Lesson Weekend

Introduction to Programming (/introduction-to-programming) / Arrays and Looping (/introduction-to-programming/arrays-and-looping) / Bracket Notation

Text Cheat sheet

Now that we know what an array is, let's learn how to work with them. In this lesson, we'll learn how to peek inside an array to look at a value. We'll also learn about the `length` property of arrays as well as a gotcha that can cause major bugs when we are working with arrays.

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

Looking At Single Values Inside an Array

Let's take a look at our `months` array again (the unmodified version — not the one we mutated!):

```
> const months = ["january", "february", "march", "april",
"may", "june", "july", "august", "september", "october", "n
ovember", "december"];
```

Let's say we want to grab the first month of the year from this array. We can do this with **bracket notation**. Let's use bracket notation to get the first value of `months` :

```
> months[0];
"january"
```

In the example above, `[0]` is the bracket notation. This states that we want to retrieve the element at the 0 index of the array. In computer programming, the **index** of an array is just its numerical position.

An array's index always starts at 0, which means the first element has an index of 0, *not* 1 as we might expect.

We can demonstrate with another example:

```
> months[1];
"february"
```

As we can see, `months` at an index of 1 actually returns the *second* element of the array. It's very important for people new to programming to understand this concept right away. While it's a simple concept, it can be a big gotcha, especially at first. We'll cover this gotcha in more detail in just a moment.

Getting the Last Element of an Array and OBOEs

What if we want to grab the last element of an array?

Well, all arrays come with a `length` property. Note that this is a **property** of an array, not a method. We can use the `length` property to find the last element of the `months` array. We'll also illustrate a big gotcha.

```
> months.length
12
> months[months.length];
undefined
```

In the example above, we can see that the `months.length` property is `12`. That's exactly what we'd expect. Remember that we can pass in a variable, expression, or property directly into the brackets `[]` when we use bracket notation, and that's exactly what we are doing with `[months.length]`.

Since the `months` array has a length of `12` and there are twelve months in a year, and we want to find the twelfth element in the array, the code `months[months.length]` makes sense. However, the return value is `undefined`. What gives?

Well, that's the gotcha. **An array's `length` property returns the number of elements inside of it.** In our case, the 12th element of the `months` array actually has an index of 11 because we always start at 0 when determining an element's index. If we use bracket notation to try to find an element that doesn't exist in the array, the return will be `undefined`. And as we're starting to learn, `undefined` is often the bane of developers when it comes to JavaScript debugging.

To correctly get the value of the last month, we have to subtract one from the length of the array:

```
> months[months.length - 1];
"december"
```

This will *always* be the case when we are returning the final element of an array. We can even write a very simple little formula to denote this. In the example below, `array` is the array we want to get the last element from:

```
const lastElementOfArray = array.length - 1;
```

This is a simple gotcha but it's also a big one. Failure to heed this rule leads to what is called an **off-by-one error** — sometimes known as an OBOE for short. You'd think it would be a simple error to remedy and yet OBOEs are exceedingly common in programming. In this case, an OBOE is exactly what it sounds like — returning an element from an array that's one element before or after the one you actually want to return.

So let's illustrate how big a deal these OBOEs can be. Let's say we have an application that sends reminders for New Year's Eve celebrations. We aren't just sending reminders for one party — we're in charge of reminders for hundreds or even thousands of celebrations ranging from galas at non-profits to corporate events. We're not a huge company but we are making do and we have a small team of dedicated developers.

Now let's say we have an itsy-bitsy little OBOE in our application — but that OBOE comes when we are calculating the month that the date reminders should be sent out. They *should* be sent in late December — but an OBOE in one direction means they are sent in November instead! It would be a big embarrassment for the company — and we might lose some clients. But now imagine an OBOE in the other direction — our application returns `undefined` instead of a month (because there is no thirteenth month at the 12th position of the array). Now the reminders don't get sent out at all. People don't get their reminders, clients are angry, and most decide they will never use the company's product again.

As we can see, OBOEs can really wreak havoc on an application. Likely our company would have error handling that would catch the OBOE, but the point remains: be very careful when working with the index of elements in arrays and keep the possibility of OBOEs in mind.

Summary

To summarize, we can use bracket notation to find any element in an array. Here are a few more examples using the following array:

```
> const numberArray = [1,4,9,3,7,18,63];
```

Note that the `numberArray` has 7 elements:

```
> numberArray.length;
7
```

We can find the *third* element of the array with the following:

```
> numberArray[2];
9
```

We can find the *second from the last* element of the array with help from the `length` property:

```
> numberArray[numberArray.length - 2];
18
```

We can even pass a variable in if we wanted:

```
> const arrayIndex = 4;
> numberArray[arrayIndex];
7
```

Finally, if we ask for an index that doesn't exist in an array, the return will be `undefined`.

```
> numberArray[100];
undefined
> numberArray[-1];
undefined
> numberArray["cat"];
undefined
> numberArray["1"];
undefined
```

The first example above makes sense — there's no number at the 100th index of an array that only contains 7 elements. Nor can we use negative numbers to find an element in an array.

The other examples may seem a little stranger. You might think that JavaScript would get mad if you pass a string into the square brackets, but nope. Not surprisingly, there's no array element at the `"cat"` index. And `"1"` is a string, not a number, so there's no element at that index, either. It would be nice if JavaScript would throw a helpful error in the last two cases above, but that's just not how JS does things. There are reasons we can pass strings into bracket notation (they have to do with using bracket notation to retrieve properties from objects), but we won't cover that until the next section.

In this lesson, we covered some simple concepts in greater depth because they are so important to understanding arrays. It's essential to be able to peek into an array and grab a value, whether that's working from the beginning or the end of an array. It's equally important to be aware of OBOEs and how devastating they can be to a codebase. Finally, we'll be using the `length` property of arrays a lot — not just when we are grabbing elements from arrays but also when we begin looping later in this section.

Now we're ready to start learning other array methods. We'll do that in the next lesson!

Previous (/introduction-to-programming/arrays-and-looping/introduction-to-arrays)
Next (/introduction-to-programming/arrays-and-looping/array-methods)

Lesson 6 of 50
Last updated March 24, 2023

disable dark mode