ES6 Array Extensions in Depth

Hello traveler! This is ES6 – "Oh cool, I like `Array`" – in Depth series. If you've never been around here before, start with A Brief History of ES6 Tooling. Then, make your way through destructuring, template literals, arrow functions, the spread operator and rest parameters, improvements coming to object literals, the new classes sugar on top of prototypes, `let`, `const`, and the "Temporal Dead Zone", iterators, generators, Symbols, Maps, WeakMaps, Sets, and WeakSets, proxies, proxy traps, more proxy traps, reflection, `Number`, and `Math`. Today we'll learn about new `Array` extensions.

Nicolás Bevacqua *Published a year ago | 26 minute read | 4*

Like I did in previous articles on the series, I would love to point out that you should probably set up Babel and follow along the examples with either a REPL or the `babel-node` CLI and a file. That'll make it so much easier for you to **internalize the concepts** discussed in the series. If you aren't the *"install things on my computer"* kind of human, you might prefer to hop on CodePen and then click on the gear icon for JavaScript – *they have a Babel preprocessor which makes trying out ES6 a breeze.* Another alternative that's also quite useful is to use Babel's online REPL – *it'll show you compiled ES5 code to the right of your ES6 code for quick comparison.*

Before getting into it, let me *shamelessly ask for your support* if you're enjoying my ES6 in Depth series. Your contributions will go towards helping me keep up with the schedule, server bills, keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

keeping me fed, and maintaining **Pony Foo** as a veritable source of JavaScript goodies.

Thanks for reading that, and let's go into `Array` extensions. For a bit of context you may want to look at the articles on iterators, generators, arrow functions and collections.

Upcoming `Array` Methods

There's plenty to choose from. Over the years, libraries like Underscore and Lodash spoke loudly about features we were missing in the language, and now we have a ton more tools in the functional array arsenal at our disposal.

First off, there's a couple of static methods being added.

- `Array.from` – create `Array` instances from arraylike objects like `arguments` or iterables
- `Array.of`

Then there's a few methods that help you manipulate, fill, and filter arrays.

- `Array.prototype.copyWithin`
- `Array.prototype.fill`
- `Array.prototype.find`
- `Array.prototype.findIndex`

There's also the methods related to the iterator protocol.

- `Array.prototype.keys`
- `Array.prototype.values`
- `Array.prototype.entries`
- `Array.prototype[Symbol.iterator]`

There’s a few more methods coming in ES2016 *(ES7)* as well, but we won’t be covering those today.

- `Array.prototype.includes`
- `Array.observe`
- `Array.unobserve`

Let’s get to work!

`Array.from`

This method has been long overdue. Remember the quintessential example of converting an arraylike into an actual array?

```
function cast ()
  return Array.prototype.slice.call(arguments)
}
cast('a', 'b')
// <- ['a', 'b']
```

Or, a shorter form perhaps?

```
function cast ()
  return [].slice.call(arguments)
}
```

To be fair, we've already explored even more terse ways of doing this at some point during the ES6 in depth series. For instance you could use the spread operator. As you probably remember, the spread operator leverages the iterator protocol to produce a sequence of values in arbitrary objects. The downside is that the objects we want to cast with spread **must implement** `@@iterator` through `Symbol.iterator`. Luckily for us, `arguments` does implement the iterator protocol in ES6.

```
function cast ()
  return [...arguments]
}
```

Another thing you could be casting through the spread operator is DOM element collections like those returned from `document.querySelectorAll`. Once again, this is made possible thanks to ES6 adding conformance to the iterator protocol to `NodeList`.

```
[...document.querySelectorAll('div')]
// <- [<div>, <div>, <div>, ...]
```

What happens when we try to cast a jQuery collection through the spread operator? Actually, you'll **get an exception** because they haven't implemented `Symbol.iterator` quite yet. You can try this one on jquery.com in Firefox.

```
[...$('div')]
TypeError: $(...)[Symbol.iterator] is not a function
```

The new `Array.from` method is different, though. It doesn't *only* rely on iterator protocol to figure out how to pull values from an object. It also has support for arraylikes out the box.

```
Array.from($('div'))
// <- [<div>, <div>, <div>, ...]
```

The one thing you cannot do with either `Array.from` nor the spread operator is to pick a start index. Suppose you wanted to pull every `<div>` after the first one. With `.slice.call`, you could do it like so:

```
[].slice.call(document.querySelectorAll('div'), 1)
```

Of course, there's nothing stopping you from using `.slice` *after* casting. This is probably way easier to read, and looks more like functional programming, so there's that.

```
Array.from(document.querySelectorAll('div')).slice(1)
```

`Array.from` actually has three arguments, *but only the* `input` *is required*. To wit:

- `input` – the arraylike or iterable object you want to cast
- `map` – a mapping function that's executed on every item of `input`

- `context` – the `this` binding to use when calling `map`

With `Array.from` we cannot slice, but we can dice!

```
function typesOf () {
  return Array.from(arguments, value => typeof value)
}
typesOf(null, [], NaN)
// <- ['object', 'object', 'number']
```

Do note that you could also just combine rest parameters and `.map` if you were just dealing with `arguments`. In this case in particular, we may be better off just doing something like the snippet of code found below.

```
function typesOf (...all) {
  return all.map(value => typeof value)
}
typesOf(null, [], NaN)
// <- ['object', 'object', 'number']
```

In some cases, like the case of jQuery we saw earlier, it makes sense to use `Array.from`.

```
Array.from($('div'))
// <- [<div>, <div>, <div>, ...]
Array.from($('div'), el => el.id)
// <- ['', 'container', 'logo-events', 'broadcast', ...]
```

I guess you get the idea.

Array.of

This method is exactly like the first incarnation of `cast` we played with in our analysis of `Array.from`.

```
Array.of = function of () {
  return Array.prototype.slice.call(arguments)
}
```

You can't just replace `Array.prototype.slice.call` with `Array.of`. They're different animals.

```
Array.prototype.slice.call([1, 2, 3])
// <- [1, 2, 3]
Array.of(1, 2, 3)
// <- [1, 2, 3]
```

You can think of `Array.of` as an alternative for `new Array` that doesn't have the `new Array(length)` overload. Below you'll find some of the strange ways in which `new Array` behaves thanks to its single-argument `length` overloaded constructor. If you're confused about the `undefined x ${number}` notation in the browser console, that's indicating there are array holes in those positions.

```
new Array()
// <- []
new Array(undefined)
// <- [undefined]
new Array(1)
// <- [undefined x 1]
new Array(3)
```

```
// <- [undefined x 3]
new Array(1, 2)
// <- [1, 2]
new Array(-1)
// <- RangeError: Invalid array length
```

In contrast, `Array.of` has more consistent behavior because it doesn't have the special `length` case.

```
Array.of()
// <- []
Array.of(undefined)
// <- [undefined]
Array.of(1)
// <- [1]
Array.of(3)
// <- [3]
Array.of(1, 2)
// <- [1, 2]
Array.of(-1)
// <- [-1]
```

There's not a lot to add here – let's move on.

`Array.prototype.copyWithin`

This is the most obscure method that got added to `Array.prototype`. I suspect use cases lie around buffers and typed arrays – *which we'll cover at some point, later in the series.* The method copies a sequence of array elements *within the array* to the *"paste position"* starting at `target`. The elements that should be copied are taken from the `[start, end)` range.

Here's the signature of the `copyWithin` method. The `target` *"paste position"* **is required**. The `start` index where to take elements from defaults to `0`. The `end` position defaults to the length of the array.

```
Array.prototype.copyWithin(target, start = 0, end = this.length)
```

Let's start with a simple example. Consider the `items` array in the snippet below.

```
var items = [1, 2, 3, ,,,,,,,]
// <- [1, 2, 3, undefined x 7]
```

The method below takes the `items` array and determines that it'll start *"pasting"* items in the **sixth position**. It further determines that the items to be copied will be taken starting in the **second position** *(zero-based)*, until the **third position** *(also zero-based)*.

```
items.copyWithin(6, 1, 3)
// <- [1, 2, 3, undefined × 3, 2, 3, undefined × 2]
```

Reasoning about this method can be pretty hard. *Let's break it down.*

If we consider that the items to be copied were taken from the `[start, end)` range, then we can express that using the `.slice` operation. These are the items that were *"pasted"* at the `target` position. We can use `.slice` to *"copy"* them.

```
items.slice(1, 3)
// <- [2, 3]
```

We could then consider the "pasting" part of the operation as an advanced usage of `.splice` – one of those lovely methods that can do just about anything. The method below does just that, and then returns `items`, because `.splice` returns the items that were spliced from an Array, and in our case this is no good. Note that we also had to use the spread operator so that elements are inserted individually through `.splice`, and not as an array.

```
function copyWithin (items, target, start = 0, end = items.length) {
  items.splice(target, end - start, ...items.slice(start, end))
  return items
}
```

Our example would still work the same with this method.

```
copyWithin([1, 2, 3, ,,,,,,,], 6, 1, 3)
// <- [1, 2, 3, undefined × 3, 2, 3, undefined × 2]
```

The `copyWithin` method accepts negative `start` indices, negative `end` indices, and negative `target` indices. Let's try something using that.

```
[1, 2, 3, ,,,,,,,].copyWithin(-3)
// <- [1, 2, 3, undefined x 4, 1, 2, 3]
copyWithin([1, 2, 3, ,,,,,,,], -3)
// <- [1, 2, 3, undefined x 4, 1, 2, 3, undefined x 7]
```

Turns out, that thought exercise was useful for understanding `Array.prototype.copyWithin`, but it wasn't actually correct. Why are we seeing `undefined x 7` at the end? Why the discrepancy? The problem is that we are seeing the array holes at the end of `items` when we do

`...items.slice(start, end)` .

```
[1, 2, 3, ,,,,,,,]
// <- [1, 2, 3, undefined x 7]
[1, 2, 3, ,,,,,,,].slice(0, 10)
// <- [1, 2, 3, undefined x 7]
console.log(...[1, 2, 3, ,,,,,,,].slice(0, 10))
// <- 1, 2, 3, undefined, undefined, undefined, undefined, undefined, undefine
```

Thus, we *end up splicing the holes* onto `items` , while the original solution is not. We could get rid of the holes using `.filter` , which conveniently discards array holes.

```
[1, 2, 3, ,,,,,,,].slice(0, 10)
// <- [1, 2, 3, undefined x 7]
[1, 2, 3, ,,,,,,,].slice(0, 10).filter(el => true)
// <- [1, 2, 3]
```

With that, we can update our `copyWithin` method. We'll stop using `end - start` as the splice position and instead use the amount of `replacements` that we have, as those numbers may be different now that we're discarding array holes.

```
function copyWithin (items, target, start = 0, end = items.length) {
  var replacements = items.slice(start, end).filter(el => true)
  items.splice(target, replacements.length, ...replacements)
  return items
}
```

The case were we previously added extra holes now works as expected. Woo!

```
[1, 2, 3, ,,,,,,,].copyWithin(-3)
// <- [1, 2, 3, undefined x 4, 1, 2, 3]
copyWithin([1, 2, 3, ,,,,,,,], -3)
// <- [1, 2, 3, undefined x 4, 1, 2, 3]
```

Furthermore, our polyfill seems to work correctly *across the board* now. I wouldn't rely on it for anything other than educational purposes, though.

```
[1, 2, 3, ,,,,,,,].copyWithin(-3, 1)
// <- [1, 2, 3, undefined x 4, 2, 3, undefined x 1]
copyWithin([1, 2, 3, ,,,,,,,], -3, 1)
// <- [1, 2, 3, undefined x 4, 2, 3, undefined x 1]
[1, 2, 3, ,,,,,,,].copyWithin(-6, -8)
// <- [1, 2, 3, undefined x 1, 3, undefined x 5]
copyWithin([1, 2, 3, ,,,,,,,], -6, -8)
// <- [1, 2, 3, undefined x 1, 3, undefined x 5]
[1, 2, 3, ,,,,,,,].copyWithin(-3, 1, 2)
// <- [1, 2, 3, undefined x 4, 2, undefined x 2]
copyWithin([1, 2, 3, ,,,,,,,], -3, 1, 2)
// <- [1, 2, 3, undefined x 4, 2, undefined x 2]
```

It's decidedly better to just use the actual implementation, but at least now we have **a better idea of how the hell it works!**

`Array.prototype.fill`

Convenient utility method to fill all places in an `Array` with the provided `value`. Note that array holes will be filled as well.

```
['a', 'b', 'c'].fill(0)
// <- [0, 0, 0]
new Array(3).fill(0)
// <- [0, 0, 0]
```

You could also determine a start index and an end index in the second and third parameters respectively.

```
['a', 'b', 'c',,,].fill(0, 2)
// <- ['a', 'b', 0, 0, 0]
new Array(5).fill(0, 0, 3)
// <- [0, 0, 0, undefined x 2]
```

The provided value can be arbitrary, and not necessarily a number or even a primitive type.

```
new Array(3).fill({})
// <- [{}, {}, {}]
```

Unfortunately, you can't fill arrays using a mapping method that takes an `index` parameter or anything like that.

```
new Array(3).fill(function foo () {})
// <- [function foo () {}, function foo () {}, function foo () {}]
```

Moving along…

`Array.prototype.find`

Ah. One of those methods that JavaScript desperately wanted but didn't get in ES5. The `.find` method returns the *first* `item` that matches `callback(item, i, array)` for an `array` Array. You can also optionally pass in a `context` binding for `this`. You can think of it as an equivalent of `.some` that returns the matching element *(or `undefined`)* instead of merely `true` or `false`.

```
[1, 2, 3, 4, 5].find(item => item > 2)
// <- 3
[1, 2, 3, 4, 5].find((item, i) => i === 3)
// <- 4
[1, 2, 3, 4, 5].find(item => item === Infinity)
// <- undefined
```

There's really not much else to say about this method. It's just that simple! We did want this method a lot, as evidenced in libraries like Lodash and Underscore. Speaking of those libraries… – `.findIndex` was also born there.

`Array.prototype.findIndex`

This method is also an equivalent of `.some` and `.find`. Instead of returning `true`, like `.some`; or `item`, like `.find`; this method returns the `index` position so that `array[index] === item`. If none of the elements in the collection match the `callback(item, i, array)` criteria, the return value is `-1`.

```
[1, 2, 3, 4, 5].find(item => item > 2)
// <- 2
[1, 2, 3, 4, 5].find((item, i) => i === 3)
// <- 3
[1, 2, 3, 4, 5].find(item => item === Infinity)
// <- -1
```

```
// <- -1
```

Again, quite straightforward.

`Array.prototype.keys`

Returns an iterator that yields a sequence holding the keys for the array. The returned value is an iterator, meaning you can use it with all of the usual suspects like `for..of`, the spread operator, or by hand by manually calling `.next()`.

```
[1, 2, 3].keys()
// <- ArrayIterator {}
```

Here's an example using `for..of`.

```
for (let key of [1, 2, 3].keys()) {
  console.log(key)
  // <- 0
  // <- 1
  // <- 2
}
```

Unlike `Object.keys` and most methods that iterate over arrays, this sequence doesn't ignore holes.

```
[...new Array(3).keys()]
// <- [0, 1, 2]
Object.keys(new Array(3))
// <- []
```

Now onto values.

`Array.prototype.values`

Same thing as `.keys()` , but the returned iterator is a sequence of values instead of indices. In practice, you'll probably just iterate over the array itself, but sometimes getting an iterator can come in handy.

```
[1, 2, 3].values()
// <- ArrayIterator {}
```

Then you can use `for..of` or any other methods like a spread operator to pull out the sequence. The example below shows how using the spread operator on an array's `.values()` doesn't really make a lot of sense – *you already had that collection to begin with!*

```
[...[1, 2, 3].values()]
// <- [1, 2, 3]
```

Do note that the returned array in the example above is *a different array* and not a reference to the original one.

Time for `.entries` .

`Array.prototype.entries`

Similar to both preceding methods, but this one returns an iterator with a sequence of key-value pairs.

```
['a', 'b', 'c'].entries()
// <- ArrayIterator {}
```

Each entry contains a two dimensional array element with the key and the value for an item in the array.

```
[...['a', 'b', 'c'].entries()]
// <- [[0, 'a'], [1, 'b'], [2, 'c']]
```

Great, last one to go!

`Array.prototype[Symbol.iterator]`

This is ~~basically~~ **exactly** the same as the `.values` method. The example below combines a spread operator, an array, and `Symbol.iterator` to iterate over its values.

```
[...['a', 'b', 'c'][Symbol.iterator]()]
// <- ['a', 'b', 'c']
```

Of course, you should probably just omit the spread operator and the `[Symbol.iterator]` part in most use cases. Same time tomorrow? We'll cover changes to the `Object` API.