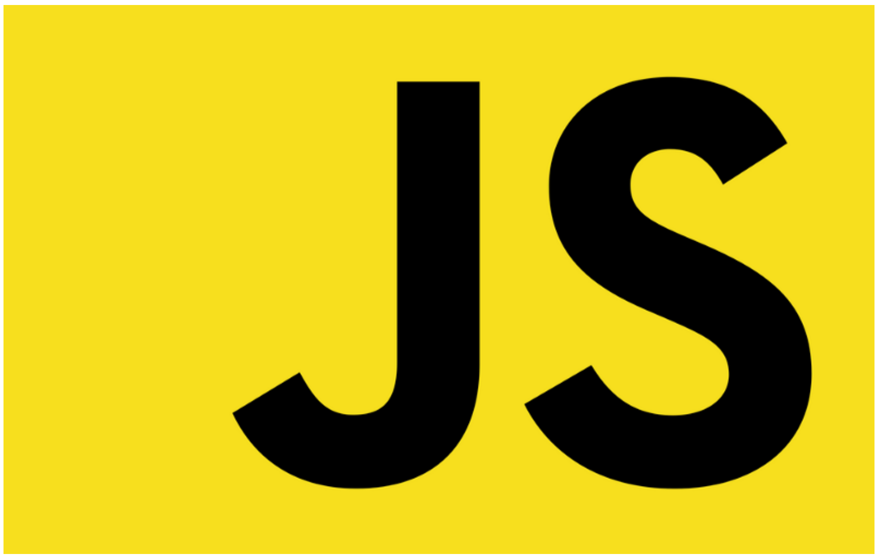
Arnav Aggarwal Follow
Full-Stack Developer
May 31 · 8 min read

Master Map & Filter, Javascript's Most Powerful Array Functions

Learn how Array.map and Array.filter work by writing them yourself



This article is for those who have written a `for` loop before, but don't quite understand how `Array.map` or `Array.filter` work. You should also be able to write a basic function. By the end of this, you'll have a complete understanding of both functions, because you'll have seen how they're written. Let's jump right in.

Array.map

`Array.map` is meant to transform one array into another by performing some operation on each of its values. The original array is left untouched and the function returns a new, transformed array. For example, say we have an array of numbers and we want to **multiply each number by three**. We also don't want to change the original array. To do this without `Array.map`, we can use a standard for-loop.

for-loop

```
var originalArr = [1, 2, 3, 4, 5];
var newArr = [];

for(var i = 0; i < originalArr.length; i++) {
    newArr[i] = originalArr[i] * 3;
}

console.log(newArr); // -> [3, 6, 9, 12, 15]
```

Simple enough. Let's abstract this loop into its own function so that we can turn any array we like into a new array with each element multiplied by 3. In other words, we're trying to write a function that will take in an any array (`[1, 2, 3]`) and spit out a brand new array with its numbers multiplied by three (`[3, 6, 9]`). All we have to do is take the code we wrote above and turn it into a function so that we can reuse that loop over and over. This might seem difficult, but try to get through it.

Multiply by three

```
var originalArr = [1, 2, 3, 4, 5];

function multiplyByThree(arr) {
    var newArr = [];

    for(var i = 0; i < arr.length; i++) {
        newArr[i] = arr[i] * 3;
    }

    return newArr;
}

var arrTransformed = multiplyByThree(originalArr);
console.log(arrTransformed); // -> [3, 6, 9, 12, 15]
```

Beautiful. Now we can pass any array into `multiplyByThree` and get a new array out with its values multiplied. Now, we're going to add some code that might seem useless, but bear with me here. Let's take a single line in that function— `newArr[i] = thisItem * 3` —and turn it into its own function as well. The result will be code that is equivalent to that in the block above, but we'll need it this way right after.

```
var originalArr = [1, 2, 3, 4, 5];

function timesThree(item) {
    return item * 3;
}

function multiplyByThree(arr) {
    var newArr = [];

    for(var i = 0; i < arr.length; i++) {
        newArr[i] = timesThree(arr[i]);
    }

    return newArr;
}

var arrTransformed = multiplyByThree(originalArr);
console.log(arrTransformed); // -> [3, 6, 9, 12, 15]
```

This block does the same exact thing as the one before it. It just takes one piece out and turns it into its own function.

What if we wanted to multiply all items in an array by 5? or 10? Would we want to make a new looping function for each of those? No, not at all. That would be tedious and repetitive.

Multiply by anything

Let's change the `multiplyByThree` code to be able to multiply by anything. Let's rename it to just `multiply`. **This is the hardest part** and might take some time to wrap your head around, but try to get through it. Afterwards, it's easy.

We're turning this:

```
function multiplyByThree(arr) {
    var newArr = [];

    for(var i = 0; i < arr.length; i++) {
        newArr[i] = timesThree(arr[i]);
    }

    return newArr;
}
```

Into this.

```
function multiply(arr, multiplyFunction) {
    var newArr = [];

    for(var i = 0; i < arr.length; i++) {
        newArr[i] = multiplyFunction(arr[i]);
    }

    return newArr;
}
```

We've renamed the function and given it an extra argument to take in. That argument itself will be a function (commonly referred to as a callback). Now, we're passing a function in to `multiply` ourselves, telling `multiply` how we want each item transformed. Using our brand new function, let's **multiply by 3** again.

```
var originalArr = [1, 2, 3, 4, 5];

function timesThree(item) {
    return item * 3;
}

var arrTimesThree = multiply(originalArr, timesThree);
console.log(arrTimesThree); // -> [3, 6, 9, 12, 15]
```

We're giving our `multiply` function the instructions it needs to transform each value in the array by passing in the `timesThree` function. What if we want to **multiply by 5** instead? We just give it different instructions, or a different function.

```
var originalArr = [1, 2, 3, 4, 5];

function timesFive(item) {
    return item * 5;
}

var arrTimesFive = multiply(originalArr, timesFive);
console.log(arrTimesFive); // -> [5, 10, 15, 20, 25]
```

It's as simple as swapping out the `timesThree` function for a `timesFive` function. Repeating this technique, we can multiply by any number we want—we just write a new, very simple function.

We've written one single for-loop and can multiply an array by whatever we want.

Map

Let's make `multiply` even more powerful. Instead of multiplying by something, let's allow the function to transform our array any way we want. Let's rename `multiply` to, oh, I don't know… how about `map`? So, we're turning this:

```
function multiply(arr, multiplyFunction) {
    var newArr = [];

    for(var i = 0; i < arr.length; i++) {
        newArr[i] = multiplyFunction(arr[i]);
    }

    return newArr;
}
```

Into this.

```
function map(arr, transform) {
    var newArr = [];

    for(var i = 0; i < arr.length; i++) {
        newArr[i] = transform(arr[i]);
    }

    return newArr;
}
```

Look closely at the bold and see what we changed between the `multiply` and `map` functions directly above. The only things we changed were the *name of the function and the name of the second parameter that it takes in*. That's it. Turns out, `multiply` was already what we wanted, named differently.

We can pass in any function we want to `map`. We can do transform an array any way we want. Say we have an array of strings, and we want to turn them all uppercase:

```
function makeUpperCase(str) {
   return str.toUpperCase();
}

var arr = ['abc', 'def', 'ghi'];
var ARR = map(arr, makeUpperCase);

console.log(ARR); // -> ['ABC', 'DEF, 'GHI']
```

We've effectively just written `Array.map`. Pretty neat, huh?

Using Array.map

How does the `map` function compare to the actual native `Array.map`? The usage is slightly different. Firstly, we don't need to pass in an array as the first argument. Instead, the array used is the one to the left of the dot. As an example, the following two are equivalent. Using our function:

```
function func(item) {
   return item * 3;
}

var arr = [1, 2, 3];
var newArr = map(arr, func);

console.log(newArr); // -> [3, 6, 9]
```

Using the native `Array.map`, we don't pass in the array. We *call* the `Array.map` method *on our array* and only pass in the function:

```
function func(item) {
   return item * 3;
}

var arr = [1, 2, 3];
var newArr = arr.map(func);

console.log(newArr); // -> [3, 6, 9]
```

That's pretty much it. You've written `Array.map` by yourself and now know how it works.

More `Array.map` arguments

There's a key difference we've skipped over. `Array.map` will provide your given function with an additional two arguments: the index, and the original array itself.

```
function logItem(item) {
    console.log(item);
}

function logAll(item, index, arr) {
    console.log(item, index, arr);
}

var arr = ['abc', 'def', 'ghi'];

arr.map(logItem); // -> 'abc', 'def', 'ghi'

arr.map(logAll); // -> 'abc', 0, ['abc', 'def', 'ghi']
                 // -> 'def', 1, ['abc', 'def', 'ghi']
                 // -> 'ghi', 2, ['abc', 'def', 'ghi']
```

This allows you to use the index and the original array inside your transformation function if you choose. For example, say we want to turn an array of items into a numbered shopping list. We'd want to use the index:

```
function multiplyByIndex(item, index) {
    return (index + 1) + '. ' + item;
}

var arr = ['bananas', 'tomatoes', 'pasta', 'protein
shakes'];
var mappedArr = arr.map(multiplyByIndex);

console.log(mappedArr); // ->
// ["1. bananas", "2. tomatoes", "3. pasta", "4. protein
shakes"]
```

Our complete `map` function should incorporate this functionality.

```
function map(arr, transform) {
    var newArr = [];

    for(var i = 0; i < arr.length; i++) {
        newArr[i] = transform(arr[i], i, arr);
    }


    return newArr;
}
```

This short function is the essence of `Array.map`. The actual function will have some error-checking and optimizations, but this is its core functionality.

Lastly, you can also write a function directly in the `map` call. Reworking our multiplyByThree example:

```
var arr = [1, 2, 3, 4, 5];


var arrTimesThree = arr.map(function(item) {
   return item * 3;
});


console.log(arrTimesThree); // -> [3, 6, 9, 12, 15]
```

Done. Phew.

Array.filter

The idea here is similar to Array.map, except instead of transforming individual values, we want to filter existing values. Without any functions (besides `Array.push`), say we want to filter out values in an array that are less than 5:

for-loop

```
var arr = [2, 4, 6, 8, 10];
var filteredArr = [];


for(var i = 0; i < arr.length; i++) {
    if(arr[i] >= 5) {
        filteredArr.push(arr[i]);
    }
}


console.log(filteredArr); // -> [6, 8, 10]
```

Let’s abstract this to a function so we can remove values below 5 in any array.

```
function filterLessThanFive(arr) {
    var filteredArr = [];

    for(var i = 0; i < arr.length; i++) {
        if(arr[i] >= 5){
            filteredArr.push(arr[i]);
        }
    }

    return filteredArr;
}

var arr1 = [2, 4, 6, 8, 10];
var arr1Filtered = filterLessThanFive(arr1);

console.log(arr1Filtered); // -> [6, 8, 10]
```

Let’s make it so we can filter out all values below any arbitrary value.

```
function isGreaterThan5(item) {
    return item > 5;
}

function filterLessThanFive(arr) {
    var filteredArr = [];

    for(var i = 0; i < arr.length; i++) {
        if(isGreaterThan5(arr[i])) {
            filteredArr.push(arr[i]);
        }
    }

    return filteredArr;
}

var originalArr = [2, 4, 6, 8, 10];
var newArr = filterLessThanFive(originalArr);

console.log(newArr); // -> [6, 8, 10]
```

→ Abstracting out the filtering functionality

```
function filterBelow(arr, greaterThan) {
    var filteredArr = [];

    for(var i = 0; i < arr.length; i++) {
        if(greaterThan(arr[i])) {
            filteredArr.push(arr[i]);
        }
    }

    return filteredArr;
}

var originalArr = [2, 4, 6, 8, 10];
```

→ Filtering out anything below 5, using filterBelow

```
function isGreaterThan5(item) {
    return item > 5;
}

var newArr = filterBelow(originalArr, isGreaterThan5);

console.log(newArr); // -> [6, 8, 10];
```

→ Filtering out anything below 7, using filterBelow

```
function isGreaterThan7(item) {
    return item > 7;
}

var newArr2 = filterBelow(originalArr, isGreaterThan7);

console.log(newArr2); // -> [8, 10];
```

filter

So we have a function `filterBelow` that will filter out anything below a certain value, based on the `greaterThan` function we give it (this is identical to the `filterBelow` function above):

```
function filterBelow(arr, greaterThan) {
    var filteredArr = [];
    for(var i = 0; i < arr.length; i++) {
        if(greaterThan(arr[i])) {
            filteredArr.push(arr[i]);
        }
    }

    return filteredArr;
}
```

Let’s rename it.

```
function filter(arr, testFunction) {
    var filteredArr = [];

    for(var i = 0; i < arr.length; i++) {
        if(testFunction(arr[i])) {
            filteredArr.push(arr[i]);
        }
    }

    return filteredArr;
}
```

And we’ve written `filter`. It’s basically the same as `Array.filter`, again except for usage:

```
var arr = ['abc', 'def', 'ghijkl', 'mnopuv'];

function longerThanThree(str) {
    return str.length > 3;
}

var newArr1 = filter(arr, longerThanThree);
var newArr2 = arr.filter(longerThanThree);

console.log(newArr1); // -> ['ghijkl', 'mnopuv']
console.log(newArr2); // -> ['ghijkl', 'mnopuv']
```

Again, `Array.filter` passes in the index and the original array to your function.

```
function func(item, index, arr) {
    console.log(item, index, arr);
}

var arr = ['abc', 'def', 'ghi'];

arr.filter(func); // -> 'abc', 0, ['abc', 'def', 'ghi']
                  // -> 'def', 1, ['abc', 'def', 'ghi']
                  // -> 'ghi', 2, ['abc', 'def', 'ghi']
```

So we should make our function do the same.

```
function filter(arr, testFunction) {
    var filteredArr = [];

    for(var i = 0; i < arr.length; i++) {
        if(testFunction(arr[i], i, arr)) {
            filteredArr.push(arr[i]);
        }
    }

    return filteredArr;
}
```

Wow. That's it. You've learned how to use *and write* `Array.map` and `Array.filter`. Go write some code.

Notes

If there's interest in covering `Array.forEach` or `Array.reduce` as well, let me know in the comments. forEach is pretty similar to map and filter, but reduce is a bit more tricky.

If you liked this, please press the heart and feel free to check out my other publications.

- Master the Power Behind Javascript's Logical Operators
- Master Javascript's New, Cutting-Edge Object Spread Operator