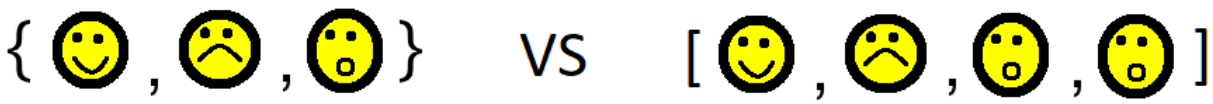
ES6 — Set vs Array — What and when?

Maya Shavin
Jan 24, 2018 · 6 min read



Set VS Array

. . .

What is Set and what is Array?

Everyone who works with JS until now is familiar with **Array** (*don't tell me you don't*). But what exactly is **Array**?

Well, in general, **Array** is type of structure representing **block of data** (numbers, objects, etc…) **allocated in consecutive memory.**

Example: [1,2,3,2]

How about Set?

Set, more familiar as a Math concept, is an **abstract data type** which contains **only distinct** elements/objects **without** *the need of being allocated orderly by index.*

Example: {1,2,3}

Yup, by definition, Array and Set are technically *different concepts*.

One of the biggest differences here, you may notice, is that **elements in Array can be duplicate** (unless you tell it not to be), and **in Set, they just can't** (regardless what you decide).

In addition, **Array** is considered as "**indexed collection**" type of data structure, while **Set** is considered as "**keyed collection**".

A quick reminder for those who don't remember,

> ***Indexed collections*** *are collections of data which are ordered by an index value*
>
> ***Keyed collections*** *are collections which use keys; these contain elements which are iterable in the order of insertion.*

Easy right? Now one may wonder, if they are different, why we bother compare between them?

In programming world, taken same data set (no duplicates), we can either use Array or Set to be our chosen structure to store this data set. However, depending on the use case, ***choosing the correct structure contributes to deliver optimal solution*** — and we want to achieve that. And in order to understand which to choose, we need understand firstly who they are, how they are built and what they are capable of. Since we have done with "who they are", let's move on to "how to build one" in JS.

Constructing

Array

Array is very straight-forward. To declare new array in JS, you can either use normal literal directly:

```
var arr = []; //Empty array
var arr = [1,2,3]; //Array which contains 1,2,3
```

Or use built-in constructor

```
var arr = new Array(); //empty array
var arr = new Array(1,2,3);//Array which contains 1,2,3
```

Or even cooler:

```
var arr = Array.from("123"); //["1","2","3"]
```

Side notes:

Just one piece of advice — **don't use new Array()** unless you really really really need to, since:

- It performs much *slower* than the normal [] literal. (this will be explained in a different article — maybe ;)).
- [] *saves more typing time* (try it :))
- You may *end up making some classical mistakes* such as:

```
var arr1 = new Array(10); //arr1[0] = undefined but arr1.length = 10
var arr2 = [10]; // arr2[0] = 10 and arr2.length = 1;

var arr3 = new Array(1,2,3); //[1,2,3]
var arr4 = [1,2,3];//[1,2,3]
```

So #1 thumb rule — keep it simple!

Set

Set has built-in constructor. Yup, there is no short-cut as in array.

> *Set([iterable])*

In order to create new set, we have to use ***new*** syntax, for example:

```
var emptySet = new Set();
var exampleSet = new Set([1,2,3]);
```

But definitely not:

```
new Set(1);
```

Set receives **iterable** object as its input parameter, and will create set object respectively. Hence, we can construct a set from an array — but *it will only include distinct elements from that array*, aka no duplicate.

And of course, we can also convert a set back to array using ***Array.from()*** method.

```
var set = new Set([1,2,3]); // {1,2,3}
var arr = Array.from(set);//[1,2,3]
```

OK, now that we know how to create them, what about their capabilities? Let's do a small comparison between the most basic methods Array/Set provides, which are:

Locating an element / Accessing an element

- First of all, **Set does not support random access to an element by index** like in Array, which means:

```
console.log(set[0]); //undefined
console.log(arr[0]); //1
```

- More important, because the Array data is stored in consecutive memory, the CPU will be able to access the data much faster due to pre-fetching. Hence in general ***accessing an elements in Array (one after the other such as in a for loop) is quicker and more efficient if you*** compared to other type of abstract data types.
- Checking if an element is in Set has simpler syntax than Array by using ***Set.prototype.has(value)*** VS ***Array.prototype.indexOf(value)***

```
console.log(set.has(0)); // boolean - false
console.log(arr.indexOf(0)); // -1

console.log(set.has(1)); //true
console.log(arr.indexOf(1)); //0
```

Which means in Array, we need to ask an extra check if we want to make a condition where the element is in Array:

```
var isExist = arr.indexOf(1) !== -1;
```

Note: ES6 does provide **Array.prototype.includes()** which behaves similarly to **has**(), however, it is not supported widely — aka not in IE yet (surprise :)!).

Add/Insert new element

- Adding new element to Array can be done quickly in **O(1)** by using **Array.prototype.push()** — element will be added to the end of the array.

```
arr.push(4); //[1,2,3,4]
```

- Or it can also be done in ***O(n)*** by using **Array.prototype.unshift()** — add element to the beginning of array — with *n* is the length of current array.

```
arr.unshift(3); //[3,1,2,3]
arr.unshift(5, 6); //[5,6,3,1,2,3]
```

- In **Set**, there is only one way to add new element — **Set.prototype.add()**. Because Set has to maintain the "***distinct***" property between its set members, in each time calling **add()**, Set needs to check through all members to make sure no duplicate before continuing. Generally **add()** will take **O(n)** running time. However, thanks to ***hash table*** **implementation approach**, `add()` in Set will likely take only **O(1).**

```
set.add(3); //{1,2,3}
set.add(4); //{1,2,3,4}
```

Hence here Set performs almost the same with Array in adding element. How about removing?

Remove element

- One of the nice things that make Array so popular is because it provides a lot of different methods to remove an element, for example:

Pop() — removes and returns the **last** element. This takes **O(1)**.

```
arr.pop();//return 4, [5,6,1,2,3]
```

Shift() — removes and return **first** element. This takes **O(n)**.

```
arr.shift(); //return 5; [6,1,2,3]
```

Splice(index, deleteCount) — remove a **number deleteCount of element (s) starting from index.** This can take up to O(n).

```
arr.splice(0,1); //[1,2,3]
```

- Meanwhile, in Set, we will use

Delete(element) — remove a **specific given element** from Set.

```
set.delete(4); //{1,2,3}
```

Clear() — remove **all elements** from Set.

```
set.clear(); //{}
```

- While Array **doesn't support native-built method** to remove a specific given element (except if we know its index), we will *need the help of an extra external function* to look for that element's index and perform splice(), Set has **delete() —** simple and easy to use.

In addition, *Array does provide us a lot more native functionalities* (reduce(), reverse(), sort(), etc…), compared to Set which currently only have the most basic functionalities mentioned above. So, some of you may think, why would we ever prefer Set over Array at all?

So, when is Set better? And when is Array better?

- Firstly, Set is different than Array. It is not meant to replace Array entirely, but to *provide additional support type to complete what Array is missing*.
- Since Set only contains distinct elements, it makes life much easier if we know in advance we want to *avoid saving duplicate data to our structure*.
- Basic operations of Set like **union(), intersect(), difference()**, etc… are easily implemented effectively based on the native built-in operations provided. Due to the delete() method, it makes intersect/union between 2 Sets much more comfortable than doing the same to 2 Arrays.
- Array is meant for scenarios when we want to *keep elements ordered for quick access*, or do *heavy modification* (removing and adding elements) or *any action required*

direct index access to elements (for example, try doing Binary Search on Set instead of Array — how do you access the middle located element?)

. . .

Conclusion

In general, to my opinion, **Set** doesn't really have a huge clear advantage over **Array**, except in specific scenarios such as when we want to maintain "distinct" data with minimum effort, or to work with a lot of distinct data sets together using the most basic set operations, without the need of direct accessing element.

Otherwise, **Array** should always the choice. Reason? *Less CPU work to fetch the element when it is needed*.

Do you agree? Feel free to discuss ;).

*P.S — Corrected the running time for **Set.add()** since in general **Set** is implemented optimally using Hash table approach. Thanks @André Patrício for pointing this out* 😃

. . .

Coming up next —ES6 Map VS Object…. See you soon! :)

More on ES6:

- Spread (…) syntax and its magic
- ES6 Cool stuffs — var, let and const in depth
- ES Cool stuffs — Destructuring statement in depth
- Template literals in depth
- Let's divide our phones into Classes

If you like this post and want to read more, feel free to check out my articles.

If you'd like to catch up with me sometimes, follow me on Twitter | Facebook or simply visit my portfolio website.

Maya Shavin (@MayaShavin) | Twitter
The latest Tweets from Maya Shavin (@MayaShavin). always be happy and motivated...

twitter.com

JavaScript ES6 Front End Development Programming Data Structures