Shawn Reisner Follow
Full Stack JavaScript Expert. I am the code.
Jun 15 · 4 min read

Here's Why Mapping a Constructed Array in JavaScript Doesn't Work

And How To Do It Correctly

Scenario

For the sake of demonstration, suppose you need to generate an array of numbers from 0 to 99. How might you do this? Here's one option:

```
const arr = [];

for (let i = 0; i < 100; i++) {
  arr[i] = i;
}
```

If you're like me, seeing a traditional for-loop in JavaScript makes you slightly uncomfortable. In fact, I haven't written a traditional for-loop in *years* thanks to higher-order functions such as forEach, map, filter, and reduce. Declarative functional programming for the win!

Maybe you haven't yet drunk the functional programming Kool-aid and you're thinking the above solution seems perfectly fine. It technically is, but after you've had a taste of the magic of higher-order functions, you're probably thinking, "*There **must** be a better way.*"

My first reaction to this problem was, "*I know! I'll create an empty array of length 100 and map the indices to each element!*" JavaScript allows you create an empty array of length *n* with the Array constructor, like this:

```
const arr = Array(100);
```

Perfect, right? I have an array of length 100, so now I just need to map the index to each element.

```
const arr = Array(100).map((_, i) => i);

console.log(arr[0] === undefined);  // true
```

What the h*ck!? The first element of the array should be *0*, but it's actually *undefined*!

Explanation

There's an important technical distinction I need to make here in order to explain why this happened. Internally, JavaScript arrays **are** objects,

with numbers as keys. For example:

```
['a', 'b', 'c']
```

is essentially equivalent to this object:

```
{
  0: 'a',
  1: 'b',
  2: 'c',
  length: 3
}
```

When you access the element at index 0 in an array, you're really just accessing an object property whose key is 0. This is important because when you think of arrays as objects in conjunction with the way these higher-order functions are implemented (don't worry, I did that part for you), the cause of our problem makes perfect sense.

When you create a new array with the Array constructor, it creates a new array object with its *length* property set to the value you passed in, but otherwise *the object is a vacuum*. There are no index keys in the object representation of the array whatsoever.

```
{
  //no index keys!
  length: 100
}
```

You get *undefined* when you try to access the array value at index 0, but it's not that the value *undefined* is stored at index 0, it's that the default behavior in JavaScript is to return *undefined* if you try to access the value of an object for a key that does not exist.

It just so happens that the higher-order functions *map*, *reduce*, *filter*, and *forEach* iterate over the index keys of the array object from 0 to *length*, but the callback is only executed *if the key exists* on the object. This explains why our callback is never called and nothing happens when we call the map function on the array—there are no index keys!

Solution

As you now know, what we need is an array whose internal object representation contains a key for each number from 0 to *length*. The best way to do this is to spread the array out into an empty array.

```
const arr = [...Array(100)].map((_, i) => i);

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

Spreading the array into an empty array results in an array that's filled with *undefined* at each index.

```
{
  0: undefined,
  1: undefined,
  2: undefined,
  ...
  99: undefined,
  length: 100
}
```

This is because the spread operator is simpler than the map function. It simply loops through the array (or any iterable, really) from 0 to *length* and creates a new index key in the enclosing array with the value returned from the spreading array at the current index. Since JavaScript returns *undefined* from our spreading array at each of its indices (remember, it does this by default because it doesn't have the index key for that value), we end up with a new array that's actually filled with index keys, and therefore *map*-able (and *reduce*-able, *filter*-able, and *forEach*-able).

Conclusion

We discovered some of the implications of arrays being internally represented as objects in Javascript, and learned the best way to create arrays of arbitrary length filled with whatever values you need.

As always, leave your comments, questions, and feedback below!

Happy coding! 👍

. . .

Follow me!

If you liked this post, follow me! Or at least throw me a clap or two. You can spare one measly clap, right?

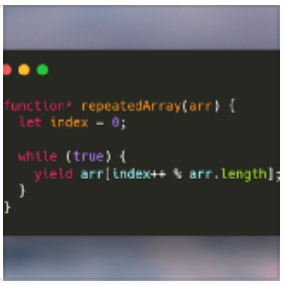
Medium: @shawn.webdev
Twitter: @ReisnerShawn
dev.to: @sreisner
Instagram: @shawn.webdev

Check out more of my work

A Quick, Practical Use Case for ES6 Generators

Building an Infinitely Repeating Array

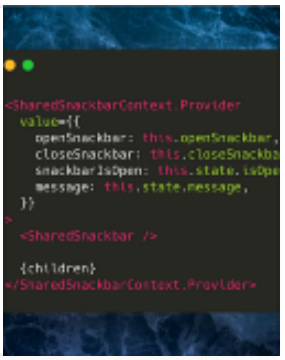
itnext.io



Learn the React Context API with a Practical Example You Can Bring to Your Apps

Building a Shared Material UI Snackbar for In-App Notifications

itnext.io



Building a "Mask Toggle" Password Input Component w/ React and Material UI

As everyone knows by now, allowing users to toggle password input visibility removes friction...

itnext.io