Objets natifs standards –Built in objects

- JavaScript prédéfinit un certain nombre d'objets globaux (objets natifs standards de portée globale)
  - Objets fondamentaux

Constructeur	Description
Object	classe de base de toutes les classes
Function	représente une fonction sous forme d'objet , sert de <prototype> pour les fonctions
Boolean	représente le type primitif booléen sous forme d'objet
Symbol	utilisé pour représenter des identifiants pour des propriétés d'un objet.
Error	classe d'erreur générique
EvalError	classe d'erreurs survenant lors de l'interprétation de code JavaScript par `eval`
RangeError	classe d'erreurs survenant lors de l'utilisation de nombres dépassant les bornes autorisées (MIN_VALUE, MAX_VALUE)
ReferenceError	classe d'erreurs survenant lors de l'utilisation d'une référence incorrecte
SyntaxError	classe relative aux erreurs de syntaxe
TypeError	classe relative aux erreurs de typage
URIError	classe relative aux erreurs d'utilisation des méthodes de traitement d'URI de Globals
SyntaxError	classe relative aux erreurs de syntaxe

Objets prédéfinis en JavaScript

- Nombres et dates

Constructeur	Description
Number	représente un nombre (entier, réel) sous forme d'objet
Math	objet natif dont les méthodes et propriétés permettent l'utilisation de constantes et fonctions mathématiques
Date	représente une date sous forme d'objet

- Manipulation de textes

Constructeur	Description
`String`	représente un chaîne de caractères sous forme d'objet
RegExp	représente une expression régulière

- Collections

Classe	Description
`Array`	représente un tableau sous forme d'objet
Map	représente un dictionnaire
Set	représente un ensemble
…	…..

- … https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux

Tableaux en JavaScript

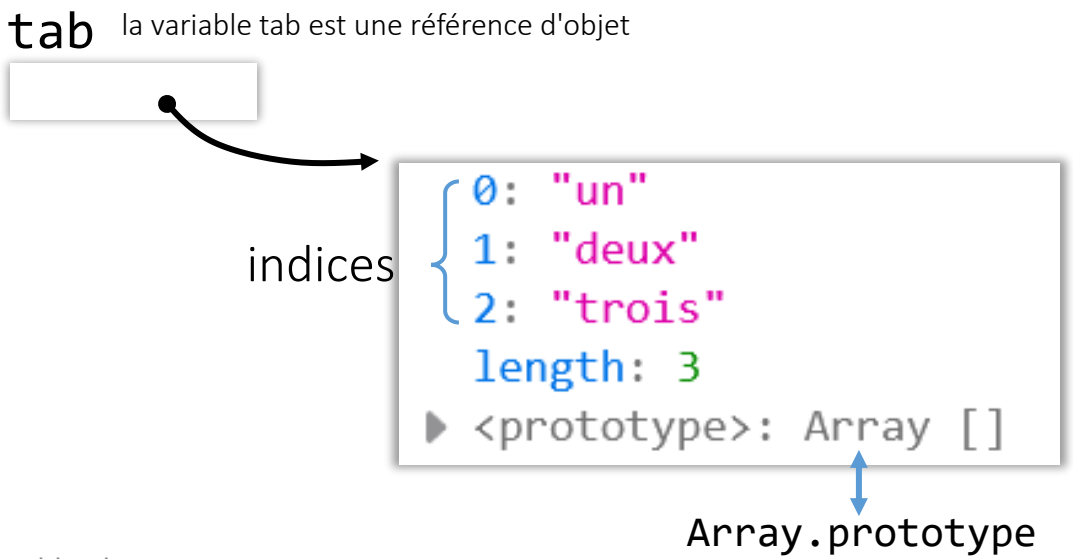
Philippe Genoud
Philippe.Genoud@univ-grenoble-alpes.fr

Tableaux

- Tableau
  - Ensemble ordonné de valeurs accessibles par un indice numérique
- En JavaScript
  - comme en C, C++, Java … cet indice démarre à 0
  - Les tableaux sont des objets
  - `Array` est le type objet prédéfini (fonction constructeur) utilisé pour créer un tableau
  - `length` est un attribut qui donne la taille du tableau

```
let tab = [ "un" , "deux",  "trois" ];
// tab = new Array("un" , "deux",  "trois");
console.log(typeof tab);
console.log(tab.length);
```

```
---> object
---> 3
```

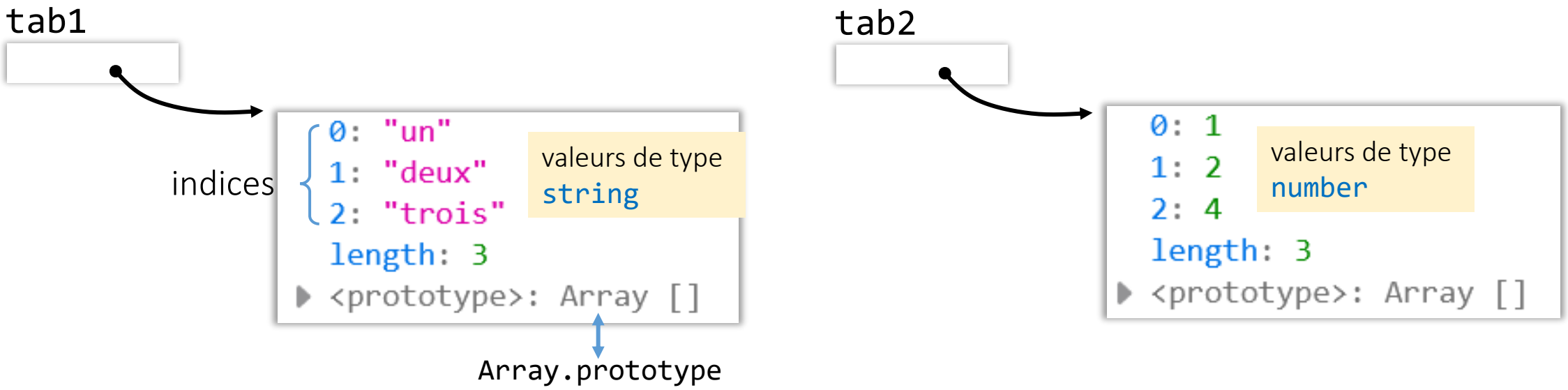


Création d'un tableau

- Expression littérale `[element0, element1, ..., elementN]`
  - Définit un tableau initialisé avec les éléments donnés
  - Les éléments peuvent être de n'importe quel type (valeurs primitives, objets, fonctions)

```
let tab1 = [ "un" , "deux",  "trois" ];
let tab2 = [ 1, 2, 4];
```

Tableaux **homogènes**
Tous les éléments sont de même type

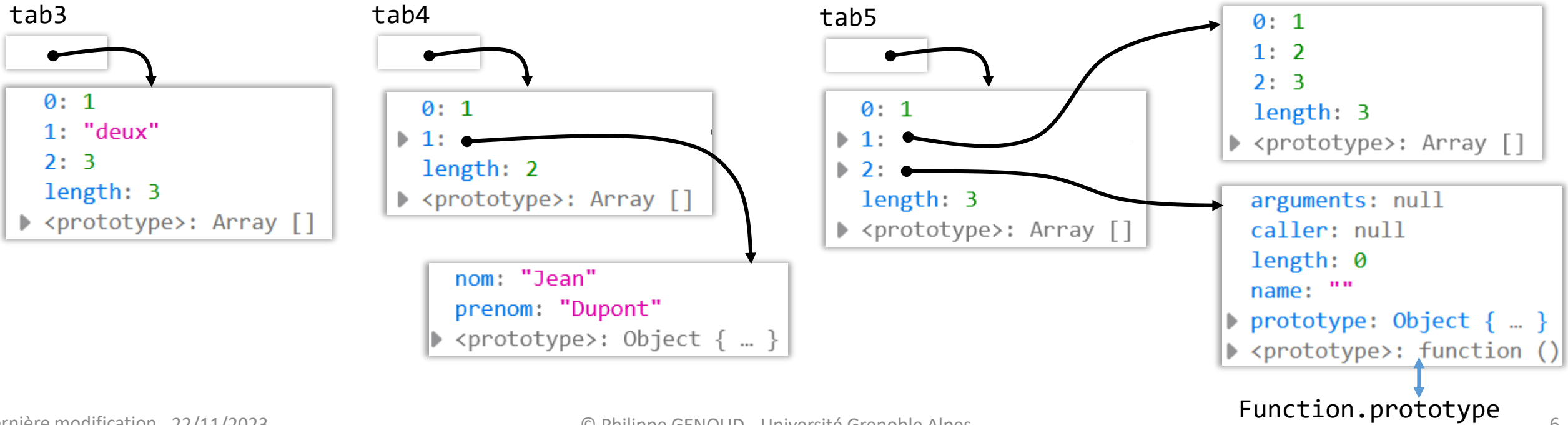


Création d'un tableau

- Expression littérale `[element0, element1, ..., elementN]`
  - Définit un tableau initialisé avec les éléments donnés
  - Les éléments peuvent être de n'importe quel type (valeurs primitives, objets, fonctions)

```
let tab3 = [ 1, "deux", 3];
let tab4 = [ 1, { nom: "Jean", prenom: "Dupont"} ];
let tab5 = [ 1, [ 1, 2, 3], function() { console.log("Hi !") } ];
```

Tableaux hétérogènes
Les éléments ne sont pas tous de même type



Création d'un tableau

- Expression littérale `[element0, element1, ..., elementN]`
  - possibilité de mettre une `,` après le dernier élément (*trailling comma*)

```
let languages =  [
    "JavaScript",
    "Java",
    "C",
    "Python",
    "Rust",
];
```

```
"PHP" ,
```

Toutes les lignes sont identiques

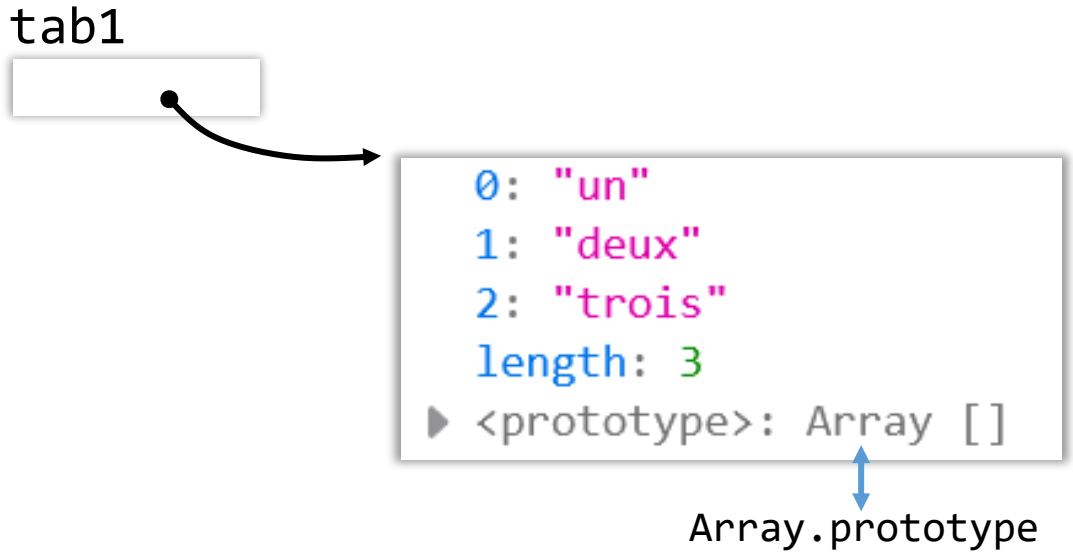
➔

Facilite l'ajout/suppression d'items

Accès aux éléments d'un tableau

- *nomTableau*[*indice*]
  - *indice* expression entière

```
let tab1 = [ "un" , "deux",  "trois" ];

console.log(tab1.length);  ---> 3

console.log(tab1[0]);   ---> "un"

console.log(tab1[1]);   ---> "deux"

console.log(tab1[10]);  ---> undefined

console.log(tab1[-1]);  ---> undefined

console.log(tab1[1.4]); ---> undefined

console.log(tab1["un"]) ---> undefined

console.log(tab1["1"]); ---> "deux"
```



En fait un tableau est un objet
les indices des éléments sont des noms de propriétés que l'on peut accéder par la notation **[ ]**
si on accède à une propriété qui n'est pas définie dans l'objet (où dans sa chaine de **<prototypes>**) on obtient la valeur **undefined**

Les tableaux et objets littéraux

```
function display(x, prop) {
    console.log(typeof x);
    console.log("length : " + x.length);

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

```
let t1 = [
   "un" ,
   "deux",
   "trois"
];

display(t1, "unités");
```

```
object
length :  3
0 : un
1 : deux
2 : trois
```

```
let t2 = {
   0: "un" ,
   1: "deux",
   2: "trois",
   length: 3
} ;

display(t2, "unités");
```

Mais alors , `t1` et `t2` est-ce la même chose ?

Les tableaux et objets littéraux

```
let t1 = [
   "un" ,
   "deux",
   "trois"
];
```

t1 et **t2** est-ce la même chose ?

NON !

```
let t2 = {
   0: "un" ,
   1: "deux",
   2: "trois",
   length: 3
} ;
```

```
0: "un"
1: "deux"
2: "trois"
length: 3
<prototype>: Array []
```

Array.prototype

t1 et **t2** sont tous deux des objets mais n'ont pas le même **<prototype>**

```
0: "un"
1: "deux"
2: "trois"
length: 3
<prototype>: Object { … }
```

Object.prototype

Array Object

méthodes particulières pour traiter des collections ordonnées de valeurs

```
t1.indexOf("deux");  → 1
t1.sort();   → Array(3) ["deux", "trois", "un"]
```

la représentation interne des tableaux est optimisée pour travailler avec des données contiguës en mémoire

Array-likeObject

ne disposent pas de ces méthodes

```
t2.indexOf("deux");  →
```

```
Uncaught TypeError: t1.inedxOf is not a function
    <anonymous> debugger eval code:1
[En savoir plus]
```

Dans l'environnement JavaScript, il peut y a voir de tels objets avec une propriété **length** et des propriétés indicées, par exemple **arguments** qui permet d'accéder aux arguments d'une fonction, **HTMLCollection** retournée par **document.getElementsByTagName()**

Why do you need to know about Array-Like Objects daily.dev/blog – Tapas Adhikary

A propos de `length`

```
let tab1 = [ "un" , "deux", "trois" ];
console.log(tab1[3]);    ------> undefined
console.log(tab1.length);------> 3

tab1[3] = "quatre";
console.log(tab1[3]);     ------> "quatre"
console.log(tab1.length);------> 4

tab1[10] = "onze";
console.log(tab[10]);     ------> "dix"
console.log(tab1.length); ------> 11
```

```
Array(3) [ "un", "deux", "trois" ]
  0: "un"
  1: "deux"
  2: "trois"
  length: 3
▶ <prototype>: Array []
```

```
▼ Array(4) [ "un", "deux", "trois", "quatre" ]
    0: "un"
    1: "deux"
    2: "trois"
    3: "quatre"
    length: 4
  ▶ <prototype>: Array []
```

```
Array(11) [ "un", "deux", "trois", "quatre", <6 empty slots>, … ]
  0: "un"
  1: "deux"
  2: "trois"
  3: "quatre"
  10: "onze"
  length: 11
▶ <prototype>: Array []
```

ce n'est pas une bonne idée d'avoir un tableau avec des 'trous' car cela casse les optimisations spécifiques aux tableaux

- La propriété `length` est automatiquement mise à jour lorsque le tableau est modifié, pour être précis `length` n'est pas le nombre d'éléments du tableau mais le plus grand index numérique + 1

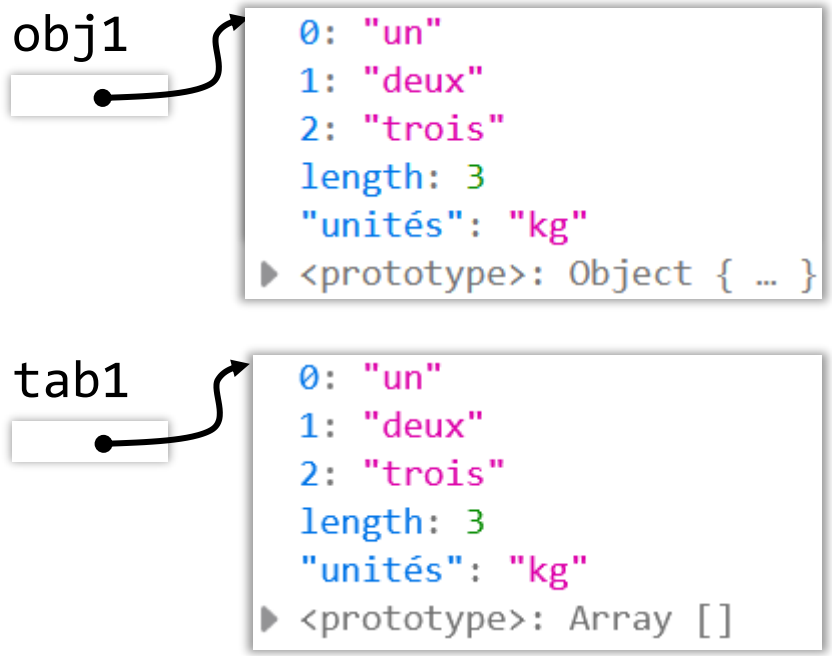
A propos de `length`

```
let obj1 = { 0: "un" , 1: "deux",  2: "trois", length: 3} ;
obj1.unités = "kg";
```

```
let tab1 = ["un" , "deux", "trois" ];
tab1.unités = "kg";
```

Les tableaux sont des objets, on peut donc leur associer des propriétés propres autres que `length`.

```
function display(x) {
    console.log(typeof x);
    for (let prop in x) {
        console.log(prop + " : " + x[prop]);
    }
}
```




```
display(obj1);   --->
object
0 : un
1 : deux
2 : trois
length: 3
unités : kg
```

```
display(tab1);   --->
object
0 : un
1 : deux
2 : trois
unités : kg
```

- La propriété `length` d'un tableau n'est pas énumérable

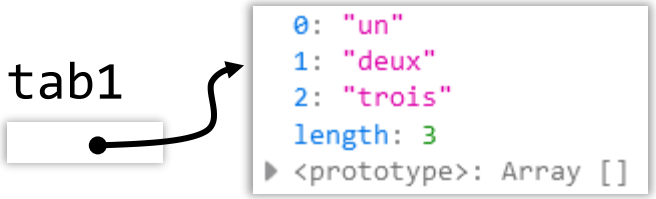
```
console.log(Object.getOwnPropertyDescriptor(tab1,'length'));
---> { value: 3, writable: true, enumerable: false, configurable: false }
```

```
console.log(Object.getOwnPropertyDescriptor(obj1,'length'));
---> { value: 3, writable: true, enumerable: true, configurable: false }
```

A propos de `length`

- La propriété `length` n'est pas énumérable mais elle est modifiable (*writable*)

```
let tab1 = ["un" , "deux", "trois"];
console.log(tab1[2]);---> trois
console.log(tab1);    ---> ["un", "deux", "trois"]
```

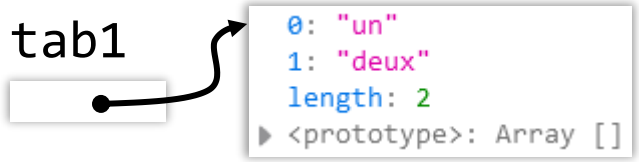


- On peut l'augmenter (même si ce n'est pas recommandé)

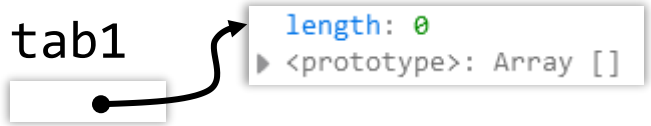
```
tab1.length = 6;
console.log(tab1[3]); ---> undefined
console.log(tab1);    ---> ["un", "deux", "trois", <3 empty slots>]
```

- On peut la diminuer : dans ce cas le tableau est tronqué de manière **irréversible**

```
tab1.length = 2;
console.log(tab1[2]); ---> undefined
console.log(tab1);    ---> ["un", "deux"]
```




```
tab1.length = 0;  // permet de 'vider' un tableau
console.log(tab1);  ---> []
```

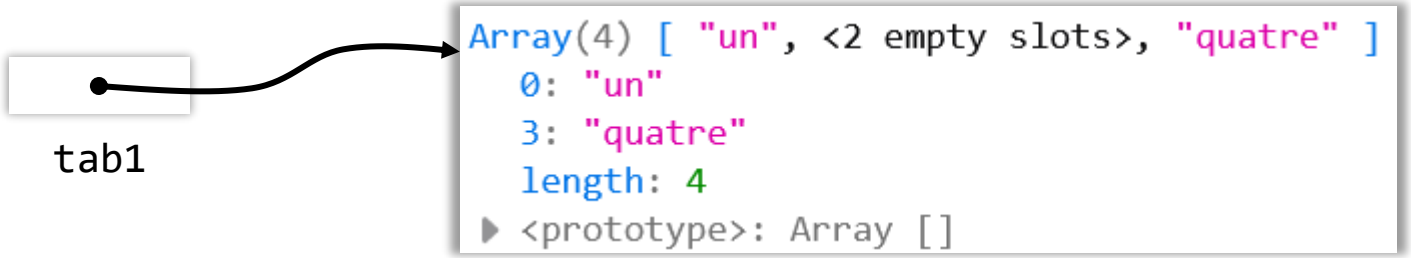


Création d'un tableau

- Expression littérale `[element0, element1, ..., elementN]`
  - Définit un tableau initialisé avec les éléments donnés
  - Les éléments peuvent être de n'importe quel type (valeurs primitives, objets, fonctions)

```
let tab1 = [ "un" , , , "quatre" ];
```

Tableau incomplet




```
let tab2 = [];
```

Tableau vide


```
let tab3 = null;
```

Référence nulle

Création d'un tableau

- Utilisation du constructeur `Array`

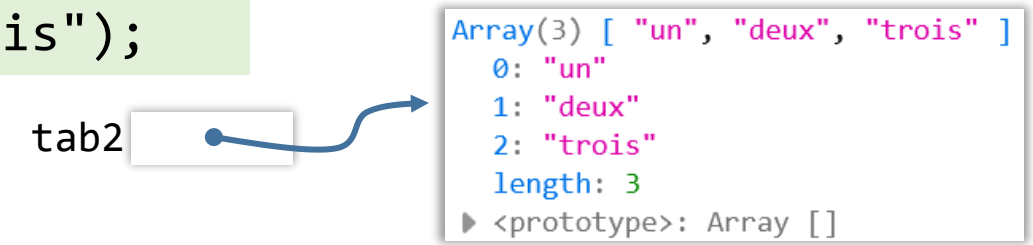
```
new Array()
new Array(element0, element1, ..., elementN)
new Array(arrayLength)
```

```
let tab1 = new Array();
```

crée un tableau vide ⇔ `let tab1 = [];`


```
let tab2 = new Array("un" , "deux",  "trois");
```

crée un tableau un tableau avec trois valeurs
⇔ `let tab2 = [ "un" , "deux",  "trois" ];`

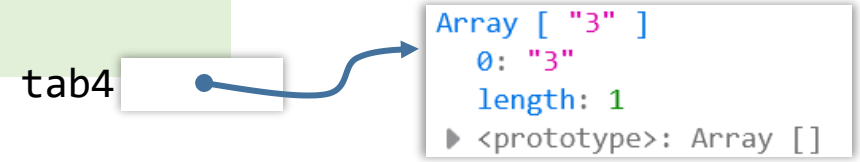



```
let tab3 = new Array(3);
```

crée un tableau de longueur 3 mais sans éléments
⇔ `let tab3 = []; tab1.length = 3;`

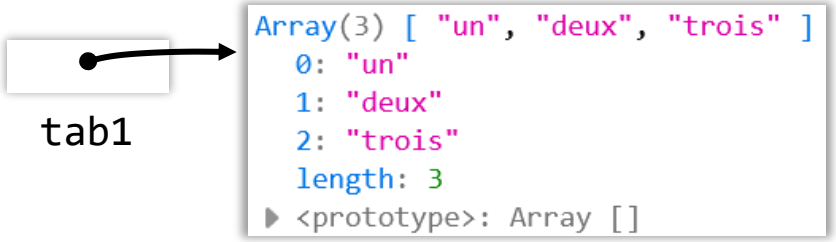

```
let tab4 = new Array("3");
```

crée un tableau avec une valeur
⇔ `let tab4 = [ "3" ];`



Parcours d'un tableau

```
let tab1 = [ "un" , "deux" , "trois" ];
```



- parcours "classique" : boucle `for` sur les indices

```
for (let i = 0; i < tab1.length; i++) {
  console.log(i + ": " + tab1[i]);
}
```

```
---> 0: un
     1: deux
     2: trois
```

- boucle sur les éléments : boucle `for … of`

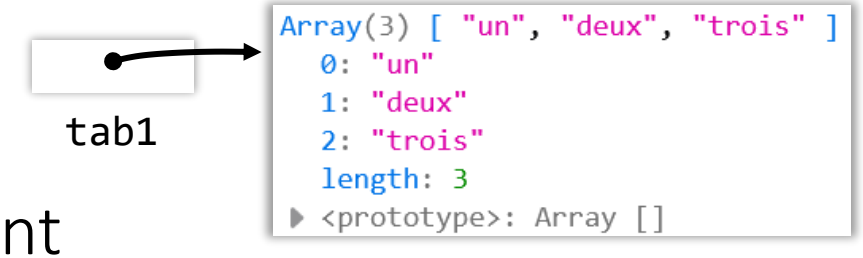
```
for (let nb of tab1) {
  console.log(nb);
}
```

```
---> un
     deux
     trois
```

Boucle `for … of` ne donne pas accès à l'indice des éléments, plus compact quand cet indice n'est pas nécessaire

Parcours d'un tableau

```
let tab1 = [ "un" , "deux" , "trois" ];
```



- les tableaux étant des objet on peut techniquement utiliser des boucles `for … in`

```
for (let prop in tab1) {
  console.log(prop + ": " + tab1[prop]);
}
```

```
---> 0: un
     1: deux
     2: trois
```

mais en général ce n'est pas une bonne idée

- moins efficace que les autres boucles optimisées pour les tableaux
- la boucle itère sur toutes les propriétés énumérables, pas uniquement sur les propriétés numériques.

Parcours d'un tableau

```
let tab1 = [ "un" ,  ,  "trois" ];
tab1[5] = "six";
tab1.name = "tab1";
```

tab1

```
Array(6) [ "un", <1 empty slot>, "trois", <2 empty slots>, "six" ]
  0: "un"
  2: "trois"
  5: "six"
  length: 6
  name: "tab1"
▶ <prototype>: Array []
```

```
for (let i = 0; i < tab1.length; i++) {
  console.log(i + ": " +tab1[i]);
}
```

------>

```
0: un
1: undefined
2: trois
3: undefined
4: undefined
5: six
```

```
for (let nb of tab1) {
  console.log(nb);
}
```

---------->

```
un
undefined
trois
undefined
undefined
six
```

```
for (let prop in tab1) {
  console.log(prop + ": " + tab1[prop]);
}
```

------>

```
0: un
2: trois
5: six
name: tab1
```

Toutes les propriétés auxquelles a été affecté une valeur sont présentes

```
» console.log(tab1)
▶ Array(6) [ "un", <1 empty slot>, "trois", <2 empty slots>, "six" ]
```

Seules les propriétés numériques correspondant aux indices auxquels a été affecté une valeur sont présentes

A propos des tableaux associatifs

- tableaux avec des noms comme indexes (supporté dans de nombreux langages: PHP…)
  - permet d'associer clés/valeurs

```
let tabAssoc1 = [];
tabAssoc1["cle1"] = "valeur1";
tabAssoc1["clé2"] = "valeur2";
tabAssoc1["clé3"] = "valeur3";

console.log("tabAssoc1");
for (let cle in tabAssoc1) {
 console.log(cle + " -->" + tabAssoc1[cle]);
}

tabAssoc1["clé2"] = "valeur2bis";
tabAssoc1["clé4"] = "valeur4";

console.log("tabAssoc1 après modification");
for (let cle in tabAssoc1) {
   console.log(cle + " -->" + tabAssoc1[cle]);
}
```


```
---> cle1 -->valeur1
     clé2 -->valeur2
     clé3 -->valeur3
```

```
---> cle1 -->valeur1
     clé2 -->valeur2bis
     clé3 -->valeur3
     clé4 -->valeur4
```

```
console.log(tabAssoc1[0]);
```

```
---> undefined
```

```
console.log(tabAssoc1.length);
```

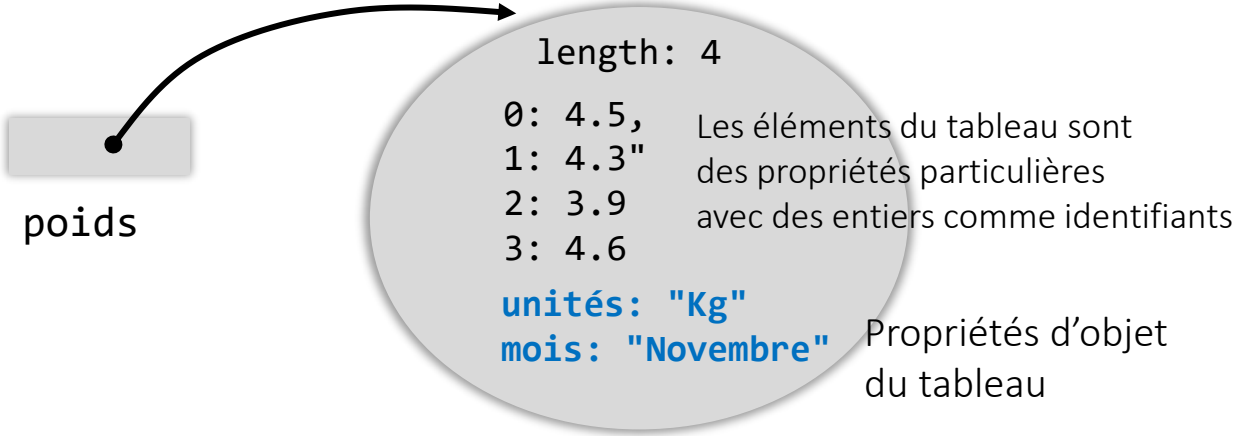
```
---> 0
```

Utiliser ou accéder à des index non entiers, en utilisant la notation avec crochets (ou avec point) ne va pas définir ou récupérer un élément sur le tableau lui-même, mais une variable associée à la collection de propriétés d'objet de ce tableau.

A propos des tableaux associatifs

- Les tableaux sont des objets, on peut donc leur associer des propriétés propres autres que `length`.

```
let poids = [4.5, 4.3, 3.9, 4.6];

poids.unités = "Kg";
poids.mois = "Novembre"  ;
```




```
for (let cle in poids) {
   console.log(cle + " --> " + poids[cle]);
}
```

```
---> 0 --> 4.5
     1 --> 4.3
     2 --> 3.9
     3 --> 4.6
     unités --> Kg
     mois --> Novembre;
```

A propos des tableaux associatifs

- Utiliser un objet et non pas un tableau pour créer un tableau associatif


```
let tabAssoc1 = [];
tabAssoc1["cle1"] = "valeur1";
tabAssoc1["clé2"] = "valeur2";
tabAssoc1["clé3"] = "valeur3";

console.log(tabAssoc1["cle1"]);

console.log(tabAssoc1[0]);

console.log(tabAssoc1.length);
```

valeur1

undefined

0

on utilisera un objet
et non pas un tableau

```
var tabAssoc1 = {};
tabAssoc1["cle1"] = "valeur1";
tabAssoc1["clé2"] = "valeur2";
tabAssoc1["clé3"] = "valeur3";

console.log(tabAssoc1["cle1"]);
```

valeur1

ES6: objet **Map**

http://andrewdupont.net/2006/05/18/javascript-associative-arrays-considered-harmful/

Méthodes de l'objet `Array`

```
let tab = [ "un" , "deux",  "trois" ];
// tab = new Array("un" , "deux",  "trois");
console.log(typeof tab);
```

```
---> object
```

OK les tableaux sont des objets, mais comment savoir si un objet est un tableau ?

La méthode `isArray` de `Array` permet de répondre à cette question

```
console.log(Array.isArray(tab));
```

```
---> true
```

- méthodes statiques de l'objet **`Array`**.

  `Array.isArray()`
  renvoie true si la variable est un tableau, false sinon.

  `Array.of()`
  crée une nouvelle instance d'**`Array`** à partir d'un nombre variable d'arguments (peu importe la quantité ou le type des arguments utilisés).

  `Array.from()`
  permet de créer une nouvelle instance d'**`Array`** à partir d'un objet semblable à un tableau (*array-like object*) ou d'un *itérable*.

Méthodes de `Array.prototype`

- Méthodes applicables à toutes tableaux

- Mutateurs
  - Méthodes modifiant le tableau
- Accesseurs
  - Méthodes ne modifiant pas l'état du tableau et en retournant une représentation.
- Méthodes d'itération
  - Méthodes utilisant des fonctions comme argument afin de traiter, d'une façon ou d'une autre, chaque élément du tableau.

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array

Méthodes de `Array.prototype`

- Mutateurs
  - Méthodes modifiant le tableau

`Array.prototype.copyWithin()`
copie une série d'éléments de tableau dans le tableau.

`Array.prototype.fill()`
remplit tous les éléments d'un tableau avec une même valeur, éventuellement entre un indice de début et un indice de fin.

`Array.prototype.pop()`
supprime le dernier élément d'un tableau et retourne cet élément.

`Array.prototype.push()`
ajoute un ou plusieurs éléments à la fin d'un tableau et retourne la nouvelle longueur du tableau.

`Array.prototype.reverse()`
renverse l'ordre des éléments d'un tableau - le premier élément devient le dernier, et le dernier devient le premier.

`Array.prototype.shift()`
supprime le premier élément d'un tableau et retourne cet élément.

`Array.prototype.sort()`
trie en place les éléments d'un tableau et retourne le tableau.

`Array.prototype.splice()`
permet d'ajouter et/ou de retirer des éléments d'un tableau.

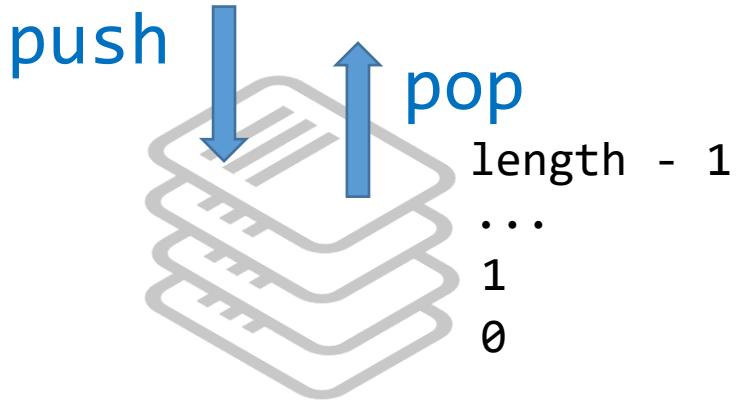
`Array.prototype.unshift()`
permet d'ajouter un ou plusieurs éléments au début d'un tableau et renvoie la nouvelle longueur du tableau.

Méthodes de Array.prototype

- `push` et `shift` permettent d'utiliser un tableau comme une file d'attente (*queue*)

FIFO
First In First Out

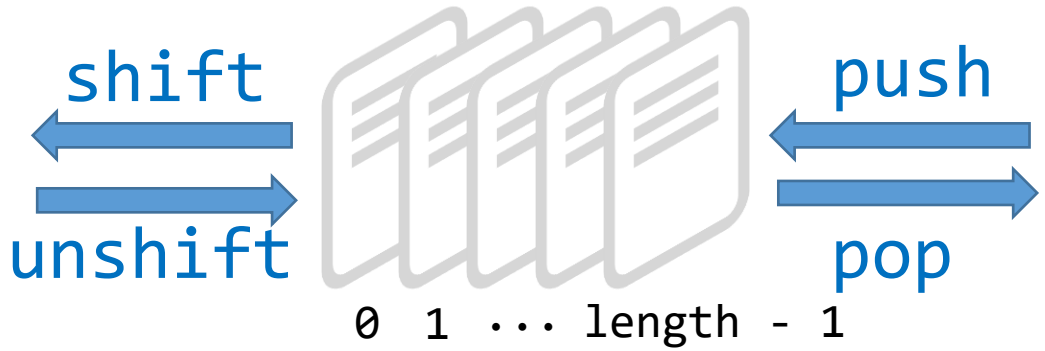
- `push` et `pop` permettent d'utiliser un tableau comme pile (*stack*)



LIFO
Last In First Out

Méthodes de Array.prototype

- `push` , `pop`, `shift` et `unshift` permettent d'utiliser un tableau comme une file d'attente à double entrée (*double-ended queue* abrégé en *deque*)



FIFO

First In First Out

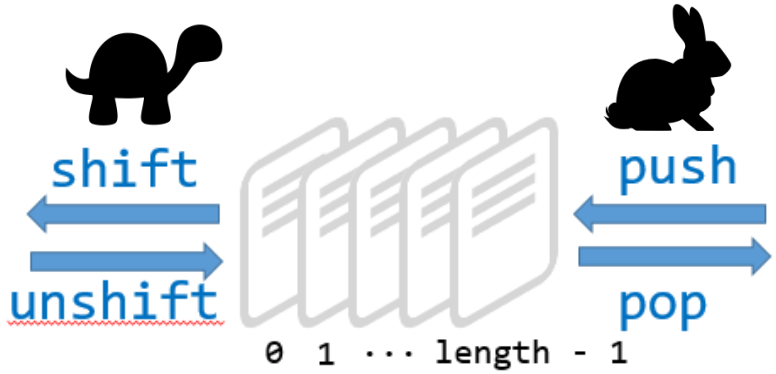
+

LIFO

Last In First Out

Méthodes de Array.prototype

- performances
  - `push` , `pop` sont rapides
  - `shift`, `unshift` sont plus lentes



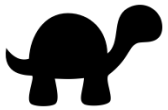
`tab.shift()`

1. Retirer l'élément d'indice 0
2. Décaler les éléments vers la gauche
   1 ---> 0, 2 ---> 1, etc…
3. Diminuer la taille (length) du tableau

`tab.unshift(value)`

1. Augmenter la taille (length) du tableau
2. Décaler les éléments vers la droite
   length -2 ---> length - 1, …, 1 ---> 2, 0 ---> 1
3. Insérer value à l'indice 0

Plus le tableau est grand, plus il y a d'opérations en mémoire pour décaler les éléments, plus la méthode prend du temps (complexité O(n))



`tab.pop()`

1. Retirer l'élément d'indice length - 1
2. Diminuer la taille (length) du tableau

`tab.push()`

1. Augmenter la taille (length) du tableau
2. Insérer value à l'indice length - 1

Pas d'opérations de décalage des éléments, la complexité de la méthode est indépendante de la taille du taleau

Méthodes de Array.prototype

- `splice` le couteau suisse pour les tableaux; elle peut tout faire : insérer, supprimer et remplacer des éléments !

`tab.splice(start [, deleteCount [, elem1, … , elemN]])`

modifie le tableau tab en partant de l'index **`start`** : supprime **`deleteCount`** éléments et insère à leur place les éléments **`elem1`**, …, **`elemN`**. La méthode retourne un tableau contenant les éléments supprimés

```
tab = ["un", "deux", "trois", "quatre", "cinq"];
removed = tab.splice(2);
console.log(tab);   ----------------------------------------------> [ 'un', 'deux' ]
console.log(removed); -------------------------------------------> [ 'trois', 'quatre', 'cinq' ]
```

quand **`start`** est négatif, on définit une position de départ à partir de la fin du tableau

```
tab = ["un", "deux", "trois", "quatre", "cinq"];
removed = tab.splice(-2);
console.log(tab); -----------------------------------------------> [ 'un', 'deux', 'trois' ]
console.log(removed); -------------------------------------------> [ 'quatre', 'cinq' ]
```

Méthodes de Array.prototype

`tab.splice(start [, deleteCount [, elem1, … , elemN]])`

modifie le tableau tab en partant de l'index **`start`** : supprime **`deleteCount`** éléments et insère à leur place les éléments **`elem1`**, …, **`elemN`**. La méthode retourne un tableau contenant les éléments supprimés

```
tab = ["un", "deux", "trois", "quatre", "cinq"];
removed = tab.splice(2, 2, "rouge", "vert", "bleu");
console.log(tab);  ----------------------------------------> ['un', 'deux', 'rouge', 'vert', 'bleu', 'cinq']
console.log(removed); -------------------------------------> [ 'trois', 'quatre' ]
```

```
tab = ["un", "deux", "trois", "quatre", "cinq"];
removed = tab.splice(-3, 1, "rouge", "vert");
console.log(tab);  ----------------------------------------> ['un', 'deux', 'rouge', 'vert', 'quatre', 'cinq']
console.log(removed); -------------------------------------> [ 'trois' ]
```

Méthodes de `Array.prototype`

- exemple `sort`
  - `arr.sort( [ fonctionComparaison ])`
    - `fonctionComparaison`,
      - paramètre optionnel, permet de spécifier une fonction définissant l'ordre de tri.
      - Si absent, le tableau est trié selon la valeur de point de code Unicode de chaque caractère, après la conversion en chaine de caractères de chaque élément.

```
let tab1 = ["Danielle", "Audrey", "Mathieu", "Elodie"];
console.log(tab1.sort());
```

```
---> ["Audrey", "Danielle", "Elodie", "Mathieu"]
```

```
let tab2 = [10, 117, 19, 17, 144, 11, 111, 110 ];
console.log(tab2.sort());
```

```
---> [10, 11, 110, 111, 117, 144, 17, 19]
```

Comparaison des éléments convertis en chaînes de caractères
ordre des caractères unicodes pour les chiffres `"0" < "1" < "2" < "3" < "4" < "5" < "6" < "7" < "8" < "9"`
➔
`"10" < "11" < "110" < "111" < "117" < "144" < "17" < "19"`

Méthodes de `Array.prototype`

- `arr.sort(`*`fonctionComparaison`*`)`
  - les éléments du tableau (qui ne valent pas `undefined`) sont triés selon la valeur de retour de la `fonctionComparaison`. Si `a` et `b` sont deux éléments à comparer, alors :
    - si `fonctionComparaison(a, b)` < 0 alors `a` classé avant `b`.
    - si `fonctionComparaison(a, b)` = 0 alors le classement de `a` par rapport à `b` demeure inchangé.
    - si `fonctionComparaison(a, b)` > 0 alors `a` classé après `b`.

```
let tab2 = [10, 117, 19, 17, 144, 11, 111, 110 ];
console.log(tab2.sort());
```

ordre des caractères unicodes pour les chiffres

```
---> [10, 11, 110, 111, 117, 144, 17, 19]
```

```
tab2.sort(function(a,b) {
    return a - b;
});
console.log(tab2);
```

```
---> [ 10, 11, 17, 19, 110, 111, 117, 144 ]
```

```
tab2.sort((a,b) => b - a);
console.log(tab2);
```

```
---> [ 144, 117, 111, 110, 19, 17, 11, 10 ]
```

Méthodes de `Array.prototype`

- `arr.sort(`*`fonctionComparaison`*`)`

```
let tab4 = ["Zorro", "Zoé", "Zut", "Zéphir", "Zarathoustra" ];
console.log(tab4.sort());
```

```
---> [ 'Zarathoustra', 'Zorro', 'Zoé', 'Zut', 'Zéphir' ]
```

Dans Unicode A < B … < Z < a < b < … < z < à < é …

Pour la prise en compte des caractères accentués utiliser la méthode `localCompare` des Strings

```
tab4.sort(function(a, b) {
   return a.localeCompare(b);
});

console.log(tab4);
```

```
---> [ 'Zarathoustra', 'Zéphir', 'Zoé', 'Zorro', 'Zut' ]
```

Méthodes de `Array.prototype`

- Accesseurs
  - Méthodes ne modifiant pas l'état du tableau et en retournant une représentation.

`Array.prototype.concat()`
renvoie un nouveau tableau constitué de ce tableau concaténé avec un ou plusieurs autre(s) tableau(x) et/ou valeur(s).

`Array.prototype.includes()`
détermine si le tableau contient ou non un certain élément. Elle renvoie true ou false selon le cas de figure.

`Array.prototype.indexOf()`
retourne le premier (plus petit) index d'un élément égal à la valeur passée en paramètre à l'intérieur du tableau, ou -1 si aucun n'a été trouvé.

`Array.prototype.join()`
concatène tous les éléments d'un tableau en une chaîne de caractère. Par défaut, ils sont séparés par des virgules, mais possibilité de spécifier un séparateur

`Array.prototype.lastIndexOf()`
retourne le dernier (plus grand) index d'un élément égal à la valeur passée en paramètre à l'intérieur du tableau, ou -1 si aucun n'a été trouvé.

`Array.prototype.slice()`
extrait une portion d'un tableau pour retourner un nouveau tableau constitué de ces éléments.

`Array.prototype.toString()`
renvoie une chaîne de caractères représentant le tableau et ses éléments.

`Array.prototype.toLocaleString()`
retourne une chaîne de caractères représentant le tableau et ses éléments en tenant compte de la localisation.

Méthodes de `Array.prototype`

- Itérateurs
  - Méthodes utilisant des fonctions comme argument afin de traiter, d'une façon ou d'une autre, chaque élément du tableau.
    - `Array.prototype.forEach(`*`callBack`*`)`
      - appelle la fonction ***`callBack`*** sur chacun des éléments du tableau.
      - Les paramètres de ***`callBack`*** sont
        - **`valeurCourante`** : la valeur de l'élément du tableau en cours de traitement.
        - **`index`** (optionnel) : l'indice de l'élément du tableau en cours de traitement.
        - **`array`** (optionnel) : le tableau sur lequel la méthode **`forEach`** est appliquée.

Boucle for « classique »

```
tab1 = [1, 2, 3, 4, 5, 6];

for (let i = 0; i < tab1/length; i++) {
    console.log("tab1[" + i + "] = " + tab1[i]);
}
```

```
---> tab1[0] = 1
     tab1[1] = 2
     tab1[2] = 3
     tab1[3] = 4
     tab1[4] = 5
     tab1[5] = 6
```

Boucle forEach

```
tab1 = [1, 2, 3, 4, 5, 6];

tab1.forEach(function(val, index) {
    console.log("tab1[" + index + "] = " + val);
});
```

Méthodes de `Array.prototype`

- Itérateurs

`Array.prototype.every()`
renvoie true si chaque élément du tableau satisfait la fonction de test passée en paramètre.

`Array.prototype.some()`
renvoie true si au moins un élément du le tableau satisfait la fonction de test passée en paramètre.

`Array.prototype.find()`
renvoie la valeur d'un élément trouvé dans le tableau et qui satisfait la fonction de test passée en paramètre, undefined sinon.

`Array.prototype.findIndex()`
renvoie l'index d'un élément trouvé dans le tableau qui satisfait la fonction de test passée en paramètre ou -1 si aucun ne la satisfait.

`Array.prototype.filter()`
crée un nouveau tableau contenant tous les éléments du tableau pour lesquels la fonction de filtrage passée en argument retourne true.

Méthodes de `Array.prototype`

- Itérateurs

`Array.prototype.map(`*`mappingFunction`*`)`
crée un nouveau tableau contenant les images de chaque élément du tableau de départ par la fonction passée en paramètre.

`Array.prototype.reduce(`*`reductionFunction`*`)`
`Array.prototype.reduce(`*`reductionFunction, initialValue`*`)`
applique une fonction sur un accumulateur et sur chaque valeur du tableau (de gauche à droite) de façon à obtenir une unique valeur à la fin.

`Array.prototype.reduceRight(`*`reductionFunction`*`)`
`Array.prototype.reduceRight(`*`reductionFunction, initialValue`*`)`
applique une fonction sur un accumulateur et sur chaque valeur du tableau (de droite à gauche) de façon à obtenir une unique valeur à la fin.

Paramètres de *`reductionFunction`*

- `accumulateur` : valeur précédemment retournée par le dernier appel du callback, ou valeurInitiale, si elle est fournie (voir ci-après) (c'est la valeur « accumulée » au fur et à mesure des appels
- `valeurCourante` : la valeur de l'élément du tableau en cours de traitement.
- `index` (optionnel) : indice de l'élément du tableau en cours de traitement.
- `array` (optionnel) : tableau sur lequel la méthode est appliquée.

Méthodes de `Array.prototype`

```
personnel = [
  {
    id: 5,
    name: "Luke Skywalker",
    pilotingScore: 98,
    shootingScore: 56,
    isForceUser: true,
  },
  {
    id: 82,
    name: "Sabine Wren",
    pilotingScore: 73,
    shootingScore: 99,
    isForceUser: false,
  },
  {
    id: 22,
    name: "Zeb Orellios",
    pilotingScore: 20,
    shootingScore: 59,
    isForceUser: false,
  },
  {
    id: 15,
    name: "Ezra Bridger",
    pilotingScore: 43,
    shootingScore: 67,
    isForceUser: true,
  },
  {
    id: 11,
    name: "Caleb Dume",
    pilotingScore: 71,
    shootingScore: 85,
    isForceUser: true,
  },
];
```

Calculer le score total (pilotage + tir) des utilisateur de la Force seulement

1. Filtrer les utilisateurs de la Force

```
let jediPersonnel = personnel.filter(function (person) {
  return person.isForceUser;
});

// Result: [{...}, {...}, {...}] (Luke, Ezra and Caleb)
```

2. Construire un tableau avec le score total de chaque Jedi

```
let jediScores = jediPersonnel.map(function (jedi) {
  return jedi.pilotingScore + jedi.shootingScore;
});

// Result: [154, 110, 156]
```

3. Combiner les valeurs pour calculer le total

```
let totalJediScore = jediScores.reduce(function (acc, score) {
  return acc + score;
}, 0);

// Result: 420
```

D'après Simplify your JavaScript – Use .map(), .reduce(), and .filter() Etienne Talbot
https://medium.com/poka-techblog/simplify-your-javascript-use-map-reduce-and-filter-bd02c593cc2d

Méthodes de `Array.prototype`

- Chaînage des méthodes de traitement

```
totalJediScore = personnel
  .filter(function (person) {
    return person.isForceUser;
  })
  .map(function (jedi) {
    return jedi.pilotingScore + jedi.shootingScore;
  })
  .reduce(function (acc, score) {
    return acc + score;
  }, 0);
```

- Utilisation de la syntaxe des « fonctions flèche » (*arrow functions*)

```
totalJediScore = personnel
  .filter((person) => person.isForceUser)
  .map((jedi) => jedi.pilotingScore + jedi.shootingScore)
  .reduce((acc, score) => acc + score, 0);
```

D'après Simplify your JavaScript – Use .map(), .reduce(), and .filter() Etienne Talbot
https://medium.com/poka-techblog/simplify-your-javascript-use-map-reduce-and-filter-bd02c593cc2d