JS : Les collections

L2 MPCIE - UE Développement Web

David Lesaint

david.lesaint@univ-angers.fr

Janvier 2019

Les collections

Deux sortes de collections

- Les collections indexées : tableaux.
- Les collections avec clés : ensembles, tables de hachage.

Les tableaux

Pas de type primitif “tableau”.

Les tableaux sont des objets `Array`.

L’objet natif `Array`

Fournit propriétés et méthodes pour créer et manipuler des tableaux indicés :

- longueur et propriétés en lien avec les regex.
- fonctions d’itération, accès, ajout, retrait, inversion, tri, fusion,...

Le prototype `Array.prototype`

Méthodes	Description
`copyWithin`	Copie une série d'éléments de tableau dans lui-même.
`fill`	Remplit tous les éléments du tableau avec une même valeur, éventuellement entre un indice de début et un indice de fin.
`pop`	Supprime le dernier élément du tableau et retourne cet élément.
`push`	Ajoute un ou plusieurs éléments à la fin du tableau et retourne la longueur résultante du tableau.
`reverse`	Inverse l'ordre des éléments du tableau.
`shift`	Supprime le premier élément du tableau et retourne cet élément.
`sort`	Trie les éléments d'un tableau et retourne le tableau.
`splice`	Permet d'ajouter ou de retirer des éléments du tableau.
`unshift`	Ajoute un ou plusieurs éléments au début du tableau et renvoie la longueur résultante du tableau.
`concat`	Concatène le tableau concaténé avec le(s) tableau(x) et/ou valeur(s) en arguments et renvoie le tableau résultant.
`includes`	Teste si le tableau contient ou non un certain élément.
`indexOf` / `lastIndexOf`	Retourne le plus petit/grand index d'un élément égal à la valeur passée en paramètre à l'intérieur du tableau, ou -1 sinon.
`join`	Concatène avec délimiteur tous les éléments d'un tableau en une chaîne.
`slice`	Extrait une portion du tableau pour retourner un nouveau tableau constitué de ces éléments.
`toSource`	Renvoie la représentation littérale du tableau spécifié.
`toString` / `toLocaleString`	Renvoie une chaîne de caractères représentant le tableau et ses éléments (/en tenant compte de la locale).

Construction de tableau

Instructions

- `var arr = new Array(élément0, élément1, ..., élémentN);`
- `var arr = Array(élément0, élément1, ..., élémentN);`
- `var arr = [élément0, élément1, ..., élémentN];`
- `var arr = new Array(longueurTableau);`
- `var arr = Array(longueurTableau);`

collections-array-construction.js

```
1  var arr = new Array(1, 2, 3);
2  var arr = Array(1, 2, 3);
3  var arr = [ 1, 2, 3 ]; // littéral de tableau
4
5  /* Tableau de longueur 4 mais non initialisé */
6  var arr = new Array(4);
7  // Alternatives :
8  var arr = Array(4);
9  var arr = []; arr.length = 4;
10
11 /* !! Utiliser un littéral pour tableau à un élément numérique */
12 var arr1 = [42];
```

Remplissage de tableau

Instructions

- `var arr = [];`
- `arr[0] = élément0;`
- `arr[1] = élément1; ...`
- `arr[n] = élémentN;`

collections-array-remplissage.js

```
1 var tab = [];
2 tab[2] = "Louise Michel";
3 tab[0] = "Edouard Vaillant";
4 tab[1] = "Jules Vallès";
5 console.log(tab);
```

Accès aux éléments d'un tableau

Par indice à partir de 0

Un tableau stocke chaque élément comme une propriété dont le nom est l'indice de l'élément.

collections-array-acces.js

```
1 var arr = [ "un", "deux", "trois" ];
2 arr[2]; // "trois"
3 arr["length"]; // 3 - on accède à la propriété "length" du tableau
```

Longueur de tableau

Stockée dans la propriété `length`

- Egale au plus grand indice des éléments plus 1.
- Peut être explicitement réduite pour tronquer le tableau.

collections-array-length.js

```
1  var chats = [];
2  chats[30] = [ 'FatCat' ];
3  console.log(chats.length); // 31
4
5  var chats = [ 'in', 'in', 'out' ];
6  console.log(chats.length); // 3
7  chats.length = 2;
8  console.log(chats); // "in,in" - out a été retiré
9  cats.length = 0;
10 console.log(chats); // tableau vide
11 cats.length = 3;
12 console.log(chats); // [undefined, undefined, undefined]
```

Parcours de tableau

Avec boucle `for`

collections-array-for.js

```
1 var couleurs = ['bleu', 'blanc', 'rouge'];
2 for (var i = 0; i < couleurs.length; i++) {
3   console.log(couleurs[i]);
4 }
5
6 var divs = document.getElementsByTagName('div');
7 for (var i = 0, div; div = divs[i]; i++) {
8   /* traitement sur les   div */
9 }
```

Ne pas utiliser `for...in`

Parcourt toutes les propriétés énumérables d'un tableau.

Parcours de tableau

Avec boucle `foreach`

Ne parcourt pas les éléments sans valeur, sauf si explicitement initialisés à `undefined`.

collections-array-foreach.js

```
1  var T = ['ett', 'två', 'tre'];
2  T.forEach(function(e) {console.log(e);});
3  // Avec fonction fléchée
4  T.forEach(tab => console.log(tab));
5
6  T = ['uno', 'dos', , 'quatro'];
7  T.forEach(function(e) {console.log(e);});// ['uno', 'dos', 'quatro'];
8  if(T[2] === undefined)
9    {console.log('T[2] vaut undefined');} // true
10
11 var T = ['eins', 'zwei', undefined, 'vier'];
12 T.forEach(function(e) {console.log(e);});
13 //['eins', 'zwei', undefined, 'vier'];
```

Fusion

- `tab.concat(tab1[,tab2[,...[,tabN]...]])` renvoie un tableau concaténant `tab`, `tab1`,...,`tabN`.
- `tab.join(d=',')` concatène les éléments de `tab` sous forme de chaîne séparés par `d`.

collections-array-concat-join.js

```
1 var tab = new Array("1", "2", "3");
2 tab = tab.concat(["a", "b", "c"]);
3 // ["1", "2", "3", "a", "b", "c"]
4
5 var tab = new Array("Air", "Eau", "Feu");
6 var list = tab.join(" - ");
7 // "Air - Eau - Feu"
```

Ajout et retrait d'éléments

- Ajout/retrait en fin avec `push()`/`pop()`.
- Ajout/retrait au début avec `unshift()`/`shift()`.

collections-array-push-unshift.js

```
1 var tab = new Array("1", "2");
2 tab.push("3"); // ["1", "2", "3"]
3 var dernier = tab.pop(); // dernier="3", tab=["1", "2"]
4
5 var premier = tab.shift(); // premier="1", tab=["2"]
6 tab.unshift("0", "1"); // tab=["0", "1", "2"]
```

Extraction et insertion de plages

- `tab.slice(d,f)` renvoie le tableau d'éléments de `tab` situés aux indices `{d,...,f-1}`.
- `splice(d,D,v1,...,vN)` retire `D` éléments de `tab` à partir de l'indice `d` et y insère `v1,...vN`.

collections-array-slice-splice.js

```
1 var tab = new Array ("a", "b", "c", "d", "e");
2 tab = tab.slice(1, 4); //[ "b", "c", "d"]
3
4 var tab = new Array ("1", "2", "3", "4", "5");
5 tab.splice(1, 3, "a", "b", "c", "d"); //["1", "a", "b", "c", "d", "5"]
```

Inversion et tri

- `tab.reverse()` inverse les positions d'éléments de `tab`.
- `tab.sort([f])` trie `tab`.
  - Critère de tri donné par la fonction de rappel `f` qui doit retourner `-1`, `0` ou `1`.

collections-array-reverse-sort.js

```
1  var tab = new Array("1", "2", "3");
2  tab.reverse(); // [ "3", "2", "1" ]
3  var tab = new Array("Air", "Feu", "Eau");
4  tab.sort(); // [ "Air", "Eau", "Feu" ]
5
6  var sortFn = function(a, b) {
7    if (a[a.length - 1] < b[b.length - 1])
8      return -1;
9    if (a[a.length - 1] > b[b.length - 1])
10     return 1;
11   if (a[a.length - 1] == b[b.length - 1])
12     return 0;
13 }
14 tab.sort(sortFn); // ["Air","Feu","Eau"]
```

Recherche d'indice de valeur

- Première occurrence de `v` avec `indexOf(v[,idDepart])`.
- Dernière occurrence de `v` avec `lastIndexOf(v[,idDepart])`.

collections-array-indexof.js

```
1  var a = [ 'a', 'b', 'a', 'b', 'a' ];
2  console.log(a.indexOf('b')); // 1
3  // Recherche après la première correspondance :
4  console.log(a.indexOf('b', 2)); // 3
5  console.log(a.indexOf('z')); // -1 car 'z' n'a pas été trouvé
6
7  var a = [ 'a', 'b', 'c', 'd', 'a', 'b' ];
8  console.log(a.lastIndexOf('b')); // 5
9  // Recherche après la première correspondance en fin de tableau
10 console.log(a.lastIndexOf('b', 4)); // 1
11 console.log(a.lastIndexOf('z')); // -1
```

Méthodes itératives

- `tab.foreach(f[,thisArg])` exécute `f` sur chaque élément de `tab` et renvoie `undefined`.
- `tab.map(f[,thisArg])` renvoie le tableau de l'exécution de `f` sur chaque élément de `tab`.
  - `f(vCourante[,idCourant[,tabUtilisé]])` .
  - `thisArg` : valeur à utiliser pour `this` lors de l'exécution de `f`.

collections-array-foreach-map.js

```
1  function logArrayElements(element, index, array) {
2    console.log("a[" + index + "] = " + element);
3  } // a[0] = a //a[1] = b //a[3] = c
4  a = [ "a", "b", , "c" ];
5  a.forEach(logArrayElements);
6
7  var a1 = a.map(function(item) {
8    return item.toUpperCase();
9  });
10 console.log(a1); // A,B,C
```

Méthodes itératives

- `tab.every(f[,thisArg])` renvoie `true` si `f` renvoie `true` pour chaque élément de `tab`.
- `tab.some(f[,thisArg])` renvoie `true` si `f` renvoie `true` pour au moins un élément de `tab`.
  - `f` : fonction de rappel booléenne.

collections-array-every-some.js

```
 1 function isNumber(value) {
 2   return typeof value === 'number';
 3 }
 4 var a1 = [ 1, 2, 3 ];
 5 console.log(a1.every(isNumber)); // true
 6 var a2 = [ 1, '2', 3 ];
 7 console.log(a2.every(isNumber)); // false
 8
 9 console.log(a1.some(isNumber)); // true
10 var a3 = [ '1', '2', '3' ];
11 console.log(a3.some(isNumber)); // false
```

Méthodes itératives

- `tab.filter(f[,thisArg])` renvoie le tableau des éléments de `tab` pour lequels `f` renvoie `true`.
  - `f` : fonction de rappel booléenne.
- `tab.reduce(f[,vInit])` réduit `tab` en appliquant `f` à l'ensemble de ses éléments.
  - `f(vAccumulée[,vCourante[,idCourant[,tabUtilisé]]])`.
  - `vInit` : valeur utilisée en 1er argument au 1er appel de `f`.

collections-array-filter-reduce.js

```
1  var a1 = [ 'a', 10, 'b', 20, 'c', 30 ];
2  var a2 = a1.filter(function(item) {
3    return typeof item == 'number';
4  });
5  console.log(a2); // 10,20,30
6
7  var a = [ 10, 20, 30 ];
8  var total = a.reduce(function(vAccumulée, vCourante) {
9    return vAccumulée + vCourante;
10 }, -60);
11 console.log(total) // 0
```

Les tableaux multi-dimensionnels

Un élément de tableau peut être un tableau.

collections-array-multidimensionnel.js

```
 1 var a = new Array(4);
 2 for (i = 0; i < 4; i++) {
 3   a[i] = new Array(4);
 4   for (j = 0; j < 4; j++) {
 5     a[i][j] = "[" + i + "," + j + "]";
 6   }
 7 }
 8
 9 console.log(a.map(row => row.map(cell => cell+" ").join("
")).join("\n"));
10 /*
11 [0,0]  [0,1]  [0,2]  [0,3]
12 [1,0]  [1,1]  [1,2]  [1,3]
13 [2,0]  [2,1]  [2,2]  [2,3]
14 [3,0]  [3,1]  [3,2]  [3,3]
15 */
```

Les objets semblables à des tableaux

`arguments`, `String`, `NodeList`, ...

Peuvent être manipulés comme des tableaux à l'aide du prototype de `Array`.

collections-array-like.js

```
1 function alertArguments() {
2   Array.prototype.forEach.call(arguments, function(item) {
3     console.log(item);
4   });
5 }
6
7 Array.prototype.forEach.call("une chaîne", function(chr) {
8   console.log(chr);
9 });
```

Les tableaux associatifs

Avec les objets `Map`

Collection de paires clés/valeurs.

- Clés et valeurs peuvent être de n'importe quel type.
- Itération selon l'ordre d'insertion avec `for...of`.

collections-map.js

```
1  var sayings = new Map();
2  sayings.set("dog", "woof");
3  sayings.set("cat", "meow");
4  sayings.set("elephant", "toot");
5  sayings.size; // 3
6  sayings.get("fox"); // undefined
7  sayings.has("bird"); // false
8  sayings.delete("dog");
9
10 for (var [key, value] of sayings) {
11   console.log(key + " goes " + value);
12 }
13 // "cat goes meow"
14 // "elephant goes toot"
```

Le prototype `Map.prototype`

Propriétés	Description
`size`	Le nombre de paires de clé-valeur contenues dans le tableau associatif.

Méthodes	Description
`clear()`	Supprime toutes les paires clé-valeur du tableau associatif.
`delete(k)`	Supprime l'élément de clé `k` du tableau associatif et renvoie `true`. Renvoie `false` si aucun élément n'existe avec cette clé.
`entries`	Renvoie un nouvel objet `Iterator` qui contient la paire clé/valeur pour chaque élément du tableau associatif, dans leur ordre d'insertion.
`foreach(f [,thisArg])`	Appelle la fonction `f` pour chaque paire clé-valeur tableau associatif dans leur ordre d'insertion. Si le paramètre `thisArg` est fourni, il sera utilisé comme valeur pour `this` pour chaque appel de `f`.
`get(k)`	Renvoie la valeur associée à la clé `k` et `undefined` s'il n'y en a pas.
`has(k)`	Renvoie un booléen indiquant si une valeur associée à la clé `k` a été trouvée dans le tableau associatif.
`keys()`	Renvoie un nouvel objet `Iterator` contenant les clés de chaque élément du tableau associatif dans leur ordre d'insertion.
`set(k,v)`	Associe la valeur `v` à la clé `k` dans le tableau associatif. Renvoie `undefined`.
`values()`	Renvoie un nouvel objet `Iterator` contenant les valeurs de chaque élément du tableau associatif dans leur ordre d'insertion.

Les ensembles

Avec les objets `Set`

Ensembles de valeurs.

- Une valeur ne peut apparaître qu'une seule fois.
- Itération selon l'ordre d'insertion avec `for...of`.
- Conversion en tableau avec `Array.from`.

collections-set.js

```
1  var monEnsemble = new Set();
2  monEnsemble.add(1);
3  monEnsemble.add("texte");
4  monEnsemble.add("toto");
5  monEnsemble.has(1); // true
6  monEnsemble.delete("toto");
7  monEnsemble.size; // 2
8  for (let item of monEnsemble) console.log(item);// 1 puis "texte"
9  T = Array.from(monEnsemble);
10 T.forEach(function(e) {console.log(e);});// 1 puis "texte"
```

Le prototype `Set.prototype`

Propriétés	Description
`size`	Le nombre de valeurs contenues dans l'ensemble.

Méthodes	Description
`add(v)`	Ajoute la valeur `v` à l'ensemble et le renvoie.
`clear()`	Supprime tous les éléments de l'ensemble.
`delete(v)`	Supprime l'élément `v` de l'ensemble et renvoie `true`. Renvoie `false` si cet élément n'appartient pas à l'ensemble.
`entries()`	Renvoie un nouvel objet `Iterator` contenant un tableau de paires valeur-valeur pour chaque élément de l'ensemble dans leur ordre d'insertion (chaque élément est utilisé à la fois comme valeur et comme clé pour chaque paire).
`foreach(f [,thisArg])`	Appelle la fonction `f` sur chaque élément de l'ensemble dans leur ordre d'insertion. Si le paramètre `thisArg` est fourni, il sera utilisé comme valeur pour `this` pour chaque appel de `f`.
`has(v)`	Renvoie un booléen indiquant si la valeur `v` appartient à l'ensemble.
`keys()`	Même méthode que `values()`.
`values()`	Renvoie un nouvel objet `Iterator` contenant les valeurs de l'ensemble dans leur ordre d'insertion.

Implémenter les opérations ensemblistes

collections-set-operations.js

```
Set.prototype.isSuperset = function(subset) {
    for (var elem of subset) {if (!this.has(elem)) {return false;}
    }
    return true;
}
Set.prototype.union = function(setB) {
    var union = new Set(this);
    for (var elem of setB) {union.add(elem);}
    return union;
}
Set.prototype.intersection = function(setB) {
    var intersection = new Set();
    for (var elem of setB) {if (this.has(elem)) {intersection.add(elem);}
    }
    return intersection;
}
Set.prototype.difference = function(setB) {
    var difference = new Set(this);
    for (var elem of setB) {difference.delete(elem);}
    return difference;
}

var setA = new Set([1, 2, 3, 4]),
    setB = new Set([2, 3]),
    setC = new Set([3, 4, 5, 6]);
setA.isSuperset(setB); // => true
setA.union(setC); // => Set [1, 2, 3, 4, 5, 6]
setA.intersection(setC); // => Set [3, 4]
setA.difference(setC); // => Set [1, 2]
```