Lesson Weekend

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

Text Cheat sheet

Over the last two lessons, we've learned the basics of how arrays work, the `length` property that every array has, and how we can look inside of arrays. We also learned about off-by-one errors — and there will be plenty more reminders about OBOEs throughout this section (mostly when we make the OBOE in our code) because they are so common.

In this lesson, we'll learn about some of the most common array methods. In the process, we'll make an important distinction because some methods **mutate** the arrays they are called on while some don't. When a method mutates an array, it is permanently changed, while methods that don't mutate arrays actually create (and return) brand new arrays.

As you read through this lesson, optionally use the DevTools console to try out the new code.

Array Methods

`Array.prototype.push()` Versus `Array.prototype.concat()`

Let's look at an example of a method that mutates an array versus one that doesn't. We'll start by returning to `Array.prototype.push()`. Yes, we've already learned this method — however, it's so important that it's worth reviewing it. It's also a key example of a method that mutates the array it's called on. These kinds of methods are also known as **destructive methods** because they change the receiver (the thing the method is called on).

```
> let numArr = [1,2,3];
> numArr.push(4);
4
> numArr;
[1,2,3,4]
```

As we can see in this example, `numArr` has been mutated. `Array.prototype.push()` only returns the length of the array (`4`), and not a new array like non-destructive methods do.

Let's compare this to the `Array.prototype.concat()` method, which is a non-destructive method.

```
> const originalArray = [1,2,3];
> const modifiedArray = originalArray.concat(4);
> originalArray;
[1,2,3]
> modifiedArray;
[1,2,3,4]
```

First, note that we used `const` here instead of `let`. This is because `originalArray` isn't being modified when we call `Array.prototype.concat()` on it. This method returns a new array, which is what all **non-destructive** methods do. The value of this

new array is the original array plus the arguments passed into `Array.prototype.concat()`. In this case, it's doing the same thing as if we just used `Array.prototype.push()`, right? So what's the difference?

Well, it's a pretty big deal that our original array isn't modified. What if we want to use the original array for another operation later? It would be better to use `Array.prototype.concat()`. Also, we really need to make sure we store the return of `Array.prototype.concat()` in a variable (because it won't be saved otherwise) while we don't need to do so with `Array.prototype.push()`.

There's also a whole school of thought that we should *never* modify arrays or other objects — only create new ones. This is a central concept of a programming style called **functional programming** which we will learn about once we reach the React course.

For now, we aren't making distinctions about one being better or worse. It's just important to know both of these methods — and to know which methods mutate the array they are called on and which don't.

Using Bracket Notation to Modify Elements in an Array

In the last lesson, we learned how to use bracket notation to see the value of an array. For example:

```
> let array = [1,2,3];
> array[0];
1
```

We can also use bracket notation to modify elements in an array:

```
> let array = [1,2,3]
> array[0] = "We just modified the array at position zer
o.";
> array;
["We just modified the array at position zero.",2,3];
```

In this example, we assigned a new value to `array[0]`: `"We just modified the array at position zero."`.

Modifying elements in arrays like this is super useful. Make a mental note of this now so you can add this to your JS toolbox right away.

As you can probably guess, you can reassign elements in an array even if it's a `const`. If an array is a `const`, it's only the array *itself* that can't be reassigned.

We can also use bracket notation to add an element to an array. Be careful, though — it's not as effective as using `Array.prototype.push()` or other methods. Here's why:

```
> let array = [1,2,3];
> array[5] = 4;
4
> array;
[1, 2, 3, , , 4]
```

In the example above, we assign the number `4` to `array` at an index of `5`. There is no index at `5`, though. JavaScript is happy to expand our array so it can insert this value at the 5th index — but that means it has to add some empty elements (the DevTools console will display this as `[1, 2, 3, empty × 2, 4]`). Yuck! In general, we don't want to have empty spots in an array, though

there is a use case for just about everything in JavaScript. It would certainly work to do the following (and have the exact same effect as `Array.prototype.push()`):

```
> array[array.length] = 4;
```

Since the length of an array would be one element *past* the final element in the current array, this works correctly. It's a convoluted approach to adding an element in an array, though, and you'll probably never see it in the real world. As always, though, it's good to see the flexibility and power of each tool we have in our toolbox.

`Array.prototype.unshift()` Versus `Array.prototype.shift()`

Okay, the difference between these two methods is confusing — and don't worry... lots of developers feel this way.

We use `Array.prototype.unshift()` to add an element to the beginning of an array. It's like `Array.prototype.push()`, except for the beginning of the array instead of the end.

Here's an example:

```
> let numberArray = [2,3,4];
> numberArray.unshift(1);
> numberArray;
[1,2,3,4]
```

On the other hand, `Array.prototype.shift()` *removes* the first element of the array:

```
> let numberArray = [2,3,4];
> numberArray.shift();
> numberArray;
[3,4]
```

Yes, the common sentiment out there is that this is backwards and confusing. When you *shift* something over, shouldn't that mean adding something to the beginning because we are shifting over what's already there? And when you *unshift* it, shouldn't that mean taking something away from the beginning?

Nope, it's the opposite. The best way to remember how `Array.prototype.shift()` and `Array.prototype.unshift()` work is that their effect seems counter-intuitive to how they are named. There is actually a good reason they are named this way — it's based on binary shifting. That's not something you need to look up, though, unless you are curious to learn about something completely tangential.

Also, it's worth noting — both of these methods are destructive. They change the receiver.

`Array.prototype.pop()`

Here's an easy one to remember! Don't want to *push* something onto the end of an array? *Pop* it off. `Array.prototype.pop()` removes the last element of an array:

```
> let arrayToPop = ["a", "b", "c", "d"];
> arrayToPop.pop();
"d"
> arrayToPop;
["a", "b", "c"]
```

A couple of things to note here. One, this method is destructive. Secondly, when we call this method, the return is the value of the element that was popped. Pay close attention here! Sometimes we'll want to do something with this value.

`Array.prototype.join()`

In this section, we'll work on some projects where we want to do things to different characters in a string. This means turning a string into an array. Once we are done doing all the things, we may want the final thing to once again be a string. That's where `Array.prototype.join()` comes in. It will take an array and turn it into a string. We can also pass in a separator as an argument. The examples below should make this clear:

```
> const epicodus = ["e","p","i","c"];
> const epic = epicodus.join();
> epic;
"e,p,i,c"
> const epicWithoutCommas = epicodus.join("");
> epicWithoutCommas;
"epic"
> const reallyEpic = epicodus.join("...");
> reallyEpic;
"e...p...i...c"
```

If we call `Array.prototype.join()` without arguments, the array will be turned into a string with commas as separators between each element from the array. That's usually not what we'll want. More commonly, we'll want to do away with the commas, which means passing in `""` as an argument. This just means the separator should be no spaces. If we passed in `" "`, the separator between each element would be a space. We can pass anything we want as a separator — especially if we want to be really epic and add `...` between each character.

Note that this is a non-destructive method — so we need to save the return value to a variable.

`Array.prototype.slice()`

We'll cover one more common array method. We can use `Array.prototype.slice()` to literally slice off parts of an array. It's kind of like `Array.prototype.shift()` and `Array.prototype.pop()` mixed together in one, but more flexible and powerful.

We can pass in one or two arguments. We have to slice from the beginning of the array — and we can optionally slice from the end.

The first argument denotes the index we should slice up to:

```
> const fruits = ["rambutan", "durian", "kiwi", "guava", "m
angosteen"];
> const slicedFruits = fruits.slice(2);
> slicedFruits;
["kiwi", "guava", "mangosteen"]
```

Here, we are slicing *everything up to but not including* an index of 2 — so everything before `"kiwi"`. Note that `Array.prototype.slice()` isn't destructive and we have to save its return (a sliced copy of the array) in a variable. That's probably for the best — too much slicing and dicing going on!

Now let's slice our fruits a different way — and also remove some fruits from the end of our array.

```
> const dicedFruits = fruits.slice(1,3);
["durian", "kiwi"]
```

We are slicing *up to* an index of 1 — so `"rambutan"` gets sliced off. Then we are slicing *everything past and including* an index of 3. So the second argument handles things a little differently than the first because the slicing starts *when* the index is reached, not *after*. Be careful or you might slice more fruits than your codebase can handle.

Other Array Methods

This is just the tip of the iceberg in terms of what we can do with arrays. There are a lot of other array methods — and later in this section we'll learn how to use looping to really enhance what we can do with arrays. We'll also learn some other powerful array methods throughout the remainder of the program.

To see a list of array methods, check the Mozilla array documentation (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array). Scroll down the page a bit and you'll see a list of methods in the left-hand pane. You don't need to memorize them — after all, that's what documentation is for. However, take some time to acquaint yourself with the methods available. All the methods we've covered in this lesson are on that list.

Previous (/introduction-to-programming/arrays-and-looping/bracket-notation)
Next (/introduction-to-programming/arrays-and-looping/comparing-and-cloning-arrays)

Lesson 7 of 50
Last updated March 24, 2023