**Introduction**

Test-driven development (TDD) is a programming technique where you write test code before implementation code. Test code is written to define the desired behavior of your program. The test output provides descriptive error messages that inform the implementation of your program.

**The Red-Green-Refactor Cycle**

One of the driving forces of TDD is the *red-green-refactor* cycle. “Red” and “green” correspond to the color of the text that our test framework produces in the terminal while running the tests in our suite. Red signifies failing tests and green corresponds to passing tests.

Use [this red, green, refactor diagram](https://s3.amazonaws.com/codecademy-content/programs/tdd-js/articles/red-green-refactor-tdd.png) to help you as you read the steps below:

* Red — Write tests that describe the intended behavior of implementation code, and then compare developer expectations with the actual results of implementation code. The tests should always fail at first because the implementation code for the desired behavior will be written in response to the failing test.
* Green — Write just enough implementation code to make the test pass. The tests return *green* because the implementation code executes the intended behavior described by the test in the *red* phase.
* Refactor — Clean up and optimize code following the [characteristics of a good test](https://www.codecademy.com/articles/tdd-u2-good-test). Refactoring involves actively considering test and implementation code and making revisions to the code base. The tests are passing and should continue to pass throughout this phase of the cycle.

In this lesson, you will use a test-driven development approach to program a method that calculates the sum of an array of numbers.

**Instructions**

**1.**

Locate the **index\_test.js** file in your editor.

This code has an error in it! You can identify the error by running the test suite and reviewing the output.

Use npm test in the terminal to run the test suite.

Based on the feedback in your terminal, fix the code, then run your test again. Check your work once it passes.

const assert = require('assert');

const Calculate =  require('../index.js')

describe('Calculate', () =>{

  describe('.sum', () =>{

     it('returns the sum of an array of numbers', () =>{

     })

  })

})

# Getting Into The Red I

When we use the red, green, refactor cycle, we:

1. Write test code that fails
2. Write implementation code to make the test pass
3. Consider refactoring the code

As you learn TDD, it may feel backward to intentionally write and run code that will return errors. In TDD we react to the error messages by continually re-writing our implementation code so it behaves in the way that our test specifies.

Imagine you were trying to create a method named .initials, inside of an object named Phrase.

The desired behavior of .initials() is that it should return the first letter of each word in a phrase that is passed to it as an argument.

#### Step 1: Write The Test

The first step to writing a test with Mocha is to use describe and it blocks to describe the desired behavior of your code. It’s very important for tests to thoroughly describe the desired behavior with natural language. This will create the most helpful error messages and make it easy for you to understand the behavior that your test failed in executing.

Before running our test, we would use an assert statement to compare the desired result of our method with the actual result.

The first step in developing this method would be writing a test that could look like this:

describe('Phrase', () => {

describe('.initials', () => {

it('returns the first letter of each word in a phrase.', () => {

assert.equal(Phrase.initials('Nelson Mandela'), 'NM');

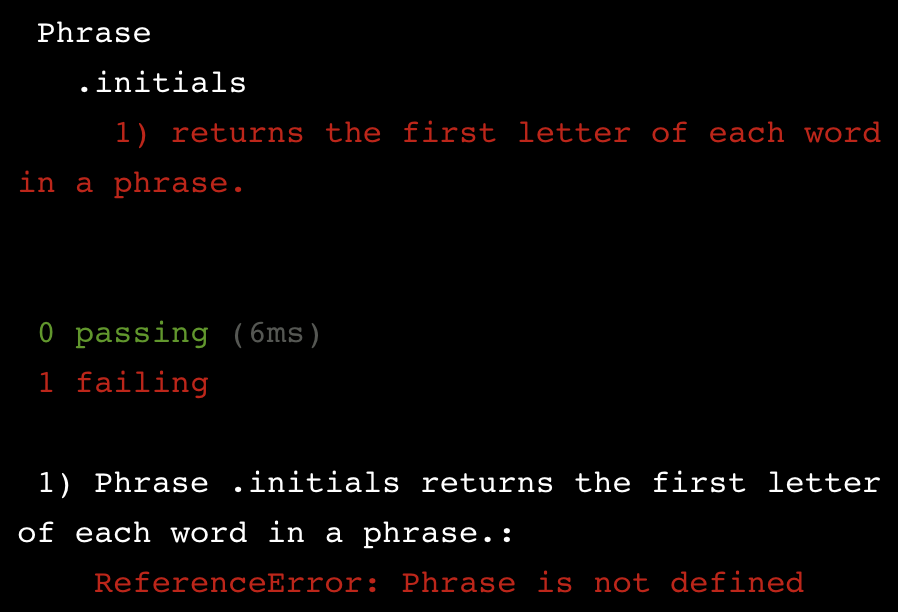
})

})

})

#### Step 2: Run the test

If we ran this test we would get the following error message in the terminal:



#### Step 3: The test fails (yea!)

The error message tells us that the error is related to the Phrase.initials code block. The ReferenceError tells us that the error is thrown because we don’t have a Phrase object. In the next exercise, we’ll show you the minimum possible code required to get this test to pass.

**Instructions**

**1.**

Complete the test for Calculate.sum() In **index\_test.js** by writing an assert.equal()statement to verify that Calculate.sum([1,2,3]) is equal to 6.

Hint

assert statements should follow this syntax:

assert.equal(actual, expected);

* The first argument in your assert.equal() statement should be Calculate.sum([1,2,3])
* The second argument should be 6

const assert = require('assert');

const Calculate =  require('../index.js')

describe('Calculate', () => {

  describe('.sum',() => {

    it('returns the sum of an array of numbers', () => {

      assert.equal(Calculate.sum([1,2,3]), 6);

    });

  });

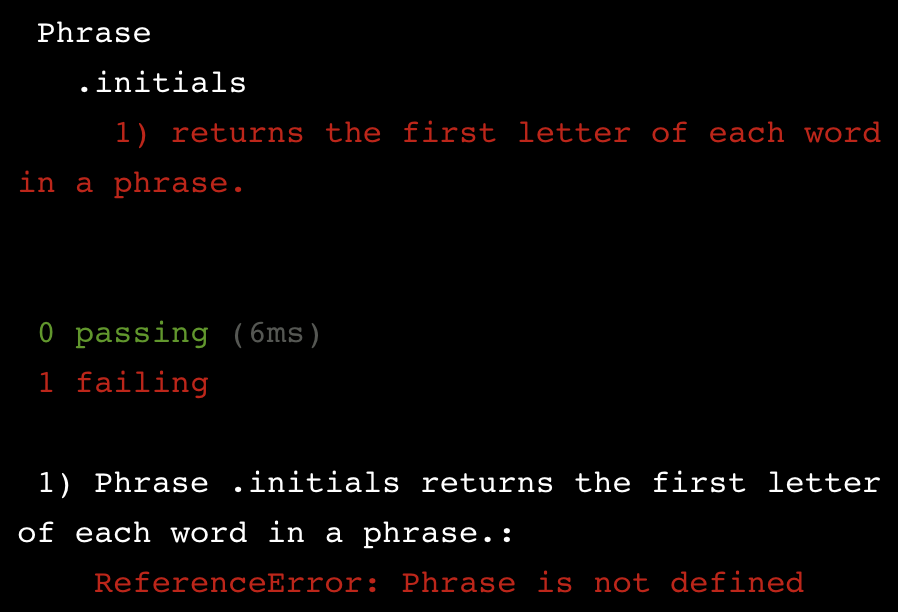
});

# Red To Green I

#### Congrats you’re in the red!

The red error messages describe the failures of our implementation code, so we can specifically address each issue that is preventing our test from passing.

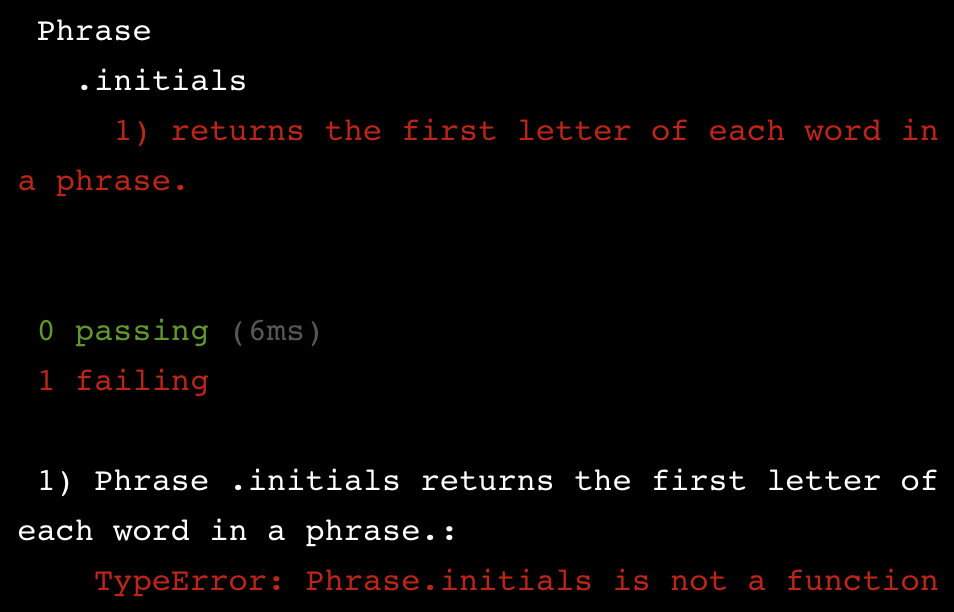
Let’s look at the error message from our Phrase.initials() example in the last exercise:



The error tells us that Phrase is not defined. This is because we have not created an object named Phrase yet. Let’s do that now:

const Phrase = {}

When we run our test again the error would look like this:



We’re still in the red, but we have a new error. The error says that Phrase.initials is not a function. That’s because we haven’t added an .initials method to the Phrase object. Let’s do that now:

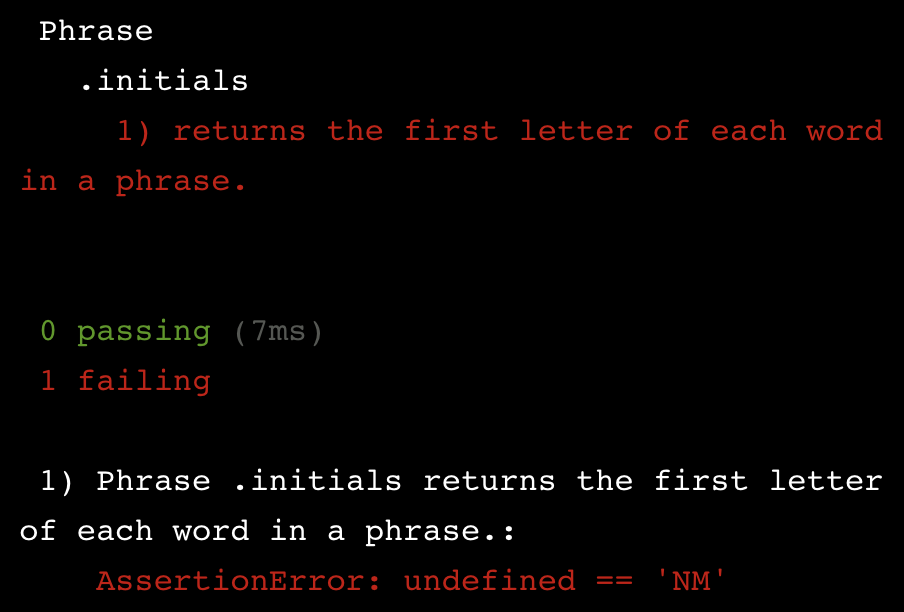
const Phrase = {

initials() {

}

}

Now when we run our test we get the following error:



This is the first error that is referencing the actual behavior of the .initials method.

The assert statement we wrote looked like this:

assert.equal(Phrase.initials('Nelson Mandela'), 'NM');

We expected the result of passing Nelson Mandela to Phrase.initials() to be NM, but the actual result was undefined. This is because our method has not been programmed to return anything.

Following TDD, the next step would be writing the minimum possible implementation code to make our test pass. In this example, that would involve adding a line of implementation code so that .initials() returns our expected result, which is NM.

The minimum possible implementation code to make the test pass:

const Phrase = {

initials(phr) {

return 'NM';

}

}

Now our test would pass and we would be in the green. But that doesn’t mean our method is working as intended. If we entered “Jody Williams” our method would return ‘NM’, not ‘JW’. We will address this issue when it is time, but following TDD, now that we have progressed from the red to the green, we will move into the refactor phase.

**Instructions**

**1.**

Go to your **index.js** file and above the line of code that is exporting your implementation code, create an empty object named Calculate.

Then run the test suite in your terminal using npm test. Notice the error — you get that .sum() is not a function as an out of the box error message.

Check your work.

Checkpoint 2 Passed

Hint

Make sure that

* you declare Calculate as a const
* the last line of code in **index.js** is:

module.exports = Calculate;

**2.**

Inside your Calculate object, add a .sum() method that takes one argument.

Run the test suite again in the terminal. You should now get an AssertionError.

Check your work.

Checkpoint 3 Passed

**3.**

Address the AssertionError in the terminal by making .sum() return the same value as the expected result you defined in your test.

Run the test suite. Your test should pass!

Check your work.

Checkpoint 4 Passed

Hint

You can see your expected result by reviewing the test you wrote in **index\_test.js**

const Calculate = {

  sum(arr){

    return 6

  }

}

module.exports = Calculate;

#### Congrats you’re in the green!

Once all your tests pass, you can confidently refactor your code — restructure and improve it without changing its external behavior. The confidence comes from knowing that our tests will catch us if we make a misstep.

When refactoring, it’s critical to test early and often — if our tests turn red, then we know that something went wrong while we were refactoring, and we can undo those changes.

A good place to start with refactoring is to restructure tests to reflect the four phases of a good test: setup, exercise, verification, teardown.

Let’s consider the test for our Phrase.initials method. We could rewrite the test with setup, exercise, and verification stages to make it more expressive and maintainable.

describe('Phrase', () => {

describe('.initials', () => {

it('returns the first letter of each word in a phrase.', () => {

// Setup

const inputName = 'Nelson Mandela';

const expectedInitials = 'NM';

// Exercise

const result = Phrase.initials(inputName);

// Verification

assert.equal(result, expectedInitials);

});

});

});

Here, we have re-written the same test, but we have created variables to hold the values that we will use to test the behavior of .initials(), and broken up the test into distinct phases, which are marked by the comments in the code.

Now that we have refactored our test code, we can refactor our implementation code. Our earlier code looked like this:

const Phrase = {

initials(phr) {

return 'NM';

}

}

In order to make our implementation code more expressive, we can change the name of the input argument of .initials() like this:

const Phrase = {

initials(inputName) {

return 'NM';

}

}

Refactoring will look different for each project and each iteration of the red-green-refactor cycle. In some cases you won’t need to refactor at all! Regardless, you should consider how you can improve your code and — although it is optional — it can set you up for success moving forward.

**Instructions**

**1.**

In **index\_test.js**, refactor the code in your Calculate.sum test to include setup, exercise, and verify phases.

* Create descriptive const variables for your expected result, and then your input array
* Create a descriptive const variable for the result of your method
* Rewrite your assert.equal statement to compare the exercised result to the expected result.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Optional: Go to **index.js**. Refactor your implementation code and set the .sum() argument name to inputArray. This descriptive name makes your implementation code more expressive.

Required: Run the test suite to make sure you are still in the green.

Checkpoint 3 Passed

Hint

Run npm test in the terminal

const assert = require('assert');

const Calculate =  require('../index.js')

describe('Calculate', () => {

  describe('.sum',() => {

    it('returns the sum of an array of numbers', () => {

      const expectedResult = 6;

      const inputArray = [1,2,3]

      const result = Calculate.sum(inputArray)

      assert.equal(result, expectedResult);

    });

  });

});

# Getting into the Red II

Congratulations, you have made it through your first red-green-refactor cycle using TDD! The next step is to repeat this cycle to build a more complete implementation of the .initials method.

Once you have a baseline test for your feature, you can start to write additional test cases that force you to write better implementation code.

Let’s consider the test suite for our Phrase.initials method. We have one test that checks if Phrase.initial("Nelson Mandela") returns "NM".

describe('Phrase', () => {

describe('.initials', () => {

it('returns the first letter of each word in a phrase.', () => {

const inputName = 'Nelson Mandela';

const expectedInitials = 'NM';

const result = Phrase.initials(inputName);

assert.equal(result, expectedInitials);

});

});

});

Let’s write another test that pushes us to implement a Phrase.initials() method that returns the first letter of each word for “Nelson Mandela” and a different name.

To do this, we will add another it block to our code, and inside the callback function we will again follow the setup, exercise, verification phases for writing tests. This time we will write a test based on the circumstance that the string passed to .initials() has three names: 'Juan Manuel Santos'.

Here is what a second test for .initials() would look like:

describe('Phrase', () => {

describe('.initials', () => {

. . .

it('returns the initials of a name', () => {

const nameInput = 'Juan Manuel Santos';

const expectedInitials = 'JMS';

const result = Phrase.initials(nameInput);

assert.equal(result, expectedInitials);

});

});

});

Although the new it block is similar to the existing example, the two together force you to implement a method that will return initials that pass both tests.

The idea of TDD is to continue building tests and going through the red-green-refactor process until you feel confident that your implementation code performs as expected.

**Instructions**

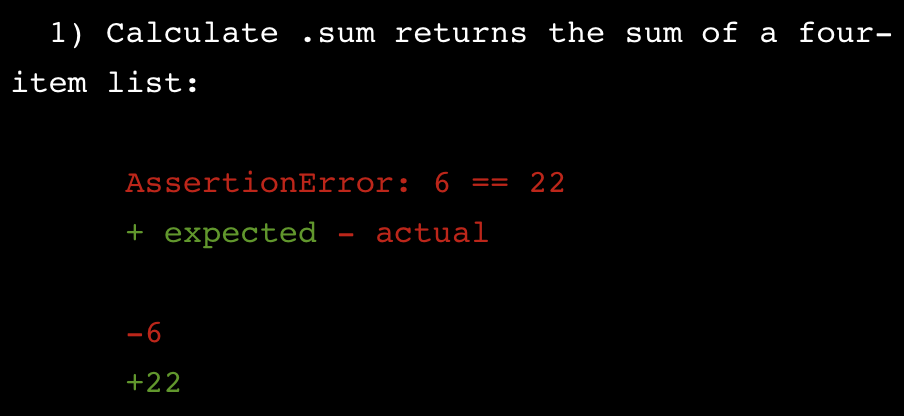
**1.**

Add another test to your describe block that checks if the Calculate.sum([4, 5, 6, 7]) returns 22.

Create a new it block with the string: 'returns the sum of a four-item list'

Inside the it block, add setup, exercise, and verify phases.

You should see the following error message when you run your test suite.



Hint

Use the following in your setup, exercise, and verify phases:

* **Setup** — set a const variable called expectedResult to 22. Set a const variable named inputArray to [4, 5, 6, 7].
* **Exercise** — set a const variable named result to the result of passing inputArray to Calculate.sum().
* **Verify** — assert that the two arguments result and expectedResult are equal.

const assert = require('assert');

const Calculate =  require('../index.js')

describe('Calculate', () => {

  describe('.sum',() => {

    it('returns the sum of an array of numbers', () => {

      const expectedResult = 6;

      const inputArray = [1,2,3]

      const result = Calculate.sum(inputArray)

      assert.equal(result, expectedResult);

    });

    it('returns the sum of a four item list', ()=>{

      const expectedResult = 22;

      const inputArray = [4,6,5,7];

      const result = Calculate.sum(inputArray);

      assert.equal(result, expectedResult)

    });

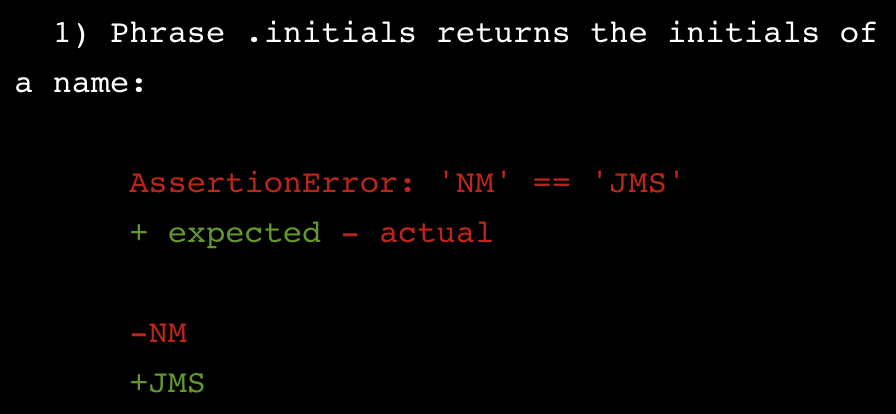
  });

});

**Red to Green II**

Now that we have tests that cover more than one condition, it is time to program the actual behavior that we want our .initials method to execute.

Let’s think about the current state of Phrase.initials(). We have one passing test and one failing test that raises the following error when you run the test suite:



This is not surprising, considering our current implementation always returns ‘NM.’

To write a more complete implementation, that could get us back into the green, we could write something like this:

const Phrase = {

initials(inputName) {

// Create an empty array for initials

const initials = [];

// Create an array of strings

const words = inputName.split(" ");

// Iterate through the array of strings and push the first character of each to array

words.forEach((word) => {

initials.push(word.charAt(0));

});

// Return the initials as one string

return initials.join("");

}

}

This implementation will successfully pass the tests we have written. Each additional test pushes us to build a complete implementation of the .initials method.

**Instructions**

**1.**

Now, in your **index.js** file, you are going to replace the line of code that returns 6, with the actual code to calculate the sum of the input array, so that your implementation code will pass both of the tests you have written.

Run your test suite to get the error message that will lead you towards writing the most concise production code to make your test pass.

Run your test suite again to make sure you are in the green.

Check your work.

Hint

Try the following:

* Create a let variable named totalSum and set it equal to 0.
* Below it, create a const variable named inputArrayLength and set it equal to inputArray.length.
* Write a for loop that iterates inputArrayLength number of times and adds the value of each element to totalSum.
* Return totalSum.

 const Calculate = {

  sum(inputArray) {

    let totalSum = 0;

    const inputArrayLength = inputArray.length;

    for (let i = 0; i < inputArrayLength; i++){

      totalSum += inputArray[i]

    }

    return totalSum

  }

}

module.exports = Calculate;

# Refactor II

Congrats, you’re back in the green!

Which means the next step is to refactor the code so that your tests are fast, complete, reliable, isolated, maintainable, and expressive. Also, you want to make sure your implementation code is descriptive and efficient.

Sometimes refactoring can take place in test and implementation code, either one, or neither. The purpose of the refactor phase is to think critically about the code you have and decide whether there is anything unnecessary, redundant, or that could be done more clearly or efficiently.

**Instructions**

**1.**

Refactor the code in your **index.js** file by replacing the current implementation code inside Calculate.sum() with a function that uses the built-in JavaScript method .reduce() to accumulate the total value of an array of numbers.

You can read more about .reduce() in the [Mozilla Developer Network documentation](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce?v=a).

Run your test to make sure you are still in the green.

Hint

You can use the .reduce method to accumulate each element in an array— reduced to one value.

return myArray.reduce((accumulator, currentValue) => {

return accumulator + currentValue;

});

const Calculate = {

  sum(inputArray) {

    return inputArray.reduce((sum, value) => {

      return sum + value;

    })

  }

}

module.exports = Calculate;

**Edge Case**

Now that we have written and refactored a few tests, it’s time to consider *edge cases*.

An edge case is a problem or situation that occurs only at an extreme (maximum or minimum) operating parameter — you can think of these as special cases that you need to account for.

Take our Phrase.initials() example from earlier. What happens if we pass .initials() a number instead of a string? We could write a test and then implementation code that responds to this behavior by raising an error message that instructs you to only use strings.

The test for throwing an error when a number is passed to Phrase.initials() would look like this:

it('raises an error if the parameter passed in is not a string', () => {

// Setup

const nameInput = 14;

// Exercise

const exercise = () => { Phrase.initials(nameInput) };

// Verification

assert.throws(exercise, /only use string/);

})

After getting our intended error message we would write just enough implementation code to pass the test. Which in this case would be adding the following to Phrase.initials():

if (typeof inputName !== "string") {

throw new Error("ERROR: only use string");

}

The complete implementation code would now be back in the green and look like this:

const Phrase = {

initials(inputName) {

if (typeof inputName !== "string") {

throw new Error("ERROR: only use string");

}

const initials = [];

const words = inputName.split(" ");

words.forEach((word) => {

initials.push(word.charAt(0));

});

return initials.join("");

}

}

This is an example of creating a test for edge cases.

**Instructions**

**1.**

Now you will write your own test and implementation code for an edge case in your Calculate module, which covers the behavior of an empty array passed to .sum().

Add another it block to your describe block that has the docstring:

"returns zero for an empty array"

Inside the new it block, use the following steps to write the setup, exercise, and verify phases:

* Set a const variable named expectedResult equal to 0.
* Set a const variable named inputArray equal to an empty array.
* Set a const variable named result equal to the result of Calculate.sum(inputArray)
* Assert that your result is equal to your expectedResult.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

In your **index.js** file, follow the steps below to add an if block that returns0 if your inputArray is empty.

* Add an if block above the .reduce() method.
* Check if the length of your inputArray is equal to 0.
* Inside the if block, return 0 .

Checkpoint 3 Passed

Hint

Use the following code above the .reduce() method to return 0 when the inputArray.length is equal to zero.

if (inputArray.length === 0) {

return 0;

}

const assert = require('assert');

const Calculate =  require('../index.js')

describe('Calculate', () => {

  describe('.sum',() => {

    it('returns the sum of an array of numbers', () => {

      const expectedResult = 6;

      const inputArray = [1,2,3]

      const result = Calculate.sum(inputArray)

      assert.equal(result, expectedResult);

    });

    it('returns the sum of a four item list', ()=>{

      const expectedResult = 22;

      const inputArray = [4,6,5,7];

      const result = Calculate.sum(inputArray);

      assert.equal(result, expectedResult)

    });

    it('returns zero for an empty array', ()=>{

      const expectedResult = 0;

      const inputArray = [];

      const result = Calculate.sum(inputArray);

      assert.equal(result, expectedResult)

    });

  });

});

**Review**

We have just covered Test Driven Development with Mocha.

**At a high-level the process is:**

* Write The Test — Start with a test describing the functionality we’d like to see.
* Fail The Test — Write code in response to the test that fails.
* Pass The Test — The tests fail and communicate feedback to developers through error messages. It’s our responsibility to read those messages, then respond by writing the minimum amount of code to address those messages.
* Refactor Your Code — See below.

**The development process is guided by the red-green-refactor cycle.**

**Red**

Write a test that covers the functionality you would like to see implemented. You don’t have to know what your code looks like at this point, you just have to know what it will do.

Run the test. You should see it fail. Most test runners will output red for failure and green for success. While we say “failure” do not take this negatively. It’s a good sign! By seeing the test fail first, we know that once we make it pass, our code works.

**Green**

Read the error message from the failing test, and write as little code as possible to fix the current error message. By only writing enough code to see our test pass, we tend to write less code as a whole. Continue this process until the test passes.

This may involve writing intermediary features covering lower level functionality which require their own Red, Green, Refactor cycle. The **edge-case** was an example of this.

Do not focus on code quality at this point. Be shameless! We simply want to get our new test passing.

**Refactor**

Clean up your code, reducing any duplication you may have introduced. This includes your code as well as your tests.

Treat your test suite with as much respect as you would your live code, as it can quickly become difficult to maintain if not handled with care. You should feel confident enough in the tests you’ve written that you can make your changes without breaking anything.

**Instructions**

Take a look at our final workspace here. We have both implementation code that we are confident in, and a test suite that can catch regressions.

Here we can see the benefits of using Mocha with Test-Driven Development.