Lesson | Wednesday

Introduction to Programming (/introduction-to-programming) / Arrays and Looping (/introduction-to-programming/arrays-and-looping) / When to use for

Text | Cheat sheet

In this lesson, we'll compare use cases for `Array.prototype.forEach()` and `for` loops. As we'll learn, we'll favor using a `for` loop when we're not iterating over an array, and if we need to break out of an array and not loop over every value in it. As we move through the examples in this lesson, put them into the DevTools console to try them out.

When to Use `for` Loops

In general, we should use `Array.prototype.forEach()` when we want to loop through *every* element in an array. If we are using a loop that isn't iterating through an array, we can't use `Array.prototype.forEach()` for that loop — that's pretty obvious.

However, sometimes we *can't* use `Array.prototype.forEach()` when we are looping through an array. Specifically, if we want to *stop* the loop, we can't use `Array.prototype.forEach()`. Looping through every element in an array is a requirement of `Array.prototype.forEach()`.

Let's look at an example when this might matter. Let's say we have a series of arrays that hold sequences of DNA. These sequences can potentially be very long — and we want to stop searching an array as soon as we find a pattern. We'll call that pattern "A" (to represent a specific DNA sequence). We'll also add a few other letters to represent other DNA sequences. A biologist won't be impressed by our approximation... but we are programmers and this is a programming example!

Here's our sample "sequence":

```
const dnaSequence = ["X", "A", "Y", "M", "D"];
```

In this example, `"A"` is the second element in this array. We could use an `Array.prototype.forEach()` loop to flag the sequence if it includes `"A"`. Here's the code:

```
> let dnaFlag = false;
> const dnaSequence = ["X", "A", "Y", "M", "D"];
> dnaSequence.forEach(function(element) {
  if (element === "A") {
    dnaFlag = true;
  };
  console.log("Looped!")
});
> dnaFlag;
Looped!
Looped!
Looped!
Looped!
Looped!
true
```

In the code above, when an iteration through the loop finds a match with `"A"`, it will switch the `dnaFlag` boolean to `true`. Using boolean flags in this manner is very common in coding.

Note, however, that the *second* character in our array matched the letter `"A"`. What if our array were a million elements long? (DNA sequences can be very complex.) Our loop needs to loop through *all* million elements — even though we really only need to do two iterations of the loop to switch `dnaFlag` to `true`. This would be horribly inefficient. We've added a `console.log("Looped!")` to show how many times this loop runs.

In order to break out of loop, we need to use the `break;` keyword. This keyword does exactly what it sounds like: stops a loop.

What happens if we try to add a `break;` statement to the loop above?

```
> let dnaFlag = false;
> const dnaSequence = ["X", "A", "Y", "M", "D"];
> dnaSequence.forEach(function(element) {
  if (element === "A") {
    dnaFlag = true;
    break;
  };
});
> dnaFlag;
Uncaught SyntaxError: Illegal break statement
```

We'll get the following error: `Uncaught SyntaxError: Illegal break statement`. In short, we can't break out of this kind of loop.

However, we can use a break statement with a `for` loop. Try the following in the console:

```
> let dnaFlag = false;
> const dnaSequence = ["X", "A", "Y", "M", "D"];
> for (let i = 0; i < dnaSequence.length; i +=1) {
  if (dnaSequence[i] === "A") {
    dnaFlag = true;
    break;
  };
  console.log("Looped!");
}
> dnaFlag;
Looped!
true
```

We've added a `console.log("Looped!")` to this code, too — and you'll see it only triggers once. The second time through the loop, the condition is met and the loop breaks due to the `break;` keyword.

There's something that's not so great about the code above. As of now, the `dnaFlag` variable is globally scoped. In an ideal world, we'd wrap this all in a function. Let's do this now, not just because it's better code but also so we can see what happens when we try to `return` out of both a `for` loop and an `Array.prototype.forEach()` loop.

We'll start by wrapping the `for` loop in a function called `dnaPatternDetector`:

```
> function dnaPatternDetector(dnaSequence, pattern) {
  for (let i = 0; i < dnaSequence.length; i +=1) {
    if (dnaSequence[i] === pattern) {
      return true;
    };
    console.log("Looped!");
  }
  return false;
}
```

We've done a bunch of things to refactor this code. Our function takes two parameters. The `dnaSequence` is an array of characters. The pattern is the character we are looking for (we are looking for `"A"`).

We actually don't need to use the `dnaFlag` anymore. If the pattern is met (`dnaSequence[i] === pattern`), then the loop will stop running and our function will return `true`. If the pattern is never met, our loop will complete and the final part of our function will be reached: `return false;`.

Let's try the following out in the DevTools console. First, some variables — a `sequence` variable that includes an array and two variables for patterns: `pattern1` and `pattern2`:

```
> const sequence = ["X", "A", "Y", "M", "D"];
> const pattern1 = "A";
> const pattern2 = "Z";
```

Next, we'll also need to add our `dnaPatternDetector()` function in the console as well.

Finally, try this out:

```
> dnaPatternDetector(sequence, pattern1);
Looped!
true
```

We'll see that the function returns `true` — exactly what we'd expect because `sequence` includes `"A"`. We also see that it only hits the `console.log("Looped!")` statement once. `return` has the same effect as the `break;` statement here — however, it doesn't just break us out of the loop, it returns from the function.

Now try out the second pattern, which is a character that's *not* included in `sequence`:

```
> dnaPatternDetector(sequence, pattern2);
Looped!
Looped!
Looped!
Looped!
Looped!
false
```

Our loop iterates through every character in the array and never finds the character `"Z"`. The loop completes (so "Looped!" is printed to the screen five times). This is the only way the final part of the code can be reached: `return false`. The function returns false.

A function that actually checks DNA sequences would probably be quite a bit more complicated than this example. However, this does show a potential use case for when we'd want to break out of a loop — either with a `break;` or a `return` statement. This same function wouldn't work as intended with an `Array.prototype.forEach()` loop — if we try to put a `return` statement inside the loop, we'll get an error. The rules for

`Array.prototype.forEach()` loops are strict — we *must* iterate through every element in the array — we can't break or return from it.

To summarize, you should favor `Array.prototype.forEach()` loops over `for` loops except in the following cases:

- You're not looping through an array. `Array.prototype.forEach()` can only be used with arrays.
- You need to break out of the loop — either with `break;` or a `return` statement.

We can add one more case just for the sake of practice:

- You should favor `for` loops to loop through every element in an array until you understand exactly how they work. They are a cornerstone of JavaScript and it's essential to understand them even if you'll mostly use other types of loops in the future.

In the next lesson, we'll make changes to our text analyzer application. Both will be situations where a `for` loop will serve our purposes better.

You can use any kind of loop you want for this section's independent project. The project prompt can be solved with `Array.prototype.forEach()` but you are welcome to use `for` (or another kind of loop you learn later in this section) as well.

Previous (/introduction-to-programming/arrays-and-looping/practice-looping-with-for)
Next (/introduction-to-programming/arrays-and-looping/for-loops-with-text-analyzer)

(http://www.epicodus.com)