],
  [ 'css', 12 ]
}
*/
```

TypeScript

Pour parcourir un `Map`, on peut utiliser `entries()` (solution ES5)

```
for (let elt of map.entries())
    console.log(elt[0] + " " + elt[1]);
/* affiche
java 10
c 12
html 20
css 12
*/
```

TypeScript

Pour parcourir un `Map`, on peut utiliser `entries()` (solution ES5)

```
for (let elt of map.entries())
    console.log(elt[0] + " " + elt[1]);
/* affiche
java 10
c 12
html 20
css 12
*/
```

Depuis ES6, on peut faire

```
for (let [key, value] of map) {
    console.log(key, value);
}
/* affiche
java 10
c 12
html 20
css 12
*/
```

TypeScript

On peut le faire aussi avec `keys()`

```
for (let key of map.keys()) {
    console.log(key + " " + map.get(key));
}
/* affiche
java 10
c 12
html 20
css 12
*/
```

TypeScript

Une deuxième solution consiste à utiliser `forEach` (affiche seulement les valeurs)

```
map.forEach(elt => console.log(elt));

/* affiche
10
12
20
12
*/
```

TypeScript

On peut aussi définir une méthode et l'appeler dans `forEach` (affiche seulement les valeurs)

```
map.forEach(elt => afficher(elt));

function afficher(elt: number) {
    console.log(elt)
}
```

TypeScript

On peut aussi définir une méthode et l'appeler dans `forEach` (affiche seulement les valeurs)

```
map.forEach(elt => afficher(elt));

function afficher(elt: number) {
    console.log(elt)
}
```


On peut encore simplifier l'appel de la fonction avec les callback (affiche seulement les valeurs)

```
map.forEach(afficher);

function afficher(elt: number){
    console.log(elt)
}
```

TypeScript

Pour afficher les clés et les valeurs

```
map.forEach(afficher);

function afficher (value: number, key: string) {
    console.log(value, key)
}

/* affiche
10 java
12 c
20 html
12 css
*/
```

TypeScript

Ou aussi

```
map.forEach((v: number, k: string) => console.log(k,v));

/* affiche
10 java
12 c
20 html
12 css
*/
```

TypeScript

Étant donné ce dictionnaire

```
let langages: Map<string, number> = new Map([
    ['php', 4],
    ['java', 5],
    ['python', 1],
    ['typescript', 3],
]);
```

Exercice

Écrire un programme **TypeScript** qui permet de répéter l'affichage de chaque clé de ce dictionnaire selon la valeur associée

Résultat attendu :

```
phpphpphpphp javajavajavajavajava python
  typescripttypescripttypescript
```

TypeScript

Correction

```
let result: string = '';

langages.forEach((v: number, k: string) => {
    for (let i = 0; i < v; i++) {
        result += k;
    }
    result += ' ';
});

console.log(result);
```

TypeScript

Exercice : Étant donnée le tableau suivant :

```
let liste: Array<any> =  [2, 5, 'Bonjour', true, 'c'
   , "3", "b", false, 10];
```

Écrire un programme TypeScript qui permet de stocker dans un dictionnaire (`Map`) les types présents dans la liste `liste` ainsi que le nombre d'éléments de cette liste appartenant à chaque type.

Résultat attendu :

```
Map { 'number' => 3, 'string' => 4, 'boolean' => 2 }
```

TypeScript

Correction

```
let liste: Array<any> = [2, 5, 'Bonjour', true, 'c', "3",
    "b", false, 10];

let occurrencesType: Map<string, number> = new Map();

liste.forEach((elt: any) => {
    occurrencesType.set(typeof elt, (occurrencesType.get(
      typeof elt) ?? 0) + 1);
});

console.log(occurrencesType);
// Map { 'number' => 3, 'string' => 4, 'boolean' => 2 }
```

Comparaison entre `Object` et `Map` : `Object`

- Non ordonné et non itérable
- Les clés peuvent seulement être de type `string`, `number` ou `symbol`
- Impossible de connaître la taille sans passer par l'ensemble des clés ou des valeurs : `object.keys(obj).length`
- Un support direct pour le format JSON

Comparaison entre `Object` et `Map` : `Object`

- Non ordonné et non itérable
- Les clés peuvent seulement être de type `string`, `number` ou `symbol`
- Impossible de connaître la taille sans passer par l'ensemble des clés ou des valeurs : `object.keys(obj).length`
- Un support direct pour le format JSON

Comparaison entre `Object` et `Map` : `Map`

- Ordonné et itérable
- Pour `Map`, les clés peuvent être de tout type
- La taille peut être déterminée facilement : `map.size`
- Pas de support pour le format JSON

TypeScript

Set

- Une collection ne contenant pas de doublons
- Acceptant les types simples et objets
- Les éléments sont stockées et récupérés dans l'ordre d'insertion
- Disponible depuis ES5 puis modifié dans ES6

TypeScript

Pour créer un `Set`

```
let marques = new Set(["peugeot", "ford", "fiat", "mercedes"]);
console.log(marques);
// affiche  Set { 'peugeot', 'ford', 'fiat', 'mercedes' }
```

TypeScript

Pour créer un `Set`

```
let marques = new Set(["peugeot", "ford", "fiat", "mercedes"]);
console.log(marques);
// affiche  Set { 'peugeot', 'ford', 'fiat', 'mercedes' }
```

Ajouter un élément à un `Set`

```
marques.add('citroen');
```

TypeScript

Pour créer un `Set`

```
let marques = new Set(["peugeot", "ford", "fiat", "mercedes"]);
console.log(marques);
// affiche  Set { 'peugeot', 'ford', 'fiat', 'mercedes' }
```

Ajouter un élément à un `Set`

```
marques.add('citroen');
```


Pour ajouter plusieurs éléments à la fois

```
marques.add('citroen')
       .add('renault');
```

TypeScript

On ne peut ajouter un élément deux fois

```
marques.add('peugeot');
console.log(marques);
// affiche Set { 'peugeot', 'ford', 'fiat', 'mercedes', '
   citroen', 'renault' }
```

TypeScript

On ne peut ajouter un élément deux fois

```
marques.add('peugeot');
console.log(marques);
// affiche Set { 'peugeot', 'ford', 'fiat', 'mercedes', '
   citroen', 'renault' }
```

Pour vérifier l'existence d'un élément

```
console.log(marques.has('fiat'));
// affiche true
```

TypeScript

On ne peut ajouter un élément deux fois

```
marques.add('peugeot');
console.log(marques);
// affiche Set { 'peugeot', 'ford', 'fiat', 'mercedes', '
   citroen', 'renault' }
```

Pour vérifier l'existence d'un élément

```
console.log(marques.has('fiat'));
// affiche true
```

Pour supprimer un élément

```
marques.delete('ford');
console.log(marques);
// affiche Set { 'peugeot', 'fiat', 'mercedes', 'citroen', '
   renault' }
```

TypeScript

Autres méthodes sur les `Set`

- `A.subSet(B)` : retourne `true` si `A` est un sous-ensemble de `B`, `false` sinon.
- `A.union(B)` : retourne un **Set** regroupant les éléments de `A` et de `B`.
- `A.intersection(B)` : retourne un **Set** contenant les éléments de `A` qui sont dans `B`.
- `A.difference(B)` : retourne un **Set** contenant les éléments de `A` qui ne sont pas dans `B`.

TypeScript

Pour parcourir un `Set`

```
for(let marque of marques) {
    console.log(marque)
}
/* affiche
peugeot
fiat
mercedes
citroen
renault
*/
```

TypeScript

Une deuxième solution consiste à utiliser `forEach`

```
marques.forEach(elt => console.log(elt));

/* affiche
peugeot
fiat
mercedes
citroen
renault
*/
```

TypeScript

On peut aussi définir une méthode et l'appeler dans `forEach`

```
marques.forEach(elt => afficher(elt));

function afficher(elt: string){
    console.log(elt)
}
```

TypeScript

On peut aussi définir une méthode et l'appeler dans `forEach`

```
marques.forEach(elt => afficher(elt));

function afficher(elt: string){
    console.log(elt)
}
```

On peut encore simplifier l'appel de la fonction avec les callback

```
marques.forEach(afficher);

function afficher(elt: string){
    console.log(elt)
}
```