# Mapping Arrays

(https://github.com/learn-co-curriculum/phase-1-mapping-arrays-readme) (https://github.com/learn-co-curriculum/phase-1-mapping-arrays-readme/issues/new)

## Learning Goals

- Implement a `map()` function from scratch
- Demonstrate using `map()`

## Introduction

In the previous lesson, we learned about `.filter()`, a built-in array method that searches through a collection, passes each element to a provided callback function, and returns an entirely new array comprised of elements for which the callback returned a truthy value.

Another very common built-in array method is `.map()`, which transforms every element in an array to another value. For example, it can be used to square every value in an array of numbers: `[1, 2, 3]` -> `[1, 4, 9]`. Like `.filter()`, `.map()` accepts a callback function, and passes each element in turn to the callback:

```
[1, 2, 3].map(function (num) {
  return num * num;
});
// => [1, 4, 9]
```

While both `.filter()` and `.map()` return a new array, `.filter()` returns a subset of the original array (unless all elements meet the provided condition) in which the elements are unchanged. `.map()`, on the other hand, returns a new array that's the same length as the original array in which the elements have been modified.

Let's quickly run through how we could create our own version of the `.map()` method.

## Implementing `.map()` From Scratch

# Abstracting the iteration

Right off the bat, we know that our function needs to accept the array from which we'd like to *map* values as an argument:

```
function map(array) {
  // Map magic to follow shortly
}
```

Inside the function, we need to iterate over each element in the passed-in array, so let's fall back on our trusty `for...of` statement:

```
function map(array) {
  for (const element of array) {
    // Do something to each element
  }
}
```

# Callback city

As we've discussed, `map()` is used to transform each element of an array and return a new array containing the transformed values. However, for code organization and reusability it's best to keep the logic of the transformation decoupled from the `map()` function itself. `map()` should really only be concerned with iterating over the collection and passing each element to a callback that will handle the transformations. Let's set up `map()` to take that callback function as its second argument:

```
function map(array, callback) {
  for (const element of array) {
    // Do something to each element
  }
}
```

And inside our iteration, we'll want to invoke the callback, passing in the elements from `array` :

```
function map(array, callback) {
  for (const element of array) {
    callback(element);
  }
}
```

Let's make sure this is working so far:

```
map([1, 2, 3], function (num) {
  console.log(num * num);
});
// LOG: 1
// LOG: 4
// LOG: 9
```

## Returning a brand new collection

Logging each squared number out to the console is fun, but `map()` should really be returning an entirely new array containing all of the squared values. First, let's create that new array:

```
function map(array, callback) {
  const newArr = [];

  for (const element of array) {
    callback(element);
  }
}
```

Inside the `for...of` statement, let's `.push()` the return value of each callback invocation into `newArr`:

```
function map(array, callback) {
  const newArr = [];
```

```
  for (const element of array) {
    newArr.push(callback(element));
  }
}
```

And at the end of our `map()` function we're going to want to return the new array:

```
function map(array, callback) {
  const newArr = [];

  for (const element of array) {
    newArr.push(callback(element));
  }

  return newArr;
}
```

Let's test it out!

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

const squaredNumbers = map(originalNumbers, function (num) {
  return num * num;
});

originalNumbers;
// => [1, 2, 3, 4, 5]

squaredNumbers;
// => [1, 4, 9, 16, 25]
```

# Demonstrate Using `map()` on Flatbook's Expanding Engineering Team

Now let's try using our version of the `map()` function on a trickier data structure — a list of recently onboarded engineers. First off, we need to flip each new engineer's account from a normal user to an admin:

```
const oldAccounts = [
  { userID: 15, title: "Developer Apprentice", accessLevel: "user" },
  { userID: 63, title: "Developer Apprentice", accessLevel: "user" },
  { userID: 97, title: "Developer Apprentice", accessLevel: "user" },
  { userID: 12, title: "Developer Apprentice", accessLevel: "user" },
  { userID: 44, title: "Developer Apprentice", accessLevel: "user" },
];

const newEngineers = map(oldAccounts, function (account) {
  return Object.assign({}, account, { accessLevel: "admin" });
});

oldAccounts;
// => [
//      { userID: 15, title: "Developer Apprentice", accessLevel: "user" },
//      { userID: 63, title: "Developer Apprentice", accessLevel: "user" },
//      { userID: 97, title: "Developer Apprentice", accessLevel: "user" },
//      { userID: 12, title: "Developer Apprentice", accessLevel: "user" },
//      { userID: 44, title: "Developer Apprentice", accessLevel: "user" }
//    ]

newEngineers;
// => [
//      { userID: 15, title: "Developer Apprentice", accessLevel: "admin" },
//      { userID: 63, title: "Developer Apprentice", accessLevel: "admin" },
//      { userID: 97, title: "Developer Apprentice", accessLevel: "admin" },
//      { userID: 12, title: "Developer Apprentice", accessLevel: "admin" },
```

```
//      { userID: 44, title: "Developer Apprentice", accessLevel: "admin" }
//    ]
```

As before, we are calling our version of the `map()` function and passing in the collection and a callback. Notice that we're using `Object.assign()` to create a **new** object with updated values instead of mutating the original object's `accessLevel` property. Nondestructive updating is an important concept to practice — destructively modifying objects at multiple points within a code base is one of the biggest sources of bugs.

Next, we just need a simple array of the new engineers' `userID` s that we can shoot over to the system administrator:

```
const userIDs = map(newEngineers, function (eng) {
  return eng.userID;
});

userIDs;
// => [15, 63, 97, 12, 44]
```

Finally, we'll update our engineer objects to indicate that all the new engineers have been provided a new work laptop. This time, though, let's use JavaScript's built-in `Array.prototype.map()` method:

```
const equippedEngineers = newEngineers.map(function (eng) {
  return Object.assign({}, eng, { equipment: "Laptop" });
});

equippedEngineers;
// => [
//      { userID: 15, title: "Developer Apprentice", accessLevel: "admin", equipment: "Laptop" },
//      { userID: 63, title: "Developer Apprentice", accessLevel: "admin", equipment: "Laptop" },
//      { userID: 97, title: "Developer Apprentice", accessLevel: "admin", equipment: "Laptop" },
//      { userID: 12, title: "Developer Apprentice", accessLevel: "admin", equipment: "Laptop" },
//      { userID: 44, title: "Developer Apprentice", accessLevel: "admin", equipment: "Laptop" }
//    ]
```

Note how similar this method call is to the one using our version of `map()` : the only difference is that we call the built-in `.map()` method *on* our array, rather than passing the array as an argument. There *is* one big difference between the two, though: we didn't have to do all the work of building `Array.prototype.map()` !

Now that we understand how the built-in `.map()` array method is implemented, we can stick to the native method and get rid of our copycat `map()` function.

# Resources

- **MDN — `Array.prototype.map()` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map)**