Lesson Tuesday

# Introduction to Programming (/introduction-to-programming) / Arrays and Looping (/introduction-to-programming/arrays-and-looping) / Text Analyzer with TDD: numberOfOccurrencesInText()

Text Cheat sheet

In this lesson, we'll complete the tests and functionality for a new function in the Text Analyzer project, called `numberOfOccurrencesInText()`. The goal of this lesson is to demonstrate the TDD process, including all of the decisions we need to make along the way.

**You should code along with this and following lessons that build out the Text Analyzer project. In the upcoming practice lesson, you'll be tasked with adding more functionality to this project.**

## Writing and Testing a Second Function for Text Analyzer

Let's get a little more practice by writing a second function. This one will also use a loop as well. The purpose of this function will be to determine how many times a specific word occurs in a passage. We'll call this function... um... `wordCounter()`? No. We've got to be clear with our code and that's already taken. We'll call it `numberOfOccurrencesInText()`. The name is a bit lengthy but this function states exactly what it does which will help us communicate with other developers.

Let's say we have the following passage of text:

```
"red blue red red green red"
```

If we ask our function how many times the color *red* occurs, it should correctly return 4.

However, that is *not* our first test. We can start even smaller than that.

## Our First Test

How small can we go? We can start with how many times a word occurs in an empty string, which should be 0 no matter what. That's probably as small as we can go.

We'll want to start a new group of tests for this function, which means a new **Describe** block. We can add this test below the other tests in our README.

```
Describe: numberOfOccurrencesInText()

Test: "It should return 0 occurrences of a word for an empt
y string."
Code:
const text = "";
const word = "red";
numberOfOccurrencesInText(word, text);
Expected Output: 0
```

Once again, this basic test can really help us get started. Let's take a look.

**js/scripts.js**

```
// Business Logic

// wordCounter() function omitted for brevity.

function numberOfOccurrencesInText(word, text) {
  return 0;
}
```

We add the `numberOfOccurrencesInText()` function beneath our `wordCounter()` function. See how nicely our business logic is coming together because we aren't thinking about the UI? When we start working on our user interface logic, it will be much easier to keep things separate.

While our function is basic so far, it allows us to establish a couple of key things. First, this function needs two parameters, one for the `word` we want to find and one for the `text` itself. Secondly, just like with our `wordCounter()` function, it will return a number.

## Our Second Test

Next, let's see what happens when we are searching text that is just one word.

```
Test: "It should return 1 occurrence of a word when the wor
d and the text are the same."
Code:
const text = "red";
const word = "red";
numberOfOccurrencesInText(word, text);
Expected Output: 1
```

Let's update our function. Once again, we are aiming to keep it as simple as possible. It's okay if it looks nothing like the final product yet. We are just taking baby steps.

**js/scripts.js**

```
function numberOfOccurrencesInText(word, text) {
  if (word === text) {
    return 1;
  }
  return 0;
}
```

We add a simple conditional. If the word equals the text, we should return 1. Otherwise, we should return 0. Very simple. Both tests will pass now.

## Our Third Test

Are we ready to move onto multiple words? Well, we should verify that it doesn't return a match if the word and the text aren't the same first.

Here's the test:

```
Test: "It should return 0 occurrences of a word when the wo
rd and the text are different."
Code:
const text = "red";
const word = "blue";
numberOfOccurrencesInText(word, text);
Expected Output: 0
```

This test will pass already so you might wonder what the point is. Well, first of all, it's always good to verify, not assume. You don't ever want to tell the team lead or your boss that you assumed something would work when everything goes terribly awry. Also, with automated testing, we might find later in the process that something breaks this specific test while all of our other tests pass correctly. Then we could more easily go back and fix the issue.

## Our Fourth Test

Let's move onto multiple words.

```
Test: "It should return the number of occurrences of a wor
d."
Code:
const text = "red blue red red red green";
const word = "red";
numberOfOccurrencesInText(word, text);
Expected Output: 4
```

You might be wondering why we are moving up to so many words and occurrences already. Why not just move up to two words first? Well, they should work exactly the same — and we are less likely to get a false positive. On the other hand, our function already returns 1 sometimes — if we just have two words and one of them is red, well, our code may return the right answer even if it's broken — just as a broken clock is right twice a day.

Let's update our code to get our new test passing:

**js/scripts.js**

```
function numberOfOccurrencesInText(word, text) {
  const textArray = text.split(" ");
  let wordCount = 0;
  textArray.forEach(function(element) {
    if (word === element) {
      wordCount++
    }
  });
  return wordCount;
}
```

This doesn't look very different from our previous code — but we actually modified the conditional from our previous test to use within our loop. Instead of having to write all the code at once, we took the time to get a sense of what our parameters are and return argument should look like, and we also got a good start on our conditional.

Once again, we split the text passage into an array and create a `wordCount` that starts at 0. We loop through this array, and if the `word` we've passed into our function is equal to the `element` in `textArray`, we've found an instance of the word and we can increment `wordCount` by one. Finally, we return `wordCount`.

It works! Yay!

## Our Fifth Test

But what about...

```
"Red RED red"
```

We need to account for upper and lowercase. `"Red"` and `"red"` are still the same word — but our function will not recognize this. Once again, let's start with a test.

```
Test: "It should return a word match regardless of case."
Code:
const text = "red RED Red green Green GREEN";
const word = "Red";
numberOfOccurrencesInText(word, text);
Expected Output: 3
```

Note that our test will be a bit more thorough because we are also changing the case of the `word` variable. It should be evident here that we need to do something that makes both the `word` and all instances of that word in the `text` variable consistent, such as lower-casing them. If the words that are being compared are different cases, our function won't see them as a match.

Try getting the test passing on your own first. The passing code is below:

**js/scripts.js**

```
function numberOfOccurrencesInText(word, text) {
  const textArray = text.split(" ");
  let wordCount = 0;
  textArray.forEach(function(element) {
    if (word.toLowerCase() === element.toLowerCase()) {
      wordCount++;
    }
  });
  return wordCount;
}
```

As we can see, we just need to call `String.prototype.toLowerCase()` on both `word` and `element` to get the test passing.

## Important Considerations

This is one of those tests where we really do need to think carefully about what we are testing and how we can make sure that our function works as expected. It would be easy to write a test that just lowercases the text and doesn't take account of the fact that a user might type in `"RED"` instead of `"red"` for the `word` parameter. Fortunately, our test accounts for both the case of the `word` parameter *and* the case of each `element` in the `text` array.

It may also be tempting to just lowercase a user's input in the user interface section of the code — similarly to how we've used `parseInt()` to make sure that a number input on a form is converted from a string to a number. However, this wouldn't be a good separation of logic. Remember, it's our function's job to correctly analyze any strings it receives. If we did that in the UI instead, it would be harder to test, harder to track, and more prone to bugs. Our function would also be less resilient and reusable.

## Our Sixth Test

Let's move onto our next test. Can you think of anything else that still needs to be tested? Are there any other situations where our function won't correctly compute a matching string when it should?

What about this string?

```
"Red! Red. I like red, don't you?"
```

If we split this string by spaces, we'll get the following array:

```
["Red!", "Red.", "I", "like", "red,", "don't", "you?"]
```

Well, `"red"` should match `"red."` Currently, though, it won't. So let's write a test.

```
Test: "It should return a word match regardless of punctuat
ion."
Code:
const text = "Red! Red. I like red, green, and yellow.";
const word = "Red";
numberOfOccurrencesInText(word, text);
Expected Output: 3
```

Now let's get our test passing. There are several ways in which we can solve this problem.

- One way is to use the method `String.prototype.includes()` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/includes), which checks to see if a string *includes* another string or character. We're going to use this approach. `String.prototype.includes()` is a very handy string method and one which you'll likely use multiple times throughout this section.
- The other approach is to use a regular expression. We are going to cover this approach in a *further exploration* lesson on regular expressions. *Further exploration* means it's *not* required to learn about regular expressions and you won't need to use them in this section or on the independent project, though you can experiment with them if you like.

Let's solve the problem using `String.prototype.includes()`. First, let's see what this method actually does.

`String.prototype.includes()` returns a boolean. If a string contains another string, the method will return `true`. For instance, the string `"epicodus"` contains the string `"epic"`. If the string doesn't include the substring, the method will return `false`. We can do something like this:

```
function includesRarestLetter(word) {
  if (word.toLowerCase().includes("q")) {
    return true;
  }
  return false;
}
```

Q is the rarest letter in the English alphabet and this function checks whether a word contains the letter, returning `true` if it does and false otherwise.

We can also use `String.prototype.includes()` with longer strings as well — such as checking whether a substring includes `"red"` . Let's try this in the DevTools console:

```
> "red! red. red?".includes("red");
true
```

Let's update our function to get our newest test passing:

```
function numberOfOccurrencesInText(word, text) {
  const textArray = text.split(" ");
  let wordCount = 0;
  textArray.forEach(function(element) {
    if (element.toLowerCase().includes(word.toLowerCase()))
{
      wordCount++;
    }
  });
  return wordCount;
}
```

We've updated our conditional to check if the following is true:

```
element.toLowerCase().includes(word.toLowerCase())
```

So if an `element` in the `text` array (such as `"red."`) includes the word we are searching for (`"red"`), `wordCount` will be incremented — and our test will pass!

`String.prototype.includes()` is a *very* helpful method. There is a problem, though:

```
> "redo".includes("red");
true
```

Yes, the word `"redo"` *contains* the word `"red"` — but it's *not* an occurrence of the word. We aren't going to worry about this issue, though you are welcome to refactor the application on your own to fix this with an additional test.

So our `numberOfOccurrencesInText()` function isn't perfect but that's okay. Once again, the main purpose here is to learn about Test-Driven Development: how it works, how to apply it, and how to write pseudocode tests to gradually build up robust functions and solve problems.

For the next several sections, you will use this Test-Driven Development approach with pseudocode tests. As we've mentioned before, in the Test Driven Development and Environments with JavaScript (https://www.learnhowtoprogram.com/intermediate-javascript/test-driven-development-and-environments-with-javascript) course section, we will start using Jest for our tests.

**To view all of the tests that we wrote for both `wordCounter()` and `numberOfOccurrencesInText()`, check out the cheat sheet.**

Previous (/introduction-to-programming/arrays-and-looping/text-analyzer-with-tdd-wordcounter)
Next (/introduction-to-programming/arrays-and-looping/practice-extending-text-analyzer-business-logic-with-tdd)

Lesson 26 of 50
Last updated February 28, 2023

disable dark mode

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