**Les arrays**

Les arrays sont des structures de données. Elles permettent d'organiser des données de façon ordonnée et leur utilisation générale est de regrouper des valeurs qui représentent une même chose (par exemple les notes des élèves d'une classe).

Pour créer un array, on utilisera simplement des crochets qui contiendront des valeurs séparées par des virgules.

**monArray = [2, 4, 6, 8];**

Dans cet array chacune des valeurs est attribuée à un index, c'est sa place dans l'array.

Petite différence par rapport à tous les jours, on commencera à compter à partir de 0.

Donc la valeur 2 est à l'index (ou place) 0 et la valeur 8 est à l'index 3.

Avec l'index on peut soit récupérer une valeur se trouvant à un index soit y changer la valeur ou encore donner une valeur à un index encore non existant.

Pour récupérer une valeur à un index particulier en utilisant l'array précédemment attribué à la variable **monArray**:

**monArray[0]; // Renvoie 2**

**monArray[2]; // Renvoie 6**

On utilise le nom de l'array suivi de l'index entre deux crochets.

Pour changer une valeur d'un array :

**monArray[1] = 13;**

**monArray[3] = 47;**

**console.log(monArray); //** Affiche [2, 13, 6, 47]

On fera un réassignement comme habituellement mais en précisant l'index entre crochets juste après la variable qui contient l'array.

On peut également utiliser cette écriture avec un index encore non-existant pour l’ajouter :

**monArray[5] = 300;**

**console.log(monArray); //** Affiche [2, 13, 6, 47, <1 empty slot>, 300]

Notre 300 est bien apparu à l'index 5 mais quelque chose de spécial a eu lieu, on a dorénavant une valeur "<1 empty slot>" à l'index 4.

Cet index sera l'équivalent d'une valeur undefined généralement mais provoquera dans certains cas des comportements différents par exemple lors de l'utilisation de méthodes liées aux arrays. C'est pourquoi on préféra utiliser la méthode **push** reprise plus bas pour rajouter des éléments à la fin d'un array sans prendre le risque de se tromper et de laisser des "trous" dans l'array.

Il est possible de savoir la longueur d'un array grâce à son attribut **length**, pour ce faire on utilise l'array/le nom de l'array suivi d'un point et du mot **length** :

**[1, 9, 2, 8, 3, 6].length //** Renvoie 6, le nombre d'éléments de l'array

Modifier ce nombre (voir exemples plus bas) a un impact directement sur l'array. Si on augmente la valeur de **length,** la longueur de l'array augmentera elle aussi et les nouvelles valeurs seront des "<empty slots>". Dans le cas contraire, si **length** est réduite alors l'array perdra ses dernières valeurs pour arriver une nouvelle longueur équivalente à la valeur de **length**.

**myArray = [1, 9, 2, 8, 3, 6]; myArray.length = 8;**

**console.log(myArray); //** Renvoie [1, 9, 2, 8, 3, 6, <2 empty slots>] **myArray.length = 4; console.log(myArray); //** Renvoie [1, 9, 2, 8] 🡪on a perdu 2 valeurs

Les principales utilisations de **length** sont dans le but d'indiquer à une boucle combien de fois elle doit tourner afin de passer sur chaque valeur d'un array et afin de récupérer la ou les dernières valeurs de cet array.

Pour boucler sur les valeurs d'un array :

myArray = [10, 9, 8, 7, 6, 5]; console.log(myArray.length); // Renvoie 6, il y a donc 6 éléments dans l'array

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

}

Ce code va passer sur chaque valeur de l'array et l'afficher. L'attribut **length** servira à boucler 6 fois en tout car la condition est équivalente dans notre cas à "i < 6". Dans le console.log "myArray[i]" utilise l'index "i" qui contiendra tour à tour les nombres de 0 à 5, donc les index de chacune des valeurs.

Pour récupérer la valeur du dernier index d'un array de façon automatique :

**myArray = [10, 9, 8, 7, 6, 5];**

**console.log(myArray[myArray.length – 1]); //** Affiche depuis l'array l'index qui est la longueur de l'array moins un (Car on commence à compter à partir de 0)

À noter qu'il existe une syntaxe simplifiée pour une boucle for qui doit passer sur chaque élément d'un tableau ou d'un élément semblable à un tableau.

Il s'agit du **for(nomVariable of Array)** qui consiste à passer tout à tour sur chacun des index du tableau et à attribuer leur valeur à la variable juste avant le **of** (si la variable n'existe pas elle sera créée et on peut limiter la portée de la variable avec les mots-clefs habituels **var, let, const**). Le désavantage comparé à une boucle normale for est qu'on n'a pas d'accès à l'index de l'élément actuel :

**for (let x of [1, 3, 5, 7]) {**

**console.log(x);**

**}**

Affichera dans l'ordre les chiffre 1, 3, 5 et 7.

Il existe également de nombreuses méthodes (une méthode est une fonction qui fait partie d'un objet, et les arrays sont des objets en Javascript) qui s'utilisent avec les arrays, en voici les plus utiles :

[Array.isArray(valeur)](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/isArray) : Cette méthode est différente de celles vues auparavant car elle utilise l'objet global **Array** suivi d'un point et de la méthode à la place de l'array avec lequel on travaille. Elle permettra de vérifier si une valeur fournie est bien un array ou pas en retournant un booléen **true** ou **false**.

**Array.isArray([9, 18, 27]); //** Renvoie true

[Array.prototype.pop()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/pop) : Les méthodes qui sont écrites avec "prototype" dans les manuels font référence au fait que l'on doit utiliser notre objet (donc notre array dans ce cas) suivi d'un point, de la méthode voulue comme on l'a fait jusqu'à présent.

La méthode **pop** permet de supprimer le dernier élément de l'array et de renvoyer cette valeur.

**myArray = [9, 18, 27];**

**myArray.pop(); //** Renvoie 27

**console.log(myArray);** // Affiche [9, 18];

[Array.prototype.push()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/push) : La méthode **push** prend un ou plusieurs arguments et les ajoute à la fin de l'array utilisé. Renvoie la nouvelle longueur du tableau.

**myArray = [9, 18, 27];**

**myArray.push(36, 45);** // Renvoie 5

**console.log(myArray);** // Affiche [9, 18, 27, 36, 45]

[Array.prototype.shift()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/shift) :La méthode **shift** est la méthode inverse de **pop**, elle supprime la première valeur de l'array spécifié et retourne cette valeur.

**myArray = [9, 18, 27];**

**myArray.shift();** // Renvoie 9

**console.log(myArray);** // Affiche [18, 27]

[Array.prototype.unshift()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/unshift) :Et voici la méthode opposée à **push**. Les arguments fournis à **unshift** seront rajoutés au début de l'array et la nouvelle longueur de l'array sera renvoyée.

**myArray = [9, 18, 27];**

**myArray.unshift(36, 45); //** Renvoie 5 **console.log(myArray); //** Affiche [36, 45, 9, 18, 27]

[Array.prototype.reverse()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/reverse) : La méthode **reverse** inverse l'ordre des éléments du tableau et renvoie ce tableau modifié.

**myArray = [9, 18, 27];**

**myArray.reverse();** // Renvoie [27, 18, 9]

**console.log(myArray);** // Affiche [27, 18, 9]

[Array.prototype.splice()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/splice) : La méthode **splice** permet de supprimer ou de rajouter des éléments dans un tableau à partir d'un index précis.

Elle peut prendre de multiples arguments :

* D'abord l'index de début à partir duquel l'array sera modifié
* Ensuite le nombre d'éléments à supprimer à partir de cet index. Attention si aucune valeur n'est fournie **splice** supprimera tous les éléments à partir de l'index de départ !
* Finalement et optionnellement un nombre variables d'éléments à rajouter à cette même position.

La méthode retourne un array contenant tous les éléments supprimés.

**myArray = [9, 18, 27];**

**myArray.splice(1, 1, 13, 17, 21);** // Retourne [18]

**console.log(myArray);** // Affiche [9, 13, 17, 21, 27]

[Array.prototype.includes()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/includes) : Avec la méthode **includes** on peut savoir si une valeur est présente dans un array. Elle prend un argument obligatoire et un optionnel, d'abord on lui passe la valeur à rechercher et si besoin l'index à partir duquel commencer à la recherche. Le booléen **true** ou **false** est renvoyé.

**myArray = [9, 18, 27];**

**myArray.includes(18);** // Renvoie true

**myArray.includes(18, 2);** // Renvoie false car on cherche à partir de l'index 2

[Array.prototype.indexOf()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/indexOf) : Comparée à **includes** la méthode **indexOf** va

permettre de récupérer l'index d'une valeur si elle est présente dans l'array ou renverra la valeur -1 si non-présente. Elle prend les mêmes arguments c'est à dire d'abord la valeur à trouver et optionnellement un index à partir duquel chercher.

**myArray = [9, 18, 27]; myArray.indexOf(18); // Renvoie 1 myArray.indexOf(18, 2); // Renvoie -1 car on cherche à partir de l'index 2**

**Attention lors de l'utilisation avec les conditions ! Cette méthode peut retourner 0 si la valeur se trouve à cet index et 0 est une valeur falsy.**

[Array.prototype.join()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/join) : La méthode **join** nous retourne une string à partir de l'array en précisant quel sera le séparateur entre chaque élément de l'array. La valeur par défaut du séparateur est la virgule mais on peut fournir en argument le séparateur souhaité.

**myArray = [9, 18, 27]; myArray.join(); // Renvoie '9,18,27' myArray.join(' || '); // Renvoie '9 || 18 || 27'**

[Array.prototype.slice()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/slice) : Cette méthode **slice** permet copier une partie d'un array et renvoie cette copie. Elle prend deux arguments facultatifs, l'index de début et l'index de fin. Si non précisés le début et la fin de l'array seront utilisés. La valeur à l'index de début est inclue mais celle de l'index de fin ne l'est pas !

**myArray = [9, 18, 27]; myArray.slice(1); // Renvoie [18, 27] index de fin absent -> fin de l'array myArray.slice(1, 2); // Renvoie [18] car on s'arrête sans inclure l'index de fin console.log(myArray); // Affiche [9, 18, 27], l'array n'est pas modifié**

À partir des méthodes ci-dessous c'est un peu différent, car chacune d'entre-elles va passer sur chaque élément tour à tour (dans l'ordre des index numériques) pour accomplir un but final qui dépendra de chaque méthode.

Mais de plus elles nécessiteront généralement un argument qui sera une fonction à effectuer sur chacun des éléments de l'array.

Cette fonction à donner en argument recevra dans tous les cas 3 arguments, d'abord l'élément de l'array sur lequel cette fonction passe actuellement, ensuite l'index de cet élément, et au final l'array en lui-même. A noter que vous pouvez donner les noms de paramètres comme bon vous semble car vous écrirez la définition de la fonction et non son appel et qu’additionnellement il n'est pas nécessaire d'avoir 3 paramètres si vous en utiliserez moins que ça !

**Ex : (element, indexElement, array) => {instructions à accomplir} element => {instructions à accomplir}** // Utilise seulement le premier argument

[Array.prototype.forEach()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/forEach) : La méthode **forEach** prend obligatoirement un argument fonction qui sera appliqué à chaque élément dans l'array mais qui ne modifiera d'aucune façon cet array. On passe sur chaque élément tour à tour Equivalent d’un arrêt for.

Si les boucles et **for (... of ...)** n'étaient pas assez pour vous cette méthode vous sera utile. A noter tout de même que comparées aux autres **forEach** ne peut pas être interrompue avec un **break** ! Mais on a accès à l’index des éléments – ce qui n’est jamais le case avec un for of .

**myArray = [9, 18, 27]; myArray.forEach( e => {console.log(e \* 2)} ); // Affichera 18, 36, 54 console.log(myArray); // Affiche [9, 18, 27], l'array n'est pas modifié**

[Array.prototype.map()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/map) : Dans la même veine que **forEach** mais dans ce cas **map** renverra un nouvel array avec toutes les valeurs retournées par la fonction fournie à la méthode. L'array de base n'est pas changé.

**myArray = [9, 18, 27];**

**myNewArray = myArray.map( e => {return e \* 2} ); console.log(myNewArray); //** Affiche [18, 36, 54], c'est un nouvel array **console.log(myArray); //** Affiche [9, 18, 27], l'array de base n'est pas changé

exo : //[1, 2, 3, 4, 5] à partir de cet array, créez un nouvel array qui contiendra l'exposant deux de l'élément respectif

tableau = [1, 2, 3, 4, 5];

function callback (e) {

return e\*\*2;

}

tableau.map(callback)

Array(5) [ 1, 4, 9, 16, 25 ]

[Array.prototype.find()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/find) : La méthode **find** permet de trouver la première valeur dans un array qui valide une fonction de validation (fonction à fournir en argument et qui doit renvoyer **true** ou **false** ou valeur équivalente).

Elle s'arrête immédiatement de tourner dès qu'elle tombe sur une valeur étant validée, la fonction de validation n'étant donc pas effectuée sur les éléments restants, et renvoie cette valeur. Si aucune valeur ne passe le test alors la méthode renvoie **undefined**.

Elle s’arrête immédiatement quand elle trouve TRUE. Si elle ne trouve jamais de TRUE, elle donnera undifined

Une autre différence avec les autres méthodes est que la méthode **find** (et **findIndex**) font partie des rares méthodes à passer sur les emplacements vides ("empty slots") et que cela peut avoir un impact sur l'optimisation d'une solution. **MyArray = [9, 18, 27];**

**myArray.find( e => {return e % 2 === 0} );** {return le modulo de 2 strictement égal à 0}

Les opérateurs de comparaisons donnent toujours un TRUE ou FALSE donc pas besoin de convertir ma fonction.

**//** Renvoit la valeur 18 car c'est la première (et la seule dans ce cas) valeur qui est un nombre pair

**myArray.find( e => {return e % 4 === 0} );**

**//** Renvoit la valeur undefined car aucune valeur dans l'array n'est un nombre multiple de 4

**La fonction fournie en argument renvoie true ou false grâce à l'opérateur de comparaison**

mesEleves=["Jaouad", "Valérie", "Coralie", "Tom", "Robbins"]

Je veux le premier nom commençant par la lettre T

function callback(e){

return e[0] = = = "T";

}

mesEleves.find(callback)

"Tom"

[Array.prototype.findIndex()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/findIndex) : Clone presque exacte de **find**, la méthode **findIndex** renverra l'index de la valeur qui sera validée comme avant par une fonction fournie en argument. Dans le cas où aucune valeur n'est validée, **-1** est renvoyé. Voir méthode **find** pour toutes les informations. **MyArray = [9, 18, 27];**

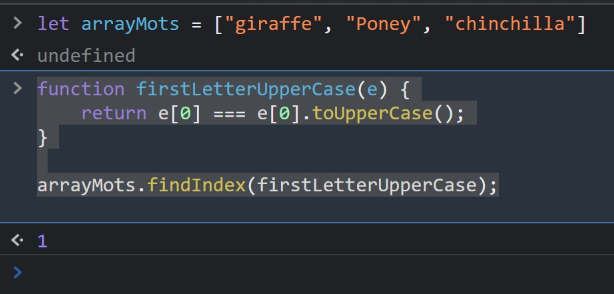
**myArray.findIndex( e => {return e % 2 === 0} );**

**//** Renvoit l'index 1 car c'est l'index de la première (et seule) valeur qui est un nombre pair

**myArray.findIndex( e => {return e % 4 === 0} );**

// Renvoit un non-index **-1** puisqu'aucune valeur dans l'array n'est un nombre multiple de 4

e 🡪 = élément



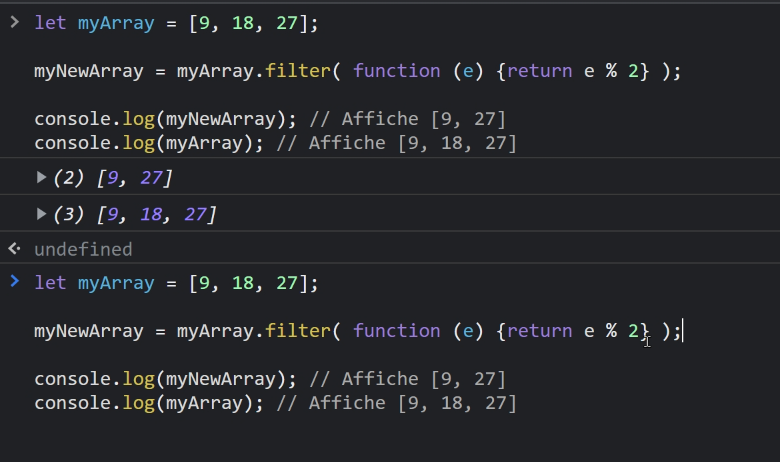
[Array.prototype.filter()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/filter) : Cette méthode nécessite également une

fonction en argument qui servira à valider des valeurs et **filtrer** renverra un nouvel array (donc l'array de départ ne subit aucune modification) contenant seulement les valeurs qui auront été validées.

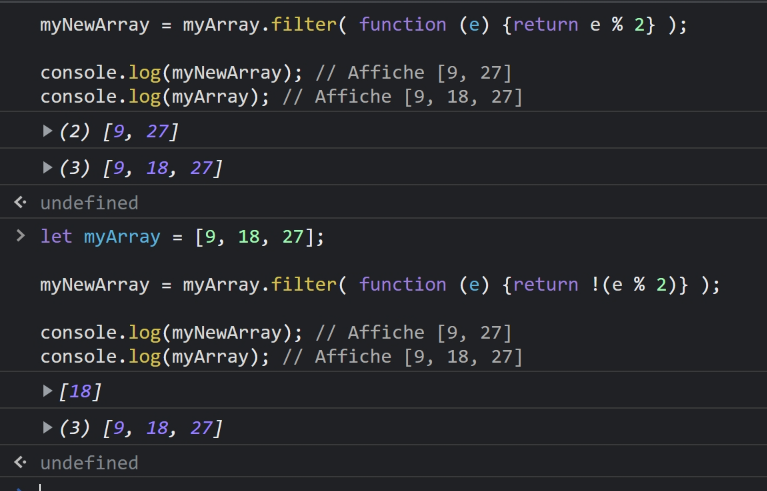
Sert à filtrer pour ne garder que les valeurs qui sont validées, les autres ne sont pas présentes. C’est un nouvel array 🡪 le tableau de départ n’a pas été détruit.

Une valeur est considérée valide si la fonction retourne une valeur **true** (=valeur qui est vraie) ou **truthy** (= valeur qui n’est pas fausse).

DONC Tous les éléments que l’array aura gardé ce que c’est TRUE ou TRUTHY



Pour faire l’inverse, je dois demander le contraire de TOUT l’élément



Par opposition à **find** et **findIndex**, **filter** passera sur toutes les emplacements de l'array (qui ne sont pas des "empty slots"). 🡪 Ne passe donc pas sur les empty slots

**myArray = [9, 18, 27];**

**myNewArray.filter( e => {return e % 2} );**

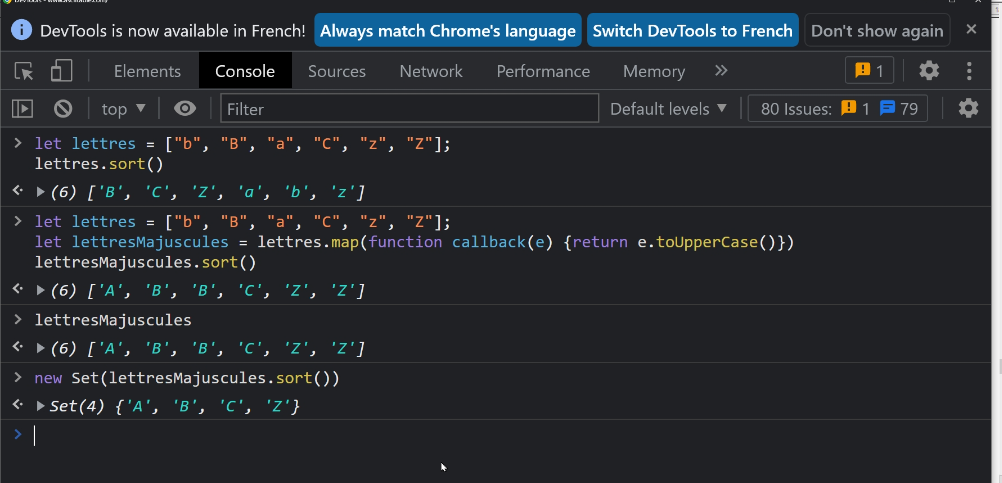
**console.log(myNewArray); //** Affiche [9, 27] console.log(myArray); // Affiche [9, 18, 27]

**L'instruction "e % 2" nous donne le modulo (reste d'une division entière) de l'élément par 2, les nombres impairs ont donc un reste de 1 ce qui est considéré comme une valeur truthy. (0 est considéré comme une valeur fausse et est donc filtré)**

Les deux dernières méthodes notées plus bas prennent également une fonction en premier argument mais ces fonctions sont assez spécifiques à la méthode en question et sont donc reprises à part.

[Array.prototype.sort()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/sort) : La méthode **sort** est une méthode pour ordonner un array. **Sera retourné l'array qui aura été modifié pour correspondre à l'ordre spécifié**. **Méthode destructive !** donc faire attention avant d’utiliser, nécessite parfois de faire une copie de sauvegarde.

En contraste à la plupart des méthodes qui passent sur les éléments d'un array, un argument fonction est *optionnel* et si aucune fonction n'est fournie elle transformera chaque élément en chaîne de caractères pour les trier de manière similaire à un dictionnaire (Les strings sont triés par rapport à leur séquence en UTF-16).



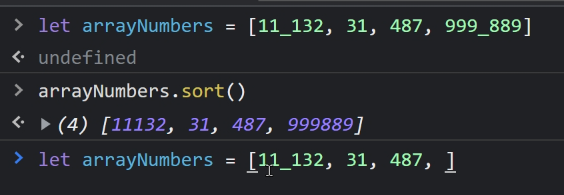
Map va créer un nouvel array sans modifier le précédent.

Pour faire encore mon *« .sort »* en plus rapide, je peux faire



Si je veux mettre en ordre croissant, je peux également le faire avec *« .sort ».*

Attention : il va se baser sur le premier chiffre.

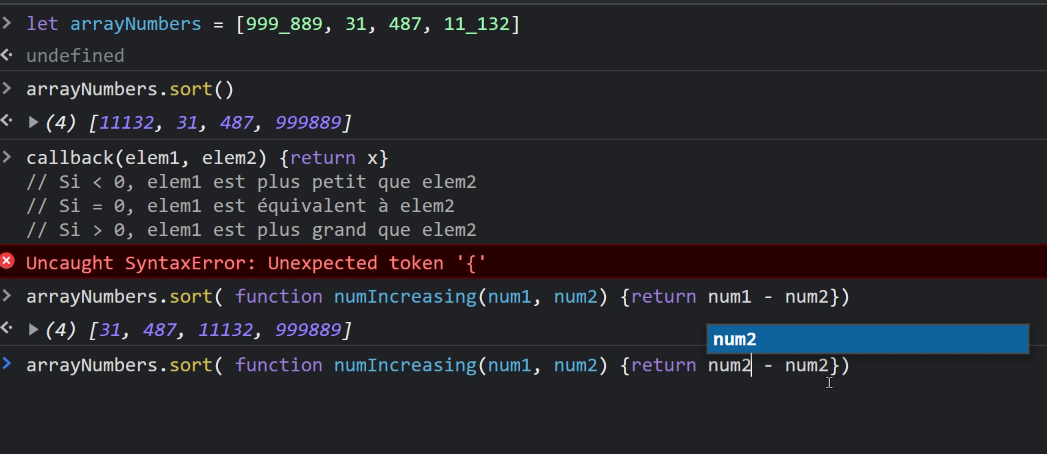


Du coup, voir la suite du cours ^^

Dans le cas où l'on veut un ordre de tri personnalisé on fera alors appel à la fonction en argument qui recevra elle-même 2 arguments, le premier élément à comparer et le deuxième élément à comparer.

Ensuite pour communiquer à la méthode **sort** quel élément est plus grand ou plus petit on fera attention à ce que la bonne valeur soit retournée :

* Si on retourne une valeur plus petite que 0 le premier élément est considéré comme étant plus petit. (Et arrive donc le premier parmi les deux)
* Si on retourne une valeur plus grande que 0 le premier élément est considéré comme étant plus grand. (Arrive en dernier parmi les deux)
* Si la valeur retournée est 0 alors aucun changement dans l'ordre n'a lieu.

****

**myArray = ['pomme', 'banane', 'poire', 'Pomme', 'Banane', 'Poire', 20,2, 9]; myArray.sort();**

**console.log(myArray); // Affiche [2, 20, 9, "Banane", "Poire", "Pomme", "banane", "poire", "pomme"]**

**L'array de départ est modifié et ordonne selon le code UTF-16. Ce qui en pratique correspond à la manière d'un dictionnaire – caractère par caractère - aux chiffres classés suivi par les mots en ordre alphabétique – majuscules en premières -**

**myArray = [28, 9, 94, 65, 34, 1, 6] myArray.sort( (a, b) => a – b); console.log(myArray); // Affiche [1, 6, 9, 28, 34, 65, 94]**

**myArray = [28, 9, 94, 65, 34, 1, 6] myArray.sort((a, b) => b - a); console.log(myArray); // Affiche [94, 65, 34, 28, 9, 6, 1]**

**Les deux derniers exemples permettent de trier des nombres par ordre croissant ou décroissant en jouant sur le fait que sort classe l'élément a en premier si on retourne un nombre inférieur à 0 et que tout calcul a – b renverra un chiffre négative si a est plus petit et donc un chiffre positif si a est plus grand**

[Array.prototype.reduce()](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/reduce) : Dans le cas où on a besoin de réduire un ensemble de valeurs à une seule valeur (valeur qui pourrait également être un array ou un objet) on utilisera la méthode **reduce** à la laquelle on fournira une fonction en argument.

Cette fonction recevra automatiquement à chaque passage sur un élément les 4 arguments suivants :

* Un **accumulateur**, variable servant à tenir en mémoire la valeur réduite le temps de faire le tour des éléments
* La valeur de l'élément actuel
* L'index de l'élément actuel
* L'array sur lequel on travaille

Les seules valeurs dont nous auront besoin au minimum sont l'accumulateur et la valeur de l'élément actuel.

La valeur renvoyée à chaque étape par la fonction argument sera la nouvelle valeur de l'accumulateur pour la prochaine étape. Cet accumulateur sera la valeur de retour de **reduce** une fois qu'il n'y aura plus d'éléments restants.

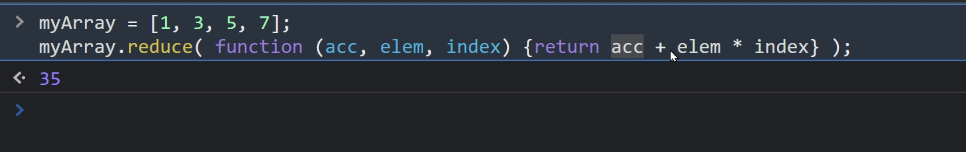
Voici un exemple pour mieux cerner **reduce**.

**myArray = [1, 3, 5, 7];**

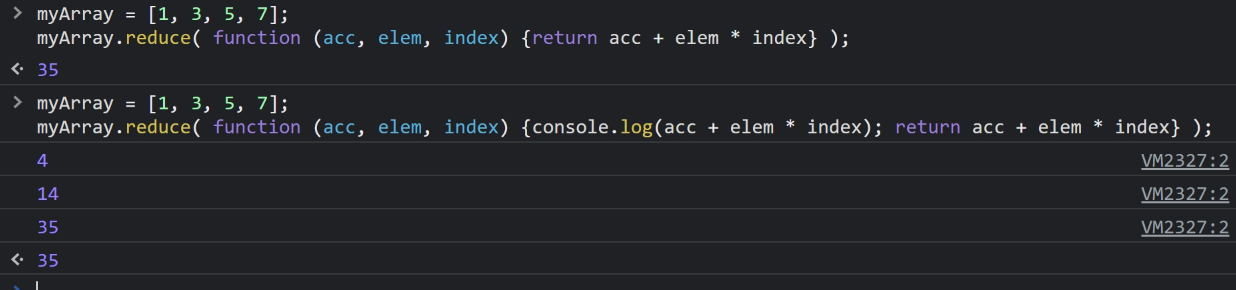
**myArray.reduce ((acc, elem) => {return acc + elem});**

**Sera renvoyé la valeur 16 car en passant tour à tour sur les éléments.**

**Cela fait 1 + 3 + 5 + 7 => 16**



Cela donne en détaillé :



En fait il y a un autre point important à savoir avec la méthode **reduce**, c'est que l'accumulateur prend de base la valeur du premier élément et la fonction de réduction n'est appelée qu'à partir du deuxième élément.

Mais en réalité la méthode **reduce** peut prendre un second argument optionnel après celui de la fonction de réduction et il s'agit d'une valeur initiale pour l'accumulateur. Si une valeur de départ est fournie alors la fonction de réduction commencera au premier élément de l'array.

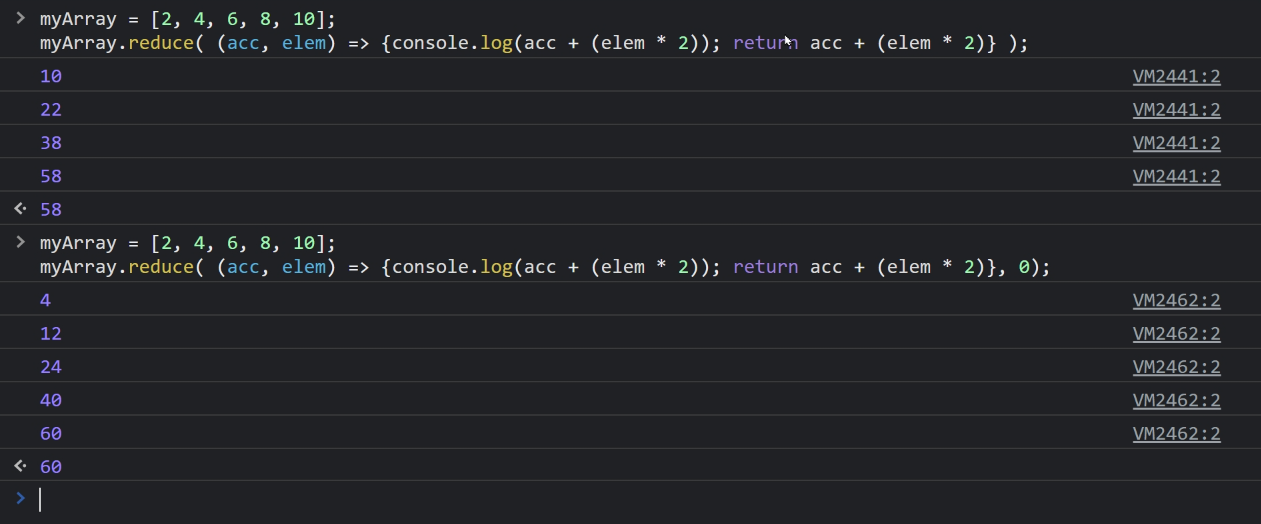
Voilà deux exemples pour montrer la différence et en quoi cela est important.

**myArray = [2, 4, 6, 8, 10]; myArray.reduce((acc, elem) => {return acc + (elem \* 2)});**

**Retourne 58 car l'accumulateur au départ vaudra 2 et la fonction ne sera appelée qu'à partir du second élément donc 2 + 8 + 12 + 16 + 20 => 58**

**myArray = [2, 4, 6, 8, 10]; myArray.reduce((acc, elem) => {return acc + (elem \* 2)}, 0);**

**Retourne 60 car ici on donne une valeur de départ à l'accumulateur qui vaudrait 0 initialement mais qui permettra à la fonction de réduction de s'effectuer sur le premier élément de l'array ce qui donne 0 + 4 + 8 + 12 + 16 + 20 = 60**



Et pour clore cette section sur la méthode **reduce**, un dernier exemple qui montre une utilisation qui retourne un array.

**myArray = [3, 14, 8, 7, 24, 33, 12];**

**myArray.reduce((acc, elem) => {**

**if (elem % 2) {**

**acc[1] += elem;**

**} else {**

**acc[0] += elem;**

**}**

**return acc;**

**}, [0, 0]); // Renvoie [58, 43]**

**Ici on cherche à renvoyer un array qui contiendra 2 éléments, la somme des chiffres pairs et la somme des chiffres impairs de l'array, à chaque étape on incrémente le bon élément dans l'accumulateur et on renvoie cet accumulateur**