Lesson Tuesday

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

Text Cheat sheet

In the last lesson, we wrote a very basic function for counting words in a paragraph. It doesn't work very well. It doesn't care about the difference between numbers and words in a string. It returns `1` even when we pass in an empty string. So let's keep working on this function and make it better. We're ready to actually start building our application.

In this lesson, we'll complete the tests and functionality for the Text Analyzer's `wordCounter()` function. Then, in the next lesson, we'll create a new function for our Text Analyzer project using TDD. The goal of these lessons 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 the 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.**

## Setting Up the "Text Analyzer" Example Project

Create a directory called `text-analyzer` with two files: a `README.md` and a directory called `js` that includes a `scripts.js` file. Over the next several lessons, you'll follow along with building the `text-analyzer` application.

If you're wondering why there is no `index.html` file or `css` directory, it's because we aren't ready for them yet. **Test-driven development is all about building your business logic first.** It doesn't necessarily mean that *all* business logic has to be created before a user interface. However, if we start mingling business logic and user interface right now, we'll probably run into problems. We'll likely have issues with separation of logic. We might also end up with bugs in our code that are hard to track. For instance, if something isn't working, we might not be able to tell whether the error is in our business logic or user interface logic because everything is mixed up and we haven't tested it.

Here's our `scripts.js` file so far based on the code we covered in the last lesson:

**js/scripts.js**

```
// Business Logic

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

A quick reminder: as we stated in the last lesson, we are going back to using a loop because we need the practice. We've also added the comment `// Business Logic` to make it easier to see how our business and UI logic are separated because they are in the same file. We're just using this comment for clarity and organization — it's not something you'll see in real-world code bases. In Intermediate JavaScript, we'll learn to separate our code into different files and we won't need these comments anymore.

We mentioned this in the last lesson, but you may have missed it: all of our pseudocode tests should go in our README. Here's what the tests in our `README.md` should look like so far:

```
Describe: wordCounter()

Test: "It should return 1 if a passage has just one word."
Code:
const text = "hello";
wordCounter(text);
Expected Output: 1

Test: "It should return 2 if a passage has two words."
Code:
const text = "hello there";
wordCounter(text);
Expected Output: 2
```

Note that this *isn't* a complete README. It's just the two pseudocode tests we have created so far for the `wordCounter()` function. You are still responsible for adding any other README information for your projects.

## Our Third `wordCounter()` Test

Let's deal with the next simplest behavior. There are actually several different behaviors we could tackle next, so don't get too caught up on whether one behavior is simpler to implement than another if the distinction is not obvious. There is no preset route to building an application with TDD. Just do your best to implement one small behavior at a time.

We'll start by dealing with the fact our function will return `1` for an empty string. Here's the pseudocode test:

```
Test: "It should return 0 for an empty string."
Code: wordCounter("");
Expected Output: 0
```

Note that this test still belongs to the group of tests we've written so far, which means it doesn't have a separate line for **Describe**. Also, we've put the code inline because there's just one line of code. How you format your own pseudocode tests is up to you. Just make sure they are easy to read.

Now let's update the code. We encourage you to write the solution yourself.

**js/scripts.js**

```
// Business Logic

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

All we need to do is write a conditional that checks if the length of `text` is equal to `0`. If it is, the function will return `0`. If we try `wordCounter("");` in the console, it will return `0` as expected.

## Our Fourth `wordCounter()` Test

So now we're ready to move on. Or are we? Think carefully. Have we solved the empty string problem fully yet?

Try this:

```
> wordCounter("             ");
```

According to our function, that's 13 words. Not good. We could update our empty string test or write a new one. To be thorough and practice, we'll write a new test. Again, we are taking baby steps here.

```
Test: "It should return 0 for a string that is only space
s."
Code: wordCounter("             ");
Expected Output: 0
```

It's a quick fix:

**js/scripts.js**

```
// Business Logic

function wordCounter(text) {
  if (text.trim().length === 0) {
    return 0;
  }
  let wordCount = 0;
  const textArray = text.split(" ");
  textArray.forEach(function(word) {
    wordCount++;
  });
  return wordCount;
}
```

We can use `String.prototype.trim()` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/trim) to trim all whitespace from both ends of a string. Since the string is all whitespace, that will reduce it to `""`, which has a length of `0`.

Another test passing! These little details can lead to big bugs down the road if we don't think about them early on.

## Our Fifth `wordCounter()` Test

Next, let's think about words. What is a word exactly? We aren't going to enforce whether something is legally a word or not. And nor would Google Docs. If we were to write a fantasy novel with Xoeo and Myxtmidia as the main characters, Google Docs is more than happy to call those words. Google Docs also counts numbers as words but we are going to be more precise than that. A spelled number (`"seven"`) is a word but the number `7` is a number so we won't add it in the word count. So let's get started with a test.

```
Test: "It should not count numbers as words."
Code: wordCounter("hi there 77 19");
Expected Output: 2
```

In our test, we mix together words and numbers. Our function should properly count the words but ignore the numbers in the count. If we actually test `wordCounter("hi there 77 19");` in the console right now, it will return 4. That's not what we want. However, we will acknowledge that we could have characters named Epsilon72 and Eri9er in our upcoming science fiction novel. (We don't know how to pronounce Eri9er, but it's the future and they've figured that kind of thing out.)

So let's update the function to get this test passing.

Once again, see if you can do this on your own.

**js/scripts.js**

```
// Business Logic

function wordCounter(text) {
  if (text.trim().length === 0) {
    return 0;
  }
  let wordCount = 0;
  const textArray = text.split(" ");
  textArray.forEach(function(element) {
    if (!Number(element)) {
      wordCount++;
    }
  });
  return wordCount;
}
```

The change above will get the result we want. All we have to do is add a conditional. The built-in JavaScript `Number()` function (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number#description) either returns a number or `NaN`. `Number("16")` will return 16 while `Number("hi")` returns `NaN`. So if something's not a number (`NaN`), we will increment our `wordCount`. If it is a number, we won't increment it.

Note also that we changed the callback function's parameter to `element` because it might not always be a word. It's a good practice to rename variable names if you realize they more accurately communicate your code.

Once again, this was a pretty small change. It's much easier to do this incrementally.

There's another thing we haven't thought about: punctuation. Dealing with punctuation is easier to handle with a regular expression. However, we haven't learned about regular expressions yet and they are a further exploration topic, so won't worry about them right now. Even though we could do more to make this function robust, for the purposes of demonstrating how to use TDD practices, this is robust enough. Let's move on to the next function!

**To view all of the tests we wrote for the `wordCounter()` function, view the cheat sheet.**

Previous (/introduction-to-programming/arrays-and-looping/overview-of-test-driven-development-tdd-with-text-analyzer)
Next (/introduction-to-programming/arrays-and-looping/text-analyzer-with-tdd-numberofoccurrencesintext)

Lesson 25 of 50
Last updated February 28, 2023

disable dark mode

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