# Map

Learn how `Array.map` works by learning how to write it yourself. We'll see what makes this function so powerful and how it allows us to work efficiently with large amounts of data. You'll use this functions thousands of times in your JavaScript career.

## `Array.map`

Learn how `Array.map` works by writing it yourself.

This function allows us to manipulate data effectively and efficiently. It's meant to copy an array and change it a little bit in the process. You'll be using it for the rest of your JavaScript career.

`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 reference to a new array.

Because the original data source, the original array, is left untouched, `Array.map` is a pure function. It's the first of a few methods that will allow us to really use functional programming to perform useful tasks, without feeling like we're adding complexity.

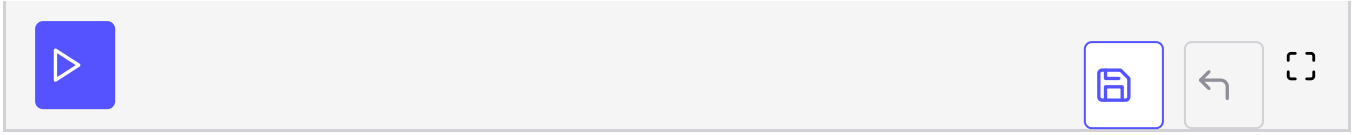
# Introduction

Let's 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

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

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

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

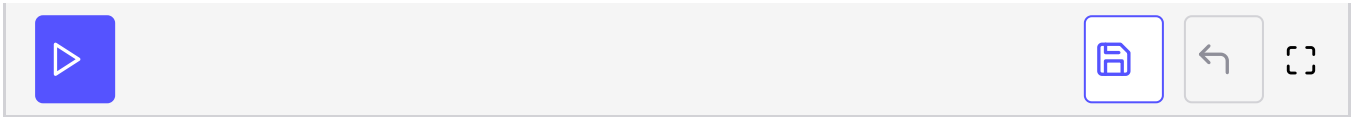


Simple enough. Let's abstract this loop into its own function so that we can turn any array we like into a new array with each element multiplied by 3. In other words, we're trying to write a function that will take in an any array (`[1, 2, 3]`) and spit out a brand new array with its numbers multiplied 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

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

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

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

    return newArr;
}

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

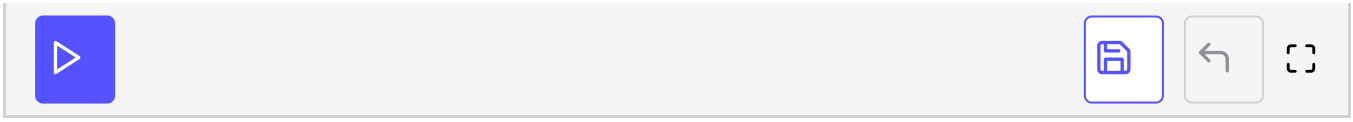


Beautiful. Now we can pass any array into `multiplyByThree` and get a new array out with its values multiplied. Now, we're going to add some code that might seem useless, but bear with me here.

Let's take a single line in that function — line 7 — 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.

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

function timesThree(item) {
```

```
    return item * 3;
}

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

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

    return newArr;
}

const 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) {
    const newArr = [];

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

    return newArr;
}
```

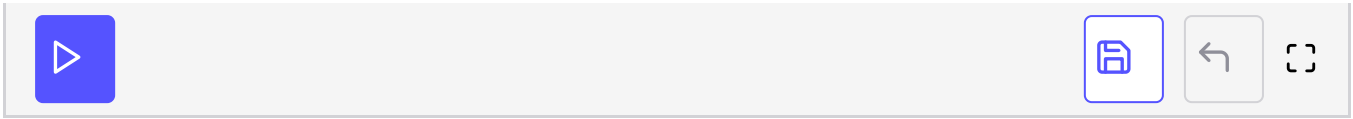
Into this. Differences are on lines 1 and 5.

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

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


    return newArr;
}
```

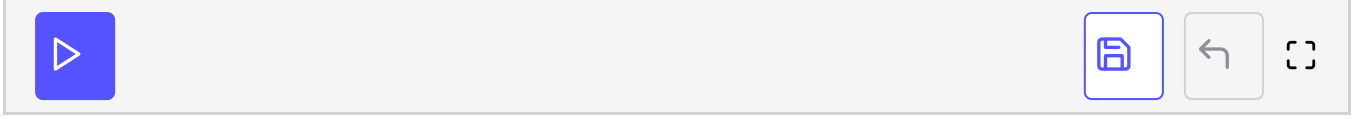
We've renamed the function and given it an extra argument to take in. That argument itself will be a callback function. Now, we're passing a function in to `multiply` ourselves, telling `multiply` how we want each item transformed. Using our brand new function, let's **multiply by 3** again.

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

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

    return newArr;
}

const originalArr = [1, 2, 3, 4, 5];

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

const 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.

```
// While 'multiply' from above isn't visible,
// it's still available in this code block.

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. With one single for-loop, we 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) {
    const newArr = [];

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

    return newArr;
}
```

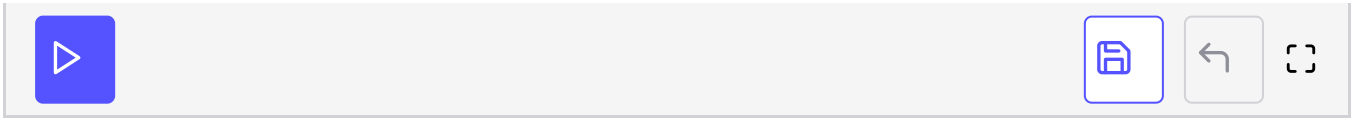
Into this.

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

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

    return newArr;
}
```

Look closely at lines 1 and 5 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 map(arr, transform) {
    const newArr = [];

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

    return newArr;
}

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

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

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

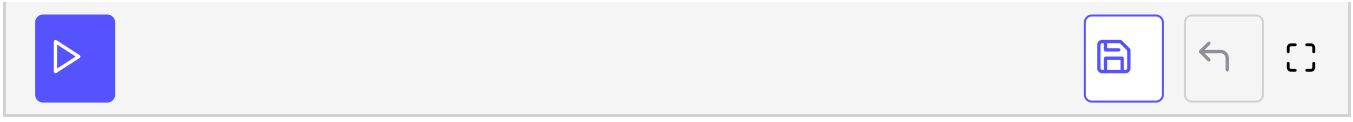


We've effectively just written `Array.map`. Pretty neat, huh?

## 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:

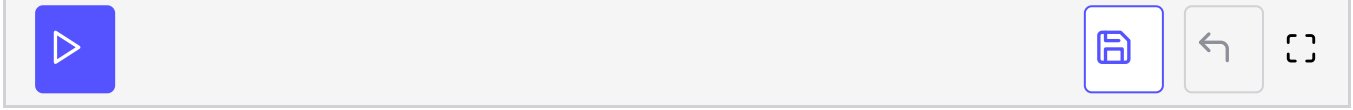
```
// While 'map' from above isn't visible,
// it's still available in this code block.

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

const arr = [1, 2, 3];
const 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) {
```

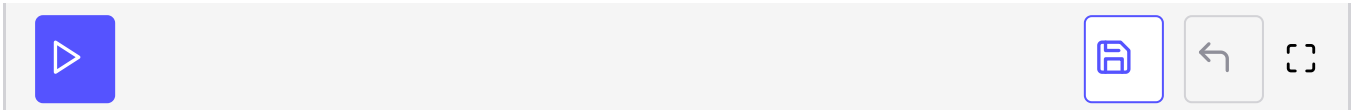
```
    return item * 3;
}

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

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



## More arguments

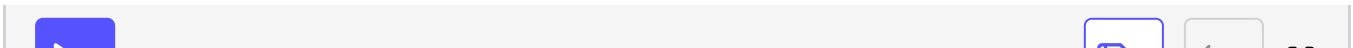
There's a key difference we've skipped over. `Array.map` will provide your given function with an additional two arguments: the index, and the original array itself.

```
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 us to use the index and the original array inside our transformation function if we 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 numberItem(item, index) {
    return (index + 1) + '. ' + item;
}

const arr = ['bananas', 'tomatoes', 'pasta', 'avocado'];
const mappedArr = arr.map(numberItem);

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



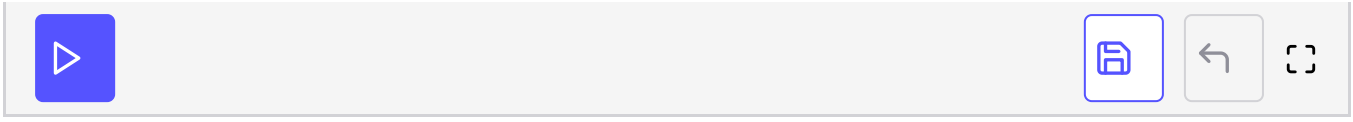
Our complete `map` function should incorporate this functionality.

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

    for(let 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.

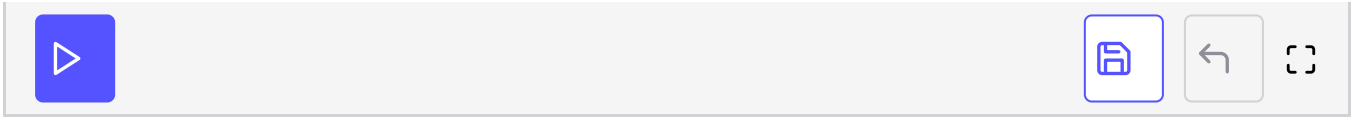
Lastly, of course, we can also write a function directly in the map call. Reworking our `multiplyByThree` example:

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

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

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



And this looks even better with arrow functions.

```
const arr = [1, 2, 3, 4, 5];
const arrTimesThree = arr.map(item => item * 3);
console.log(arrTimesThree); // -> [3, 6, 9, 12, 15]
```



Done. Phew.

We'll cover `Array.filter` next.