Initiation au développement

7. Tableaux (`Array`)

Objets, encapsulation et référence

Les tableaux en TypeScript sont des **objets**.

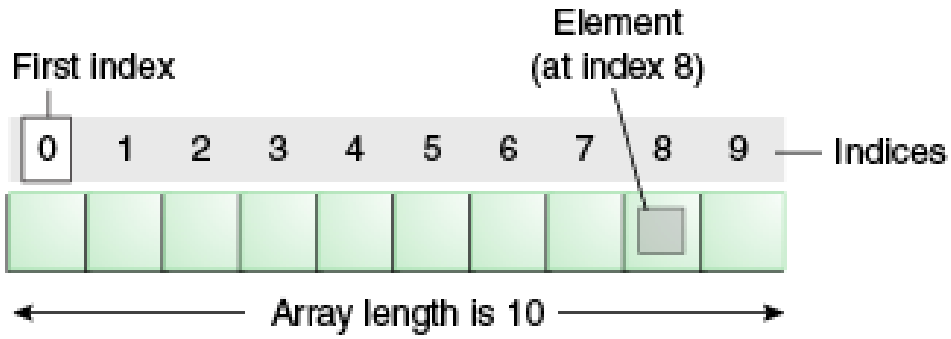
Ils illustrent clairement le concept d'**encapsulation** : un tableau est une structure de données qui contient des éléments et des fonctions pour accéder / modifier ces éléments. Par contre, on ne connaît pas la manière dont ces éléments sont stockés en mémoire.

Un autre aspect commun à de nombreux langages manipulant des objets est le fait que les tableaux se manipulent par **référence**. Cela signifie que lorsqu'on affecte un tableau à une autre variable, on ne copie pas le tableau mais on crée une nouvelle référence sur le même tableau.

Un dernier point (qui ne sera pas vu dans ce cours) qu'illustre les tableaux en TypeScript est la **généricité**. Un tableau est une structure de données qui peut stocker des éléments de n'importe quel type. On peut donc définir un tableau de chaînes de caractères, un tableau d'entiers, un tableau de tableaux ... Quelque soit le type stocké, les fonctions pour accéder / modifier les éléments sont les mêmes.

Définition

Un tableau est une structure de données séquentielles comme les chaînes de caractères.



- Un tableau stocke des éléments qui sont tous du même type
- Un tableau stocke les éléments dans une région contiguë de la mémoire
- Un variable de type tableau représente l'adresse du premier élément en mémoire
- On peut récupérer un élément via son indice

Déclaration d'un tableau

Il existe deux manières de déclarer un tableau :

```
let tableau1: number[];
let tableau2: Array<number>;
```

Les deux instructions ci-dessus sont équivalentes. On doit indiquer quel est le type des données que le tableau doit stocké. Ci-dessus, le tableau contient des entiers. On peut stocker tous les types existants. Ci-dessous quelques exemples :

```
let unTableauDeChaînesDeCaractères: Array<string>;
let unTableauDeBooléens: Array<boolean>;
let unTableauDeTableauxDeNombres: Array<Array<number>>;
```

- **Remarque :** il faut toujours affecter une valeur à un tableau avant de l'utiliser

Valeur littérale d'un tableau

Une valeur littérale pour un tableau est une séquence de valeurs séparées par des virgules entre `[]`.

```
let fruits = ["pommes", "orange", "banane"];
let chiffres = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
```

Remarque : le type des éléments du tableau est correctement inféré. On n'est donc pas obligé de déclarer explicitement le type de la variable.

On peut aussi créer un tableau vide.

```
let tableauVide = [];
```

Allocation d'un tableau avec `new`

L'instruction `new` permet d'**allouer** un tableau en mémoire. On peut créer un tableau vide de la manière suivante :

```
let tab = new Array<number>();
```

Remarque : le type de la variable est aussi déterminée implicitement.

On peut aussi remplir des éléments à la déclaration :

```
let jours = new Array<string>(
  "lundi",
  "mardi",
  "mercredi",
  "jeudi",
```

```
  "vendredi",
  "samedi",
  "dimanche"
);
```

Enfin, on peut créer un tableau avec une taille donnée :

```
let tab = new Array<number>(5);
```

Attention : la valeur des éléments est `undefined`, il faut les affecter avant de les utiliser

Longueur d'un tableau

Tout comme les chaînes de caractères, les tableaux possèdent un attribut `length` permettant de récupérer leur taille.

```
let tab = new Array<number>();
let jours = new Array<string>(
  "lundi",
  "mardi",
  "mercredi",
  "jeudi",
  "vendredi",
  "samedi",
  "dimanche"
);

console.log("tab.length =", tab.length);
console.log("jour.length =", jours.length);
```

```
tab.length = 0
jour.length = 7
```

Accès à un élément : opérateur `[]`

Comme pour les chaînes de caractères, les éléments dans un tableau `tab` sont indicés de `0` à `tab.length - 1`. Comme pour les chaînes de caractères, l'accès à l'élément `i` se fait par `tab[i]`.

Par exemple, on peut afficher les éléments d'un tableau de la manière suivante :

```
let jours = new Array<string>(
  "lundi",
  "mardi",
```

```
  "mercredi",
  "jeudi",
  "vendredi",
  "samedi",
  "dimanche"
);

for (let i = 0; i < jours.length; i++) console.log("jours[" + i + "] = " + jours[i]);
```

```
jours[0] = lundi
jours[1] = mardi
jours[2] = mercredi
jours[3] = jeudi
jours[4] = vendredi
jours[5] = samedi
jours[6] = dimanche
```

Rappels : toute expression entière peut être utilisée comme indice

Mutabilité des tableaux

A la différence des chaînes de caractères, il est possible de modifier un élément dans un tableau.

```
jours[1] = "tuesday";

for (let i = 0; i < jours.length; i++) console.log("jours[" + i + "] = " + jours[i]);
```

```
jours[0] = lundi
jours[1] = TUESDAY
jours[2] = mercredi
jours[3] = jeudi
jours[4] = vendredi
jours[5] = samedi
jours[6] = dimanche
```

```
let tab = new Array<number>(1, 2, 3, 4, 5);

for (let i = 0; i < tab.length; i++) tab[i] = 2 * tab[i] + 1;

console.log(tab);
```

```
[ 3, 5, 7, 9, 11 ]
```

Validité des indices

Attention : TypeScript ne vérifie pas la validité d'un indice

1. si l'indice n'est pas un entier ou est strictement négatif, l'accès retourne la valeur `undefined`

```
let tab = [1, 2, 3, 4, 5];
console.log(tab[1.5]);
```

```
undefined
```

2. si l'indice n'est pas un entier ou est strictement négatif, l'affectation fait *quelque chose* (mais il est peu probable que ce soit ce que vous vouliez)

```
let tab = [1, 2, 3, 4, 5];
tab[1.5] = 42;
tab[-1] = -42;
console.log(tab);
```

```
[ 1, 2, 3, 4, 5, '1.5': 42, '-1': -42 ]
```

3. Si l'indice est bien un entier entre 0 et la longueur du tableau - 1, l'accès retourne la valeur de l'élément et l'affectation modifie l'élément

```
let tab = [1, 2, 3, 4, 5];
tab[2] = 42;
let x = 2 * tab[2];

console.log("tab =", tab);
console.log("x =", x);
```

```
tab = [ 1, 2, 42, 4, 5 ]
x = 84
```

4. si l'indice est supérieur ou égal à la longueur du tableau :
    1. la taille du tableau est modifiée pour être égale à l'indice plus 1
    2. les nouveaux éléments reçoivent la valeur 10

```
let tab = [1, 2, 3, 4, 5];
tab[8] = 42;

for (let i = 0; i < tab.length; i++) console.log("tab[" + i + "] = " + tab[i]);
```

```
tab[0] = 1
tab[1] = 2
tab[2] = 3
tab[3] = 4
tab[4] = 5
tab[5] = undefined
tab[6] = undefined
tab[7] = undefined
tab[8] = 42
```

Ajouter un élément à la fin d'un tableau

Un tableau est un **objet** ≈ des données + des fonctions pour accéder / modifier. L'utilisation d'une méthode se fait par une notation `.` sur une variable de type tableau.

Parmi les méthodes définies sur les tableaux, la méthode `push` permet d'ajouter un élément à la fin d'une liste.

Par exemple, le programme suivant construit le tableau contenant les entiers pairs plus petits strictement que `n`.

```
let n = 10;

let tab = new Array<number>();

for (let i = 0; i < n; i += 2) {
  console.log("Itération pour i =", i);
  console.log("tab avant push =", tab);
  tab.push(i);
  console.log("tab après push =", tab);
  console.log();
}

console.log("tab après la boucle");
console.log(tab);
```

```
Itération pour i = 0
tab avant push = []
tab après push = [ 0 ]
```

```
Itération pour i = 2
tab avant push = [ 0 ]
tab après push = [ 0, 2 ]

Itération pour i = 4
tab avant push = [ 0, 2 ]
tab après push = [ 0, 2, 4 ]

Itération pour i = 6
tab avant push = [ 0, 2, 4 ]
tab après push = [ 0, 2, 4, 6 ]

Itération pour i = 8
tab avant push = [ 0, 2, 4, 6 ]
tab après push = [ 0, 2, 4, 6, 8 ]

tab après la boucle
[ 0, 2, 4, 6, 8 ]
```

Retirer un élément à la fin d'un tableau

La méthode `pop` permet de retirer le dernier élément d'un tableau. La méthode retourne l'élément qui vient d'être retiré.

```
let tab = [0, 2, 4, 8];

while (tab.length > 0) {
  console.log("tab avant pop =", tab);
  console.log("élément retiré =", tab.pop());
  console.log("tab après pop =", tab);
  console.log();
}
```

```
tab avant pop = [ 0, 2, 4, 8 ]
élément retiré = 8
tab après pop = [ 0, 2, 4 ]

tab avant pop = [ 0, 2, 4 ]
élément retiré = 4
tab après pop = [ 0, 2 ]

tab avant pop = [ 0, 2 ]
élément retiré = 2
tab après pop = [ 0 ]

tab avant pop = [ 0 ]
```

```
élément retiré = 0
tab après pop = []
```

Appliquer `pop` sur un tableau vide ne cause pas d'erreur et retourne la valeur `undefined`.

```
let tab = new Array<number>(); // tab est un tableau vide

console.log("on tente de retirer un élément =", tab.pop());
```

```
on tente de retirer un élément = undefined
```

Parcours de tableau

Pour parcourir un tableau, on peut faire une boucle sur une variable qui sert d'indice :

```
let tab = [0, 2, 4, 8];

for (let i = 0; i < tab.length; ++i) console.log("tab[" + i + "] = " + tab[i]);
```

```
tab[0] = 0
tab[1] = 2
tab[2] = 4
tab[3] = 8
```

Comme pour les chaînes de caractères, on peut utiliser la boucle `for ... of` pour itérer directement sur les valeurs du tableau :

```
let tab = [0, 2, 4, 8];

for (let valeur of tab) console.log(valeur);
```

Pour les tableaux, il existe une troisième boucle `for ... in` qui itère sur les indices :

```
let tab = [0, 2, 4, 8];

for (let i in tab) console.log("tab[" + i + "] = " + tab[i]);
```

Remarque : dans ce cas, la variable `i` est de type `string`.

Aliasing

Question : que se passe-t'il quand on affecte une variable de type tableau à une autre variable de type tableau ?

Considérons le programme suivant :

```
let t1 = [1, 2, 3, 4, 5];
let t2 = t1;

console.log(t1);
console.log(t2);
```

```
[ 1, 2, 3, 4, 5 ]
[ 1, 2, 3, 4, 5 ]
```

Question : quel est le rapport entre `t1` et `t2` ?

- **Possibilité 1 :** `t1` et `t2` font référence à deux tableaux distincts qui ont la même valeur (la séquence `1, 2, 3, 4, 5`)
- **Possibilité 2 :** `t1` et `t2` font référence au même tableau, ce qui signifie que modifier `l1` entraîne une modification de `l2` et vice versa.

Faisons un test :

```
let t1 = [1, 2, 3, 4, 5];
let t2 = t1;

t2[2] = 42;

console.log(t1);
console.log(t2);
```

```
[ 1, 2, 42, 4, 5 ]
[ 1, 2, 42, 4, 5 ]
```

On est donc dans le second cas.

L'association d'une variable avec un tableau est appelée une **référence**.

Un même tableau peut être donc être référencé par plusieurs variables.

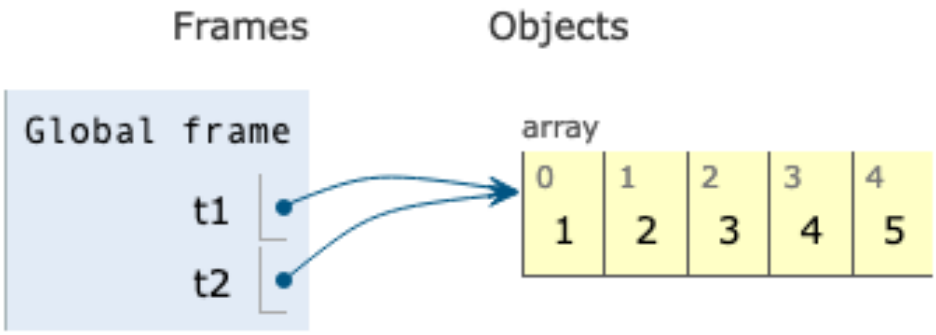
La modification d'un tableau via une variable qui le référence est donc visible via toutes les autres variables qui le référencent.

Cet effet est appelé **aliasing**.

On peut illustrer cela du point de la mémoire. Considérons le code :

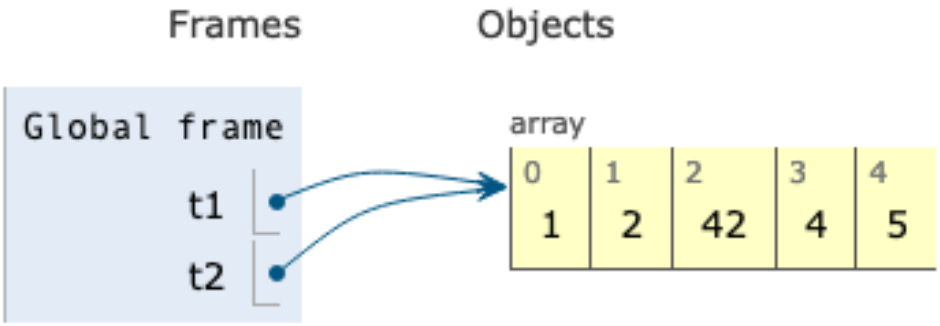
```
let t1 = [1, 2, 3, 4, 5];
let t2 = t1;
```

En mémoire, `t1` et `t2` font bien référence au même tableau :



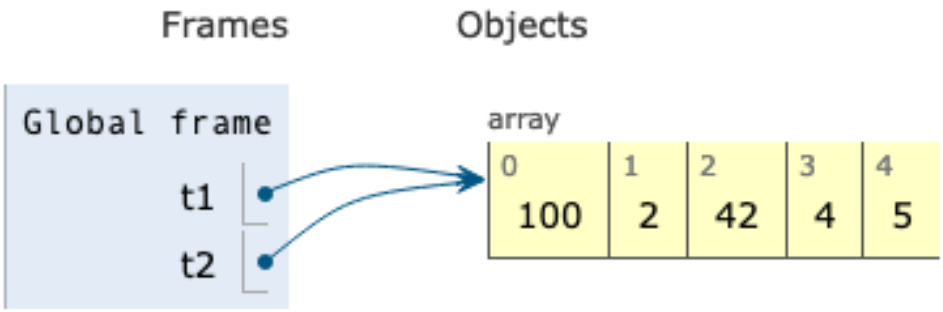
Si on modifie `t2`, c'est bien le tableau partagé en mémoire que l'on modifie :

```
t2[2] = 42;
```



De même, si on modifie `t1` :

```
t1[0] = 100;
```



Copie profonde

Question : comment faire une copie d'un tableau qui crée un nouveau tableau ? (c'est que l'on appelle une **copie profonde**)

Solution 1 : écrire sa propre fonction

```
function copie(t: Array<number>): Array<number> {
  let nouveau: Array<number> = new Array<number>();

  for (let valeur of t) nouveau.push(valeur);

  return nouveau;
}

let t1 = [1, 2, 3, 4, 5];
let t2 = copie(t1);
```

Solution 2 (meilleure) : utiliser l'opérateur **spread** `...`

```
let t1 = [1, 2, 3, 4, 5];
let t2 = [...t1];
```

Comparaison de tableaux

Considérons les instructions suivantes :

```
let t1: Array<number> = [1, 2, 3, 4, 5];
let t2: Array<number> = [1, 2, 3, 4, 5];

console.log(t1 === t2);
```

Question : Qu'est-ce qui est testée ? Les références ou la séquence ?

Regardons l'affichage :

```
false
```

On teste donc les références et pas la valeur

Pour tester si deux tableaux ont la même séquence, il faut écrire une fonction ad-hoc :

```
function égal(t1: Array<number>, t2: Array<number>): boolean {
  if (a === b) return true;
  if (a == null || b == null) return false;
  if (t1.length != t2.length) return false;

  for (let i in t1) if (t1[i] !== t2[i]) return false;

  return true;
}
```

Tableau et fonction

Lorsqu'un tableau est passé en argument d'une fonction, on passe la référence sur l'objet et non une copie.

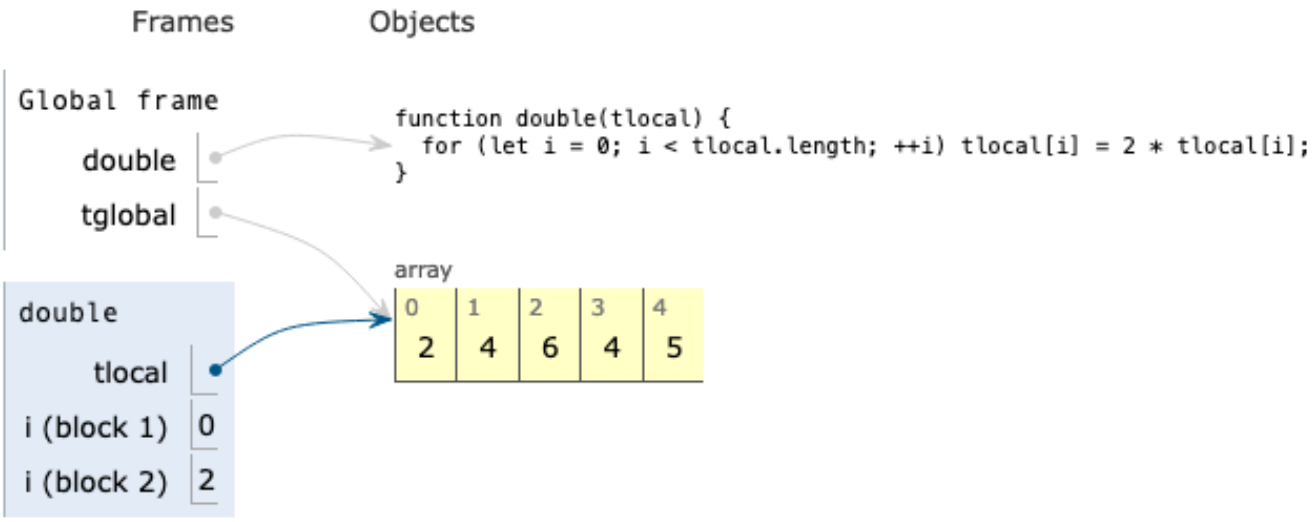
Conséquence : si la fonction modifie le tableau, la modification est visible au niveau de l'appel.

```
function double(tlocal: Array<number>): void {
  for (let i in tlocal) tlocal[i] = 2 * tlocal[i];
}

let tglobal = [1, 2, 3, 4, 5];
double(tglobal);
```

```
[ 2, 4, 6, 8, 10 ]
```

Voici ce qui se passe du point de vue de la mémoire :



Une fonction peut très bien avoir un tableau en type de retour. Il faut allouer (créer) le tableau dans la fonction 😒

```
function créeTableauAléatoire(n: number): Array<number> {
  let t = new Array<number>();

  for (let i = 0; i < n; i++) t.push(Math.random());

  return t;
}

let nouveauTableaAléatoire = créeTableauAléatoire(10);
```