Lesson Tuesday

Introduction to Programming (/introduction-to-programming) / Arrays and Looping (/introduction-to-programming/arrays-and-looping) / Overview of Test-Driven Development (TDD) with Text Analyzer

Text Cheat sheet

As our projects get more complex, we need to break our code down into a series of smaller, more manageable programming tasks. It's absolutely essential we do this. Otherwise, it's easy to get overwhelmed if we try to take on too much at once. When we are working on complex business logic, we can use a development technique called **Test-Driven Development**, or **TDD**, to break our logic down into smaller problems that are easier to solve and reason about.

In this lesson, we'll give an overview of what Test-Driven Development is and how we'll use it, covering:

- Test syntax: comparing tests written with Jest's syntax and pseudocode syntax. (Until we get to automated testing, we'll be writing pseudocode tests and manually testing them.)
- The TDD process by writing two tests for our Text Analyzer's `wordCounter()` function.
- How to figure out what to test for, and how to test and write code incrementally.
- Opportunities for refactoring.

- The TDD process as far as actually testing our code in the DevTools console:
    - first writing the test,
    - then writing the code that will make your test pass,
    - and then running the test's code in the DevTools console to manually verify that your test passes (the actual output matches what you expected), which ultimately means that the code for your function is correct.

In the two lessons that follow, we'll actually build the Text Analyzer application (start with the business logic), demonstrating how to use TDD. So for now, just read through this lesson with the goal of understanding.

Breaking Down Problems with TDD

In TDD, we break our code down into the smallest pieces of functionality we can. Next, we write a test for a piece of functionality *before* we add that functionality to our code. Finally, we add the code to make that test pass.

These tests are also known as **specs**. For now, we are going to manually test all of our specs. Then, when we get to Test-Driven Development and Environments with JavaScript (https://www.learnhowtoprogram.com/intermediate-javascript/test-driven-development-and-environments-with-javascript), we'll start using software that automates the testing process. We'll also dive more deeply into the testing process by learning the Red Green Refactor workflow (https://www.learnhowtoprogram.com/intermediate-javascript/test-driven-development-and-environments-with-javascript/red-green-refactor-workflow) when we test with Jest.

However, we aren't quite there yet. In the past, students have found automated testing overwhelming when we introduce it too early in the program. For that reason, we wait until Intermediate JavaScript to cover automated testing. However, we believe that the thought

process behind Test-Driven Development is absolutely essential to breaking down tough problems into more manageable solutions. That's why we'll start writing tests in pseudocode now. We'll put all of our pseudocode tests in our project's README. Instead of using automated tests, we'll test our code in the console until it passes.

Also note that we'll only be writing tests for our business logic. While there is software for testing User Interface logic, we won't learn how to do that in this program.

Test Examples & Our First Test

First, here's an example of **a test written in JavaScript using Jest**, a testing framework we'll learn about in Intermediate JavaScript. By the way, you don't need to write any code yet — just read along for now. We'll start building the application in the next lesson.

```
describe('word counter', () => {
  test('should return 1 if a passage has just one word', ()
=> {
    const text = "hello";
    expect(wordCounter(text)).toEqual(1);
  });
});
```

The test above takes a word and then tests whether a function named `wordCounter()` correctly returns the correct word count. This will actually be our first test for our text analyzer application, too.

Instead of using Jest syntax, though, we will write our tests in pseudocode. Here's an example of **the same test written in pseudocode**:

```
Describe: wordCounter()

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

Test Syntax

We use specific syntax in our tests to communicate our testing goals — what we are testing, what outcome we're expecting, and how we are testing it. The tests we write in pseudocode are borrowing the syntax from Jest, the JavaScript testing framework we will use to write automated tests in the next course, Intermediate JavaScript. Using our first pseudocode test (shown above), let's explore the syntax "describe", "test", "code", and "expected output".

- **Describe** is a common term in testing. It's used to organize tests. For example, we might make a little application with three different functions, all of which need several tests. Here, we'll use one describe statement for each function, and group all of the tests for that function below the describe statement. In our example, we're describing the `wordCounter()` function.
- **Test** breaks down what the test is doing in plain English. In the example above, we see that the test `'It should return 1 if a passage has just one word'` is actually not code for the machine. Instead, it communicates to other developers (and ourselves) what the test is supposed to do.
- **Code** is where we'll put any code we need to run in order to check that our test is working. We don't actually need to have a separate variable called `text`. We could also just pass `"hello"` directly into the function. We split it out here just to show that you can add multiple lines of code if needed. Also, we don't

actually define the `wordCounter()` function here. That code goes in our actual scripts. Instead we just call on the code we want to test in the "code" section of our pseudocode test.

- **Expected Output** is what we expect our test result to be. This usually means seeing if a function we've written returns what we expect. In this case, we expect the output of `wordCounter(text);` to equal `1`.

Always Test Small Pieces of Functionality, Incrementally

You might be wondering why we are writing such a simple test. It's so basic: input a single word and have the function return `1`? We hardly have to do anything to pass the test. We could just do the following to get our test passing manually in the console:

```
function wordCounter(word) {
  return 1;
}
```

Try this out in the console and then put in the code from the **Code** section of the test. You'll see it returns `1`.

It seems so simple but that's the point. This is the first step of our journey. We've actually determined some important things for our function including its name, how many parameters it has (one) and what we should call that parameter (`word`), and what the function returns (a number). It's a start!

Our test should test the smallest possible thing we can test. Then we should write the minimum amount of code needed to pass the test.

Let's reiterate that, except in bold: **Test the smallest thing possible. Then write the minimum amount of code to make the test pass.** Yes, it's that important. We could even go so far as to say

a journey of a thousand miles begins with... baby steps. If we are baby stepping through our code, it will be *much* easier to solve hard problems. It won't quite be child's play but we'll have an essential tool at our disposal that can help us avoid getting overwhelmed and frustrated.

Our Second Test

The next step is... you guessed it. Another baby step. What's the next smallest thing we can test? Well, how about two words?

Let's write another pseudocode test and add it to the first.

```
Describe: wordCounter()

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

<!-- Our second test. -->
Test: "It should return 2 if a passage has two words."
Code:
const text = "hello there";
wordCounter(text);
Expected Output: 2
```

Let's take a look at what we've added. First of all, we don't need to use **Describe** again because we are still testing the same function. Remember, we can group all of our tests related to this function in one **Describe** block.

Other than that, our test looks very similar to the first test. We've added a word to the `text` variable and now the **Expected Output** is `2`.

Next, we need to update the function so that *both* tests pass. We can't just do something like this:

```
function wordCounter(text) {
  return 2;
}
```

That breaks the first test. So we need to actually update our method and write some new code. Because we are practicing loops, let's use `Array.prototype.forEach()` to solve the problem. It's not the easiest way to solve this problem but it's good practice and we can refactor it soon.

Before looking at the function below, we recommend trying to write the loop yourself with your pair. (You can do so in the DevTools console.)

Here are a few hints:

- You can use `String.prototype.split()` to split strings into arrays.
- You'll also need a variable to keep track of the `wordCount` since the loop will go through each element in the array, incrementing the `wordCount` each time.

Once you're ready, here is the function.

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

Let's walk through the function line by line to reiterate what we've learned about `Array.prototype.forEach()` and looping in general.

- Our loop is going to be counting all the words in a text passage, incrementing the total number of words by one each time through the loop. We need to store that information somewhere. That's why we have a `wordCount` variable starting at `0`. Otherwise, we won't be able to increment the word count.
- We also can't iterate through a string with `Array.prototype.forEach()`. We need to turn it into an array first. We can do that with `Array.prototype.split()`. For this method, we need to pass in a separator as an argument to define how our string should be split it up. We want the method to split up the string based on spaces between words so we use the separator `" "`. We save the split array in a variable called `textArray`. This will return `["hello", "there"]` for the string `"hello there"`.
- Note that `wordCount` uses `let` because it will change while `textArray` uses `const` because once a value is assigned to `textArray`, `textArray` will not change again. As always, think carefully about when you should use `let` and `const`. We *could* still change values in `textArray` even if it's a `const`, but `const` communicates to other developers that `textArray` won't change in this function.
- Now it's time to iterate through each element in our `textArray`. We use `Array.prototype.forEach()` to do so. The parameter of the callback function passed into `Array.prototype.forEach()` should be accurately named so we don't get confused. Since `textArray` is split into words, it makes sense to name the parameter `word`.
- Each time through the loop, we increment `wordCount` by one. We are using some new notation here. `++` indicates that the value should increase by increments of one. Saying

`wordCount++` is a shorter way of saying `wordCount = wordCount + 1`. You will see the `++` notation a lot and we recommend you use it, too.

- Finally, once we are done looping, we return the value of `wordCount`.

Try it out in the DevTools console. Both of our tests will "pass" and return the correct value.

What Should We Test For Next?

In fact, our function will now work for *any* word count except... well, there are quite a few exceptions. This is what testing is all about — identifying exceptions! For example, what about this string?

```
"There are 17 cats."
```

`17` isn't a word, it's a number.

What about an empty string? Try this out in the DevTools console and you'll see that the value of `wordCounter("");` is `1` even though the string we passed in has *no* words.

What about punctuation? What if a value that's not a string gets passed into the function? What about... well, you see what we mean. We need to write tests for as many of these use cases, making our function really robust. And with each change to the function, our function still needs to pass *all* of our tests. The testing framework that we'll use in Intermediate JavaScript makes checking that all tests are passing really easy, because the testing is automated. Since we are doing things manually, we don't have to manually check *every* test every time. That would be too tedious. But we do still need to check our latest test and it can be useful to spot check any other tests you might be concerned about.

Looking for Opportunities to Refactor

Also, each time we update our code, we should think about opportunities to refactor. For example, we can solve this problem without any loop with the `Array.length` property:

```
function wordCounter(text) {
  return text.split(" ").length;
}
```

While this simplifies our code — and all of our tests so far will still be passing — we'd still need to think about numbers, punctuations, empty strings, and so on. Note that when we build out this function further in the next lesson, we will go back to using a loop. We're going to do things the hard way to improve our looping skills.

Testing Your Code

When we get to Intermediate JavaScript, our tests will be automated and written in code, but for now, we'll test our functions in the DevTools console to verify that they return what we expect.

To test your function, paste the function you're testing into the console and then add the **Code** section of the test that you wrote. For instance, we might copy in `wordCounter("Hi there, friend!")` to make sure it equals 3 (an **Expected Output**).

Working in this manner might feel like slow going, but we should always be testing our code to make sure it is working correctly even if we have to do so manually. Manual testing will also allow us to solidify coding concepts and really take a look at what's going wrong (or right) with our code. Remember that manual testing includes writing the pseudocode test first, then writing/updating the function in your scripts, and then running your test in the DevTools console by calling the function to see whether the result matches what you expected.

If writing tests is still a bit murky, don't worry. We'll continue to use test-driven development in the next lesson as we start building out our application.

Previous (/introduction-to-programming/arrays-and-looping/building-a-text-analyzer)
Next (/introduction-to-programming/arrays-and-looping/text-analyzer-with-tdd-wordcounter)

Lesson 24 of 50
Last updated March 24, 2023

disable dark mode

(http://www.epicodus.com)