Javascript cours n°2

A l'issue de ce cours vous saurez utiliser les tableaux en Javascript

Deux types de tableaux

- Sans forcément parler de langage :
  - Tableaux classiques : la clé est un entier
  - Tableaux associatifs : la clé est de n’importe quel type :
    - Chaîne
    - Objet
    - Etc.

Déclaration

- Rappel : trois possibilités
  - Utilisation du constructeur d'`Array`
  - Utilisation littérale recommandée → [ ]

```
let o0=new Array();
let o1=Array();
let o2=[];
let o3=[5, 6, 7];
let o4=Array( items: 5,6,7);
let o5=new Array( items: 5,6,7);
let o6=Array( arrayLength: 3);
let o7=new Array( arrayLength: 3);
```

Boucles sur un tableau

- `for` classique ou *foreach* : affichage des valeurs

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

for (const elt of tab) {
    console.log(elt);
}
```

- *foreach* également possible sur les indices / clés, à ne pas confondre !

```
for (const elt in tab) {
    console.log(elt);
}
```

Propriété `length`

- Nombre de cases du tableau
- Accessible en écriture !

```
let tab = [3, 1, 7];
console.log(tab.length);
console.log(tab);

let tab2 = [];
tab2.length=5;
console.log(tab2.length);
console.log(tab2);
```

```
3
▸ Array(3) [ 3, 1, 7 ]
5
▸ Array(5) [ <5 empty slots> ]
```

- Ce qui peut être dangereux…

```
let tab = [3, 1, 7];
console.log(tab);
tab.length=2;
console.log(tab);
```

```
▸ Array(3) [ 3, 1, 7 ]
▸ Array [ 3, 1 ]
```

- Un tableau est un objet, autre possibilité d'accès à une propriété :

```
console.log(tab2["length"]);
```

Fonctions applicables

- `push` : ajout d'éléments en fin de table

```
let tab = [3, 1, 7];
console.log(tab);
tab.push(10, 20);
console.log(tab);
```

```
▸ Array(3) [ 3, 1, 7 ]
▸ Array(5) [ 3, 1, 7, 10, 20 ]
```

- `unshift` : ajout d'éléments en début de table

```
let tab = [3, 1, 7];
console.log(tab);
tab.unshift( items: 10, 20);
console.log(tab);
```

```
▸ Array(3) [ 3, 1, 7 ]
▸ Array(5) [ 10, 20, 3, 1, 7 ]
```

Fonctions applicables

- `pop` : retrait de l'élément en fin de table

```
let tab = [3, 1, 7];
console.log(tab);
tab.pop();
console.log(tab);
```

```
▸ Array(3) [ 3, 1, 7 ]
▸ Array [ 3, 1 ]
```

- `shift` : retrait de l'élément en début de table

```
let tab = [3, 1, 7];
console.log(tab);
tab.shift();
console.log(tab);
```

```
▸ Array(3) [ 3, 1, 7 ]
▸ Array [ 1, 7 ]
```

- Dans les deux cas, aucune erreur si le tableau est déjà vide

Fonctions applicables

- `slice` : création d'un nouveau tableau avec des éléments du tableau source (*slice* = tranche)

```
let tab = [3, 1, 7, 10, 20];
let tab2 = tab.slice(2,4);
console.log(tab2);
```

```
▶ Array [ 7, 10 ]
```

- `concat` : création d'un nouveau tableau en concaténant deux tableaux source

```
let tab = [3, 1, 7];
let tab2 = [10, 20];
let tab3 = tab.concat(tab2);
console.log(tab3);
```

```
▶ Array(5) [ 3, 1, 7, 10, 20 ]
```

Fonctions applicables

- `splice` : suppression, et éventuel ajout à la place, d'éléments

```
let tab = [3, 1, 7, 10, 20];
tab.splice( start: 2,  deleteCount: 1,  items: 8, 9);
console.log(tab);
```

```
▶ Array(6) [ 3, 1, 8, 9, 10, 20 ]
```

- `reverse` : inversion des éléments

```
let tab = [3, 10, 7, 1, 20];
tab.reverse();
console.log(tab);
```

```
▶ Array(5) [ 20, 1, 7, 10, 3 ]
```

- `indexOf, lastIndexOf, includes` :

```
let tab = [3, 10, 7, 1, 20, 10];
console.log(tab.indexOf(10), tab.indexOf(44));
console.log(tab.lastIndexOf(10), tab.lastIndexOf(44));
console.log(tab.includes(10), tab.includes(44));
```

```
1 -1
5 -1
true false
```

Programmation fonctionnelle

- `sort` : tri du tableau, avec ou sans fonction de *callback*

```
let tab = [3, 10, 7, 1, 20];
tab.sort();
console.log(tab);
```

```
tab.sort( compareFn: (a :number , b :number ) => {
    return a < b ? -1 : a > b ? 1 : 0;
})
console.log(tab);
```

```
function intReverseSorter(a, b) {
    let val=0;
    if (a < b) {
        val=1;
    } else if (a > b) {
        val = -1;
    }
    return val;
}
tab.sort(intReverseSorter)
console.log(tab);
```

- résultats des trois versions :

```
▸ Array(5) [ 1, 10, 20, 3, 7 ]
▸ Array(5) [ 20, 10, 7, 3, 1 ]
▸ Array(5) [ 1, 3, 7, 10, 20 ]
```

Programmation fonctionnelle

- Autres fonctions avec *callback*, très puissantes, certaines très utilisées dans les frameworks JS

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

let tab2 = tab.filter( elt => elt % 2===0);
console.log(tab2);

let tab3 = tab.map(elt => elt * 10);
console.log(tab3);

tab.forEach(console.log);

let tousPair = tab.every(elt => elt%2===0)
console.log(tousPair);

let existePair = tab.some(elt => elt%2===0)
console.log(existePair);
```

```
▸ Array [ 2, 4 ]
▸ Array(4) [ 10, 20, 30, 40 ]
1 0 ▸ Array(4) [ 1, 2, 3, 4 ]
2 1 ▸ Array(4) [ 1, 2, 3, 4 ]
3 2 ▸ Array(4) [ 1, 2, 3, 4 ]
4 3 ▸ Array(4) [ 1, 2, 3, 4 ]
false
true
```

Programmation fonctionnelle

- Remarque : les arrow functions ne sont pas obligatoires
- Les deux codes suivants sont équivalents :

```
const t3 = t1.map(function (n) { return n * 10; });

const t4 = t1.map(n => n * 10);
```

Tableaux à plusieurs dimensions

- Chaque case de tableau peut contenir, un tableau, ce qui permet d'obtenir des tableaux à *n* dimensions
- La taille de la deuxième dimension peut être initialisée pour chaque case (potentiellement différente)

```
let tab = new Array( arrayLength: 10);
for (let i = 0; i < tab.length; i++) {
    tab[i] = new Array( arrayLength: 10);
    for (let j = 0; j < tab[i].length; j++) {
        tab[i][j] = "[" + i + "," + j + "]";
        console.log(tab[i][j]);
    }
}
```

- Rappel : ce qui suit crée un tableau de 2 cases !!!

```
let tab2 = new Array( items: 10,10);
console.log(tab2)
```

Tableaux : gestion par référence

- Attention aux copies de tableaux
  - Affectation : copie de la référence (du pointeur vers le tableau)
  - Les deux variables pointent vers le même tableau

```
const tab = [2, 3, 4];

const tabReferenceCopy = tab;

tabReferenceCopy[0] = 2000;
tabReferenceCopy.push(5000);

console.log("tab", tab);
console.log("tabReferenceCopy", tabReferenceCopy);
```

```
tab ▸ Array(4) [ 2000, 3, 4, 5000 ]
tabReferenceCopy ▸ Array(4) [ 2000, 3, 4, 5000 ]
```

Tableaux : gestion par référence

- Attention aux copies de tableaux
  - Utilisation de l'**opérateur de décomposition** … (*spread operator*)
  - Création d'un nouveau tableau en mettant à l'intérieur la copie des cases du premier
  - Les deux variables pointent chacune vers un tableau différent

```
const tab = [2, 3, 4];
const tabCopy = [...tab];

tabCopy[0] = 2000;
tabCopy.push(5000);

console.log("tab", tab);
console.log("tabCopy", tabCopy);
```

```
tab ▸ Array(3) [ 2, 3, 4 ]
tabCopy ▸ Array(4) [ 2000, 3, 4, 5000 ]
```

Tableaux : gestion par référence

- Attention la solution précédente n'est pas une *deep copy* lorsque le tableau contient des éléments manipulés par référence (tableaux, objets)

- Exemple

```
const o1 = { id: 5, nom: "1a"};
const o2 = { id: 8, nom: "2a"};
const o3 = { id: 9, nom: "lp wmce"};

const tab = [o1, o2];
const tabCopy = [...tab];

tabCopy[0] = o3;
tabCopy[1].nom="les 2a";

console.log("tab", tab);
console.log("tabCopy", tabCopy);
```

```
tab ▼ (2) […]
    ▶ 0: Object { id: 5, nom: "1a" }
    ▶ 1: Object { id: 8, nom: "les 2a" }
      length: 2
    ▶ <prototype>: Array []
```

```
tabCopy
▼ (2) […]
  ▶ 0: Object { id: 9, nom: "lp wmce" }
  ▶ 1: Object { id: 8, nom: "les 2a" }
    length: 2
  ▶ <prototype>: Array []
```

Tableaux associatifs

- Cela n'existe pas en Javascript, mais on peut utiliser des objets, en utilisant `[ ]` au lieu de `.`

```
let promos = {}
promos["lpwmce"] = {"nb" : 25, "rentree" : "21/09/2020"}
promos["lpasr"] = {"nb" : 15, "rentree" : "14/09/2020"}
for (const promo in promos) {
    console.log(promo);
    console.log(promos[promo]);
}
```

```
lpwmce
▸ Object { nb: 25, rentree: "21/09/2020" }
lpasr
▸ Object { nb: 15, rentree: "14/09/2020" }
```

- `Object.keys(promos)` pour obtenir un tableau des clés

Tableaux associatifs

- Le code suivant donne des choses étranges...

```
let tab3 = [];
tab3["lpwmce"] = {"nb" : 25, "rentree" : "21/09/2020"};
tab3["lpasr"] = {"nb" : 15, "rentree" : "14/09/2020"};

console.log(tab3["lpwmce"])
console.log(tab3.length);
console.log(tab3)
```

```
▸ Object { nb: 25, rentree: "21/09/2020" }
0
▾ []
    length: 0
  ▸ lpasr: Object { nb: 15, rentree: "14/09/2020" }
  ▸ lpwmce: Object { nb: 25, rentree: "21/09/2020" }
  ▸ <prototype>: Array []
```

- Il s'agit d'un objet auquel on a ajouté des propriétés (*lpasr, lpwmce*) !!!
  → à ne pas faire !