Lesson | Thursday

Introduction to Programming (/introduction-to-programming) / Arrays and Looping (/introduction-to-programming/arrays-and-looping) / Array Mapping

Text | Cheat sheet

Note: While you are not required to use `Array.prototype.map()` for this independent project, it is one of the most powerful JavaScript looping methods. We will be covering it more in depth in the React curriculum as well.

So far, we've explored two commonly used JavaScript functions for looping:

- `Array.prototype.forEach()`
- `for`

Both of the JavaScript techniques for looping we've learned are very open-ended. They simply loop until a condition is met. We can do whatever we want with that loop, whether that's create a new array with modified elements, sum a value, and so on.

However, JavaScript also provides some other high level methods that don't just iterate — they *transform* the array they are looping over. In this lesson, we'll discuss `Array.prototype.map()`, a high level method. Later when we learn about functional programming in the React course, we'll learn about other higher level looping methods like `Array.prototype.reduce()` and `Array.prototype.filter()`.

Array.prototype.map()

First, what do we mean by *high level*? Well, in the sense of a programming language, a *low level* language is one that's close to how a machine operates (such as the assembly language) while a *high level* language is one that's closer to how humans think and communicate — such as JavaScript.

While methods like `for` aren't *low level*, they are a little bit closer to how our machine thinks: we have to set the start and end conditions as well as an incrementer. We also have to write out all the code that dictates what will happen during each iteration.

`Array.prototype.forEach()` is higher level — we don't have to specify start and end conditions or an incrementer — `Array.prototype.forEach()` will automatically iterate through every element in the array. While this makes it "easier" to use in many ways, it's also a bit more abstract — which can be challenging for beginning coders.

`Array.prototype.map()` takes this one step further. If we want to have a new array with modified elements, this method will handle that for us. There is no need to first initialize a new array and then push modified elements into it. For that reason, it's cleaner and uses less code.

Let's return to our element-doubling example. We can double each element in an array and save the doubled elements using `Array.prototype.forEach()`:

```
> const array = [0,1,2,3,4,5];
> let doubledArray = [];
> array.forEach(function(element) {
  doubledArray.push(element * 2);
});
> doubledArray;
(6) [0,2,4,6,8,10]
```

Let's see how we can do the same thing with `Array.prototype.map()`:

```
> const array = [0,1,2,3,4,5];
> const doubledArray = array.map(function(element) {
  return element * 2;
});
> doubledArray;
(6) [0,2,4,6,8,10]
```

As we can see here, we no longer need to initialize an empty array. Instead, we can save the results of `Array.prototype.map()` inside a variable (which we call `doubledArray`).

Like `Array.prototype.forEach()`, `Array.prototype.map` takes a function as an argument, which is called a callback function.

However, there's a huge difference here: we can't use a `return` statement with `Array.prototype.forEach()`. With `Array.prototype.map()`, we *must* use a `return` statement.

Each time we iterate with `Array.prototype.map()`, we are specifying how the element should be transformed. The transformation occurs and the transformed element will be placed in a new array for us. That's why we need to have a `return` statement within `Array.prototype.map()` — to make sure the transformed element gets saved to the new array.

It should be clear how this even more "higher level" than `Array.prototype.forEach()` — `Array.prototype.map()` does more for us, abstracting away the need to initialize a new array and push elements into it.

While we *could* omit the `return` keyword from our `Array.prototype.map()` callback (JavaScript won't complain), it will break our code. Without the `return` keyword, `doubledArray` will look like this:

```
[undefined, undefined, undefined, undefined, undefined, und
efined]
```

This is exactly what happens when we forget to return from a function: the value of the variable storing the invoked function will be `undefined`.

Another big thing to note here — `Array.prototype.map()` returns a transformed array. We can't use it to sum the values of an array (as we can with `Array.prototype.forEach()`). There's actually another method we can use to sum values called `Array.prototype.reduce()`. While you're welcome to explore that method on your own, we won't cover it until we discuss functional programming.

When to Use `Array.prototype.map()`

When should we favor `Array.prototype.map()` over `Array.prototype.forEach()`? Well, first a word of caution — you should really feel solid using `Array.prototype.forEach()` before you start experimenting with `Array.prototype.map()` too much. At this stage in your coding development, it's fine if you only use `Array.prototype.forEach()`, at least until you feel very comfortable with it.

However, in your long-term development as a coder, there will come a point (sooner rather than later) where you should favor `Array.prototype.map()` any time you want to create a new array where all the elements have been transformed.

We can't use it to return a string as we do when we determine things we like. However, we could do this instead:

```
> const arrayOfThingsILike = ["bubble baths", "kittens", "g
ood books", "clean code"];
> const thingsILike = arrayOfThingsILike.map(function(thin
g) {
  return "I like " + thing + "!";
});
> thingsILike.join(" ");
> thingsILike;
(4) ['I like bubble baths!', 'I like kittens!', 'I like goo
d books!', 'I like clean code!']
```

In this example, we create an array of transformed strings and then join the elements in the array to get 4 separate sentences.

`Array.prototype.map()` is extremely powerful. It is, in fact, one of the most powerful and important methods in JavaScript. If you feel comfortable using `for` and `` `Array.prototype.forEach() `` loops, you can experiment with `Array.prototype.map()` now. If you are still getting the hang of `for` and `Array.prototype.forEach()` (which is completely expected and even likely considering we've just started learning about loops), store this information away for now and focus on `for` and `Array.prototype.forEach()`.

While you are expected to loop for the next independent project, you are not expected to use `` `Array.prototype.map() `` — though you may do so if you wish.

Documentation on MDN

For more information, check out the Mozilla documentation on `Array.prototype.map()`. (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map)

Previous (/introduction-to-programming/arrays-and-looping/further-exploration-regular-expressions-with-text-analyzer)
Next (/introduction-to-programming/arrays-and-looping/practice-array-mapping)

disable dark mode

(http://www.epicodus.com)