Lesson Wednesday

Introduction to Programming (/introduction-to-programming) / Arrays and Looping (/introduction-to-programming/arrays-and-looping) / for Loops with Text Analyzer

Text Cheat sheet

Over the last few lessons, we've learned how and when to use a `for` loop. Now let's look at a couple of situations where a `for` loop would be better than `Array.prototype.forEach()` in our text analyzer application.

We will not use TDD for these examples. That's because we've already covered TDD in depth and the goal here is to focus on `for` loops. However, you are expected to continue using TDD in your own projects as well as on this section's independent project. We also won't incorporate the function into the UI. You can do that on your own.

Updating Text Analyzer To Use `for` Loops

Adding `firstInstanceOfWord()`

We'll start with adding new functionality to our application. Specifically, we'll add a function that finds the position of the **first** instance of a word in a passage of text. This is a good use case for a `for` loop because we can break out of the loop as soon as we find the first instance of the word.

Here's how we might write this function:

```
> function firstInstanceOfWord(word, text) {
  const textArray = text.split(" ");
  for (let i = 0; i < textArray.length; i++) {
    console.log(i);
    if (word === textArray[i]) {
      return i;
    }
  }
  return -1;
}
```

Add this code to the DevTools console and then call it with the following:

```
> firstInstanceOfWord("hi", "hey hi hey hey hey hey hey hey
what");
0
1
1
```

You'll see that the `console.log()` only logs 0 and 1 before breaking out of the loop and returning `1`, the index location of `"hi"`. This is in contrast to how we'd have to solve this same problem with `Array.prototype.forEach()`:

```
> function firstInstanceOfWord(word, text) {
  const textArray = text.split(" ");
  let position = -1;
  textArray.forEach(function(element, index) {
    console.log(index);
    if ((word === element) && (position === -1)) {
      position = index;
    }
  });
  return position;
}
```

If we put this version of the function in the DevTools console, it will return the correct answer as well but the function will loop through every element in the array even though it finds what it's looking for in the second element.

Also, there are some slightly wonky things we have to do. `let position = -1` is like a flag that's set to `false`. You might wonder why both functions return `-1` if they don't return a matching word. Why not just return `false`? Well, it's common for JavaScript methods to return `-1` when there's not a match. We should be returning a number or a boolean, not a number sometimes and a boolean other times. This makes our function more consistent.

Note our conditional, too: `if ((word === element) && (position === -1))`. This is because we only want `position` to be updated *once*: the first time we find a match. When that happens, `position` will no longer be `-1`. If we didn't do this, the function would always return the *last* match, not the first.

So in addition to needing to continuing looping even after finding a match, there is some other code we have to add to solve this problem with `Array.prototype.forEach()`. That is not ideal. A `for` loop does a better job in this situation.

Of course, the function above doesn't account for case sensitivity, punctuation, and so on. You can add all of this functionality in your own application — and include tests as well.

Updating `isEmpty()` To Use Built-In JS `arguments` Object

Now let's look at one other use case for a `for` loop in our application. This is not something you are required to know for the independent project but it's a cool little thing we can use to expand the capacity of our `isEmpty()` function.

Remember our little utility method?

scripts.js

```
// Utility Logic

function isEmpty(testString) {
  return (testString.trim().length === 0);
}
```

Well, currently it only checks one input. This means anytime we need to check two separate strings, we need to call `isEmpty()` twice, like in the `boldPassage()` function:

scripts.js

```
function boldPassage(word, text) {
  if (isEmpty(word) || isEmpty(text)) {
    return null;
  }
  ...
  return p;
}
```

While this is a good and working solution, we can improve our `isEmpty()` function by making it work with **any** number of arguments. We'll do this by using JavaScript's arguments object (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments).

Check this out:

```
> function isEmpty() {
  for (let i=0; i < arguments.length; i++) {
    console.log(arguments[i]);
    if (arguments[i].trim().length === 0) {
      return true;
    }
  }
  return false;
}
```

So what exactly is happening here? This function has *no* parameters. So where is `arguments` coming from? Well, JavaScript makes an arguments object (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments) available inside functions that includes the values of *all* arguments that are passed into the function. **You are *not* expected to ever use this on an independent project but it's a handy thing to know about.**

Also, it illustrates another great use for `for` loops. That's because **`arguments` is an "array-like" object, just like the `NodeList` and `HTMLCollection` objects we learned about at the start of this section.** We can access each individual argument inside the `arguments` object with bracket notation. However, it's not actually an array so we can't use `Array.prototype.forEach()` to loop over the `arguments` object, unless we convert it into an array.

So in the `isEmpty()` function, we use a `for` loop. In each iteration through the loop, we check the value of `arguments[i]`. In this way, we can check out `arguments[0]`, `arguments[1]`, and so on for as many arguments as we pass into `isEmpty()` when we call it.

Let's try it out in the console. Copy the update `isEmpty()` function into the console and then input the following:

```
> isEmpty("hi", "bye", "", "bonjour!");
hi
bye

true
```

As we can see from the return values, each argument is being logged, including the empty string, and then the function returns `true`. It never logs `bonjour` because the function detects an empty string as a parameter, which means not all of the parameters qualify as inputted words.

With this update to `isEmpty()`, we can use this utility function *regardless of the number of arguments* passed into it. Our utility function just got more useful. Now we can update how we call `isEmpty()` in our `boldPassage()` function:

scripts.js

```
// Utility Logic

function isEmpty() {
  for (let i=0; i < arguments.length; i++) {
    if (arguments[i].trim().length === 0) {
      return true;
    }
  }
  return false;
}

// Business Logic

...

// UI Logic

function boldPassage(word, text) {
  if (isEmpty(word, text)) {
    return null;
  }
  ...
  return p;
}

...
```

While `Array.prototype.forEach()` will usually get the job done — and should generally be favored whenever we need to loop through each item in an array — there are plenty of use cases where `for` loops are the right tool, including both of the use cases above in our text analyzer application.

Previous (/introduction-to-programming/arrays-and-looping/when-to-use-for)
Next (/introduction-to-programming/arrays-and-looping/practice-pig-latin)

Lesson 39 of 50
Last updated February 28, 2023

disable dark mode

Epicodus (http://www.epicodus.com)